Advances in Database Technology — EDBT 2019

22nd International Conference
on Extending Database Technology
Lisbon, Portugal, March 26–29, 2019
Proceedings

Editors

Melanie Herschel
Helena Galhardas
Berthold Reinwald
Irini Fundulaki
Carsten Binnig
Zoi Kaoudi

open
proceedings

Advances in Database Technology — EDBT 2019
Proceedings of the 22nd International Conference
on Extending Database Technology
Lisbon, Portugal, March 26–29, 2019

Series ISSN: 2367-2005

Editors

Melanie Herschel, University of Stuttgart, Germany
Helena Galhardas, INESC-ID and IST, Universidade de Lisboa, Portugal
Berthold Reinwald, IBM Research, USA
Irini Fundulaki, ICS Forth, Greece
Carsten Binnig, TU Darmstadt, Germany
Zoi Kaoudi, QCRI, HBKU, Qatar

OpenProceedings.org
University of Konstanz
University Library
78457 Konstanz, Germany

OpenProceedings ISBN: 978-3-89318-081-3

DOI of this front matter: `10.5441/002/edbt.2019.01`

In memoriam

Christine Collet
1956–2019

Foreword

The International Conference on Extending Database Technology (EDBT) is an established and renowned forum for the exchange of the latest research results and advances in data management. This year, the 22nd edition of EDBT takes place in Lisbon, Portugal, from March 26 to March 29, 2019. It is jointly organized with the International Conference on Database Theory (ICDT). In a world where increasingly many aspects of our lives and society are data-driven, data management technology continues to broaden its reach and extends its tradition of contributing models, algorithms, and architectures to novel applications adapted to new hardware and software.

As in previous years, EDBT 2019 solicited contributions both on novel research results and on experience and analysis results that focus on a comprehensive and detailed performance evaluation. For the first time, EDBT 2019 further solicited papers that describe innovative systems as part of its main research track. We also continued the recently established short paper track, offering a forum to present research in progress and visionary ideas during plenary poster sessions of the conference. To complement the scientific program, EDBT further solicited demonstrations of research prototypes, descriptions of industrial and application achievements, and proposals for tutorials.

The EDBT 2019 program committee reviewed 157 full research papers, of which 36 were accepted. For short papers, 28 papers out of 122 were selected. Among the 24 submissions to the industry and application track, 8 papers were accepted. The 21 demonstrations presented at the conference were selected among 42 demonstration proposals. Finally, we accepted 3 out of 10 tutorials. All these contributions will be presented at the conference. The program additionally features five workshops, an EDBT/ICDT joint session on research challenges, and four invited EDBT/ICDT joint keynotes.

Shaping the exciting program of EDBT 2019 is the result of a large community effort, and I take this opportunity to thank all persons involved. First, I would like to thank all authors for their high-quality submissions and contributions. I also would like to thank all reviewers who served on the EDBT 2019 program committee and the chairs responsible for our different tracks, namely Berthold Reinwald (IBM, United States) who chaired the industrial and application track, Carsten Binnig (TU Darmstadt, Germany) who served as demonstration chair, our tutorial chair Irini Fundulaki (ICS FORTH, Greece), and Paolo Papotti (EUROCOM, France) who served as workshop chair. The special session on joint EDBT/ICDT research challenges was organized by Julia Stoyanovich (NYU, USA). I also thank Laura Haas (UMass Amherst, USA) and Alon Halevy, who generously accepted to serve on the Test of Time Award Committee. Many thanks also to Paolo Atzeni (Universita' Roma Tre, Italy), Wei Wang (UNSW Sydney, Australia), and Jeffrey Xu Yu (Chinese University of Hong Kong) for serving on the Best Paper Award committee.

The conference would not have been possible without the tireless effort of the general chair Helena Galhardas (INESC-ID and IST, Universidade de Lisboa, Portugal) and the local organization team. Special thanks to the finance chair Manuel J. Fonseca (Universidade de Lisboa, Portugal), the local executive chairs José Borbinha and Luís Rodrigues, the sponsorship chairs João Garcia and Miguel Pardal, the publicity chair Paolo Romano, the student volunteers chair Hugo Nicolau (all from INESC-ID and IST, Universidade de Lisboa, Portugal), and the website chair António Higgs (INESC-ID, Portugal). These proceedings have been produced thanks to our proceedings chair Zoi Kaoudi (QCRI, Qatar). Norman Paton was most helpful in advising and coordinating with the EDBT Executive Board.

I really look forward to an interesting program and exciting conference on March 26–29, 2019 and to meeting you in Lisbon.

Melanie Herschel
EDBT 2019 Program Chair

Program Committee Members

Research Program Committee

Adriane Chapman, U Southampton
Ahmed Elmagarmid, QCRI HBKU
Alekh Jindal, Microsoft
Alessandro Campi, Politecnico di Milano
Alexander Böhm, SAP SE
Alfons Kemper, TU Munich
Alfredo Cuzzocrea, ICAR-CNR and U Calabria
Amelie Marian, Rutgers U
Amr El Abbadi, UC Santa Barbara
Angela Bonifati, U Lyon 1
Anton Dignös, Free U Bozen-Bolzano
Antonios Deligiannakis, TU Crete
Arash Termehchy, Oregon State U
Arnab Bhattacharya, IIT Kanpur
Avigdor Gal, Technion
Bernhard Seeger, U Marburg
Boris Glavic, Illinois IT
Chuan Lei, IBM Research-Almaden
Daisy Zhe Wang, U Florida
Daniele Dell'Aglio, U Zurich
Dario Colazzo, LAMSADE - Universit
Davide Martinenghi, Politecnico di Milano
Demetrios Zeinalipour-Yazti, U Cyprus
Elena Ferrari ,U Insubria
Elisa Quintarelli, Politecnico di Milano
Eric Lo, Chinese U Hong Kong
Ernest Teniente, U Politècnica de Catalunya
Esther Pacitti, LIRMM and INRIA U Montpellier 2
Farouk Toumani Limos, B. Pascal U Clermont-Ferrand
Felix Naumann, HPI
Francois Goasdoue, U Rennes 1
George H. L. Fletcher, Eindhoven UT
Georgia Koutrika, Athena Research Center
Goetz Graefe, Google
Gottfried Vossen, ERCIS Muenster
Guoliang Li, Tsinghua U
Haruo Yokota, Tokyo Institute of Technology
Heiko Schuldt, U Basel
Isabel Cruz, U Illinois Chicago
Jean-Marc Petit, U Lyon & INSA Lyon
Jian Pei, Simon Fraser U
Jianwen Su, UC Santa Barbara
Johann Gamper, Free U Bozen-Bolzano
Kai-Uwe Sattler, TU Ilmenau
Karl Aberer, EPF Lausanne
Katja Hose, Aalborg U
Klemens Böhm, Karlsruhe I.T.
Kostas Magoutis, U Ioannina
Kostas Stefanidis, U Tampere
Marta Patino, U Politécnica de Madrid
Martin Theobald, U Luxembourg
Matthias Boehm, IBM
Matthias Renz, George Mason U
Maya Ramanath, IIT Delhi
Michael Grossniklaus, U Konstanz
Minos Garofalakis, TU Crete
Mohammad Sadoghi UC Davis
Nicoleta Preda, U Versailles
Nikos Mamoulis, U Ioannina
Nikos Pelekis, U Piraeus
Panagiotis Karras, Aarhus U
Paolo Missier, Newcastle U
Periklis Andritsos, U Toronto
Peter Triantafillou, U Warwick
Riccardo Torlone, U Roma Tre
Sabrina De Capitani di Vimercati, U Milan
Sherif Sakr, U New South Wales
Stefan Deßloch, TU Kaiserslautern
Tore Risch, Uppsala U
Vassilis Christophides, INRIA
Verena Kantere, U Geneva
Will Wu, Google
Wolfgang Lehner, TU Dresden
Xiaochun Yang, Northeastern U
Xiaofang Zhou, U Queensland
Xin Huang, Hong Kong Baptist U
Yang Cao, U Edinburgh
Yannis Theodoridis, U Piraeus
Yannis Velegrakis, U Trento
Yaron Kanza, AT&T Labs - Research
Yi-Leh Wu, Nat. Taiwan U Sc. & Tech.
Zhifeng Bao, RMIT U
Ziawasch Abedjan, TU Berlin

Industrial/Applications Program Committee

Alkis Simitsis, HP Labs
Amol Ghoting, LinkedIn
Andrey Balmin, Workday
Anil Goel, SAP Canada
Benjamin Schlegel, Oracle Labs
Calisto Zuzarte, IBM Toronto
Cliff Leung, Huawei
David Simmen, Apple
Martin Grund, Amazon AWS
Michael Rys, Microsoft
Nesreen Ahmed, Intel
Niketan Pansare, IBM
Norbert Ritter, U Hamburg
Paul Brown, Teradata Corp
Songting Chen, Tigergraph
Yannis Sismanis, Google

Demonstration Program Committee

Alexander Böhm, SAP SE
Arvind Arasu, Microsoft
Dirk Habich, TU Dresden
Jana Gicheva, Imperial College London
Manos Athanassoulis, Boston U
Martin Theobald, U Luxembourg
Michael Gertz, Heidelberg U
Rick Cole, Tableau Software
Sanjay Krishnan, U Chicago
Stefan Manegold, CWI
Sven Helmer, Free U Bozen-Bolzano
Thibault Sellam, Columbia U
Tilmann Rabl, TU Berlin
Uwe Roehm, U Sydney
Viktor Leis, TU Munich
Ziawasch Abedjan, TU Berlin

Test-of-Time Award

Established in 2014, the Test-of-Time Award awarded by the Extended Database Technology (EDBT) Conference recognizes papers presented at EDBT Conferences that have had the most impact in terms of research, methodology, conceptual contribution, or transfer to practice over the past ten years.

The 2019 Test-of-Time Award committee was formed by Laura Haas (University of Massachusetts, USA), Alon Halevy, and Melanie Herschel, the EDBT 2019 PC chair. The committee was charged with selecting a paper from the EDBT 2009 Proceedings.

After careful consideration, the Test-of-Time Award committee decided for the following paper from the 2009 EDBT Conference held in Saint Petersburg, Russia to receive the award:

Shore-MT: a scalable storage manager for the multicore era
by Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi
published in the EDBT 2009 Proceedings, pp. 24–35, DOI: 10.1145/1516360.1516365.

The committee members agreed that this paper clearly stands out in terms of methodology, impact, and influence. It has catalyzed and enabled substantial follow-up research and has demonstrated its high relevance to industry.

Abstract:

Historically, database engines focused on the ability to efficiently overlap many requests over a small number processor cores, with I/O latencies and scalability as the main design driver. However, the advent of increasingly multicore hardware circa 2000 brought new challenges because concurrent transactions begin to stress the limits of the storage manager's thread scalability by accessing its internal structures simultaneously and in large numbers. This EDBT 2009 paper shows the results of experiments running benchmarks on four (then and still) popular open-source storage managers (Shore, BerkeleyDB, MySQL, and PostgreSQL) on a multicore machine. The results show that all systems suffer from scalability bottlenecks at the storage engine level. From that research emerged Shore-MT, an open-source multithreaded and highly scalable storage manager, built with Shore as a base. We learned that designers should favor scalability over single-thread performance, and we identified several other key principles for architecting scalable storage engines.

Ten years later, Shore-MT work has concluded, although the system still serves as a research platform in the space. Meanwhile, research on transaction processing scalability continues to mature, the move to main-memory transaction processing and their higher TPS increased the need for scalable storage managers, while the popular open-source systems, such as MySQL and PostgreSQL, significantly improved their scalability. In particular, a significant amount of research and industrial developments in the ten years since the Shore-MT paper focused on improving the scalability of individual components of a storage manager, such as latches, the logging subsystem and access methods. This research was partly carried out by our research group as follow-on work, but other research groups and database vendors have made important contributions as well. Another significant amount of effort has focused on scalable concurrency control protocols, again both within and outside our research group. The knowledge that we have gained from building Shore-MT has been invaluable in maintaining scalability in this new, multi-dimensional ecosystem.

The EDBT Test-of-Time award for 2019 will be presented during the EDBT/ICDT 2019 Conference as part of the Awards session on Wednesday, March 27, 2019, by Anastasia Ailamaki (EPFL, Switzerland).

Table of Contents

Research Papers

Tutorials

Industry and Applications Papers

Demonstrations

Short Papers

Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP

Chen Luo*
University of California, Irvine
cluo8@uci.edu

Pınar Tözün†
IT University of Copenhagen
pito@itu.dk

Yuanyuan Tian
IBM Research - Almaden
ytian@us.ibm.com

Ronald Barber
IBM Research - Almaden
rjbarber@us.ibm.com

Vijayshankar Raman
IBM Research - Almaden
ravijay@us.ibm.com

Richard Sidle
IBM
ricsidle@ca.ibm.com

ABSTRACT

The rising demands of real-time analytics have emphasized the need for Hybrid Transactional and Analytical Processing (HTAP) systems, which can handle both fast transactions and analytics concurrently. Wildfire is such a large-scale HTAP system prototyped at IBM Research - Almaden, with many techniques developed in this project incorporated into the IBM's HTAP product offering. To support both workloads efficiently, Wildfire organizes data differently across multiple *zones*, with more recent data in a more transaction-friendly zone and older data in a more analytics-friendly zone. Data *evolve* from one zone to another, as they age. In fact, many other HTAP systems have also employed the multi-zone design, including SAP HANA, MemSQL, and SnappyData. Providing a *unified* index on the large volumes of data across multiple zones is crucial to enable fast point queries and range queries, for both transaction processing and real-time analytics. However, due to the scale and evolving nature of the data, this is a highly challenging task. In this paper, we present Umzi, the *multi-version* and *multi-zone* LSM-like indexing method in the Wildfire HTAP system. To the best of our knowledge, Umzi is the first indexing method to support evolving data across multiple zones in an HTAP system, providing a consistent and unified indexing view on the data, despite the constantly on-going changes underneath. Umzi employs a flexible index structure that combines hash and sort techniques together to support both equality and range queries. Moreover, it fully exploits the storage hierarchy in a distributed cluster environment (memory, SSD, and distributed shared storage) for index efficiency. Finally, all index maintenance operations in Umzi are designed to be non-blocking and lock-free for queries to achieve maximum concurrency, while only minimum locking overhead is incurred for concurrent index modifications.

1 INTRODUCTION

The popularity of real-time analytics, e.g., risk analysis, online recommendations, and fraud detection etc., demands data management systems to handle both fast concurrent transactions (OLTP) and large-scale analytical queries (OLAP) over fresh data. These applications ingest data at high-speed, persist them into disks or shared storage, and run analytical queries simultaneously over newly ingested data to derive insights promptly.

The necessity of real-time analytics prompts the emergence of Hybrid Transactional and Analytical Processing (HTAP) systems, e.g., MemSQL [7], SnappyData [28], SAP HANA [21], and among others. HTAP systems support both OLTP and OLAP queries in a single system, thus allowing real-time analytics over freshly ingested data. Wildfire [15] is a large-scale HTAP system, prototyped at IBM research - Almaden. Many of the techniques developed in this research project have been incorporated into the IBM Db2 Event Store offering [4]. Wildfire leverages the Spark ecosystem [10] to enable large-scale data processing with different types of complex analytical requests (SQL, machine learning, graph analysis, etc), and compensates Spark with an underlying engine that supports fast transactions with snapshot isolation and accelerated analytics queries. Furthermore, it stores data in open format (Parquet [8]) on shared storage, so that other big data systems can access consistent snapshots of data in Wildfire. The back-end shared storage that Wildfire supports includes distributed file systems like Hadoop Distributed File System (HDFS) and GlusterFS [2], as well as object-based storage on cloud like Amazon S3 and IBM Cloud Object Storage.

To support efficient point lookups and range queries for high-speed transactional processing and real-time analytics, fine-grained indexing is mandatory in a large-scale HTAP system like Wildfire. However, indexing large volumes of data in an HTAP system is highly non-trivial due to the following challenges.

Challenges due to shared storage. First of all, for large-scale HTAP, memory-only solutions are not enough. As a result, most HTAP systems, including Wildfire, persist data in highly-available fault-tolerant shared storage, like HDFS and Amazon S3, etc. However, most of these shared storage options are not good at random access and in-place update. For example, HDFS only supports append-only operations and optimizes for block-level transfers, and object storage on cloud allows neither random access inside an object nor update to an object. To accommodate the unique characteristics of shared storage, index operations, e.g., insert, update and delete, have to leverage sequential I/Os without in-place updates. Naturally, LSM-like index structures are more appealing.

Furthermore, a typical shared storage prefer a small number of large files to a large number of small files. This is not only because of the overhead in metadata management, e.g., the maximum number of files supported by an HDFS cluster is determined by how much memory is available in the namenode, but more importantly because of the reduced seek time overhead when accessing larger files.

Finally, accessing remote shared storage through networks for index lookups is costly. Thus, indexing methods on HTAP must fully exploit the storage hierarchy in a distributed cluster environment for efficiency. Particularly, nowadays, we can take advantage of large memories and SSDs in modern hardware. Due to the large scale of data in HTAP systems, however, only the most frequently accessed portions of indexes can be cached locally,

*†Work done while at IBM Research - Almaden.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.02

while leaving cold entries in shared storage. Effective caching mechanisms must be developed to facilitate index lookup.

Challenge due to evolving nature of data. Since HTAP systems have to support both transactional and analytical workloads efficiently, many of them [7, 14, 21, 23, 28] store data in different organizations, typically one organization good for transactions on the more recent data and one organization good for analytics on the older data. We call the different data organizations *zones*. As data age in the system, they evolve from the transaction-friendly zone to the analytics-friendly zone. In Wildfire, transactions first append writes into a transaction log, which is then groomed into columnar data blocks. The groomed data is further periodically post-groomed to a more analytics-friendly organization that is optimal for queries by creating data versions, data partitioning, and larger data blocks. SAP HANA organizes data into a read-optimized main store and a write-optimized delta store. Writes are first buffered into the row-major delta store, which is further transformed into the columnar main store to facilitate analytical queries. Some loosely-coupled HTAP solutions employ NoSQL stores, like HBase [3] or Cassandra [1], for operational workloads, and periodically copy data from the NoSQL stores into files in columnar format like Parquet or ORC-File on the shared storage, so that SQL-on-Hadoop engines, like Hive [35] or SparkSQL [13], can efficiently query them. The data evolution across different zones in these HTAP systems/solutions is constantly on-going, posing a significant challenge to building and maintaining indexes.

Existing indexing solutions on multi-zone HTAP systems either support index on the transaction-friendly zone only, like in SnappyData [28] and the loosely coupled HTAP solutions, or support separate indexes on different zones, like in MemSQL [7]. First of all, being able to efficiently query historical data is very important for real-time analytics, especially for analytical queries that are part of a transaction in the true HTAP scenario. As a result, the index needs to cover both recent data and historical data. Secondly, having separate indexes on different zones exposes a divided view of data. This requires queries to perform extra work to combine index query results that span multiple zones. In particular, with the constant evolving nature of HTAP data, it is non-trivial for queries to make sure that there is no duplicate or missing data in the final results. Therefore, it is highly desirable to have a consistent and unified index across the different zones in an HTAP system.

Contributions. To tackle the challenges of indexing in a large-scale HTAP system, we present Umzi, the multi-version and multi-zone LSM-like index in the context of Wildfire. Umzi provides a consistent and unified indexing view across the groomed and post-groomed zones in Wildfire. To the best of our knowledge, Umzi is the first unified multi-zone indexing method for large-scale HTAP systems.

Umzi employs an LSM-like structure with lists of sorted index runs to avoid in-place updates. A novel index-run format that combines hash and sort techniques is introduced to flexibly answer equality/range queries as well as the combination of both using the index. Runs are organized into multiple levels as in today's NoSQL systems, e.g., LevelDB [6] and RocksDB [9]. A new run is added into the lowest level, i.e., level 0, and runs are periodically merged into higher levels within a zone. However, when data evolve from one zone to another, an *index evolve* operation is introduced to build new index runs in the new zone and garbage-collect obsolete index runs from the old zone in a coordinated way, so that the entire index is always in a consistent state. To fully exploit the storage hierarchy, lower index levels can be made non-persistent to speed up frequent merges, and we dynamically adjust cached index runs from memory and SSD to speed-up index lookups and transactional processing. In Umzi, all operations are carefully designed to be non-blocking such that readers, i.e., index queries, are always lock-free while only negligible locking overhead is incurred for index maintenance.

Paper organization. Section 2 provides the background on Wildfire. Section 3 describes an overview of the Umzi index. Section 4 presents the internal structure of Umzi components. Section 5 describes index maintenance operations in Umzi. Section 6 discusses some design decisions of Umzi to exploit the storage hierarchy. Section 7 introduces methods for processing index queries, i.e., range scans and point lookups. Section 8 reports the experimental evaluation of Umzi. Section 9 surveys related work. Finally, Section 10 concludes this paper.

2 BACKGROUND

2.1 Wildfire

Wildfire [15] is a distributed multi-master HTAP system consisting of two major pieces: Spark and the Wildfire engine. Spark serves as the main entry point for the applications that Wildfire targets, and provides a scalable and integrated ecosystem for various types of analytics on big data, while the Wildfire engine adds the support for high-speed transactions, accelerates the processing of application requests, and enables analytics on newly-ingested data. All inserts, updates, and deletes in Wildfire are treated as *upserts* based on the user-defined primary key. Wildfire adopts last-writer-wins semantics for concurrent updates to the same key, and snapshot isolation of quorum-readable content for queries, without having to read the data from a quorum of replicas to satisfy a query.

A table in Wildfire is defined with a *primary key*, a *sharding key*, and optionally a *partition key*. Sharding key is a subset of the primary key, and it is primarily used for load balancing of transaction processing in Wildfire. Inserted records are routed by the sharding key to different shards. A table shard is replicated into multiple nodes, where one replica serves as the shard-leader while the rest are slaves. Any replica of a shard can ingest data. A table shard is the basic unit of a lot of processes in Wildfire, including grooming, post-grooming, and indexing (details later). In contrast, the partition key is for organizing data in a way that benefits the analytics queries. Typically, the sharding key is very different from the partition key. For example, an IoT application handling large volumes of sensor readings could use the device ID as the sharding key, but the date column as the partition key to speed up time-based analytical queries.

Wildfire adds the following three hidden columns to every table to keep track of the life of each record and support snapshot isolation as well as time travel. The column beginTS (begin timestamp) indicates when the record is first ingested in Wildfire; endTS (end timestamp) is the time when the record is replaced by a new record with the same primary key; prevRID (previous record ID) holds the RID (record ID) of the previous record with the same key.

In order to support both OLTP and OLAP workloads efficiently within the same system, Wildfire divides data into multiple zones where transactions first append their updates to the OLTP-friendly zone, which are gradually migrated to the OLAP-friendly zone. Figure 1 depicts the data lifecycle in Wildfire across multiple zones.

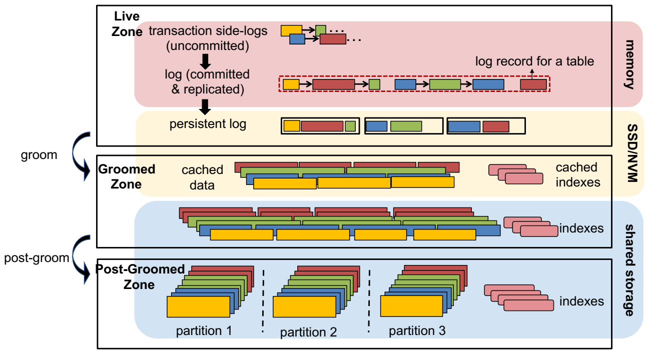


Figure 1: Data Lifecycle in Wildfire

Live Zone. A transaction in Wildfire first appends uncommitted changes in a transaction local side-log. Upon commit, the transaction tentatively sets beginTS for each record using the local wall clock time (beginTS is reset later in the groomed zone), and appends its side-log to the committed transaction log, which is also replicated for high-availability. The committed log is kept in memory for fast access, and also persisted on the local SSDs using the Parquet columnar-format. Since this zone has the latest ingested/updated data, it is called the *live zone*.

Groomed Zone. To bound the growth of the committed log and resolve conflicts from different replicas, each shard in Wildfire has a designated *groomer* which periodically (e.g., every second) invokes a *groom* operation to migrate data from the live zone to the groomed zone.

The groom operation merges, in the time order, transaction logs from shard replicas, resolves conflicts by setting the monotonic increasing beginTS for each record, and creates a Parquet columnar-format data file, called a *groomed block*, in the shared storage as well as the local SSD cache. Each groomed block is uniquely identified by a monotonic increasing ID called *groomed block ID*. Note that the beginTS set by the groomer is composed of two parts. The higher order part is based on the groomer's timestamp, while the lower order part is the transaction commit time in the shard replica. Thus, the commit time of transactions in Wildfire is effectively postponed to the groom time. The groomer also builds indexes over the groomed data.

Post-Groomed Zone. The groomer only sets the beginTS for each record. EndTS and prevRID still need to be set to support snapshot isolation and time travel. In addition, up to the groomed zone, data is still organized according to the sharding key, and they need yet to be re-organized based on the more analytics-friendly partition key. To achieve these tasks, another separate process, called *post-groomer*, periodically (e.g., every 10 minutes) performs *post-groom* operations to evolve the newly groomed data to the post-groomed zone.

The post-groom operation first utilizes the post-groomed portion of the indexes to collect the RIDs of the already post-groomed records that will be replaced by the new records from the groomed zone. Then, it scans the newly groomed blocks to set the prevRID fields using the RIDs collected from the index, and re-organizes the data into post-groomed blocks on the shared storage according to the OLAP-friendly partition key. The post-groomer also uses the same set of RIDs from the index to directly locate the to-be-replaced records and sets their endTS fields. Since the post-groomer is carried out less frequently than the groomer, it usually generates much larger blocks, which results in better access performance on shared storage. At the end, the post-groomer also notifies the indexer process to build indexes on the newly post-groomed blocks.

While both groomed and post-groomed blocks reside in the shared storage, based on the access patterns of a node, they are also cached in the local SSDs of that node, similarly to the indexes.

2.2 LSM-tree

Since Umzi employs an LSM-like structure, here we brief introduce the background of LSM-trees. Interested readers can refer to [27] for a survey of recent research on LSM-trees. The Log-Structured Merge-tree (LSM-tree) is a write-optimized persistent index structure. It first appends all writes into a memory table. When the memory table is full, it is flushed into disk as a sorted run using sequential I/Os. A query over an LSM-tree has to look up all runs to perform reconciliation, i.e., to find the latest version of each key.

As runs accumulate, query performance tends to degrade. To address this, runs are periodically merged together to improve query performance and reclaim disk spaced occupied by obsolete entries. Two LSM merge policies are commonly used in practice, i.e., leveling and tiering [27]. In both merge policies, runs are organized into levels, where a run at level $L + 1$ is T times larger than the run at level L. The leveling policy optimizes for queries by limiting only one run per level. A run is merged with the one at the higher level when its size reaches a threshold. In contrast, the tiering policy optimizes for write amplification by allowing multiple runs at each level. Multiple runs at level L are merged together into a new run at level $L + 1$.

3 UMZI OVERVIEW

In Wildfire, depending on the freshness requirement, a query may need to access data in the live zone, groomed zone, and/or the post-groomed zone. We choose to build indexes over the groomed zone and the post-groomed zone. Indexing does not cover the live zone for a few reasons. First, since groomer runs very frequently, data in the live zone is typically small, which alleviates the need for indexing. Secondly, to support fast data ingestion, we cannot afford maintaining the index on a record basis.

In a nutshell, Umzi employs an LSM-like [31] structure with multiple runs. Each groom operation produces a new index run, and runs are merged over time to improve query performance. Even though Umzi is structurally similar to LSM, it has significant differences from existing LSM-based indexes. First, existing LSM-based indexes either store the records directly into the index itself, e.g., LevelDB [6] and RocksDB [9], or store the records separately with the assumption that each record has a fixed RID, e.g., WiscKey [25] [1]. However, both approaches do not work well in Wildfire. Storing records into the LSM-tree incurs too much write amplification during merges, significantly affecting ingestion performance. While for the second approach, as data evolve from one zone to another, RIDs also change.[2] To accommodate the multi-zone design and the evolving nature, Umzi divides index runs into multiple zones accordingly. Runs in each zone are chained and merged, as in LSM indexes. When data evolve from one zone to another, Umzi performs the evolve operation

[1] The same assumption is generally held by non-LSM-based indexes as well

[2] In Wildfire, an RID is identified by the combination of zone, block ID, and record offset.

to migrate affected index entries to the target zone accordingly. Since data evolution in Wildfire is accomplished by a number of loosely-coupled distributed processes, it is critical for index evolve operations to require minimum coordination among distributed processes and incur negligible overhead for queries.

Furthermore, Umzi targets at multi-tier storage hierarchies in distributed environment, including memory, SSD and shared storage. Index runs are persisted on shared storage for durability, while Umzi aggressively exploits local memory and SSD as a caching layer to speed up index queries. Umzi dynamically adjusts cached index runs based on the space utilization and the age of the data, even without ongoing queries. To improve merge performance and avoid frequent rewrites on shared storage, Umzi allows certain lower levels to be non-persisted, i.e., their runs are only stored in local memory and optionally SSD.

Even though we present Umzi with two zones, i.e., groomed zone and post-groomed zone, this structure does not limits the applicability of Umzi to other systems. It is straightforward to extend Umzi to support other HTAP systems with arbitrary number of zones. To this end, one needs to structure Umzi with multiple run lists, each of which corresponds to one zone of data. When data evolves from one zone to another, the indexing process should be notified to trigger an index evolve operation to migrate index entries accordingly.

As mentioned in Section 2, a table shard is the basic unit of groomer and post-groomer processes in Wildfire. This is also the case for indexing. In the distributed setting of Wildfire, each Umzi index structure instance serves a single table shard. There are a number of indexer daemons running in the cluster. Each runs independently, and is responsible for building and maintaining index for one or more index structure instances. As a result, this paper describes the Umzi index design from the perspective of one table shard.

The following four sections describe the detailed design of Umzi and its index maintenance operations. Without loss of generality, we assume Umzi is used as a primary index throughout the paper.

4 INDEX STRUCTURE

This section details the internal structure of Umzi. We first describe the index definition of Umzi, followed by the singe-run storage format and multi-run structure respectively.

4.1 Index Definition

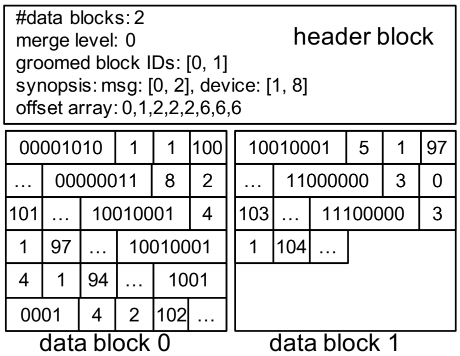
Umzi is designed for supporting both equality queries and range queries, as well as facilitating index-only access plans if possible. Index definitions in Umzi reflect these design goals. An index is defined with key columns plus optionally included columns. Index key columns can be a composition of equality columns (for equality predicates) and sort columns (for range predicates). Included columns are extra columns to enable efficient index-only queries. If equality columns are specified, we also store the hash value of equality column values to speed-up index queries, which makes Umzi a combination of hash and range index. In an example IoT application, the user can define the deviceID as the equality column, while the message number as the sort column. As a special case, the user could leave out the equality column(s), which makes Umzi a complete range index. Similarly, omitting the sort columns would turn Umzi into a hash index. The flexibility of this index structure helps Umzi to answer equality queries, range queries, and combinations of the two, with one index.

	hash	device	msg	beginTS	RID
0	0000 0101	1	1	100	...
1	0010 0011	8	2	101	...
2	1001 0001	4	1	97	...
3	1001 0001	4	1	94	...
4	1001 0001	4	2	102	...
5	1001 0001	5	1	97	...
6	1110 0000	3	0	103	...
7	1110 0000	3	1	104	...

(a) Example Run Data

	offset
000	0
001	1
010	2
011	2
100	2
101	6
110	6
111	6

(b) Offset Array



(c) Physical Layout of An Index Run

Figure 2: Example Index Run: *device* is the equality column, *msg* is the sort column, and there is no included columns. For simplicity, we assume the hash value takes only one byte.

4.2 Run Format

Each run in Umzi can be logically viewed as a sorted table containing the hash column, index key columns, included columns, beginTS and RID. As mentioned before, the hash column stores the hash value of equality columns (if any) to speed up queries with equality predicates. The beginTS column indicates the timestamp when the indexed record is inserted, which is generated by groomers in Wildfire (Section 2). The RID column defines the exact location of the indexed record. Index entries are ordered by the hash column, equality columns, sort columns, and descending order of beginTS. We sort the beginTS column in descending order to facilitate the access of more recent versions. All ordering columns, i.e., the hash column, equality columns, sort columns and beginTS, are stored in lexicographically comparable formats, similar to LevelDB [6], so that keys can be compared by simply using memory compare operations when processing index queries. Figure 2 shows an example index run, where *device* is the equality column and *msg* is the sort column. There is no included columns in this example, and we assume the hash value takes only one byte. Figure 2a shows the index rows in this run, where the hash value is shown in the binary format.

An index run is physically stored as a header block plus one or more fixed-size data blocks. The header block contains the metadata information of the index run, such as the number of data blocks, the merge level this run belongs to (Section 5), and the range of groomed block IDs to which this run corresponds.

To prune irrelevant runs during index queries, we also store a synopsis in the header block. The synopsis contains the range (min/max values) of each key column stored in this run. A run can be skipped by an index query if the input value of some key column does not overlap with the range specified by the synopsis. Figure 2c shows an example index run layout which contains one header block and two data blocks.

When equality columns are specified in the index definition, we store in the header block an offset array of 2^n integers to facilitate index queries. The offset array maps the value of the most significant n bits of hash values to the offset in the index run. When processing index queries, the offset array can be used to provide a more compact start and end offset for binary search. For example, Figure 2b shows the offset array with the most significant 3 bits of hash values from Figure 2a.

4.3 Multi-Run Structure

An example multi-run structure of Umzi is shown in Figure 3. Similar to LSM indexes, Umzi contains multiple runs. A groom operation produces a new run to level 0, and runs from lower levels are gradually merged into higher levels to improve query performance. Each run is further labeled with the range of groomed block IDs, where larger IDs correspond to newer groomed blocks.

In this meanwhile, to accommodate the multi-zone design and data evolving nature of the HTAP systems, Umzi divides index runs in multiple zones accordingly. For concurrency control purpose, runs in each zone are chained together based on their recency into a list, where the header points to the most recent run. We will further discuss concurrency control of Umzi in Section 5.1. Based on this multi-zone design, Umzi only merges runs within the same zone. When data evolves from one zone to another, an index evolve operation is triggered to migrate index entries to the target zone accordingly.

The assignment of levels to zones are configurable in Umzi. For example in Figure 3, levels 0 to 5 are configured as the groomed zone, while levels 6 to 9 are configured as the post-groomed zone.

5 INDEX MAINTENANCE

In this section we describe index maintenance operations in Umzi, including index build, merge, and evolve. Before presenting the details of index maintenance operations, we first discuss concurrency control in Umzi since index maintenance is performed concurrently with queries. Finally, we also briefly discuss how recovery is performed in Umzi.

5.1 Concurrency Control

Umzi aims at providing non-blocking and lock-free access for queries. To this end, Umzi relies on atomic pointers and chains runs in each zone together into a linked list, where the header points to the most recent run. All maintenance operations are carefully designed so that each index modification, i.e., a pointer modification, always results in a valid state of the index. As a result, queries can always traverse run lists sequentially without locking to get correct results. To minimize contentions caused by concurrent index maintenance operations, each level is assigned a dedicated index maintenance thread. A short duration lock is acquired when modifying the run list to prevent concurrent modifications. Note that the locking overhead is negligible since locks are only used to prevent concurrent modifications to the run list, which happens infrequently. Moreover, these locks never block any index queries.

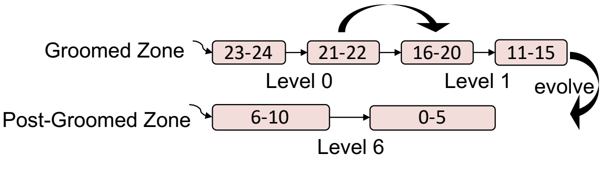


Figure 3: Multi-Run Structure in Umzi

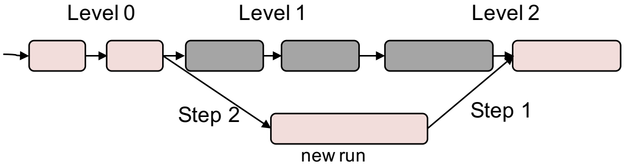


Figure 4: Index Merge Example

5.2 Index Build

After a groom operation is completed, Umzi builds an index run over the newly groomed data block. This is done by simply scanning the data block and sorting index entries in ascending order of hash values, sort columns, equality columns and descending order of beginTS. Along with writing sorted index entries back to data blocks, the offset array can be computed on-the-fly. Finally, the new run becomes the new header of the run list for the groomed zone. Note that the order of pointer modifications is important to guarantee the correctness for concurrent queries, where the new run must be set to point to the header before the header pointer is modified.

5.3 Index Merge

In order to easily trade-off write amplification and query performance, Umzi employs a hybrid merge policy similar to [20]. This policy is controlled by two parameters K, the maximum number of runs per level, and T, the size ratio between runs in adjacent levels. Each level L maintains the first run as an active run, while the remaining runs are inactive. Incoming runs from level $L-1$ are always merged into the active run of level L. When the active run in level L is full, i.e., its size is T times larger than an inactive run in level $L-1$, it is marked inactive and a new active run is created. Finally, when level L contains K inactive runs, they are merged together with the active run in level $L+1$.

After a merge, the new run replaces old runs in the run list. As shown in Figure 4, this is done by first setting the new run point to the next run of the last merged run, and then set the previous run before the first merged run point to the new run. A lock over the run list is acquired during the replacement, since otherwise pointers could be concurrently modified by other maintenance threads. There is no need for a query to acquire any locks when traversing the list; it sees correct results no matter whether the old runs or the new run are accessed.

5.4 Index Evolve

In Wildfire, the post-groomer periodically moves groomed data blocks to the post-groomed zone. After a post-groom operation, groomed data blocks are marked deprecated and eventually deleted to reclaim storage space. As a result, index entries in Umzi must be migrated as well so that deprecated groomed blocks are no longer referenced. However, index evolving in distributed HTAP systems is non-trivial due to the following problems.

First, HTAP systems like Wildfire are often composed of several loosely-coupled distributed processes. The post-groomer in Wildfire is a separate process running on a different node from the indexer process. In a distributed environment, one requirement for index evolving is to minimize the coordination among multiple processes. Second, an index evolve operation must apply multiple modifications to the index. This requires index evolve operations to be carefully designed to ensure the correctness for concurrent queries without blocking them.

To tackle the first problem, an index evolve operation in Umzi is performed asynchronously by the indexer process with minimum coordination, as shown in Figure 5. Each post-groom operation is associated with a post-groom sequence number (PSN). After a post-groom operation, the post-groomer publishes the metadata for this operation and updates the maximum PSN. In the meanwhile, the indexer keeps track of the indexed post-groom sequence number, i.e., IndexedPSN, and keeps polling the maximum PSN. If IndexedPSN is smaller than the maximum PSN, the indexer process performs an index evolve operation for IndexedPSN+1, which guarantees the index evolves in a correct order, and increments IndexedPSN when the operation is finished. Note that asynchronous index evolution has no impact on index queries since a post-groom operation only copies data from one zone to another without producing any new data. For a query, it makes no difference to access a record from the groomed zone or post-groomed zone.

For the concurrency control issue, we decompose the index evolve operation into a sequence of atomic sub-operations. Each sub-operation is guaranteed to result in a valid state of the index, ensuring that concurrent queries always see correct results when traversing the run lists. Specifically, an index evolve operation for a given PSN is performed as follows. First, the indexer builds an index run for post-groomed data blocks associated with this PSN, and adds it atomically to the post-groomed run list. Note that this run still contains the range of groomed block IDs it corresponds to. Second, the indexer atomically updates the maximum groomed blocked ID covered by the post-groomed run list, based on the newly built run. All runs in the groomed run list with end groomed block ID no larger than this value would be automatically ignored by queries since entries in these runs are already covered by the post-groomed run list. Finally, these obsolete runs are garbage collected from the groomed run list. Note that during each step, a lock over the run list is acquired when modifying a run list to prevent concurrent modifications by other maintenance threads.

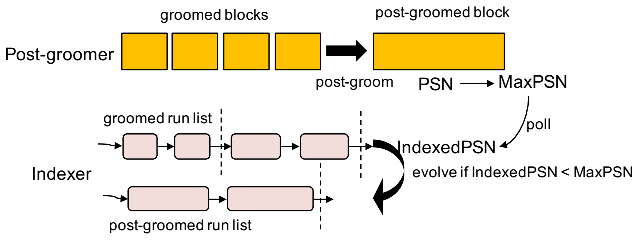


Figure 5: Interaction between Post-groomer and Indexer

We further illustrate the index evolve operation with an example depicted in Figure 6. Suppose the groomed blocks 11 to 18 have been post-groomed, and the indexer now performs an index evolve operation for this post-groom operation accordingly. First, the indexer builds a new run labeled 11-18 for the newly post-groomed data, and atomically adds it to the post-groomed run list. The indexer then atomically updates the maximum groomed blocked ID of the post-groomed run list from 10 to 18. At this moment, run 11-15 will be ignored by subsequent queries since it is fully covered by run 11-18. Finally, this obsolete run is garbage collected from the groomed list, which concludes this index evolve operation.

It is straightforward that each step of an evolve operation only makes one modification to the index, and is thus atomic. Between any two of the above three steps, the index could contain duplicates, i.e., a record with the same version could appear in both a groomed run and a post-groomed run. Moreover, even after the last step of the index evolve operation, the index may still contain duplicates since groomed blocks consumed by a post-groom operation may not align perfectly with the boundaries of index runs. However, duplicates are not harmful to index queries. Since a query only returns the most recent version of each key, duplicates are removed on-the-fly during query processing (Section 7).

5.5 Recovery

We assume runs in Umzi are persisted in shared storage. After each index evolve operation, the maximum groomed blocked ID for the post-groomed run list and IndexedPSN are also persisted. However, an indexer process could crash, losing all data structures in the local node. To recover an index, we mainly need to reconstruct run lists based on runs stored in shared storage, and cleanup merged and incomplete runs if any.

A run list can be recovered by examining all runs in shared storage. Runs are first sorted in descending order of end groomed blocked IDs, and are added to the run list one by one. If multiple runs have overlapping groomed block IDs, the one with largest range is selected, while the rest are simply deleted since they have already been merged.

6 UMZI ON MULTI-TIER STORAGE HIERARCHY

Recall that Umzi is designed for large-scale distributed HTAP systems running on multi-tier storage hierarchy, i.e., memory, SSD, and distributed shared storage. Even though shared storage provides several key advantages for distributed HTAP systems such as fault tolerance and high availability, it brings significant challenges when designing and implementing an indexing component. Shared storage generally does not support in-place

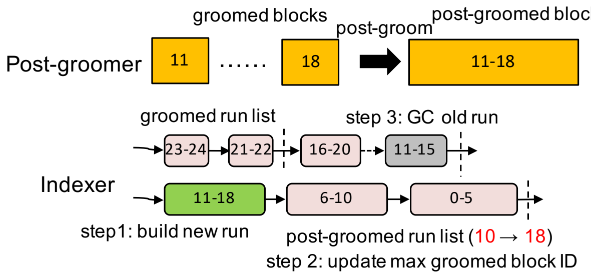


Figure 6: Index Evolve Example

updates and random I/Os, and prefers a small number of large files to reduce metadata overhead. Furthermore, accessing shared storage through networks is often costly, incurring high latency for index queries.

So far, we only discussed how Umzi eliminates random I/Os and in-place updates by adopting an LSM-like structure. In this section, we present solutions adopted by Umzi to improve storage efficiency in a multi-tier storage hierarchy.

6.1 Non-Persisted Levels

In a traditional LSM design, on-disk runs of all levels are persisted on disk equivalently. Even though this design garbage collects olds runs after a merge, it introduces a large overhead on shared storage because of writing a large number of (potentially small) files. To avoid frequently rewriting a large number of small files on shared storage, Umzi supports non-persisted levels, i.e., certain low levels of the groomed zone can be configured as non-persisted.

Runs in persisted levels are always stored in shared storage for fault tolerance and can be cached in local memory and SSD to speedup queries. However, runs in non-persisted levels are only cached in local memory and optionally spilled to SSD if memory is full, but they are not stored in shared storage for efficiency.

Introducing non-persisted levels complicates the recovery process of Umzi, since after a failure all runs in non-persisted levels could be lost. To address this, Umzi requires level 0 must be persisted to ensure that we do not need to rebuild any index runs from groomed data blocks during recovery. Moreover, if level L to K are configured as non-persisted, runs in level L-1 cannot be deleted immediately after they are merged into level L. Otherwise, if the node crashes, we would again lose index runs since the new run is not persisted in shared storage. To handle this, when merging into non-persisted levels, old runs from level L-1 are not deleted but rather recorded in the new run. When the new run is finally merged into a persisted level again, i.e., level K+1, its ancestor runs from level L-1 can be safely deleted.

6.2 Cache Management

As mentioned before, accessing shared storage through networks is often costly and incurs high latencies for index queries. To address this, Umzi aggressively caches index runs using local memory and SSD, even without ongoing queries. We assume most frequently accessed index runs fit into the local SSD cache so that shared storage is mainly used for backup. However, when the local SSD cache is full, Umzi has to remove some index runs to free up the cache space. For this purpose, we assume recent data is accessed more frequently. As the index grows, Umzi dynamically purges old runs, i.e., runs in high levels, from the SSD cache to free up the cache space. In contrast, when the local SSD is spacious, Umzi aggressively loads old runs from shared storage to speedup future queries.

To dynamically purge and load index runs, Umzi keeps track of the current cached level that separates cached and purged runs, as shown in Figure 7. When the SSD is nearly full, the index maintenance thread purges some index runs and decrements the current cached level if all runs in this level have been purged. When purging an index run, Umzi drops all data blocks from the SSD while only keeps the header block for queries to locate data blocks. On the contrary, when the SSD has free space, Umzi loads recent runs (in the reverse direction of purging) into SSD, and increments the current cached level when all runs in the current cached level have been cached. Umzi further adopts a write-through cache policy when creating new index runs during merge or evolve. That is, a new run is directly written to the SSD cache if it is below (lower than) the current cache level.

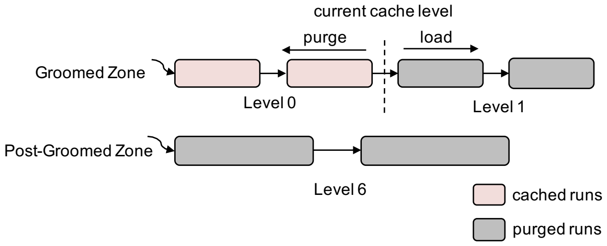


Figure 7: Cache Management in Umzi

7 INDEX QUERY

In this section, we discuss how to process index queries on Umzi. Since Umzi is a multi-version index, a query has to specify a query timestamp (queryTS), and only the most recent version for each matching key is returned, i.e., the version with largest beginTS such that beginTS ≤ queryTS. In general, two types of index queries are supported. The range scan query specifies values for all equality columns (if any) and bounds for the sort columns, and returns the most recent version of each matching key. The point lookup query specifies the entire index key (i.e., the primary key), and at most one matching record is returned.

A query first collects candidate runs by iterating the run lists and checking run synopses. A run is considered as a candidate only if all column values as specified in the query satisfy the column ranges in the synopsis. Also note that all runs are read from the SSD cache. In case that a query must access purged runs, we first transfer runs from shared storage to the SSD cache on a block-basis, i.e., the entire run data block is transferred at a time, to facilitate future accesses. After the query is finished, the cached data blocks are released, which are further dropped in case of cache replacement. Depending on the query type, the details of query processing are as follows.

7.1 Range Scan Query

For a range scan query, we first discuss how to search a single run to get matching keys. Results returned from multiple runs are further reconciled, since results from newer runs could override those from older runs, to guarantee only the most recent version is returned for each matching key.

7.1.1 Search Single Run. Searching a single run returns the most recent version for each matching key in that run. Since a run is a table of sorted rows, the query first locates the first matching key using binary search with the concatenated lower bound, i.e., the hash value, equality column values, and the lower bound of sort column values. If the offset array is available, the initial search range can be narrowed down by computing the most significant n bits of the hash value (denoted as i) and taking the i-th value in the offset array.

After the first matching key is determined, index entries are then iterated until the concatenated upper bound is reached, i.e., the concatenation of the hash value, equality column values, and the upper bound of sort column values. During the iteration, we further filter out entries failing timestamp predicate beginTS ≤ queryTS. For the remaining entries, we simply return for each key the entry with the largest beginTS, which is straightforward since entries are sorted on the index key and descending order of beginTS.

Consider again the example run in Figure 2. Recall that *device* is the equality column, while *msg* is the sort column. Consider a range scan query with *device* = 4, 1 ≤ *msg* ≤ 3, and queryTS = 100. We first take the most significant 3 bits of hash(4) = 1001 0001, i.e., 100, to obtain the initial search range from the offset array, i.e., 2 to 6. The first matching key is still entry 2 after binary search with the input lower bound (1001 0001, 4, 1). We then iterate index entries starting from entry 2. Entry 2 is returned since it is the most recent version for key (4, 1), while entry 3 is filtered out since it is an older version of entry 2. However, entry 4 is filtered out because its beginTS 102 is beyond the queryTS 100. We stop the iteration at entry 5, which is beyond the input upper bound (1001 0001, 4, 3).

7.1.2 Reconcile Multiple Runs. After searching each run independently, we have to reconcile results returned from multiple runs to ensure only the most recent version is returned for each matching key. In general, two approaches can be used for reconciliation: the set approach and the priority queue approach.

Set Approach. In the set approach, the query searches from the newest to the oldest runs sequentially, and maintains a set of keys which have already been returned to the query. If a key has not been returned before, i.e., not in the set, it is added to the set and the corresponding entry is returned to the query; otherwise, the entry is simply ignored since we have already returned a more recent version from the newer runs. The set approach mainly works well for small range queries since it requires intermediate results to be kept in memory during query processing.

Priority Queue Approach. In the priority queue approach, the query searches multiple runs together and feeds the results returned from each run into a priority queue to retain a global ordering of keys, which is similar to the merge step of merge sort. Once keys are ordered, we can then simply select the most recent version for each key and discard the rest without remembering the intermediate results.

7.2 Point Lookup Query

The point lookup query can be viewed as a special case of the range scan query, where the entire primary key is specified such that at most one entry is returned. As an optimization, one can search from newest runs to oldest runs sequentially and stop the search once a match is found. Here we can use exactly the same approach from above to search the single run, where the lower bound and upper bound of sort column values are the same.

For a batch of point lookups, we first sort the input keys by the hash value, equality column values, and sort column values, to improve search efficiency. The sorted input keys are searched against each run sequentially from newest to oldest, one run at a time, until all keys are found or all runs to be searched are exhausted. This guarantees that each run is accessed sequentially and only once.

8 EXPERIMENTAL EVALUATION

In this section, we report the experimental evaluation of Umzi. We first evaluate the index build performance, index query performance, and further study the end-to-end query performance with concurrent data ingestion. We first outline the general experiment setup, then report and discuss the experimental results.

As mentioned in Section 3, there are a distributed cluster of indexer daemons running in Wildfire, each independently responsible for building and maintaining index for one or more Umzi index structure instances (one per table shard). As a result, Umzi scales up and down nicely with more or less indexer daemons. Since the goal of our experiments is to demonstrate the performance of Umzi index structure, we focus on a single shard setting for our experiments.

Note that since all experiments were conducted inside Wildfire, which is closely tied to an IBM product, we cannot report absolute performance numbers. As a result, we report normalized performance numbers, with the normalization process explained for each experiment.

8.1 Experiment Setup

All experiments are performed against a single table shard on a single node using a dual-socket Intel Xeon E5-2680 server (2.40GHZ) with 14 cores in a socket (28 with hyper-threading) and 1.5TB of RAM. The operating system is Ubuntu 16.04.2 with Linux kernel 4.4.0-62. The node uses an Intel 750 Series SSD as the SSD cache. For end-to-end experiments, we use GlusterFS [2] as the shared storage layer.

We use a synthetic data generator to generate keys with include columns used in all experiments, where keys can be sequential or random. Note that our index only stores key and include columns instead of entire records, thus a key generator is sufficient for our experiments. Throughout the experiments, we consider three different index definitions as below:

- I1: one equality column, one sort column, and one include column
- I2: two equality columns and one include column
- I3: one equality column and one include column

Each column is a long type with 8 bytes. Unless otherwise noted, we use index definition I1 as the default case. Each of the following experiments was reported for three times, and the average number is reported.

Moreover, for the scope of this paper, we only focus on the time of index lookups, while omitting the time of retrieving records based on the fetched RIDs, since the latter depends on the record storage format and is orthogonal to the indexing method.

8.2 Index Building Performance

In the first set of experiments, we evaluate the performance of building index runs, which is the primitive operation for Umzi's index maintenance after a groom or post-groom cycle. Figure 8 shows the results for the time it takes to build an index run using the three different index definitions mentioned above as we increase the number of entries in a run. The running time is normalized against the time of building a run with 1000 tuples using I1. As the graph shows, index building almost scales linearly with the number of rows. Furthermore, the index building time for index I3 is always faster than I1 and I2, since I3 has one fewer key column. The impact of the number of indexed columns on the index building time, however, is negligible, compared to the overhead of sorting entries during index building.

8.3 Index Query Performance

Next set of experiments evaluate the performance of querying Umzi under various settings. By default, an index contains 20 runs, where each index run has 100000 entries. We execute index

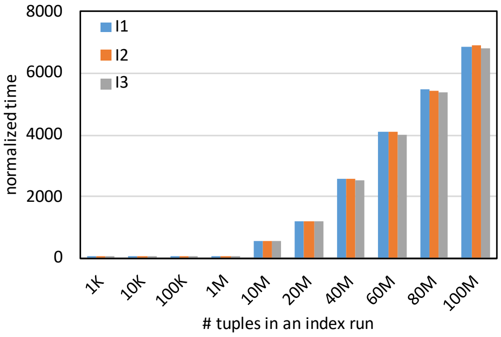
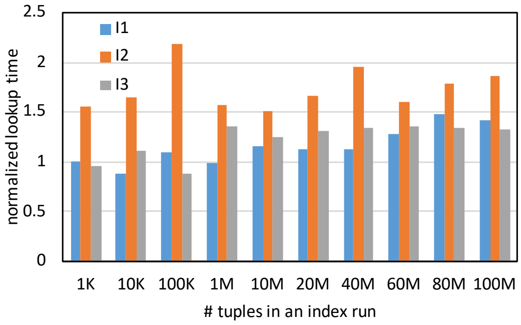
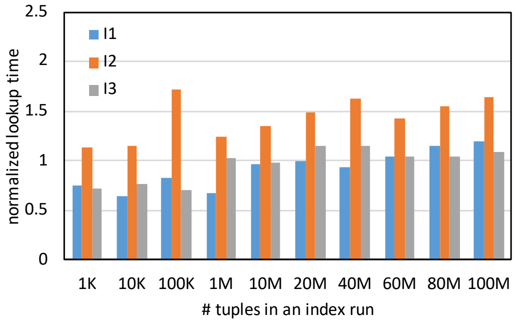


Figure 8: Index Building Performance



(a) Sequential queries



(b) Random queries

Figure 9: Single Run Query Performance

queries in a batch, where the default batch size is 1000. All index runs used in this set of experiments are cached in the local SSD.

We consider two kinds of entry characteristics for the following experiments, i.e., ingested with sequential keys or random keys. Sequential keys are sequentially generated by our synthetic key generator to simulate the time correlated keys, while random keys are randomly sampled from a uniform distribution without any temporal correlation. We further consider two kinds of key distribution in index queries: sequential and random. As the name suggests, sequential and random queries use sequentially and randomly generated keys in a batch, respectively.

8.3.1 Single Run. We first evaluate the index query performance against a single run. For brevity, we only report experiment results with sequentially ingested keys in this experiment. Since entries in a run are sorted on hash values, there is no difference to use sequentially or randomly ingested data. Figure 9 shows the normalized lookup time with varying run sizes and index definitions.

The time is normalized against the index lookup time of the sequential query over the run with 1000 tuples under the index definition I1. In general, the query time increases with larger index runs, since search keys spread across the run and potentially more I/Os are required to process the query batch when the run size increases. However, the impact of run size is limited because we use the hash offset array to locate the initial search range, and further use binary search to locate the exact location. Moreover, index lookup performance of index definition of I1 is comparable to that of I3, but index lookup performance of index definition I2 is generally slower since I2 contains two equality columns, making the hash offset array less effective in terms of locating the initial search range.

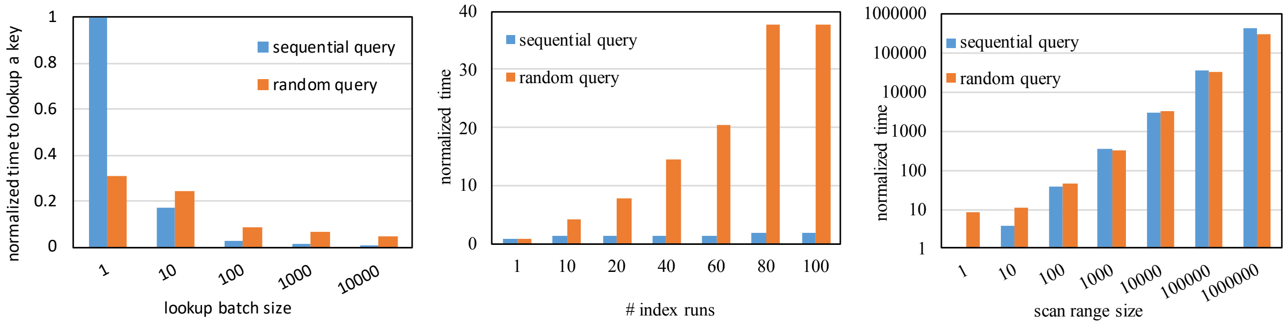
8.3.2 Multiple Runs with Sequential Keys. In this section we evaluate the index query performance against multiple index runs with sequentially ingested keys. We vary the query batch size and the number of index runs for the lookup queries, and the scan range for the range queries. The experiment results are shown in Figure 10.

Figure 10a shows the impact of the batch size on the index lookup performance. The index lookup time per key is normalized against the lookup time of the sequential query with batch size one. In general, sequential queries perform much better than random ones since the run synopsis enables pruning most of the irrelevant runs and leaving only a small fraction of the runs need to be searched (except for the case of batch size 1, where the sequential queries take longer because of some variances in the experiments). Furthermore, batching greatly improves index lookup performance, since once an index block is fetched into memory for a the lookup of a particular key, no additional I/O is required to fetch that block again for looking up other keys in the batch.

Figure 10b shows the index lookup performance with varying number of index runs. The query time is normalized against the time it takes to complete the sequential query against one run. As the result shows, the number of runs has limited impact over the sequential queries, since most irrelevant runs are simply pruned because of the run synopsis. However, the time of the random queries grows almost linearly with more runs, since more runs need to be searched to complete the batch of lookups.

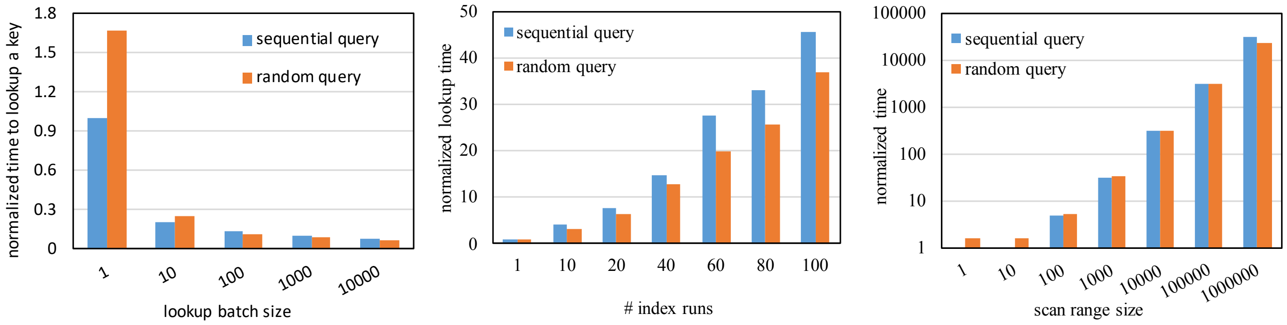
Finally, the performance of the range scan queries using the priority queue method is in Figure 10c. The time is normalized against the query time of the sequential query with range one. In general, the query time of the range scan query grows linearly with the query range, since larger ranges require more time to read index entries and output matching keys. Moreover, sequential or random ranges have little impact over the query performance, since the time of locating start position is negligible compared to scanning the index entries.

8.3.3 Multiple Runs with Random Keys. After investigating the index query time using sequentially ingested keys, this section evaluates the query performance against multiple index runs with randomly ingested keys. The results are shown in Figure 11. The numbers on the y-axes are normalized the same way as the corresponding numbers in Figure 10. In general, random keys render the run synopsis less useful, which decreases the performance of sequential queries since more runs need to be searched. However, the impact on the random queries is almost negligible,



(a) Varying batch size **(b) Varying number of runs** **(c) Varying scan ranges**

Figure 10: Query performance of multiple runs with sequentially ingested keys



(a) Varying batch size **(b) Varying number of runs** **(c) Varying scan ranges**

Figure 11: Query performance of multiple runs with randomly ingested keys

since the pruning capability of the run synopses is anyway limited when we have random keys in the query batch. As a result, the performance of sequential queries becomes similar to that of random queries.

8.4 End-to-end Experiments

The last set of experiments evaluates the end-to-end performance of Umzi in Wildfire as we perform data ingest and index lookups concurrently while Umzi's index maintenance operations are also handled in the background. By default, for each experiment, we ingest roughly 100000 random records per second. The groomer runs every second, and the post-groomer runs every 20 seconds. We also submit batches of 1000 random index lookup queries continuously. Each experiment lasts for 100 seconds.

For this set of experiments, we generate data with update rates that mimic a realistic IoT application, where the recent data are updated more frequently. The update rates are calculated based on the groom cycles: the ingested data for the latest groom cycle updates $p\%$ of data from the last groom cycle, and $0.1 \times p\%$ of data from the last 50 cycles, and $0.01 \times p\%$ of data in the last 100 cycles. By default, we set $p\% = 10\%$.

With this experimental setup, we investigate the impact of concurrent readers, percentage of updates, purged runs, and index evolve operations on index lookup time. The results are summarized and discussed below.

8.4.1 Concurrent Readers. Figure 12 shows how varying the number of concurrent readers impact the average index lookup time. Each reader thread submits batches of 1000 lookup queries continuously. For brevity, we show results for only 4, 16, 28, 40, and 52 readers, and the experiment results are normalized against the lookup time with 1 reader from the beginning of the experiment. In addition, Figure 12 zooms into a 30 second period from the middle of the overall experiment to focus on the impact of concurrency as opposed to the index behavior over time as the index grows. As one can see, more concurrent readers have small impact on the query performance, which demonstrates the advantages of Umzi's lock-free design for the readers. The varying performance of the index lookup operation in this graph (and the rest of the graphs in this section) is due to the random input keys we generate for the index lookup requests. Based on the distribution of these random keys, the search for a key can lead to reading fewer or more index runs, which impacts the performance of a point lookup as seen previously.

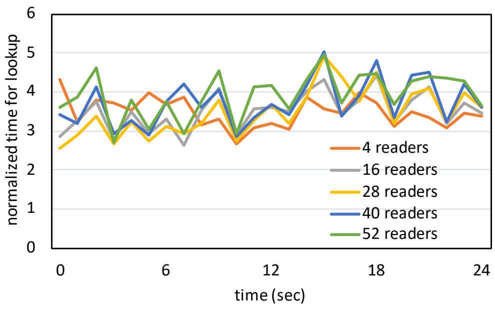


Figure 12: Performance with concurrent readers

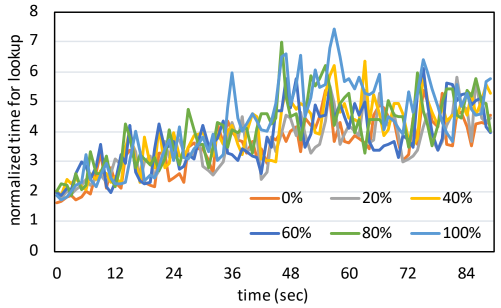


Figure 13: Varying percentage of update workloads

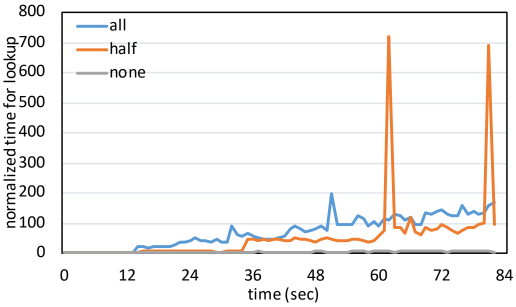


Figure 14: Performance with various purge levels

8.4.2 Updates. Figure 13 analyzes the impact of varying level of updates in a workload on the query performance. We change the update rate $p\%$ from 0% (read-only workload) to 100% (all ingested records are updates after the first groom cycle). As the graph demonstrates, updates have limited impact on the average query performance. The slightly increasing lookup time over time, which can be observed in all the experiments in this section, is due to the growing of the index run chain of Umzi.

8.4.3 Purged Runs. The impact of purged runs on the query performance is shown in Figure 14, where we manually set the purge level to control the percentage of purged runs. The running time is normalized against the performance of the no-purge case in the beginning of this experiment. As expected, Figure 14 emphasizes the significance of the SSD cache on the query performance. The latency of the lookup queries is much lower when all the index runs are cached (*none*) compared to the cases where the *half* or *all* of the runs are purged. Moreover, when some runs are purged, we observe unpredictable latency spikes. The reason is that when purged runs are first accessed after being merged or evolved, they have to be fetched from the shared storage into the local SSD cache on a block by block basis as the queries require them.

8.4.4 Index Evolve Operations. Finally, we evaluate the impact of the index evolve operations on the query performance by enabling/disabling the post-groomer. The results are shown in Figure 15. The running time is normalized against the lookup time in the beginning of the experiment where the post-groomer (including index evolution), is enabled. As the graph illustrates, the index evolve operation has certain overhead over the query performance, since often the query may experience several cache misses after runs have been evolved. However, the overhead again is limited, since in the meanwhile the index evolve operation reduces the total of number of runs, which in turn improves the query performance.

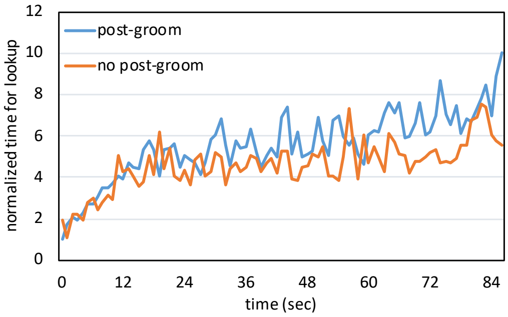


Figure 15: Impact of index evolve operations

9 RELATED WORK

In this section, we survey related work in indexing methods for HTAP systems, as well as LSM-like indexes.

Indexing in HTAP Systems. To satisfy the demand of fast transactions and analytical queries concurrently, many HTAP systems and solutions have been proposed recently. A recent survey of HTAP systems can be found in [32]. In-memory HTAP engines, e.g., SAP HANA [21], HyPer [22], Pelaton [14], Oracle TimesTen [23] and DBIM [29] take unique advantage of large main memories, e.g., random writes and in-place updates, while large-scale HTAP systems that go beyond the memory limit have to face inherently different challenges in the presence of disks and shared storage. MemSQL [7] supports skip-list or hash indexes over the in-memory row store, and LSM-like indexes over the on-disk column store. However, column store indexes cannot be combined with row indexes to provide a unified view for queries. SnappyData [28] only supports indexes over row tables, while providing no indexing support for column tables.

Another category of HTAP solutions typically glue multiple systems together to handle OLTP and OLAP queries. For example, one typical solution is to use a key-value store, such as HBase [3] or Cassandra [1], as the updatable storage layer, while resorting to SQL-on-Hadoop systems such as Spark-SQL [13] to process analytical queries with the help of data connectors. Other systems directly build upon updatable storage engines to handle both transactional and analytical queries, such as Hive [35] on HBase [3] and Impala [16] on Kudu [5]. In these solutions, indexes, if any, are exclusively managed by the storage engine to support efficient point lookups. However, none of these solutions support both fast ingestion and data scans, which is different from our system where ingested data evolves constantly to be more analytics-friendly.

LSM-like Index. The LSM-tree [31] is a persistent index structure optimized for write-heavy workloads. Instead of updating entries in place, which potentially requires a random I/O, the LSM-tree batches inserts into memory and flushes the data to disk using sequential I/O when the memory is full. It was subsequently adopted by many NoSQL systems, such as HBase [3], Cassandra [1], LevelDB [6] and RocksDB [9], for its superior write performance. Many variations of the original LSM-tree have been proposed as well. LHAM [30] is a multi-version data access method based on LSM for temporal data. FD-tree [24] is designed for SSDs by limiting random I/Os and uses fractional cascading [18] to improve search performance. bLSM [34] uses bloom filters to improve point lookup performance, and proposes a dedicated merge scheduler to bound write latencies. AsterixDB [12] proposes a general framework for using LSM-tree as secondary indexes. Ahmad and Kemme [11] present an approach to improve the merge process by offloading merge to dedicated nodes and a cache warm-up algorithm to alleviate cache misses after merge. LSM-Trie [36] organizes runs using a prefix tree-like structure to improve point lookup performance by sacrificing the ability to do range queries. WiscKey [25] reduces write amplification by only storing keys in the LSM tree while leaving values in a separate log. The work [26] presented efficient maintenance strategies for LSM-based auxiliary structures, i.e., secondary indexes and filters, to facilitate query processing. Monkey [19] is an analytical approach for automatic performance tuning of LSM trees. Dostoevsky [20] presents a lazy leveling merge policy for better trade-offs among query cost, write cost and space amplification. Accordion [17] optimizes LSM on large memories by in-memory flushes and merges. SlimDB [33] optimizes LSM-based key-value stores for managing semi-sorted data.

Different from existing work on LSM indexes, which focuses on a key-value store setting, Umzi is an end-to-end indexing solution in distributed HTAP systems. It supports the multi-zone design commonly adopted by large-scale HTAP systems, and evolves itself as data migrates without blocking queries. We further discuss how Umzi is designed to accommodate the multi-tier storage hierarchy, i.e., memory, SSD, and shared storage, to improve storage efficiency.

10 CONCLUSION

This paper describes Umzi, the first unified multi-version and multi-zone indexing method for large-scale HTAP systems in the context of Wildfire. Umzi adopts the LSM-like design to avoid random I/Os in shared storage, and supports timestamped queries for multi-version concurrency control schemes. Unlike existing LSM indexes, Umzi addresses the challenges posed by the multi-zone design of modern HTAP systems, and supports migrating index contents as data evolves from one zone to another. It also utilizes an interesting combination of hash and sort techniques to enable both equality and range queries using one index structure. Furthermore, it fully exploits the multi-level storage hierarchy of HTAP systems for index persistence and caching.

In the future, we plan to extend Umzi to build and maintain secondary indexes in HTAP systems. Then, we would like to perform more experimental evaluation on Umzi to study its performance under various workloads. Finally, we would also like to study other SSD cache management strategies, and evaluate their impact on query performance.

REFERENCES

[1] 2018. Cassandra. http://cassandra.apache.org/.

[2] 2018. GlusterFS. https://www.gluster.org/.

[3] 2018. Hbase. https://hbase.apache.org/.

[4] 2018. IBM DB2 Event Store. //https://www.ibm.com/us-en/marketplace/db2-event-store/.

[5] 2018. Kudu. https://kudu.apache.org/.

[6] 2018. LevelDB. http://leveldb.org/.

[7] 2018. MemSQL. http://www.memsql.com.

[8] 2018. Parquet. https://parquet.apache.org/.

[9] 2018. RocksDB. http://rocksdb.org/.

[10] 2018. Spark. http://spark.apache.org/.

[11] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction Management in Distributed Key-value Datastores. *Proc. VLDB Endow.* 8, 8 (April 2015), 850–861. https://doi.org/10.14778/2757807.2757810

[12] Sattam Alsubaiee et al. 2014. Storage Management in AsterixDB. *PVLDB* 7, 10 (2014), 841–852.

[13] Michael Armbrust et al. 2015. Spark SQL: Relational data processing in Spark. In *SIGMOD*. ACM, 1383–1394.

[14] Joy Arulraj et al. 2016. Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD*. ACM, New York, NY, USA, 583–598.

[15] Ronald Barber et al. 2017. Evolving Databases for New-Gen Big Data Applications.. In *CIDR*.

[16] MKABV Bittorf et al. 2015. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*.

[17] Edward Bortnikov et al. 2018. Accordion: Better Memory Organization for LSM Key-value Stores. *PVLDB* 11, 12 (2018), 1863–1875.

[18] Bernard Chazelle and Leonidas J Guibas. 1986. Fractional cascading: I. A data structuring technique. *Algorithmica* 1, 1 (1986), 133–162.

[19] Niv Dayan et al. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 79–94.

[20] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *SIGMOD*. 505–520.

[21] Franz Färber et al. 2012. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[22] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*. IEEE Computer Society, Washington, DC, USA, 195–206.

[23] Tirthankar Lahiri et al. 2013. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Eng. Bull.* 36, 2 (2013), 6–13.

[24] Yinan Li et al. 2010. Tree Indexing on Solid State Drives. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 1195–1206.

[25] Lanyue Lu et al. 2017. WiscKey: Separating keys from values in SSD-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 5.

[26] Chen Luo and Michael J. Carey. 2018. Efficient Data Ingestion and Query Processing for LSM-based Storage Systems. *CoRR* abs/1808.08896 (2018). arXiv:1808.08896

[27] Chen Luo and Michael J. Carey. 2018. LSM-based Storage Techniques: A Survey. *CoRR* abs/1812.07527 (2018). arXiv:1812.07527 http://arxiv.org/abs/1812.07527

[28] Barzan Mozafari et al. 2017. SnappyData: A Unified Cluster for Streaming, Transactions and Interactive Analytics.. In *CIDR*.

[29] Niloy Mukherjee et al. 2015. Distributed Architecture of Oracle Database In-memory. *PVLDB* 8, 12 (2015), 1630–1641.

[30] Peter Muth et al. 2000. The LHAM Log-structured History Data Access Method. *The VLDB Journal* 8, 3-4 (Feb. 2000), 199–221.

[31] Patrick O'Neil et al. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385.

[32] Fatma Özcan et al. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *SIGMOD*. ACM, 1771–1775.

[33] Kai Ren et al. 2017. SlimDB: A Space-efficient Key-value Storage Engine for Semi-sorted Data. *PVLDB* 10, 13 (2017), 2037–2048.

[34] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 217–228. https://doi.org/10.1145/2213836.2213862

[35] Ashish Thusoo et al. 2010. Hive-a petabyte scale data warehouse using hadoop. In *ICDE*. IEEE, 996–1005.

[36] Xingbo Wu et al. 2015. LSM-trie: an LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 71–82.

A Highly Scalable Labelling Approach for Exact Distance Queries in Complex Networks*

Muhammad Farhan, Qing Wang, Yu Lin, Brendan McKay
Australian National University
Canberra, Australia
{muhammad.farhan,qing.wang,yu.lin,brendan.mckay}@anu.edu.au

ABSTRACT

Answering exact shortest path distance queries is a fundamental task in graph theory. Despite a tremendous amount of research on the subject, there is still no satisfactory solution that can scale to billion-scale complex networks. Labelling-based methods are well-known for rendering fast response time to distance queries; however, existing works can only construct labelling on moderately large networks (million-scale) and cannot scale to large networks (billion-scale) due to their prohibitively large space requirements and very long preprocessing time. In this work, we present novel techniques to efficiently construct distance labelling and process exact shortest path distance queries for complex networks with billions of vertices and billions of edges. Our method is based on two ingredients: (i) a scalable labelling algorithm for constructing minimal distance labelling, and (ii) a querying framework that supports fast distance-bounded search on a sparsified graph. Thus, we first develop a novel labelling algorithm that can scale to graphs at the billion-scale. Then, we formalize a querying framework for exact distance queries, which combines our proposed highway cover distance labelling with distance-bounded searches to enable fast distance computation. To speed up the labelling construction process, we further propose a parallel labelling method that can construct labelling simultaneously for multiple landmarks. We evaluated the performance of the proposed methods on 12 real-world networks. The experiments show that the proposed methods can not only handle networks with billions of vertices, but also be up to 70 times faster in constructing labelling and save up to 90% of labelling space. In particular, our method can answer distance queries on a billion-scale network of around 8B edges in less than 1ms, on average.

1 INTRODUCTION

Finding the shortest-path distance between a pair of vertices is a fundamental task in graph theory, and has a broad range of applications [5, 11, 20, 26, 30, 31, 33]. For example, in web graphs, ranking of web pages based on their distances to recently visited web pages helps in finding the more relevant pages and is referred to as context-aware search [30]. In social network analysis, distance is used as a core measure in many problems such as centrality [11, 26] and community identification [5], which require distances to be computed for a large number of vertex pairs. However, despite extensive efforts in addressing the shortest-path distance problem for many years, there is still a high demand for scalable solutions that can be used to support analysis tasks over large and ever-growing networks.

Traditionally, one can use the Dijkstra algorithm [27] for weighted graphs or a breadth-first search (BFS) algorithm for unweighted graphs to query shortest-path distances. However, these algorithms are not scalable, i.e., for large graphs with billions of vertices and edges, they may take seconds or even longer to find the shortest-path distance between one pair of vertices, which is not acceptable for large-scale network applications where distances need to be provided in the order of milliseconds. To improve query time, a well-established approach is to precompute and store shortest-path distances between all pairs of vertices in an index, also called *distance labelling*, and then answer a *distance query* (i.e., find the distance between two vertices) in constant time with a single lookup in the index. Recent work [15] shows that such labelling-based methods are the fastest known exact distance querying methods on moderately large graphs (million-scale) having millions of edges, but still fail to scale to large graphs (billion-scale) due to quadratic space requirements and unbearable indexing construction time.

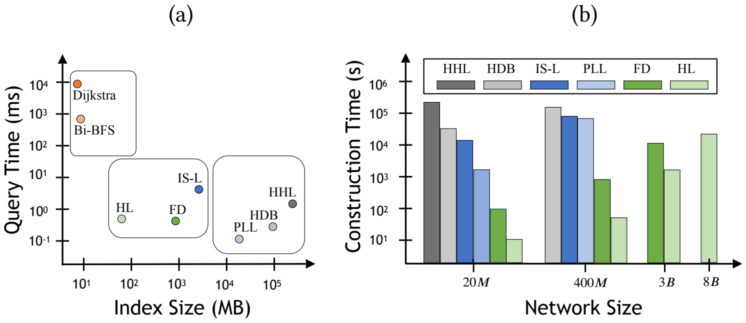
Thus, the question is still open as to how scalable solutions to answer exact distance queries in billion-scale networks can be developed. Essentially, there are three computational factors to be considered concerning the performance of algorithms for answering distance queries: construction time, index size, and query time. Much of the existing work has focused on exploring trade-offs among these computational factors [1–4, 8, 12, 14, 15, 17, 19, 22, 23, 29, 32], especially for the 2-hop cover distance labelling [3, 9]. Nonetheless, to handle large graphs, we believe that a scalable solution for answering exact distance queries needs to have the following desirable characteristics: (1) the construction time of a distance labelling is scalable with the size of a network; (2) the size of a distance labelling is minimized so as to reduce the space overhead; (3) the query time remains in the order of milliseconds, even in graphs with billions of nodes and edges.

In this work, we aim to develop a scalable solution for exact distance queries which can meet the aforementioned characteristics. Our solution is based on two ingredients: (i) a scalable labelling algorithm for constructing minimal distance labelling, and (ii) a querying framework that supports fast distance-bounded search on a sparsified graph. More specifically, we first develop a novel labelling algorithm that can scale to graphs at the billion-scale. We observed that, for a given number of landmarks, the distance entries from these landmarks to other vertices in a graph can be further minimized if the definition of 2-hop cover distance labelling is relaxed. Thus, we formulate a relaxed notion for labelling in this paper, called the *highway cover distance labelling*, and develop a simple yet scalable labelling algorithm that adds a significantly small number of distance entries into the label of each vertex. We prove that the distance labelling constructed by our labelling algorithm is minimal, and also experimentally verify that the construction process is scalable.

*Produces the permission block, and copyright information

Series ISSN: 2367-2005

10.5441/002/edbt.2019.03



(c)

Method	Ordering-dependent?	2HC-minimal?	HWC-minimal?	Parallel?
HL (ours)	no	n/a	yes	landmarks
FD [15]	no	no	no	neighbours
IS-L [12]	yes	no	no	no
PLL [3]	yes	yes	no	neighbours
HDB [16]	yes	no	no	no
HHL [2]	yes	no	no	no

Figure 1: High-level overview of the state-of-the-art methods and our proposed method (HL) for exact distance queries: (a) performance w.r.t. query time and labelling size on networks of size up to 400M, (b) scalability w.r.t. labelling construction time and network size, and (c) several important properties related to labelling methods.

Then, we formalize a querying framework for exact distance queries, which combines our proposed highway cover distance labelling with distance-bounded searches to enable fast distance computation. This querying framework is capable of balancing the trade-off between construction time, index size and query time through an offline component (i.e. the proposed highway cover distance labelling) and an online component (i.e. distance-bounded searches). The basic idea is to select a small number of highly central landmarks that allow us to efficiently compute the upper bounds of distances between all pairs of vertices using an offline distance labelling, and then conduct distance-bounded search over a sparsified graph to find exact distances efficiently. Our experimental results show that the query time of distance queries within this framework is still in millionseconds for large graphs with billions of vertices and edges.

Figure 1 summarizes the performance of the state-of-the-art methods for exact distance queries [2, 3, 8, 12, 15, 16, 21, 27], as well as our proposed method in this paper, denoted as HL. In Figure 1(a)-1(b), we can see that, labelling-based methods PLL [3], HDB [16], and HHL [2] can answer distance queries in a considerably small amount of time. However, they have very large space requirements and very long labelling construction time. On the contrary, traditional online search methods such as Dijkstra [27] and bidirectional BFS (denoted as Bi-BFS) [21] are not applicable to large-scale networks where distances need to be provided in the order of milliseconds because of their very high response time. The hybrid methods FD [12], IS-L [15] and HL (our method) combine an offline labelling and an online graph traversal technique, which can provide better trade-offs between query response time and labelling size. In Figure 1(b), we can also see that only our proposed method HL can handle networks of size 8B, and is scalable to perform distance queries on networks with billions of vertices and billions of edges.

Figure 1(c) presents a high-level overview for several important properties of labelling methods. The column Ordering dependent refers to whether a distance labelling depends on the ordering of landmarks when being constructed by a method. Only our method HL and FD are not ordering-dependent. The columns 2HC-minimal and HWC-minimal refer to whether a distance labelling constructed by a method is minimal in terms of the 2-hop cover (2HC) and highway cover (HWC) properties, respectively. PLL is 2HC-minimal, but not HWC-minimal. Our method HL is the only method that is HWC-minimal. The column Parallel refers to what kind of parallelism a method can support. FD and PLL support bit-parallelism for up to 64 neighbours of a landmark. Our method HL supports parallel computation for multiple landmarks, depending on the number of processors. Other methods did not mention any parallelism.

In summary, our contributions in this paper are as follows:

- We introduce a new labelling property, namely highway cover labelling, which relaxes the standard notion of 2-hop cover labelling. Based on this new labelling property, we propose a highly scalable labelling algorithm that can scale to construct labellings for graphs with billions of vertices and billions of edges.
- We prove that the proposed labelling algorithm can construct HWC-minimal labellings, which is independent of any ordering of landmarks. Then, due to this determinstric nature of labelling, we further develop a parallel algorithm which can run parallel BFSs from multiple landmarks to speed up labelling construction.
- We combine our novel labelling algorithm with online bounded-distance graph traversal to efficiently answer exact distance queries. This querying framework enables us to balance the trade-offs among construction time, labelling size and query time.
- We have experimentally verified the performance of our methods on 12 large-scale complex networks. The results that our methods can not only handle networks with billions of vertices, but also be up to 70 times faster in constructing labelling and save up to 90% of labelling space.

The rest of the paper is organized as follows. In Section 2, we present basic notations and definitions used in this paper. Then, we discuss a novel labelling algorithm in Section 3, formulate the querying framework in Section 4, and introduce several optimization techniques in Section 5. In Section 6 we present our experimental results and in Section 7 we discuss other works that are related to our work here. The paper is concluded in Section 8.

2 PRELIMINARIES

Let $G = (V, E)$ be a graph where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. We have $n = |V|$ and $m = |E|$. Without loss of generality, we assume that the graph G is connected and undirected in this paper. Let $V' \subseteq V$ be a subset of vertices of G. Then the induced subgraph $G[V']$ is a graph whose vertex set is V' and whose edge set consists of all of the edges in E that have both endpoints in V'. Let $N_G(v) = \{u \in V | (u, v) \in E\}$ denote a set of neighbors of a vertex $v \in V$ in G.

The *distance* between two vertices s and t in G, denoted as $d_G(s, t)$, is the length of the shortest path from s to t. We consider

$d_G(s,t) = \infty$, if there does not exist a path from s to t. For any three vertices $s, u, t \in V$, the following *triangle inequalities* are satisfied:

$$d_G(s,t) \leq d_G(s,u) + d_G(u,t) \quad (1)$$

$$d_G(s,t) \geq |d_G(s,u) - d_G(u,t)| \quad (2)$$

If u belongs to one of the shortest paths from s to t, then $d_G(s,t) = d_G(s,u) + d_G(u,t)$ holds.

Given a special subset of vertices $R \subseteq V$ of G, so-called *landmarks*, a *label* $L(v)$ for each vertex $v \in V$ can be precomputed, which is a set of *distance entries* $\{(u_1, \delta_L(u_1, v)), \dots, (u_n, \delta_L(u_n, v)\}$ where $u_i \in R$ and $\delta_L(u_i, v) = d_G(u_i, v)$ for $i = 1, \dots, n$. The set of labels $L = \{L(v)\}_{v \in V}$ is called a *distance labeling* over G. The *size* of a distance labelling L is defined as size(L)=$\Sigma_{v \in V} |L(v)|$.

Using such a *distance labeling* L, we can query the distance between any pair of vertices $s, t \in V$ in graph G as follows,

$$\text{Q}(s,t,L) = \texttt{min}\{\delta_L(u,s) + \delta_L(u,t) | (u, \delta_L(u,s)) \in L(s), (u, \delta_L(u,t)) \in L(t)\} \quad (3)$$

We define $\text{Q}(s,t,L) = \infty$, if $L(s)$ and $L(t)$ do not share any landmark. If $\text{Q}(s,t,L) = d_G(s,t)$ holds for any two vertices s and t of G, L is called a *2-hop cover distance labeling* over G [2, 9].

Given a graph G and a set of landmarks $R \subseteq V$, the *distance querying problem* is to efficiently compute the shortest path distance $d_G(s,t)$ between any two vertices s and t in G, using a distance labeling L over G in which labels may contain distance entries from landmarks in R.

3 HIGHWAY COVER LABELLING

In this section, we formulate the highway cover labelling problem and propose a novel algorithm to efficiently construct the highway cover distance labelling over graphs. Then, we provide theoretical analysis of our proposed algorithm.

3.1 Highway Cover Labelling Problem

We begin with the definitions of highway and highway cover.

Definition 3.1. (*Highway*) A highway H is a pair (R, δ_H), where R is a set of landmarks and δ_H is a *distance decoding function*, i.e. $\delta_H : R \times R \to \mathbb{N}^+$, such that for any $\{r_1, r_2\} \subseteq R$ we have $\delta_H(r_1, r_2) = d_G(r_1, r_2)$.

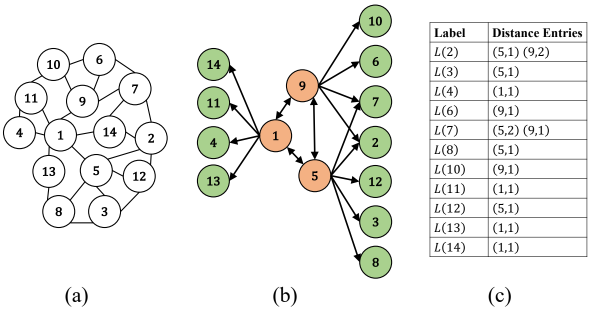
Given a landmark $r \in R$ and two vertices $s, t \in V \backslash R$ (i.e. $V \backslash R = V - R$), a *r-constrained shortest path* between s and t is a path between s and t satisfying two conditions: (1) It goes through the landmark r, and (2) It has the minimum length among all paths between s and t that go through r. We use P_{st} to denote the set of vertices in a shortest path between s and t, and P^r_{st} to denote the set of vertices in a r-constrained shortest path between s and t.

Definition 3.2. (*Highway Cover*) Let $G = (V, E)$ be a graph and $H = (R, \delta_H)$ a highway. Then for any two vertices $s, t \in V \backslash R$ and for any $r \in R$, there exist $(r_i, \delta_L(r_i, s)) \in L(s)$ and $(r_j, \delta_L(r_j, t)) \in L(t)$ such that $r_i \in P_{rs}$ and $r_j \in P_{rt}$, where r_i and r_j may equal to r.

If the label of a vertex v contains a distance entry $(r, \delta_L(r, v))$, we also say that the vertex v is *covered* by the landmark r in the distance labelling L. Intuitively, the highway cover property guarantees that, given a highway H with a set of landmarks R and $r \in R$, any r-constrained shortest path distance between two vertices s and t can be found using only the labels of these two vertices and the given highway. A distance labelling L is called a *highway cover distance labelling* if L satisfies the highway cover property.

Example 3.3. Consider the graph G depicted in Figure 2(a), the highway H has three landmarks $\{1, 5, 9\}$ as highlighted in red in Figure 2(b). Based on the graph in Figure 2(a) and the highway in Figure 2(b), we have $\langle 11, 1, 4 \rangle$ which is a shortest path between the vertices 11 and 4 constrained by the landmark 1, i.e. 1-constrained shortest path between 11 and 4. In contrast, neither of the paths $\langle 11, 10, 9, 1, 4 \rangle$ and $\langle 11, 4 \rangle$ is a 1-constrained shortest path between 11 and 4.

In Figure 2(b), the outgoing arrows from each landmark point to vertices in G that are covered by this landmark in the highway. The distance labelling in Figure 2(c) satisfies the highway cover property because for any two vertices that are not landmarks and any landmark $r \in \{1, 5, 9\}$, we can find the r-constrained shortest path distance between these two vertices using their labels and the highway.



Label	Distance Entries
$L(2)$	(5,1) (9,2)
$L(3)$	(5,1)
$L(4)$	(1,1)
$L(6)$	(9,1)
$L(7)$	(5,2) (9,1)
$L(8)$	(5,1)
$L(10)$	(9,1)
$L(11)$	(1,1)
$L(12)$	(5,1)
$L(13)$	(1,1)
$L(14)$	(1,1)

(a) (b) (c)

Figure 2: An illustration of highway cover distance labelling: (a) an example graph G, (b) a highway structure H and (c) a distance labelling that fulfills the highway cover property over (G, H).

Definition 3.4. (*Highway Cover Labelling Problem*) Given a graph G and a highway H over G, the *highway cover labelling problem* is to efficiently construct a highway cover distance labelling L.

Several choices naturally come up: (1) One is to add a distance entry for each landmark into the label of every vertex in $V - R$, as the approach proposed in [15]; (2) Another is to use the pruned landmark labelling approach [3] to add the distance entry of a landmark r into the labels of vertices in $V - R$ if the landmark has not been pruned during a BFS rooted at r; (3) Alternatively, we can also extend the pruned landmark labelling approach to construct the highway cover labeling by replacing the 2-hop cover pruning condition with the one required by the highway cover as defined in Definition 3.2 at each step of checking possible labels to be pruned.

In all these cases, the labelling construction process would not be scalable nor be suitable for large-scale complex networks with billions of vertices and edges. Moreover, these approaches would potentially lead to the construction of distance labellings with different sizes. A question arising naturally is how to construct a minimal highway cover distance labelling without redundant labels? In a nutshell, it is a challenging task to construct a highway cover distance labelling that can scale to very large networks, ideally in linear time, but also with the minimal labelling size.

3.2 A Novel Algorithm

We propose a novel algorithm for solving the highway cover labelling problem, which can construct labellings in linear time.

The key idea of our algorithm is to construct a *label* $L(v)$ for vertex $v \in V\backslash R$ such that the distance entry $(r_i, \delta_L(r_i, v))$ of each landmark $r_i \in R$ is only added into the label $L(v)$ iff there does not exist any other landmark that appears in the shortest path between r_i and v, i.e. $P_{r_i v} \cap R = \{r_i\}$. In other words, if there is another landmark $r \in R$ and r_i is in the shortest path between r and v, then $(r_i, \delta_L(r_i, v))$ is added into $L(v)$ iff r_i is the "closest" landmark to v. To compute such labels efficiently, we conduct a breadth-first search from every landmark $r_i \in R$ and add distance entries into labels of vertices that do not have any other landmark in their shortest paths from r_i.

Example 3.5. Consider vertex 7 in Figure 2(c), the label $L(7)$ contains the distance entries of landmarks $\{5, 9\}$, but no distance entry of landmark 1. This is because 5 and 9 are the closest landmarks to vertex 7 in the shortest paths $\langle 5, 7\rangle$ and $\langle 9, 7\rangle$, respectively. However, for either of two shortest paths $\langle 1, 9, 7\rangle$ and $\langle 1, 5, 7\rangle$ between 1 and 7, there is another landmark (i.e. 5 or 9) that is closer to 7 compared with 1 in these shortest paths. Thus the distance entry of landmark 1 is not added into $L(7)$.

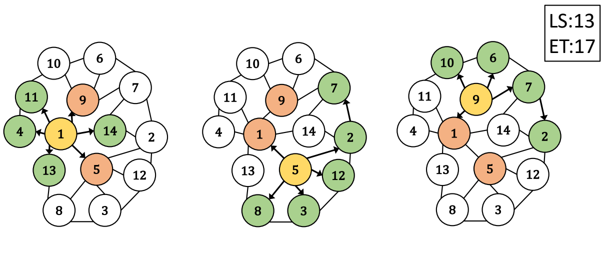
Our highway cover labelling approach is described in Algorithm 1. Given a graph G and a highway H over G, we start with an empty *highway cover distance labelling* L, where $L(v) = \emptyset$ for every $v \in V\backslash R$. Then, for each landmark $r_i \in R$, we compute the corresponding distance entries as follows. We use two queues Q_{label} and Q_{prune} to process vertices to be labeled or pruned at each level of a breadth-first search (BFS) tree, respectively. We start by processing vertices in Q_{label}. For each vertex $u \in Q_{label}$ at depth n, we examine the children of u at depth $n + 1$ that are unvisited. For each unvisited child vertex $v \in N_G(u)$ at depth $n + 1$, if $v \in R$ then we prune v, i.e., we do not add a distance entry of the current landmark r_i into $L(v)$ and we also enqueue v to the pruned queue Q_{prune} (Line 11). Otherwise, we add $(r_i, \delta_{BFS}(r_i, v))$ to the *label* of v, i.e., we add it into $L(v)$ and we also enqueue v to the labeled queue Q_{label} (Lines 13-14). Here, $\delta_{BFS}(r_i, v)$ refers to BFS decoded distance from root r_i to v. Then we process the pruned vertices in Q_{prune}. These vertices are either landmarks or have landmarks in their shortest paths from r_i, and thus do not need to be labeled. Therefore, for each vertex $v \in Q_{prune}$ at depth n, we enqueue all unvisited children of v at depth $n + 1$ to the pruned queue Q_{prune}. We keep processing these two queues, one after the other, until Q_{label} is empty.

Example 3.6. We illustrate how our algorithm conducts pruned BFSs in Figure 3. The pruned BFS from landmark 1 is depicted in Figure 3(a), which labels only four vertices $\{4, 11, 13, 14\}$ because the other vertices are either landmarks or contain other landmarks in their shortest paths to landmark 1. Similarly, in the pruned BFS from landmark 5 depicted in Figure 3(b), only vertices $\{7, 2, 12, 3, 8\}$ are labelled, and none of the vertices 4, 11, 13 and 14 is labelled because of the presence of landmark 1 in their shortest paths to landmark 5. Indeed, we can get the distance between landmark 5 to these vertices by using the highway, i.e. $\delta_H(5, 1)$, and distance entries in their labels to landmark 1. The pruned BFS from landmark 9 is depicted in Figure 3(c), which works in a similar fashion.

Note that, although a highway H is given in Algorithm 1, we can indeed compute the distances δ_H for a given set of landmarks R along with Algorithm 1.

Algorithm 1: Constructing the highway cover labelling L

```
   Input: $G = (V.E)$, $H = (R, \delta_H)$
   Output: $L$
1  $L(v) \leftarrow \emptyset, \forall v \in V\backslash R$
2  foreach $r_i \in R$ do
3      $Q_{label} \leftarrow \emptyset$
4      $Q_{prune} \leftarrow \emptyset$
5      $n \leftarrow 0$
6      Enqueue $r_i$ to $Q_{label}$ and set $r_i$ as the root of BFS
7      while $Q_{label}$ is not empty do
8          foreach $u \in Q_{label}$ at depth $n$ do
9              foreach unvisited child $v$ of $u$ at depth $n + 1$ do
10                 if $v$ is a landmark then
11                     Enqueue $v$ to $Q_{prune}$
12                 else
13                     Enqueue $v$ to $Q_{label}$
14                     Add $\{(r_i, \delta_{BFS}(r_i, v))\}$ to $L(v)$
15                 end
16             end
17         end
18         $n \leftarrow n + 1$
19         foreach $v \in Q_{prune}$ at depth $n$ do
20             Enqueue unvisited children of $v$ at depth $n + 1$
                 to $Q_{prune}$
21         end
22     end
23 end
24 return $L$
```



(a) (b) (c)

Figure 3: An illustration of the highway cover labelling algorithm: (a), (b) and (c) describe the pruned BFSs that are rooted at the landmarks 1, 5 and 9, respectively, where yellow vertices denote roots, green vertices denote those being labeled, red vertices denote landmarks, and white vertices are not labelled. LS and ET at the top right corner denote the labelling size and the number of edges traversed during the pruned BFSs, respectively.

3.3 Correctness

Here we prove the correctness of our labelling algorithm.

Lemma 3.7. *In Algorithm 1, for each pruned BFS rooted at $r_i \in R$, $(r_i, \delta_L(r_i, v))$ is added into the label of a vertex $v \in V\backslash R$ iff there is no any other landmark appearing in the shortest path between r_i and v, i.e., $P_{r_i v} \cap R = \{r_i\}$.*

Proof. Suppose that Algorithm 1 is conducting a pruned BFS rooted at r_i and v is an unvisited child of another vertex u in Q_{label} (start from $Q_{label} = \{r_i\}$) (Lines 6-9). If $v \in R$ (Line 10), then we have $(P_{r_i w} \cap R) \supseteq \{r_i, v\}$ (Lines 11, 19-21), $(r_i, \delta_L(r_i, w))$ cannot be added into the label of any child w of v, i.e., put w into Q_{prune}. Otherwise, by $v \notin R$ and v is an unvisited child of a vertex u in Q_{label} (Lines 8-9), we know that $P_{r_i v} \cap R = \{r_i\}$ and thus $(r_i, \delta_L(r_i, v))$ is added into $L(v)$ (lines 12-14). □

Then, by Lemma 3.7, we have the following corollary.

Corollary 3.8. *Let $r \in R$ be a landmark, $v \in V\backslash R$ a vertex, and L a distance labelling constructed by Algorithm 1, if $(r, \delta_L(r, v)) \notin L(v)$, then there must exist a landmark r_j such that $(r_j, \delta_L(r_j, v)) \in L(v)$ and $d_G(r, v) = \delta_L(r_j, v) + \delta_H(r, r_j)$.*

Theorem 3.9. *The highway cover distance labelling L over (G, H) constructed using Algorithm 1 satisfies the highway cover property over (G, H).*

Proof. To prove that, for any two vertices $s, t \in V\backslash R$ and for any $r \in R$, there exist $(r_i, \delta_L(r_i, s)) \in L(s)$ and $(r_j, \delta_L(r_j, t)) \in L(t)$ such that $r_i \in P_{rs}$ and $r_j \in P_{rt}$, we consider the following 4 cases: (1) If $r \in L(s)$ and $r \in L(t)$, then $r = r_i = r_j$. (2) If $r \in L(s)$ and $r \notin L(t)$, then $r_i = r$ and by Lemma 3.8, there exists another landmark r_j such that r_j is in the shortest path between t and r and $(r_j, \delta_L(r_j, t)) \in L(t)$. (3) If $r \notin L(s)$ and $r \in L(t)$, then similarly we have $r_j = r$, and by Lemma 3.8, there exists another landmark r_i such that r_i is in the shortest path between s and r and $(r_i, \delta_L(r_i, s)) \in L(s)$. (4) If $r \notin L(s)$ and $r \notin L(t)$, then by Lemma 3.8 there exist another two landmarks r_i and r_j such that r_i is in the shortest path between s and r and $(r_i, \delta_L(r_i, s)) \in L(s)$, and r_j is in the shortest path between t and r and $(r_j, \delta_L(r_j, t)) \in L(t)$. The proof is done. □

3.4 Order Independence

In previous studies [1–3, 9], given a graph G, a distance labelling algorithm builds a unique canonical distance labelling subject to a labelling order (i.e., the order of landmarks used for constructing a distance labelling). It has been well known that such a labelling order is decisive in determining the size of the constructed distance labelling [24]. For the same set of landmarks, when using different labelling orders, the sizes of the constructed distance labelling may vary significantly.

The following example shows how different labelling orders in the pruned landmark labelling approach [3] can lead to the distance labelling of different sizes.

Example 3.10. In Figure 4, the size of the distance labelling constructed using the labelling order $\langle 1, 5, 9\rangle$ in Figure 4(a)-4(c) is different from the size of the distance labelling constructed using the labelling order $\langle 9, 5, 1\rangle$ in Figure 4(d)-4(f). In both cases, the first BFS adds the distance entry of the current landmark into all the vertices in the graph. Then, the following BFSs check each visited vertex whether the shortest path distance between the current landmark and the visited vertex can be computed via the 2-hop cover property based on their labels added by the previous BFSs. A distance entry is only added into the label of a vertex if the shortest path distance cannot be computed by applying the 2-hop cover over the existing labels. Thus, the choice of the labelling order could affect the size of labels significantly. Take the vertex 11 for example, its label contains only one distance entry (1, 1) using the labelling order depicted in Figure 4(a)-4(c), but contains three distance entries (1.1), (5, 2), and (9, 2) when the labelling order depicted in Figure 4(d)-4(f) is used.

Unlike all previous approaches taken with distance labelling, our highway cover labelling algorithm is order-invariant. That is, regardless of the labelling order, the distance labellings constructed by our algorithm using different labelling orders over the same set of landmarks always have the same size. In fact, we can show that our algorithm has the following stronger property: the distance labelling constructed using our algorithm is deterministic (i.e., the same label for each vertex) for a given set of landmarks.

Lemma 3.11. *Let $G = (V, E)$ be a graph and $H = (R, \delta_H)$ a highway over G. For any two different labelling orders over R, the highway cover distance labellings L_1 and L_2 over (G, H) constructed by these two different labelling orders using Algorithm 1 satisfy $L_1(v) = L_2(v)$ for every $v \in V\backslash R$.*

Proof. Let O_{L_1} and O_{L_2} be two different labelling orders over R. For any landmark r in O_{L_1} and O_{L_2}, Algorithm 1 generates exactly the same pruned BFS tree. This implies that, for each vertex $v \in V\backslash R$, either the same distance entry $(r, \delta_{BFS}(r, v))$ is added into $L_1(v)$ and $L_2(v)$, or no distance entry is added to $L_1(v)$ and $L_2(v)$. Thus, Algorithm 1 satisfy $L_1(v) = L_2(v)$ for every $v \in V\backslash R$. □

3.5 Minimality

Here we discuss the question of minimality, i.e., whether the highway cover distance labelling constructed by our algorithm is always minimal in terms of the labelling size. We first prove the following theorem.

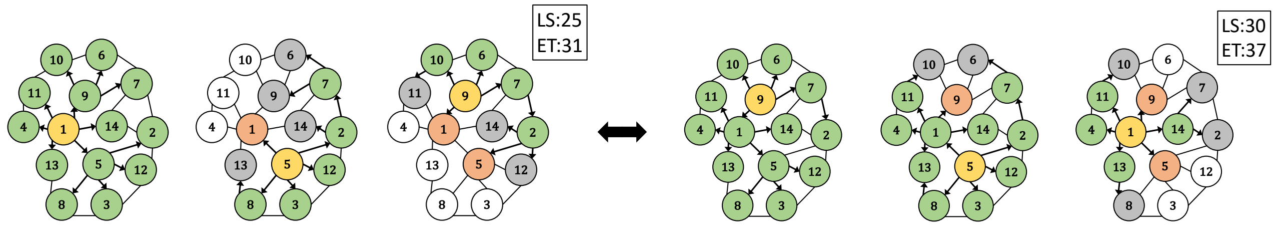
Theorem 3.12. *The highway cover distance labelling L over (G, H) constructed using Algorithm 1 is minimal, i.e., for any highway cover distance labelling L' over (G, H), $size(L') \geq size(L)$ must hold.*

Proof. We prove this by contradiction. Let us assume that there is a highway cover distance labelling L' with $size(L') < size(L)$. Then, this would imply that there must exist a vertex $v \in V\backslash R$ and a landmark $r \in R$ such that $r \in L(v)$ and $r \notin L'(v)$. By Lemma 3.7 and $r \in L(v)$, we know that there is no any other landmark in R that is in the shortest path between r and v. However, by the definition of the highway cover property (i.e. Definition 3.2) and $r \notin L'(v)$, we also know that there must exist another landmark $(r_i, \delta_L(r_i, v)) \in L(v)$ and $r_i \in P_{rv}$, which contradicts with the previous conclusion that there is no any other landmark in the shortest path between r and v. Thus, $size(L') \geq size(L)$ must hold for any highway cover distance labelling L'. □

The state-of-the-art approaches for distance labelling is primarily based on the idea of 2-hop cover [1, 3, 12]. One may ask the question: how is the highway cover labelling different from the 2-hop cover labelling, such as the pruned landmark labelling [3]? It is easy to verify the following lemma that each pruned landmark labelling satisfies the highway cover property for the same set of landmarks.

Lemma 3.13. *Let L be a pruned landmark labelling over graph G constructed using a set of landmarks R. Then L also satisfies the highway cover property over (G, H) where $H = (R, \delta_H)$.*

As the pruned landmark labelling algorithm [3] prunes labels based on the 2-hop cover property, but our highway cover labeling algorithm prunes labels based on the property described in Lemma 3.7, by Theorem 3.12, we also have the following corollary, stating that, for the same set of landmarks, the size of the



(a) (b) (c) (d) (e) (f)

Figure 4: An illustration of the pruned landmark labelling algorithm [3]: (a)-(c) show an example of constructing labels through pruned BFSs from three landmarks in the labelling order $\langle 1, 5, 9\rangle$; (d)-(f) show an example of constructing labels using the same three landmarks but in a different labelling order $\langle 9, 5, 1\rangle$. Yellow vertices denote landmarks that are the roots of pruned BFSs, green vertices denote those being labeled, grey vertices denote vertices being visited but pruned, and red vertices denote landmarks which have already been visited.

highway cover labelling is always smaller than the size of any pruned landmark labelling.

Corollary 3.14. *For a highway cover distance labelling L_1 produced by Algorithm 1 over (G, H), where $H = (R, \delta_H)$, and a pruned landmark labelling L_2 over G using any labelling order over R, we always have $|L_1| \leq |L_2|$.*

Example 3.15. Figure 4 shows the labelling size (LS) of the pruned landmark labelling at the top right corner, which is constructed using two different orderings. The first ordering $\langle 1, 5, 9\rangle$ labels 25 vertices whereas the second ordering $\langle 9, 5, 1\rangle$ labels 30 vertices. On the other hand, the LS of the highway cover distance labelling is 13 as shown in Figure 3. Note that the LS of the highway cover distance labelling does not change, irrespective of ordering. Since the highway cover distance labelling constructed by our algorithm is always minimal, the LS of the highway cover distance labelling in Figure 3 is much smaller than the LS of either pruned landmark labelling in Figure 4.

4 BOUNDED DISTANCE QUERYING

In this section, we describe a bounded distance querying framework that allows us to efficiently compute exact shortest-path distances between two arbitrary vertices in a massive network.

4.1 Querying Framework

We start with presenting a high-level overview of our querying framework. To compute the shortest path distance between two vertices s and t in graph G, our querying framework proceeds in two steps: (1) an upper bound of the shortest path distance between s to t is computed using the highway cover distance labelling; (2) the exact shortest path distance between s to t is computed using a distance-bounded shortest-path search over a sparsified graph from G.

Given a graph G and a highway $H = (R, \delta_H)$ over G, we can precompute a highway cover distance labelling L using the landmarks in R, which enables us to efficiently compute the length of any r-constrained shortest path between two vertices in $V \backslash R$. The length of such a r-constrained shortest path must be greater than or equal to the exact shortest path distance between these two vertices and can thus serve as an upper bound in Step (1). On the other hand, since the length of such a r-constrained shortest path between two vertices in $V \backslash R$ can always be efficiently computed by the highway cover distance labelling L, the distance-bounded shortest-path search only needs to be conducted over a sparsified graph G' by removing all landmarks in R from G, i.e. $G' = G[V \backslash R]$.

More precisely, we define the bounded distance querying problem in the following.

Definition 4.1. (*Bounded Distance Querying Problem*) Given a sparsified graph $G' = (V', E')$, a pair of vertices $\{s, t\} \in V'$, and an upper (distance) bound $d^{\top}_{st}$, the *bounded distance querying problem* is to efficiently compute the shortest path distance d_{st} between s and t over G' under the upper bound $d^{\top}_{st}$ such that,

$$d_{st} = \begin{cases} d_{G'}(s,t), & \text{if } d_{G'}(s,t) \leq d^{\top}_{st} \\ d^{\top}_{st}, & \text{otherwise} \end{cases}$$

In the following, we discuss the two steps of this framework in detail.

4.2 Computing Upper Bounds

Given any two vertices s and t, we can use a *highway cover distance labelling* L to compute an upper bound $d^{\top}_{st}$ for the shortest path distance between s and t as follows,

$$\begin{aligned} d^{\top}_{st} = \mathtt{min}\{\delta_L(r_i, s) + \delta_H(r_i, r_j) + \delta_L(r_j, t) | \\ (r_i, \delta_L(r_i, s)) \in L(s), \\ (r_j, \delta_L(r_j, t)) \in L(t)\} \end{aligned} \quad (4)$$

This corresponds to the length of a shortest path from s to t passing through landmarks r_i and r_j, where $\delta_L(r_i, s)$ is the shortest path distance from r_i to s in $L(s)$, $\delta_H(r_i, r_j)$ is the shortest path distance from r_i to r_j through highway H, and $\delta_L(r_j, t)$ is the shortest path distance from r_j to t in $L(t)$.

Example 4.2. Consider the graph in Figure 2(a), we may use the labels $L(2)$ and $L(11)$ to compute the upper bound for the shortest path distance between two vertices 2 and 11. There are two cases: (1) for the path $\langle 2, 5, 1, 11\rangle$ that goes through landmarks 5 and 1, we have $\delta_L(5, 2) + \delta_H(5, 1) + \delta_L(1, 11) = 1 + 1 + 1 = 3$, and (2) for the path $\langle 2, 9, 1, 11\rangle$ that goes through landmarks 9 and 1, we have $\delta_L(9, 2) + \delta_H(9, 1) + \delta_L(1, 11) = 2 + 1 + 1 = 4$. Thus, we take the minimum of these two distances as the upper bound, which is 3 in this case.

4.3 Distance-Bounded Shortest Path Search

We conduct a bidirectional search on the sparsified graph $G[V \backslash R]$ which is bounded by the upper bound $d^{\top}_{st}$ from the highway cover distance labelling. For a pair of vertices $\{s, t\} \subseteq V \backslash R$, we run breadth-first search algorithm from s and t, simultaneously [15]. Algorithm 2 shows the pseudo-code of our distance-bounded shortest path search algorithm. We use two sets of vertices $\mathcal{P}_s$ and $\mathcal{P}_t$ to keep track of visited vertices from s and t. We use two queues Q_s and Q_t to conduct both a forward search from s and a reverse search from t. Furthermore, we use two integers d_s and d_t to maintain the current distances from s and t, respectively.

During initialization, we set $\mathcal{P}_s$ and $\mathcal{P}_t$ to $\{s\}$ and $\{t\}$, and enqueue s and t into Q_s and Q_t, respectively. In each iteration, we increment d_s or d_t and expand $\mathcal{P}_s$ or $\mathcal{P}_t$ by running either a forward search (FS) or a reverse search (RS) as long as $\mathcal{P}_s$ and $\mathcal{P}_t$ have no any common vertex or $d_s + d_t$ is equal to the upper bound $d^{\top}_{st}$, and Q_s and Q_t are not empty. In the forward search from s, we examine the neighbors $N_{G[V\backslash R]}(v)$ of each vertex $v \in Q_s$. Suppose we are visiting a vertex $w \in N_{G[V\backslash R]}(v)$, if w is included in vertex set $\mathcal{P}_t$, then it means that we find a shortest path to vertex t of length $d_s + 1 + d_t$, because the reverse search from t had already visited w with distance d_t. At this stage, we return $d_s + 1 + d_t$ as the answer since we already know $d_s + d_t + 1 \leq d_G(s,t) \leq d^{\top}_{st}$. Otherwise, we add vertex w to $\mathcal{P}_s$ and enqueue w into a new queue Q_{new}. When we could not find the shortest distance in the iteration, we replace Q_s with Q_{new} and increase d_s by 1, and check if $d_s + d_t = d^{\top}_{st}$. If it holds, then we return $d^{\top}_{st}$ since $d^{\top}_{st} \leq d_G(s,t) \leq d_s + d_t + 1$.

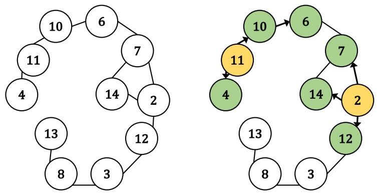
Algorithm 2: Distance-Bounded Shortest Path Search

```
Input: $G[V\backslash R]$, $s$, $t$, $d^{\top}_{st}$
Output: $d_{G[V\backslash R]}(s,t)$
$\mathcal{P}_s \leftarrow \{s\}, \mathcal{P}_t \leftarrow \{t\}, d_s \leftarrow 0, d_t \leftarrow 0$
Enqueue $s$ to $Q_s$, $t$ to $Q_t$
while $Q_s$ and $Q_t$ are not empty do
    if $|\mathcal{P}_s| \leq |\mathcal{P}_t|$ then
        $found \leftarrow \mathsf{FS}(Q_s)$
    else
        $found \leftarrow \mathsf{RS}(Q_t)$
    end
    if $found = true$ then
        return $d_s + 1 + d_t$
    else if $d_s + d_t = d^{\top}_{st}$ then
        return $d^{\top}_{st}$
    end
end
return $\infty$
```

Example 4.3. In Figure 5(b), the upper distance bound between vertices 2 and 11 is 3, as computed in Example 4.2. Suppose that we run BFSs from vertices 2 and 11 respectively. First, a forward search from 2 enqueues its neighbors 7, 12 and 14 into Q_2 and increases d_2 by 1. Then a reverse search from 11 enqueues 4 and 10 into Q_{11} and also sets d_{11} to 1. At this stage, we have not found any common vertex between Q_2 and Q_{11}, and $d_2 + d_{11} = 2$ which is less the upper bound 3. Therefore, we continue to start a search from the vertices in Q_{11}, which enqueues 5 into Q_{11} and increments d_{11} to 2. Now, we have $d_2 + d_{11} == 3$ reaching the upper bound, hence we do not need to continue our search.

4.4 Correctness

The correctness of our querying framework can be proven based on the following two lemmas. More specifically, Lemma 4.4 can be derived by the highway cover property and the definition of $d^{\top}_{st}$. Lemma 4.5 can also be proven by the property of shortest path and the definition of the sparsified graph $G[V\backslash R]$.



(a) (b)

Figure 5: An illustration of the distance-bounded shortest path search algorithm [15]: (a) shows the sparsified graph after removing three landmarks $\{1,5,9\}$ from the graph in Figure 2(a); (b) shows an example of computing the bounded distance between vertices 2 and 11 as highlighted in yellow, and green vertices denote the visited vertices in the forward and reverse searches.

Lemma 4.4. *For a highway cover distance labelling L over (G, H), we have $d^{\top}_{st} \geq d_G(s,t)$ for any two vertices s and t of G, where $d^{\top}_{st}$ is computed using L and H.*

Lemma 4.5. *For any two vertices $\{s,t\} \subseteq V\backslash R$, if there is a shortest path between s and t in G that does not include any vertex in R, then $d_G(s,t) = d_{G[V\backslash R]}(s,t)$ holds.*

Thus, the following theorem holds:

Theorem 4.6. *Let $G = (V, E)$ be a graph, H a highway over G and L a highway cover distance labelling. Then, for any two vertices $\{s,t\} \subseteq V$, the querying framework over (G, H, L) yields $d_G(s,t)$.*

Proof. We consider two cases: (1) P_{st} contains at least one landmark. In this case, By Lemma 4.4 and the definition of the highway cover property, we have $d^{\top}_{st} = d_G(s,t)$. (2) P_{st} does not contain any landmark. By Lemma 4.5, we have $d_{G[V\backslash R]}(s,t) = d_G(s,t)$. □

5 OPTIMIZATION TECHNIQUES

In this section, we discuss optimization techniques for label construction, label compression, and query processing.

5.1 Label Construction

A technique called Bit-Parallelism (BP) has been previously used in several methods [3, 15] to speed up the label construction process. The key idea of BP is to perform BFSs from a given landmark r and up to 64 of its neighbors simultaneously, and encode the relative distances (-1, 0 or 1) of these neighbors w.r.t. the shortest paths between r and each vertex v into a 64-bit unsigned integer. In the work [3], BP was applied to construct bit-parallel labels from initial vertices without pruning, which aimed to leverage the information from these bit-parallel labels to cover more shortest paths between vertices. Then, both bit-parallel labels and normal labels are constructed in the pruned BFSs. The work in [15] also used BP to construct thousands of bit-parallel shortest-path trees (SPTs) because it is very costly to construct thousands of normal SPTs in memory owing to their prohibitively large space requirements and very long construction time.

In our work, we develop a simple yet rigorous parallel algorithm (HL-P) which can run parallel BFSs from multiple landmarks (depending on the number of processors) to construct

Table 1: Datasets, where $|G|$ denotes the size of a graph G with each edge appearing in the forward and reverse adjacency lists and being represented by 8 bytes.

Dataset	Network	Type	n	m	m/n	avg. deg	max. deg	$\|G\|$	Sources
Skitter	computer	undirected	1.7M	11M	6.5	13.081	35455	85 MB	[28]
Flickr	social	undirected	1.7M	16M	9.1	18.133	27224	119 MB	[28]
Hollywood	social	undirected	1.1M	114M	49.5	98.913	11467	430 MB	[6, 7]
Orkut	social	undirected	3.1M	117M	38.1	76.281	33313	894 MB	[28]
enwiki2013	social	directed	4.2M	101M	21.9	43.746	432260	701 MB	[6, 7]
LiveJournal	social	directed	4.8M	69M	8.8	17.679	20333	327 MB	[28]
Indochina	web	directed	7.4M	194M	20.4	40.725	256425	1.1 GB	[6, 7]
it2004	web	directed	41M	1.2B	24.9	49.768	1326744	7.7 GB	[6, 7]
Twitter	social	directed	42M	1.5B	28.9	57.741	2997487	9.0 GB	[6, 7]
Friendster	social	undirected	66M	1.8B	22.5	45.041	4006	13 GB	[18]
uk2007	web	directed	106M	3.7B	31.4	62.772	979738	25 GB	[6, 7]
ClueWeb09	computer	directed	2B	8B	5.98	11.959	599981958	55 GB	[25]

labelling in an extremely efficient way for massive networks, with much less time as will be demonstrated in our experiments.

5.2 Label Compression

The choice of the data structure for labels may significantly affect the performance of index size and memory usage. As noted in [19], some works [2, 10] did not elaborate on what data structure they have used for representing labels. Nonetheless, for the works that are most relevant to ours, such as FD [15] and PLL [3], they used 32-bit integers to represent vertices and 8-bit integers to represent distances for normal labels. In addition to this, they also used 64-bits to encode the distances from a landmark to up to 64 of its neighbors in their shortest paths to other vertices. Since our approach only selects a very small number of landmarks to construct the highway cover labelling (usually no more than 100 landmarks), we may use 8 bits to represent landmarks and another 8 bits to store distances for labels. In order to fairly compare methods from different aspects, we have implemented our methods using both 32 bits and 8 bits for representing vertices in labels. However, different from the BP technique that uses 64-bits to encode the distance information of up to 64 neighbours of a landmark, our parallel algorithm (HL-P) does not use a different data structure for labels constructed in parallel BFSs.

5.3 Query Processing

We show that computing the upper bound $d^{\top}_{st}$ can be optimized based on the observation, captured by the following lemma.

Lemma 5.1. *For a highway cover distance labelling L over (G, H), where $G = (V, E)$ and $H = (R, \delta_H)$, and any $\{s, t\} \subseteq V \backslash R$, if a landmark r appears in both $L(s)$ and $L(t)$, then $\delta_L(r, s) + \delta_L(r, t) \leq \delta_L(r, s) + \delta_H(r, r') + \delta_L(r', t)$ holds for any other $r' \in R$.*

Proof. By the definition of the highway cover property, we know that r is not in the shortest path between r' and t. Then by triangle inequality in Equation 1, this lemma can be proven. □

Thus, in order to efficiently compute the upper bound $d^{\top}_{st}$, for any landmarks that appear in both $L(s)$ and $L(t)$, we compute the r-constrained shortest path distance between s and t using Equation 2, while for a landmark r' that only appear in one of $L(s)$ and $L(t)$, we use Equation 4 to calculate the r'-constrained shortest path distance between s and t. This would lead to more efficient computations for queries when the landmarks appear in both labels of two vertices.

6 EXPERIMENTS

To compare the proposed method with baseline approaches, we have implemented our method in C++11 using STL libraries and compiled using gcc 5.5.0 with the -O3 option. We performed all the experiments using a single thread on a Linux server (having 64 AMD Opteron(tm) Processors 6376 with 2.30GHz and 512GB of main memory) for sequential version of the proposed method and up to 64 threads for parallel version of the proposed method.

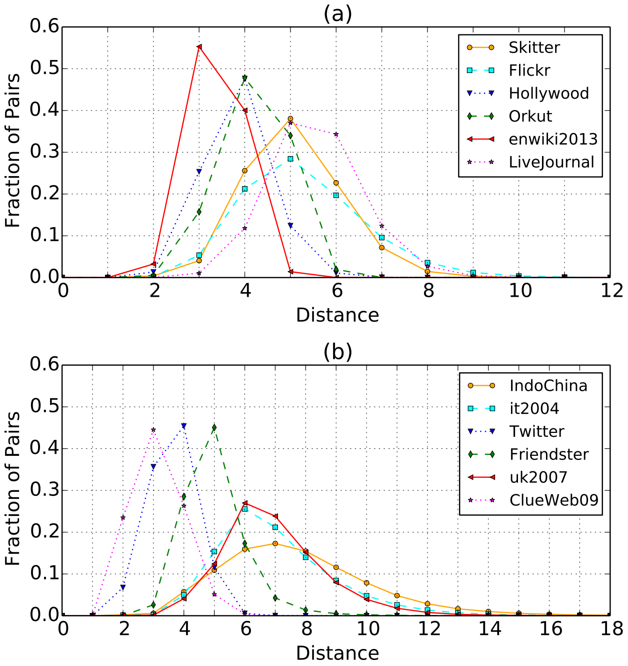


Figure 6: Distance distribution of 100,000 random pairs of vertices on all the datasets.

6.1 Datasets

In our experiments, we used 12 large-scale real-world complex networks, which are detailed in Table 1. These networks have vertices and edges ranging from millions to billions. Among them, the largest network is ClueWeb09 which has 2 billions of vertices and 8 billions of edges. We included this network in our experiments for the purpose of evaluating the robustness and

Table 2: Comparison of construction times and query times between our methods, i.e., HL-P and HP, and the state-of-the-art methods, where CT denotes the CPU clock time in seconds for labelling construction, QT denotes the average query time in milliseconds, and ALS denotes the average number of entries per label.

Dataset	CT[s]					QT[ms]					ALS			
	HL-P	HL	FD	PLL	IS-L	HL	FD	PLL	IS-L	Bi-BFS	HL	FD	PLL	IS-L
Skitter	2	13	30	638	1042	0.067	0.043	0.008	3.556	3.504	12	20+64	138+50	51
Flickr	2	14	41	1330	8359	0.015	0.028	0.01	33.760	4.155	10	20+64	290+50	50
Hollywood	3	17	107	31855	DNF	0.047	0.075	0.051	-	6.956	12	20+64	2206+50	-
Orkut	10	62	366	DNF	DNF	0.224	0.251	-	-	21.086	11	20+64	-	-
enwiki2013	9	77	308	22080	DNF	0.190	0.131	0.027	-	19.423	10	20+64	471+50	-
LiveJournal	9	77	166	DNF	20583	0.088	0.111	-	56.847	17.264	13	20+64	-	69
Indochina	8	50	144	9456	DNF	1.905	1.803	0.02	-	9.734	5	20+64	441+50	-
it2004	66	304	1623	DNF	DNF	2.684	2.118	-	-	92.187	10	20+64	-	-
Twitter	133	1380	1838	DNF	DNF	1.424	0.432	-	-	426.949	14	20+64	-	-
Friendster	135	2229	9661	DNF	DNF	1.091	1.435	-	-	534.576	19	20+64	-	-
uk2007	110	1124	6201	DNF	DNF	11.841	18.979	-	-	355.688	8	20+64	-	-
ClueWeb09	4236	28124	DNF	DNF	DNF	0.309	-	-	-	-	2	-	-	-

Table 3: Comparison of labelling sizes between our methods, i.e., HL(8) and HL, and the state-of-the-art methods.

Dataset	HL(8)	HL	FD	PLL	IS-L
Skitter	42MB	102MB	202MB	2.5GB	507MB
Flickr	34MB	81MB	178MB	3.7GB	679MB
Hollywood	28MB	67MB	293MB	13GB	-
Orkut	70MB	170MB	756MB	-	-
enwiki2013	83MB	200MB	743MB	12GB	-
LiveJournal	123MB	299MB	778MB	-	3.8GB
Indochina	81MB	192MB	999MB	21GB	-
it2004	855MB	2GB	5.6GB	-	-
Twitter	1.2GB	2.8GB	4.8GB	-	-
Friendster	2.5GB	5.2GB	11.8GB	-	-
uk2007	1.8GB	4.3GB	14.1GB	-	-
ClueWeb09	4.7GB	9GB	-	-	-

scalability of the proposed method. In previous works, the largest dataset that has been reported is uk2007 which has only around 100 millions of vertices and 3.7 billions of edges. For all these networks, we treated them as undirected and unweighted graphs.

To investigate the query time of finding the distance between two vertices, we randomly sampled 100,000 pairs of vertices from all pairs of vertices in each network, i.e., $V \times V$. The distance distribution of these 100,000 randomly sampled pairs of vertices are shown in Figure 6(a)-6(b), from which we can confirm that most of pairs of vertices in these networks have a small distance ranging from 2 to 8.

6.2 Baseline Methods

We compared our proposed method with three state-of-the-art methods. Two of these methods, namely fully dynamic (FD) [15] and IS-L [12], combine a distance labelling algorithm with a graph traversal algorithm for distance queries on complex networks. The third one is pruned landmark labelling (PLL) [3] which is completely based on distance labelling to answer distance queries. Besides these, there are a number of other methods for answering distance queries, such as HDB [16], RXL and CRXL [10], HCL [17], HHL [2] and TEDI [32]. However, since the experimental results of the previous works [3, 15] have shown that FD outperforms HDB, RXL and CRXL, and PLL outperforms HCL, HHL and TEDI, we omit the comparison with these methods.

In our experiments, the implementations of the baseline methods FD, IS-L and PLL were provided by their authors, which were all implemented in C++. We used the same parametric settings for running these methods as suggested by their authors. For instance, the number of landmarks is chosen to 20 for FD [15], the number of bit-parallel BFSs is set to 50 for PLL [3], and k is 6 for graphs larger than 1 million vertices for IS-L [12].

6.3 Comparison with Baseline Methods

To evaluate the performance of our proposed approach, we compared our approach with the baseline methods in terms of the construction time of labelling, the size of labelling, and querying time. The experimental results are presented in Tables 2 and 3, where DNF denotes that a method did not finish in one day or ran out of memory. In order to make a consistent comparison with the baseline methods [3, 12, 15], we chose top 20 vertices as landmarks after sorting based on decreasing order of their degrees, and also used 32-bit integers to represent vertices and 8-bit integers to represent distances.

6.3.1 Construction Time. As shown in Table 2, our proposed method (HL) has successfully constructed the distance labelling on all the datasets for a significantly less amount of time than the state-of-the-art methods. As compared to FD, our method is on average 5 times faster and have results on all the datasets. In contrast to this, FD failed to construct labelling for the largest dataset ClueWeb09. PLL failed for 7 out of 12 datasets, including the datasets Orkut and LiveJournal which have less than 120 millions of edges, due to its prohibitively high preprocessing time and memory requirements for building labelling. IS-L failed to construct labelling for all the datasets that have edges more than 100 million due to its very high cost for computing independent sets on massive networks, i.e. failed for 9 out of 12 datasets. We can also see from Table 2 that the parallel version of our method (HL-P) is much faster than the sequential version (HL). Compared with FD, HL-P is more than 50-70 times faster for the two large datasets Friendster and uk2007. This confirms that our method can construct labelling very efficiently and is scalable on large networks with billions of vertices and edges.

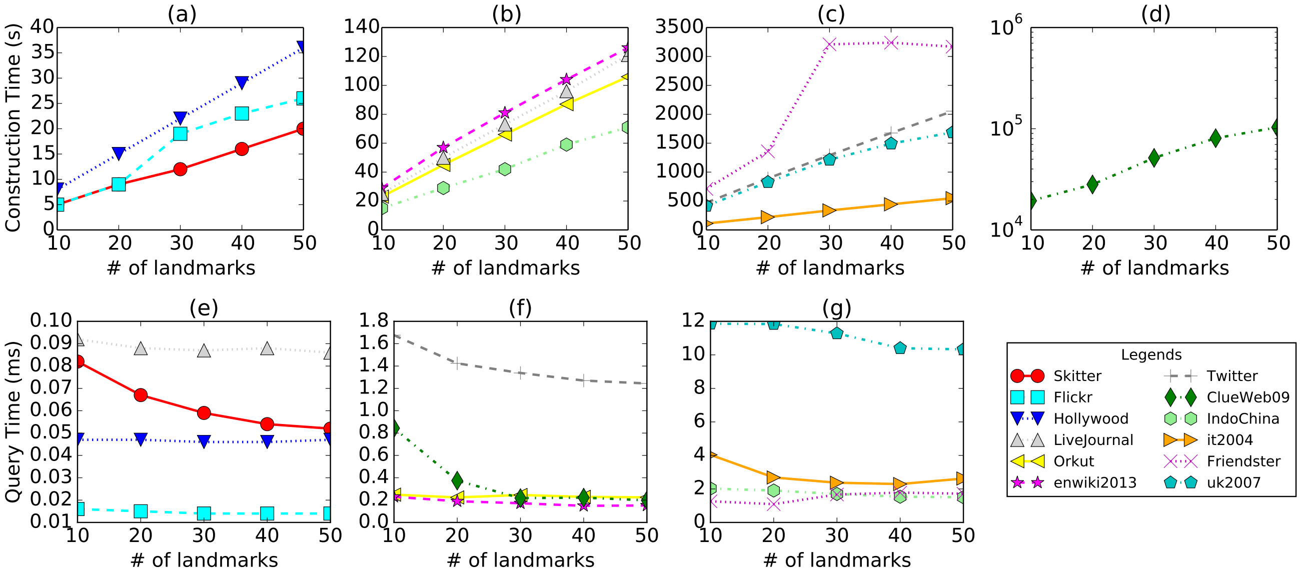


Figure 7: (a)-(d) Construction times using our method HL under 10-50 landmarks on all the datasets; (e)-(g) Query times using our method HL under 10-50 landmarks on all the datasets.

6.3.2 Labelling Size. As we can see from Table 3 that the labelling sizes of all the datasets constructed by the proposed method are significantly smaller than the labelling sizes of FD and much smaller than PLL and IS-Label. Specifically, our labelling sizes using 32-bits representation of vertices (HL) are 2-5 times smaller than FD except for ClueWeb09 (as discussed before, FD failed to construct labelling for ClueWeb09), 7 times smaller than IS-Label on Skitter, Flickr and LiveJournal and more than 60 times smaller than PLL for Skitter, Flickr, Hollywood, enwiki2013 and Indochina. The compressed version of our method that uses 8-bits representation of vertices (i.e. HL(8)) produces further smaller index sizes as compared to uncompressed version (HL). Here, It is important to note that the labelling sizes of almost all the datasets are also significantly smaller than the original sizes of the datasets shown in Table 1. This also shows that our method is highly scalable on large networks in terms of the labellng sizes.

6.3.3 Query Time. The average query times of our method (HL) are comparable with FD and PLL and faster than IS-L. Particularly, the average query time of our method on Hollywood is even faster than FD and PLL. This is due to a very small average labelling size (i.e., 12) as compared with FD and PLL (i.e., 20+64 and 2206+50, respectively) and a very small average distance. The average query time of HL on Twitter is 3 times slower than FD. This may be due to a large portion of covered pairs by FD as shown in Figure 9 which contributes towards an effective bounded traversal on the sparsified network since the landmarks of Twitter have very high degrees and the average distance is also very small. Moreover, the average query times of HL and FD on Indochina, it2004, Friendster and uk2007 are more than 1ms due to comparatively large average distances than other datasets as shown in Figure 6(b). Note that all the baseline methods are not scalable enough to have results for ClueWeb09 and the average query time on ClueWeb09 of our method HL is small because of a very large portion of covered pairs and a small average label size. We also reported the average query time for online bidirectional BFS algorithm (Bi-BFS) using randomly selected 1000 pairs of vertices in Table 2. As we can see that Bi-BFS has considerably long query times, which are not practicable in applications for performing distance queries in real time.

6.4 Performance under Varying Landmarks

We have also evaluated the performance of our method (HL) by varying the number of landmarks between 10 and 50, which are again selected based on highest degrees.

6.4.1 Construction Time. The construction times of our method HL against different numbers of landmarks (from 10 to 50) are shown in Figure 7(a)-7(d). We can see that the construction times are linear in terms of the number of landmarks, which confirms the scalability of our method. In Figure 7(a)-7(b), our method is able to construct labelling for 7 datasets under 50 landmarks from 20 seconds to 2 minutes, which is not possible with any state-of-the-art methods. In Figure 7(c), the construction time using 50 landmarks of Friendster is 3 times faster and the construction time of uk2007 is 4 times faster than FD using only 20 landmarks as shown in Table 2. Figure 7(d) shows the construction time for ClueWeb09 which has 2 billion vertices and 8 billion edges. The significant improvement in construction time allows us to compute labelling for a large number of landmarks, leading to better pair coverage ratios to tighten upper distance bounds (will be further discussed in Section 6.4.4).

6.4.2 Labelling Size. Figure 8 shows the labelling sizes of HL using 10, 20, 30, 40 and 50 landmarks on all the dataset, and of FD using only 20 landmarks on all the datasets except for ClueWeb09 (as discussed before, FD failed to construct labeling for ClueWeb09). It can be seen that the labelling sizes of HL increase linearly with the increased number of landmarks, and even the labelling sizes of HL using 50 landmarks are almost always smaller than the labelling sizes constructed by FD using only 20 landmarks. This reduction in labelling sizes enables us to save space and memory, thus makes our method scalable on large networks.

6.4.3 Query Time. Figure 7 shows the impact of using different numbers of landmarks between 10 and 50 on average query

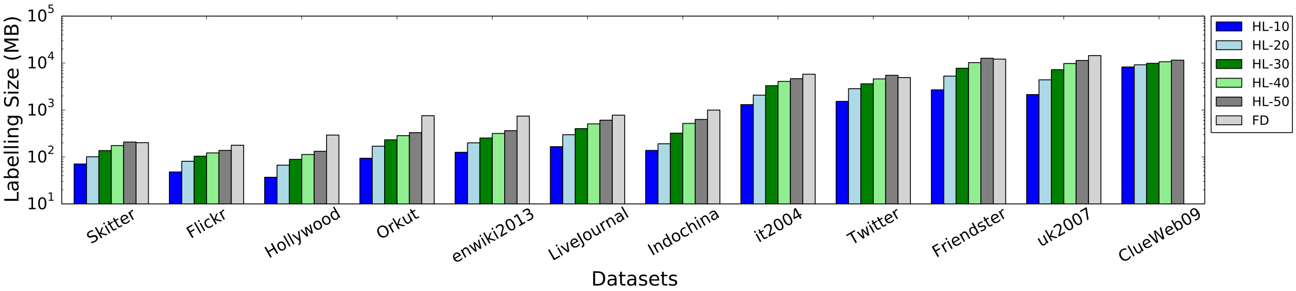


Figure 8: Labelling sizes using our method HL under 10-50 landmarks and using FD on all the datasets.

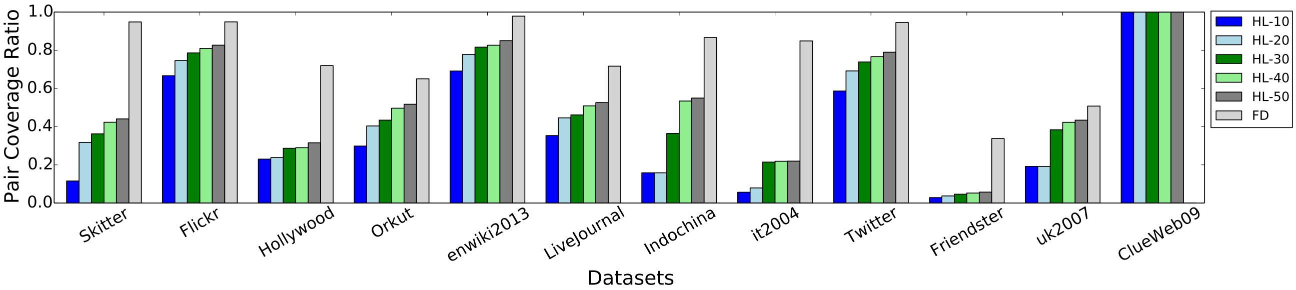


Figure 9: Pair coverage ratios using our method HL under 10-50 landmarks and using FD on all the dataset.

time of our method. The average query times either decrease or remain the same when the number of landmarks increases, except for Orkut when using 30 landmarks and for Friendster when using landmarks greater than 20. In particular, on Friendster, labelling sizes are very large as shown in Figure 8 and the fraction of covered pairs (i.e., pair coverage ratio) is very small as shown in Figure 9, which may have slowed down our query processing due to a longer time for computing upper distance bounds and ineffective use of bounded-distance traversal.

6.4.4 Pair Coverage. Figure 9 presents the ratios of pairs of vertices covered by at least one landmark (i.e., pair coverage ratios) in HL using 10-50 landmarks and in FD using 20 landmarks. As we can observe that the pair coverage ratios for HL increase when the number of landmarks increases and 40 turns out to be the better choice on the number of landmarks for most of the datasets. Specifically, pair coverage ratios on Orkut, enwiki2013, Indochina and uk2007 with 40 landmarks are good, resulting in better query times than using 20 landmarks, as shown in Figure 7. On datasets such as Hollywood and it2004, 30 landmarks are a better option than 40 landmarks because they only slightly differ in the pair coverage ratios and query times w.r.t. using 40 landmarks, but with reduced labelling sizes. The pair coverage ratios by FD are greater than HL on all the datasets except for ClueWeb09, which may be the reason behind its better query times for some datasets as shown in Table 2. Note that, on ClueWeb09, we obtain almost hundred percentage for pair coverage due to its very high degree landmarks.

7 RELATED WORK

A naive solution for exact shortest-path distance computation is to run the Dijkstra search for weighted graphs or BFS for unweighted graphs, from a source vertex to a destination vertex [27]. To improve search efficiency, a bidirectional scheme can be used to run two such searches: one from the source vertex and the other from the destination vertex [21]. Later on, Goldberg et al. [13] combined the bidirectional search technique with the A* algorithm to further improve the search performance. In their method, they precomputed labeling based on landmarks to estimate the lower bounds, and used that estimate with a bidirectional A* search for efficient computation of shortest-path distances. However, this method is known to work only for road networks and do not scale well on complex networks [15].

To efficiently answer exact shortest-path distance queries on graphs, labelling-based methods have been developed with great success [1–3, 12, 17, 19]. Most of them construct a labeling based on the idea of 2-hop cover [9]. It has also been shown that computing a minimal 2-hop cover labeling is NP-hard [2, 9]. In [1], the authors proposed a hub-based labeling algorithm (HL) which constructs hub labelling by processing contraction hierarchies (CH) and is among the fastest known algorithms for distance queries in road networks. However, the method is not feasible for complex networks as reported by the same authors and they thus proposed a hierarchical hub-labeling (HHL) algorithm for complex networks in [2]. In this work, a top-down method was used to maintain a shortest-path tree for every vertex in order to indicate all uncovered shortest-paths at each vertex. Due to very high storage and computation requirements, the method is also not scalable for handling large graphs. Another method called Highway Centric Labeling (HCL) was proposed by Jin et al. [17] which exploits highway structure of a graph. This method aimed to find a spanning tree which can assist in optimal distance labelling and used that spanning tree as a highway to compute a highway-based 2-hop labelling for fast distance computation. After that, in [3], Akiba et al. proposed the pruned landmark labeling (PLL) method which precomputes a distance-aware 2-hop cover index by performing a pruned breadth-first search (BFS) from every vertex. The idea is to prune vertices whose distance information can be obtained using a partially available 2-hop index constructed via previous BFSs. This work helps to achieve low construction cost and smaller index size due to reduced search space on million-scale networks. It has been shown that PLL outperforms other state-of-the-art methods available at the time of publication, including HHL [2], HCL [17] and TEDI [32]. However, PLL is still not feasible for constructing 2-hop cover indices for billion-scale networks due to a very high memory requirement for labelling construction.

Fu et al. [12] proposed IS-Label (IS-L) which gained significant scalability in precomputing 2-hop cover distance labellings for large graphs with hundreds of millions of vertices and edges. IS-L uses the notion of an independent set of vertices in a graph. First, it computes an independent set of vertices from a graph, then it constructs a graph by removing the independent set of vertices from the previous graph recursively and augments edges that preserve distance information after the removal of the independent set of vertices. All the vertices in the remaining graph preserve their distance information to/from each other. Generally, IS-L is regarded as a hybrid method that combines distance labelling with graph traversal for complex networks [19]. Following the same line of thought, very recently, Akiba et al. [15] proposed a method to accelerate shortest-path distances computation on large-scale complex networks. To the best of our knowledge, this work is most closely related to our work presented in this paper. The key idea of the method in [15] is to select a small set of landmarks R and precompute shortest-path trees (SPTs) rooted at each $r \in R$. Given any two vertices s and t, it first computes the upper bound by taking the minimum length among the paths that pass through R. Then a bidirectional BFS from s to t is conducted on the subgraph $G\backslash R$ to compute the shortest-path distances that do not pass through R and take the minimum of these two results as the answer to an exact distance query. The experiments in [15] showed that this method can scale to graphs with millions of vertices and billions of edges, and outperforms the state-of-the-art exact methods PLL [3], HDB [16], RXL and CRXL [10] with significantly reduced construction time and index size, while the query times are higher but still remain among 0.01-0.06 for most of graphs with less than 5M vertices.

Although the method proposed in [15] has been tested on a large network with millions of vertices and billions of edges, it still fails to construct labelling on billion-scale networks in general, particularly with billions of vertices. In contrast, our proposed method not only constructs labellings linearly with the number of landmarks in large networks with billions of vertices, but also enables the sizes of labellings to be significantly smaller than the original network sizes. In addition to these, the deterministic nature of labelling allows us to achieve further gains in computational efficiency using parallel BFSs over multiple landmarks, which is highly scalable for handling billion-scale networks.

8 CONCLUSION

We have presented a scalable solution for answering exact shortest path distance queries on very large (billion-scale) complex networks. The proposed method is based on a novel labelling algorithm that can scale to graphs at the billion-scale, and a querying framework that combines a highway cover distance labelling with distance-bounded searches to enable fast distance computation. We have proven that the proposed labelling algorithm can construct HWC-minimal labellings that are independent of the ordering of landmarks, and have further developed a parallel labelling method to speed up the labelling construction process by conducting BFSs simultaneously for multiple landmarks. The experimental results showed that the proposed methods significantly outperform the state-of-the-art methods. For future work, we plan to investigate landmark selection strategies for further improving the performance of labelling methods.

REFERENCES

[1] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *SEA*. 230–241.

[2] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2012. Hierarchical hub labelings for shortest paths. In *ESA*. 24–35.

[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *ACM SIGMOD*. 349–360.

[4] Takuya Akiba, Christian Sommer, and Ken-ichi Kawarabayashi. 2012. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *EDBT*. 144–155.

[5] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *ACM SIGKDD*. 44–54.

[6] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW*. 587–596.

[7] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW*. 595–601.

[8] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Hong Cheng, and Miao Qiao. 2012. The exact distance to destination in undirected world. *The VLDB Journal* 21, 6 (2012), 869–888.

[9] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[10] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. 2014. Robust distance queries on massive networks. In *ESA*. 321–333.

[11] Linton C Freeman. 1977. A set of measures of centrality based on betweenness. *Sociometry* (1977), 35–41.

[12] Ada Wai-Chee Fu, Huanhuan Wu, James Cheng, and Raymond Chi-Wing Wong. 2013. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *VLDB* 6, 6 (2013), 457–468.

[13] Andrew V Goldberg and Chris Harrelson. 2005. Computing the shortest path: A search meets graph theory. In *SODA*. 156–165.

[14] Andrey Gubichev, Srikanta Bedathur, Stephan Seufert, and Gerhard Weikum. 2010. Fast and accurate estimation of shortest paths in large graphs. In *CIKM*. 499–508.

[15] Takanori Hayashi, Takuya Akiba, and Ken-ichi Kawarabayashi. 2016. Fully Dynamic Shortest-Path Distance Query Acceleration on Massive Networks. In *CIKM*. 1533–1542.

[16] Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong, and Yanyan Xu. 2014. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *VLDB* 7, 12 (2014), 1203–1214.

[17] Ruoming Jin, Ning Ruan, Yang Xiang, and Victor Lee. 2012. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *ACM SIGMOD*. 445–456.

[18] Jure Leskovec and Andrej Krevl. 2015. SNAP Datasets:Stanford Large Network Dataset Collection. (2015).

[19] Ye Li, Man Lung Yiu, Ngai Meng Kou, et al. 2017. An experimental study on hub labeling based shortest path algorithms. *VLDB* 11, 4 (2017), 445–457.

[20] Silviu Maniu and Bogdan Cautis. 2013. Network-aware search in social tagging applications: instance optimality versus efficiency. In *CIKM*. 939–948.

[21] Ira Pohl. 1971. Bi-derectional search. *Machine intelligence* 6 (1971), 127–140.

[22] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. 2009. Fast shortest path distance estimation in large networks. In *CIKM*. 867–876.

[23] Miao Qiao, Hong Cheng, Lijun Chang, and Jeffrey Xu Yu. 2014. Approximate shortest distance computing: A query-dependent local landmark scheme. *IEEE TKDE* 26, 1 (2014), 55–68.

[24] Yongrui Qin, Quan Z Sheng, Nickolas JG Falkner, Lina Yao, and Simon Parkinson. 2017. Efficient computation of distance labeling for decremental updates in large dynamic graphs. *WWW* 20, 5 (2017), 915–937.

[25] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. http://networkrepository.com

[26] Gert Sabidussi. 1966. The centrality index of a graph. *Psychometrika* 31, 4 (1966), 581–603.

[27] Robert Endre Tarjan. 1983. *Data structures and network algorithms*. Vol. 44. Siam.

[28] KONECT the Koblenz Network Collection. 2017. (2017). http://konect.uni-koblenz.de/networks/

[29] Konstantin Tretyakov, Abel Armas-Cervantes, Luciano García-Bañuelos, Jaak Vilo, and Marlon Dumas. 2011. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *CIKM*. 1785–1794.

[30] Antti Ukkonen, Carlos Castillo, Debora Donato, and Aristides Gionis. 2008. Searching the wikipedia with contextual information. In *CIKM*. 1351–1352.

[31] Monique V Vieira, Bruno M Fonseca, Rodrigo Damazio, Paulo B Golgher, Davi de Castro Reis, and Berthier Ribeiro-Neto. 2007. Efficient search ranking in social networks. In *CIKM*. 563–572.

[32] Fang Wei. 2010. TEDI: efficient shortest path query answering on graphs. In *ACM SIGMOD*. 99–110.

[33] Sihem Amer Yahia, Michael Benedikt, Laks VS Lakshmanan, and Julia Stoyanovich. 2008. Efficient network aware search in collaborative tagging sites. *VLDB* 1, 1 (2008), 710–721.

Stratified Random Sampling from Streaming and Stored Data

Trong Duc Nguyen
Iowa State University, USA

Ming-Hung Shih
Iowa State University, USA

Divesh Srivastava
AT&T Labs–Research, USA

Srikanta Tirthapura
Iowa State University, USA

Bojian Xu
Eastern Washington University, USA

ABSTRACT

Stratified random sampling (SRS) is a widely used sampling technique for approximate query processing. We consider SRS on continuously arriving data streams, and make the following contributions. We present a lower bound that shows that any streaming algorithm for SRS must have (in the worst case) a variance that is $\Omega(r)$ factor away from the optimal, where r is the number of strata. We present S-VOILA, a streaming algorithm for SRS that is *locally variance-optimal*. Results from experiments on real and synthetic data show that S-VOILA results in a variance that is typically close to an optimal offline algorithm, which was given the entire input beforehand. We also present a variance-optimal offline algorithm VOILA for stratified random sampling. VOILA is a strict generalization of the well-known *Neyman allocation*, which is optimal only under the assumption that each stratum is abundant, i.e. has a large number of data points to choose from. Experiments show that VOILA can have significantly smaller variance (1.4x to 50x) than Neyman allocation on real-world data.

1 INTRODUCTION

Random sampling is a widely-used method for data analysis, and features prominently in the toolbox of virtually every approximate query processing system. The power of random sampling lies in its generality. For many important classes of queries, an approximate answer whose error is small in a statistical sense can be efficiently obtained through executing the query over an appropriately derived random sample. Sampling operators are part of all major database products, *e.g.,* Oracle, Microsoft SQL Server, and IBM Db2. The simplest method for random sampling is uniform random sampling, where each element from the entire data (the "population") is chosen with the same probability. Uniform random sampling may however lead to a high variance in estimation. For instance, consider a population $D = \{1, 2, 4, 2, 1, 1050, 1000, 1200, 1300\}$, and suppose we wanted to estimate the population mean. A uniform random sample of size two leads to an estimate with a variance of approximately 1.6×10^5.

An alternative sampling method is *stratified random sampling* (SRS), where the population is partitioned into subgroups called "strata". From within each stratum, uniform random sampling is used to select a per-stratum sample. All per-stratum samples are combined to derive the "stratified random sample". Suppose that the population is divided into two strata, one with elements $\{1, 2, 4, 2, 1\}$ and the other with elements $\{1000, 1050, 1200, 1300\}$. A stratified random sample of size two that chooses one element from each stratum yields an estimate with variance 2.47×10^3, much smaller than a uniform random sample of the same size.

SRS provides the flexibility to emphasize some strata over others through controlling the allocation of sample sizes; for instance, a stratum with a high standard deviation can be given a larger allocation than another stratum with a smaller standard deviation. In the above example, if we desire a stratified sample of size three, it is best to allocate a smaller sample of size one to the first stratum and a larger sample size of two to the second stratum, since the standard deviation of the second stratum is higher. Doing so, the variance of estimate of the population mean further reduces to approximately 1.23×10^3. The strength of SRS is that a stratified random sample can be used to answer queries not only for aggregates such as the mean, and sum of the entire population, but also of subsets of the population defined by selection predicates that are provided at query time. SRS has been used widely in database systems for approximate query processing [1–3, 8, 14, 30, 31].

A problem with handling large streaming data is that current methods for SRS are predominantly offline methods that assume all data is available before sampling starts. As a result, systems that rely on SRS (*e.g.,* [3, 14, 31]) cannot easily adapt to new data and have to recompute stratified random samples from scratch, as more data arrives. With the advent of streaming data warehouses such as Tidalrace [29], it is imperative to have methods for SRS that work on dynamic data streams, and maintain stratified random samples in an incremental manner.

We address the shortcoming of current methods through a study of SRS on streaming data. The difficulty of SRS on streaming data is that there are two logical processes simultaneously at work. One is sample size allocation, which allocates samples among the different strata in a manner that minimizes the variance of an estimate. The second is the actual sampling of elements from within each stratum. While each of these two steps, sample size allocation and sampling, can be done individually in a streaming fashion, it is far more challenging to do them simultaneously. We present lower bounds as well as algorithms for the task of maintaining a stratified random sample on a data stream. The quality of a stratified random sample is measured using the variance of an estimate of a population statistic, computed using the sample.

1.1 Our Contributions

– **Streaming Lower Bound:** We present a lower bound showing that in the worst case, any streaming algorithm for SRS that uses a memory of M records must have a variance that is $\Omega(r)$ away from the variance of the optimal offline algorithm that uses the same memory of M records, where r is the number of strata. We show that this lower bound is tight, by construction.

– **Practical Streaming Algorithm for SRS:** We present S-VOILA (Streaming Variance OptImaL Allocation) a streaming algorithm for SRS that is locally variance-optimal. Upon receiving new elements, it (re-)allocates sample sizes among strata so as to obtain the smallest variance among all possible re-allocations. S-VOILA can also deal with the case when a minibatch of multiple data

Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.04

items is seen at a time, as in systems such as Spark streaming [42]. Re-allocations made by S-VOILA are locally optimal with respect to the entire minibatch, and the quality of re-allocations improve as the minibatch size increases. Since S-VOILA can deal with minibatches of varying sizes, it is well-suited to real-world streams that may have bursty arrivals.

– **Variance-Optimal Sample Size Reduction:** The streaming algorithm (S-VOILA) re-allocates sample sizes based on a novel method for reducing the size of an existing stratified random sample down to a desired target size in a variance-optimal manner. This novel technique for sample size reduction may be of independent interest in other tasks, e.g., sub-sampling from a given stratified random sample.

– **Sampling from a Sliding Window of a Stream:** We present an algorithm for sampling from a sliding window of the most recent elements in a stream. This algorithm uses memory much smaller than the size of the window, and results in a sample whose variance is close to that obtained by an optimal offline algorithm that is allowed multiple passes through data in the window.

– **Variance Optimal Offline SRS:** We present the first offline algorithm for variance-optimal SRS. Our algorithm VOILA computes an allocation with provably optimal variance among all possible allocations of sample sizes to strata. The well known **Neyman Allocation** [36] (NeyAlloc), which originated from the statistics literature, assumes that each stratum has an abundance of data to choose from. However, this assumption may not hold in databases, since each stratum is a subset of a database table, and the size of a stratum may be small. VOILA does not make such assumptions, and computes a variance optimal allocation no matter how large/small the sizes of the strata. Hence, VOILA is a strict generalization of NeyAlloc. In addition, VOILA does not make any assumption on how data is stratified.

– **Experimental Evaluation:** We present a detailed experimental evaluation using real and synthetic data, considering both the quality of sample and accuracy of query answers using the sample. In our experimental study, we found that (a) the variance of S-VOILA is typically close to that of the optimal offline algorithm VOILA, and the allocation of S-VOILA also closely tracks that of VOILA. S-VOILA also improves significantly upon prior work [5]. The variance of S-VOILA improves as the size of the minibatch increases, and a minibatch of size 100 provides most of the benefits of S-VOILA. (b) Samples produced using S-VOILA yield accurate answers to a range of queries that involve a selection followed by aggregation, where the selection predicate is provided at query time, and the aggregation function can be one of sum, average, and standard deviation[1]. (c) In the offline setting, VOILA can have significantly smaller variance than NeyAlloc.

1.2 Related Work

Sampling has been widely used in approximate query processing on both static and streaming data [17, 28, 33, 38, 39]. The reservoir sampling [34, 41] algorithm for uniform sampling from a stream has been known for decades, and many variants and generalizations have been considered, such as weight-based sampling [11, 22], insertion and deletion of elements [25], distinct sampling [26], sampling from a sliding window [9, 12, 23], time-decayed sampling [19, 20], and distributed streaming sampling [15, 16, 18, 40].

SRS in the online setting can be viewed as weight-based reservoir sampling where the weight of each stream element depends on the stratum it belongs to. Since the weight of a stream element changes dynamically (even after it has been observed) prior work on weighted reservoir sampling [22] does not apply, since it assumes that the weight of an element is known at the time of observation and does not change henceforth. Meng [35] considered streaming SRS using population-based allocation. Al-Kateb *et al.* [4, 5] considered streaming SRS using power allocation, based on their prior work on adaptive reservoir sampling [6]. Lang *et al.* [32] consider machine learning methods for determining the per-item probability of inclusion in a sample. This work is meant for static data, and can be viewed as a version of weighted random sampling where the weights are learnt using a query workload. Prior work on streaming SRS neither considers provable guarantees on the quality of the resulting samples, nor lower bounds for streaming SRS, like we do here.

A majority of prior work on using SRS in approximate query processing [1–3, 8, 14, 30, 31] has assumed static data. With the emergence of data stream processing systems [7] and data stream warehousing systems [29], it is important to devise methods for streaming SRS with quality guarantees.

2 PRELIMINARIES

Stratified sampling can be viewed as being composed of three parts – stratification, sample allocation, and sampling. Stratification is a partitioning of the universe into a number of disjoint strata. Equivalently, it is the assignment of each data element to a unique stratum. In database applications, stratification is usually a pre-defined function of one or more attributes of the data [17]. For example, the works of Chaudhuri et al. [14] and Agarwal et al. [3] on approximate query answering stratify tuples in a database table based on the set of selection predicates in the query workload that the tuple satisfies, and the work of Kandula et al. [31] on approximate query answering stratify rows of a table using the group ids derived from a group-by query. Note that our methods do not assume that stratification is performed in any specific manner, and work regardless of the method used to stratify data.

Our work considers sample allocation, the partitioning of the available memory budget of M samples among the different strata. In streaming SRS, the allocation needs to be continuously readjusted as more data arrives, and the characteristics of different strata change. In offline sampling, allocation needs to be done only once, after knowing the data in its entirety.

The final sampling step considers each stratum and chooses the assigned number of samples uniformly at random. In offline stratified sampling, the sampling step can be performed in a second pass through the data using reservoir sampling on the subset of elements belonging to each stratum, after a first pass has determined the sample size allocation. In the case of streaming sampling, the sampling step needs to occur simultaneously with sample (re-)allocation, which may change allocations to different strata over time.

Variance-Optimal Allocation. The quality of a stratified random sample is measured through the variance of an estimate that is derived using the sample. Consider a data stream

[1] Note that a query for the variance or standard deviation of data is distinct from the variance or standard deviation of an estimate.

$R = \{v_1, v_2, \ldots, v_n\}$ of current size n, whose elements are stratified into r strata, numbered $1, 2, \ldots, r$. Let n_i denote the number of elements in stratum i. For each $i = 1 \ldots r$, let S_i be a uniform random sample of size s_i drawn without replacement from stratum i. Let $\mathbb{S} = \{S_1, S_2, \ldots, S_n\}$ denote the stratified random sample. The sample mean of each per-stratum sample S_i is: $\bar{y}_i = \frac{\sum_{v \in S_i} v}{s_i}$. The population mean of R, μ_R can be estimated as: $\bar{y} = \frac{\sum_{i=1}^{r} n_i \bar{y}_i}{n}$. It can be shown that the expectation of $\bar{y}$ equals μ_R. Given a memory budget of $M \leq n$ elements to store all the samples, so that $\sum_i s_i = M$, the following question of *variance-optimal allocation* of sample sizes has been considered in prior work [36]: *How to split the memory budget M among the s_is to minimize the variance of $\bar{y}$?* The variance of $\bar{y}$ can be computed as follows (e.g. see Theorem 5.3 in [17]):

$$V = V(\bar{y}) = \frac{1}{n^2} \sum_{i=1}^{r} n_i (n_i - s_i) \frac{\sigma_i^2}{s_i} = \frac{1}{n^2} \sum_{i=1}^{r} \frac{n_i^2 \sigma_i^2}{s_i} - \frac{1}{n^2} \sum_{i=1}^{r} n_i \sigma_i^2 \tag{1}$$

While the theory around SRS in both statistics and database communities has used the variance of the population mean as a minimization metric, variance-optimal SRS is useful for other types of queries as well, including predicate-based selection queries, sum queries across a subset of the strata, queries for the variance, and combinations of such queries [3, 14] – also see our experiments section.

`NeyAlloc` for Abundant Strata. Prior studies on variance-optimal allocation have primarily considered static data. Additionally, they assume that every stratum has a very large volume of data, so that there is no restriction on the size of a sample that can be chosen from this stratum. This may not be true for the scenario of databases. Especially in a streaming context, each stratum starts out with very little data. Given a collection of data elements R, we say that a stratum i is *abundant* if $n_i \geq M \cdot (n_i \sigma_i) / \left(\sum_{j=1}^{r} n_j \sigma_j \right)$. Otherwise, the stratum i is said to be *bounded*. Under the assumption that each stratum is abundant, the popularly used "Neyman Allocation" `NeyAlloc` [17, 36] minimizes the variance V, and allocates a sample size for stratum i as $M \cdot (n_i \sigma_i) / \left(\sum_{j=1}^{r} n_j \sigma_j \right)$. We note that `NeyAlloc` is no longer optimal if one or more strata are bounded. Our methods of sample size reduction and online (`S-VOILA`) and offline (`VOILA`) algorithms do not have this restriction and work under the general case whether or not strata are bounded.

Our solution to streaming SRS consists of two parts – sample size re-allocation, and per-stratum random sampling. Both parts execute continuously and in an interleaved manner. Sample size re-allocation is achieved using a reduction to a "sample size reduction" in a variance-optimal manner. Given a stratified random sample $\mathbb{S}_1$ of size larger than a target M, sample size reduction seeks to find a stratified sample $\mathbb{S}_2$ of size M that is a subset of $\mathbb{S}_1$ such that the variance of $\mathbb{S}_2$ is as small as possible.

Roadmap: In Section 3, we consider streaming SRS, and present a tight lower bound for any streaming algorithm, followed by `S-VOILA` an algorithm for streaming SRS. This uses as a subroutine a variance-optimal sample size reduction method that we describe in Section 4. We start with `SingleElementSSR` for reducing the size of the sample by one element, followed by a general algorithm `SSR` for reducing the size by $\beta \geq 1$ elements. We then present an algorithm `MultiElementSSR` with a faster runtime. We then consider the case of sliding windows in Section 5, followed by the optimal offline algorithm in Section 6. We present an experimental study of our algorithms in Section 7.

3 STREAMING SRS

We now consider SRS from a data stream, whose elements are arriving continuously. As more elements are seen, the allocations as well as samples need to be dynamically adjusted. We first note there is a simple two-pass streaming algorithm with optimal variance that uses $O(k + r)$ space, where k is the desired sample size and r the number of strata. In the first pass, the size, mean, and standard deviations of each stratum are computed using $O(r)$ space, constant space for each stratum. At the end of the first pass, the allocations to different strata are computed using an optimal offline algorithm, say `VOILA`. In the second pass, since the desired sample sizes are known for each stratum, samples are computed using reservoir sampling within the substream of elements belonging to each stratum. The above two-pass algorithm *cannot be* converted into a one-pass algorithm. The difficulty is that as more elements are seen, allocations to different strata may change, and the sampling rate within a stratum cannot in general be (immediately) dynamically adjusted in order to satisfy variance optimality. We first show a lower bound that it is in general not possible for any streaming algorithm to have optimal variance compared with an offline algorithm that is given the same memory.

3.1 A Lower Bound for Streaming SRS

Given a data stream $\mathcal{R}$ with elements belonging to r strata, and a memory budget of M elements, let V^* denote the optimal sample variance that can be achieved by an offline algorithm for SRS that may make multiple passes through data. Clearly, the sample produced by any streaming algorithm must have variance that is either V^* or greater. Suppose a stratified random sample R is computed by a streaming algorithm using memory of M elements. Let $V(R)$ denote the variance of this sample. For $\alpha \geq 1$, we say R *is an SRS with multiplicative error of α, if: (1) the sample within each stratum in R is chosen uniformly from all elements in the stratum, and (2) $V(R) \leq \alpha \cdot V^*$.*

Theorem 3.1. *Any streaming algorithm for maintaining an SRS over a stream with r strata using a memory of M elements must, in the worst case, result in a stratified random sample with a multiplicative error $\Omega(r)$.*

The idea in the proof is to construct an input stream with r strata where the variance of different strata are the same until a certain point in time, at which the variance of a single stratum starts increasing to a high value – a variance-optimal SRS will respond by increasing the allocation to this stratum. However, we show that a streaming algorithm is unable to do so quickly. Though a streaming algorithm may compute the variance-optimal allocation to different strata in an online manner, it cannot actually maintain these dynamically sized samples using limited memory.

Proof. Consider an input stream where for each $i = 1 \ldots r$, the ith stratum consists of elements in the range $[i, i + 1)$. The stream so far has the following elements. For each i, $1 \leq i \leq r$, there are $(\alpha - 1)$ copies of element i and one copy of $(i + \varepsilon)$ where $\varepsilon = 1/(r - 1)$ and $\alpha \geq 3$. After observing these elements, for stratum i we have $n_i = \alpha$, $\mu_i = \left(i + \frac{\varepsilon}{\alpha}\right)$, and it can be verified that $\sigma_i = \frac{\sqrt{\alpha - 1}}{\alpha} \varepsilon$.

Since the total memory budget is M, at least one stratum (say, Stratum 1) has a sample size no more than M/r. Suppose an element of value $(2 - \varepsilon)$ arrives next. This element belongs to stratum 1. Let n'_1, μ'_1, and σ'_1 denote the new size, mean, and standard deviation of stratum 1 after this element arrives. We have $n'_1 = \alpha + 1$ and $\mu'_1 = 1 + \frac{1}{\alpha+1}$. It can be verified that $\sigma'_1 = \sqrt{\frac{\varepsilon^2+(1-\varepsilon)^2-\frac{1}{\alpha+1}}{\alpha+1}}$. It follows that:

$$(\alpha + 1)\sqrt{\frac{\frac{1}{2} - \frac{1}{\alpha+1}}{\alpha + 1}} \leq n'_1\sigma'_1 \leq (\alpha + 1)\sqrt{\frac{1 - \frac{1}{\alpha+1}}{\alpha + 1}} \tag{2}$$

$$\Longrightarrow \frac{\sqrt{\alpha}}{2} \leq n'_1\sigma'_1 \leq \sqrt{\alpha} \qquad \text{(Note: } \alpha > 2\text{)} \tag{3}$$

In 2, the left inequality stands when $\varepsilon = 1/2$ and the right inequality stands when $\varepsilon = 0$ or 1. We also have: $\sum_{i=2}^{r} n_i\sigma_i = (r-1)\alpha\frac{\sqrt{\alpha-1}}{\alpha}\varepsilon = \sqrt{\alpha - 1}$, where we have used $\varepsilon = \frac{1}{r-1}$. Thus,

$$\frac{\sqrt{\alpha}}{2} \leq \sum_{i=2}^{r} n_i\sigma_i \leq \sqrt{\alpha} \qquad \text{(Note: } \alpha > 2\text{)} \tag{4}$$

Let V denote the sample variance of $\mathcal{A}$ after observing the stream of $(r\alpha + 1)$ elements. Let V^* denote the smallest sample possible with a stratified random sample of size M on this data. Let $\Delta = \left(n'_1\sigma'^2_1 + \sum_{i=2}^{r} n_i\sigma_i^2\right)/n^2$.

We observe that after processing these $(r\alpha + 1)$ elements, the sample size $s_1 \leq M/r + 1$. Using this fact and the definition of sample variance in Eq 1:

$$V = \frac{1}{n^2}\left(\frac{n'^2_1\sigma'^2_1}{s_1} + \sum_{i=2}^{r}\frac{n_i^2\sigma_i^2}{s_i}\right) - \Delta \geq \frac{1}{n^2}\left(\frac{n'^2_1\sigma'^2_1}{\frac{M}{r}+1} + \sum_{i=2}^{r}\frac{n_i^2\sigma_i^2}{\frac{M}{r-1}}\right) - \Delta$$

$$\geq \frac{1}{n^2}\left(\frac{\alpha/4}{\frac{M}{r}+1} + \sum_{i=2}^{r}\frac{(\alpha-1)\varepsilon^2}{M/(r-1)}\right) - \Delta = \frac{1}{n^2}\left(\frac{\alpha/4}{\frac{M}{r}+1} + \frac{\alpha - 1}{M}\right) - \Delta$$

On the other hand, the smallest sample variance V^* is achieved by using Neyman allocation. By Inequalities 3 and 4, we know that if Neyman allocation is for the current stream of $r\alpha + 1$ elements, stratum 1 uses at least $M/3$ memory space, whereas all other strata equally share at least $M/3$ elements since all $n_i\sigma_i$ are equal for $i = 2, 3, \ldots, r$. Using these observations into Equation 1:

$$V^* \leq \frac{1}{n^2}\left(\frac{n'^2_1\sigma'^2_1}{M/3} + \sum_{i=2}^{r}\frac{n_i^2\sigma_i^2}{M/(3(r-1))}\right) - \Delta$$

$$\leq \frac{1}{n^2}\left(\frac{\alpha}{M/3} + \sum_{i=2}^{r}\frac{(\alpha-1)\varepsilon^2}{M/(3(r-1))}\right) - \Delta = \left(\frac{1}{n^2}\frac{6\alpha - 3}{M}\right) - \Delta$$

Since $\Delta \geq 0$ and $M > r$, we have: $\frac{V}{V^*} \geq \frac{V+\Delta}{V^*+\Delta} = \Omega(r)$. □

We note that the above lower bound is tight (up to constant factors). Consider the algorithm which always allocates M/r memory to each of r strata that have been observed so far. It can be verified that this algorithm has a variance within an $O(r)$ multiplicative factor of the optimal. While theoretically such an algorithm (which we call the "senate" algorithm due to allocating every stratum the same resources) meets the worst case lower bound, it performs poorly in practice, since it treats all strata equally, irrespective of their volume or variance (see the experiments section).

3.2 S-VOILA: Streaming Algorithm for SRS

We now present a streaming algorithm S-VOILA that can maintain a stratified random sample on a stream with a good (though not optimal) variance. Given a memory budget of M items, S-VOILA maintains a SRS of size M with the following properties: (1) the samples within each stratum are chosen uniformly from all the stream elements seen in the stratum so far, (2) the sizes of samples allocated to different strata adapt to new stream elements by making "locally optimal" decisions that lead to the best allocations given the new stream elements. S-VOILA conceptually has to solve two problems. One is sample size re-allocation among strata, and the second is uniform sampling within each stratum. Let $\mathcal{R}$ denote the stream observed so far, and $\mathcal{R}_i$ the elements in $\mathcal{R}$ that belong to stratum i.

We first consider sample size re-allocation. Suppose due to the addition of new elements, the stream went from $\mathcal{R}^1$ to $\mathcal{R}^2$, and suppose that the stratified random sample at $\mathcal{R}^1$ allocated sample sizes to strata in a specific manner, S^1. Due to the new elements, the sizes and variances of different strata change, and as a result, the optimal allocation of samples in $\mathcal{R}^2$ may be different from the previous allocation S^1. Our approach is to first add new elements to the sample, and then re-allocate sample sizes using a "variance-optimal sample size reduction" optimization framework. *Given a current allocation of sample sizes to different strata, suppose new elements are added to the sample, causing it to exceed a memory threshold M. What is a way to reduce the current sample to a sample of size M such that the variance of the new sample is as small as possible?* In the following section (Section 4), we present algorithms for sample size reduction.

The second issue is to maintain a uniform random sample S_i of $\mathcal{R}_i$ when s_i, the size of the sample is changing. A decrease in an allocation to s_i can be handled easily, through discarding elements from the current sample S_i until the desired sample size is reached. What if we need to increase the allocation to stratum i? If we simply start sampling new elements according to the higher allocation to S_i, then recent elements in the stream will be favored over the older ones, and the sample within stratum i is no longer uniformly chosen. In order to ensure that S_i is always chosen uniformly at random from $\mathcal{R}_i$, newly arriving elements in $\mathcal{R}_i$ need to be held to the same sampling threshold as older elements, even if the allotted sample size s_i increases. S-VOILA resolves this issue in the following manner. An arriving element from $\mathcal{R}_i$ is assigned a random "key" drawn uniformly from the interval $(0, 1)$. The sample is maintained using the following invariant: *S_i is the set of s_i elements with the smallest keys among all elements so far in $\mathcal{R}_i$*. It is easy to verify that this is indeed a uniform sample drawn without replacement from $\mathcal{R}_i$. The consequence of this strategy is that if we desire to increase the allocation to stratum i, it may not be accomplished immediately, since a newly arriving element in $\mathcal{R}_i$ may not be assigned a key that meets this sampling threshold. Instead, the algorithm has to wait until it receives an element in $\mathcal{R}_i$ whose assigned key is small enough. To ensure the above invariant, the algorithm maintains for each stratum i a variable d_i that tracks the smallest key of an element in $\mathcal{R}_i$ that is not currently included in S_i. If an arriving element in $\mathcal{R}_i$ has a key that is smaller than or equal to d_i, it is included within S_i; otherwise, it is not.

Algorithms 1 and 2 respectively describe the initialization and insertion of a minibatch of elements. S-VOILA supports the insertion of a minibatch of any size $b > 0$, where b can change from one minibatch to another. As b increases, we can expect

Algorithm 1: S-VOILA: Initialization

```
Input: $M$ – total sample size, $r$ – number of strata.
// $S_i$ is the sample for stratum $i$, and $\mathcal{R}_i$ is the
   substream of elements from Stratum $i$
1 Load the first $M$ stream elements in memory, and partition
    them into per-stratum samples, $S_1, S_2, \ldots, S_r$, such that $S_i$
    consists of $(e, d)$ tuples from stratum $i$, where $e$ is the
    element, $d$ is the key of the element, chosen independently
    and uniformly at random from $(0, 1)$.
2 For each stratum $i$, compute $n_i, \sigma_i$. Initialize $d_i \leftarrow 1$ ; // $d_i$
    tracks the smallest key among all elements in
    $\mathcal{R}_i$ not selected in $S_i$
```

Algorithm 2: S-VOILA: Process a new minibatch B of b elements. Note that b need not be known in advance, and can vary from one minibatch to the other.

```
1  $\beta \leftarrow 0$;                    // #selected elements from $B$
2  for each $e \in B$ do
3      Let $\alpha = \alpha(e)$ denote the stratum of $e$
4      Update $n_\alpha$ and $\sigma_\alpha$ ;  // per-stratum mean and std.
         dev. maintained in a streaming manner
5      Assign a random key $d \in (0, 1)$ to element $e$;
6      if $d \le d_\alpha$ then              // element $e$ is sampled
7          $S_\alpha \leftarrow \{e\} \bigcup S_\alpha$; $\beta \leftarrow \beta + 1$;
   /* Variance-optimal reduction by $\beta$ elements    */
8  if $\beta = 1$ then      // faster for evicting 1 element
9      $\ell \leftarrow$ SingleElementSSR($M$);
10     Delete one element of largest key from $S_\ell$;
11     $d_\ell \leftarrow$ smallest key discarded from $S_\ell$;
12 else if $\beta > 1$ then
13     $\mathcal{L} \leftarrow$ MultiElementSSR($M$);
14     for $i = 1 \ldots r$ do      // Actual element evictions
15         if $\mathcal{L}[i] < s_i$ then
16             Delete $s_i - \mathcal{L}[i]$ elements from $S_i$ with the
                 largest keys;
17             $d_i \leftarrow$ smallest key discarded from $S_i$;
```

S-VOILA to have a lower variance, since its decisions are based on greater amount of data. Lines 2–7 make one pass through the minibatch to update the mean and standard deviations of the strata, and store selected elements into the per-stratum samples. If $\beta > 0$ elements from the minibatch get selected into the sample, in order to balance the memory budget at M, β elements need to be evicted from the stratified random sample using the variance-optimal sample size reduction technique from Section 4.

A sample size reduction algorithm takes a current allocation to a stratified random sample, the statistics (volume, mean, and variance) of different strata, and a target sample size M, and returns the final allocation whose total size is M. For the special case of evicting one element, we can use the faster algorithm SingleElementSSR; otherwise, we can use MultiElementSSR. Lemma 3.2 shows that the sample maintained by S-VOILA within each stratum is a uniform random sample, showing this is a valid stratified sample, and Lemma 3.3 presents the time complexity analysis of S-VOILA. Proofs are omitted due to space constraints.

Lemma 3.2. *For each $i = 1, 2, \ldots, r$ sample S_i maintained by* S-VOILA *(Algorithm 2) is selected uniformly at random without replacement from stratum R_i.*

Lemma 3.3. *If the minibatch size $b = 1$, then the worst-case time cost of* S-VOILA *for processing an element is $O(r)$. The expected time for processing an element belonging to stratum α is $O(1 + r \cdot s_\alpha / n_\alpha)$, which is $O(1)$ when $r \cdot s_\alpha = O(n_\alpha)$. If $b > 1$, then the worst-case time cost of* S-VOILA *for processing a minibatch is $O(r \log r + b)$.*

We can expect S-VOILA to have an amortized per-item processing time of $O(1)$ in many circumstances. When $b = 1$: After observing enough stream elements from stratum α, such that $r \cdot s_\alpha = O(n_\alpha)$, the expected processing time of an element becomes $O(1)$. Even if certain strata have a very low frequency, the expected time cost for processing a single element is still expected to be $O(1)$, because elements from an infrequent stratum α are unlikely to appear in the minibatch. When $b > 1$: The per-element amortized time cost of S-VOILA is $O(1)$, when the minibatch size $b = \Omega(r \log r)$.

4 VARIANCE-OPTIMAL SAMPLE SIZE REDUCTION

Suppose it is necessary to reduce a stratified random sample (SRS) of total size M to an SRS of total size $M' < M$. This will need to reduce the size of the samples of one or more strata in the SRS. Since the sample sizes are reduced, the variance of the resulting estimate will increase. We consider the task of *variance-optimal sample size reduction (*VOR*), i.e.,* how to partition the reduction in sample size among the different strata in such a way that the increase in the variance is minimized. Note that once the new sample size for a given stratum is known, it is easy to subsample the stratum to the target sample size.

Consider Equation 1 for the variance of an estimate derived from the stratified random sample. Note that, for a given data set, a change in the sample sizes of different strata s_i does not affect the parameters n, n_i, and σ_i. VOR can be formulated as the solution to the following non-linear program.

$$\text{Minimize } \sum_{i=1}^{r} \frac{n_i^2 \sigma_i^2}{s_i'} \tag{5}$$

subject to constraints:

$$\sum_{i=1}^{r} s_i' = M' \qquad \text{and} \qquad 0 \le s_i' \le s_i \text{ for each } i = 1, 2, \ldots, r, \tag{6}$$

In the rest of this section, we present efficient approaches for computing the VOR.

4.1 Sample Size Reduction by One Element

We first present an efficient algorithm for the case where the size of a stratified random sample is reduced by one element. An example application of this case is in designing a streaming algorithm for SRS, when stream items arrive one at a time. The task is to choose a stratum i (and discard a random element from the stratum) such that after reducing the sample size s_i by one, the increase in variance V (Equation 1) is the smallest.

Our solution is to choose stratum i such that the partial derivative of V with respect to s_i is the largest over all possible choices of i.

$$\frac{\partial V}{\partial s_i} = -\frac{n_i^2 \sigma_i^2}{n^2} \frac{1}{s_i^2}.$$

Given a memory budget M and stratum i, let $M_i = M \cdot n_i\sigma_i / \sum_{j=1}^{r} n_j\sigma_j$ denote the amount of memory that NeyAlloc would allocate to stratum i. We choose stratum ℓ where:

$$\ell = \arg\max_i \left\{ \frac{\partial V}{\partial s_i} \,\middle|\, 1 \le i \le r \right\} = \arg\min_i \left\{ \frac{n_i\sigma_i}{s_i} \,\middle|\, 1 \le i \le r \right\}$$

Lemma 4.1. *When required to reduce the size of a stratified random sample by one, the increase in variance of the estimated population mean is minimized if we reduce the size of S_ℓ by one, where $\ell = \arg\min_i \left\{ \frac{n_i\sigma_i}{s_i} \,\middle|\, 1 \le i \le r \right\}$.*

In the case where we have multiple choices for ℓ using Lemma 4.1, we choose the one where the current sample size s_ℓ is the largest. Algorithm SingleElementSSR for reducing the sample by a single element is a direct implementation of the condition stated in Lemma 4.1. We omit the pseudocode due to space constraints. It is straightforward to observe this can be done in time $O(r)$.

4.2 Reduction by $\beta \ge 1$ Elements

We now consider the general case, where the sample needs to be reduced by $\beta \ge 1$ elements. A possible solution idea is to repeatedly apply the one-element reduction algorithm (Algorithm SingleElementSSRfrom Section 4.1) β times. Each iteration, a single element is chosen from a stratum such that the overall variance increases by the smallest amount. However, this greedy approach may not yield a sample with the smallest variance. On the other hand, an exhaustive search of all possible evictions is not feasible either, since the number of possible ways to partition a reduction of size β among r strata is $\binom{\beta+r-1}{r-1}$, which can be very large. For instance, if $r = 10$, this is $\Theta(\beta^{10})$. We now present efficient approaches to VOR. We first present a recursive algorithm, followed by a faster iterative algorithm. Before presenting the algorithm, we present the following useful characterization of a variance-optimal reduction.

Definition 4.2. We say that stratum i is *oversized* under memory budget M, if its allocated sample size $s_i > M_i$. Otherwise, we say that stratum i is *not oversized.*

Lemma 4.3. *Suppose that E is the set of β elements that are to be evicted from a stratified random sample such that the variance V after eviction is the smallest possible. Then, each element in E must be from a stratum whose current sample size is oversized under the new memory budget $M' = M - \beta$.*

Proof. We use proof by contradiction. Suppose one of the evicted elements is deleted from a sample S_α such that the sample size s_α is not oversized under the new memory budget. Because the order of the eviction of the β elements does not impact the final variance, suppose that element e is evicted after the other $\beta - 1$ evictions have happened. Let s_α denote the size of sample S_α at the moment t right after the first $\beta - 1$ evictions and before evicting e. The increase in variance caused by evicting an element from S_α is

$$\begin{aligned}\Delta &= \frac{1}{n^2}\left(\frac{n_\alpha^2\sigma_\alpha^2}{s_\alpha(s_\alpha-1)}\right) = \left(\frac{\sum_{i=1}^{r} n_i\sigma_i}{nM'}\right)^2 \frac{M'^2_\alpha}{s_\alpha(s_\alpha-1)} \\ &> \left(\frac{\sum_{i=1}^{r} n_i\sigma_i}{nM'}\right)^2\end{aligned}$$

where $M'{}_\alpha = M'\frac{n_\alpha\sigma_\alpha}{\sum_{i=1}^{r} n_i\sigma_i}$. The last inequality is due to the fact that S_α is not oversized under budget M' at time t, i.e., $s_\alpha \le M'_\alpha$.

Note that an oversized sample exists at time t, since there are a total of $M' + 1$ elements in the stratified random sample at time t, and the memory target is M'. Instead of evicting e, if we choose to evict another element e' from an oversized sample $S_{\alpha'}$, the resulting increase in variance will be:

$$\begin{aligned}\Delta' &= \frac{1}{n^2}\left(\frac{n_{\alpha'}^2\sigma_{\alpha'}^2}{s_{\alpha'}(s_{\alpha'}-1)}\right) = \left(\frac{\sum_{i=1}^{r} n_i\sigma_i}{nM'}\right)^2 \frac{M'^2_{\alpha'}}{s_{\alpha'}(s_{\alpha'}-1)} \\ &< \left(\frac{\sum_{i=1}^{r} n_i\sigma_i}{nM'}\right)^2\end{aligned}$$

where $M'_{\alpha'} = M'\frac{n_{\alpha'}\sigma_{\alpha'}}{\sum_{i=1}^{r} n_i\sigma_i}$ The last inequality is due to the fact that $S_{\alpha'}$ is oversized under budget M' at time t, i.e., $s_{\alpha'} > M'_{\alpha'}$. Because $\Delta' < \Delta$, at time t, evicting e' from $S_{\alpha'}$ leads to a lower variance than evicting e from S_α. This is a contradiction to the assumption that evicting e leads to the smallest variance, and completes the proof. □

Algorithm 3: SSR($\mathcal{A}$, M, $\mathcal{L}$): Variance-Optimal Sample Size Reduction

```
Input: $\mathcal{A}$ – set of strata under consideration.
       $M$ – target sample size for all strata in $\mathcal{A}$.
Output: For $i \in \mathcal{A}$, $\mathcal{L}[i]$ is the final size of sample for
        stratum $i$.
$O \leftarrow \emptyset$ // oversized samples
for $j \in \mathcal{A}$ do
    $M_j \leftarrow M \cdot n_j\sigma_j / \sum_{t\in\mathcal{A}} n_t\sigma_t$ // Neyman allocation
        if memory $M$ divided among $\mathcal{A}$
    if $(s_j > M_j)$ then $O \leftarrow O \cup \{j\}$
    else $\mathcal{L}[j] \leftarrow s_j$ // Keep current allocation

if $O = \mathcal{A}$ then
    // All samples oversized. Recursion stops.
    for $j \in \mathcal{A}$ do $\mathcal{L}[j] \leftarrow M_j$
else
    // Recurse on $O$, under remaining mem budget.
    SSR($O, M - \sum_{j\in\mathcal{A}-O} s_j, \mathcal{L}$)
```

Lemma 4.3 implies that it is only necessary to reduce samples that are oversized under the target memory budget M'. Samples that are not oversized can be given their current allocation, even under the new memory target M'. Our algorithm based on this observation first allocates sizes to the samples that are not oversized. The remaining memory now needs to be allocated among the oversized samples. We note that this can again be viewed as a sample size reduction problem, while focusing on a smaller set of (oversized) samples, and accomplish it using a recursive call under a reduced memory budget; See Lemma 4.4 for a formal statement of this idea. The base case for this recursion is when all samples under consideration are oversized, in which case we can simply use NeyAlloc under the reduced memory budget M' (Observation 1). Our algorithm SSR is shown in Algorithm 3.

Let $\mathbb{S} = \{S_1, S_2, \ldots, S_r\}$ be the current stratified random sample. Let $\mathcal{A}$ denote the set of all strata under consideration, initialized to $\{1, 2, \ldots, r\}$. Let O denote the set of oversized samples, under target memory budget for $\mathbb{S}$, and $\mathcal{U} = \mathbb{S} - O$ denote the collection of samples that are not oversized. When the context is clear, we use O, $\mathcal{U}$, and $\mathcal{A}$ to refer to the set of stratum identifiers as well as the set of samples corresponding to these identifiers.

Table 1: An example of variance-optimal sample size reduction from 400×10^6 down to 200×10^6.

i	1	2	3	4	5	6
$n_i\sigma_i$ $(\times 10^9)$	10	8	30	20	8	24
s_i $(\times 10^6)$	15	50	50	45	60	180
round 1 M_i $(\times 10^6)$	20	< 50	60	< 45	< 60	< 180
round 2 M_i $(\times 10^6)$	-	< 50	-	45	< 60	< 180
round 3 M_i $(\times 10^6)$	-	18	-	-	18	54
s'_i $(\times 10^6)$	15	18	50	45	18	54

Lemma 4.4. *A variance-optimal eviction of β elements from $\mathbb{S}$ under memory budget M' requires a variance-optimal eviction of β elements from O under memory budget $M' - \sum_{j\in\mathcal{U}} s_j$.*

Proof. Recall that s'_i denotes the final size of sample S_i after β elements are evicted. Referring to the variance V from Equation 1, we know a variance-optimal sample size reduction of β elements from $\mathbb{S}$ under memory budget M' requires minimization of

$$\sum_{i\in\mathcal{A}} \frac{n_i^2\sigma_i^2}{s'_i} - \sum_{\in\mathcal{A}} \frac{n_i^2\sigma_i^2}{s_i} \tag{7}$$

By Lemma 4.3, we know $s_i = s'_i$ for all $i \in \mathcal{U}$. Hence, minimizing Formula 7 is equivalent to minimizing

$$\sum_{i=O} \frac{n_i^2\sigma_i^2}{s'_i} - \sum_{i\in O} \frac{n_i^2\sigma_i^2}{s_i} \tag{8}$$

The minimization of Formula 8 is exactly the result obtained from a variance-optimal sample size reduction of β elements from oversized samples under the new memory budget $M' - \sum_{i\in\mathcal{U}} s_i$. □

Observation 1. *In the case every sample in the stratified random sample is oversized under target memory M'*, i.e., *$\mathbb{S} = O$, the variance-optimal reduction is to reduce the size of each sample $S_i \in \mathbb{S}$ to M'_i under the new memory budget M'.*

The following theorem summarizes the correctness and time complexity of Algorithm SSR.

Theorem 4.5. *Algorithm 3 (*SSR*) finds a variance-optimal reduction of the stratified random sample $\mathcal{A}$ under new memory budget M. The worst-case time of* SSR *is $O(r^2)$, where r is the number of strata.*

Proof. Correctness follows from Lemmas 4.3–4.4 and Observation 1. The worst-case time happens when each recursive call sees only one stratum that is not oversized. In such a case, the time of all recursions of SSR on a stratified random sample across r strata is: $O(r + (r-1) + \ldots + 1) = O(r^2)$. □

An Example (Table 1). Suppose we have 6 strata with their statistics ($n_i\sigma_i$) and current sample sizes (s_i) showin in Table 1 using a total size of $\sum_{i=1}^{6} s_i = 400$. Suppose that we wish to reduce the sample size down to 200 by reducing each s_i to the target sample size s'_i. The computation involves a sequence of recursive rounds. In the initial round, we allocate 200 samples among all 6 strata using Neyman allocation. Strata 1 and 3 turn out to be not oversized ($M_1 \geq s_1$, $M_3 \geq s_3$), and therefore we set $s'_1 = s_1$ and $s'_3 = s_3$. In Round 2, we exclude strata 1 and 3 from consideration, and the available memory budget which now becomes $200 - 15 - 50 = 135$. This is allocated among strata 2, 4, 5, and 6 using Neyman allocation. Stratum 4 is not oversized ($M_4 \geq s_4$) and therefore we set $s'_4 = s_4$. At the next round 3, we further exclude stratum 4 from consideration, and the available memory budget now becomes $135 - 45 = 90$. When this is allocated among the remaining strata, it turns out that all of them are oversized ($M_i < s_i, i = 2, 5, 6$). We simply set $s'_i = M_i$ for each $i \in \{2, 5, 6\}$, and the recursion exits. Each stratum i now has a new sample size s'_i such that $s'_i \leq s_i$ for every i, and $\sum_{i=1}^{6} s'_i = 200$.

Algorithm 4: `MultiElementSSR`($\mathcal{A}, M$): A fast implementation of Sample Size Reduction without using recursion.

```
Input: The strata under consideration is $\mathcal{A} = \{1, 2, \ldots, r\}$,
       and the volumes and standard deviations. $M$ is the
       target total sample size.
Output: For $1 \leq i \leq r$, $\mathcal{L}[i]$ is set to the final size of sample
        for stratum $i$, such that the increase of the variance
        $V$ is minimized.
Allocate $\mathcal{L}[1..r]$, an array of numbers
Allocate $Q[1..r]$, an array of $(x, y, z)$ tuples
for $i = 1 \ldots r$ do $Q[i] \leftarrow (i, n_i\sigma_i, s_i/(n_i\sigma_i))$;
Sort array $Q$ in ascending order on the $z$ dimension
for $i = (r-1)$ down to 1 do
    $Q[i].y \leftarrow Q[i].y + Q[i+1].y$
$M_{new} \leftarrow M$; $D \leftarrow Q[1].y$
for $i = 1 \ldots r$ do
    $M_{Q[i].x} \leftarrow M \cdot n_{Q[i].x}\sigma_{Q[i].x}/D$
    if $s_{Q[i].x} > M_{Q[i].x}$ then break
    $\mathcal{L}[Q[i].x] \leftarrow s_{Q[i].x}]$
    $M_{new} \leftarrow M_{new} - s_{Q[i].x}$
    // Check the next sample, which must exist.
    $M_{Q[i+1].x} \leftarrow M \cdot n_{Q[i+1].x}\sigma_{Q[i+1].x}/D$
    if $s_{Q[i+1].x} > M_{Q[i+1].x}$ then // oversized
        $M \leftarrow M_{new}$; $D \leftarrow Q[i+1].y$
// Reduce sample size to target.
for $j = i..r$ do
    // Desired size for $S_{Q[j].x}$
    $\mathcal{L}[Q[j].x] \leftarrow M \cdot n_{Q[j].x}\sigma_{Q[j].x}/D$
return $\mathcal{L}$
```

Faster Sample Size Reduction. We present a faster algorithm for variance-optimal sample size reduction, `MultiElementSSR`, with time complexity $O(r \log r)$. `MultiElementSSR` shares the same algorithmic foundation as SSR, but uses a faster iterative method based on sorting. We omit proofs due to space constraints.

Theorem 4.6. *(1) The* `MultiElementSSR` *procedure in Algorithm 4 finds the correct size of each sample of a stratified random sample, whose memory budget is reduced to M, such that the increase of the variance V is minimized. (2) The worst-case time cost of*`MultiElementSSR` *on a stratified random sample across r strata is $O(r \log r)$.*

5 STREAMING SRS OVER A SLIDING WINDOW

We consider the maintenance of an SRS drawn from a *sequence-based* sliding window of the most recent elements from the stream. Given a window size W, the sliding window consists of the W

most recent elements observed in the data stream. We consider the case when the window size W is very large, so that it is not feasible to store the entire window in memory. Similar to the algorithm for infinite window, there are two parts to the algorithm, sample re-allocation and sampling, which are interleaved with each other. We provide the algorithm idea and omit detailed descriptions.

For re-allocating sample sizes, we need the current statistics of each stratum within the sliding window. The mean and variance of a given stratum in an infinite window can be maintained in $O(1)$ space easily in a single pass. However, maintaining the mean and variance over a sliding window is much harder. In fact, it is known that exact computation of the mean and variance over a sliding window requires memory linear in the stream size [21] – thus, if we require these statistics exactly, we have to store the entire window, just to maintain the statistics of different strata! Fortunately, it is possible to approximate these statistics using space poly-logarithmic in the size of the stream; for the mean, see [21, 27], and for the variance [43].

Random sampling over a sliding window is also quite different from the case of infinite windows, and there is significant prior work on this e.g. [9, 13, 24]. We adapt algorithms from prior work to assign to each arriving element a random key, chosen uniformly in $[0, 1]$. The random sample of a certain size within a stratum is defined to be those elements in the stratum that have the smallest keys. Borrowing from prior work [9], we maintain additional recent elements within the window even if they don't belong to the set of keys that are currently the smallest – the reason is that these elements may become the elements with the smallest keys once the window slides and other elements with smaller keys "expire" from the window. The additional space required by these keys is a logarithmic factor in the size of the window (Section 2 in [9]). For each stratum, the algorithm continuously monitors the smallest key that has been discarded from the window.

When new elements arrive in the stream, these are sampled into the SRS, which may cause the size of the sample to increase beyond the memory allocated to the stratum. When this happens, we rely on variance-optimal sample size reduction (Algorithm `MultiElementSSR`) to give us new sample size allocations to different strata, and different strata are sub-sampled according to the new allocations (sub-sampling within a given stratum is handled through selecting only the elements with the smallest keys that are active in the window).

6 VOILA: VARIANCE-OPTIMAL OFFLINE SRS

We now present an algorithm for computing the variance-optimal allocation of sample sizes in the general case when there may be strata that are bounded. Note that once the allocation of sample sizes is determined, the actual sampling step is straightforward for the offline algorithm – samples can be chosen in a second pass through the data, using reservoir sampling within each stratum. Hence, in the rest of this section, we focus on determining the variance-optimal allocation. Consider a static data set R of n elements across r strata, where stratum i has n_i elements, and has standard deviation σ_i. *How can a memory budget of M elements be partitioned among the strata in a variance-optimal manner?* We present VOILA (**V**ariance-**O**pt**I**ma**L** **A**llocation), an efficient offline algorithm for variance-optimal allocation that can handle strata that are bounded.

Algorithm 5: VOILA (M): Variance-optimal stratified random sampling for bounded data

```
Input: $M$ is the memory target
for $i = 1 \ldots r$ do
    $s_i \leftarrow n_i$ // assume total memory of $n$
$\mathcal{L} \leftarrow$ MultiElementSSR($M$)
return $\mathcal{L}$    /* $\mathcal{L}[i] \le n_i$ is the sample size for
    stratum $i$ in a variance-optimal stratified
    random sample. */
```

Neyman Allocation assumes there are no bounded strata (strata with small volumes). Note that it is not possible to simply eliminate strata with a low volume, by giving them full allocation, and then apply Neyman allocation on the remaining strata. The reason is as follows: suppose bounded strata are removed from further consideration. Then, remaining memory is divided among the remaining strata. This may lead to further bounded strata (which may not have been bounded earlier), and Neyman allocation again does not apply.

The following two-step process reduces variance-optimal offline SRS to variance-optimal sample size reduction.

Step 1: Suppose we start with a memory budget of n, sufficient to store all data. Then, we will just save the whole data set in the stratified random sample, and thus each sample size $s_i = n_i$. By doing so, the variance V is minimized, since $V = 0$ (Equation 1).

Step 2: Given the stratified random sample from Step 1, we reduce the memory budget from n to M such that the resulting variance is the smallest. This can be done using variance-optimal sample size reduction, by calling `SSR` or `MultiElementSSR` with target sample size M.

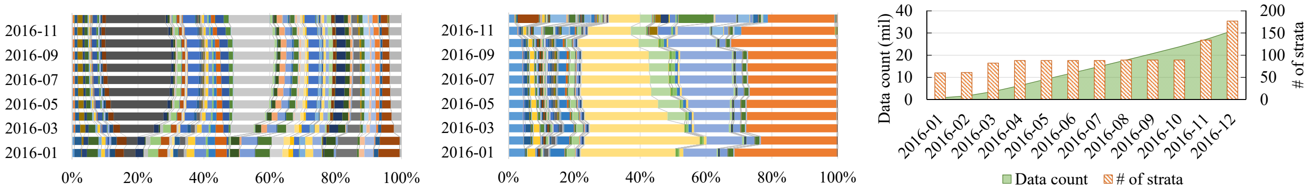
VOILA (Algorithm 5) simulates this process. The algorithm only records the sample sizes of the strata in array $\mathcal{L}$, without creating the actual samples. The actual sample from stratum i is created by choosing $\mathcal{L}[i]$ elements from stratum i, using a method for uniform random sampling without replacement.

Theorem 6.1. *Given a data set $\mathcal{R}$ with r strata, and a memory budget M,* VOILA *(Algorithm 5) returns in $\mathcal{L}$ the sample size of each stratum in a variance-optimal stratified random sample. The worst-case time cost of* VOILA *is $O(r \log r)$.*

Proof. The correctness follows from the correctness of Theorem 4.6, since the final sample is the sample of the smallest variance that one could obtain by reducing the initial sample (with zero variance) down to a target memory of size M. The run time is dominated by the call to `MultiElementSSR`, whose time complexity is $O(r \log r)$. □

7 EXPERIMENTAL EVALUATION

We present the results of an experimental evaluation. The input for our experiment is a (finite) stream of records from a data source, which is either processed by a streaming algorithm or by an offline algorithm at the end of computation. A streaming sampler must process data in a single pass using limited memory. An offline sampler has access to all data received, and can compute a stratified random sample using multiple passes through data. We evaluate the samplers in two ways. The first is a direct evaluation of the sample quality through the resulting allocation and the variance of estimates obtained using the samples. The second is through the accuracy of approximate query processing using the maintained samples for different queries.



(a) Relative (cumulative) frequencies of different strata. The x-axis is the fraction of points observed so far.

(b) Relative (cumulative) standard deviations of different strata.

(c) The number of strata received so far, and the number of records in data.

Figure 1: Characteristics of the OpenAQ dataset.

7.1 Sampling Methods

We compared our stream sampling method S-VOILA to Reservoir, ASRS and Senate sampling. Reservoir is a well-known stream sampling method that maintains a uniform random sample chosen without replacement from the stream - we expect the number of samples allocated to stratum *i* by Reservoir to be proportional to n_i. Senate [1] is a stratified sampling method that allocates each stratum an equal amount of sample space. For each stratum, Reservoir sampling is used to maintain a uniform sample.

ASRS is an adaptive stratified sampling algorithm due to Al-kateb *et al.* (Algorithm 3 in [5]). Their algorithm considers reallocations of memory among strata using a different method, based on power allocation [10], followed by reservoir sampling within each stratum. We chose the power allocation parameter to be 1 in order to obtain a sample of the entire population.

We also implemented three offline samplers VOILA, NeyAlloc, and an offline version of Senate. Each uses two passes to compute a stratified random sample of a total size of *M* records. The first pass is to determine strata characteristics used to allocate the space between strata. The second pass is to collect the samples accordingly to the computed allocation.

7.2 Data

We used a real-world dataset called OpenAQ [37], which contains more than 31 million records of air quality measurements (concentrations of different gases and particulate matter) from 7, 923 locations in 62 countries around the world in 2016. Data is replayed in time order to generate the stream and is stratified based on the country of origin and the type of measurement, e.g., all measurements of carbon monoxide in the USA belong to one stratum, all records of sulphur dioxide in India belong to another stratum, and so on. The total number of strata at different points in time are shown in Figure 1c. We also experimented with another method of stratifying data, based only on the city of origin, whose results are shown at the end of this section. We also experimented with a synthetic dataset. The results obtained were qualitatively similar to the real-world data, and we omit these results due to space constraints.

Each stratum begins with zero records, and in the initial stages, every stratum is bounded. As more data are observed, many of the strata are not bounded anymore. As Figure 1c shows, new strata are added as more sensors are incorporated into the data stream. Figures 1a and 1b respectively show the cumulative frequency and standard deviation of the data over time; clearly these change significantly with time. As a result, the variance-optimal sample-size allocations to strata also change over time, and a streaming algorithm needs to adapt to these changes.

7.3 Allocations of Samples to Strata

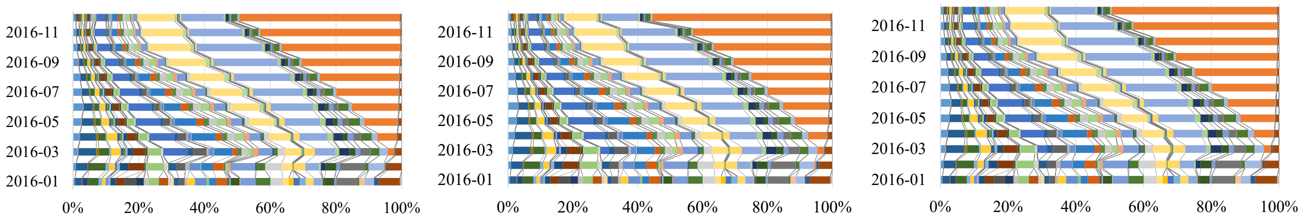
We measured the allocation of samples to different strata. Unless otherwise specified, the sample size *M* is set to 1 million records. For all experiments on allocations or variance, each data point is the mean of five independent runs. The allocation can be seen as a vector of numbers that sum up to *M* (or equivalently, normalized to sum up to 1), and we observe how this vector changes as more elements arrive.

Figures 2a, 2b and 2c show the change in allocations over time resulting from VOILA, S-VOILA with single element processing, and S-VOILA with minibatch processing. Unless otherwise specified, in the following discussion, the size of a minibatch is set to equal one day's worth of data. Visually, the allocations produced by the three methods track each other over time, showing that the streaming methods follow the allocation of VOILA. To understand the difference between the allocations due to VOILA and S-VOILA quantitatively, we measured the cosine distance between the allocation vectors from VOILA and S-VOILA. While detailed results are omitted due to space constraints, our results show that allocation vectors due to S-VOILA and VOILA are very similar, and the cosine distance is close to 0 most of the time and less than 0.04 at all times.

7.4 Comparison of Variance

We compared the variance of the estimates (Equation 1) produced by different methods. The results are shown in Figures 3 and 4. Generally, the variance of the sample due to each method increases over time, since the volume of data and the number of strata increase, while the sample size is fixed.

The comparison of different streaming algorithms is shown in Figure 4. Among the streaming algorithms, we first note that both variants of S-VOILA have a variance that is lower than ASRS, and typically close to the optimal (VOILA). The variance of S-VOILA with minibatch processing is typically better than with single element processing. We note that the variances of both variants of S-VOILA are nearly equal to that of VOILA until March, when they start increasing relative to VOILA, and then converge back. From analyzing the underlying data, we see that March is the time when a number of new strata appear in the data (Figure 1c), causing a substantial change in the optimal allocation of samples to strata. An offline algorithm such as VOILA can resample more elements at will, since it has access to all earlier data from the stratum. However, a streaming algorithm such as S-VOILA cannot do so and must wait for enough new elements to arrive in these strata before it can "catch up" to the allocation of VOILA. Hence, S-VOILA with single element as well as with minibatch processing show an increasing trend in the variance at such a point. When data becomes stable again the relative



(a) VOILA (b) S-VOILA with single element (c) S-VOILA with minibatch of size one day

Figure 2: Change in allocation over time. OpenAQ data.

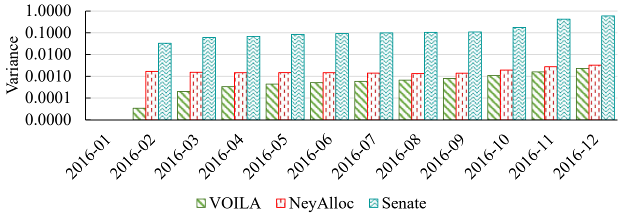


Figure 3: Variance of VOILA compared to NeyAlloc and Senate. Sample size: 1M records, OpenAQ data.

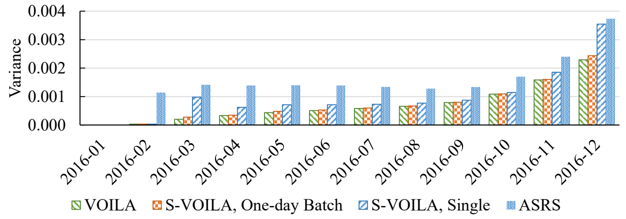


Figure 4: Variance of S-VOILA compared with ASRS and offline VOILA. Sample size 1M records, OpenAQ data.

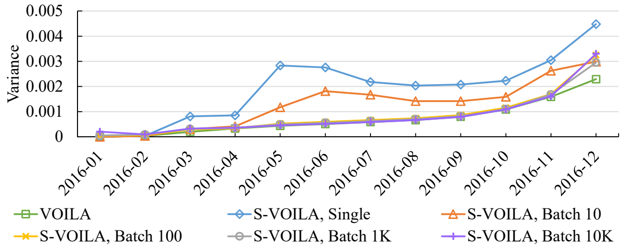


Figure 5: Impact of Minibatch Size on Variance, OpenAQ.

performance of S-VOILA improves. In November and December, new strata appear again, and the relative performance is again affected.

Among offline algorithms, we observe from Figure 3 that Senate performs poorly, since it blindly allocates equal space to all strata. NeyAlloc results in a variance that is larger than VOILA, by a factor of 1.4x to 50x. While NeyAlloc is known to be variance-optimal under the assumption of having all strata being abundant, these results show that it is far from variance-optimal for bounded strata.

Impact of Sample Size: To observe the sensitivity to the sample size, we conducted an experiment where the sample size is varied from 5000 to 1 million. We fixed the minibatch size to 100 thousand records. As expected, in both S-VOILA and VOILA, with single element and minibatch processing, the variance decreases when the sample size increases. The general trend was that the variance decreased by approximately a factor of 10 when the sample size increased by a factor of 10. We omit detailed results due to space constraints.

Impact of Minibatch Size: We further conducted an experiment where the minibatch size is chosen from $\{1, 10, 10^2, 10^3, 10^4\}$. The results are shown in Figure 5. A minibatch size of 10 elements yields significantly better results than single element S-VOILA. A minibatch size of 100 or greater makes the variance of S-VOILA nearly equal to the optimal variance.

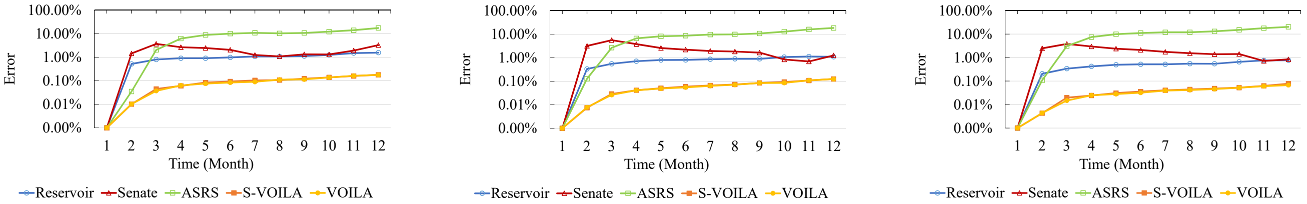
7.5 Query Performance, Infinite Window

We now evaluate the quality of these samples indirectly, through their use in approximate query processing. Samples constructed using S-VOILA and VOILA are used to approximately answer a variety of queries on the data so far. For evaluating the approximation error, we also implement an exact (but expensive) method for query processing Exact that stores all records in a MySQL database. Identical queries are made at the same time points in the stream to the different streaming and offline samplers, as well as to the exact query processor.

A range of queries are used. Each query *selects a subset of data through a selection predicate supplied at query time, and applies an aggregate.* This shows the flexibility of the sample, since it does not have any a priori knowledge of the selection predicate. We have chosen predicates with selectivity equal to one of 0.25, 0.50, and 1.00. We consider four aggregation functions: SUM, the sum of elements; SSQ, the sum of squares of elements; AVG, the mean of elements; and STD, the standard deviation. Each data point is the mean of five repetitions of the experiment with the same configuration. Each query was executed over all received data after one month of data arrived, up to entire year of 2016 in the OpenAQ dataset with thirty-one million records.

Figures 6 and 7 show the relative errors of different aggregations as the size of streaming data increases, while the sample size is held fixed. Both figures show that S-VOILA outperforms other streaming samplers across queries with different aggregation and selectivity. This result shows that S-VOILA maintains a better quality of stratified sample to answer an aggregation over a subset of data accurately. In addition, S-VOILA performs very closely to its offline version, VOILA, which samples from the entire received data. We note that when ASRS evicts elements from per-stratum samples, there may not always be new elements to take their place, hence it often does not use its full quota of allocated memory.

Alternate Methods of Stratification. We also experimented with the OpenAQ data set stratified in a different manner, using the city where the observation was made. Sample results are shown in Figure 8. We still see that S-VOILA outperforms

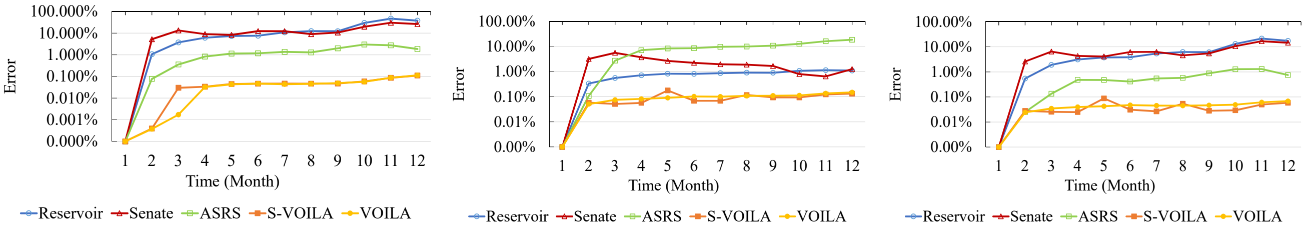


(a) Selectivity=0.25

(b) Selectivity=0.50

(c) Selectivity=1.00

Figure 6: Streaming samplers. SUM with different selectivities, sample size = 1 million. OpenAQ data.



(a) SSQ

(b) AVG

(c) STD

Figure 7: Streaming samplers. SSQ, AVG, and STD with selectivity 0.50, sample size = 1 million. OpenAQ data.

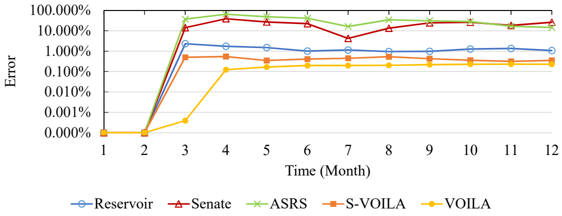


Figure 8: Streaming samplers, data stratified by the city (SUM with selectivity 0.5)

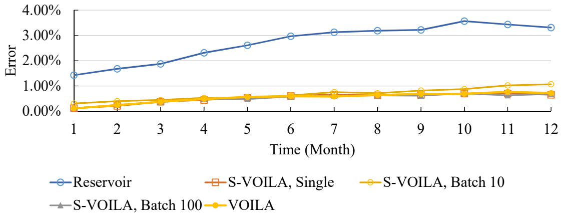


Figure 9: Streaming samplers, impact of minibatch size, sample size = 100,000. (SUM with selectivity 0.5)

Reservoir, Senate, and ASRS. This supports our observation that the sample maintained by S-VOILA is of a higher quality than other streaming samplers, no matter how data is stratified.

Impact of Sample Size. We also explored different sample sizes varied from 500, 000 to 1 million. All methods benefit from increased sample size and the relative performance between different methods remains the same across different sizes.

Impact of Minibatch Size. Figure 9 shows the impact of the minibatch size on the accuracy of streaming samplers for the SUM query with selectivity 0.5. The sample size is set to one hundred thousand for each sampler. S-VOILA with different minibatch sizes has an error less that 1%, often much smaller, while Reservoir has an error that is often 3% or larger. In addition, we observe that S-VOILA with different minibatch sizes is very close to VOILA.

7.6 Sliding Window Streaming

We experimented with streaming algorithms Reservoir and S-VOILA with a sliding window of size $W = 10^6$. The version of Reservoir that was used here maintains a uniform sample over the window by sampling each record with the same selection probability of $\frac{M}{W}$, so it may be more accurately termed "Bernoulli sampling". S-VOILA uses stratified sampling with single element processing, as described in Section 5. As the window slides, we periodically ask for sum of the *value* attribute in the current window. We report the error by compare the estimates from samples with the ground-truth answer. Figure 10 shows the average errors of 5 runs. With a 10% sample rate, as expected, S-VOILA provide an answer with less than 1% error, while Reservoir has an error of about 2-3%.

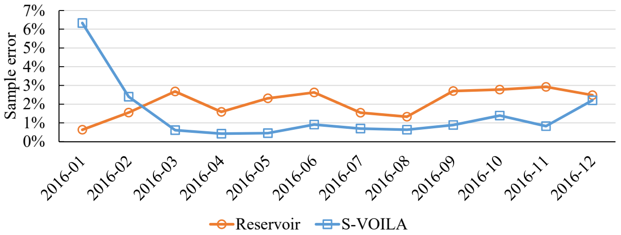
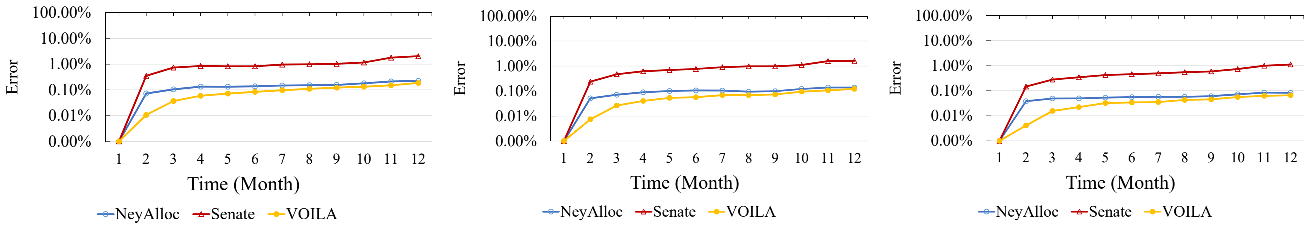


Figure 10: Sample error of window sum query, streaming data with sliding window of 10^6 and sample size of 10^5 records, OpenAQ data.

7.7 Offline Sampling

We also compared VOILA with other offline samplers for the SUM query with different selectivities. Figure 11 shows that VOILA always has better performance than Senate and NeyAlloc. Our experiments with other aggregations also showed similar results.



(a) Selectivity=0.25

(b) Selectivity=0.50

(c) Selectivity=1.00

Figure 11: Offline samplers. SUM with different selectivities, sample size = 1 million. OpenAQ data.

8 CONCLUSIONS

We presented S-VOILA, an algorithm for streaming SRS with minibatch processing, which interleaves a continuous, locally variance-optimal re-allocation of sample sizes with streaming sampling. Our experiments show that S-VOILA results in variance that is typically close to VOILA, which was given the entire input beforehand, and which is much smaller than that of algorithms due to prior work. We also show an inherent lower bound on the worst-case variance of any streaming algorithm for SRS – this limitation is not due to the inability to compute the optimal sample allocation in a streaming manner, but is instead due to the inability to increase sample sizes in a streaming manner, while maintaining uniformly weighted sampling within a stratum. Our work also led to a variance-optimal method VOILA for offline SRS from data that may have bounded strata. Our experiments show that on real and synthetic data, an SRS obtained using VOILA can have a significantly smaller variance than one obtained by Neyman allocation.

There are several directions for future research, including (1) restratification in a streaming manner, (2) handling group-by queries and join queries, (3) incorporating general versions of time-decay, and (4) SRS on distributed data.

Acknowlegdment: Nguyen and Tirthapura were supported in part by NSF grants 1527541 and 1725702.

REFERENCES

[1] S. Acharya, P. Gibbons, and V. Poosala. 2000. Congressional Samples for Approximate Answering of Group-by Queries. In *Proc. SIGMOD.* 487–498.

[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. 1999. The Aqua Approximate Query Answering System. In *Proc. SIGMOD.* 574–576.

[3] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proc. EuroSys.* 29–42.

[4] M. Al-Kateb and B. S. Lee. 2010. Stratified Reservoir Sampling over Heterogeneous Data Streams. In *Proc. SSDBM.* 621–639.

[5] M. Al-Kateb and B. S. Lee. 2014. Adaptive stratified reservoir sampling over heterogeneous data streams. *Information Systems* 39 (2014), 199–216.

[6] M. Al-Kateb, B. S. Lee, and X. S. Wang. 2007. Adaptive-Size Reservoir Sampling over Data Streams. In *Proc. SSDBM.* 22.

[7] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. 2002. Models and Issues in Data Stream Systems. In *Proc. PODS.* 1–16.

[8] B. Babcock, S. Chaudhuri, and G. Das. 2003. Dynamic Sample Selection for Approximate Query Processing. In *Proc. SIGMOD.* 539–550.

[9] B. Babcock, M. Datar, and R. Motwani. 2002. Sampling from a Moving Window over Streaming Data. In *SODA.*

[10] M. D Bankier. 1988. Power allocations: determining sample sizes for subnational areas. *The American Statistician* 42, 3 (1988), 174–177.

[11] V. Braverman, R. Ostrovsky, and G. Vorsanger. 2015. Weighted Sampling Without Replacement from Data Streams. *Inf. Process. Lett.* 115, 12 (2015), 923–926.

[12] V. Braverman, R. Ostrovsky, and C. Zaniolo. 2009. Optimal Sampling from Sliding Windows. In *Proc. PODS.* 147–156.

[13] V. Braverman, R. Ostrovsky, and C. Zaniolo. 2009. Optimal Sampling from Sliding Windows. In *PODS.*

[14] S. Chaudhuri, G. Das, and V. Narasayya. 2007. Optimized Stratified Sampling for Approximate Query Processing. *ACM TODS* 32, 2 (2007).

[15] Y. Chung and S. Tirthapura. 2015. Distinct Random Sampling from a Distributed Stream. In *IPDPS.* 532–541.

[16] Y. Chung, S. Tirthapura, and D. Woodruff. 2016. A Simple Message-Optimal Algorithm for Random Sampling from a Distributed Stream. *IEEE TKDE* 28, 6 (2016), 1356–1368.

[17] W. G. Cochran. 1977. *Sampling Techniques* (third ed.). John Wiley & Sons, New York.

[18] G. Cormode, S. Muthukrishnan, K. Yi, and Q. Zhang. 2012. Continuous Sampling from Distributed Streams. *JACM* 59, 2 (2012).

[19] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu. 2009. Forward Decay: A Practical Time Decay Model for Streaming Systems. In *Proc. ICDE.* 138–149.

[20] G. Cormode, S. Tirthapura, and B. Xu. 2009. Time-decaying Sketches for Robust Aggregation of Sensor Data. *SIAM J. Comput.* 39, 4 (2009), 1309–1339.

[21] M. Datar, A. Gionis, P. Indyk, and R. Motwani. 2002. Maintaining stream statistics over sliding windows. *SIAM J. Comput.* 31, 6 (2002), 1794–1813.

[22] P. S. Efraimidis and P. G. Spirakis. 2006. Weighted Random Sampling with a Reservoir. *Inf. Process. Lett.* 97, 5 (2006), 181–185.

[23] R. Gemulla and W. Lehner. 2008. Sampling Time-based Sliding Windows in Bounded Space. In *Proc. SIGMOD.* 379–392.

[24] R. Gemulla and W. Lehner. 2008. Sampling Time-based Sliding Windows in Bounded Space. In *SIGMOD.*

[25] R. Gemulla, W. Lehner, and P. J. Haas. 2008. Maintaining Bounded-size Sample Synopses of Evolving Datasets. *The VLDB Journal* 17, 2 (2008), 173–201.

[26] P. B. Gibbons and S. Tirthapura. 2001. Estimating Simple Functions on the Union of Data Streams. In *Proc. SPAA.* 281–291.

[27] P. B. Gibbons and S. Tirthapura. 2002. Distributed streams algorithms for sliding windows. In *SPAA.* 63–72.

[28] P. J. Haas. 2016. Data-Stream Sampling: Basic Techniques and Results. In *Data Stream Management.* Springer, 13–44.

[29] T. Johnson and V. Shkapenyuk. 2015. Data Stream Warehousing In Tidalrace. In *Proc. CIDR.*

[30] S. Joshi and C. Jermaine. 2008. Robust Stratified Sampling Plans for Low Selectivity Queries. In *Proc. ICDE.* 199–208.

[31] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *SIGMOD.* 631–646.

[32] K. Lang, E. Liberty, and K. Shmakov. 2016. Stratified Sampling Meets Machine Learning. In *Proc. ICML.* 2320–2329.

[33] S. L. Lohr. 2009. *Sampling: Design and Analysis* (2nd ed.). Duxbury Press.

[34] I. Mcleod and D. Bellhouse. 1983. A Convenient Algorithm for Drawing a Simple Random Sample. *Journal of the Royal Statistical Society. Series C. Applied Statistics* 32 (1983), 182–184.

[35] X. Meng. 2013. Scalable Simple Random Sampling and Stratified Sampling. In *Proc. ICML.* 531–539.

[36] J. Neyman. 1934. On the Two Different Aspects of the Representative Method: The Method of Stratified Sampling and the Method of Purposive Selection. *Journal of the Royal Statistical Society* 97, 4 (1934), 558–625.

[37] OpenAQ [n. d.]. http://openaq.org. ([n. d.]).

[38] S. K. Thompson. 2012. *Sampling* (3rd ed.). Wiley.

[39] Y. Tillé. 2006. *Sampling Algorithms* (1st ed.). Springer-Verlag.

[40] S. Tirthapura and D. P Woodruff. 2011. Optimal random sampling from distributed streams revisited. In *DISC.* 283–297.

[41] J. S. Vitter. 1983. Optimum Algorithms for Two Random Sampling Problems. In *Proc. FOCS.* 65–75.

[42] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP.* 423–438.

[43] L. Zhang and Y. Guan. 2007. Variance estimation over sliding windows. In *PODS.* 225–232.

Boosting SimRank with Semantics

Tova Milo, Amit Somech and Brit Youngmann
Tel Aviv University
{milo,amitsome,brity}@post.tau.ac.il

ABSTRACT

The problem of estimating the similarity of a pair of nodes in an information network draws extensive interest in numerous fields, e.g., social networks and recommender systems. In this work we revisit SimRank, a popular and well studied similarity measure for information networks, that quantifies the similarity of two nodes based on the similarity of their neighbors. SimRank's popularity stems from its simple, declarative definition and its efficient, scalable computation. However, despite its wide adaptation, it has been observed that for many applications SimRank may yield inaccurate similarity estimations, due to the fact that it focuses on the *network structure* and ignores the *semantics* conveyed in the node/edge labels. Therefore, the question that we ask is *can SimRank be enriched with semantics while preserving its advantages?*

We answer the question positively and present SemSim, a modular variant of SimRank that allows to inject into the computation any semantic similarly measure, which satisfies three natural conditions. The probabilistic framework that we develop for SemSim is anchored in a careful modification of SimRank's underlying random surfer model. It employs Importance Sampling along with a novel pruning technique, based on unique properties of SemSim. Our framework yields execution times essentially on par with the (semantic-less) SimRank, while maintaining negligible error rate, and facilitates direct adaptation of existing SimRank optimizations. Our experiments demonstrate the robustness of SemSim, even compared to task-dedicated measures.

1 INTRODUCTION

Estimating node similarity in information networks is the cornerstone of many applications, e.g., retrieving similar users in social networks, and a fundamental component in numerous network analysis algorithms, such as link prediction and clustering.

In this work we consider SimRank [13], a well-studied similarity measure for information networks. The intuition behind SimRank is that similar objects are referenced by similar objects, and thus it quantifies node similarity based on the compound similarity of their neighbors. SimRank's popularity stems from its simple declarative definition and its efficient computation, incorporating a broad range of optimizations [15, 39]. However, despite its wide adaptation, it has been observed that for many applications SimRank may yield inaccurate estimations [37, 40], as it focuses solely on the *network structure* and ignores the *semantic information* conveyed in the node/edge labels. Thus, the question we address is the following:

Can SimRank be enriched with semantics while preserving its intuitive, declarative definition and efficient computation?

We answer the question positively, and present SemSim, a refined variant of SimRank, that weights nodes' structural similarity with their semantic similarity and edge weights, yielding an effective, comprehensive measure. SemSim's probabilistic framework, anchored in a careful modification of SimRank's underlying random surfer model, together with dedicated optimizations, allows for execution times essentially on par with the semantic-less SimRank, while maintaining similar negligible error rates. Direct adaptation of existing SimRank optimizations is also enabled.

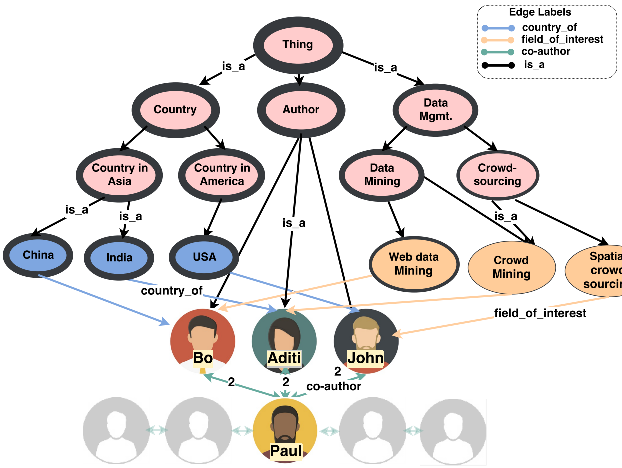


Figure 1: Example information network.

We demonstrate the problem that we tackle with an illustrative example.

Example 1.1. The simple information network depicted in Figure 1 represents a bibliographic database. It includes nodes describing authors, countries and research fields, with edges linking authors to their co-authors, country of origin and fields of interest. A semantic taxonomy is also reflected in this network (pink nodes) where entities are linked to their hypernyms, as indicated by the "is-a" edges. Edge weights reflect the strength of the relations (for conciseness, some weights are omitted but should be assumed to have an identical arbitrary default value). To visually represent the prevalence of a concept in the dataset, we use the width of the borders surrounding the nodes (an explanation of this quantification is provided in the sequel).

We wish to determine which of the authors, Bo or John, is more similar to Aditi. Observe that: (1) all three collaborated with Paul twice (as indicated by the edge weights); (2) their origin countries are all highly prevalent (as indicated by the borders' width), compared to the authors' fields of interest, thus the latter is more *informative* and should have a greater effect on the similarity[1]; (3) Crowdsourcing, the common field of John and Aditi, is more particular (less prevalent) than Data Mining, the field shared by Bo and Aditi. Hence, intuitively, Crowd Mining is semantically closer to Spatial Crowdsourcing than to Web Data Mining. Consequently, John is more similar to Aditi than Bo, even though they reside in different continents. Note, however, that ignoring semantics and considering the *network structure* alone (even including edge weights), Bo and Aditi seem more similar,

[1] Following standard argumentation, an estimation of similarity increases more drastically when indicated by a less frequent event [32].

and indeed, SimRank (like other measures [37, 43]) erroneously derive a higher similarity score for them.

Several refinements for SimRank have been proposed in the literature (see Related Work). For instance, SimRank++ [2] is a variant of SimRank that also considers edge weights, yet semantics is ignored, and, importantly, scalability is not addressed. Other works (e.g. [37]) partially account for semantics by considering only meaningful *meta-paths* (i.e. paths of a specific label patterns) between objects. But as can be seen in the above example (and in our experiments) this does not always suffice to accurately differentiate objects. Alternatively, several semantic measures have been proposed [23, 32], but they typically gauge the similarity based on ontological information and the *Information Content* (IC) of nodes, while the rest of the network structure is ignored. In an attempt to fully account for both structure and semantics, recent works abandon SimRank and rely instead on representation learning, using techniques such as node embedding [4, 30]. While this approach often outperforms a naive combination of structural and semantic similarity measures, a key drawback is that results are hard to explain and interpret, as is often the case with machine learning. Interestingly, we show that our SimRank variant not only retains its intuitive, declarative flavor but also yields more accurate estimations compared to these works.

Next, we provide a brief overview of our results.

The SemSim *similarity measure.* We refine SimRank by weighting nodes' neighbors similarity with their semantic similarity and edge weights. Our definition is modular and allows to inject into the computation any semantic measure, as long as it satisfies three intuitive conditions that are typically satisfied by existing measures. We present SemSim iterative formulation, analogous to SimRank iterative formulation [13]. We prove that SemSim's solution always exists (as was shown for SimRank), and show that its iterative formula converges to its fix-point at least as fast as SimRank, and possibly faster, due to an additional semantic factor (Section 2).

Random-surfer model. SimRank's underlying random-surfer model serves as the basis for many of its optimizations. We establish a corresponding model for SemSim. First, we define the notion of a *Semantic-Aware Random Walk* (SARW), which refines the traditional random walk definition, and prove that SemSim can be computed using SARWs. This interpretation considers the *node-pair graph* G^2, in which a node represents an ordered pair of nodes from the original graph G. Interestingly, we prove that given a threshold s.t. only similarity scores above it are of interest, the semantics can effectively be used to reduce the size of G^2, with the computation of SemSim over the reduced graph yielding the same results as those computed via the full graph G^2. Our experiments demonstrate that a significant reduction of up to three orders of magnitude is achieved in multiple datasets (Section 3).

Approximated similarity scores. Exact computation may still be expensive for large graphs, despite the speed-up gained by the graph reduction. For SimRank, the excessive size of G^2 motivated a battery of optimizations based on a Monte-Carlo (MC) procedure [15, 34, 39]. SimRank basic MC framework returns an approximated SimRank score in $O(n_w \cdot t)$ time, where n_w is the number of walks sampled from each node and t is a bound on their length. To efficiently approximate SemSim we develop an analogous MC framework, thereby enabling a direct application of SimRank optimizations. First, we show that a naïve solution of simply replacing SimRank's underlying uniform distribution with the semantic-aware distribution leads to a quadratic increase of the sample size. To overcome this, we employ *Importance Sampling*, and devise an unbiased estimator for SemSim, which avoids this increase and returns the estimated SemSim score in average time of $O(n_w \cdot t \cdot d^2)$, where d is the average in-degree in the graph G. To further reduce computation cost, we devise a dedicated pruning technique that avoids the computations of unpromising node-pairs and irrelevant (low probability) walks, at the cost of a controlled additive error to the approximated scores. While the worst-case time complexity remains the same, our experiments show pruning to be extremely effective in practice, yielding running times on par with SimRank (Section 4).

Experimental study. We conduct an experimental study over real data, demonstrating the effectiveness of SemSim in multiple practical scenarios. Our results demonstrate the robust quality of SemSim, even compared to task-dedicated measures. The results further exhibit the efficiency and accuracy of our framework and its ability to boost SimRank with semantics while preserving its performance (Section 5).

Finally, related work and conclusions are presented in sections 6 and 7, respectively. For space constraints, proofs are deferred to a technical report [27].

2 PRELIMINARIES

We first explain the data model used in our setting, then present our novel measure, SemSim.

2.1 Data Model

Following [36], we refer to the objects graph as a *Heterogeneous Information Network* (HIN), a flexible graph model that can capture and integrate various types of data. Let $\mathcal{V}$ be the domain of vertices, $\mathcal{L}$ the domain of labels, and $\mathcal{R}^+$ the domain of real positive numbers > 0.

Definition 2.1 (Heterogeneous Information Network). A HIN is a directed weighted graph $G = (V, E, \phi, \psi, W)$, where: $V \subseteq \mathcal{V}$ is a finite set of vertices; $E \subseteq V \times V$ is a set of edges; $\phi : V \longrightarrow \mathcal{L}$ and $\psi : E \longrightarrow \mathcal{L}$ are vertex and edge labeling functions, resp., and $W : E \longrightarrow \mathcal{R}^+$ is an edge weight function.

The edge weight function W associates each edge with a real positive number indicating the strength of the relation. When no knowledge about this strength is available, the weight is set to a default value. For example, in Figure 1, weights are available only for edges with the label *co-author*, where the weights reflect the number of collaborations between authors. Since no information about other weights is available, all other weights were set to 1. For a node v, we denote by $I(v), O(v)$ the set of in and out neighbors of v, resp. An individual in-neighbor is denoted as $I_i(v)$, for $1 \leq i \leq |I(v)|$, if $I(v) \neq \emptyset$ ($O_i(v)$, resp). Throughout the paper we use the variables u, v, u', v' to denote nodes in V. Here we consider directed graphs, but stress that all results can be adapted to the undirected model with minor modifications.

In many cases, the HIN is composed of two subgraphs that are aligned together: one consists of individual objects and their relations, e.g. authors/countries, and their collaboration/residence relationships. The second, ontological-style subgraph is comprises of semantic categories and relationships, e.g. the pink nodes and their "is-a" relations. Objects of the former can be connected to their corresponding categories. For example, in Figure 1 all author nodes are connected to the category *Author*. When semantic information is not included, one can enrich the graph by aligning it with publicly available ontology [5, 26] by applying existing entity alignment tools [28]. Ontologies typically contain a hierarchical taxonomy of concepts, e.g., that *USA* "is-a" a *Country in America* and *Country in America* "is-a" *Country*. Such taxonomies are often leveraged to define semantic similarity measures.

2.2 Similarity Notions

As described above, our semantic-rich graph model contains various types of linked entities, as well as additional knowledge that is captured by the edge weights. Next, we devise a refined version of SimRank [13] that effectively considers all information. We start with a short background on SimRank, then provide the formal definition for SemSim.

SimRank follows the intuitive assumption that: "two nodes are similar if they are related to similar nodes". Formally, given two nodes $u, v \in V$, their SimRank score is defined as follows. If $u = v$ then $simrank(u, v) = 1$, else: $simrank(u, v)$ is given by the following recursive formula *without the red colored parts.*

$$sim(u, v) = \qquad (1)$$

$$\frac{sem(u,v) \cdot c}{N_{u,v}} \sum_{i}^{|I(u)|} \sum_{j}^{|I(v)|} sim(I_i(u), I_j(v)) \cdot W(I_i(u), u) \cdot W(I_j(v), v)$$

where c is a decay factor in $(0, 1)$, $N_{u,v} = |I(u)| \cdot |I(v)|$ and $sim(\cdot, \cdot)$ is the SimRank score of the neighboring pair-nodes. If $I(u)$ or $I(v)$ are $\emptyset$, then the score is defined to be zero.

In SimRank, the assumed graph model is an unweighted homogeneous graph, where all edges and nodes belong to a single type, thus it ignores the labels' semantics and edge weights. We enrich SimRank by weighting, at each step of the computation, the neighbors' similarity with the edge weights and the nodes' semantic similarity. Formally, given a semantic similarity measure $sem(\cdot, \cdot)$, the red parts indicate our refinements to SimRank standard formula: (i) an additional semantic factor is added; (ii) the edge weights are taken into consideration. Correspondingly, the normalization factor is set to:

$$N_{u,v} = \sum_{i}^{|I(u)|} \sum_{j}^{|I(v)|} W(I_i(u), u) \cdot W(I_j(v), v) \cdot sem(I_i(u), I_j(v))$$

where $sim(\cdot, \cdot)$ in the refined formula denotes the refined similarity of the neighbors. Here too, if $I(u)$ or $I(v)$ are the $\emptyset$, then the similarity score is defined to be zero.

Note that according to our definition of similarity, the semantic similarity of the neighboring pairs of nodes appears as well (as the definition is recursive), and therefore, the similarity of two nodes u, v is, in fact, proportional to the semantic similarity of their neighbors.

Importantly, our definition of SemSim considers all neighbor-pairs. An alternative could be to take edge labels into consideration and restrict attention to neighbor-pairs that are pointed by edges having the same label. However, while such formulation requires only minimal technical changes and all our results remain unchanged, our experiments showed it to be less accurate, as this definition may overlook possibly important relations among the objects. Moreover, both definitions yield essentially the same running times and we thus omit this restriction.

Semantic Similarity. Multiple semantic measures have been proposed in the literature [16, 20]. In general, any similarly function $sem(\cdot, \cdot)$ can be employed in SemSim, as long as it satisfies the following constraints. For all $u, v \in V$:

(1) *Symmetry.* $sem(u, v) = sem(v, u)$.
(2) *Maximum self similarity.* $sem(u, u) = 1$.
(3) *Fixed value range.* $sem(u, v) \in (0, 1]$.

Those requirements are used to prove the soundness of SemSim (Theorem 2.3). The first two are typically satisfied by common semantic measures (e.g., [16, 23, 32]). For the third constraint, scores can be normalized into a $[0 + \epsilon, 1]$ range, for a small $\epsilon > 0$ value [29].

We next briefly overview a simple and effective semantic measure that we have used in our experiments (see Section 6 for a discussion on alternatives). Lin [23] is an *Information Content* (IC)-based measure that is defined over concept taxonomies. The IC of a node v is quantify as the negative of its log likelihood: $IC(v) = -\log(P[v])$, where $P[v]$ denotes the frequency of v. I.e., the more prevalent a concept is, the lower its IC value. Intuitively, the similarity between concepts measures the ratio of the amount of information needed to state their commonality to the information needed to describe them. Given two nodes u and v in a taxonomy, their Lin score is defined as:

$$Lin(u, v) = \frac{2 \cdot IC(LCA(u, v))}{IC(u) + IC(v)}$$

where $LCA(u, v)$ is the lowest common ancestor of u and v in the taxonomy.

Note that Lin satisfies the constraints, only if the IC values are in $(0, 1]$ (proof omitted). To estimate the IC of a concept, we adapted the Seco formula [33] in our implementation, providing a simple linear-time (in the size of the taxonomy) algorithm and extended it to our setting. This adaptation ensures the IC values lie within $(0, 1]$ (see [27] for more details).

Entity	IC value
Thing	0.001
Author, Country	0.01
Country in Asia, Country in America	0.015
China, India, US	0.02
Data Management	0.2
Data Mining	0.3
Crowdsourcing	0.85
Web data mining	0.7
Crowd Mining, Spatial Crowdsourcing	0.9
Bo, John, Aditit, Paul	1.0

Table 1: IC values for Figure 1 entities.

We next provide the full computation of SimRank and SemSim for the example introduced in the Introduction (Example 1.1), while using Lin as the integrated semantic measure.

Example 2.2. We computed the IC values (depicted in Table 1) on the same domain ontology used for the AMiner dataset (which includes a taxonomy of CS terms as well as a geographic taxonomy, see experimental results), and set absent edge weights to 1. For both SimRank and SemSim, we set the decay factor c to 0.8 and the number of iterations k was set to 3.

We first review the relevant Lin scores: since all author-nodes are leafs in the taxonomy, their corresponding IC values are all 1, thus *Lin(Bo,Aditi)* = *Lin(John,Aditi)*= 0.01 (which also serves as the upper bound on their SemSim scores). Using the IC values above, we get: *Lin(Spatial Crowdsourcing,Crowd Mining)*= 0.94 and *Lin(Web Data Mining,Crowd Mining)*= 0.37. Next, we briefly overview SemSim and SimRank computation. At the first iteration, for both measures, $R_0 = 0$ for all authors pairs. Iteratively, at the next step, since all three authors share two common neighbors, *Author* and *Paul*, yet the common field-of-interest of Aditi and John is more semantically similar than the common field of Aditi and Bo, we get for SemSim that: R_1*(John,Aditi)* = 0.0073, while R_1*(Bo,Aditi)*= 0.066. Note that in this step the semantic similarity of common neighbors propagates into the computation. On the other hand, according to SimRank, in this step both pairs similarity scores are equal to 0.1. At the last step, according to SemSim R_2*(John,Aditi)* = 0.0076, while R_2*(Bo,Aditi)*= 0.0073, thus, SemSim obtains the desire result that while all authors are fairly similar, John's similarity to Aditi is a bit greater than Bo's. In contrast, according to SimRank, R_2*(John,Aditi)* = 0.12, while R_2*(Bo,Aditi)*= 0.16. These results are due to the fact that both Aditi and Bo reside in the same continent.

We also computed the SimRank scores solely over the collaboration network (i.e., ignoring the semantic relations). Not surprisingly, since the resulted network is symmetric, the obtained similarity scores for both pairs were exactly the same.

2.3 Basic Properties of SemSim

We next show a few of SemSim's properties which will then be used to present a naïve algorithm for computing SemSim, that serves as a baseline which we improve in the following sections.

Following SimRank's iterative form [13], a solution to Equation (1) can be reached by iterating to a fix-point. For the k-th iteration, an iterative function $R_k(u,v)$ denotes the similarity score of u and v in the k-th iteration. Initially, $R_0(u,v)$ is defined as 0 if $u \neq v$ and 1 otherwise. Iteratively, $R_{k+1}(u,v)$ is computed from $R_k(\cdot,\cdot)$ as follows:

$$R_0(u,v) = \begin{cases} 0, u \neq v \\ 1, u = v \end{cases} \tag{2}$$

$$R_{k+1}(u,v) = \tag{3}$$

$$\frac{sem(u,v)\cdot c}{N_{u,v}} \sum_{i}^{|I(u)|} \sum_{j}^{|I(v)|} R_k(I_i(u), I_j(v)) \cdot W(I_i(u),u) \cdot W(I_j(v),v)$$

We can prove that the iterative SemSim form has the following properties:

Theorem 2.3. *$\forall u,v \in V$ and for every $0 \leq k \in \mathbb{N}$:*
(1) Symmetry. $R_k(u,v) = R_k(v,u)$.
(2) Maximum self similarity.$R_k(u,u) = 1$.
(3) Monotonicity. $0 \leq R_k(u,v) \leq R_{k+1}(u,v) \leq 1$.
(4) Existence. The solution always exists.
$0 \leq c < min(argmin_{N_{u,v}}(N_{u,v}), 1)$, the solution is unique.

First, note that the decay factor's upper bound can be found in average time of $O(n^2 \cdot d^2)$, where d is the average in-degree in the graph, by simply iterating over all node-pairs. Second, we observe that the uniqueness property here is a weaker version than the one that was proven for SimRank, where the solution is unique for every $0 \leq c < 1$. Yet, our experiments show that for real-life networks, the upper bound is high enough to comfortably accommodate typical c values chosen for SimRank (e.g., 0.6, as used in [24, 39]).

We can also show (following similar proof for SimRank [46]) that not only the scores are monotone (i.e, $R_k(u,v) \leq R_{k+1}(u,v)$), their differences in consecutive iterations are bounded.

Proposition 2.4. *For every $u,v \in V$ and $k > 0$: $0 \leq R_{k+1}(u,v) - R_k(u,v) \leq sem(u,v) \cdot c^{k+1}$*

This suggests that the iterative form of SemSim converges as fast as SimRank (where the convergence was shown to be c^{k+1} [46]), and possibly faster due to the additional semantic factor. Another useful property is that $sem(\cdot,\cdot)$, the semantic similarity of two nodes, provides a natural upper bound on their SemSim score. This property is highly effective since, as we will show, it can be used to prune un-promising node-pairs.

Proposition 2.5. *For every two nodes $u,v \in V$: $sim(u,v) \leq sem(u,v)$.*

To conclude, Theorem 2.3 provides a simple algorithm for computing SemSim, that computes its iterative form to its fix-point (or up to a required precision bound). We assume that the computation of a single-pair semantic similarity score can be done in constant time (possibly after pre-processing), without materializing the $n \times n$ matrix of scores. Indeed, this is the case for numerous semantic measures [16, 32], Lin's measure included. Given this, the complexity of the iterative algorithm is equivalent to SimRank's complexity [13]: The time complexity is $O(k \cdot d^2 \cdot n^2)$, where $n = |V|$, d is the average in-degree in G and k is the number of iterations. The worst case complexity for a given k is $O(n^4)$.

3 RANDOM SURFER-PAIRS MODEL

The iterative algorithm provided in the previous section has two main disadvantages: (i) it computes all pair-wise scores, even if one is interested only in a single-pair, and (ii) its complexity is prohibitive for large graphs. To address these issues, we provide an alternative interpretation to SemSim, based on the *random surfer model* for SimRank, then, explain how SemSim can be computed efficiently. In essence, we show that with careful adjustments, an analogous random surfer model can be establish for SemSim. The key challenge is to incorporate semantics. We show that SemSim measures how soon two random surfers are expected to meet, if they start in two nodes and randomly walk on the graph backward, while being aware of both edge weights and semantics. We define *Semantic-Aware Random Walks* (SARW), then prove that SemSim can be computed using them. Interestingly, we will see that semantics can be leveraged to speed up the computation.

3.1 Semantic-Aware Random Walks (SARW)

Following [13], we use the definition of a *node-pair graph G^2*, in which each node represents an ordered pair of nodes from G. An edge $e = ((u,u'),(v,v')) \in G^2$ iff both (u,v) and (u',v') are $\in G$. We extend the definition with an assignment of weights: The weight of an edge $e = ((u,u'),(v,v'))$ is defined as: $W_{G^2}(e) := W(u,v) \cdot W(u',v')$. For simplicity, we use the notation of $W(e)$ to indicate an weight in both G and G^2, when the context is clear.

Let us assume that all edges in G have been reversed. For example, Figures 2a and 2b display a graph G and all out-edges from *(A,B)* (after reversal). For simplicity, all edge weights are set to 1. We call a node $(u,v) \in V^2$ a *singleton node* if $u = v$. In SimRank, a surfer chooses the next node uniformly at random out of all out-neighbors of the current node. To incorporate semantics and weights, we devise the following distribution.

Definition 3.1 (Semantic-Aware Probability Distribution). The probability a random surfer traveling G^2 in a current node (u,u') would next move to its out-neighbor (v,v') is:

$$P[(u,u') \to (v,v')] := \frac{W((u,u'),(v,v')) \cdot sem(v,v')}{\sum_{i=1}^{|O((u,u'))|} W((u,u'),O_i(u,u')) \cdot sem(O_i(u,u'))}$$

Using the distribution above, we define SARWs as follows. A walk in G^2 represents a pair of walks in G. Let $w = \langle w_1, ..., w_k \rangle$ denote a walk in G^2, where $w_1, ..., w_k \in V^2$, and $l(w) = |w|$. The walk w has the probability $P[w]$ of traveling within it, where $P[w] := \prod_{i=1}^{k-1} P[w_i \to w_{i+1}]$.

Importantly, this distribution defines a positive probability to *all* paths in G^2. However, as the choice of the next step relies on the semantic similarity of node-pairs, pairs of higher semantic similarity are preferred over pairs of low similarity (typically, pairs whose nodes belong to different categories). Namely, paths that share the same edge label in each step, are likely to get higher probabilities. Nonetheless, even paths that do not share the same labels are considered, as they may also provide relevant information[2]. We provide here an illustrative example for a computation of SARWs.

Example 3.2. Consider again Figure 2b. Observe that author A's current country is *Canada*, and author B's origin country is the *USA*. Noticeably, even though the two edges do not share the same label, this information may be important when assessing similarity. According to our definition we get that since the entities *Canada* and *USA* are semantically similar ($Lin(Canada, USA) = 0.8$), the probability that a random surfer in the node *(A,B)* will move next to its neighbor *(Canada, USA)* is:

$$P[(A,B)\rightarrow(Canada, USA)] = \frac{0.8}{0.8+0.2+0.2+1.0} = 0.36$$

On the other hand, as the two entities *Author* and the *USA* are not semantically similar ($Lin(Author, USA) = 0.2$), the corresponding probability is lower:

$$P[(A,B)\rightarrow(Author, USA)] = \frac{0.2}{0.8+0.2+0.2+1.0} = 0.09$$

The SimRank score of a node $(u,v) \in V^2$ can be computed using all walks from it leading to a singleton node in G^2. Analogously, we prove that SemSim can be computed using all semantic-aware walks from (u,v) leading to singleton nodes in G^2. Let $W = \{(u,v) \rightsquigarrow (x,x)\}$ be the set of all walks in G^2 form (u,v) to any singleton node (x,x). If no such paths exist, then $W = \emptyset$. By definition, (x,x) is the only singleton node in w (after the first meeting, the two surfers halt). Let: $s'(u,v) = sem(u,v)\sum_{w\in W} P[w]\cdot c^{l(w)}$. Theorem 3.3 provides an alternative way to compute the SemSim score of a single pair.

Theorem 3.3. *$\forall u, v$, given c which ensures the uniqueness of $sim(\cdot,\cdot)$: $s'(u,v) = sim(u,v)$*

Using our refined model, one may compute (single pair or all pairs) SemSim scores over G^2. However, for large graphs, its size may be too large. We next explain how the semantics can be effectively employed to reduce the size of G^2.

3.2 Reducing the Size of G^2

In many practical applications one is interested only in node-pairs whose similarity scores are above a given threshold. Semantics provides an efficient tool to prune G^2 in such situations. Intuitively, Prop. 2.5 provides a semantic-based upper bound on the similarity scores, which can be used to avoid materializing un-promising node-pairs. We devise a reduced version of G^2 on which the computation of SemSim (for node-pairs with similarity scores are above a given threshold) yields the same result as that computed via the full graph G^2. Indeed, our experimental results demonstrate a significant reduction in the size of G^2.

Given a threshold $0 < \theta < 1$, we define the graph G^2_θ, which includes only pairs s.t. their semantic scores are $> \theta$. However, simply omitting nodes from G^2 directly affects the similarity scores, thus, may lead to inaccurate scores. We therefore incorporate omitted paths by refining the edge weights and possibly adding new edges. Intuitively, each omitted path is replaced by a corresponding edge, whose weight reflects its probability. If such an edge already exists in G^2, the omitted edge's weight is added to the existing edge weight. Moreover, the weight of omitted path is further weighted by the decay factor c, to ensure the similarity scores would not be affected. Last, to ensure that the probability of choosing a neighbor remains the same as in the original graph G^2, the graph G^2_θ includes a new vertex D, that has only in-neighbors, and serves as a “drain".

Definition 3.4. [G^2_θ] Given a node-pair graph G^2 and a threshold $0 < \theta < 1$, $G^2_\theta = (V_\theta \cup \{D\}, E_\theta, W_\theta)$, where: $V_\theta \subseteq V^2$ is a set of nodes and D is a new node, $E_\theta \subseteq (V_\theta \cup \{D\} \times V_\theta \cup \{D\})$ is the edges set and W_θ is a weight function, defined as follows.

- **Nodes:** A node $(u,v) \in V_\theta$ iff $sem(u,v) > \theta$.
- **Edges:** An edge $e = ((u,u'),(v,v')) \in E_\theta$ iff at least one of the following conditions holds
  (1) The nodes $(u,u'),(v,v')$ are adjacent in G^2.
  (2) There exists a walk in G^2, where $w = \langle(u,u'), w_1, \ldots, w_k, (v,v')\rangle$ and the node-pairs $w_1 \ldots, w_k \notin V_\theta$.
- **Weights:** The weight of an edge $e = ((u,u'),(v,v'))$ is defined as $W_\theta(e) = W_1(e) + W_2(e)$ where: $W_1(e) = W_{G^2}(e)$ if $e \in G^2$ and $(u,u'),(v,v') \in V_\theta$ and 0 otherwise, and $W_2(e) = \sum_{w:(u,u')\rightsquigarrow(v,v')} P[w]\cdot c^{l(w)-1}$, where $t = \langle(u,u'), w_1, ..., w_k, (v,v')\rangle$ is a path in G^2 and the node-pairs $w_1, ..., w_k \notin V_\theta$.
- **Edges to D:** Edges to the vertex D are added as follows: $\forall(u,u') \in V_\theta$ if the sum of all out-edges of (u,u') in the graph G^2_θ is different then the sum of all out-edges of (u,u') in the graph G^2, then $((u,u'),D) \in E_\theta$ and $W_\theta((u,u'),D)$ is set to be the difference.

In the last point, to ensure that all weights are strictly positive, we can prove that for every node in G^2_θ, the sum of out-edges in the original graph G^2 is always $\geq$ than the sum of out-edges in the reduced graph G^2_θ. Additional edge pruning can be done by the removal of all out-edges from singleton nodes. Since only the first meeting point of the surfers affects similarity scores, such edges can be omitted without changing scores (proof omitted). For example, Figures 2b and 2c depict a partial graph G^2 and its reduced version G^2_θ (faded nodes are dropped).

The similarity scores over G^2_θ, denoted as $s_\theta(\cdot,\cdot)$, are defined as the result of the random surfing computation on the reduced graph. I.e., if $(u,v) \notin V_\theta$ then $s_\theta(u,v) = 0$, else: $s_\theta(u,v) = sem(u,v)\sum_{w:(u,v)\rightsquigarrow(x,x)} P[w]\cdot c^{l(w)}$, where w is a path in G^2_θ and $l(w)$ is its length. We can now provide an alternative way to compute SemSim scores over the graph G^2_θ.

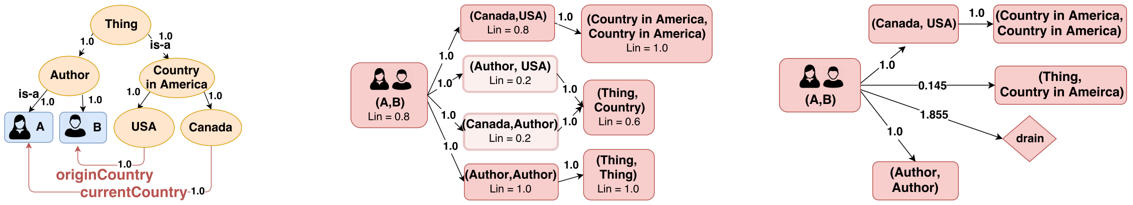
Theorem 3.5. $\forall(u,v) \in V_\theta : s_\theta(u,v) = sim(u,v)$

In conclusion, as we show in our experiments, the size of the graph G^2_θ is considerably smaller than that of G^2 and consequently, computing SemSim over G^2_θ requires exploring far less and shorter paths, hence it is more efficient. However, when considering very large graphs, even this compact representation might still be excessively large. To that end, in the next section, we present an alternative approach that simulates two random surfers directly over G.

4 APPROXIMATED SEMSIM

We next present an alternative approach for an efficient computation of SemSim, based on a solution originally proposed for

[2] In contrast to the meta-path approach [37] that restricts attention only to same-labels paths.



(a) Sample graph G. **(b) All paths from** ***(A,B)*** **in** G^2. **(c) All paths from** ***(A,B)*** **in** G^2_θ.

Figure 2: Example graph G**, its reversed graph** G^2 **and its reduced version** G^2_θ ($\theta = 0.3$, $c = 0.8$).

SimRank [9]. First, we prove that a naïve solution of simply replacing the underlying uniform distribution with the semantic aware distribution leads to a quadratic increase in the sample size. To overcome this, we employ *Importance Sampling*. As employing it still entails a computational overhead, we provide a complementary pruning technique that significantly speeds up computation while maintaining low error rates. We first recall SimRank optimizations while addressing the emerging challenges of their implementation in SemSim, then present our refined framework.

4.1 SimRank's Basic MC Framework

Suppose that we have two reverse walks w_1 and w_2 from the nodes $u, v \in V$, resp., and they first meet at the τ-th step. I.e., the τ-th steps of w_1 and w_2 are identical, but for any $l < \tau$ their l-th steps are different. If the walks do not meet, then $\tau \to \infty$. Given two random walks of length $k-1$, $w_1 = \langle u_1, ..., u_k \rangle$ and $w_2 = \langle v_1, ..., v_k \rangle$, let w denote their *coupled random walk*, where $w = \langle (u_1, v_1), ..., (u_k, v_k) \rangle$. It has been shown in [13] that $simrank(u, v) = \mathbb{E}[c^\tau]$. The authors of [9] suggested a *Monte Carlo* (MC) approximation framework, utilizing this equality, by sampling separated random walks, and approximating the similarity score using the average meeting distance. Specifically, to approximate SimRank, the framework precomputes a set of reversed random walks from each node in G, s.t (i) each set has n_w walks, and (ii) each walk is truncated at step t. Then the estimated SimRank score of u and v is defined as:

$$\frac{1}{n_w} \sum_{l=1}^{n_w} c^{\tau_l}$$

where τ_l denotes the step at which the two walks, sampled from u and v resp., first met, and ∞ otherwise. The space and preprocessing time complexities of this framework are both $O(n \cdot n_w \cdot t)$, and the method takes $O(n_w \cdot t)$ time to answer a single-pair SimRank query.

An important observation underlying SimRank's MC framework is the fact that the probability of a coupled random walk sampled from G^2, can be computed by simply multiplying the two marginal probabilities of the separate walks, sampled from G. Formally, given a coupled walk $w = \langle (u_1, v_1), ..., (u_k, v_k) \rangle$, its probability is:

$$Pr[w] = \prod_{i=1}^{k-1} \frac{1}{|O(u_i, v_i)|}$$

Considering its probability using the separated walks sampled from G, we get:

$$\prod_{i=1}^{k-1} \frac{1}{|O(u_i)|} \prod_{i=1}^{k-1} \frac{1}{|O(v_i)|} = \prod_{i=1}^{k-1} \frac{1}{|O(u_i)||O(v_i)|} = \prod_{i=1}^{k-1} \frac{1}{|O(u_i, v_i)|}$$

That is, in SimRank, one can simply sample walks from each node separately, directly from G, without materializing G^2. We will next show that this is not the case for SemSim, and present a refined sampling method for the SARWs.

4.2 Naïve MC framework for SemSim

Analogously, for SemSim we have: $sim(u, v) = sem(u, v) \cdot \mathbb{E}_P[c^\tau]$, where P is the semantic-aware probability. Note that when using the semantic-aware probability, one can no longer sample the walks separately. To account for the semantic similarity during the sampling process, one must consider *a pair of nodes* in each step. A naïve solution would be to generate a set of SARWs from every node-pair, then directly apply the MC framework. Namely, we can get an adjusted framework for SemSim with the same time complexity and error rate as in SimRank (because the time complexity depends on the number of walks from each pair of nodes, and this solution has the same number, n_w, of walks from each pair). However, in SimRank, the sampling set is of size $O(n_w \cdot t \cdot n)$, whereas this solution requires a much larger sampling set, i.e. $O(n_w \cdot t \cdot n^2)$ walks, as it samples n_w walks for each pair. To avoid this quadratic increase of data storage, we use importance sampling [10].

4.3 IS-based MC framework for SemSim

The core idea of our solution is to sample separate walks directly from G, using a different distribution than the "unknown" distribution P, then, apply importance sampling to estimate the desired similarity scores [10]. For completeness of this paper, we provide a short overview on the importance sampling technique, then present our adjusted framework.

Importance sampling is a general technique for estimating properties of a distribution while only having samples generated from a different one. For a single pair $u, v \in V$, we wish to estimate the expected value of $sem(u, v) \cdot c^{l(w)}$, where w is a coupled random walk drawn from P, i.e.,

$$\mathbb{E}_P[sem(u, v) \cdot c^{l(x)}] = sem(u, v) \cdot \sum P(w) \cdot c^{l(w)}$$

Given n_w samples $w_1, ..., w_{n_w}$ of coupled random walks drawn from P, an empirical estimate of $\mathbb{E}_P[sem(u, v) \cdot c^{l(w)}]$ is:

$$\mathbb{E}_P[sem(u, v) \cdot c^{l(w)}] \approx sem(u, v) \cdot \frac{1}{n_w} \sum_{i=1}^{n_w} c^{l(w_i)}$$

Using a simple manipulation we get:

$$\mathbb{E}_P[c^{l(w)}] = \sum \frac{P(w) \cdot Q(w) \cdot c^{l(w)}}{Q(w)} \approx \frac{1}{n_w} \sum_{i=1}^{n_w} c^{l(w_i)} \frac{P(w_i)}{Q(w_i)}$$

where Q is a distribution s.t $\forall w$ if $Q(w) = 0$ then $P(w) = 0$. Namely, we get an unbiased estimator of the function $c^{l(w)}$ under the distribution P, using samples drawn from Q. In our case, we can only sample separated walks from G (to avoid materializing G^2), while the desired distribution is defined over walks from G^2. Indeed, the expected value of the new estimator is equal to the desired one, that is, for every node-pair u, v we have:

$$sem(u, v) \cdot \mathbb{E}_Q[\frac{P(w) \cdot c^{l(w)}}{Q(w)}] = \tag{4}$$

$$sem(u, v) \cdot \mathbb{E}_P[c^{l(w)}] = sim(u, v)$$

where w is a coupled random walk from u and v, P is the semantic-aware distribution and Q is the proposal distribution. Note that this equality holds for any choice of Q and $sem(\cdot,\cdot)$. Let $\hat{s}_Q(u,v)$ denote the score obtained using the MC simulation with a distribution Q. We can prove the following proposition, that ensures the approximation method has a bounded error (as was proven for SimRank [9]).

Proposition 4.1. *For a node-pair u,v, with at least $1-2e^{-n_w\cdot\frac{\epsilon^2}{2\cdot(1+\epsilon/3)}}$ probability: $|\mathbb{E}[\hat{s}_Q(u,v)]-\hat{s}_Q(u,v)|\leq\epsilon$, where n_w is the number of walks from each node and ϵ is the error rate.*

However, we note that $\mathbb{E}[\hat{s}_Q(u,v)]\neq sim(u,v)$, due to the truncation imposed on the sampled walks. To address this issue, following the analysis provided in [39] we get:

$$|\mathbb{E}[sim(u,v)]-\hat{s}_Q(u,v)|=$$

$$|\mathbb{E}[sem(u,v)\cdot c^{\tau}]-sem(u,v)\cdot Pr[\tau\leq t]\mathbb{E}[c^{\tau}|\tau\leq t]|=$$

$$sem(u,v)\cdot|Pr[\tau>t]\cdot\mathbb{E}[c^{\tau}|\tau>t]|\leq sem(u,v)\cdot c^{t+1}\leq c^{t+1}$$

By Prop. 4.1 and the inequality above, using union bound, we can prove the following.

Proposition 4.2. *For any node-pair u,v and $0<\epsilon,\delta<1$, if $t>log_c(\frac{\epsilon}{2})$ and $n_w\geq\frac{14}{3\epsilon^2}(log(\frac{2}{\delta})+2log(n))$, with at least $1-\delta$ probability: $|\hat{s}_Q(u,v)-sim(u,v)|\leq\epsilon$*

Furthermore, we can prove that the probability of interchanging two nodes in the similarity ranking of a node u converges to zero exponentially in the number of sampled walks n_w.

Proposition 4.3. *For every nodes u,v and v', such that $\delta=sim(u,v)-sim(u,v')>0$ we have:*

$$Pr[\hat{s}_Q(u,v)<\hat{s}_Q(u,v')]\leq 2e^{-\frac{n_w\cdot\delta^2}{2+2\frac{\delta}{3}}}$$

Note, however, that the accuracy of estimation depends on the variance of the estimator, $Var(\hat{s}_Q(u,v))$, which in turn depends on the distribution Q. In general, $Var(\hat{s}_Q(\cdot,\cdot))$ is bounded in $[0,1]$, since the similarity scores are bounded in $[0,1]$. Therefore, we wish to find a distribution Q s.t. (i) the sampling process and probabilities computation can be done efficiently and (ii) $Var(\hat{s}_Q(\cdot,\cdot))$ is minimal. Here, since we do not have a-priori knowledge on either the semantic similarity or the meeting points of coupled walks, we choose Q to be the uniform distribution. See [27] for a discussion of other possible choices.

We are now ready to present Algorithm 1, an MC framework for computing single-pair SemSim scores, assuming, w.l.o.g., that Q is the uniform distribution. Ignore, for now, the lines highlighted in red. At preprocessing, we generate n_w random walks from each node, drawn from Q. Then, when a single-pair query arrives, $sim(u,v)$, we consider the set of coupled random walks starting from u and v. For each coupled walk, if the two walks indeed meet, the probability of their prefix until the meeting point is computed according to the distributions P and Q (lines 10 – 16). Then, the obtained score is added to the total similarity score (line 19). Finally, the estimated score is divided by the number of samples n_w (line 20).

Proposition 4.4. *For every $u,v\in V$, the expected output of Algorithm 1 is $sim(u,v)$, and the average time complexity is $O(n_w\cdot d^2\cdot t)$, where n_w is the number of sampled walks from each node, t is their length and d is the average in-degree in the graph G.*

That is, an additional factor of d^2 is added to our framework's running-time. However, as we will show next, we can compensate for this by employing a pruning-based optimization.

Algorithm 1: IS-based MC framework for SemSim.

```
   Input  : $n_w$ walks of length $t$ from each node, a decay
            factor $c$, and a threshold $\theta$.
1  $sim = 0$
2  if $sem(u,v) \leq \theta$ then
3      return 0
4  for $i$ = 1,...,$n_w$ do
5      Let $w_i$ denote the coupled walk of the $i$-th walks from $u$ and
         $v$
6      Let $k$ be the samllest offset s.t the $i$-th walk from $u$ and from
         $v$ meet
7      if such $k$ exists then
8          Let $\tau(w_i)$ denote the prefix of $w_i$ up to offset $k$
9          Denote $\tau(w_i) = \langle(u_1, v_1), \ldots, (u_k, v_k)\rangle$, $Pw = 1$,
             $Q_W = 1$, $sim_w = 1$.
10         for $i = 1, \ldots, k-1$ do
11             $Pw\cdot = sem(u_{i+1}, v_{i+1})\cdot W(u_{i+1}, u_i)\cdot W(v_{i+1}, v_i)$,
                 $SO = 0$
12             for $I_j(u_i)$ in $I(u_i)$ do
13                 for $I_z(v_i)$ in $I(v_i)$ do
14                     $SO+ = W(I_j(u_i), u_i)\cdot W(I_z(v_i), v_i)\cdot$
                         $sem(I_j(u_i), I_z(v_i))$
15             $Pw/= SO$, $Q_W\cdot = |I(u_i)|\cdot|I(v_i)|$
16             $sim_w = sim_w\cdot\frac{Pw}{Q_W}\cdot c$
17             if $sim_w \leq \theta$ then
18                 break
19         $sim = sim + sim_w$
20 return $\frac{sem(u,v)\cdot sim}{n_w}$
```

4.4 Pruning

Similarly to the idea presented for G^2, we provide a pruning technique for the MC framework, avoiding the similarity computation of low probability walks. Recall that for G^2_θ the suggested technique ensures that scores above a given threshold would not be effected. Here, the approximation error may increase up to a controlled threshold. While in G^2 it is good pruning-wise to use high thresholds, here, if we use too high value the error grows and the scores may become meaningless, thus lower values are advisable. As opposed to the unbiased estimator we devised in the previous section, our pruning technique adds a one-sided additive error to scores. However, as we demonstrate in our experiments, it accelerates the performance significantly, while successfully distinguishing highly similar pairs from the rest.

For a coupled random walk w, let $s(w)$ denote the contribution of w to the similarity score. Since in every estimation we consider n_w coupled walks, it follows that:

$$s(w)=\frac{1}{n_w}\cdot\frac{P[w]\cdot c^{(l(w))}}{Q[w]}$$

Instead of computing the exact score of $s(w)$, our idea is to upper bound it. Concretely, given a coupled walk $w=\langle w_1,..,w_k\rangle$, with a closer look on $s(w)$ we get:

$$s(w)=\prod_{i=1}^{k-1}\frac{P[w_i\rightarrow w_{i+1}]\cdot c}{Q[w_i\rightarrow w_{i+1}]}\leq\prod_{i=1}^{l}\frac{P[w_i\rightarrow w_{i+1}]\cdot c}{Q[w_i\rightarrow w_{i+1}]}$$

where $0<l<k-1$. Namely, $s(w)$ can be bounded in each step in the chain. Therefore, given a threshold θ, we can avoid computing the exact score, if in a certain step, the obtained score is smaller than the bound θ, as from this step on the score can only decrease. Formally, given a threshold $\theta\in[0,1]$ and a coupled walk w, we define $\hat{s}(w)$ as follows:

Table 2: Datasets.

Dataset	Size	Tasks
AMiner	|V| = 0.35M |E| =3M	Entity Resolution
Amazon	|V| = 0.6M |E| =6M	Link Prediction
Wikipedia	|V| = 4.7K |E| =101K	Relatedness
WorNet	|V| = 82K |E| =128K	Relatedness

Definition 4.5 (Approximated coupled walk score). The approximated score of a coupled walk w is defined as:

$$\hat{s}(w) := \prod_{i=1}^{l} \frac{P[w_i \to w_{i+1}] \cdot c}{Q[w_i \to w_{i+1}]} \leq \theta$$

where l is the smallest index this equation holds. If no such l exists, then $\hat{s}(w) = s(w)$.

As in the pruning done for G^2_θ, we can avoid computing scores of all pairs u, v s.t. $sem(u, v) < \theta$ (and again, obtain an error bounded by θ). In such cases, the result score is set to 0.

Consider again Algorithm 1. The highlighted red lines indicate our pruning refinements. In particular, in lines $2 - 3$, similarity scores of node-pairs with low semantic scores are set to 0, and in lines $17 - 18$ we ensure that scores of coupled walks are above θ, and otherwise, are bounded. We can prove that given $\theta \in [0, 1]$, the additional additive error is bounded by θ.

Proposition 4.6. *Given $\theta \in [0, 1]$, the additional additive error of the IS-based MC framework with pruning is bounded by θ.*

To ensure the estimated scores lies in $[0, 1]$, we add the following constraint on θ.

Lemma 4.7. *For every $\theta \in [0, 1 - c]$ and $u, v \in V$, the approximated similarity score $\hat{s}_Q(u, v)$ obtained by Algorithm 1 with pruning lies in $[0, 1]$, where $c \in [0, 1)$.*

This lemma implies that the MC framework with pruning can efficiently capture big differences among similarity scores. But when it comes to small differences, the error of approximation obscures the actual similarity ranking, and an almost arbitrary reordering is possible. However, for many similarity search applications it is sufficient to distinguish between very similar, modestly similar, and dissimilar nodes. In terms of run-time, while the worst-case time complexity remains the same (no pruning is done), our experiments show pruning to be extremely effective in practice, yielding running times on par with SimRank.

Concluding, as mentioned in the Introduction, multiple optimizations techniques have been developed for SimRank based on SimRank's MC framework. Our framework extends for them as well. We discuss this in more details in our technical report [27], providing several examples, and also demonstrate this in our experimental study.

5 EXPERIMENTAL RESULTS

We complement our work with an experimental study, conducted to examine the performance of our measure as well as its usefulness in capturing objects' similarity, compared to measures proposed in previous work.

5.1 Experimental setup

We implemented SemSim in Java 7, and demonstrate its performance using Lin as the integrated semantic measure. All experiments were conducted on a Linux machine with 96GB of memory. We next describe the graph datasets we examined, then detail the parameters setting.

Datasets. We used several graph datasets, commonly used in the literature, which suitably include objects possessing both structural information and semantic meaning, as depicted in Table 2. Unless stated otherwise, all edge weights were set to 1.

AMiner. This graph was extracted from [1], and contains data on 1.5M academic papers. We extracted a weighted co-author graph focused on 30 database conferences. From each paper, we extracted its authors and relevant terms. In addition, we incorporated a domain taxonomy, built by aligning the terms with concepts from DBpedia [5]. The graph includes edges of three types: (1) collaboration edges (with weights reflecting the number of collaborations between two authors); (2) terms-authors edges (where weights correspond to the prevalence of the term in a given author's papers) and (3) taxonomy edges.

Amazon. This dataset was obtained from [18]. It contains 0.5M items from different categories, a domain taxonomy (obtained from Amazon product categorization), and information about co-purchased items. The edge types are: (1) edges between co-purchase items (with weights reflecting the number of times two items were bought together) and (2) taxonomy edges, linking between products to their categories, as well as categories to their super-categories.

Wikipedia. This dataset, obtained from [18], contains 4.7K Wikipedia articles, each is represented by a node. The domain taxonomy was built from Wikipedia categories. The edge types are: (1) links between articles and (2) taxonomy edges.

WordNet. This dataset is the noun sub-part of the lexical base WordNet [26]. The edge types are:(1) part-of relations, the non-hierarchical relations in WordNet and (2) taxonomy edges.

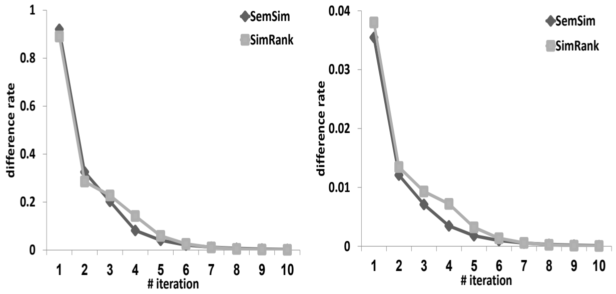
For both AMiner and Amazon datasets, we extracted a smaller versions to be used in the execution of the costly iterative forms of SemSim and SimRank. In AMiner, the small version includes the top 7K authors by number of publications, and in Amazon it includes the top 5K most bought items.

Parameter setting. For all datasets, we found the upper bound on the decay factor c by iterating on all node-pairs u, v computing $N_{u,v}$. We report that in all cases this value was > 0.6, a commonly used value for the decay factor in SimRank [24, 39]. Unless mentioned otherwise, for both SemSim and SimRank we used the following system parameters: The decay factor (c) was set to 0.6, and for the probabilistic framework, a set of 150 random walks of length 15 was sampled from each node. As for the threshold parameter θ used for pruning, we set $\theta = 0.05$. According to our experiments, this choice of the parameters allows for fast execution times, while maintaining negligible error rate.

5.2 Performance Evaluation

We review the performance of SemSim from five aspects: The convergence rate of its iterative form, the size of the reduced graph G^2_θ compared to the full graph G^2, the performance of Algorithm 1, in terms of execution times and error rate and the preprocessing phase.

Convergence. Our experiments validate empirically Prop. 2.4, showing that SemSim converges as fast as, and even faster than SimRank. We measured the average relative and absolute differences between similarity scores at consecutive iterations, for both SemSim and SimRank iterative forms, on different datasets. Results are depicted in Figure 3. Indeed, SemSim converged faster



(a) Avg. relative difference **(b) Avg. absolute difference**

Figure 3: Scores differences in consecutive iterations.

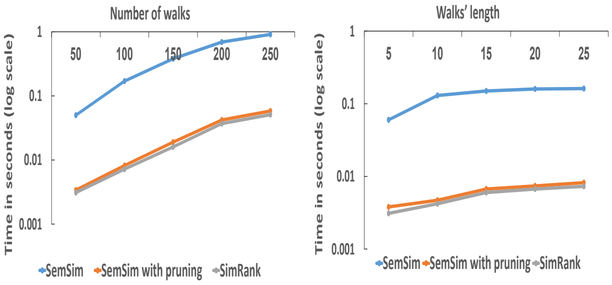
Dataset		G^2	$G^2_\theta, \theta = 0.90$	$G^2_\theta, \theta = 0.95$
AMiner	# nodes	60M	14K	9K
	#edges	2.2B	39M	7.8M
	Avg. # of paths to singletons	17	10	5
	Avg. paths' length	4	3	3
Wikipedia	# nodes	22M	10K	6K
	#edges	10.2B	23.5M	4.7M
	Avg. # of paths to singletons	19	10	6
	Avg. paths' length	5	4	3

Table 3: The size of G^2 and G^2_θ for $\theta = 0.9$ (top ≈ 5K) or $\theta = 0.95$ (top ≈ 1K).

than SimRank, and in all experiments it converged after 5 iteration, i.e., the average (relative and absolute) difference between similarity scores was smaller than 10^{-}3.

Size of G^2_θ. We analyze the effectiveness of the G^2 pruning, demonstrating that when only highly similar objects are of interest (e.g., top-k queries), pruning with high θ values (e.g., 0.9, 0.95) is highly effective, and reduces the size of G^2 significantly. We refer the reader to Table 3, depicting a detailed comparison between the reduced graph G^2_θ (while setting $\theta = 0.9$ and $\theta = 0.95$), and the original graph G^2, constructed from the Amazon and Wikipedia datasets. In addition to the significant reduction in the number of nodes and edges, one can see that the average length of a path and the number of paths leading to singleton nodes (i.e., the number of paths that are considered while computing SemSim) were greatly reduced as well. However, while the measured size of G^2_θ is smaller than G^2 in approximately 3 orders of magnitude, this approach does not trivially scale for immense networks, in which the approximated framework is preferable.

Execution Times. We examine the running time of our MC framework (with and without pruning) as a function of the number of walks, n_w, and the truncation point t, compared to SimRank MC framework. Figure 4 depicts the average measured running times on the Amazon dataset (the results obtained over the other dataset demonstrated similar trends, and thus omitted from presentation). Not surprisingly, without pruning, the average time of SemSim is indeed slower than of SimRank: 0.217 ms and 0.0035 ms for SemSim and SimRank, resp. However, the running times with pruning are significantly faster, becoming close to those of SimRank (in average 0.0038 ms, where $\theta = 0.05$). Additionally, we examined the performance of both measures, using SLING optimization [39] recently suggested for SimRank (described in our technical report [27]), while storing probabilities only for node-pairs with semantic similarity scores >= 0.1. For both measures we achieved a significant improvement in times - 0.00021 ms and 0.00023 ms for SimRank and SemSim resp., with a memory consumption induced by the SLING index of size 1.1GB and 3.2GB, resp.



(a) t is set to 15 **(b) n_w is set to 150**

Figure 4: Average running times for single-pair similarity query.

Dataset		SemSim with pruning $\theta = 0.05$	SemSim	SimRank
AMiner	Pearson's r	0.89	0.91	0.92
	Mean var	0.001	0.001	0.0004
	Max var	0.025	0.027	0.004
	Mean rel. err	0.405	0.397	0.274
	Max rel. err	0.478	0.468	0.364
	Mean abs. err	0.063	0.019	0.018
	Max abs. err	0.080	0.035	0.029
Amazon	Pearson's r	0.92	0.93	0.93
	Mean var	0.001	0.001	0.0005
	Max var	0.022	0.021	0.006
	Mean rel. err	0.366	0.320	0.298
	Max rel. err	0.428	0.399	0.389
	Mean abs. err	0.056	0.020	0.020
	Max abs. err	0.075	0.027	0.025

Table 4: Accuracy of approximation.

Approximation Error. We compared SemSim and SimRank approximated scores to scores computed by their iterative forms, to asses the cost of incorporating semantics. We then evaluated the error of approximation in terms of Pearson's r-correlation (by comparing approximated scores to the ground truth), variance, and error size. We report the error of approximation as measured for the (larger versions of) Amazon and AMiner dataset. In each experiment, we randomly selected 1K node-pairs, and computed the approximated similarity scores in 100 runs (rebuilding the random walks index before each run). The results, provided in Table 4, depict the Pearson's r-correlation values (achieved by comparing the approximated scores to the ground truth ones, obtained by the iterative forms), the (mean and maximal) variance of the estimators, and the (mean and maximal, relative and absolute) errors, as obtained by SemSim (with or without pruning) and SimRank.

As expected, SemSim's mean error is slightly higher than that of SimRank, yet both are in the same order of magnitude (0.366, 0.32, 0.298 for SemSim with pruning, without pruning, and SimRank, resp.). As discussed in Section 5.2, while the error of approximation for SemSim (with and without pruning) is slightly higher than for SimRank, the number of interchanges between the approximated scores and ground truth ones, as measured by the Pearson r-correlation[3], is significantly low and essentially equivalent to SimRank's (with or without pruning). This positive

[3] The Pearson r-correlation measures the "strength" of the linear association between ground truth scores and approximated ones.

result indicates that applying IS does not cause far apart scores to interchange, while maintaining execution times essentially on-par with SimRank.

Preprocessing. We complement the running times experiments by providing details regarding preprocessing time and the space costs of our framework. The offline processing, in which random walks are sampled, took approximately 2.5 minutes (in average over all datasets). While the sampling procedure performed as in SimRank, SemSim requires an additional work due to the semantic similarity measure. Following [11], we processed the taxonomical subpart of the graphs to facilitate constant-time Lin semantic similarity computations at run time. In all cases, the processing time took less 10 minutes. For example: In the Amazon dataset, where the taxonomy contains 2.5M edges, this phase took approximately 7 minutes.

The memory consumption of SemSim's MC framework is prominently due to the random walk index[4]. Additional memory costs for SemSim were due to the Lin semantic measure: storing the IC values and the data structure that allows for a constant time similarity computation (as described in [11]). Overall, the storage size was varied between 5 − 9MB, for all datasets, depending on the size of the taxonomical subpart of the particular graph.

5.3 Quality Evaluation

We examine the usefulness of SemSim compared to an extensive set of alternative measures for assessing node similarity, demonstrating the utility of SemSim when used in typical tasks, in which a similarity measure is required.

We used the following baselines from three common approaches for similarity assessment:

I. Structural-based measures: *SimRank* [13], *SimRank++* [2], a weighted variant of SimRank which ignores semantic information, and *Panther* [43], a random-walks based measure which considers edge weights as well.

II. Semantic similarity measures: *Lin* measure [23], as described in Section 2.2.

III. Measures combining structural and semantic information:. First, we employed *LINE* [38], an ML, node embedding-based similarity measure which accounts for latent semantic relations among the graph nodes. This serves a representative example for the state-of-the-art approach for assessing node similarity. Additionally, we tested *PathSim* [37], a HIN-dedicated similarity measure, which considers edge labels, and *Relatedness* [25] a semantic-aware measure which considers the properties' relating concepts. Last, we employed the *Multiplication* and *Average* competitors, returning the product (resp., average) of independent structural and semantic scores obtained by SimRank and Lin. These measures serve as baselines to our approach that interweaves structure and semantics throughout a recursive computation.

These baselines were examined in typical tasks in which a similarity measure is required: *Term Relatedness*, a problem requiring a measure aware of both semantic and structural knowledge, (tested on Wikipedia and WordNet datasets); *Link Prediction*, in which we used the measures to predict co-purchases in Amazon dataset, and *Entity Resolution* was tested on AMiner dataset, to detect duplication of entities. A ground truth was defined for each task, used to quantitatively evaluate the effectiveness of each competitor.

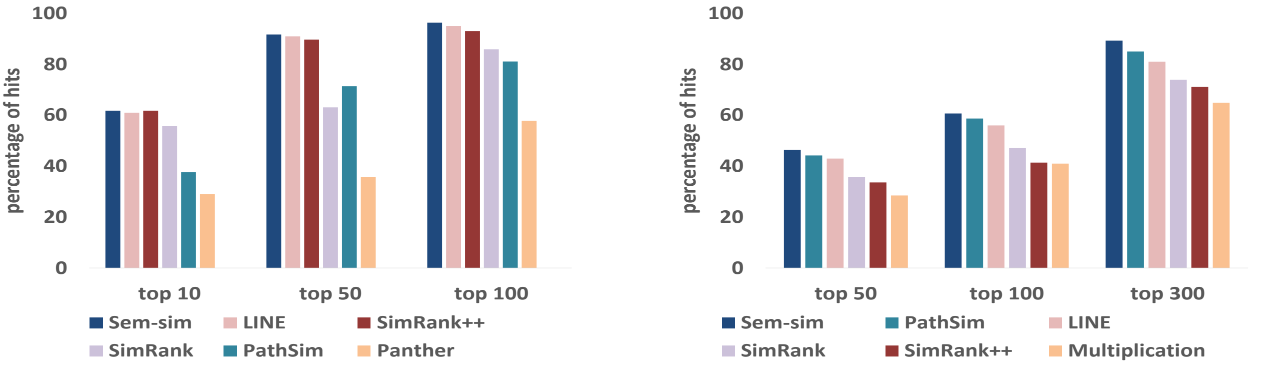
Method	r (Wiki)	p (Wiki)	r (WN)	p (WN)
Panther	0.323	0.0376	0.206	10^{-3}
PathSim	0.293	0.0662	0.332	10^{-3}
Simrank	0.295	0.0641	0.397	10^{-4}
Simrank++	0.296	0.0644	0.395	10^{-4}
Average	0.36	0.0514	0.401	10^{-4}
Multiplication	0.37	0.0508	0.409	10^{-4}
Lin	0.485	0.0015	0.449	10^{-4}
LINE	0.493	0.0001	0.470	10^{-4}
Relatedness	0.510	0.0007	0.488	10^{-4}
SemSim	**0.585**	0.0001	**0.501**	10^{-4}

Table 5: Pearson's r and p-value in the WordsSim-test on Wikipedia (Wiki) and WordNet (WN).

Term Relatedness. Relatedness between terms is a well studied problem that requires a measure aware of both semantic and structural knowledge. To examine the adequacy of SemSim for capturing relatedness, we used two datasets that contain relations between terms: Wikipedia and WordNet. The ground truth was defined by the WordsSim-353 test [8], a public and commonly used benchmark containing pairs of words alongside their *relatedness* scores, as computed by people (e.g. "computer-keyboard" has the score of 0.76). We then compared the scores obtained by each competitor, using the Pearson correlation measure (a commonly used measure to evaluate the accuracy result for this benchmark [25]). We removed pairs of words that were missing in the graph from the benchmark, retaining 40 pairs for Wikipedia and 342 for WordNet. Table 5 depicts the results for all baselines. We note that other corpus-based designated methods were suggested for this task (e.g., [42]), but they require external sources besides the input graph, thus we did not include them in our benchmark. Observe that the structural based measures (e.g., SimRank, SimRank++ and Panther) demonstrate inferior results, as this task greatly relies on the semantic relations among the concepts. Furthermore, naïve semantic measures such as Lin, that perform a rather simplistic similarity comparison (i.e., rely only on the taxonomy "is-a" edges), surpassed both the structural measures, and the the *Average and Multiplication* competitors, yet were outperformed by LINE and the Relatedness measure, which better combine the structural and semantic aspects, and consider all edges in the graph. The Relatedness measure, designated specifically for this task, exceeded the ML-based measure LINE, yet interestingly, it was surpassed by SemSim.

Link Prediction. We next demonstrate how SemSim may be used to predict co-purchases in the Amazon dataset. To compare between different baselines, we omitted 7.5K edges between items from the original dataset, and examined how well the measures can be used to predict those missing links as follows: Given an endpoint of a removed link, we performed a top-k search to find similar nodes to the given endpoint. If, for a given measure, the returned k nodes include the pair endpoint, we counted a "hit", and otherwise a "miss". A similar idea was employed to evaluate similarity search in [43]. The results are depicted in Figure 5(a). For compactness, we omitted measures with particularly low scores. As opposed to the *Relatedness* task, this task relies mostly on structural knowledge, hence structural-based measures (e.g., SimRank++, Panther) outperformed the semantic-based ones (e.g., Lin). Here, LINE was able to outperform most competitors, yet SemSim managed to obtain a slight advantage,

[4] Storage optimization techniques previously developed for SimRank can be directly applied in our settings as well (e.g. [14, 34]).



(a) Link Prediction **(b) Entity resolution**

Figure 5: Prediction in top-k

due to the additional semantic information it accounts for which LINE ignores (i.e., the IC values and the taxonomy relations).

Entity Resolution. Last, we used the similarity measures to identify multiple distinct entries representing the same author (e.g, *Susan B. Davidson* and *Susan Davidson*), or the same term (e.g, *Data structures* and *Data structure*). Using the Levenshtein string distance, we identified 30 pairs representing the same entry (24 term-pairs and 6 author-pairs), and used the baselines to predict duplicate nodes following similar lines to the link-prediction task (i.e., a top-k similarity search). The results are depicted in Figure 5(b), reporting for each baseline the precision in top k, for various values of k, (here again, measures with particularly low were omitted). First, note that the results for all baselines are not particularly high, since information commonly used for entity resolution (such as mail addresses, affiliations, string edit distance, etc.) was not included in the graph. As in the link-prediction task, the structural based measures outperformed the semantic based one (omitted from presentation). This stems from the particular characteristics of the AMiner graph, in which the semantic similarity of all authors nodes is identical (i.e., all authors nodes "is-a" *Author*). Hence, the semantic similarity between authors in this setting is not informative. Here again, the Multiplication/Average baselines demonstrate inferior results. It should be pointed out that PathSim outperforms most competitors, as it is a structural measure that also considers the edge labels, thus accounts for some semantics. However, SemSim managed to get an advantage, even if sometimes marginal, for all tested values of k.

In summery, as demonstrated, SemSim outperformed the competitors in all tasks mentioned above. In some tasks the advantage was marginal compared to the second-best for a specific task, yet we note that the second-best competitor was *different* in each task, illustrating the robustness of SemSim. For the rest of the baselines, results varied depending on the amount of semantics conveyed in the dataset, and of the degree to which the task is semantically complex. To conclude, the experiments indicate that even in cases where only partial semantics is available, SemSim serves as a robust measure and exploits this information to get an edge over the competitors.

6 RELATED WORK

SimRank is a popular measure and its potency was demonstrated in various scenarios [2, 6]. Several extensions that enrich it with more information (e.g. edge weights [2], particular paths [6] or second order walks [41]) were suggested, but they do not make full use of semantics available and thus, as illustrated in our experiments, yield less accurate estimations in semantic-sensitive tasks. Moreover, the optimization technique used in SimRank++ was build on *matrix multiplication* rather than random walks, and consequently, scalability issues were ignored. One of the contributions of this work is an efficient computation scheme, applicable also to several of these variants (e.g. [2, 45]).

As mentioned, a prominent body of work has focused on SimRank approximation techniques. These works are categorized into (a) matrix-based approaches [19], and (b) random walk-based approaches [14, 39]. A recent survey [44] advocates that the latter approach is more scalable, compatible with updates in the graph, and can be trivially parallelized. Therefore, we chose to extend it in accordance with our setting. As mentioned in Section **??**, the contributions of these works are applicable for SemSim, requiring only minor adaptations.

In our experiments we use Lin [23], a simple and effective *Information Content* (IC)-based semantic measure. However, as explained in Section 2, any other measure can be incorporated, given that it satisfies three intuitive constraints. Examples of other applicable semantic measures include: (i) IC-based measures [32], (ii) Edge-counting measures, which use the length of the shortest path between nodes in the estimation of similarity [31], and (iii) Feature-based measures [20, 42]. The former two regard a domain ontology, while the latter usually involves additional sources (e.g. textual corpus [20]).

Heterogeneous Information Network (HIN) is a ubiquitous model for real-world data, as it enables enriching simple graphs with additional useful information [36]. This, however, makes the assessment of node similarity challenging, as HIN paths convey latent semantic information. The majority of existing similarity measures for general networks do not consider all available information in the HIN. Specifically, measures such as [13, 43, 45] focus solely on the network structure, while measures suggested in [23, 32] concern only with the semantic information as implied by hierarchical relations. More recent works propose HIN-dedicated measures [35, 37], advocate considering only meaningful meta-paths between objects. But the choice of appropriate paths is made a-priori, and requires intimate knowledge of the dataset and the specific information needs[5] [22]. In contrast, SemSim is a generic measure that encompasses all available information, and automatically prioritizes meaningful paths.

As opposed to the declarative approach, recent work in the field of *representation learning* [4, 30] suggests *embedding* techniques that discover low-dimensional representations of graph nodes in a vector space. While this approach often outperforms

[5] Otherwise, an average of all paths can be taken, resulting in inferior results.

dedicated similarity measures for HINs (as demonstrated), a key disadvantage is that the results are hard to explain and interpret. Moreover, as we showed, SemSim not only retains its declarative definition but also yields more accurate similarity estimations in multiple tasks.

Incorporating semantic and structural information when determining relations between graph objects has also been proven useful in several related domains. Works in *ontology matching* and *entity resolution* suggest using both taxonomy edges and structural properties of nodes to properly align entities [12, 28]. However, their goal is different, as they aim to identify equivalent representations of the same entity, thus some of the core techniques employed (e.g., string matching) cannot be directly applied for measuring similarity between different objects. It would be interesting to investigate in future work whether SemSim can be employed in such contexts. An example domain where we showed such incorporation to be successful is similarity estimation for images that convey semantics, as we illustrated in [7], a demo that employed SemSim for the retrieval of similar Internet Memes.

7 CONCLUSION AND FUTURE WORK

In this paper we present SemSim, a similarity measure that refines SimRank with semantics, while preserving its intuitive definition and scalable computation. We introduce Semantic-Aware Random Walks, an extension of the traditional notion of random walks, that preserves properties necessary for applying existing SimRank's optimizations. Our probabilistic framework employs Importance Sampling along with an effective pruning technique, and maintains a negligible error rate. Our experiments further demonstrate the quality and robustness of SemSim in multiple practical scenarios, as well as the efficiency of our algorithms.

Several interesting directions are left for future research. First, in practice, information networks are often dynamic and may induce uncertainty, hence it would be important to extend SemSim to such settings. The use of parallelism and compact indexing mechanisms [3, 21], to achieve further computation speedup, are also an interesting direction for future work. Last, we have focused here only on single-pair queries. We intend on developing optimizations facilitating single-source and top-k similarity queries, inspired by [17, 46].

REFERENCES

[1] aminer [n. d.]. AMiner. https://aminer.org/data. ([n. d.]).

[2] Ioannis Antonellis, Hector Garcia Molina, and Chi Chao Chang. 2008. Simrank++: query rewriting through link analysis of the click graph. *PVLDB* (2008).

[3] Yuanzhe Cai, Gao Cong, Xu Jia, Hongyan Liu, Jun He, Jiaheng Lu, and Xiaoyong Du. 2009. Efficient algorithm for computing link-based similarity in real world networks. In *ICDM'09*. IEEE.

[4] Shiyu Chang, Wei Han, Jiliang Tang, Guo-Jun Qi, Charu C Aggarwal, and Thomas S Huang. 2015. Heterogeneous network embedding via deep architectures. In *SIGKDD*.

[5] dbpedia [n. d.]. DBpedia. http://dbpedia.org/About. ([n. d.]).

[6] Beate Dorow, Florian Laws, Lukas Michelbacher, Christian Scheible, and Jason Utt. 2009. A graph-theoretic algorithm for automatic extension of translation lexicons. In *GEMS*. Association for Computational Linguistics.

[7] Maya Ekron, Tova Milo, and Brit Youngmann. 2017. SimMeme: Semantic-Based Meme Search [Demonstration]. In *CIKM (CIKM '17)*.

[8] Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. 2001. Placing search in context: The concept revisited. In *WWW*.

[9] Dániel Fogaras and Balázs Rácz. 2005. Scaling link-based similarity search. In *WWW*.

[10] Peter W Glynn and Donald L Iglehart. 1989. Importance sampling for stochastic simulations. *Management Science* (1989).

[11] Dov Harel and Robert Endre Tarjan. 1984. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing* (1984).

[12] Wei Hu, Jianfeng Chen, and Yuzhong Qu. 2011. A Self-training Approach for Resolving Object Coreference on the Semantic Web. In *WWW (WWW '11)*.

[13] Glen Jeh and Jennifer Widom. 2002. SimRank: a measure of structural-context similarity. In *SIGKDD*.

[14] Minhao Jiang, Ada Wai-Chee Fu, and Raymond Chi-Wing Wong. 2017. READS: a random walk approach for efficient and accurate dynamic SimRank. *PVLDB* (2017).

[15] Mitsuru Kusumoto, Takanori Maehara, and Ken-ichi Kawarabayashi. 2014. Scalable similarity search for SimRank. In *SIGMOD*.

[16] Juan J Lastra-Díaz, Ana García-Serrano, Montserrat Batet, Miriam Fernández, and Fernando Chirigati. 2017. HESML: A scalable ontology-based semantic similarity measures library with a set of reproducible experiments and a replication dataset. *Information Systems* (2017).

[17] Pei Lee, Laks VS Lakshmanan, and Jeffrey Xu Yu. 2012. On top-k structural similarity search. In *ICDE*.

[18] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (2014).

[19] Cuiping Li, Jiawei Han, Guoming He, Xin Jin, Yizhou Sun, Yintao Yu, and Tianyi Wu. 2010. Fast computation of simrank for static and dynamic information networks. In *EDBT*.

[20] Yuhua Li, Zuhair A Bandar, and David McLean. 2003. An approach for measuring semantic similarity between words using multiple information sources. *TKDE* (2003).

[21] Zhenguo Li, Yixiang Fang, Qin Liu, Jiefeng Cheng, Reynold Cheng, and John Lui. 2015. Walking in the cloud: Parallel simrank at scale. *PVLDB* (2015).

[22] Jiongqian Liang, Deepak Ajwani, Patrick K Nicholson, Alessandra Sala, and Srinivasan Parthasarathy. 2016. What Links Alice and Bob?: Matching and Ranking Semantic Patterns in Heterogeneous Networks. In *WWW*.

[23] Dekang Lin. 1998. An information-theoretic definition of similarity.. In *ICML*.

[24] Dmitry Lizorkin, Pavel Velikhov, Maxim Grinev, and Denis Turdakov. 2008. Accuracy estimate and optimization techniques for simrank computation. *PVLDB* (2008).

[25] Laurent Mazuel and Nicolas Sabouret. 2008. Semantic relatedness measure using object properties in an ontology. In *ISWC*.

[26] George A Miller. 1995. WordNet: a lexical database for English. *ACM* (1995).

[27] Tova Milo, Amit Somech, and Brit Youngmann. 2018. Technical report. http://courses.cs.tau.ac.il/software1/1617b/misc/main-semsim-full.pdf. (2018).

[28] DuyHoa Ngo and Zohra Bellahsene. 2016. Overview of YAM++ -(not) Yet Another Matcher for ontology alignment task. *Web Semantics: Science, Services and Agents on the World Wide Web* (2016).

[29] Normalization [n. d.]. Wikipedia: Normalization. https://en.wikipedia.org/wiki/Normalization_(statistics). ([n. d.]).

[30] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *SIGKDD*.

[31] Roy Rada, Hafedh Mili, Ellen Bicknell, and Maria Blettner. 1989. Development and application of a metric on semantic nets. *IEEE Transactions on systems, man, and cybernetics* (1989).

[32] Philip Resnik. 1995. Using information content to evaluate semantic similarity in a taxonomy. *arXiv preprint cmp-lg/9511007* (1995).

[33] Nuno Seco, Tony Veale, and Jer Hayes. 2004. An intrinsic information content metric for semantic similarity in WordNet. In *ECAI*.

[34] Yingxia Shao, Bin Cui, Lei Chen, Mingming Liu, and Xing Xie. 2015. An efficient similarity search framework for SimRank over large dynamic graphs. *PVLDB* (2015).

[35] Chuan Shi, Xiangnan Kong, Yue Huang, S Yu Philip, and Bin Wu. 2014. Hetesim: A general framework for relevance measure in heterogeneous networks. *TKDE* (2014).

[36] Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, and S Yu Philip. 2017. A survey of heterogeneous information network analysis. *TKDE* (2017).

[37] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S Yu, and Tianyi Wu. 2011. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *PVLDB* (2011).

[38] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. Line: Large-scale information network embedding. In *WWW*. WWW.

[39] Boyu Tian and Xiaokui Xiao. 2016. SLING: A near-optimal index structure for simrank. In *SIGMOD*.

[40] Wikipedia [n. d.]. Wikipedia: SimRank. https://en.wikipedia.org/wiki/SimRank. ([n. d.]).

[41] Yubao Wu, Yuchen Bian, and Xiang Zhang. 2016. Remember Where You Came from: On the Second-order Random Walk Based Proximity Measures. *PVLDB* (2016).

[42] Wen-tau Yih and Vahed Qazvinian. 2012. Measuring word relatedness using heterogeneous vector space models. In *NAACL HLT*.

[43] Jing Zhang, Jie Tang, Cong Ma, Hanghang Tong, Yu Jing, and Juanzi Li. 2015. Panther: Fast top-k similarity search on large networks. In *SIGKDD*.

[44] Zhipeng Zhang, Yingxia Shao, Bin Cui, and Ce Zhang. 2017. An Experimental Evaluation of SimRank-based Similarity Search Algorithms. *PVLDB* (2017).

[45] Peixiang Zhao, Jiawei Han, and Yizhou Sun. 2009. P-Rank: a comprehensive structural similarity measure over information networks. In *CIKM*.

[46] Weiguo Zheng, Lei Zou, Yansong Feng, Lei Chen, and Dongyan Zhao. 2013. Efficient simrank-based similarity join over large graphs. *PVLDB* (2013).

Leveraging Bitmap Indexing for Subgraph Searching

David Luaces
Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela (USC)
Santiago de Compostela, Spain
david.luaces@usc.es

José R.R. Viqueira
Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela (USC)
Santiago de Compostela, Spain
jrr.viqueira@usc.es

Tomás F. Pena
Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela (USC)
Santiago de Compostela, Spain
tf.pena@usc.es

José M. Cotos
Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela (USC)
Santiago de Compostela, Spain
manel.cotos@usc.es

ABSTRACT

Deciding whether a query graph is a subgraph of some other in a very large database of small graphs is a problem of major interest in many application domains. As an example, it arises in the searching for specific molecular substructures in currently available molecular databases, whose sizes may reach levels close to one hundred million. State of the art methods to solve this problem follow a *filter-then-verify* (FTV) paradigm, where an indexing technique is first used in a filtering stage to obtain result candidates and a subgraph isomorphism algorithm is next applied to the candidates in a verification stage to obtain the final result. Among all the available techniques of the state of the art, two of them have demonstrated better performance when applied to large datasets, namely, the GraphGrepSX (GGSX) and CT-Index (CTI). In this paper, three new indexing techniques, one based on GGSX and two based on CTI, are proposed. In particular, Bitmap GGSX (BM-GGSX) leverages the use of bitmaps in the trie structure used by GGSX to achieve performance gains of around 90% in the filtering stage. Column-Wise CT-Index (CW-CTI) exploits a column-wise representation of the fingerprints (bitmaps) used by CT-Index to reduce the filtering times around 80% for small queries (8 edges). Finally, K-Means CT-Index (KM-CTI), constructs a binary tree of bitmaps from the CT-Index fingerprints to reach filtering time reductions of around 70% for medium queries (20 edges) and 75% for large queries (40 edges).

1 INTRODUCTION

Graph database research is a topic to which much attention has been paid by the data management community during the last decade. Two major types of graph databases may be found in real problems. In a first type, the database consists of just one very large graph. This is, for example, the kind of database available in the semantic web. Amongst the required query functionality, many applications need to identify all the instances of a specific subgraph that occurs inside the database graph. This NP-complete problem, known in the literature as *subgraph matching*, is solved through the application of *subgraph isomorphism* algorithms [4, 12, 20, 21, 26, 27].

A second type of database contains a very large number of small graphs. Typical examples of this type of databases are molecular databases. For each molecule recorded in such a database it is recorded, among other properties, its structure, i.e., a small graph whose vertices are atoms and whose edges are bonds between atoms. The current size of these databases is already very large, reaching levels close to one hundred million molecules, like in the PubChem dataset[1], and it is increasing. Among other queries of interests, it is important to provide functionality to find all the molecules whose molecular structure (graph) contains a specific query substructure (subgraph) [5, 6].

Finding all the graphs in a database that contain a specific subgraph is known in the literature as the *subgraph decision* problem. A straightforward strategy to solve this problem is a linear search, i.e., the application of a *subgraph isomorphism* algorithm to each graph of the database. This strategy could be the appropriate for not selective queries that return most of the database graphs. Besides, given its simplicity, it is also straightforward to a obtain parallel implementation following this strategy. However, in the general case, a filter-then-verify (FTV) strategy based on indexing will get much better performance. In FTV approaches, query processing is split in tho stages, namely *filtering* and *verification*. In the filtering stage, an index structure is searched to obtain an initial set of candidate graphs. In the second stage, a subgraph isomorphism algorithm is applied to each candidate to refine the final result. Many FTV methods have been proposed in the literature [2, 3, 8, 15, 20, 24, 25, 28, 29].

Among the proposed FTV solutions, GraphGrepSX (GGSX) [2] and CT-Index (CTI) [15] have demonstrated an excellent performance when applied to large datasets of small graphs, as it is the case of molecular databases. Both of them rely on the encoding, in the index structures used during the filtering stage, of features contained in the graph, such as paths, trees, and cycles. In particular, GGSX generates a trie structure with all the paths contained in each graph up to a maximum length. Each node of the trie represents a specific path and references all the graphs of the database that contain such path, storing also the number of times that the path is repeated in each graph. During the filtering stage, the database trie is used to obtain all the graphs that contain all the paths of the query up to a maximum length, at least the number of times that the path is contained in the query.

The indexing technique used by CTI is completely different. CTI generates a fingerprint (bitmap) for each database graph

[1] https://pubchem.ncbi.nlm.nih.gov/

as follows. First, graph features are extracted. CTI may work with either paths or trees, and with cycles. Each feature of the graph up to a maximum length is extracted, and represented in a canonical string format. A hash function is next applied to each extracted string to obtain integer numbers that range from 0 to the fingerprint length. The bits of the fingerprint located at the positions defined by those integers are set to 1. In the filtering stage, bitmap containment is tested between the query fingerprint and the fingerprint of each database graph.

In this paper, three new FTV solutions are proposed for subgraph search in large databases of small graphs. The paper focus on the filtering stage, improving GGSX and CTI structures and algorithms to obtain filtering time reductions between 70% and 80%, whereas the verification stage of the three techniques relies on the parallel execution in a multi-thread architecture of a improved version of the VF2 [4] subgraph isomorphism algorithm already proposed by CTI [15]. In particular, the main contributions of the present work are summarized as follows.

1. Bitmap GGSX (BM-GGSX) takes advantage of the incorporation of compressed bitmaps in the representation of graph references in the GGSX trie nodes. This enables a drastic reduction of the filtering time (around 90% less than GGSX), maintaining the index size and building time almost unaltered. When compared with the techniques based on CT-Index also proposed in this paper, BM-GGSX offers good response times for medium and large queries, but it is worse for small queries. Besides, BM-GGSX is also much worse in both index size and building time.
2. Column-Wise CT-Index (CW-CTI) adopts a column-wise storage structure for the CT-Index fingerprints, where the same bit position of all the fingerprints of all the graphs are recorded together in a compressed bitmap. During the filtering stage, only the bit positions with a 1 in the query fingerprint have to be accessed, which makes the filtering time dependent on the query size. Regarding index size and building time, CW-CTI gets the best results.
3. K-Means CT-Index (KM-CTI) applies recursively the K-Means clustering method on CT-Index fingerprints to construct a bitmap binary tree that is used to discard sets of fingerprints during the filtering stage. KM-CTI index size and building time is worse than CW-CTI, but better than BM-GGSX. Regarding filtering time, it is competitive with BM-GGSX in medium and large queries, thus it is a good complement of CW-CTI.
4. An extensive evaluation of the performance of all the techniques is undertaken using real databases of molecular graphs of different sizes. A first experiment is conducted to determine the best parameter values to tune the methods. The objective of the second experiment is to test the scalability of the techniques with increasing database sizes. Real databases with sizes ranging from 200000 to one million molecules were built from the PubChem dataset. Notice that those sizes are much larger than even the synthetic datasets used in previous surveys [11, 13].

The remainder of this paper is organized as follows. Section 2 reviews related work, with special attention to the original GGSX and CT-Index methods. The proposed BM-GGSX, CW-CTI and KM-CTI methods are described in detail in Sections 3, 4 and 5, respectively. Section 6 is devoted to the discussion of the evaluation experiments and, finally, Section 7 concludes the paper and outlines some lines of potential future work.

2 RELATED WORK

The current state of the art classifies the subgraph querying problem into two different subproblems. The first one, named *matching problem*, deals with extracting all subgraph isomorphic embeddings of a query graph q in a single graph G. The second one, usually called *decision problem*, is focused in retrieving the ID of every graph g, stored in a given dataset G, which satisfies that the query graph q is a subgraph of g.

The matching problem has been widely studied over the years, and there are several proposals to solve it. As mentioned in the introduction, a subgraph isomorphism algorithm must be applied to ensure that the verified graph contains at least one subgraph that matches exactly the query graph. In [16], the authors provide a performance comparison of six of the most relevant solutions at the time. Among these algorithms, Ullmann [21], VF2 [4], and QuickSI [20] were originally designed for handling small graphs, while GraphQL [12], GADDI [26], and SPath [27] were designed for handling large graphs. The tests were accomplished in four real-world datasets with different size and characteristics, containing two of them multiple and small graphs, and the other two single and relatively large graphs. The study concluded that GraphQL was the only algorithm that completed all the queries, while QuickSI performed the best for both small and large data graphs, even though it was designed for handling small graphs, since the cost of its recursive call is the lowest. The study also concluded that all existing algorithms had problems in their join order selections. Since the survey was published, new algorithms designed for handling large graphs have been proposed [10, 19], showing an improvement in the performance by addressing the issues of matching order selection. Recently, a new approach [1] proposes a new framework to minimize the redundant Cartesian products.

The *decision problem* is typically arising in the scope of applications with big datasets of small graphs. Since this is a NP-complete problem, and datasets contain often a large number of graphs, deciding whether a query graph q is contained in every graph g in the dataset G, by applying one by one a subgraph isomorphism algorithm would be very costly. Due to this, a pruning phase must be undertaken in an early stage of the process in order to select a reasonable number of candidates to be tested with a subgraph isomorphism algorithm. This idea is carried out by the techniques based in the filter-then-verify (FTV) paradigm. FTV techniques rely in building an index of the graph dataset, through decomposing each graph into features (i.e., paths, trees, cycles, etc.), and store them in an appropriate structure (e.g., trie, fingerprints, etc.). The search algorithm of FTV techniques is divided into two different stages. The first stage, called *filtering stage*, aims at obtaining a *candidate set* of graphs. To do this, the query graph q is decomposed into its corresponding features, according to the applied technique, which are used to retrieve from the index a reduced *candidate set* with the graphs that contain all the query features. The *candidate set* is tested finally with a subgraph isomorphism algorithm in the second stage, called *verifying stage*.

A performance comparison of different indexing techniques [3, 20, 24, 25, 28, 29] for subgraph query processing is provided in [11]. The above survey is extended in [13], by including three new algorithms [2, 8, 15] and also by performing an exhaustive performance and scalability study, varying different dataset features such as number of nodes and graphs. According to the conclusions of [13], CT-Index [15] and gCode [29] have the smallest index size. Regarding query processing time, this study concludes that the approaches that build the index with graph features (Grapes [8], GraphGrepSX [2], and CT-Index [15]) outperform the others, and, among them, those that use simpler features (Grapes and GraphGrepSX) obtain the best results. The techniques that show a better scalability are GraphGrepSX, Grapes, and CT-Index. However, Grapes, fails to build an index for large datasets due its memory requirements. Finally, the survey shows that algorithms based in mining techniques (gIndex [24] and Tree+Δ [28]) are only competitive for small datasets.

Recently, two new methods that employ caching on top of existing FTV techniques have been proposed (iGQ [22] and GraphCache [23]). They take advantage of the fact that, in real-world scenarios, most current queries have subgraph or super-graph relations with the future ones. Therefore, they use a caching system with past queries and their answers, including some novel replacing strategies, to improve the performance of existing methods. Another recent work [14], after analyzing both matching and decision problems, it concludes that there exist algorithm specific straggler queries that are challenging only for specific approaches. Based on the above observation, a novel framework is proposed in [14] that leverages parallel execution and query rewriting to achieve better overall performance.

Based on the conclusions of [13], we select GraphGrepSX and CT-Index as the base of the present work, to develop improved indexing structures to solve the decision problem. Notice that our experiments consider databases whose size is between 10 and 20 times larger than those of [13], and, therefore, we discarded Grapes due to its problems with large databases. Finally, the results of the present work may be incorporated in complex query processing frameworks, as the one proposed in [14], which may also leverage in caching techniques [22, 23].

2.1 GraphGrepSX

GraphGrepSX, GGSX for short henceforth, decomposes each graph in paths up to a maximum length p specified by the user. Repeated paths are taken into account. With these paths, in the index building stage, GGSX incrementally builds a trie with a depth equal to the maximum path length p. Each node of the trie represents a path from the root to that node, and stores a list of key-value pairs, where the key represents the ID of a graph in the dataset that contains the path represented by the node, and the value is the number of times that the path appears in the graph. Figure 1 shows an example of a small trie. Each node of the trie records a large list of key-values pairs, although only two of them are depicted in the figure due to space limitations. Thus, as it is represented at the bottom left node of the trie, path “CCC” is contained 4 times in graph G_1, 5 times in graph G_3 and 3 times in graph G_n. Notice that the size of the main structure of the trie is completely dependent on the chosen maximum path length p, which in practice is a short value. On the other hand, the lists of key-value pairs recorded in each node increase their size linearly with the size of the database.

In GGSX, as in any other FTV technique, subgraph query processing is divided into two stages: filtering and verification. In the filtering stage, a trie is built using all the paths of the query whose length is lower or equal to the chosen maximum length p. The number of repetitions of each path is only recorded now for leaf nodes. The filtering algorithm performs a joint breadth-first traversal of query and database tries. When a query leaf node is reached, the number of repetitions r recorded in the query trie node is compared with the list of repetitions recorded in the database trie node, obtaining the set of graphs that contain at least r times the relevant path. Such list of graphs is maintained to be intersected with subsequent lists obtained from the remainder query trie leaf nodes. The final list of graphs resulting from all the intersections is tested, in the verification stage, using the VF2 subgraph isomorphism algorithm.

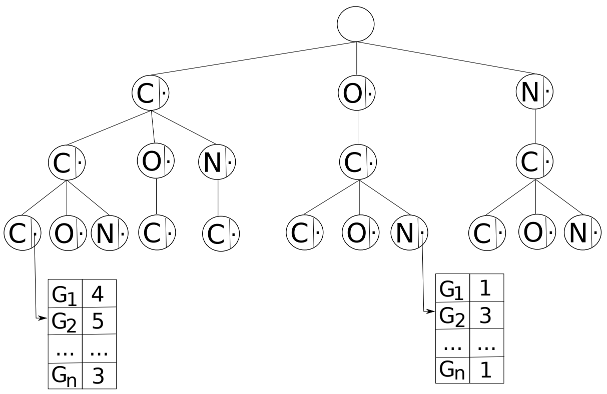


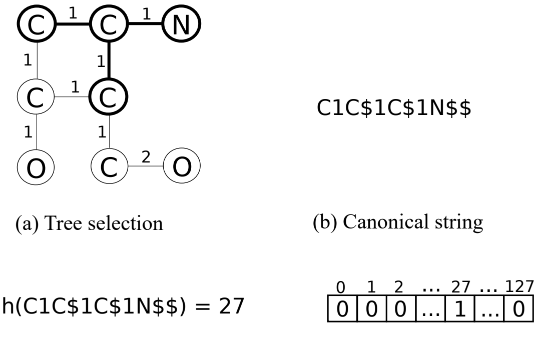
Figure 1: Trie structure used as indexing in GGSX.

Overall, it is expected that the chosen p value should have an impact in different aspects: the index size (should increase with p), the index construction time (should increase with p), the filtering time (should increase with p) and the verification time (should decrease with p). To the best of our knowledge, there is not, currently, any mathematical model for the determination of the best expected p value for a given dataset, therefore, it has to be chosen based on some benchmarking.

2.2 CT-Index

CT-Index uses a list of bitmaps of a fixed size f called fingerprints, for indexing purposes in the filtering phase. Roughly speaking, either paths or trees, and cycles of each graph are hashed to integers between 0 and f, which are next used to activate bits in the graph fingerprint. To generate all the possible either paths or subtrees, and cycles, a maximum length p is again considered. Figure 2 illustrates the creation of a fingerprint from a given graph. First, all possible features, either paths or trees, and cycles of a maximum length p are generated from the graph. A specific tree contained in a graph is depicted in Figure 2(a). Next, the extracted features are encoded in a canonical string format (see Figure 2(b)). A hash function is next applied to the generated canonical string to obtain an integer. Integer number 27 is obtained from the string in the example of Figure 2(c). Finally, the relevant bit of the graph fingerprint is set to 1 (see Figure 2(d)). It has to be noticed that the hash function may generate the same integer for different features (collisions). Such collision have a negative effect, decreasing the number of pruned graphs in the filtering stage.

CT-Index is also a FTV technique, with relevant filtering and verification stages. In the filtering stage, a fingerprint is obtained for the query graph as described above, using the same fingerprint size 2^f and the same feature maximum length p. The query fingerprint is then tested for bitmap containment with each fingerprint in the index. Bitmap containment is performed by first splitting both fingerprints (query FQ and database FD) into a sequence of long integers (query LQ and database LD), and then



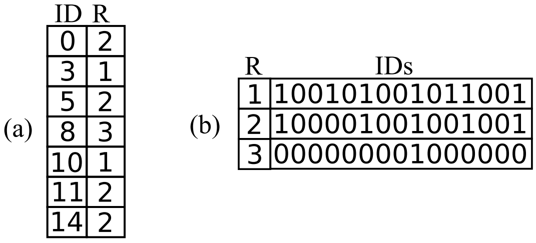
(a) Tree selection

(b) Canonical string

(c) Hashing of feature

(d) Insertion in fingerprint

Figure 2: Fingerprint construction process. In (a) a tree feature is selected. In (b) the feature is transformed into a canonical string. It is hashed in (c). In (d) the hashed value is set in the fingerprint.



(a)

ID	R
0	2
3	1
5	2
8	3
10	1
11	2
14	2

(b)

R	IDs
1	100101001011001
2	100001001001001
3	000000001000000

Figure 3: Comparison between (a) GGSX and (b) BM-GGSX graph repetitions storage technique.

testing bitmap containment between each pair of (LQ, LD) using bitmap operations $(LQ \wedge LD = LQ)$.

CT-Index uses, in the verification stage, a version of the VF2 subgraph isomorphism algorithm, improved with additional heuristics.

It is noticed that with this technique, two parameters may be chosen to construct the index, namely, the maximum feature length p and the fingerprint size 2^f. It is expected that the maximum feature length should not have an impact either in the index size or in the filtering time. However, increasing p should have a positive impact in the verification time (more graphs should be discarded in the filtering phase) and should have a negative impact in the index construction time (more and larger features have to be processed). Regarding the fingerprint size 2^f, on one hand, it has clearly a negative impact in the index size, and therefore in the filtering time (larger fingerprints have to be compared). On the other hand, however, it should have a positive impact in the verification time, by reducing the number of collisions generated by the hash function.

3 BITMAP GGSX

In this section, an evolution of the GGSX method, called Bitmap-GGSX (BM-GGSX), is explained in detail. We modify the original GGSX method in the following three aspects: i) The labels of the graph edges are now considered, improving the performance of the method in graphs such as molecule structures, whose edges contain bound type information (simple bound, double bound, etc.). ii) Very large compressed bitmaps are now exploited to provide a more compact and efficient representation of the list of pairs (graph id, repetitions) recorded in each trie node. iii) The original VF2 subgraph isomorphism algorithm used in the verification stage was replaced by the parallel execution, in a multi-thread architecture, of the improved version of VF2 algorithm provided by the CT-Index implementation.

The original GGSX method discards the edge labels during the construction of the trie structure. On one hand, this reduces the size of the structure, and as a consequence it reduces also the filtering time. But, on the other hand, it increases the number of candidates for the verification stage, which is in general more costly. To incorporate edge labels in the trie, we modify the structure of the trie nodes by adding a new field that records edge labels. Now, the interpretation of a trie node changes from interpreting it as a graph node of a path, as it is in the original GGSX, to interpreting it as the combination of a graph node with the edge that precedes it in the path. Obviously, given that paths start at node and not at edges, the nodes of the first level of the trie have always null edges. Clearly, this new trie structure has more nodes in each level, enabling the representation of a larger number of different paths of the same length.

In the original GGSX method each node of the tree records a list of pairs (i, r), where i is an identifier of a graph (integer index) that contains the path represented by the trie node and r is the number of times that such path appears in the graph. Such lists are replaced in BM-GGSX by an structure that exploits the use of bitmaps. In particular, each trie node records now an array of bitmaps. The bitmap recorded at the index r of the array has a 1 in its position i if, and only if, the graph with identifier i contains the path represented by the trie node at least r times. Figure 3 illustrates the difference between the representation of graph references and repetition in both GGSX and BM-GGSX. Bitmaps are compressed in BM-GGSX using the Enhanced Word-Aligned Hybrid method [17].

The algorithm of the filtering stage that exploits the above structure is illustrated in Figure 4. First, as it is shown in the left part of the figure, the query is processed to obtain its trie structure. Remember that, now, each node contains a pair of (node label, edge label) and that the number of repetitions in the query trie is recorded only in the leaf nodes. As in the case of GGSX, a joint breadth-first traversal of both query and database tries is performed, obtaining now, for each leaf node of the query trie, the bitmap recorded in the position of the array whose index matches the number or repetitions recorded in the leaf node. This is illustrated in the central part of the figure. Finally, a binary AND operation is performed between all the compressed bitmaps, to obtain the result bitmap, which will contain a 1 in the positions of all the graphs containing, at least the number of times required, all the paths of maximum length of the query.

Notice that, as it is shown in Figure 3, if the bitmap recorded at the index r of the array of a trie node has a 1 in position i, then all the bitmaps recorded in indices lower than r must also have a 1 at position i. This inserts redundant information in the structure, but, as a consequence, it also enables better performance during query processing, since only one of the bitmaps of each candidate node of the trie has to be accessed.

4 COLUMN-WISE CT-INDEX

The index structure used by CT-Index consists of one fingerprint for each graph of the database. Those fingerprints are sequentially generated and recorded during the index construction and sequentially processed during the query evaluation. The collection of all fingerprints might be seen as a matrix, where each

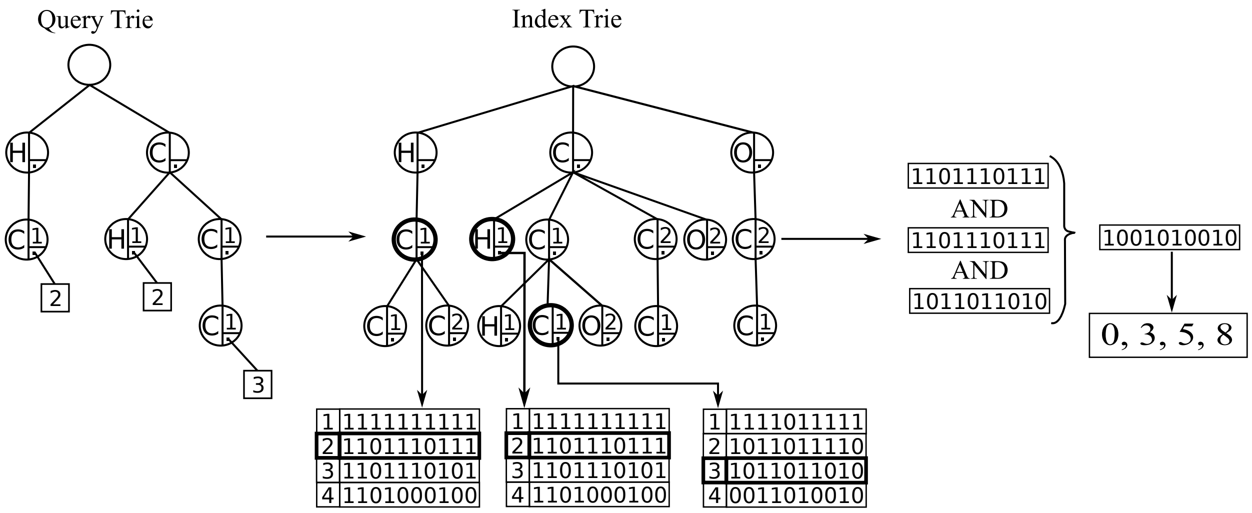


Figure 4: Example of filtering stage in BM-GGSX.

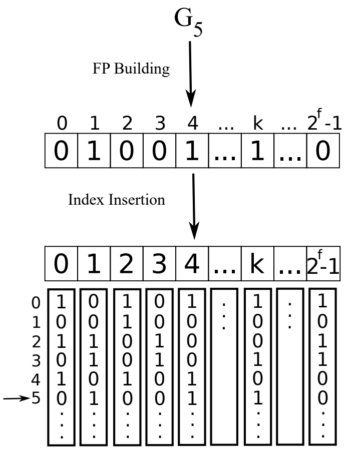


Figure 5: Example of a new graph(ID = 5) insertion in the CW-CTI index.

row is a fingerprint and each column a bit position inside the fingerprint. It is noticed that such matrix is recorded row-wise in CT-Index. The Column-Wise CT-Index, in short CW-CTI, leverages the use of compressed bitmaps by recording and processing the fingerprints column-wise. Thus, if 2^f is the chosen size for the fingerprints, then the index of CW-CTI is a sequence of 2^f bitmaps, such that fingerprint number b has a 1 at position i if, and only if, the fingerprint of molecule number i has a 1 in the position b of its fingerprint. The bitmaps of CW-CTI are much larger than the fingerprints of CT-Index, and they are likely to contain larger sequences of repeated bits, so, they are good candidates for the use of bitmap compression. Enhanced Word-Aligned Hybrid compression [17] is used in the present implementation, which enables the reduction of index size compared to the original CT-Index.

The building stage of CW-CTI is similar to that of CT-Index. Each fingerprint is constructed exactly in the same way, however, now each bit of the obtained fingerprint is appended to the end of each corresponding CW-CTI bitmap. This process is illustrated in Figure 5 for a specific graph G_5.

Figure 6 illustrates the algorithm of the filtering stage of CW-CTI. It is noticed that the algorithm is completely different from the one of CT-Index. First, the fingerprint of the query is obtained in the same way that it is obtained for CT-Index. Next, for each 1 in the query fingerprint, the compressed bitmap recorded at the relevant column of the CW-CTI structure is obtained. Remember that such compressed bitmap contains a 1 for each graph whose fingerprint has a 1 in the same position. Finally, all the obtained bitmaps are intersected to produce the final result of the filtering stage. The result compressed bitmap will contain a 1 for each candidate graph, i.e., each graph whose fingerprint is binary contained in the query fingerprint.

The verification stage in CW-CTI performs a subgraph isomorphism test to the final candidate set of graphs with the improved version of the VF2 used in the CT-Index method, except that now this algorithm is executed in parallel in a multi-thread architecture.

As it was already stated, CW-CTI requires, in general, less storage than the original CTI-Index, due to the use of bitmap compression. The number of candidates obtained from the filtering stage is exactly the same as the one obtained by CTI, however, the verification time should be better, due to the use of a parallel execution of the VF2 algorithm. Regarding the filtering time, it is noticed that the number of binary *AND* operations between compressed bitmaps required is directly determined by the number of 1s in the query, i.e., by the number of different features in the query, which is higher as the query size increases. Therefore, it is expected to behave better with smaller queries.

5 K-MEANS CT-INDEX

In this section K-Means CT-Index (KM-CTI), a new proposal based in the CT-Index method, is described. This new method was constructed with the aim of reducing the number of fingerprint comparisons made in the filtering stage of the CT-Index algorithm. It is reminded that CT-Index performs a comparison between the query fingerprint and each of the fingerprints of the database, therefore, it is expected that the filtering time will increase linearly with the database size. KM-CTI uses a binary tree of bitmaps, where the leaf nodes correspond to the database fingerprints, and parent nodes are constructed by performing

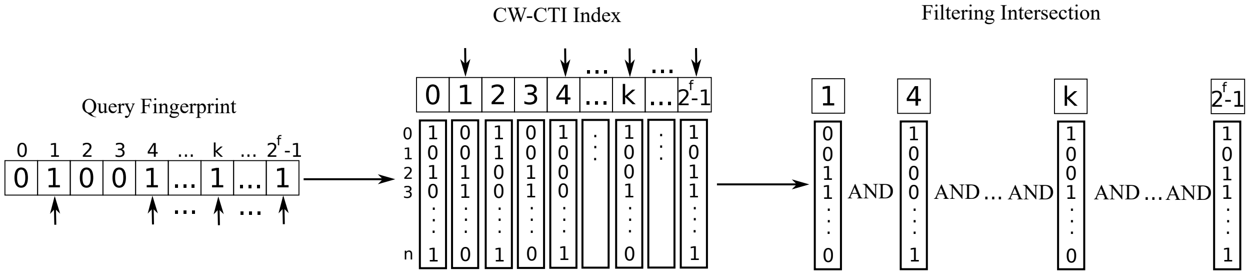


Figure 6: Example of filtering stage in CW-CTI.

the binary *OR* operation between two children. If the comparison between the query fingerprint and a given node of the tree returns false, then the whole branch below the node may be directly discarded. Thus, if the database is large enough, then searching the KM-CTI binary tree will require less comparisons than the number of database fingerprints, reducing the time of the filtering stage.

An example of the use of a KM-CTI binary tree to evaluate the fingerprint of an input query is illustrated in Figure 7. It is noticed that, in the binary tree, all the fingerprints are in the leaf nodes, and each parent node contains the binary *OR* of its children. The query is first compared with the root node. Given that the query is binary contained in the root node, then it has to be compared with each of the two children. The comparison with the left children is positive, and therefore the process has to continue through the left branch. However, the comparison with the right child is negative, therefore the whole branch may be discarded. In the figure, highlighted nodes have been compared with the query and nodes of dark color represent positive results in the comparison. Therefore, in this example, the algorithm performs 7 comparisons (instead of the 8 comparisons that the original CI-Index would require) to select two fingerprint candidates for the verification stage. Again, the chosen algorithm for the verification stage is the VF2 version proposed by the original CT-Index method, and again, this algorithm is executed in parallel in a multi-thread architecture.

The KM-CTI binary tree will offer a good performance if the internal nodes discard large numbers of fingerprints with few comparisons. To achieve this, pairs of bitmaps have to be combined under parent nodes in a way that minimizes as much as possible the number of 1s in those internal nodes. This behavior during the tree construction is achieved in KM-CTI through the application of the K-Means [7, 18] clustering algorithm, recursively to the original set of fingerprints, as it is explained below.

In general, K-Means is used to distribute a set of elements in K clusters, in a way that a given distance measure is minimized between the elements of each cluster. Broadly speaking, the algorithm works as follows. First, K elements of the set are aleatory chosen as centroids of the K clusters. Next, each element is assigned to the cluster corresponding to its nearest centroid. Once the first cluster have been defined, a new centroid is chosen for each cluster by computing some kind of mean between the elements of the cluster. The processes of assigning elements to their nearest neighbor centroid and computing new centroid are iteratively repeated until the elements of each cluster do not change in two consecutive iterations.

To construct the tree, K-Means is first applied with $K = 2$ to the original set of fingerprints, to obtain two clusters. The binary *OR* of the elements of each cluster is computed to obtain the two children of the root node of the tree. This process is recursively repeated inside each cluster until one of the two following conditions hold: i) the elements per cluster reach two (two original fingerprints), ii) the K-Means method cannot subdivide the cluster further. At this stage the tree is completed. The number of fingerprints contained in a leaf node may be greater than two if they are either equal, or so similar that they cannot be subdivided further.

To be able to apply K-Means to sets of bitmaps, both a distance measure and a mean calculation method have to be defined. The number of 1s in common between elements of the same cluster must be maximized, in order to to minimize the 1s in the binary *OR*. If b_1 and b_2 are two bitmaps, then the distance between b_1 and b_2, denoted $d(b_1, b_2)$, is defined in the current proposal as follows.

$$d(b_1, b_2) = \begin{cases} 0, & \text{if } b_1 \subseteq b_2 \lor b_2 \subseteq b_1. \\ count1s(b_1 \oplus b_2), & \text{otherwise.} \end{cases} \tag{1}$$

In the above definition, $b_i \subseteq b_j$ is used to denote that b_i is binary contained in b_j, i.e., $b_i \land b_j = b_i$. The *XOR* binary operation between two bitmaps b_i and b_j is denoted by $b_i \oplus b_j$, and $count1s(b)$ denotes the number of 1s of a bitmap b. It is noticed that the above definition of distance between bitmaps is based on the well-known Hamming distance [9], that counts the number of 1s in the *XOR* of the bitmaps.

The mean calculation method enables obtaining a representative centroid bitmap from all the bitmaps of a cluster. If $zeros(i)$ is the number of 0s in position i of all the bitmaps of the cluster and $ones(i)$ is the number of 1s in position i of all the bitmaps of the cluster, then value of the bit at position i of the mean of the bitmaps of the cluster is 0 if $zeros(i) > ones(i)$ and 1 otherwise.

The calculation of distance and mean between bitmaps if illustrated in Figure 8 through various examples. Note that KM-CTI is an in-memory index, therefore a binary tree is expected to offer better results than trees of higher order. The value of K, however, can be increased to obtain an structure suitable for secondary storage, as it is the B-tree.

As it is the case of the tree based indexes of conventional alphanumerical data, KM-CTI is expected to improve the performance of the sequential search adopted by CT-Index, if the query is selective enough. Therefore, it is expected that KM-CTI will behave better with large query graphs than with small ones.

6 EVALUATION

In this section, the experiments performed and their results are described and discussed. The system setup, datasets and queries are described in a first subsection. Next, the benchmark undertaken to select the appropriate values for maximum path length

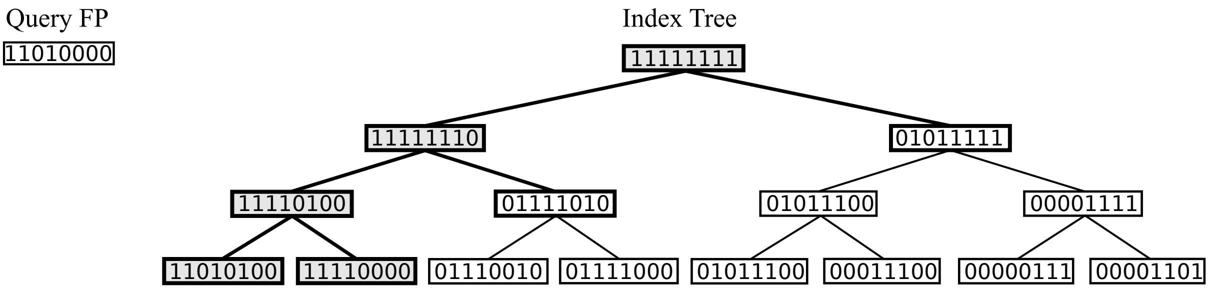


Figure 7: Example of filtering stage in KM-CTI. Highlighted nodes represent the tested ones, and the nodes filled with color are the ones that passed the test.

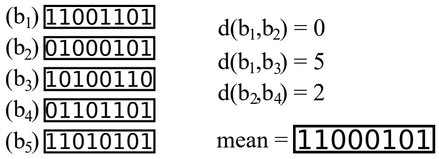


Figure 8: Example of distance and mean measures between bitmaps used in KM-CTI.

and fingerprint size is described. Finally, the last subsection is devoted to the comparison of the performance of the various techniques, with different query and database sizes.

6.1 System Setup, Datasets and Queries

The experiments were performed on a CentOS Linux release 7.4.1708, with 2 processors Intel(R) Xeon(R) CPU E5-2630 v4 (2.20 GHz, 10 cores) and 384 GB RAM. The GGSX implementation in C++ was obtained from the authors, and reimplemented in Java. For the CT-Index method, a jar was obtained from one of the authors website, and then a reverse engineering technique was applied in order to obtain the code. All the new methods were implemented in Java, reusing the above two base implementations. The experiment executions were executed using a JVM with 32GB. The verification stage in BM-GGSX, CW-CTI, and KM-CTI methods was performed using 10 threads.

Two different datasets were used to construct databases of different sizes for the experiments. The AIDS[2] dataset has already been used to evaluate other previous works of the state of the art [2, 8, 13, 15, 16]. It is composed of 42689 graphs, with an average of 45.70 vertices and 47.71 edges per graph. PubChem[3] is a molecular dataset that has currently around 96 million compounds, with an average of 41.40 vertices and 42.16 edges per graph.

In both experiments queries of sizes ranging from 4 to 40 edges (4, 8, 12, 16, 20, 24, 28, 32, 36, 40) were used. For each size, a set of 1000 query graphs was constructed. The construction of the queries was performed in the same way as in previous works [13, 15], by extracting subgraphs from the dataset. For each query, a graph was randomly selected according to a uniform distribution. A vertex of the graph was randomly selected according to a uniform distribution to act as starting point from which to make the query graph grow. From this starting vertex, a random tree was created until the desired size was reached. The size of the query was given by the number of edges of its graph. Two different sets of queries were constructed. One for the AIDS dataset, and the other for the PubChem dataset.

6.2 Parameter Selection

The objective of this first experiment was twofold. The first aim is to test the behavior of the different techniques, including the two methods obtained from the state of the art (GGSX and CT-Index), and the three new proposals, with two different datasets and query sizes. At the same time, the experiment aims at testing the performance of each technique with different values of its parameters, including maximum path length for all the methods and fingerprint size for those based in CT-Index.

The experiment was executed on two databases, AIDS on the one hand and 200k molecular structures obtained from PubChem (PubChem200k) on the other hand. The 1000 queries extracted from each dataset were evaluated 5 times with each method, and the experiment was repeated 4 times at different moments. As it was expected, the size and building time of the BM-GGSX trie structure increases with the maximum path length used. Regarding the CT-Index based methods, the size of the indexes is only affected by the fingerprint size, whereas both fingerprint size and maximum path length have a negative impact on the index building time.

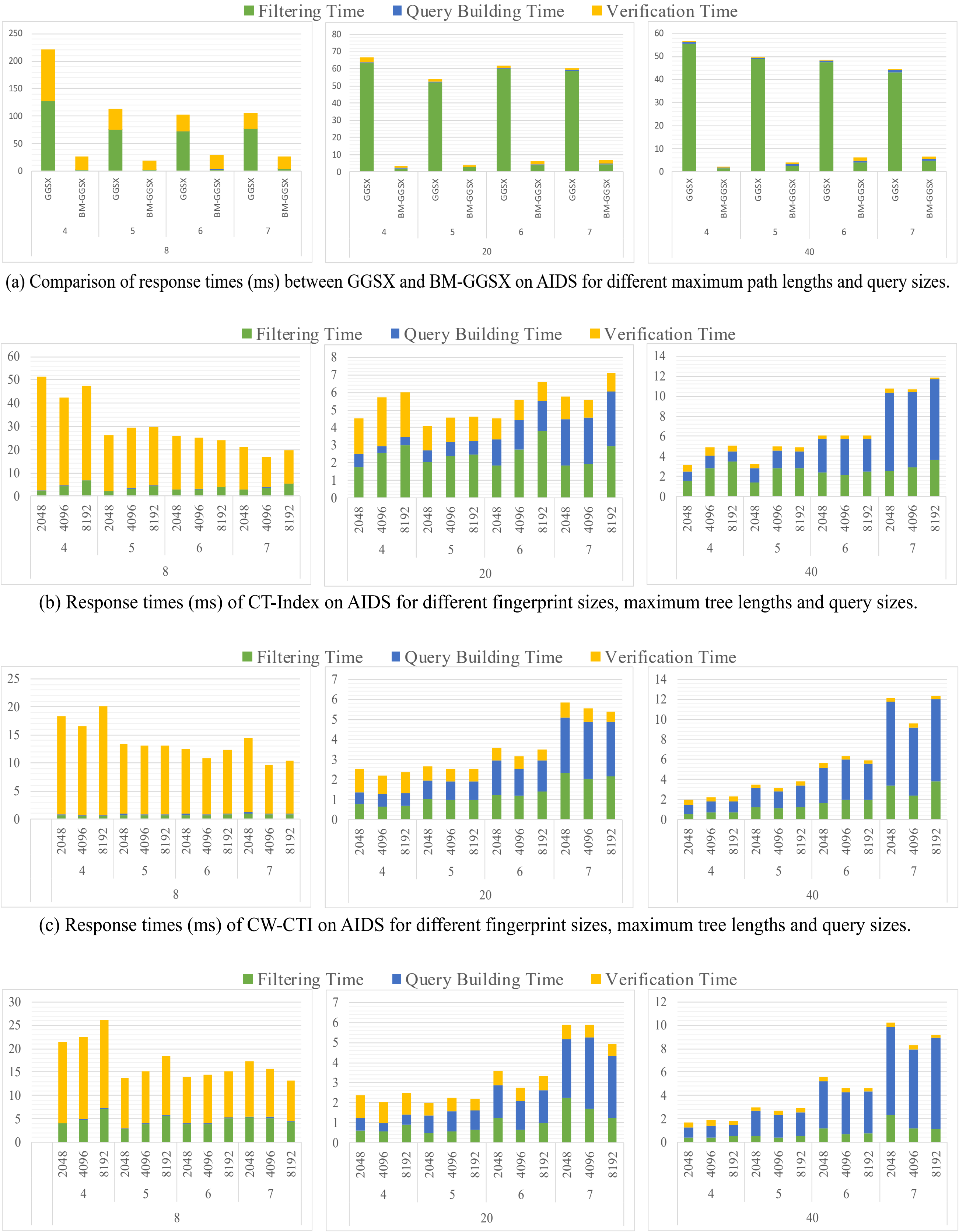
Regarding query response time, Figure 9(a) shows the comparison (subdivided into filtering, query building and verification times) between the original GGSX and BM-GGSX on the AIDS database, using different maximum path lengths and query sizes. As it is shown in the figure, the reduction in filtering time achieved by BM-GGSX is huge (around 90% of reduction in many cases). Due to this large difference, the original GGSX method was not considered for subsequent experiments. As it can be observed in the figure, using different values for the maximum path length (4, 5, 6 and 7) does not show to have a significant impact in the query response time.

Figures 9(b-d) show the query response times obtained with the three CT-Index based methods on the AIDS database, using different maximum path lengths, fingerprint sizes and query sizes. In general, for small queries, where the query building time is not important, a maximum path length higher than 4 offers better results, whereas the fingerprint size does not show a significant impact in the response time. For larger queries, the query building time becomes very significant, due to the few candidates retrieved from such a small database, and therefore, increasing the maximum path length impacts negatively in the response time.

The results of the experiment on the PubChem200k database are illustrated in Figures 10(a-d). In particular, Figure 10(a) shows

[2] https://cactus.nci.nih.gov/download/nci/

[3] https://pubchem.ncbi.nlm.nih.gov/



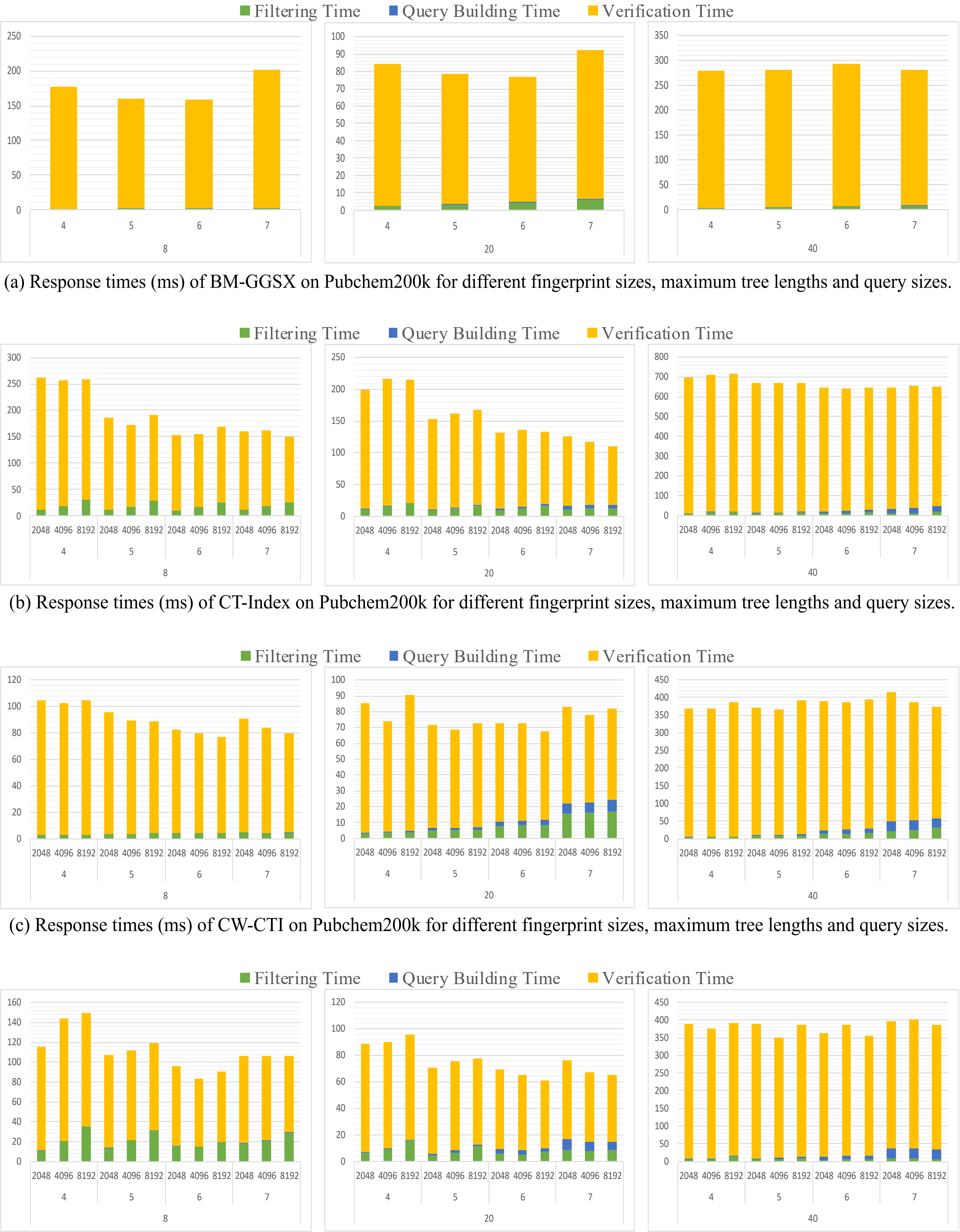
(a) Comparison of response times (ms) between GGSX and BM-GGSX on AIDS for different maximum path lengths and query sizes.

(b) Response times (ms) of CT-Index on AIDS for different fingerprint sizes, maximum tree lengths and query sizes.

(c) Response times (ms) of CW-CTI on AIDS for different fingerprint sizes, maximum tree lengths and query sizes.

(d) Response times (ms) of KM-CTI on AIDS for different fingerprint sizes, maximum tree lengths and query sizes.

Figure 9: Response times of the methods on the AIDS database using different parameters.



(a) Response times (ms) of BM-GGSX on Pubchem200k for different fingerprint sizes, maximum tree lengths and query sizes.

(b) Response times (ms) of CT-Index on Pubchem200k for different fingerprint sizes, maximum tree lengths and query sizes.

(c) Response times (ms) of CW-CTI on Pubchem200k for different fingerprint sizes, maximum tree lengths and query sizes.

(d) Response times (ms) of KM-CTI on Pubchem200k for different fingerprint sizes, maximum tree lengths and query sizes.

Figure 10: Response times of the methods on the PubChem200k database using different parameters.

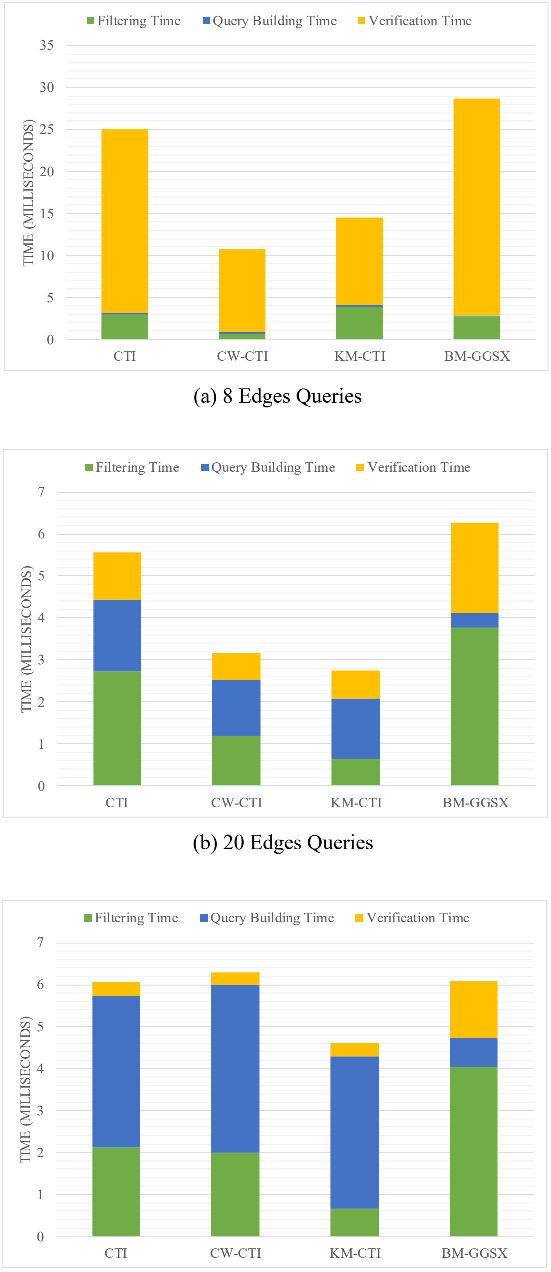
the response times obtained by BM-GGSX, with different maximum path lengths and with different query sizes. As it may be observed, the way the queries are constructed and the the much larger size of the database, compared to AIDS, causes the verification time to be the most significant one in this experiment. Again, the maximum path length does not show a clear impact in the total response time for BM-GGSX. Regarding the methods based on CT-Index (see Figures 10(b-d)), again, significant differences cannot be observed in general between different values of fingerprint size and maximum path length, although maximum path lengths higher than 4 provide slightly better results in most cases.

Based on the above experimental results, and also taking into account relevant decisions made in a previous survey [13], in the subsequent experiment we decided to use the same values of 6 for the maximum path length and of 4096 for the fingerprint size, for all the methods. The objective of such new experiment is the comparison of the performance of all the methods (except the discarded original GGSX), for increasing database and query sizes and the results and relevant discussion are given in the next subsection.

6.3 Performance comparison

In this section, the results of the performance comparison of all the methods (except the discarded GGSX) are discussed. As it was already stated, a maximum path length of 6 and a fingerprint size of 4096 were chosen. First, the results of the experiments described in the previous subsection are used to compare the performance of all the methods on the AIDS database. Figures 11(a-c) show the filtering, query building and verification times of queries of different sizes on the AIDS database. As it may be observed in Figure 11(a), CW-CTI has the best performance for small queries, both in filtering time and also in total response time. This is due to the few intersections between compressed bitmaps that the technique has to perform when few 1s are present in the query fingerprint. However, as the query size increases, the performance of CW-CTI deteriorates, reaching the same performance of CTI for large queries, whereas the performance of KM-CTI improves. The reason is that, with large queries, with many 1s in their fingerprints, many branches of the KM-CTI binary tree of bitmaps will be discarded during the filtering stage.

A second experiment was undertaken to compare all the methods, except the discarded GGSX, with increasing database and query sizes. To achieve this, 10 databases with sizes ranging from 100k to one million graphs were extracted from the PubChem dataset. All the index structures were created for each database, and relevant index size and index building time results were obtained. Figure 12 shows the index size and index building time for each of the analyzed methods. Index building time scales well for CT-Index and CW-CTI methods, growing linearly, and ranging from less than 6 minutes to index 100k graphs, to around 1 hour to index the largest dataset of one million graphs. KM-CTI index building time grows linearly as well, but with a higher slope than CT-Index and CW-CTI methods, being more than 3 times slower for the largest dataset. This is due to the recursive application of the K-Means method to construct the binary tree of bitmaps. BM-GGSX has the worst performance of all methods, taking more time to index 200k graphs, than CT-Index and CW-CTI methods to index one million. Its index building time increases drastically with the size of the dataset, and it needed more than 14 hours to index the largest database of one million graphs.



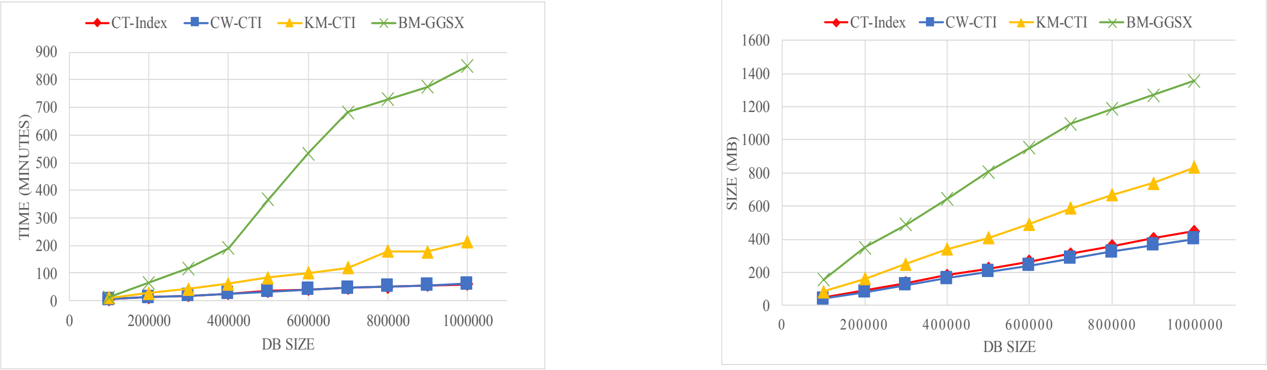
(a) 8 Edges Queries

(b) 20 Edges Queries

(c) 40 Edges Queries

Figure 11: Performance comparison of the methods on the AIDS database.

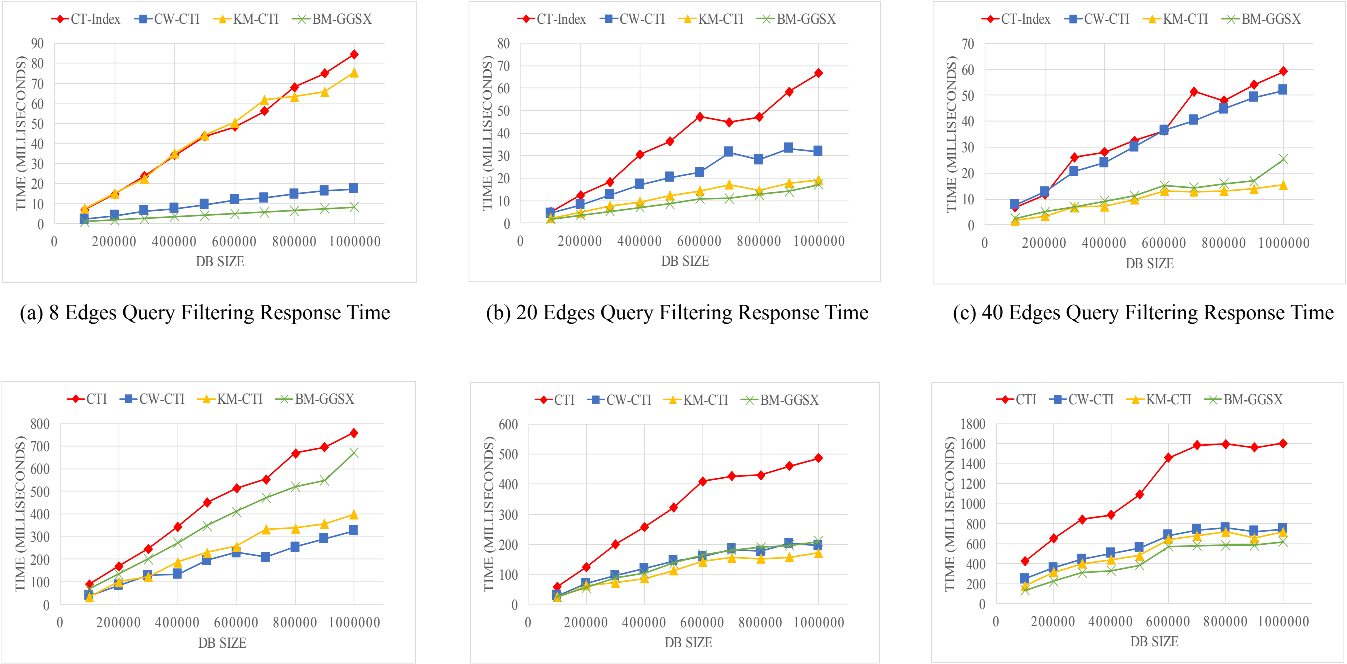
Regarding the index size, BM-GGSX performs again the worst among all the proposed methods, showing a big slope in its scalability plot, going from 155MB in the shortest dataset to almost 1.4GB in the largest one. CW-CTI performs the best, scaling slightly better than CT-Index method, due to the use of compressed bitmaps in its storage structure. KM-CTI doubles the space of CW-CTI and CT-Index, due to the additional bitmaps stored in the internal nodes of its binary tree. However, its size is still almost half of that required by BM-GGSX.



(a) Index Building Time

(b) Index Size

Figure 12: Index Size and Building Time for increasing database sizes.



(a) 8 Edges Query Filtering Response Time

(b) 20 Edges Query Filtering Response Time

(c) 40 Edges Query Filtering Response Time

(d) 8 Edges Total Query Response Time

(e) 20 Edges Total Query Response Time

(f) 40 Edges Total Query Response Time

Figure 13: Filtering time and total query response time for increasing database and query sizes.

Once the indexes were created, the 1000 queries of each size (from 4 edges to 40 edges), generated for the previous experiment from the 200k PubChem database, were executed four times for each database size, and the whole experiment was repeated three times in different moments. Figures 13(a-c) depict the filtering time of each method. It is reminded that the main contribution of the present work is on the index structures used in the filtering stage of the methods. BM-GGSX and CW-CTI obtain the fastest filtering time for small queries (around 80% of reduction with respect to CTI), as it is shown in Figure 13(a), whereas KM-CTI and CTI offer a similar performance. It is reminded that CW-CTI was expected to behave well with small queries that have few 1s in their fingerprints. On the other hand, small not selective queries, with few 1s in their fingerprints, do not leverage the binary tree of KM-CTI. As the query size increases, CTI and BM-GGSX keep their performance figures, however, CW-CTI and KM-CTI invert their positions. With queries of large sizes, with many 1s in their fingerprints, CW-CTI has to perform many intersections between large compressed bitmaps. On the other hand, KM-CTI behaves better, as many branches of its binary tree are discarded during the evaluation. This situation is clearly shown in Figures 13(b) (around 70% of reduction with respect to CTI) and 13(c) (around 75% of reduction with respect to CTI). These results are completely aligned with those already described for the AIDS database above.

To complete the analysis of the results, the total query response times are shown in Figures 13(d-f). First, looking at the scale of the total response times, it is noticed that, in these databases, contrary to what it was observed with AIDS, the filtering time does not have a great impact on the total response time, since the verification time is the largest one by difference. In spite of this, it is shown how the three methods proposed in the present work outperform the CT-Index method. This is mainly due to the parallel execution in a multi-thread architecture of the VF2 subgraph isomorphism algorithm. It is also noticed that, BM-GGSX, which had the best results in filtering time for all the query sizes, does not show a good total response time for small queries. The reason for this lost of performance is the use of paths as graph features in BM-GGSX, contrary to the use of trees in CT-Index based techniques, which enable the generation of a larger number of different features in small graphs, reducing the number of candidates to verify. The small differences between CW-CTI and KM-CTI are caused by the already mentioned differences in filtering time, since both the effectiveness of their filtering stage and their verification stage are identical.

7 CONCLUSIONS

In this paper, three new FTV methods for subgraph search on large databases of small graphs are proposed. The methods are based on already existing state of the art solutions, namely GGSX and CT-Index, and provide improvements both in filtering and verification response times. BM-GGSX improves GGSX through the incorporation of compressed bitmaps for the representation of graph references, and reaches the best filtering times for all types of queries and the best total response times for medium and large queries. However, it does not improve the bad figures that GGSX already have in both index size and index building time. CW-CTI exploits also compressed bitmap inside a column-wise structure for the storage of the CT-Index fingerprints, obtaining very good performance for small size queries. KM-CTI uses the K-Means clustering method on CT-Index fingerprints to construct a binary tree of bitmaps, which is used during the filtering stage to reduce the required fingerprint comparisons. KM-CTI has very good performance for medium and large queries, with the cost of increasing the index size and building time with respect to the original CT-Index, but keeping the figures much lower than those reached by BM-GGSX. Thus, a combination of CW-CTI and KM-CTI covers all the query sizes with performances either similar of better than those of BM-GGSX, but keeping both index size and building times inside a reasonable range.

Future research work is related to the implementation of the proposed techniques in distributed large scale data processing architectures, to reach reasonable performances for databases of sizes larger than 100 million graphs, as it will be the case of the PubChem dataset in the near future. The results will be incorporated in a query engine prototype that is being implemented in the scope of the NEXTCHROM project, co-funded by the Spanish government and the company Mestrelab Research S.L.

ACKNOWLEDGMENTS

This work has been co-funded by the Ministerio de Economía y Competitividad of the Spanish government, and by Mestrelab Research S.L. through the project NEXTCHROM (RTC-2015-3812-2) of the call Retos-Colaboración of the program Programa Estatal de Investigación, Desarrollo e Innovación Orientada a los Retos de la Sociedad.

REFERENCES

[1] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. 2016. Efficient subgraph matching by postponing Cartesian products. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. ACM, 1199–1214.

[2] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha. 2010. Enhancing graph database indexing by suffix tree structure. In *IAPR International Conference on Pattern Recognition in Bioinformatics*. Springer, 195–203.

[3] J. Cheng, Y. Ke, W. Ng, and A. Lu. 2007. FG-Index: Towards verification-free query processing on graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 857–872.

[4] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 10 (2004), 1367–1372.

[5] H.-C. Ehrlich and M. Rarey. 2012. Systematic benchmark of substructure search in molecular graphs - From Ullmann to VF2. *Journal of Cheminformatics* 4, 1 (31 Jul 2012), 13. DOI: http://dx.doi.org/10.1186/1758-2946-4-13

[6] H.-C. Ehrlich, A. Volkamer, and M. Rarey. 2012. Searching for substructures in fragment spaces. *Journal of Chemical Information and Modeling* 52, 12 (2012), 3181–3189. DOI: http://dx.doi.org/10.1021/ci300283a PMID: 23205736.

[7] E. Forgy. 1965. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics* 21 (1965), 768–780.

[8] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha. 2013. GRAPES: A software for parallel searching on biological graphs targeting multi-more architectures. *PLoS ONE* 8, 10 (2013).

[9] R. W. Hamming. 1950. Error detecting and error correcting codes. *Bell System Technical Journal* 29, 2 (1950), 147–160.

[10] W.-S. Han, J. Lee, and J.-H. Lee. 2013. TurboISO: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 337–348.

[11] W.-S. Han, J. Lee, M.-D. Pham, and J.X. Yu. 2010. iGraph: A framework for comparisons of disk based graph indexing techniques. *Proceedings of the VLDB Endowment* 3, 1 (2010), 449–459.

[12] H. He and A. K. Singh. 2008. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 405–417.

[13] F. Katsarou, N. Ntarmos, and P. Triantafillou. 2015. Performance and scalability of indexed subgraph query processing methods. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1566–1577.

[14] F. Katsarou, N. Ntarmos, and P. Triantafillou. 2017. Subgraph querying with parallel use of query rewritings and alternative algorithms. In *Advances in Database Technology - EDBT*, Vol. 2017-March. 25–36.

[15] K. Klein, N. Kriege, and P. Mutzel. 2011. CT-Index: Fingerprint-based graph indexing combining cycles and trees. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 1115–1126.

[16] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. 2012. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *Proceedings of the VLDB Endowment*, Vol. 6. 133–144.

[17] D. Lemire, O. Kaser, and K. Aouiche. 2010. Sorting improves word-aligned bitmap indexes. *Data and Knowledge Engineering* 69, 1 (2010), 3–28.

[18] J. MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. University of California Press, Berkeley, 281–297.

[19] X. Ren and J. Wang. 2015. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment* 8, 5 (2015), 617–628.

[20] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. 2008. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment* 1, 1 (2008), 364–375.

[21] J. R. Ullmann. 1976. An algorithm for subgraph isomorphism. *J. ACM* 23, 1 (1976), 31–42.

[22] J. Wang, N. Ntarmos, and P. Triantafillou. 2016. Indexing query graphs to speedup graph query processing. In *Advances in Database Technology - EDBT*, Vol. 2016-March. 41–52.

[23] J. Wang, N. Ntarmos, and P. Triantafillou. 2017. GraphCache: A caching system for graph queries. In *Advances in Database Technology - EDBT*, Vol. 2017-March. 13–24.

[24] X. Yan, P.S. Yu, and J. Han. 2004. Graph indexing: A frequent structure-based approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 335–346.

[25] S. Zhang, M. Hu, and J. Yang. 2007. TreePi: A novel graph indexing method. In *Proceedings - International Conference on Data Engineering*. 966–975.

[26] S. Zhang, S. Li, and J. Yang. 2009. GADDI: Distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT'09*. 192–203.

[27] P. Zhao and J. Han. 2010. On graph query optimization in large networks. *Proceedings of the VLDB Endowment* 3, 1 (2010), 340–351.

[28] P. Zhao, J. X. Yu, and P. S. Yu. 2007. Graph indexing: Tree + Delta >= Graph. In *33rd International Conference on Very Large Data Bases, VLDB 2007 - Conference Proceedings*. 938–949.

[29] L. Zou, L. Chen, J. X. Yu, and Y. Lu. 2008. A novel spectral coding in a large graph database. In *Advances in Database Technology - EDBT 2008 - 11th International Conference on Extending Database Technology, Proceedings*. 181–192.

Spec-QP: Speculative Query Planning for Joins over Knowledge Graphs

Madhulika Mohanty
Indian Institute of Technology, Delhi
Hauz Khas, New Delhi, India
madhulikam@cse.iitd.ac.in

Maya Ramanath
Indian Institute of Technology, Delhi
Hauz Khas, New Delhi, India
ramanath@cse.iitd.ac.in

Mohamed Yahya*
Bloomberg
London, United Kingdom
yahya.mohamed@gmail.com

Gerhard Weikum
Max Planck Institute for Informatics
Saarland Informatics Campus, Germany
weikum@mpi-inf.mpg.de

ABSTRACT

Knowledge Graphs (KGs) have become ubiquitous in organisations. They provide a unified and structured model to store the data as well as facilitate effective search to fulfill many complex information needs. One of the ways to query these KGs is to use SPARQL queries over a database engine. Since SPARQL follows exact match semantics, the queries may return too few or no results. Recent works have proposed *query relaxation* where the query engine judiciously replaces a query predicate with similar predicates using weighted relaxation rules mined from the KG. However, the space of possible relaxations is potentially too large to fully explore and users are typically interested in only top-k results, so such query engines use top-k algorithms for query processing. Nevertheless, they still process all the relaxations, many of whose answers do not contribute towards top-k answers. We propose Spec-QP, a query planning framework that speculatively determines which relaxations will have their results in the top-k answers. This reduces the computational overheads and gives faster response times with good precision over top-k results. We tested Spec-QP over two database engines, PostgreSQL and Virtuoso, with two datasets – XKG and Twitter – to demonstrate the efficiency of our planning framework at supporting relaxations in query engines.

1 INTRODUCTION

Knowledge Graphs (KGs) such as YAGO [34], DBPedia [2] and Freebase [5] are typically stored as RDF triples of ⟨s p o⟩ where s is the subject, o is the object and p is the predicate. These RDF KGs are queried using the SPARQL query language, that, at its core consists of *triple patterns*. For example, the following SPARQL query asks: "Which singers also write lyrics and play guitar and piano?".

```
SELECT ?s WHERE{
   ?s 'rdf:type' <singer>.
   ?s 'rdf:type' <lyricist>.
   ?s 'rdf:type' <guitarist>.
   ?s 'rdf:type' <pianist>
}
```

where ?s is a variable to be bound in each of the four triple patterns and to be returned as a result.

Original	Relaxations
<singer>	<vocalist>,<jazz_singer>, <artist>
<lyricist>	<writer>
<guitarist>	<musician>, <instrumentalist>
<pianist>	<percussionist>

Table 1: Example relaxations

An exhaustive list of such singers in the KG can be computed, but users who issue such queries typically want only the top-k, *ranked* results. Ranking of SPARQL query results has been studied before in [9, 12, 23] and they typically make use of *scores* for each triple in the KG[1]. However, a problem that users sometimes face when they issue such queries is *low recall*. That is, the KG may not have k results to return (in some cases, the KG may have *zero* results if one or more of the triple patterns do not have a match). In these cases, it is desirable to *relax* the query by replacing one or more of the triple patterns, while ensuring that the query still reflects the original information need. For example, a possible relaxation of the query above is to change the triple pattern ⟨?s 'rdf:type' <singer>⟩ to ⟨?s 'rdf:type' <vocalist>⟩. Previous works have dealt with doing these relaxations *automatically* and ranking the corresponding results [10, 18, 31, 42]. In this paper, we address the problem of *efficiently evaluating* these relaxed queries.

Query Processing. Processing queries and their relaxations to return top-k results is computationally expensive. For example, assuming that every triple pattern in the above query has relaxations as shown in Table 1, would lead to a total of 48 unique queries (that is, original query, query with one relaxation, query with two relaxations, etc.). A naive method would compute the results to each query, sort the results by score and return the top-k.

Since the user is looking for only top-k answers, the naive method can be improved upon by using top-k operators. They can compute results from *all* relaxations simultaneously, but in a way that drastically reduces wasteful computations. The following two top-k operators can be employed for achieving this: *Incremental Merge* [35] (to process the relaxations for a given triple pattern) and *Rank Join* [19] (to compute (partial) join results in sorted order). However, this method still results in wasted resources, since not all relaxations contribute a result to the top-k.

[*] Work done while at the Max Planck Institute for Informatics.

[1] The scores could be based on confidence values, popularity, etc.

Approach and contributions. In this paper, we propose a *speculative* approach for pruning the space of possible relaxations for a given query. We make use of precomputed statistics about the distribution of scores of the matches to triple patterns in order to speculate on the requirement of relaxations for each triple pattern in order to get top-k results. This precomputed metadata is an approximation of the score distribution of the answers from the corresponding triple pattern and not the actual scores. This is like computing histograms, or simpler. Each of these statistics can be computed in one pass as part of the statistics collection phase of any database system that does cost-based optimization of queries.

When a user enters a query, we estimate the top answer scores that can be achieved using the possible relaxations. This estimation is done using the score distributions and the join cardinality estimates. We then prune those relaxations which are unlikely to contribute triples to the top-k answers based on the top score estimates. This results in reduced computation and faster response times. Also, the amount of search space traversed is reduced by pruning unnecessary relaxations and this, in turn, leads to reduction in memory requirements. The runtime reduction combined with reduced memory requirements leads to an overall improvement on memory consumed over time. This is especially beneficial for servers where the total resource consumption per query is important, as it is inversely proportional to the achievable throughput. Or equivalently, the cost of running the server for a given load is proportional to the cost per query and this improvement implies that the server can run the service with lesser budget in terms of money. *Note that our work is orthogonal to any query engine as it can be deployed on top of any existing RDF-specific database engine.*

Our main contributions are summarized as follows.

i. A model for the score distribution of individual triple patterns.
ii. A technique to estimate the scores of answers to a query using the above model and using it to predict the presence of answers from each triple pattern's relaxations in the top-k.
iii. Pruning the space of relaxations to achieve significantly faster response times while maintaining high accuracy, thereby aiding *cost-effective* exploration of KGs.
iv. Thorough experimental evaluation of the proposed technique over two database engines – PostgreSQL and Virtuoso – with two real world datasets to demonstrate its efficiency over the baseline.

Organisation. The rest of the paper is organised as follows: section 2 introduces useful definitions and explains the top-k operators based query processing approach. Section 3 outlines Spec-QP, the proposed speculative approach to query planning and its execution. Section 4 discusses the experimental results. Section 5 lists the related work and finally section 6 concludes the paper with future work directions.

2 PRELIMINARIES

This section introduces some preliminary definitions that will be used henceforth.

Definition 2.1. **Knowledge Graphs (KGs)**
Given a set of entities $\mathbf{E}$ and predicates $\mathbf{P}$, a triple t is a tuple $t = \langle \mathtt{s\ p\ o} \rangle$ such that, $t \in \mathbf{E} \times \mathbf{P} \times \mathbf{E}$, $\mathtt{s} \in \mathbf{E}$, $\mathtt{p} \in \mathbf{P}$ and $\mathtt{o} \in \mathbf{E}$. Here, s is called the "subject", p is the "predicate" and o is the "object" of the triple t. Each triple is associated with a score, denoted by $S(t)$. These scores represent confidence values or popularity of the triples as previously studied in [9, 12, 23]. A set of such tuples can be represented as a graph, which we call a Knowledge Graph, $\mathbf{KG} \subseteq \mathbf{E} \times \mathbf{P} \times \mathbf{E}$.

Definition 2.2. **Triple pattern query**
A triple pattern is of the form $q = \langle \mathtt{SPO} \rangle$, where S, P and O could either be entities or predicates from the KG or variables. Variables are always prefixed with a question mark. A triple pattern matches any triple in the KG having the same values in the designated field. The variables are then bound to the corresponding values in the triple. A triple pattern query is a set of triple patterns, $\mathbf{Q} = \{q_1, q_2, ... q_n\}$.

Definition 2.3. **Answer for a Triple pattern query**
Given a triple pattern query $\mathbf{Q}$ and a KG, an answer for the query, denoted by A, is a mapping of the variables in $\mathbf{Q}$ to values in the KG such that the application of this mapping to each triple pattern $q_i \in \mathbf{Q}$, denoted $A(q_i)$, results in a triple in the KG. The set of all the answers to a query is denoted by the set, $\mathbf{A}$. That is,

$$\mathbf{A}(q_i) = \{A(q_i) : A \in \mathbf{A}\} \tag{1}$$

Definition 2.4. **Score of an answer**
The relative score of a triple t which matches the triple pattern q is denoted by $S(t|q)$ and is computed as follows:

$$S(t|q) = \frac{S(t)}{\max\limits_{t_i \in \mathbf{A}(q)} (S(t_i))} \tag{2}$$

The value ranges between 0 and 1. The score of an answer A to a query $\mathbf{Q}$ is the aggregation of the relative scores of the triples resulting from applying the answer mapping to each triple pattern q_i in the query. That is,

$$S(A|\mathbf{Q}) = \sum_{q_i \in \mathbf{Q}} S(A(q_i)|q_i) \tag{3}$$

The triple and answer scores have been studied previously in [10, 18, 31, 42].

Definition 2.5. **Weighted relaxation rule**
A weighted relaxation rule r is a triple $r = (q, q', w)$ which implies that triple pattern q can be relaxed to q', and $w \in [0, 1]$ denotes the reduction in scores of the triples matching the relaxed triple pattern. Automatic computation of relaxations and the corresponding weights have been studied in [10, 42].

For example, ⟨?x 'rdf:type' <singer>⟩ could be relaxed to ⟨?x 'rdf:type' <vocalist>⟩ with a weight of 0.8, i.e., r = (⟨?x 'rdf:type' <singer>⟩, ⟨?x 'rdf:type' <vocalist>⟩, 0.8).

Definition 2.6. **Relaxed Query**
Given a query $\mathbf{Q}$ and a relaxation $r = (q, q', w)$, we say that r applies to $\mathbf{Q}$ if $q \in \mathbf{Q}$. The result of applying r to $\mathbf{Q}$ is a new query $\mathbf{Q'} = (\mathbf{Q} \setminus q) \cup q'$ called the relaxed query. The relaxed query so obtained can further be relaxed by relaxing any of $\mathbf{Q'} \setminus q'$ triple patterns. The score of an answer A obtained through relaxation r applied to a query $\mathbf{Q}$ is equal to $w \times S(A|\mathbf{Q'})$.The score is reduced further for each subsequent relaxation in a similar manner. Since the same answer could be obtained from multiple relaxed queries, the score of an answer A with respect to the original query and a space of possible relaxations is defined as the maximum score obtained through any (relaxed) query,

$$S(A) = \max_{\mathbf{Q'}}(w \times S(A|\mathbf{Q'}))$$

, where $w = 1$ for the original query, $\mathbf{Q}$.

2.1 Non-Speculative Query Processing (NSpec-QP)

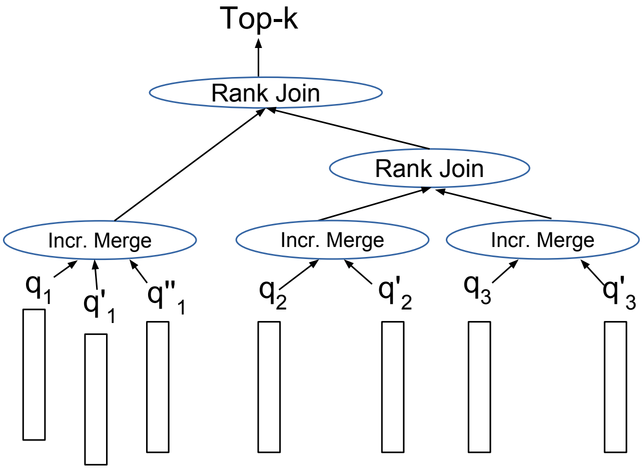


Figure 1: Query plan generated by NSpec-QP for the query Q = $\{q_1,q_2,q_3\}$. One incremental merge operator is required for each triple pattern and its relaxations. A rank join operator takes in two sorted lists and produces a ranked list of (partial) answers from the join.

As mentioned in the Introduction, we can compute results from *all* relaxations simultaneously using two top-k operators: *Incremental Merge* [35] and *Rank Join* [19]. The execution strategy is essentially a variant of the Fagin's NRA algorithm [11]. It computes the exact top-k as it computes all applicable relaxations. It has been used by systems supporting relaxations such as TriniT [42].

Given the query Q = $\{q_1,q_2,q_3\}$, and the relaxations, $r_1 = (q_1,q'_1,w_1), r_2 = (q_1,q''_1,w_2), r_3 = (q_2,q'_2,w_3)$ and $r_4 = (q_3,q'_3,w_4)$, Figure 1 shows the query plan generated using this approach. Incremental Merge is used to efficiently scan the list of matches to a triple pattern and all its relaxations to output only one merged list sorted in descending order of scores. Each of the three incremental merge operators in the example takes as inputs the sorted lists of matches[2] for each triple pattern, q_1, q_2 and q_3 and their relaxations, each multiplied by their relaxation weights. Each of them outputs a combined sorted list of triples for each triple pattern along with its relaxations. The rank join computes a join of the two sorted inputs in an incremental manner until enough results have been produced, while minimising the number of answers read from each list to get top-k answers. This helps avoid computing the entire join and then sorting over it. The inputs for Rank Joins are either the outputs of Incremental Merges or Rank Joins. Both operators use priority queues for already seen answers and maintain upper bounds to estimate scores of other answers that can be obtained by reading further into the lists at any given point. This avoids accessing entire lists of (partial) answers and aids early termination.

However, the top-k operators still process relaxations from *all* the triple patterns, many of which do not contribute triples towards the top-k answers. Our technique aims to eliminate this inefficiency.

3 SPEC-QP – THE SPECULATIVE FRAMEWORK FOR OPTIMIZING QUERY PLANS

We propose Spec-QP, a speculative query planning framework which speculates the useful relaxations for a query. It uses a predictor to predict whether the relaxations of a triple pattern in a query are likely to be required for producing the top-k answers. We eliminate the relaxations for those triple patterns which are predicted to be not required. The predictor uses an expected score estimator to estimate the expected scores at given ranks for a (relaxed) query and then, predicts the requirement of relaxations of a triple pattern for getting top-k answers based on the estimates. The estimator is based on precomputed statistics about the distribution of the scores for triple pattern matches. We first describe the estimator and then give details of the speculative planning approach.

3.1 Expected score estimator

The expected score estimator is based on order statistics and estimates the expected scores at given ranks for the original as well as relaxed queries. These are used by the query planner to predict the presence of answers from a relaxation in top-k.

The m matching triples for a triple pattern, q_i, have scores represented by the independent and identically distributed (i.i.d.) random variables $X_{i1},X_{i2},...,X_{im}$, each with a common distribution, $f_i(x)$. Here, $f_i(x)$ is the probability distribution for the scores of the answers for a triple pattern (or relaxation), q_i, from the *KG*. The cumulative distribution function (cdf) is represented by $F_i(x)$. The set $\{X_{i1},X_{i2},...,X_{im}\}$ is a sample of size m taken from the distribution $F_i(x)$. The set of the observed values of answer scores $\{x_{i1},x_{i2},...,x_{im}\}$ of random variables $\{X_{i1},X_{i2},...,X_{im}\}$ is called a realization of the sample. $X_{i(1)},X_{i(2)},...,X_{i(m)}$ are random variables resulting from arranging the values of each of $X_{i1},X_{i2},...,X_{im}$ in increasing order, and $X_{i(j)}$ is called the j^{th} order statistic.

Given these random variables and their distributions, we need to estimate the score distribution for the answers of the query, Q. $X_{Q1},X_{Q2},...,X_{Qn}$ are the random variables representing the scores of the n answers to the query, Q (possibly composed of a single triple pattern). $X_{Q(1)}$ is the first order statistic corresponding to the lowest scoring answer among all the n answers of Q, and $X_{Q(n)}$ is the n-th (or largest) order statistic corresponding to the highest scoring answer (ranked 1). A relaxed answer would appear in top-k only when its expected highest score ($X_{Q'(n')}$) amongst its n' answers exceeds the expected k^{th} highest score of the original query ($X_{Q(n-k+1)}$[3]). In order to compute the expected value at a given rank, we use the result given in [7]:

For i.i.d. random variables, $X_1,X_2,...,X_m$ each with a common distribution, $f(x)$, the expected value of i^{th} order statistic, $X_{(i)}$ can be approximated as $E(X_{(i)}) \approx F^{-1}(\frac{i}{m+1})$ where $F(x)$ denotes the cdf and m is the size of the sample.

Using this, the expectation of $X_{Q(i)}$ can be approximated as $E(X_{Q(i)}) \approx F_Q^{-1}(\frac{i}{n+1})$ where $F_Q(x)$ denotes the cdf of the scores for the answers to the query, Q and n is the no. of answers of Q .

We now give the details of the construction of the probability density function (pdf) of these random variables.

[2] Recall that each triple is associated with a score.

[3] Note that it is $n - i + 1$ and not i since the n^{th} order statistic represents the highest value with rank 1.

3.1.1 Score Distributions for the matches to Triple Patterns: For every triple pattern q_i in the *KG*, we store the following 4 precomputed statistics about the scores $\boldsymbol{\sigma^i}$ of the matching triples:

- m_i: the total number of triples matching the triple pattern.
- σ_r^i: the score of the answer at rank r where r represents the rank within which 80% of the score mass is contained for the triple pattern matches.
- S_r^i: the cumulative score of the answers over all the ranks 1 through r.
- $S_{m_i}^i$: the cumulative score of the answers over all the ranks 1 through m_i.

We now estimate the score distribution for answers to triple pattern q_i. [4] $f_i(x)$ and $F_i(x)$ are used to denote the pdf and cdf respectively.

The pdf can be modelled as a 2-bucket histogram in the following way:

$$f_i(x) = \frac{S_{m_i}^i - S_r^i}{S_{m_i}^i} \frac{1}{\sigma_r^i} \text{ for } 0 \le x < \sigma_r^i$$
$$\frac{S_r^i}{S_{m_i}^i} \frac{1}{1 - \sigma_r^i} \text{ for } \sigma_r^i \le x \le 1$$

This pdf gives us the following cdf:

$$F_i(x) = ax \text{ for } 0 \le x < \sigma_r^i$$
$$bx + c \text{ for } \sigma_r^i \le x \le 1$$

where

$$a = \frac{S_{m_i}^i - S_r^i}{S_{m_i}^i} \frac{1}{\sigma_r^i} \text{ and } b = \frac{S_r^i}{S_{m_i}^i} \frac{1}{1 - \sigma_r^i}$$
$$\text{and } c = \frac{S_{m_i}^i - S_r^i}{S_{m_i}^i} - \frac{S_r^i}{S_{m_i}^i} \frac{\sigma_r^i}{1 - \sigma_r^i}$$

Our technique depends on statistical estimates – specifically, what the score of the k^{th} result is for a specific (original) query. This estimate can be made as accurate as possible, provided sufficient space and time are available. At one extreme, we can assume uniform distribution for the score of a triple pattern and at the other, we could consider every single data point (the actual distribution). In particular, if we had every single data point as a single-bucket histogram, that would then give us 100% accuracy on the k^{th} score. But this would be no different from actually computing the result. Our solution of 2-bucket approximation strives for the sweet spot on this spectrum and is based on the fact that even though datasets are different, their score distributions typically follow a power law distribution – a "fat" head, and a "long" tail. The narrow and tall bucket represents the interval which has 80% of the score mass. The longer bucket represents the long tail having only 20% of the score mass.

3.1.2 Score Distribution for the Triple Pattern Query: The score of an answer for the triple pattern query is the sum of the scores of the individual triples in the answer. Since each triple is contributed by one triple pattern in the query and we have estimates for their scores, we can estimate the scores for answers to the query using the following approach.

```
Input: The query $\mathbf{Q} = \{q_1, q_2, ...q_n\}$.
Output: The query plan, QP
QP ← $\{\{\mathbf{Q_1}\}\}$, where $\mathbf{Q_1} = \mathbf{Q}$
$f_{\mathbf{Q}}(x) \leftarrow f_1 * f_2 * .. * f_n(x)$
Get $E_{\mathbf{Q}}(k)$ from "expected score estimator".
for $q_i \in \mathbf{Q}$ do
    $q_i' \leftarrow$ top-weighted relaxation for $q_i$
    $\mathbf{Q'} \leftarrow \mathbf{Q} - \{q_i\} \cup \{q_i'\}$
    $f_{\mathbf{Q'}}(x) \leftarrow f_1 * f_2 * ... * f_i' * .. * f_n(x)$
    Get $E_{\mathbf{Q'}}(1)$ from "expected score estimator".
    if $E_{\mathbf{Q'}}(1) > E_{\mathbf{Q}}(k)$ then
        QP = $\{\{\mathbf{Q_1}\} - \{q_i\}, \{q_i\}\}$
    end
end
return QP
```

Algorithm 1: PLANGEN generates the query plan.

Let us assume our triple pattern query, $\mathbf{Q} = \{q_1, q_2\}$. $\{X_{11}, X_{12}, ..., X_{1m}\}$ represents the m triples matching q_1 and $\{X_{21}, X_{22}, ..., X_{2m'}\}$ represents the m' triples matching q_2. The scores for triples matching these triple patterns have the distributions $f_1(x)$ and $f_2(x)$ respectively, as defined earlier. The scores of **Q**'s n answers are represented by the random variables $X_{Q1}, X_{Q2}, ..., X_{Qn}$. Each of these is a sum of two random variables, one from $\{X_{11}, X_{12}, ..., X_{1m}\}$ and another from $\{X_{21}, X_{22}, ..., X_{2m'}\}$. The pdf for the sum of the random variables is given by the convolution of the two individual pdfs, $f_1 * f_2(x)$. Hence, the pdf for the scores of the answers to the query is given by the convolution of the pdf's of the scores for matches to the constituent triple patterns. The resulting pdf is a multi-piece-wise linear function. Given the number of results in the combined distribution, $n = m_{12}$, we can estimate σ_r^{12}, S_r^{12} and S_n^{12} using the expected score computation from order statistics. This again results in a two-bucket histogram for the distribution of the scores of the answers to the query. For the computation of m_{12}, we use the estimates for join selectivity[5], ϕ_{12} as $m_{12} = m * m' * \phi_{12}$. For three or more triple patterns, we repeat the above process the required number of times to get the final histogram representing the score distribution for answers to the query.

3.1.3 Score prediction. Once we have constructed the pdf and cdf representing the scores for the answers of a given query, we can estimate the expected score, $X_{Q(n-i+1)}$ at a given rank i as $E(X_{Q(n-i+1)}) \approx F_Q^{-1}(\frac{n-i+1}{n+1})$ where $F_Q(x)$ denotes the cdf of the query answer scores and n is the no. of answers for **Q**. Given these estimates for scores at various ranks, we now generate the query plan.

3.2 Query Planning

Query Plan: Given a query **Q**, a query plan consists of subsets of triple patterns $\mathbf{Q_1}, \mathbf{Q_2}, ..., \mathbf{Q_s}$ where

i. each $\mathbf{Q_i}$ consists of one or more triple patterns from **Q**,
ii. the $\mathbf{Q_i}$'s are pairwise disjoint, and
iii. the union of $\mathbf{Q_i}$'s equals **Q**.

For example, a query plan for the query $\mathbf{Q} = \{q_1, q_2, q_3\}$, will be $\{\{q_1, q_3\}, \{q_2\}\}$. The singletons correspond to the triple patterns which require relaxations.

[4] Note that the ranks will not be explicitly reflected here, it is just the distribution of the answer score values from which each score in $\{X_{i1}, X_{i2}, ..., X_{im}\}$ is assumed to be independently sampled.

[5] Traditional database systems use multiple heuristics to estimate join selectivity. For the purpose of this work, we have taken exact join selectivity values.

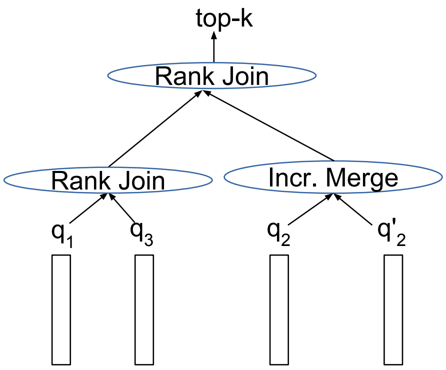


Figure 2: Query Plan when Q = $\{q_1, q_2, q_3\}$ and only q_2's relaxations are predicted to be in top-k. Only q_2 requires an incremental merge. q_1 and q_3 are joined using a rank join over the sorted answer lists for each of them. One rank join is required to join these results.

3.2.1 Query plan generation. The key task in the planning approach is to identify the triple patterns whose relaxations do not contribute towards the top-k answers. We save on computations over such triple patterns by never processing their relaxations. For each triple pattern, only the top-weighted relaxation has the highest top score due to normalization of scores as per Definition 2.4, i.e, the top score from each relaxation is equal to its weight. Hence, we need to check only the top-weighted relaxation for each triple pattern for its potential to contribute answers towards top-k.

Given a query **Q** and the score distribution for each triple pattern, the query plan is generated as outlined in Algorithm 1. PLANGEN first predicts the requirement of relaxations for each triple pattern. For prediction, the query planner uses the "expected score estimator" described in Section 3.1, which gives estimates of the expected scores at k^{th} rank for the original query, $E_{\mathbf{Q}}(k)$ and top rank for the highest weighted relaxed query, $E_{\mathbf{Q}'}(1)$ (for a given triple pattern at a time). If the topmost score from the relaxed query obtained by relaxing a given triple pattern exceeds the k^{th} score from the original query, it predicts that the triple pattern's relaxations are required. Note that our estimator takes into account join score distributions and join cardinalities for estimating the expected score for a given query.

The query plan, **QP** returned by PLANGEN will have at most one subquery, $\mathbf{Q_1}$ of size > 1, called the "join group" (non-relaxed triple patterns), the rest will be only singletons (triple patterns to be relaxed).

3.2.2 Query Execution. Given a speculative query plan **QP** = $\{\mathbf{Q_1}, \mathbf{Q_2}, .., \mathbf{Q_s}\}$ with s subsets generated by the speculative query planner, we execute it in the following manner.

(1) The join group, $\mathbf{Q_1}$ is executed as (left-deep) rank joins over the answer lists (sorted by score) for each triple pattern. Note that, none of the triple patterns in this group are relaxed.
(2) The singletons are processed by Incremental Merge operator for each.
(3) Rank joins are performed over the join group and singletons.

Figure 2 illustrates this approach for the query **Q** = $\{q_1, q_2, q_3\}$, when we predict that only q_2 needs to be relaxed. The query plan to be executed is $\{\{q_1, q_3\}, \{q_2\}\}$. We use a rank join to compute the join between sorted lists of matches for q_1 and q_3 and require incremental merge only for q_2 and its relaxations. One rank join is required to join these results. The equivalent NSpec-QP plan for this query will be $\{\{q_1\}, \{q_2\}, \{q_3\}\}$, i.e., all triple patterns occur as different subsets and each of them are processed by incremental merges followed by rank joins over all these incremental merges (refer Figure 1).

4 EXPERIMENTAL EVALUATION

This section discusses the experimental evaluation performed for demonstrating the performance of the speculative planner.

4.1 Setup

Baseline. We compare Spec-QP with the NSpec-QP system (refer Section 2.1) which involves Incremental Merges for relaxations and Rank Joins for joins. It processes *all* the relaxations and outputs the true top-k. Existing works which focus on optimized computation of top-k joins without relaxations or on determining relaxations for user queries are orthogonal to our work. The scoring scheme used by the existing systems supporting relaxations do not use fine-grained scores (scores for individual triples). For our setting, NSpec-QP is the closest baseline to the best of our knowledge. We have not shown comparisons with the naive method (i.e., compute answers to all combinations of relaxations and then sort them to get top-k) because it is obvious that it is the most inefficient technique.

Datasets. We have evaluated over the following two datasets:

i. Extended Knowledge Graph (XKG)[42]:
   a. Format: RDF format dataset consisting of YAGO2s triples and "textual" content triples constructed from Clueweb by using OpenIE techniques and Named Entity Disambiguation (NED). The triple scores for YAGO2s triples are equal to the number of inlinks into the subject. Triple score for Clueweb data is equal to the number of times a particular triple was encountered during extraction.
   b. Size: XKG has about 105 million triples.
ii. Twitter:
   a. Format: Constructed from trending tags over the month of April 2017 using Twitter Streaming API. The triples are of the form: $\langle tID, hasTag, T\rangle$ where tID is the unique ID for a tweet containing term T. The score for each triple is equal to the number of retweets for the tweet in that triple.
   b. Size: 18 million unique triples.

Queries and relaxations. The evaluation queries and relaxations for the datasets are as follows:

i. XKG: We evaluated on 65 queries which were manually constructed so as to have non-empty result sets. Each query had 2-4 triple patterns and each triple pattern had at least 10 relaxations. The relaxations were obtained using the scheme outlined in [42].
ii. Twitter: A query over this dataset queries for IDs of those tweets which have all the queried terms. For example, the following query queries for IDs of all those tweets which contain the terms '`#intoyouvideo`', '`#ariana`' and '`dangerous`':

```
SELECT ?s WHERE{
    ?s <hasTag> <#intoyouvideo>.
    ?s <hasTag> <#ariana>.
```

k	XKG	Twitter
10	0.7	0.72
15	0.88	0.78
20	0.91	0.8

Table 2: Precision over each dataset.

```
    ?s <hasTag> <dangerous>
}
```

The testset of 50 queries was constructed manually using combinations of most frequent tags and terms. Each query had either 2 or 3 triple patterns, with each triple pattern having at least 5 relaxations. The relaxations were generated using the co-occurrence frequencies i.e. the relaxation weight, w for the relaxation, r = (⟨`?s <hasTag>` <T_1>⟩,⟨`?s <hasTag>` <T_2>⟩,w) will be equal to:

$$w = \frac{\#tweets\_having\_T_1\_and\_T_2}{\#tweets\_having\_T_1}$$

For example, a possible relaxation for `<#intoyouvideo>` is `<video>`.

Note that the number of results decrease on increasing the number of triple patterns in a query. Due to this, we have restricted our testsets to have only 2-4 triple patterns' queries with non-empty result sets. Also, even though each triple pattern has at least 5-10 relaxations, relaxing only one or two triple patterns alone generates about 100 additional answers. Our planner aims to be able to predict the useful relaxations.

Metrics. We measure the following metrics for each query to demonstrate the quality and efficiency of our technique:

i. Quality:
   a. Precision: The fraction of true top-k results (of NSpec-QP) in the top-k results of Spec-QP. Note that precision and recall have identical values in our setup, because they have the same denominator k.
   b. Prediction accuracy: The number of queries for which we could identify all and only correct relaxations.
   c. Score error: The average of absolute error for Spec-QP vs. NSpec-QP top-k scores, i.e., $\frac{1}{k}\sum_{i=1..k}\left|score_i^{NSpec-QP} - score_i^{Spec-QP}\right|$ We also note the standard deviation.

ii. Efficiency:
   a. Runtimes: We measure the time taken to plan and execute each query.
   b. Memory used: We measure the total no. of answer objects created as it directly corresponds to the amount of search space traversed to arrive at top-k answers. This number includes all the intermediate answer objects encountered by Incremental Merges and Rank Joins.

System setup. The experiments were conducted on a Dell Blade server with 24 Intel(R) Xeon(R) CPU E5-2420 @ 1.90GHz processors and 32GB RAM. The database engine was used to retrieve the matches for triple patterns in sorted order. Each query was evaluated using both the techniques- NSpec-QP and Spec-QP, over two database engines, postgresql-9.5 and Virtuoso, for three values of k, namely 10, 15 and 20. To have a warm cache, we conducted 5 consecutive runs for each query and considered the average of the last 3 runs for each technique.

4.2 Quality evaluations

We first discuss the quality of results obtained by Spec-QP and then provide the statistics for runtimes and memory consumptions. *Note that the quality of results will be the same over any database engine as it depends only on the accuracy of our speculative technique.*

4.2.1 Precision. The precision values for the datasets are given in Table 2. The precision is about 0.7-0.9 for both the datasets, i.e., about 80% of the answers belonged to true top-k. Also, since the answers are sorted according to the scores, the answers outside the true top-k appeared at lower ranks. That is, for a query having a precision value of 0.8 for k=10, top-8 answers belonged to the true top-10.

4.2.2 Prediction Accuracy. A detailed analysis of the number of queries for which we could predict the correct relaxation(s) over each dataset is given in Table 3. Each query required some triple patterns to be relaxed to generate top-k answers. The prediction accuracy is at least 70% for all types of queries over XKG and queries requiring 3 relaxations over Twitter. As the value for k was increased, queries increasingly required relaxations to generate sufficient answers. For Twitter, most of the queries required all triple patterns to be relaxed. This is due to the absence of sufficient triples corresponding to each term and fewer relaxations (predicate is not relaxed) for each triple pattern. Nevertheless, we were able to identify the requirement of all the relaxations in such a scenario.

4.2.3 Average score error. To judge the quality of approximate results returned by Spec-QP, we computed the score deviations of the approximate answers at each rank given by Spec-QP from the true top-k. The average score deviations for various values of k are given in Table 4. The percentages in brackets show the average percentage deviation from the original scores. Note that the maximum possible score for an answer to a 2 triple pattern query can be 2, for a 3 triple pattern query, it will be 3 and so on.[6]

XKG. Even though k=10 has lowest precision, the score deviations from true top-k answers are low (about 0.1 for 2 triple pattern queries). That is, for a query with 2 triple patterns if the actual answer at a given rank has a score of 1.5, the score of the approximate answer would be about 1.4. The deviations are even lower (only about 0.01) for higher values of k and tolerable for achieving faster runtimes.

Twitter. There is only one 2 triple pattern query that required both triple patterns to be relaxed but had a wrong speculation of relaxations for all values of k. However, its score deviation is constant over all values of k as it has only 11 results (including relaxations). The deviations are only 0.5 for 3 triple pattern queries with $k = 10$, which is only 16% deviation from the original scores. The deviations for higher values of k are very low being only 6% in the best case. For k=20, for a query with 3 triple patterns if the actual answer at a given rank has a score of 2.5, the score of the approximate answer is about 2.32.

4.3 Efficiency evaluations

The average runtimes and memory values over PostgreSQL and Virtuoso for XKG grouped by the number of triple patterns in

[6] This is because the maximum score for a matching triple for each triple pattern can be 1.

Dataset	XKG			Twitter		
k	10	15	20	10	15	20
queries requiring 1 relaxation	5(6)	5(5)	-(-)	-	-	-
queries requiring 2 relaxations	21(30)	22(26)	18(19)	1(2)	1(2)	1(2)
queries requiring 3 relaxations	12(18)	16(19)	27(31)	35(48)	38(48)	39(48)
queries requiring 4 relaxations	7(11)	14(15)	14(15)	-	-	-

Table 3: Prediction accuracy for various values of k grouped by the number of triple patterns requiring relaxations in the queries to generate true top-k results. The number indicates the number of queries for which Spec-QP could identify all and only these relaxations. The numbers in brackets show the total number of such queries.

Dataset	XKG			Twitter	
#TP k	2	3	4	2	3
10	0.1(5%)±0.1	0.2(8%)±0.3	0.1(3%)±0.2	0.16(8%)±0.0	0.5(16%)±0.5
15	0.08(4%)±0.08	0.1(3%)±0.2	0.01(1%)±0.04	0.16(8%)±0.0	0.32(10%)±0.3
20	0.07(4%)±0.06	0.07(2%)±0.1	0.01(1%)±0.03	0.16(8%)±0.0	0.18(6%)±0.1

Table 4: Avg. score deviations for the approximate top-k from the true top-k grouped by the number of triple patterns (#TP) in the queries. The percentages in brackets show avg. percentage deviation from the score of the true answer at that rank.

the queries and the number of relaxations required by them have been given in Figures 3 and 4 respectively. The graphs for Twitter are given in Figures 5 and 6.

4.3.1 Runtime comparisons. It is evident from the runtime graphs (refer Figures 3 and 5) that Spec-QP is faster than NSpec-QP in all cases. It avoids unnecessary computation of *all* relaxations when only few relaxations are capable of giving top-k answers. Most of the queries require only 2 or 3 relaxations (Refer Table 3) to produce top-k answers and Spec-QP either identifies the correct relaxation(s) or gives good quality approximate results. Also, fewer the number of relaxations required, faster is Spec-QP over NSpec-QP. For k=15 and k=20, the gain margin lowers but Spec-QP is still faster than NSpec-QP. This is because on seeking more answers, the original query is insufficient to get top-k answers and needs multiple relaxations. It is especially prominent for XKG queries with 4 triple patterns; for k=15 and k=20, none of the queries could get top-k answers with less than 3 relaxations. The difference in the runtimes however clearly demonstrates the savings achieved by eliminating the requirement of even 1 relaxation. In particular, Spec-QP outperforms NSpec-QP by a factor of upto 1.5 for queries with 3 triple patterns.

Note that the key optimization for sorted access, in any database engine, is to use ordered index scans. This is what PostgreSQL does too. Also, even if the underlying database system is further optimized for sorted results, both techniques would benefit and therefore the gains of Spec-QP over NSpec-QP would be of the same order. PostgreSQL is faster than Virtuoso as it uses indices intensively for optimized retrieval. We have shown results over Virtuoso to demonstrate the practical applicability of our technique over any quad store.

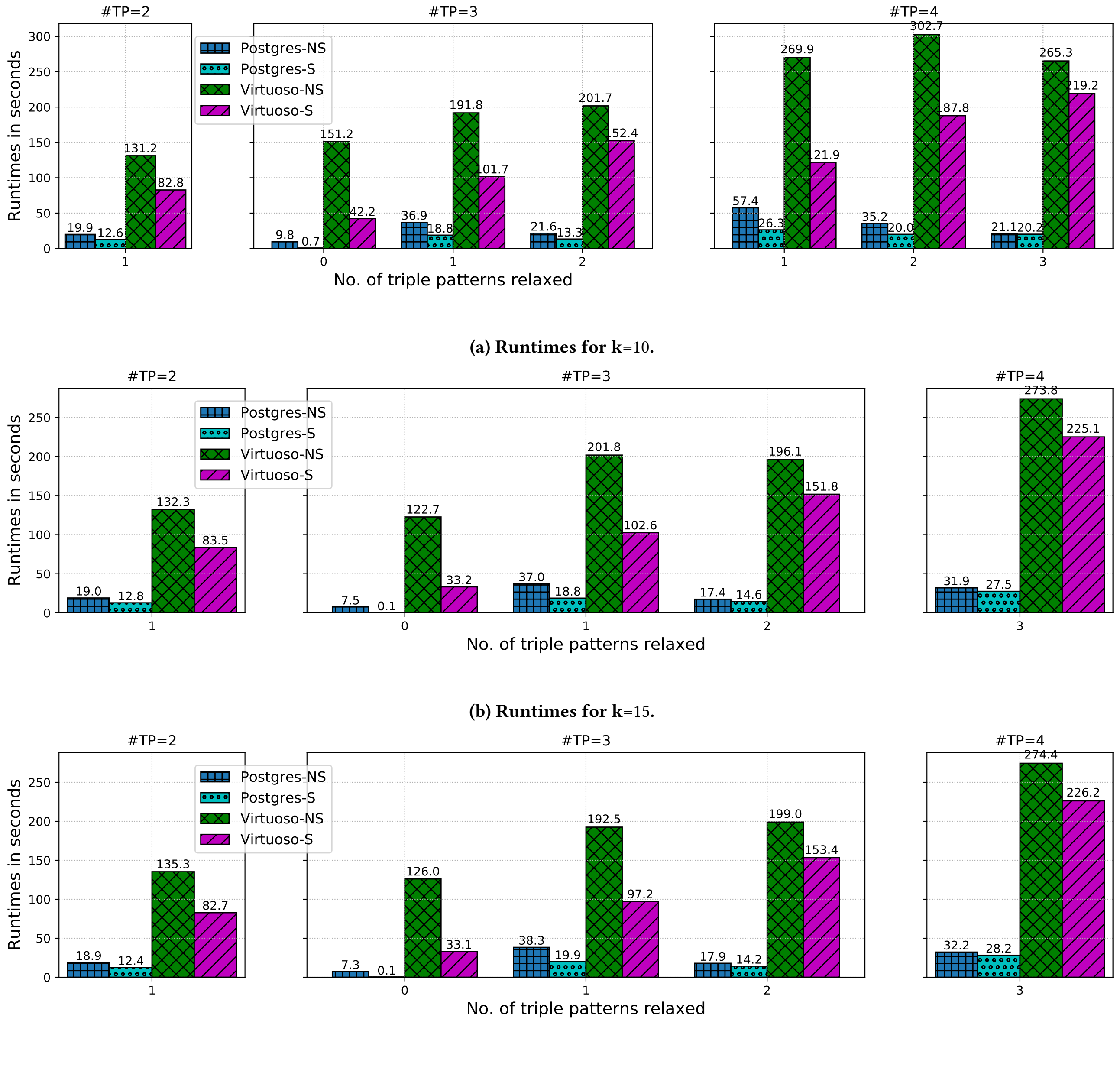
4.3.2 Memory requirement comparisons. We measured the total number of answer objects created as it directly corresponds to the amount of search space traversed to arrive at top-k answers. This number includes all the intermediate answer objects encountered by the incremental merges and rank joins.

The memory comparison graphs are given in Figures 4 and 6 for XKG and twitter respectively. We found that NSpec-QP consumes the most memory for all the cases. This is because it traverses a significant amount of the search space, consisting of the original query and all its possible relaxations, in order to compute the top-k. Spec-QP consumes upto 2-3x less memory to compute top-k answers as it prunes a significant amount of the search space. The savings by Spec-QP is achieved by eliminating the need for processing relaxations of triple patterns which do not contribute triples towards top-k answers. This is particularly evident for the cases where the queries with 3 triple patterns require only 1 or 2 relaxations. The memory requirements reduce by a factor of 2.5 over both the datasets.

4.4 Discussion and remarks

We showed that Spec-QP prunes unnecessary relaxations and has faster response times over the baseline for various values of k. It predicts the correct relaxations 70-80% of the time with good approximations for answers outside top-k. Our technique depends on statistical estimates – specifically, what the score of the k^{th} result is for a specific (original or relaxed) query. Our solution of 2-bucket approximation strives for the sweet spot between assuming a uniform distribution and the actual distribution (every single data point in individual buckets of the histogram), and is built on the fact that even though datasets are different, their score distributions typically follow a power law distribution – a "fat" head, and a "long" tail.

The strategic pruning of relaxations using our model also reduces the search space traversed and in turn, the memory requirements for each query. Spec-QP is especially useful for servers where the total resource consumption per query is inversely proportional to the achievable throughput. The cost per query determines the cost of running the server for a given load. For instance, the queries having 3 triple patterns have 1.5x faster response times and 2.5x less memory requirements resulting in an overall 4x gain. This implies that the server can run the service with 4x less money. Hence, Spec-QP provides cost-efficient support for flexible querying using relaxations over SPARQL query



(a) Runtimes for k=10**.**

(b) Runtimes for k=15**.**

(c) Runtimes for k=20**.**

Figure 3: Runtimes comparisons over XKG queries for k=10**,** 15 **and** 20 **grouped by the no. of triple patterns (#TP) in the query and the number of relaxations required. All the legends in the graphs for efficiency have 'NS' for NSpec-QP and 'S' for Spec-QP.**

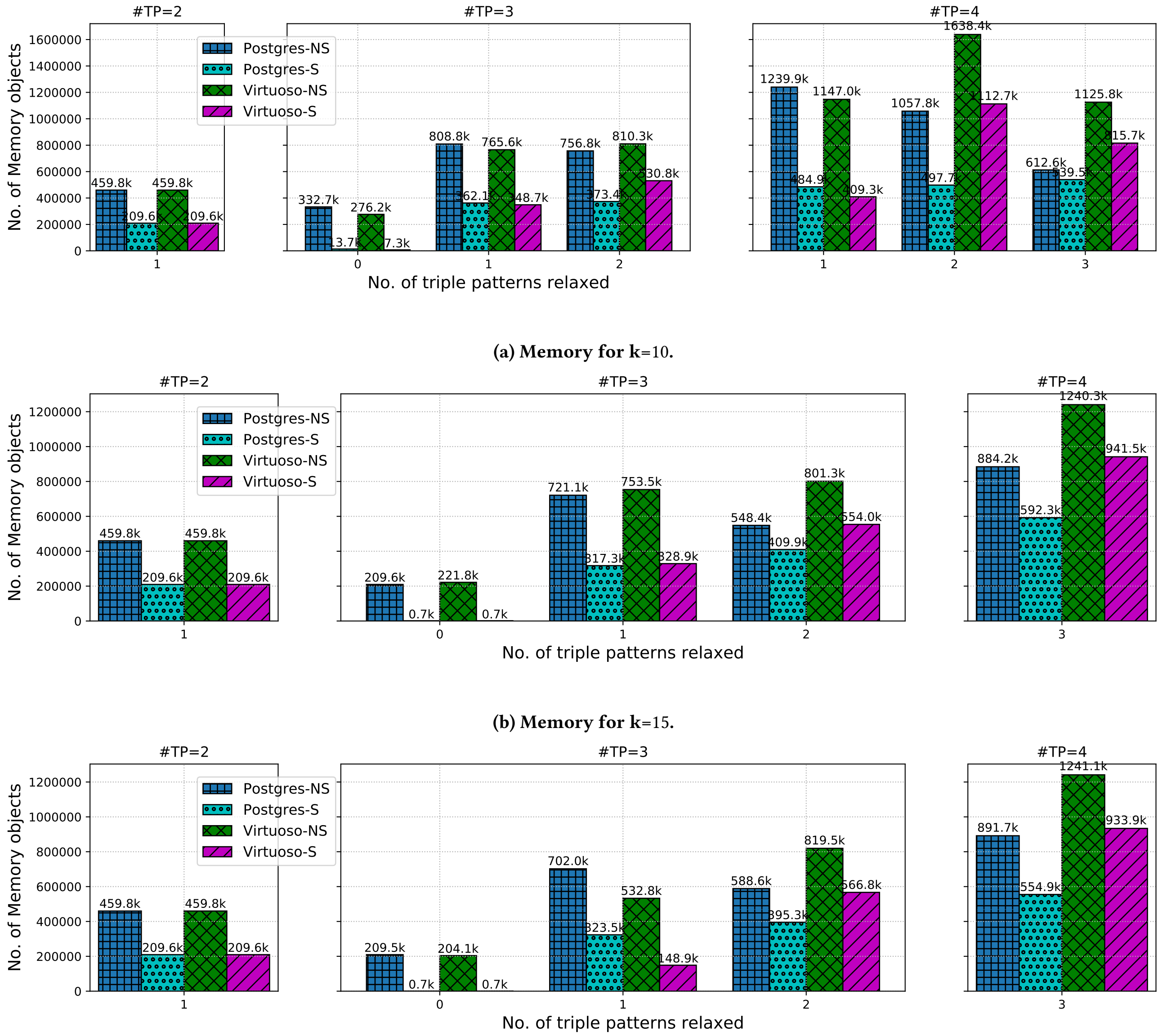
engines. This, in turn, aids effective exploration of knowledge graphs by new users.

5 RELATED WORK

Top-*k* query processing

FRPA [13] and Hash Rank-Join (HRJN*) [20] represent the state-of-the-art relational rank-join algorithms. HRJN* has been shown to perform well in practice, however, FRPA showed that it was not instance-optimal for a variant of the rank join problem that they considered. HRJN[21] is based on ripple join algorithm. It maintains two hash tables in-memory for storing the input tuples seen so far, the stored input tuples are used for finding join results. These results are then given as inputs to a priority queue, which outputs them in the order specified by the ranking function. Nested Loops Rank Join (NRJN) [19] is similar to HRJN except that unlike HRJN it does not store input tuples, but rather follows a nested-loop strategy. Pull/Bound Rank Join (PBRJ) [33] is an algorithm template that generalized previous rank join algorithms and provided tight upper bounds. DRJN [8] is an efficient algorithm for computing rank joins in distributed systems. This body of work is orthogonal to our problem.

Theobald et. al. [37] dealt with top-k query evaluation for joins over multiple index lists with pruning providing probabilistic guarantees. It uses histograms and dynamic convolutions to predict the top-*k*. Our case, however differs in that we consider



(a) Memory for k=10.

(b) Memory for k=15.

(c) Memory for k=20.

Figure 4: Memory comparisons over XKG queries for k=10, 15 and 20 grouped by the no. of triple patterns (#TP) in the query and the number of relaxations required. All the legends in the graphs for efficiency have 'NS' for NSpec-QP and 'S' for Spec-QP.

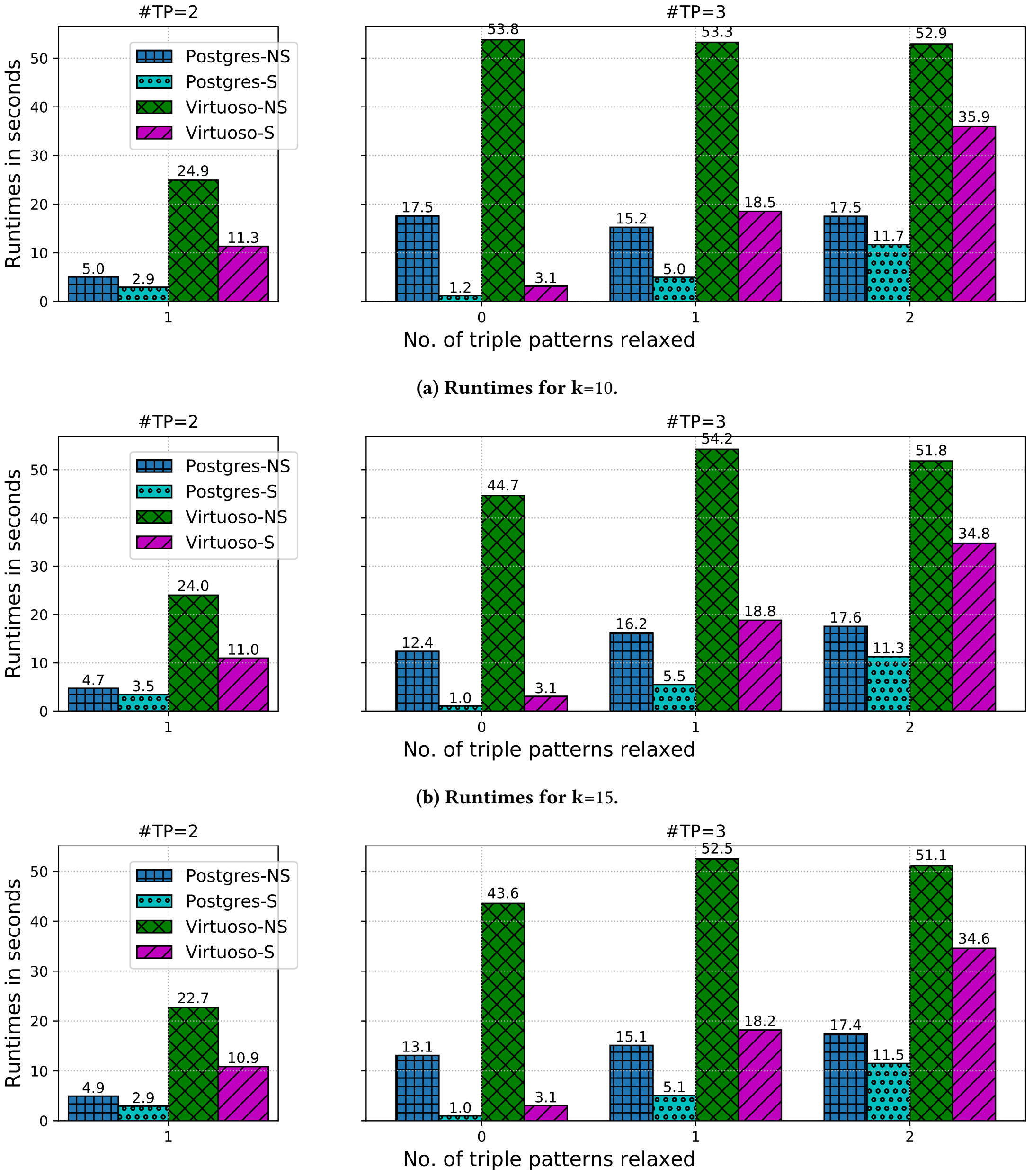
graph structured data and also, support multiple relaxations. The IO-Top-k [4] deals with top-*k* query evaluation with pruning using sorted access (SA) scheduling. Other works include top-*k* processing over xml data [36] and for data distributed over multiple nodes [44].

Top-*k* queries on graphs

Only few works address the problem of top-*k* processing over RDF graphs. The SPARQL-RANK framework proposed by [27] makes use of different index permutations used in native triple-stores for fast random access and early termination. Another framework introduced by Wang et al. [41] used MS-tree-based filtering and pattern-matching functions to evaluate top-*k* answers. The work in [43] uses an approach similar to HRJN[21] for computing top-*k* star joins. However for RDF data, SPARQL-RANK showed experimentally that it outperformed HRJN. The performance gain was attributed to the unsorted nature of numerical attributes present in indexes build by RDF engines. QUARK-X [25] proposes using extra indexes and metadata to process top-*k* queries on RDF graphs. All of these works however do not optimize over possible relaxations.

Query Reformulation in IR

Various strategies have been proposed to reformulate queries in IR over documents. These include measures of query similarity [3], or using summary information included in the query-flow graph [1]. Another approach by Hristidis et. al. [16] relies on



(a) Runtimes for k=10.

(b) Runtimes for k=15.

(c) Runtimes for k=20.

Figure 5: Runtimes comparisons over Twitter for k=10, 15 and 20 grouped by the no. of triple patterns (#TP) in the query and the number of relaxations required. All the legends in the graphs for efficiency have 'NS' for NSpec-QP and 'S' for Spec-QP.

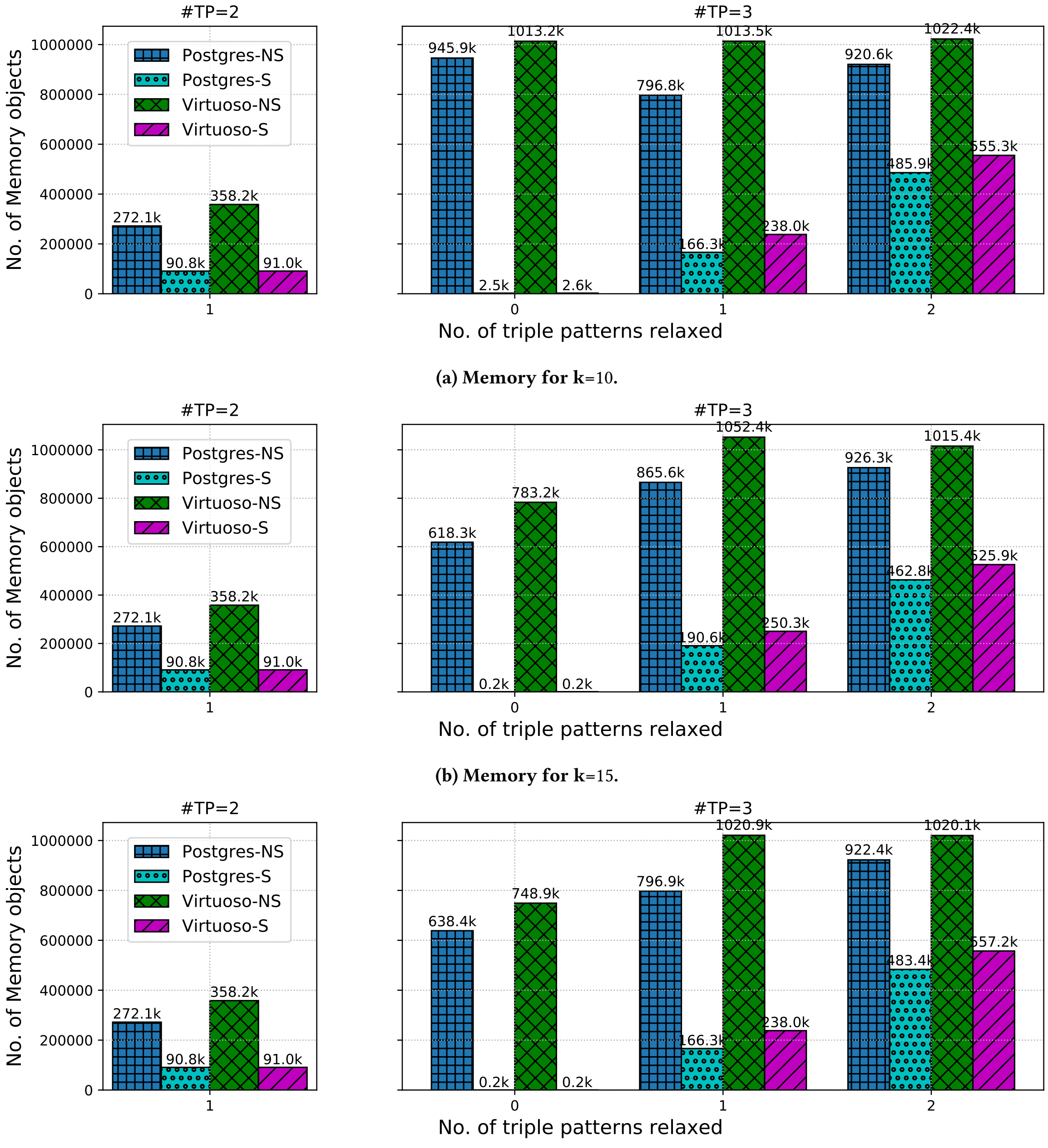
suggesting keyword relaxations by relaxing those which are least specific based on their idf score. These reformulations can be used as relaxations for our setting.

Faceted Search (Many answers problem)

A related optimization problem is the one encountered when we have many-answers, i.e. those where given an initial query that returns a large number of answers, the objective is to design an effective drill-down strategy to help the user find acceptable results with minimum effort [22, 26, 32]. We solve a related problem, where we try to solve both empty-answer and many-answers problem in an efficient manner by generating additional scored answers using relaxations.

Query Relaxation in relational databases

Query relaxation in relational databases is quite common. The work [24] relaxes joins and selections in relational databases by suggesting alternative queries based on the "minimal" shift from the original query. Another work [40] suggests user ranking of the query edges so as to generate relevant differential queries with minimum deviation. "Why Not" queries are studied in [6, 38], where, given a query Q that did not return a set of tuples S that the user was expecting to be returned, they design an alternate



(a) Memory for k=10.

(b) Memory for k=15.

(c) Memory for k=20.

Figure 6: Memory comparisons over Twitter for k=10, 15 and 20 grouped by the no. of triple patterns (#TP) in the query and the number of relaxations required. All the legends in the graphs for efficiency have 'NS' for NSpec-QP and 'S' for Spec-QP.

query Q' that (a) is very similar to Q, and (b) returns the missing tuples S, however the rest of the returned tuples should not be too different from those returned by Q. The paper [28] relaxes one constraint at a time and is interactive. It also tries to minimize the cost by suggesting low cost relaxations which lead to non-empty answers. DebEAQ [39] first tries to debug why the query is returning empty answer and then tries to relax it with minimum change to the original query. It is also limited only to property graphs.

Query Relaxation over graphs

The closest to our works are those which deal with relaxations over graphs. The work in [15, 29–31] considers query relaxation for conjunctive regular path queries. Users are allowed to specify query predicates which can have approximations and/or relaxations (using *APPROX* and *RELAX* operators respectively) during query time. The system then computes the approximations/relaxations with their relative evaluation costs to support query rewriting. Another work [17] computes approximate answers using a Bayesian network to rank and score relaxed queries. Two algorithms are described in [18]. The first algorithm is based on best-first strategy and relaxed queries are executed in order. They prune relaxations which do not give new results. The other algorithm executes the relaxed queries as a batch and avoids the unnecessary execution cost. The idea of Maximal Succeeding Subqueries (MSSs) is exploited in [14] using Lattice-based and Matrix-based approaches to minimally refine the user query. The scoring scheme used by these existing systems supporting relaxations however, do not use fine-grained scores (scores for individual triples) as in our case. TriniT [42] proposes the notion of eXtended Knowledge Graphs (XKG) with fine-grained scores for triples and allows relaxations for queries over them. It uses a technique similar to NSpec-QP to evaluate the queries.

6 CONCLUSION AND FUTURE WORK

We have proposed Spec-QP, a strategy for top-k query processing in a scenario where a query can have multiple relaxations. To achieve this, we have used a speculative approach for pruning the relaxations which are not likely to contribute answers to the top-k results. The speculation is based on precomputed statistics about the distribution of scores for triple pattern matches. The relaxations of triple patterns predicted to not contribute towards top-k answers are not processed, thereby reducing top-k computations and leading to faster response times and reduced memory requirements.

We have experimented over two real world datasets – XKG and Twitter – to show that Spec-QP is a cost-efficient technique for supporting relaxations. This is especially useful for servers in aiding exploratory querying over knowledge graphs by new users without an exponential increase in the budget. We also demonstrated the practical usability of our technique by implementing it over two popular database engines – PostgreSQL and Virtuoso. As future work, we would like to support more complicated relaxations for the queries like replacing a triple pattern with a chain of triple patterns, etc. Another orthogonal area of work is to find meaningful and useful relaxations for a given triple pattern.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. The first author was partially supported by the Student Exchange program under Indo-German Max Planck Center for Computer Science (IMPECS) and funds from Max Planck Institute for Informatics (MPI-INF) over an internship. She would also like to thank the alumni donors of the CS&E department at IIT Delhi for supporting her research.

REFERENCES

[1] Aris Anagnostopoulos, Luca Becchetti, Carlos Castillo, and Aristides Gionis. 2010. An optimization framework for query recommendation. In *WSDM*.

[2] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. 2007. DBpedia: A Nucleus for a Web of Open Data. In *ISWC*.

[3] Ricardo A. Baeza-Yates, Carlos A. Hurtado, and Marcelo Mendoza. 2004. Query Recommendation Using Query Logs in Search Engines. In *EDBT Workshops*.

[4] H. Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. 2006. IO-Top-k: Index-access Optimized Top-k Query Processing. In *VLDB*.

[5] Kurt D. Bollacker, Robert P. Cook, and Patrick Tufts. 2007. Freebase: A Shared Database of Structured General Human Knowledge. In *AAAI*.

[6] Adriane Chapman and H. V. Jagadish. 2009. Why not?. In *SIGMOD*.

[7] H.A. David and H.N. Nagaraja. 2004. *Order Statistics*. Wiley.

[8] Christos Doulkeridis, Akrivi Vlachou, Kjetil Nørvåg, Yannis Kotidis, and Neoklis Polyzotis. 2012. Processing of Rank Joins in Highly Distributed Systems. In *ICDE*.

[9] Shady Elbassuoni, Maya Ramanath, Ralf Schenkel, Marcin Sydow, and Gerhard Weikum. 2009. Language-model-based ranking for queries on RDF-graphs. In *CIKM*.

[10] Shady Elbassuoni, Maya Ramanath, and Gerhard Weikum. 2011. Query Relaxation for Entity-Relationship Search. In *ESWC*.

[11] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.* 66, 4 (2003).

[12] Azam Feyznia, Mohsen Kahani, and Fattane Zarrinkalam. 2014. COLINA: A Method for Ranking SPARQL Query Results through Content and Link Analysis. In *ISWC*.

[13] Jonathan Finger and Neoklis Polyzotis. 2009. Robust and efficient algorithms for rank join evaluation. In *SIGMOD*.

[14] Géraud Fokou, Stéphane Jean, Allel Hadjali, and Mickaël Baron. 2015. Cooperative Techniques for SPARQL Query Relaxation in RDF Databases. In *ESWC*.

[15] Riccardo Frosini, Andrea Calì, Alexandra Poulovassilis, and Peter T. Wood. 2017. Flexible query processing for SPARQL. *Semantic Web* 8, 4 (2017).

[16] Vagelis Hristidis, Yuheng Hu, and Panagiotis G. Ipeirotis. 2010. Ranked queries over sources with Boolean query interfaces without ranking support. In *ICDE*.

[17] Hai Huang and Chengfei Liu. 2010. Query Relaxation for Star Queries on RDF. In *WISE*.

[18] Hai Huang, Chengfei Liu, and Xiaofang Zhou. 2012. Approximating query answering on RDF databases. *WWW* 15, 1 (2012).

[19] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. 2003. Supporting Top-k Join Queries in Relational Databases. In *VLDB*.

[20] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. 2004. Supporting top-k join queries in relational databases. *VLDB J.* (2004).

[21] Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey Scott Vitter, and Ahmed K. Elmagarmid. 2004. Rank-aware Query Optimization. In *SIGMOD*.

[22] Abhijith Kashyap, Vagelis Hristidis, and Michalis Petropoulos. 2010. FACeTOR: cost-driven exploration of faceted query results. In *CIKM*.

[23] Gjergji Kasneci, Fabian M. Suchanek, Georgiana Ifrim, Maya Ramanath, and Gerhard Weikum. 2008. NAGA: Searching and Ranking Knowledge. In *ICDE*.

[24] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. 2006. Relaxing Join and Selection Queries. In *VLDB*.

[25] Jyoti Leeka, Srikanta Bedathur, Debajyoti Bera, and Medha Atre. 2016. *Quark-X*: An Efficient Top-K Processing Framework for RDF Quad Stores. In *CIKM*.

[26] Chengkai Li, Ning Yan, Senjuti Basu Roy, Lekhendro Lisham, and Gautam Das. 2010. Facetedpedia: dynamic generation of query-dependent faceted interfaces for wikipedia. In *WWW*.

[27] Sara Magliacane, Alessandro Bozzon, and Emanuele Della Valle. 2012. Efficient Execution of Top-K SPARQL Queries. In *ISWC*.

[28] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. 2013. A Probabilistic Optimization Framework for the Empty-Answer Problem. *PVLDB* 6, 14 (2013).

[29] Alexandra Poulovassilis. 2018. Applications of Flexible Querying to Graph Data. In *Graph Data Management, Fundamental Issues and Recent Developments*.

[30] Alexandra Poulovassilis, Petra Selmer, and Peter T. Wood. 2016. Approximation and relaxation of semantic web path queries. *J. Web Sem.* 40 (2016).

[31] Alexandra Poulovassilis and Peter T. Wood. 2010. Combining Approximation and Relaxation in Semantic Web Path Queries. In *ISWC*.

[32] Senjuti Basu Roy, Haidong Wang, Gautam Das, Ullas Nambiar, and Mukesh K. Mohania. 2008. Minimum-effort driven dynamic faceted search in structured databases. In *CIKM*.

[33] Karl Schnaitter and Neoklis Polyzotis. 2010. Optimal algorithms for evaluating rank joins in database systems. *ACM Trans. Database Syst.* (2010).

[34] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: a core of semantic knowledge. In *WWW*.

[35] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. 2005. Efficient and self-tuning incremental query expansion for top-k query processing. In *SIGIR*.

[36] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. 2005. An Efficient and Versatile Query Engine for TopX Search. In *VLDB*.

[37] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. 2004. Top-k Query Evaluation with Probabilistic Guarantees. In *VLDB*.

[38] Quoc Trung Tran and Chee-Yong Chan. 2010. How to ConQueR why-not questions. In *SIGMOD*.

[39] Elena Vasilyeva, Thomas Heinze, Maik Thiele, and Wolfgang Lehner. 2016. DebEAQ - debugging empty-answer queries on large data graphs. In *ICDE*.

[40] Elena Vasilyeva, Maik Thiele, Christof Bornhövd, and Wolfgang Lehner. 2014. Top-k Differential Queries in Graph Databases. In *ADBIS*.

[41] Dong Wang, Lei Zou, and Dongyan Zhao. 2015. Top-k queries on RDF graphs. *Inf. Sci.* 316 (2015).

[42] Mohamed Yahya, Denilson Barbosa, Klaus Berberich, Qiuyue Wang, and Gerhard Weikum. 2016. Relationship Queries on Extended Knowledge Graphs. In *WSDM*.

[43] Shengqi Yang, Fangqiu Han, Yinghui Wu, and Xifeng Yan. 2016. Fast top-k search in knowledge graphs. In *ICDE*.

[44] Hailing Yu, Hua-Gang Li, Ping Wu, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Processing of Distributed Top-k Queries. In *DEXA*.

Iterative Estimation of Mutual Information with Error Bounds

Michael Vollmer
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
michael.vollmer@kit.edu

Klemens Böhm
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
klemens.boehm@kit.edu

ABSTRACT

Mutual Information (MI) is an established measure for linear and nonlinear dependencies between two variables. Estimating MI is nontrivial and requires notable computation power for high estimation quality. While some estimation techniques allow trading result quality for lower runtimes, this tradeoff is fixed per task and cannot be adjusted. If the available time is unknown in advance or is overestimated, one may need to abort the estimation without any result. Conversely, when there are several estimation tasks, and one wants to budget computation time between them, there currently is no efficient way to adjust it dynamically based on certain targets, e.g., high MI values or MI values close to a constant. In this article, we present an iterative estimator of MI. Our method offers an estimate with low quality near-instantly and improves this estimate in fine grained steps with more computation time. The estimate also converges towards the result of a conventional estimator. We prove that the time complexity for this convergence is only slightly slower than non-iterative estimation. Additionally, with each step our estimator also tightens statistical guarantees regarding the convergence result, i.e., confidence intervals, progressively. These also serve as quality indicators for early estimates and allow to reliably discern between attribute pairs with weak and strong dependencies. Our experiments show that these guarantees can also be used to execute threshold queries faster compared to non-iterative estimation.

1 INTRODUCTION

Motivation. Detecting and quantifying dependencies between variables is an essential task in the database community [10, 13, 20, 30]. Conventional methods such as correlation coefficients and covariance matrices only detect linear or monotonous dependencies. *Mutual Information* (MI) in turn is an index that captures any linear and nonlinear dependency [1, 5]. Probability distributions of the variables in question serve as input to compute the MI. For real-world data however, these distributions are not available. In this case, MI must be estimated based on samples.

Various estimators for MI have been proposed [15, 23, 33], and some offer good results even for small samples [15]. However, continuous variables with an unknown distribution continue to be challenging, since their multivariate distribution is substituted only by a limited sample. A prominent approach for estimation of MI between continuous variables without assumption of the distribution is the nearest-neighbor based method by Kraskov et al. (KSG) [19].

While good estimators are available, they are very rigid in their time requirements and regarding the estimation quality. Once the computation has started, they impose a fixed time requirement and do not yield aby preliminary result if they are terminated prematurely. They also are unable to exploit 'easier' queries like whether the MI value is above a certain threshold but instead determined the value. Such features are highly relevant for high-dimensional data and data streams with irregular arrival rate as we showcase with the following two scenarios.

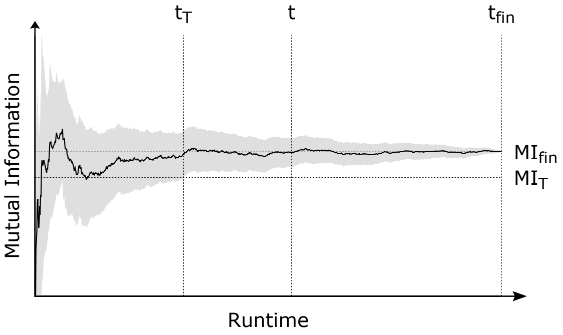


Figure 1: MI estimation with dynamic time allocation.

Scenario 1. Consider a modern production plant with smart meters installed on each machine. A first step in data exploration is determining which attributes are strongly dependent. For instance dependencies among currents or energy consumption may offer insights into production sequences. For this first step, a query like "Which pairs of measurements have a MI value above the threshold MI_T?" often suffices. With conventional MI estimators, each pair either induces high computational costs, or results are uncertain because of low estimation quality.

Scenario 2. Think of a database with financial data and its real-time analysis. To maintain a diverse portfolio, it is important to track the relationships between stocks. Because bids and trades happen irregularly, new information and market prices arrive at irregular speed. Thus, it is not known how much time is available to monitor stock relationships in the presence of incoming data. Current MI estimators cannot adapt during execution. They risk not producing a result in time, or estimates are of low quality.

To improve upon these shortcomings, we study estimation of MI with dynamic allocation of computation time. Ideally, such an estimator should not only offer preliminary results, but also indicate its remaining uncertainty. Figure 1 shows exemplary progression over time of such an estimator based on our experiments with real data. The black line indicates the preliminary estimate after a certain runtime, and the gray area shows the (expected) maximum error of the preliminary estimate. To obtain the definitive result $\mathtt{MI}_{\mathtt{fin}}$, a user would require time $\mathtt{t}_{\mathtt{fin}}$. However, he could also stop the estimator as soon as the estimate is above a threshold $\mathtt{MI}_{\mathtt{T}}$ with certainty, or he can use the preliminary result available after time $\mathtt{t}$.

In this work, we focus on iterative estimation of MI in order to offer this functionality. Here, 'iterative' means quickly providing an estimate, but with the option to improve the estimation if there is time left. In other words, improving the estimate with some time available is what we call an *iteration.* At the same time, an iterative estimator can terminate the estimation, i.e., stop iterating, when the result is good enough. For efficiency, it is important that computations from previous iterations remain useful and are not repeated or discarded in a later iteration. So far, efficient iterative estimators for MI do not exist.

Challenges. The most significant feature of an estimator is its quality of estimation. This is even more so for iterative methods because both "preliminary" and "final" estimation quality are important. In other words, the estimate should already be useful after a few iterations, and estimation quality must level up to the one of conventional estimators after many iterations. Ideally, this convergence should happen after a known, finite number of iterations. In this article, we target at respective formal guarantees.

Next, the quality of preliminary estimates is crucial for usability. Determining if a preliminary result is "good enough" or interesting enough to merit additional computation time requires some information on its certainty. The number of iterations alone is insufficient, as the result quality depends on many other factors such as data characteristics, required accuracy and time constraints. Instead, each estimate requires an individual indicator of the uncertainty remaining.

While the time spent to improve the estimate iteratively is committed dynamically, it must of course be used efficiently. Many conventional estimators use data structures that are expensive to build and cheap to use, such as space-partitioning trees [19, 31, 32]. Such an upfront activity is undesirable for an iterative estimator whose first estimate must arrive soon. At the same time, runtime and scalability do remain important characteristics of the estimator. In other words, an iterative estimator must feature guaranteed efficiency for both individual iterations and final estimates.

Contributions. In this article, we present IMIE, our Iterative Mutual Information Estimator. To prove its practical usefulness, we establish several features both formally and experimentally.

Quality of Estimation. In Section 4, we propose a design for IMIE such that estimates converge to the same value as with the KSG. To make early iterations useful, IMIE also offers statistical error bounds for its early estimates. More precisely, an early estimate provides a confidence interval for the final estimate. We describe the specifics and the statistical soundness in Section 4.3.

Complexity. We study the time complexity of initialization and of individual iterations of IMIE. In Section 5 we establish an amortized time complexity for IMIE and the nearest-neighbor search used. This complexity is competitive with existing non-iterative estimators. To be precise, we show that iterating IMIE until convergence is only slightly slower in terms of time complexity than computing the KSG directly with optimal algorithms.

Experimental Validation. We show that IMIE complements the formal guarantees established so far with good actual performance. To do so, we perform extensive experiments using both synthetic and real data sets in Section 6. On the one hand, we show that the concrete runtime and estimation results of IMIE are comparable to the ones of conventional estimation methods. On the other hand, the experiments show the practical benefits of the early results from IMIE. For instance, IMIE finds attribute pairs above a threshold value significantly faster than non-iterative estimators.

2 RELATED WORK

Iterative estimation of MI is interesting from two perspectives. On the one hand, it is methodically interesting, as it can be considered an *anytime algorithm.* On the other hand, it is interesting to consider the benefits it provides over current methods in different settings. Important application scenarios are dependency analysis in high dimensional data and data streams, cf. Scenario 1 and 2.

Anytime Algorithms. Anytime algorithms [36] use available time to increase their result quality. One can obtain a low-quality result after a short time and a better one when waiting longer. In data analysis, anytime algorithms exist for clustering [22], classification [35] and outlier detection [2]. So far however, there is no anytime algorithm to estimate MI. So while there is no direct competitor, IMIE extends the set of tools available as anytime algorithms. Additionally, there has been more general work on the optimal use of available anytime algorithms [11, 18], which may improve the performance of IMIE in larger systems.

MI on Data Streams. Estimating MI on streams has received some attention recently. The MISE framework [14] summarizes a bivariate stream such that the MI for arbitrary time frames can be queried. To this end, MISE offers parameters for the balance between accuracy of older queries and resource requirements both in terms of memory and computation time. In contrast, the DIMID estimator [4] processes a bivariate stream as sliding window for monitoring tasks. This approach provides fast updates between time steps by approximation with random projection. MI estimation in sliding windows has also been the focus of [32]. That paper provides lower bounds for estimates using Equation 5 both in general and for updates in sliding windows. It also features two dynamic data structures, DEMI and ADEMI, to maintain such estimates using either simple or complex algorithms and data structures.

These approaches have limitations. First, they all impose the necessary execution time, i.e., one cannot adapt this time after the start of stream processing. If the rate of new items increases, the estimator may be unable to keep up. If it decreases, the estimator cannot use this time to improve results. Second, the approaches are all focused on bivariate streams. While MI is defined for exactly two variables, the number of attribute pairs grows quadratically in the number of dimensions. In contrast, the only information IMIE maintains on a stream is based on individual dimensions and thus scales linearly with the dimensionality. Third, the approximate results of MISE and DIMID are difficult to use. Their estimation quality is only known on average; this average defines the perceived quality of individual estimates. So if one estimate has a very small error, it is less likely to be appreciated, while the error of a particularly bad estimate may be assumed to be smaller.

Dependencies in High Dimensional Data. Even though MI is defined for exactly two variables, it has many applications with high-dimensional data. Prominent ones are image registration [25], which uses MI between two high-dimensional variables, and feature selection [24], which targets at the MI between attributes and a classification label. But estimating the MI between all pairs of attributes has received little attention, despite being the non-linear equivalent of correlation matrices. [26] uses a different approach, i.e., kernel density estimation, and removes redundant computations that arise when using this estimator for each pair. This approach has a worse computational complexity than a pair-wise application of the KSG estimator, without offering better

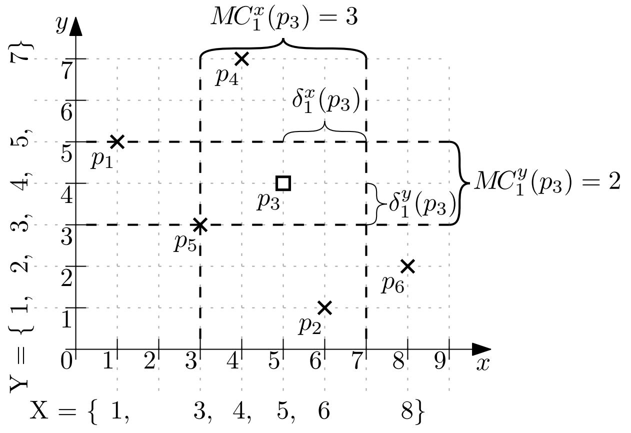


Figure 2: Illustration of terms used for the KSG.

results [15, 23]. While both scale quadratically in the number of attributes, their approach is also quadratic in the number of points. The complexity of the KSG in turn is $\Theta(n \log n)$ [32]. Additionally, it does not expose any parameter to modify the result quality. Consequently, there would not be any benefit of a direct experimental comparison with IMIE.

3 FUNDAMENTALS

We first cover the background of MI and its estimation.

Mutual Information. Shannon has introduced the notion of *entropy* [28] to quantify the expected information gained from observing a value of a random variable. $H(X)$ stands for the entropy of a random variable X. The expected information of observing two random variables X and Y is the *joint entropy* $H(X, Y)$. *Mutual Information* quantifies the amount of information that is shared or redundant between the two variables. It is defined as

$$I(X; Y) = H(X) + H(Y) - H(X; Y). \tag{1}$$

With the definition of entropy for continuous variables [6], the MI of two continuous random variables is

$$I(X; Y) = \int_X \int_Y p_{XY}(x, y) \, \log \left(\frac{p_{XY}(x, y)}{p_X(x) p_Y(y)} \right) dx \, dy, \tag{2}$$

where p_X, p_Y and p_{XY} are the marginal and joint probability density functions of X and Y. The type of logarithm used in Equation 2 determines the unit of measurement. In this work we use the natural logarithm. This means that MI is measured in the *natural unit of information* (nat).

Estimation. One can perceive many sources of data, e.g., smart meters or market prices, as random variables with unknown distribution. Since Equation 2 requires probability density functions, we cannot compute the MI of such sources exactly. Instead, we can only estimate the MI based on available samples. The popular estimator that will serve as foundation of our work is the one by Kraskov, Stögbauer and Grassberger [19], which we call KSG. It is based on the estimator for probability densities by Loftsgaarden and Quesenberry [21], which Kozachenko and Leonenko have studied further in the context of entropy [17]. In the following, we briefly review the terms and computation of the KSG.

Let $P = \{p_1 = (x_{p_1}, y_{p_1}), \ldots, p_n = (x_{p_n}, y_{p_n})\} \subseteq \mathbb{R}^2$ be a sample from a random variable with two attributes. Figure 2 illustrates the notions that we define in the following using the sample $P = \{(1, 5), (6, 1), (5, 4), (4, 7), (3, 3), (8, 2)\}$. Let $X = \{x_{p_1}, \ldots, x_{p_n}\}$ and $Y = \{y_{p_1}, \ldots, y_{p_n}\}$ be the set of values per attribute. For each point $p \in P$, its $k \in \mathbb{N}^+$ nearest neighbors in P using the maximum distance form the set $kNN(p)$. More formally, it is

$$kNN(p) = \underset{S \subseteq (P \setminus \{p\}) \; s.t. \; |S|=k}{\arg\min} \; \max_{s \in S} \|p, s\|_\infty, \tag{3}$$

with $\|p, s\|_\infty = \max(|x_p - x_s|, |y_p - y_s|)$. We define the largest distance between x_p and any x-value among the k nearest neighbors of p as $\delta_k^x(p) = \max_{s \in kNN(p)} |x_p - x_s|$. We use this distance $\delta_k^x(p)$ to define the *x-marginal count*

$$MC_k^x(p) = |\{x \in (X \setminus x_p) : |x - x_p| \leq \delta_k^x(p)\}|, \tag{4}$$

which is the number of points whose x-value is "close to p". In Figure 2, vertical dashed lines mark the area of points whose x-values are at least as close as the nearest neighbor of p_3. Since this area contains three points excluding p_3, it is $M_1^x(p_3) = 3$. The distance $\delta_k^y(p)$ and the *y-marginal count* $MC_k^x(p)$ are defined analogously. Note that $\delta_k^x(p)$ and $\delta_k^y(p)$ may differ, which results in differently sized areas for the marginal counts, as seen in Figure 2. Using these counts, the KSG estimate is defined as

$$\widehat{I}(P) = \psi(n) + \psi(k) - \frac{1}{k} - \frac{1}{n} \sum_{i=1}^{n} \psi\left(MC_k^x(p_i)\right) + \psi\left(MC_k^y(p_i)\right), \tag{5}$$

where ψ is the digamma function. This is $\psi(z) = -C + \sum_{t=1}^{z-1} \frac{1}{t}$ for $z \in \mathbb{N}^+$ and $C \approx 0.577$ being the Euler-Mascheroni constant.

While k is a parameter of this estimator, it is generally recommended [15, 16, 19] to use a small k, that is $k \leq 10$. Gao et al. [9] have proven that the KSG is a consistent estimator for fixed k, that is, it converges towards the true value with increasing sample size.

4 ITERATIVE ESTIMATION

In this section we present IMIE, our iterative estimator for MI. The core concept of our approach is considering the KSG estimate itself as the mean of a random variable with a finite population. Using subsamples of this population for early estimates offers beneficial properties such as an expected value equal to the KSG estimate and convergence to the KSG for large sample sizes.

We first present IMIE and its underlying data structure as well as the algorithms for the initialization and for subsequent iterations. Then we describe our approach for nearest neighbor search, which is better for iterative algorithms than the standard procedures. Finally, we describe the statistical bounds that IMIE provides with its estimates.

4.1 IMIE

For brevity, we introduce some notation in addition to the one from Section 3. For a point $p \in P$, we define the *pointwise estimate*

$$\Psi(p) = \psi\left(MC_k^x(p)\right) + \psi\left(MC_k^y(p)\right). \tag{6}$$

The set of all pointwise estimates is $\rho = \{\Psi(p_1), \ldots, \Psi(p_n)\}$. Seeing ρ as a finite population of size n with mean μ_ρ, Equation 5 can be rewritten as

$$\widehat{I}(P) = \psi(n) + \psi(k) - \frac{1}{k} - \mu_\rho. \tag{7}$$

Using a (random) subsample $\varrho \subseteq \rho$, its mean μ_ϱ is an (unbiased) estimation of μ_ρ. This in turn yields an (unbiased) estimate of $\widehat{I}(P)$,

$$\widehat{I_\varrho}(P) = \psi(n) + \psi(k) - \frac{1}{k} - \mu_\varrho. \tag{8}$$

Data Structure 1: IMIE

```
struct {
    Point[] $P$
    Real $Mean$, $Var$
    Int $k, m$
    Int[] $Order_R$, $Order_x$, $Order_y$
    Real $Offset$
};
```

Algorithm 2: Init (P, k)

```
Persist $k$ and $P$                                         $O(n)$
$Mean, Var, m \leftarrow 0$                                 $O(1)$
$Order_R, Order_x, Order_y \leftarrow (0, 1, \dots, |P|-1)$ $O(n)$
Sort $Order_x$ and $Order_y$                                $O(n \log n)$
$Offset \leftarrow \psi(|P|) + \psi(k) - \frac{1}{k}$       $O(1)$
```

Algorithm 3: Iterate

```
1  $ID \leftarrow$ Draw random integer from $[m, n-1]$                      $O(1)$
2  Swap values of $Order_R[m]$ and $Order_R[ID]$                            $O(1)$
3  $p \leftarrow P[Order_R[m]]$                                             $O(1)$
4  $kNN(p) \leftarrow$ NNSearch$(p)$          (see Algorithm 4)             $O(\sqrt{n})$
5  Compute $\delta^x_k(p)$, $\delta^y_k(p)$                                 $O(1)$
6  Compute $MC^x_k(p)$, $MC^y_k(p)$                                         $O(\log n)$
7  $\Psi(p) \leftarrow \psi\big(MC^x_k(p)\big) + \psi\big(MC^y_k(p)\big)$  $O(1)$
8  $m \leftarrow m + 1$                                                     $O(1)$
9  $Diff_{\text{old}} \leftarrow \Psi(p) - Mean$                            $O(1)$
10 $Mean \leftarrow Mean + \frac{Diff_{\text{old}}}{m}$                     $O(1)$
11 $Diff_{\text{new}} \leftarrow \Psi(p) - Mean$                            $O(1)$
12 $Var \leftarrow \frac{Var\cdot(m-1)+Diff_{\text{old}}\cdot Diff_{\text{new}}}{m}$   $O(1)$
```

The variance σ^2_ϱ of our subsample serves as a quality indicator of this approximation, which we further discuss in Section 4.3. The idea of IMIE is to maintain a subsample ϱ and use $\widehat{I_\varrho}(P)$ to estimate $\widehat{I}(P)$. Each iteration then increases the sample size of ϱ by one, to improve the estimate. Starting with an empty set, this means there are exactly $|P|$ iterations before IMIE yields exactly the same result as the KSG, i.e., $\widehat{I_\varrho}(P) = \widehat{I}(P)$.

Data Structure. IMIE uses and stores P and k as well as some additional information listed in Data Structure 1. In the following we use the zero-indexed array notation $P[i] = p_{i+1}$. Contrary to the original data sample P, we do not store ϱ explicitly. Instead we store its mean *Mean*, its variance *Var* and size, which is the number of performed iterations m. To maintain the current variance efficiently, we use the online algorithm by Welford [34]. To ensure that ϱ is a random subsample of ρ, we need to draw without replacement. To this end, IMIE maintains an array of indices $Order_R$, where index i at position j means that $\Psi(p_i)$ is added to ϱ in the j-th iteration. The positions of this array are randomly swapped during iterations to perform the random selection. This enables a fast selection of a random element without replacement in each iteration. In addition, we maintain two arrays $Order_x$ and $Order_y$ containing references to all points in P ordered by their x- and y-value, respectively. For instance, index i at $Order_x[0]$ means that p_i has the smallest x-value in P, i.e., $p_i = \arg\min_{p \in P} x_p$. These ordered arrays are used to find nearest neighbors, as described in Section 4.2. Finally, we store the $Offset = \psi(n) + \psi(k) - \frac{1}{k}$. With this, the (preliminary) MI estimate is available as $\widehat{I_\varrho}(P) = Offset - Mean$.

Methods. We now present the two methods Init and Iterate. See Algorithms 2 and 3, together with amortized time complexities, derived in Section 5. Init ensures the proper state of Data Structure 1 before the first iteration, i.e., preparing all variables assuming that $|\varrho| = 0$. Observe that Init is a straightforward method for the simple case of static data with two attributes. For other scenarios, such as high-dimensional or streaming data, some adjustments to the initialization may be appropriate, as discussed in Section 5.3.

Iterate increases the size of sample ϱ by one. This requires computing $\Psi(p)$ for a random $p \in P$ with $\Psi(p) \notin \varrho$. Iterate consists of three phases. In the first one (Lines 1–3), we select a random point p of P that has not been selected earlier. After $m-1$ iterations, we swap the index at position m of $Order_R$ with the index at a random position behind $m-1$. This ensures that we do not use any index twice, since positions before m are not considered, and that each unused index has the same probability of being selected. This random swap is one step of the Fisher-Yates Shuffle in the version of Durstenfeld [8], which fully randomizes the order of a sequence. The second phase (Lines 4–7) computes $\Psi(p)$ using the ordered lists $Order_x$ and $Order_y$. The last phase (Lines 8–12) performs the online algorithm [34] to maintain mean and variance of a sample, in our case ϱ.

Example 4.1. Disregarding the dashed lines for now, Figure 3 illustrates the state of Data Structure 1 after initialization and before the first iteration. For the first iteration, we draw an integer ID from $\{0, \dots, n-1\}$. Suppose that we drew 5. We swap the content of $Order_R[0]$ and $Order_R[5]$. $Order_R[0]$ now contains 6. This means that this iteration adds $\Psi(p_6)$ to our implicit sample ϱ. We then determine its nearest neighbor $1NN(p_6) = \{p_{15}\}$, the distances $\delta^x_1(p_6)$ and $\delta^y_1(p_6)$ as well as the marginal counts $MC^x_1(p_6) = 1$ and $MC^y_1(p_6) = 3$. The dashed lines in Figure 3 illustrate the area of counted points in x and y-direction, respectively, identically to Figure 2. It follows that $\Psi(p_6) = \psi(1)+\psi(3) = 0.346$. Substituting the appropriate variables, the remaining values are set accordingly, i.e., $m = 0 + 1 = 1$, $Mean = 0 + \frac{0.346}{1} = 0.346$ and $Var = \frac{0\cdot 0+0\cdot 0.346}{1} = 0$. The second iteration is analogous, drawing $ID = 6$ at random from $\{1, \dots, n-1\}$, thus choosing p_7. Its nearest neighbor is p_8, and the marginal counts are $MC^x_1(p_7) = 1$ and $MC^y_1(p_7) = 6$, cf. the dashed lines in Figure 4. As a result, it is $\Psi(p_7) = \psi(1)+\psi(6) = 1.129$. Analogously to the first iteration, the remaining values are $m = 1+1 = 2$, $Mean = 0.346 + \frac{0.783}{2} = 0.738$ and $Var = \frac{0\cdot 1+0.783\cdot 0.391}{2} = 0.153$. Figure 4 graphs the state of Data Structure 1 after both iterations, and the new MI estimate is $1.164 - 0.738 = 0.426$.

4.2 Nearest-Neighbor Search

A computation-intensive step in Iterate is the computation of nearest neighbors, which also is a key step for static estimation with the KSG. The classic solution [19, 31] is using space-partitioning trees, which are optimal in terms of computational complexity [32]. This efficiency is achieved because the slow tree construction is performed once, and each nearest-neighbor search afterwards is fast. Contrary to the traditional KSG estimation, it is not known beforehand how many nearest-neighbor searches IMIE performs. Constructing such a tree for IMIE would

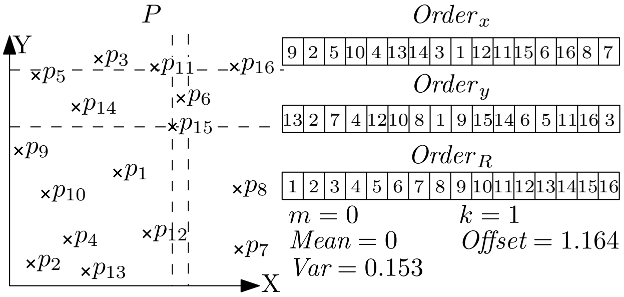


Figure 3: State of IMIE after initialization.

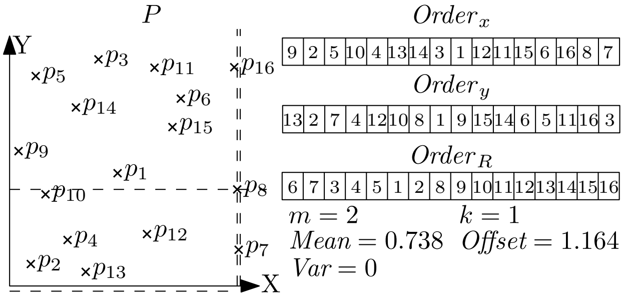


Figure 4: State of IMIE after two iterations ($\Psi(p_6)$ and $\Psi(p_7)$).

not only delay the first estimate, but may also be an inefficient choice overall if only few iterations take place. The opposite, i.e., searching nearest neighbors without any preparation, is a linear search. Each iteration would then require time linear in the number of data points. Since IMIE should offer both fast iterations and preliminary estimates after a short time, our approach is a compromise between these two options. The general idea is to use sorted arrays to perform a "guided" linear search that offers a good amortized time complexity (cf. Section 5). In the following, we elaborate on our NNSearch approach.

Let p be the point whose nearest neighbor we are searching for and q the nearest neighbor we have found so far. Then any point r with $|x_p - x_r| > \|p-q\|_\infty$ cannot be a nearest neighbor with the maximum norm. This means that we only have to consider the interval $[x_p - \|p-q\|_\infty, x_p + \|p-q\|_\infty]$ in the sorted array $Order_x$. When we find a closer point during the search, this interval gets smaller, and fewer points need to be considered. For the y-values, this is analogous. To reduce the number of worst-case scenarios, we perform this search simultaneously in both directions and terminate when either one terminates. See Algorithm 4 for an algorithmic description of NNSearch.

Example 4.2. Figure 5 illustrates an exemplary run of this procedure for $k = 1$. The figure shows four states corresponding to the variables of NNSearch(p) after $0, \ldots, 3$ loops. The query point p is the filled square, and a projection of the points to their x- and y-coordinates is shown at the bottom and the left side, respectively. These projections indicate the order of points in $Order_x$ and $Order_y$, respectively. Each state after the first loop also illustrates the variables of NNSearch. The nearest neighbor found so far is marked with a circle and is labeled NN, and the distance $\delta_{\max} = \|p - NN\|_\infty$ is used for the dashed lines that highlight the remaining area of nearest neighbor candidates. Points accessed via $Order_x$ in a previous iteration are marked with a diagonal stripe from the upper left to the lower right. This is done analogously for $Order_y$. Each loop considers the next unmarked point in both directions for both $Order_x$ and $Order_y$. Additionally, the small arrows illustrate the minimal distances $\Delta_{\circ\pm}$ for any further point accessed when iterating over $Order_x$ or $Order_y$ in the respective direction. After the third loop, the arrows of Δ_{y+} and Δ_{y-} both exceed the area of the remaining candidates, represented by the dashed lines. This means that all relevant candidates have been considered via $Order_y$, and that the current nearest neighbor is correct.

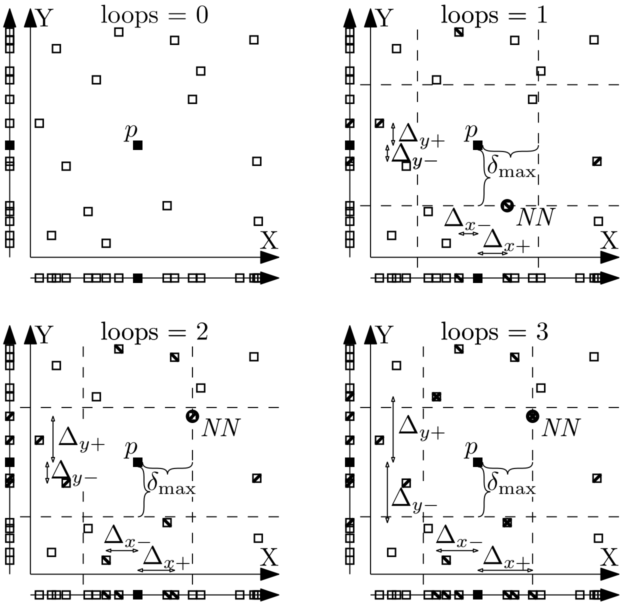


Figure 5: Illustration of Algorithm 4 for each loop.

4.3 Statistical Quality Indicators

Finally we present statistical guarantees for early estimates by IMIE. Since ϱ is a subsample of ρ, statistical tests with μ_ϱ and σ_ϱ^2 yield statistically significant assertions regarding μ_ρ. Equations 7 and 8 give way to analogous assertions for $\widehat{I}(P)$.

Theorem 4.3 ([27]). *Let ρ be a finite population of size n with mean μ_ρ and a variance σ_ρ^2. When drawing an i.i.d. sample ϱ of size m from ρ, the sample mean μ_ϱ has an expected value of $\mathbb{E}(\mu_\varrho) = \mu_\rho$ and a variance of $\sigma_{\mu_\varrho}^2 = \frac{\sigma_\rho^2}{m}\left(\frac{n-m}{n-1}\right)$.*

Proof. See [27]. □

While the classic version of the Central Limit Theorem is not formulated for finite populations, it has been proven that some variations are applicable, and that μ_ϱ is approximately normally distributed [27]. In other words, drawing a sample of size m with a sample mean μ is as likely as drawing μ from $\mathcal{N}(\mu_\rho, \sigma_{\mu_\varrho})$ with $\sigma_{\mu_\varrho} = \sqrt{\sigma_{\mu_\varrho}^2}$. So we can estimate the probability that a sample mean μ_ϱ is off by more than a specified value $\epsilon > 0$ by using the cumulative distribution function Φ of the standard normal distribution $\mathcal{N}(0, 1)$. This is illustrated in Figure 6 and is formally described as

$$\Pr[|\mu_\varrho - \mu_\rho| \geq \epsilon] = 2 \cdot \Phi\left(\frac{-\epsilon}{\sigma_{\mu_\varrho}}\right). \tag{9}$$

Algorithm 4: NNSearch(p)

```
$i_x, i_y \leftarrow$ index of $p$ in $Order_x$, $Order_y$, respectively
$\Delta_{x+}, \Delta_{x-}, \Delta_{y+}, \Delta_{y-}, loops \leftarrow 0$
$\delta_{\max} \leftarrow \infty$
$NN \leftarrow \{\}$
while $\min(\Delta_{x-}, \Delta_{x+}) < \delta_{\max} \wedge \min(\Delta_{y-}, \Delta_{y+}) < \delta_{\max}$ do
    $loops \leftarrow loops + 1$
    if $\Delta_{x+} < \delta_{\max}$ then
        $\Delta_{x+} \leftarrow |x_p - x_{Data[Order_x[i_x+loops]]}|$
        UpdateNN($P[Order_x[i_x + loops]]$)
    if $\Delta_{x-} < \delta_{\max}$ then
        $\Delta_{x-} \leftarrow |x_p - x_{Data[Order_x[i_x-loops]]}|$
        UpdateNN($P[Order_x[i_x - loops]]$)
    if $\Delta_{y+} < \delta_{\max}$ then
        $\Delta_{y+} \leftarrow |y_p - y_{Data[Order_y[i_y+loops]]}|$
        UpdateNN($P[Order_y[i_y + loops]]$)
    if $\Delta_{y-} < \delta_{\max}$ then
        $\Delta_{y-} \leftarrow |y_p - y_{Data[Order_y[i_y-loops]]}|$
        UpdateNN($P[Order_y[i_y - loops]]$)
return NN

function UpdateNN($q$)
if $\|p - q\|_\infty < \delta_{\max}$ then
    insert $q$ into $NN$
    if $|NN| > k$ then
        remove $\arg\max_{r \in NN} \|r - p\|_\infty$ from $NN$
    if $|NN| = k$ then
        $\delta_{\max} \leftarrow \max_{r \in NN} \|r - p\|_\infty$
```

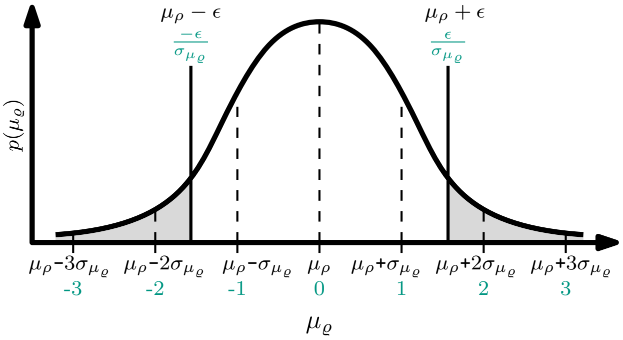


Figure 6: Illustration of the normal distributions $\mathcal{N}(\mu_\rho, \sigma_{\mu_\varrho})$ (upper labels) and $\mathcal{N}(0,1)$ (lower labels).

Alternatively, one can specify a tolerated error probability α and obtain a confidence interval. Let Φ^{-1} be the inverse cumulative distribution function of the standard normal distribution, i.e., $\Phi(\Phi^{-1}(\alpha)) = \alpha$. Then the mean of a sample deviates with probability $1 - \alpha$ by at most $|\Phi^{-1}(\frac{\alpha}{2})| \cdot \sigma_{\mu_\varrho}$ from μ_ρ. This is because both tails of the distributions have to be considered. More formally, it is

$$Pr\left[\mu_\varrho - \left|\Phi^{-1}\left(\frac{\alpha}{2}\right)\right| \sigma_{\mu_\varrho} \leq \mu_\rho \leq \mu_\varrho + \left|\Phi^{-1}\left(\frac{\alpha}{2}\right)\right| \sigma_{\mu_\varrho}\right] \approx 1 - \alpha. \qquad (10)$$

Lastly, there are two more considerations necessary to obtain these statistical guarantees from IMIE. One is that the variance σ_ρ^2, which is used to determine $\sigma_{\mu_\varrho}^2$ in Theorem 4.3, is not known. Using the approximation $\sigma_\rho^2 \approx \sigma_\varrho^2 \frac{m(n-1)}{(m-1)n}$ yields the unbiased approximation $\sigma_{\mu_\varrho}^2 \approx \frac{\sigma_\varrho^2(n-m)}{(m-1)n}$, see [27]. The other point is the *multiple testing problem.* The probabilities for errors only hold for individual tests. But when performing multiple tests to obtain a statistically significant result, the chance of an erroneous result in one test is higher. For instance, this occurs when the response to a statistically insignificant test result is to perform another test, evaluating the result without considering the first, inconclusive result. We illustrate this effect with an example.

Example 4.4. Consider an instance of IMIE that has performed some iterations so far. We use the current mean and var to perform a statistical test whether $\hat{I}(P)$ is above a threshold t. We accept an error chance of 10%. Let us assume that the result of the first test is not significant enough, i.e., the probability is less than 90% based on the current sample. We iterate our estimate a few times and perform a second test, which achieves the desired probability of 90%. However, if $\hat{I}(P)$ is below t, the likelihood that a test reports false certainty based on an unlikely sample increases with each sample. For two tests, the probability of obtaining false certainty is then $Pr[\hat{I}(P) < t] = 1 - (1 - 0.1)^2 = 0.19$.

To account for this problem, we use the correction due to Šidák [29]: To obtain an overall error chance of α, the error chance allowed for the c-th test is $\alpha_{\text{test}} = 1 - (1 - \alpha)^{\frac{1}{c}}$.

To summarize this section, we present the full formula for the c-th statistical test whether $\widehat{I}(P)$ is greater than a threshold t, using variables from IMIE.

$$\Pr[\widehat{I}(P) > t] \approx 1 - \left(1 - \Phi\left(\frac{Offset - Mean - t}{\sqrt{\frac{Var\cdot(|P|-m)}{(m-1)|P|}}}\right)\right)^c \qquad (11)$$

Since we approximate σ_ρ^2, this equation is not exact. On the other hand, the Šidák-correction is very conservative in our case. Namely, when iterating IMIE, the new sample is a superset of the previous sample. This means that the tests based on these samples are dependent, and that the effect of the multiple testing problem is less pronounced. Ultimately, we do not have any formal result to which degree these effects do cancel each other out. In all our experiments in Section 6 however, the error rate never exceeds the bounds established in this section.

5 TIME COMPLEXITY

Now we derive the time complexity of IMIE. First, we do so for our nearest-neighbor search. We then use this result to derive the complexity for initializing and iterating IMIE. Finally, we discuss potential improvements for specific scenarios.

5.1 Nearest-Neighbor Search

We establish the time complexity of Algorithm 4. Each call of UpdateNN(q) takes time in $O(k)$ to compute the (arg) $\max_{r \in NN}$ $\|r - p\|_\infty$. Additionally, let $\mathbb{I}(p)$ be the number of loops performed by NNSearch(p) before terminating. Then the time complexity is in $O(\log n + \mathbb{I}(p) \cdot k)$. Namely, the only other step that is not an elementary assignment is computing the indices of p in $Order_x$ and $Order_y$ with binary search, in $O(\log n)$. However, $\mathbb{I}(p)$ is linear in n for the worst case. Figure 7 shows such a degenerative case, where all points except for p and q are equally distributed among

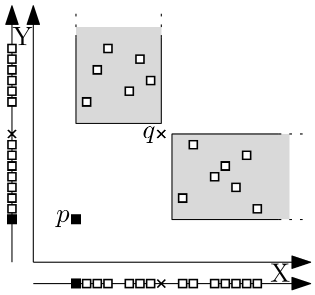


Figure 7: A degenerative case for NNSearch.

the two grey areas. In this case, NNSearch(p) cannot discover the nearest neighbor q via $Order_x$ or $Order_y$ with fewer than $\frac{n-2}{2}$ loops. However, we prove the nontrivial bound $\sum_{p\in P}\mathbb{I}(p) \le (4\cdot\sqrt{n\cdot k}+1)\cdot n$ below.

To prove this bound, we first introduce some additional notation and properties for the several executions of Algorithm 4. For each point $p \in P$, let $V_x(p)$ and $V_y(p)$ be the set of positions of $Order_x$ and $Order_y$, respectively, accessed by NNSearch(p). Additionally, let Pos_x^p be the position of $Order_x$ containing the reference to a point p, i.e., $P[Order_x[Pos_x^p]] = p$. The set of points that access this position during NNSearch(q) is $R_x(p) = \{q \in P : Pos_x^p \in V_x(q)\}$. Pos_y^p and $R_y(p)$ are defined analogously using $Order_y$ instead of $Order_x$. By definition, it is

$$\sum_{p\in P}|V_x(p)| = \sum_{p\in P}|R_x(p)|, \tag{12}$$

as both count the total number of accesses of $Order_x$ across all searches.

Note that NNSearch(q) for points $q \in R_x(p)$ often performs several loops before accessing Pos_x^p. In particular, there are only two points q^+ and q^- such that NNSearch(q^+) and NNSearch(q^-) access Pos_x^p during their first loop. These two points are the points corresponding to the neighboring positions of Pos_x^p, i.e., $q^+/q^- = P[Order_x[Pos_x^p \pm 1]]$. More specifically, for each $c \in \mathbb{N}_0$, there exist at most two points whose positions are exactly c steps away. This is because $Order_x$ is a linear order of a finite set of elements. As a result, $R_x(p)$ defines a lower bound for $\sum_{q\in R_x(p)}\mathbb{I}(q)$. Formally, for each $p \in P$ it is

$$2\cdot\sum_{i=1}^{\frac{R_x(p)-1}{2}} i \le 0+0+1+1+\cdots+\left\lfloor\frac{R_x(p)-1}{2}\right\rfloor \le \sum_{q\in R_x(p)}\mathbb{I}(q). \tag{13}$$

Next, we also consider the properties of $V_y(\cdot)$ and $R_y(\cdot)$. During each loop of a search NNSearch(p), it is $\min(\Delta_{y-},\Delta_{y+}) < \delta_{\max}$. This means that NNSearch(p) accesses at least one new position of $Order_y$ in Line 14 or Line 17. It follows that

$$\mathbb{I}(p) \le |V_y(p)| \tag{14}$$

and with Equation 13, it is

$$2\cdot\sum_{i=1}^{\frac{R_x(p)-1}{2}} i \le \sum_{q\in R_x(p)}|V_y(q)|. \tag{15}$$

Now, we use the fact that NNSearch stops accessing new positions in a certain direction when this direction cannot offer a closer nearest neighbor. In the following lemma, we use this pattern to limit the number of points p where NNSearch(p) accesses certain positions of $Order_x$ and $Order_y$. That is, for each combination of a position of $Order_x$ and $Order_y$, there is only a small number of points whose nearest neighbor search accesses both.

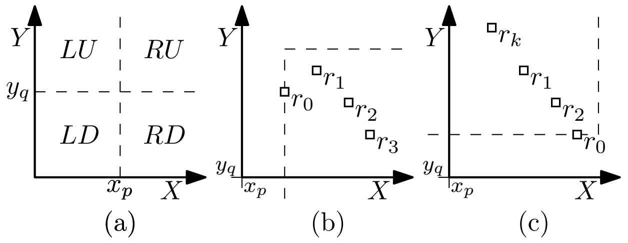


Figure 8: Illustration of arrangements in Claim 5.1. (a) Partitioning of $\mathbb{R}^2$ based on (x_p, y_p). (b),(c) Two cases of layouts of RU.

Lemma 5.1. *For any two points $p, q \in P$, it is $|R_x(p) \cap R_y(q)| \le 4\cdot k$.*

Proof. We consider a partitioning of $\mathbb{R}^2$ into four axis-aligned quadrants RU, RD, LD and LU centered at (x_p, y_q), as illustrated in Figure 8a. To ensure that any point $r \in P \setminus \{p, q\}$ is in exactly one partition, equalities such as $x_r = x_p$ and $y_r = y_q$ are resolved by their ordering in $order_x$ and $order_y$, respectively. For the sake of contradiction, suppose that there are $k+1$ points $\{r_0,\ldots,r_k\} = R_{RU} \subseteq R_x(p) \cap R_y(q)$ in the area RU. We discern between two cases regarding the arrangement of these points.

In the first case, it is $\max_{r,s\in R_{RU}}|x_r - x_s| \ge \max_{r,s\in R_{RU}}|y_r - y_s|$. That is, the largest difference in x-values among points in RU is at least as large as any difference in y-values among RU. For all r in R_{RU}, it is $Pos_x^r > Pos_x^p$. Without loss of generality, let r_0 be the point closest to p and r_k the furthest from p in $Order_x$, respectively. Formally, $r_0 = \arg\min_{r\in R_{RU}} Pos_x^r$ and $r_k = \arg\max_{r\in R_{RU}} Pos_x^r$. This implies that $|x_{r_k} - x_{r_0}| \ge \|r_k - r\|$ for all $r \in R_{RU}$. As illustrated in Figure 8b, NNSearch(r_k) accesses Pos_x^r for all $r \in R_{RU}\setminus\{r_k\}$ before accessing Pos_x^p. After accessing $Pos_x^{r_0}$ and calling UpdateNN(r_0), it holds for the variables in NNSearch(r_k) that $\delta_{\max} = \Delta_{x-}$. The dashed line in Figure 8b illustrates this. This means that NNSearch(r_k) does not access further positions of $Order_x$ in this direction, and thus there is a contradiction to $r_k \in R_x(p)$.

The second case, $\max_{r,s\in R_{RU}}|x_r - x_s| < \max_{r,s\in R_{RU}}|y_r - y_s|$, is symmetric to the first one using $Order_y$ instead of $Order_x$. With r_0 and r_k being the closest and furthest point from q in $Order_y$, NNSearch(r_k) also accesses all positions corresponding to other points in R_{RU} before Pos_y^q. Analogously, it is $\delta_{\max} = \Delta_{y-}$ after calling UpdateNN(r_0), as illustrated in Figure 8c. Thus NNSearch(r_k) does not access the position Pos_y^q, which contradicts $r_k \in R_y(q)$.

As a result there are at most k points from $R_x(p) \cap R_y(q)$ in RU. By symmetry, the same is true for RD, LD, LU. This yields the lemma. □

Combining this lemma with other equations introduced in this section yields the following limit for the total number of iterations performed by all searches.

Lemma 5.2. *For a set $P \subseteq R^2$ of points, the total number of iterations performed by NNSearch(p) for all $p \in P$ is bounded as $\sum_{p\in P}\mathbb{I}(p) \le (4\cdot\sqrt{n\cdot k}+1)\cdot n$.*

Proof. Following Lemma 5.1, each position of $Order_y$ is accessed at most $4 \cdot k$ times by searches accessing one specific position of $Order_x$. More formally, with Equation 15, it is for each $p \in P$

$$\begin{aligned} 4 \cdot k \cdot n \geq \sum_{q \in R_x(p)} |V_y(q)| &\geq 2 \cdot \sum_{i=1}^{\frac{R_x(p)-1}{2}} i \\ &= 2 \cdot \frac{\frac{R_x(p)-1}{2}\left(\frac{R_x(p)-1}{2}+1\right)}{2} \geq \left(\frac{R_x(p)-1}{2}\right)^2 \\ 2\sqrt{k \cdot n} &\geq \frac{R_x(p)-1}{2} \\ 4 \cdot \sqrt{k \cdot n} &\geq R_x(p) - 1 \qquad (16) \end{aligned}$$

Combining Equations 12, 14 and 16 yields

$$\sum_{p \in P} \mathbb{I}(p) \leq \sum_{p \in P} |V_x(p)| = \sum_{p \in P} |R_x(p)| \leq (4 \cdot \sqrt{k \cdot n} + 1) \cdot n. \quad (17)$$

□

Because k is a small constant, the time complexity of performing NNSearch for all points is in $O(n \cdot \sqrt{n})$. So the complexity for each individual search is in amortized $O(\sqrt{n})$.

Theorem 5.3. *NNSearch has an amortized time complexity of* $O(\sqrt{n})$.

5.2 Init and Iterate

We derive the time complexity for initializing and iterating IMIE.

In Init, most operations are assignments, of constant size (Lines 2, 4) or of linear size (Lines 1, 3). The only exception is sorting $Order_x$ and $Order_y$, which is $O(n \log n)$. So the overall runtime of Init is $O(n \log n)$. However, we show in the following section that more efficient variants are possible for scenarios encompassing more than one estimation task. Furthermore, our experiments in Section 6 indicate that the actual runtime of Init often is negligible in comparison to Iterate.

As for the runtime of Iterate, there are only two steps that are not elementary assignments of constant size. One step is computing the marginal counts $MC_k^x(p)$ and $MC_k^y(p)$ (Line 6). It can take place in $O(\log n)$, with binary searches on the sorted arrays as follows. Let i be the smallest integer in $\{0, \ldots, |P| - 1\}$ with $x_{P[Order_x[i]]} \geq x_p - \delta_k^x(p)$. Similarly, let j be the largest integer in $\{0, \ldots, |P| - 1\}$ with $x_{P[Order_x[j]]} \leq x_p + \delta_k^x(p)$. Because $Order_x$ contains all points sorted by x-coordinate, it is $MC_k^x(p) = j - i$. The other marginal count $MC_k^y(p)$ is available analogously, using $Order_y, y_p$ and $\delta_k^y(p)$ instead. The other step is the nearest neighbor search (Line 4), which has an amortized time complexity of $O(\sqrt{n})$ by Theorem 5.3. As a result, Iterate also has an amortized time complexity of $O(\sqrt{n})$. Since P and thus ρ contain n elements, IMIE requires time in $O(n\sqrt{n})$ to reach the final estimate, the one equal to the KSG estimate. This means that IMIE is only slightly slower in reaching the final result than the lower bound $O(n \log n)$ for algorithms without preliminary results [32].

5.3 Scenario-specific Improvements

The initialization procedure presented in Section 4 explains the core concept and properties. Init has been described in a way that is always applicable. However, in many scenarios a user has more than one estimation task based on the same or similar data. Think of estimating the MI for overlapping attribute pairs when searching for strongly dependent attributes. In such a case, it may not be necessary for each instance of IMIE to sort the arrays from scratch, which is the primary computational burden of Init. In this section we present the improvements possible for IMIE in scenarios with high-dimensional data as in Scenario 1 and with streaming data as in Scenario 2. We consider benefits over both the naïve initialization of IMIE as well as the non-iterative estimation.

High Dimensional Data. The number of attribute pairs grows quadratically with the number of attributes. If the data has d attributes, the number of pairs is $\frac{d \cdot (d-1)}{2}$. We now consider using one instance of IMIE for each pair to obtain the pairwise MI estimates. A naïve initialization of these instances would require time in $O(d^2 \cdot n \log n)$. However, we only need to sort the points once per attribute to use the respective sorted arrays for several attribute pairs. This reduces the time complexity for initialization to $O(d \cdot n \log n)$.

Non-iterative estimators for the KSG use two-dimensional space-partitioning trees [19, 31, 32]. This means that each attribute pair requires a different tree, which prohibits a similar improvement. In addition, non-iterative estimators must commit the computation time beforehand. IMIE in contrast can budget computation time between different pairs of attributes, depending on which pairs a user finds interesting, based on the preliminary estimates.

Data streams. With data streams, new data is arriving continuously, and computation time is limited. We consider estimating the current MI using all points whenever new data arrives. This means that most data points remain unchanged. When maintaining up-to-date MI estimates, IMIE can reuse the instance of Data Structure 1 used for the previous estimate instead of another initialization. Considering Data Structure 1, only the adjustment of $Order_x$, $Order_y$ and $Order_R$ does not incur constant costs when a new data point arrives. Adjusting $Order_x$ and $Order_y$ to accommodate new data can take place in $O(\log n)$.[1] Since $Order_R$ is shuffled randomly during the estimation, IMIE can also start off with a (partially) shuffled order and only needs the addition of new indices for new data items. This means that initialization of a new estimator on a data stream can take place in $O(\log n)$ instead of $O(n \log n)$.

Additionally, if the delays between items from the data streams are irregular in length, IMIE automatically offers the best estimate for the time available. Previous work regarding efficient, non-iterative MI estimation on streams [4, 14, 32] imposes a fixed computation time, and there is no easy adjustment if items arrive faster.

Takeaway. While the time complexity of Init may appear prohibitively large in the previous section, we have demonstrated in this section that concrete settings can allow for more efficient solutions. Note that the improvements described are not mutually exclusive. This means that both improvements can be combined when dealing with high-dimensional data in the form of streams. Table 1 summarizes the impact of these techniques on the initialization of IMIE for pairwise MI estimation between d data streams.

[1] From a technical perspective, this time complexity requires $Order_x$ and $Order_y$ to be implemented as binary search trees. For simplicity we keep calling them sorted arrays.

Optimization	Time Complexity
Naïve application	$O(d^2 \cdot n \log n)$
Reuse previous data structure	$O(d^2 \cdot \log n)$
Reuse sorted dimensions	$O(d \cdot n \log n)$
Both	$O(d \cdot \log n)$

Table 1: Impact of optimization techniques for initializing IMIE for pairwise MI of d data streams.

6 EXPERIMENTS

In this section we investigate the performance of IMIE in terms of runtime and estimation quality. We also perform experiments to test the potential benefits from the statistical guarantees and the anytime property of IMIE.

As reference for the performance of IMIE we use the KSG (see Equation 5), because it offers high-quality estimations, and it is the basis of IMIE. To ensure competitive runtime of the KSG we use KD-Trees for its nearest-neighbor search, resulting in the optimal computation complexity of $O(n \log n)$ [32]. As a reference point for faster estimates with lower estimation quality, we use the KSG on subsamples of the data. Since the number of points subsampled can be expressed as a percentile of all points or as an absolute number, we introduce a notation for both. Using a random sample of p% from all data points to compute the KSG is denoted as *KSG%p*. Subsampling exactly q points at random from all data points to compute the KSG on this subsample is denoted as *KSG@q*.

Setup. All approaches and experiments are implemented in C++ and compiled using the Microsoft® C/C++ Optimizing Compiler Version 19.00. We use the non-commercial ALGLIB[2] implementation of KD-Trees for the KSG. We also use the non-commercial ALGLIB[2] implementation of the cumulative density function of the standard normal distribution Φ and its inverse Φ^{-1} when computing our statistical guarantees. All experiments are conducted on Windows 10 using a single core of an Intel® Core™ i5-6300U processor clocked at 2.4 GHz and 20GB RAM.

6.1 Data

In our experiments we use both synthetic and real-world data. As synthetic data, we use dependent distributions with noise used to compare MI estimators, see [15], uniform distributions used to compare MI with the maximal coefficient, see [16], as well as independent uniform and normal distributions. As real data, we use smart meter readings from an industrial plant (HIPE) [3], recorded smart phone sensors to recognize human activities (HAR) [7], and physical quantities for condition monitoring of hydraulic systems (HYDRAULIC) [12]. As proposed by the inventors of the KSG [19], we prevent duplicate points in real-world data by adding noise with minimal intensity. Beginning with real-world data, we now briefly describe the data specifics.

HIPE. This data set, available online[3], contains high-resolution smart meter data from 10 production machines over 3 months. This data has over 2000 attributes total and over 1.5 million data points. We consider a reduced data set containing the first 1000 data points of the machines “PickAndPlaceUNIT”, “ScreenPrinter” and “VacuumPump2” with a grand total of 333 attributes.

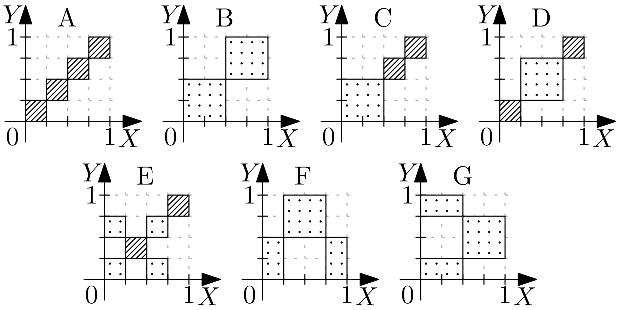


Figure 9: An overview of the uniform distributions used.

HAR. This data set, available at the UCI ML repository[4], features accelerometer and gyroscope sensor readings from smartphones to classify the activity of the human carrying the phone. The data set contains 561 attributes and a total of 5744 data points.

HYDRAULIC. This data set, available at the UCI ML repository[5], features recordings of several physical quantities such as temperature, vibrations and efficiency factors at different sampling rates. For our experiments we use all quantities with a sampling rate of 10 Hz. As a result, each of the 2205 data points has 480 attributes.

For synthetic data, we use the following distributions with known MI values [16, 23]. For distributions with a noise parameter σ_r, we vary σ_r between 0.1 and 1.0.

Linear To construct the point $p_i \in P$, we draw the value x_i from the normal distribution $N(0, 1)$. Additionally, we draw some noise r_i from the normal distribution $N(0, \sigma_r)$, where σ_r is the noise parameter of the distribution. This yields the point $p_i = (x_i, x_i + r_i)$.

Quadratic This distribution is generated analogously to the linear distribution, except that the point is $p_i = (x_i, x_i^2 + r_i)$.

Periodic For each point $p_i \in P$, we draw the value x_i from the uniform distribution $U[-\pi, \pi]$. Additionally, we draw some noise r_i from the normal distribution $N(0, \sigma_r)$, where σ_r is the noise parameter. This yields the point $p_i = (x_i, \sin(x_i) + r_i)$.

Uniform The uniform distributions A to G we use are illustrated in Figure 9. Note that the striped areas contain twice as many points as the dotted areas. For these distributions, each striped area with size $0.25 \cdot 0.25$ contains 25% of all points, while dotted areas of the same size contain 12.5% of all points.

Independent Lastly, we use the distributions U_{IND} and N_{IND}, where each point consists of two values drawn independently from $U[0, 1]$ and $N(0, 1)$, respectively.

6.2 Synthetic Benchmarks

We first evaluate the concrete runtimes of IMIE. While we have established in Section 5 that the time complexity is competitive, actual runtimes may have constant factors that time complexity does not capture. We also look at the estimation quality offered by IMIE after a variable number of iterations. Since the true MI value of real data is unknown, we perform these experiments using

[2] ALGLIB (www.alglib.net)

[3] https://www.energystatusdata.kit.edu/hipe.php

[4] http://archive.ics.uci.edu/ml/datasets/Smartphone+Dataset+for+Human+Activity+Recognition+(HAR)+in+Ambient+Assisted+Living+(AAL)

[5] http://archive.ics.uci.edu/ml/datasets/Condition+monitoring+of+hydraulic+systems

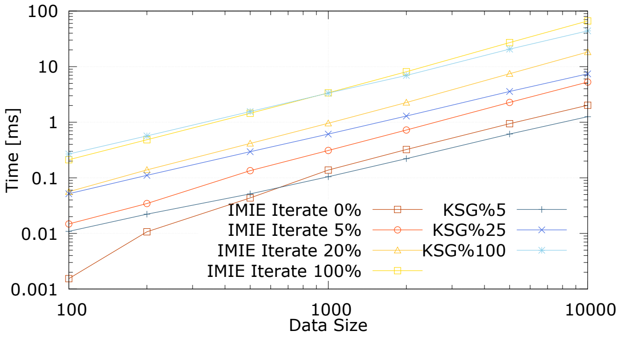


Figure 10: Average runtime depending on the data size for IMIE and subsampling variants.

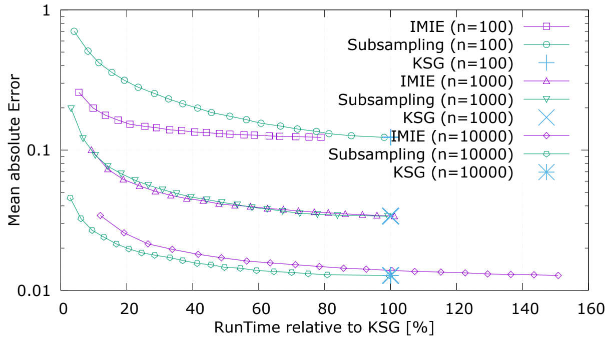


Figure 11: MAE of IMIE and subsampling depending on the runtime relative to KSG for the same data.

synthetic data. Each synthetic data set corresponds to one pair of attributes, for which we produce samples of varying sizes. For each pair, sample size and estimator, we perform 100 estimates and average the runtime and mean absolute error (MAE).

Figure 10 shows the average runtime of IMIE with various numbers of iterations and the KSG with various subsampling settings. Note that the concrete performance of IMIE when iterating until convergence and *KSG%*100 is very similar. This means that computing the exact KSG in the conventional way with a KD-tree and without preliminary results is not faster than using IMIE. Another point to observe is the difference in runtime between IMIE with only the initialization and IMIE that has performed some iterations. Even with only 5% of the iterations, IMIE already consumes more than double the time used for initialization. This shows that the time used for iterations quickly dominates the time required for initialization, even though Init has a high time complexity.

Figure 11 graphs the MAE of subsampling and IMIE depending on the runtime. The plot shows curves per estimator corresponding to a specific sample size, and the time is measured relative to the runtime of the naive KSG estimation for this size. Each point corresponds to the average runtime and absolute error of 100 estimations with the same number of iterations or subsampling size, respectively. In other words, the leftmost point corresponds to subsampling 5% or iterating 5% respectively, while the rightmost point uses all points or iterates until convergence, respectively. The result is that IMIE and KSG with subsampling offer the same time-quality-tradeoff for data of size 1000, with IMIE being somewhat faster for smaller data and somewhat slower for larger data. However, this assumes “optimal” subsampling, in the sense that it is known beforehand which subsampling size is desired. In cases where it is not clear how much time is available or how much time an estimate for a given subsample size takes, this is not given. The time spent finding a good subsampling size is discussed in Section 6.4.

6.3 Statistical Quality Indicators

Next, we investigate the practical relevance of the statistical guarantees. The scenario considered is high-dimensional data. A common information need for high-dimensional data is finding highly dependent attributes. In our experiments we want to know for each of the $\frac{d\cdot(d-1)}{2}$ pairs of attributes whether it is above or below a threshold τ. For IMIE we keep iterating the estimate and perform the test from Equation 11. To be precise, one test is performed for $I(P) > \tau$, and one test is performed for $I(P) < \tau$. To reduce the necessary Šidák-correction for our significance level α_{test} we perform these two tests only every 10 iterations. We start with a minimum sample size of 30 to reduce effects of minimal sample sizes. The exact choice of initial iterations and iterations between tests is arbitrary as long as they are not extreme, e.g., performing statistical tests with sample size one or iterating $\frac{|P|}{4}$ times between tests. Regarding the target significance level, we test different values $\alpha \in \{0.1, 0.05, 0.01, 0\}$. We use fixed percentile subsamples for comparison, i.e., *KSG%5*, *KSG%25* and *KSG%50*.

Figure 12 shows the results for the three real-world data sets with τ varying between 0 and 1. The figure contains two plots per data set. The “Error Rate” shows the number of pairs falsely classified over or under τ as a relative count of all pairs (left axis) and as absolute count (right axis). The “Run Time” shows the total execution time relative to the "naïve" estimation using the KSG (left axis) and as absolute time (right axis). The behavior depending on τ is different per data set. This is because the dependencies in the data are distributed differently. The closer τ is to the actual MI value, the easier it is for an approximate result to be above the threshold while the actual value is below, or vice versa. So it is harder to obtain statistical certainty that the actual value is above or below. To illustrate, the attributes in HIPE are largely independent. This yields MI values close to zero, resulting in high error rates for subsampling approaches and longer execution times for IMIE. Conversely, the attributes of HYDRAULIC are highly dependent. This in turn increases error rates and computation times, for subsampling and IMIE respectively, for higher threshold values.

Nevertheless, there are several common patterns. One is that IMIE does offer better time-quality-tradeoffs than subsampling. I.e., for each subsampling rate there is an α such that IMIE yields fewer errors using less time. A second pattern is that IMIE does adapt to “tough threshold values” by increasing the computation time used. Subsampling in turn makes more false claims. A third interesting pattern is that IMIE with $\alpha = 0$ is almost always faster than the naïve KSG estimation. IMIE can speed up such queries significantly with essentially no risk of error.[6]

6.4 Anytime Experiments

Now we test the performance of IMIE as anytime algorithm. In other words, the available time is not known beforehand. To

[6] Technically there could still be errors due to rounding, numerical evaluation of Φ^{-1} and the approximation in Section 4.3. However, no such error has occurred in any of our experiments.

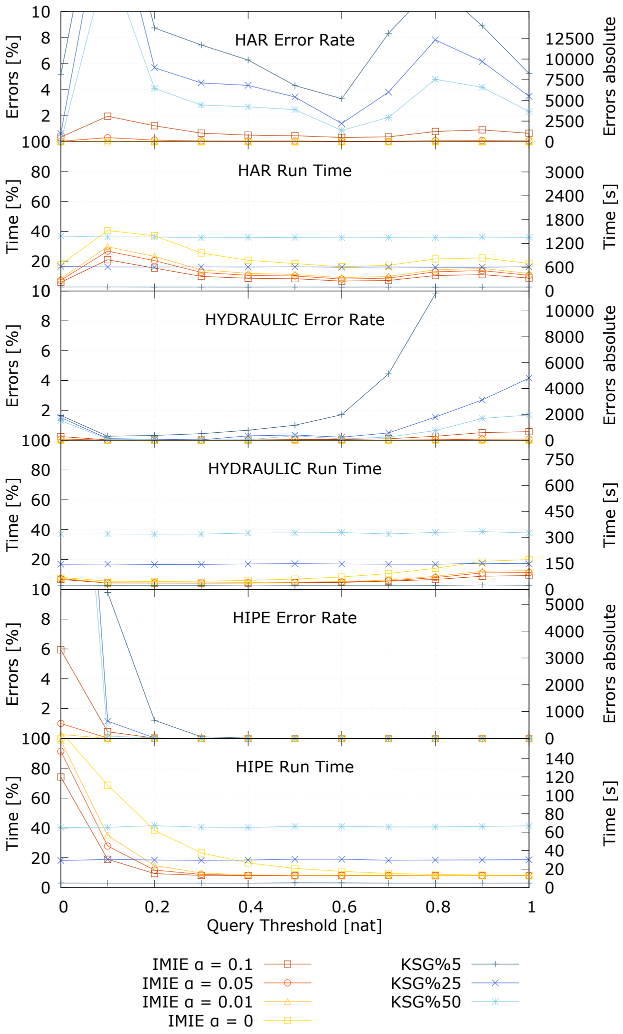


Figure 12: Time and error rate of IMIE and subsampling variants, depending on the chosen threshold τ.

mimic the behaviour of IMIE to improve the estimate with additional time, we also examine two strategies based on subsampling. KSG_{Lin} consecutively computes *KSG%10*, *KSG%20*, . . . , *KSG%100* as long as time is available. We also consider KSG_{Exp}, which computes *KSG@10*, *KSG@20*, *KSG@40*, *KSG@80*, etc. until no time is left.

For this experiment we randomly select 100 pairs of attributes from each real-world data set and estimate MI using IMIE, KSG_{Lin} and KSG_{Exp}. After some time the estimate is interrupted, and the most recent result is used. Since IMIE and subsampling appear most comparable in our synthetic benchmarks for data size $n = 1000$, we use the first 1000 data points of each attribute pair. Given the small scale of time per estimate (cf. Figure 10), we use 1000 estimators in parallel for each pair. One "iteration" then performs the next computation sequentially for each of these estimators.

Figure 13 shows the mean absolute error compared to a KSG estimate using 1000 points as well as the mean standard deviation of estimates for the same attribute pair. Additionally, for each estimate from IMIE we use the statistical quality indicator to determine the distance ϵ. Additionally, the plot displays the average value ϵ such that our preliminary estimate is wrong by at most ϵ with a confidence of 95%. This value is obtained for each estimate using Equation 10. Note that KSG_{Lin} does not consistently produce estimates with time less than 0.3 ms per estimate, and IMIE does not consistently finish the first iteration in 0.1 ms.

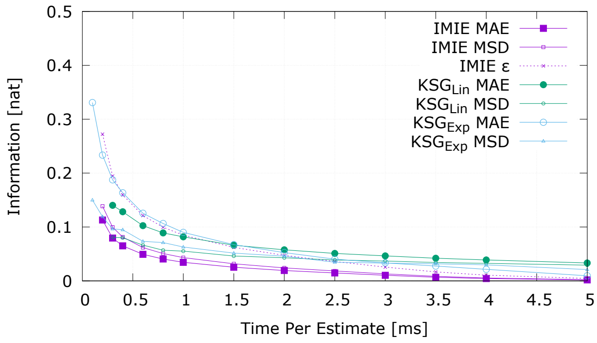


Figure 13: Mean absolute error (MAE) and mean standard deviation (MSD) of anytime approaches as well as the mean ϵ of IMIE.

A result of this experiment is that IMIE has smaller errors on average than the subsampling approaches, even though they are comparable in Figure 11. This is because the subsampling strategies are not efficient for iterative estimation. Estimates from previous iterations are discarded without further benefit, and iteration steps are less granular. This means that only a part of the overall time available is spent on the estimate that is ultimately presented.

6.5 Discussion

To summarize this section, IMIE offers a time-quality tradeoff similar to the one when estimating the KSG with varying subsampling settings. The time necessary for IMIE to converge towards the KSG result is slightly lower for small data and slightly higher for larger data, compared to the naive KSG estimation. But IMIE also offers preliminary results and achieves this time-quality tradeoff even if the time available is not known beforehand. This means that IMIE offers significant benefits for tasks that use these features, such as threshold queries or irregular data-stream processing, without notable drawbacks for regular tasks.

7 CONCLUSIONS

In this work, we have studied the iterative estimation of Mutual Information (MI). The goal has been to provide an estimator that offers a first estimate quickly and improves the estimation with additional time. It should also use the available time efficiently, even if the time available is not known beforehand. To this end, we have proposed IMIE.

By design, IMIE converges towards the same result as the popular MI estimator (KSG) by Kraskov et al. [19] after sufficiently many iterations. Before convergence, the preliminary results of IMIE also offer helpful statistical quality indicators which one can use to infer information regarding the final estimate, i.e., the KSG result. This can take the form of confidence intervals or the probability of surpassing a certain threshold. In addition to these formal results on estimation quality, we also have studied the time complexity of IMIE both in general and when tailored towards specific use cases. One result is that this complexity when computing the exact KSG estimate is only slightly larger than an optimal implementation to compute the KSG that does not offer any preliminary result.

Using extensive experiments, we have evaluated the practical performance of IMIE in terms of concrete runtimes and quality on real data. Among other results, IMIE remains competitive with its estimation quality per time, even when being compared to approaches without preliminary results. The experiments also demonstrate a significant runtime improvement when searching for attribute pairs with high MI in high-dimensional data.

ACKNOWLEDGMENTS

This work was partially supported by the DFG Research Training Group 2153: "Energy Status Data – Informatics Methods for its Collection, Analysis and Exploitation".

REFERENCES

[1] Periklis Andritsos, Renée J Miller, and Panayiotis Tsaparas. 2004. Information-theoretic tools for mining database structure from large data sets. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data.* 731–742.

[2] Ira Assent, Philipp Kranen, Corinna Baldauf, and Thomas Seidl. 2012. Anyout: Anytime outlier detection on streaming data. In *International Conference on Database Systems for Advanced Applications.* 228–242.

[3] Simon Bischof, Holger Trittenbach, Michael Vollmer, Dominik Werle, Thomas Blank, and Klemens Böhm. 2018. HIPE – an Energy-Status-Data Set from Industrial Production. In *Proceedings of ACM e-Energy (e-Energy 2018).* 599–603.

[4] Jonathan Boidol and Andreas Hapfelmeier. 2017. Fast mutual information computation for dependency-monitoring on data streams. In *Proceedings of the Symposium on Applied Computing.* 830–835.

[5] Lei Cao and Elke A Rundensteiner. 2013. High performance stream query processing with correlation-aware partitioning. *Proceedings of the VLDB Endowment* 7, 4 (2013), 265–276.

[6] Thomas M. Cover and Joy A. Thomas. 2006. *Elements of information theory* (2. ed.). 243–256 pages.

[7] Dua Dheeru and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[8] Richard Durstenfeld. 1964. Algorithm 235: Random Permutation. *Commun. ACM* 7, 7 (1964), 420.

[9] Weihao Gao, Sewoong Oh, and Pramod Viswanath. 2017. Demystifying fixed k-nearest neighbor information estimators. In *IEEE International Symposium on Information Theory (ISIT).* 1267–1271.

[10] Lukasz Golab, Howard Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. 2008. On generating near-optimal tableaux for conditional functional dependencies. *Proceedings of the VLDB Endowment* 1, 1 (2008), 376–390.

[11] Eric A Hansen and Shlomo Zilberstein. 2001. Monitoring and control of anytime algorithms: A dynamic programming approach. *Artificial Intelligence* 126, 1-2 (2001), 139–157.

[12] Nikolai Helwig, Eliseo Pignanelli, and Andreas Schütze. 2015. Condition monitoring of a complex hydraulic system using multivariate statistics. In *Instrumentation and Measurement Technology Conference (I2MTC).* 210–215.

[13] Ihab F Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data.* 647–658.

[14] Fabian Keller, Emmanuel Müller, and Klemens Böhm. 2015. Estimating mutual information on data streams. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management (SSDBM'15).*

[15] Shiraj Khan, Sharba Bandyopadhyay, Auroop R. Ganguly, Sunil Saigal, David J. Erickson, Vladimir Protopopescu, and George Ostrouchov. 2007. Relative performance of mutual information estimation methods for quantifying the dependence among short and noisy data. *Phys. Rev. E* 76, 2 (2007), 026209.

[16] Justin B Kinney and Gurinder S Atwal. 2014. Equitability, mutual information, and the maximal information coefficient. *Proceedings of the National Academy of Sciences* 111, 9 (2014), 3354–3359.

[17] LF Kozachenko and Nikolai N Leonenko. 1987. Sample estimate of the entropy of a random vector. *Problemy Peredachi Informatsii* 23, 2 (1987), 9–16.

[18] Philipp Kranen and Thomas Seidl. 2009. Harnessing the strengths of anytime algorithms for constant data streams. *Data Mining and Knowledge Discovery* 19, 2 (2009), 245–260.

[19] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. 2004. Estimating mutual information. *Phys. Rev. E* 69, 6 (2004), 066138.

[20] Sebastian Kruse and Felix Naumann. 2018. Efficient discovery of approximate dependencies. *Proceedings of the VLDB Endowment* 11, 7 (2018), 759–772.

[21] Don O Loftsgaarden and Charles P Quesenberry. 1965. A nonparametric estimate of a multivariate density function. *The Annals of Mathematical Statistics* (1965), 1049–1051.

[22] Son T Mai, Xiao He, Jing Feng, Claudia Plant, and Christian Böhm. 2015. Anytime density-based clustering of complex data. *Knowledge and Information Systems* 45, 2 (2015), 319–355.

[23] Angeliki Papana and Dimitris Kugiumtzis. 2009. Evaluation of mutual information estimators for time series. *International Journal of Bifurcation and Chaos* 19, 12 (2009), 4197–4215.

[24] Hanchuan Peng, Fuhui Long, and Chris Ding. 2005. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27, 8 (2005), 1226–1238.

[25] Josien PW Pluim, JB Antoine Maintz, and Max A Viergever. 2003. Mutual-information-based registration of medical images: a survey. *IEEE Transactions on Medical Imaging* 22, 8 (2003), 986–1004.

[26] Peng Qiu, Andrew J Gentles, and Sylvia K Plevritis. 2009. Fast calculation of pairwise mutual information for gene regulatory network reconstruction. *Computer methods and programs in biomedicine* 94, 2 (2009), 177–180.

[27] John Rice. 2006. *Mathematical statistics and data analysis.* Nelson Education, Chapter Survey Sampling, 199–220.

[28] Claude Elwood Shannon. 1948. A mathematical theory of communication. *The Bell System Technical Journal* 27 (1948), 379–423, 623–656.

[29] Zbyněk Šidák. 1967. Rectangular confidence regions for the means of multivariate normal distributions. *J. Amer. Statist. Assoc.* 62, 318 (1967), 626–633.

[30] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. 2012. Fundamentals of order dependencies. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1220–1231.

[31] Martin Vejmelka and Kateřina Hlaváčková-Schindler. 2007. Mutual information estimation in higher dimensions: A speed-up of a k-nearest neighbor based estimator. In *International Conference on Adaptive and Natural Computing Algorithms (ICANNGA'07).* 790–797.

[32] Michael Vollmer, Ignaz Rutter, and Klemens Böhm. 2018. On Complexity and Efficiency of Mutual Information Estimation on Static and Dynamic Data. In *International Conference on Extending Database Technology (EDBT '18).* 49–60.

[33] Janett Walters-Williams and Yan Li. 2009. Estimation of mutual information: A survey. In *International Conference on Rough Sets and Knowledge Technology (RSKT'08).* 389–396.

[34] BP Welford. 1962. Note on a method for calculating corrected sums of squares and products. *Technometrics* 4, 3 (1962), 419–420.

[35] Ying Yang, Geoff Webb, Kevin Korb, and Kai Ming Ting. 2007. Classifying under computational resource constraints: anytime classification using probabilistic estimators. *Machine Learning* 69, 1 (2007), 35–53.

[36] Shlomo Zilberstein. 1996. Using anytime algorithms in intelligent systems. *AI magazine* 17, 3 (1996), 73–83.

Functional Geometric Monitoring for Distributed Streams

Vasilis Samoladas
ECE, Technical University of Crete.
IMIS, Athena R.C.
vsam@softnet.tuc.gr

Minos Garofalakis
ECE, Technical University of Crete.
IMIS, Athena R.C.
minos@softnet.tuc.gr

ABSTRACT

We introduce *Functional Geometric Monitoring (FGM)*, a substantial theoretical and practical improvement on the core ideas of Geometric Monitoring. Instead of a binary constraint, each site is provided with a non-linear function, which, applied to its local summary vector, projects it to a real number. The sites collectively monitor the sum of these one-dimensional projections and as long as the global sum is subzero, the monitoring bounds are guaranteed. We demonstrate that FGM is as generally applicable as Geometric Monitoring, and provides substantial benefits in terms of performance, scalability, and robustness. In addition, in FGM it is possible to prove worst-case results, under standard monotonicity assumptions on the monitoring problem. In terms of performance, the salient quality of FGM is that it can adapt naturally to adverse changes in the monitored problem, such as lack of monotonicity or very tight monitoring bounds, where no method can deliver asymptotically good performance. We provide formal proofs for many of the properties of FGM, and present an extensive empirical performance evaluation under adverse conditions, on real data.

KEYWORDS

distributed functional monitoring, geometric monitoring, distributed streaming

1 INTRODUCTION

The explosion in the amount of data generated online is entering its next phase, as the Internet of Things (IoT) is set to increase the number of networked data sources by orders of magnitude in the near future. Thus, there is a clear need for ever more scalable techniques for *distributed* stream processing, where the tsunami of data generated by networked nodes is filtered and summarized in near-real time at (or, near) the source, drastically reducing the required communication costs.

Motivated by such needs, there has been significant research effort on the distributed functional monitoring problem, over the past decade. Much effort has concentrated on worst-case communication complexity for particular types of important queries, such as frequency moments, heavy hitters, percentiles, distinct elements etc. Early on, it became apparent that many of these problems can have very bad worst-case performance, unless certain assumptions were made, in particular with respect to monotonicity, about the input streams and/or the monitored functions.

A problem not yet addressed by previous work on monitoring algorithms, is that it can be a challenge to *integrate* them into large stream processing frameworks, such as STORM and Spark, or even more targeted systems such as Gorilla and InfluxDB. Indeed, while such frameworks are a staple of modern information systems, the algorithmic work on distributed functional monitoring has not yet found practical adoption in them. Part of the reason is, we believe, lack of uniformity; each distributed monitoring algorithm imposes different requirements on the system architecture and its communication patterns. Historically, this was also the case with databases, until adoption of the relational model (its limitations notwithstanding) created a vibrant industrial and research ecosphere.

A technique—to our knowledge, the only one—intended to be applicable to *arbitraty continuous queries* is Geometric Monitoring (GM) [27]. A strong appeal of GM is that it separates the complexities of the monitored query operator, from the communication protocol that executes the monitoring. Unfortunately, although it can be very successful at reducing the communcation cost in real-world applications, its performance can degenerate under certain circumstances, such as high stream variability, or skew between the relative rates of local streams. On the theoretical side, the GM is not known to provide any cost guarantees, even under monotonicity assumptions.

Related Work. After the introduction of (centralized) streaming algorithms in the mid '90s, several works proposed *distributed streaming* techniques for particular important problems, such as linear functions [16, 17, 23], top-*k* queries [3, 24], ratio thresholding queries [15], and polynomials of scalar variables [26]. Of particular importance is the problem of tracking sketch synopses [7] of local streams, which can be applied to the approximation of self-join and join aggregates [5, 6].

The first (and, to our knowledge, only) general-purpose technique, Geometric Monitoring (GM), was first proposed in [27, 28]. This paper ignited a rich line of work, part of which related to improving the basic method [14, 18–22], and also utilizing it in important applications (e.g., [4, 11, 12, 25] to name but a few). Interestingly, despite the rich mathematical techniques employed in this body of work, to date there have been no analytical results on the *communication cost* of the method, even under strict assumptions.

Starting with the fundamental results of Cormode et al. [9], the problem of continuous query tracking over distributed streams has also been studied in a theoretical setting in recent years, within the broad framework of communication complexity; the minimum amount of bits that needs to be exchanged between a group of communicating parties, each party observing incrementally a local dataset, so that a global function on the union of the data possessed by all players can be *continuously* tracked with some bounded error. Much of the work has concentrated on the hardness of the problem. It has been shown [1] that the worst-case communication cost of distributed function monitoring can hardly improve upon the baseline method of centralizing all local data, unless restrictive assumptions are made; this is true even for the trivial problem of maintaining a distributed counter. Despite the general negative results, there are also positive results for particular problems of interest, e.g., [1, 9, 29, 30] to name but a few.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.09

In particular, [9] provides optimal results on the communication complexity of monitoring a linear monotone function, while subsequent papers [1, 29] prove strong lower-bounds showing that guaranteeing better than linear worst-case communication costs is probably impossible for complex, non-linear query functions.

Our Contributions. We propose *Functional Geometric Monitoring (FGM)*, a technique which can conceptually be applied to any monitoring problem, in order to perform distributed monitoring with communication costs that are lower (often by orders of magnitude) compared to centralizing all data to a coordinator. The FGM comprises of a distributed algorithm which is independent of the monitoring problem. In order to perform a monitoring task, the FGM must be parameterized by a problem-specific family of functions (termed safe functions later in the paper); to this end, the FGM draws on and can utilize the extensive previous work on distributed monitoring, where such functions have been proposed for a large variety of monitoring problems [11, 13, 21]. Combined with these previous results, FGM is a technique that is ready to be utilized to real distributed monitoring applications.

The strict separation of concerns between distributed systems issues and the monitoring problem, is critically important to anyone wishing to implement distributed monitoring on a general-purpose middleware platform. In addition, despite this strict separation, the FGM offers significant improvements to the communication cost of distributed monitoring, compared to previous monitoring techniques, notably the GM. In particular, FGM has provably better performance than GM, regardless of the monitoring problem. In fact, under FGM it is possible to provide good worst-case guarantees on the communication cost of specific monitoring problems, comparable to the best known theoretical results on distributed functional monitoring. By contrast, no such results are known for GM. In this paper, we provide such worst case analytical results for monitoring frequency norms.

Another issue that has not been treated uniformly—and has often been ignored—by previous techniques, is the detection and response of the monitoring algorithm to circumstances where the monitored constraints (thresholds) are too “tight”; in such occasions, any distributed monitoring algorithm would be unable to do better than to naively centralize all data to a coordinator. Such situations occur frequently in practice, and practical monitoring algorithms should be able to smoothly transition their operation for handling such loads. An important fetaure of FGM is that it can adapt to these high variability situations seamlessly; that is, in a problem-independent manner, and within the logic of the basic protocol.

In addition, FGM’s performance is resilient to the presense of skew in the distribution of data among distributed nodes, as well as in the relative rates of local streams; its performance is fundamentally determined by the characteristics of the global stream (i.e., the union of all distributed streams). Again, this is a novel feature; the performance of previous techniques, notably of GM, is adversely affected in the presense of skew. We have performed extensive experiments that demonstrate and quantify the resilience of FGM both in adverse streaming conditions of high variability, and in the presense of skew.

2 FUNCTIONAL GEOMETRIC MONITORING

The focus of this section is to present the basic principles and protocol of Functional Geometric Monitoring (FGM). Our discussion employs standard notation and terminology from functional analysis: Vectors are denoted by boldface letters, and sets of vectors are added by Minkowski addition:

$$A + B = \{\boldsymbol{x} + \boldsymbol{y} \mid \boldsymbol{x} \in A, \boldsymbol{y} \in B\}.$$

We write $\boldsymbol{x} + A$ instead of $\{\boldsymbol{x}\} + A$; also, $\lambda A = \{\lambda \boldsymbol{x} | \boldsymbol{x} \in A\}$. Finally, in some proofs, we assume some familiarity with the properties of convex functions and sets; in particular, the biconjugate (convex hull of a function), norms and semi-norms, gauge functions and convex cones. The convex hull of A is $\operatorname{conv} A$.

2.1 Background: Approximate Query Monitoring

We adopt the standard data model for distributed data streams. Assume that there are k distributed sites, and that at each site, a local stream is generated or collected, denoted as a (very high-dimensional) vector in vector space $\mathbb{R}^D$. This vector can be the frequency vector of the stream records, or a linear sketch thereof, and changes as stream updates arrive. Let $\boldsymbol{S}_i(t)$, $i = 1 \dots k$ denote the local state vectors. Every site communicates with a coordinator, where users pose queries on the global stream. Without loss of generality, assume that the global stream state is the average of the local stream states, i.e., $\boldsymbol{S}(t) = \frac{1}{k}\sum_{i=1}^{k} \boldsymbol{S}_i(t)$. (Other linear formulas, e.g., sum, can be treated by multiplying local state vectors with scalars as needed.)

We consider two types of queries on this model. In the *one-shot* query, the coordinator needs to monitor for the event $Q(\boldsymbol{S}(t)) \le T$, where Q is a query function and T a threshold. On the other hand, for a *continuous* query, the coordinator needs to maintain at all times a close estimate $Q(\boldsymbol{E}(t))$ of the true value of the query $Q(\boldsymbol{S}(t)$, so that

$$Q(\boldsymbol{S}(t)) \in (1 \pm \varepsilon) Q(\boldsymbol{E}(t)). \tag{1}$$

This guarantee is maintained by the local sites periodically flushing the updates received to their local streams. In particular, the coordinator maintains, for each site i, an estimated state vector $\boldsymbol{E}_i$. When a flush occurs, the site transmits its *drift vector* $\boldsymbol{X}_i(t) = \boldsymbol{S}_i(t) - \boldsymbol{E}_i$, and the coordinator updates $\boldsymbol{E}_i$ by adding $\boldsymbol{X}_i$ to it, while the site resets $\boldsymbol{X}_i$ to 0. Then, the coordinator updates the global estimate $\boldsymbol{E} = \frac{1}{k}\sum_{i=1}^{k} \boldsymbol{E}_i$.

Note that a site can transmit its drift $\boldsymbol{X}_i$ either as a vector of size D, or as a list of the records that arrived since the previous flush (whichever is smaller), therefore the total communication cost for flushing is never worse than tranmitting all data to the coordinator.

In geometric monitoring, the correctness criterion is described as a geometric constraint, of the form $\boldsymbol{S} \in A$, where $A \subseteq \mathbb{R}^D$ is the *admissible region*, that is, the set of global stream states where the constraint holds. That is,

$$A = \left\{ \boldsymbol{x} \in \mathbb{R}^D \mid Q(\boldsymbol{x}) \le T \right\} \qquad \text{one-shot queries} \tag{2}$$

$$A = \left\{ \boldsymbol{x} \in \mathbb{R}^D \mid Q(\boldsymbol{x}) \in (1 \pm \varepsilon) Q(\boldsymbol{E}) \right\} \qquad \text{continuous queries} \tag{3}$$

2.2 Communication costs

We assume that each message consists of a sequence of *words*, of sufficient size. In particular, we assume that each word can store a real number; all our protocols are robust against finite precision, so this is not an unrealistic assumption.

We distinguish two directions in communication. *Downstream communication* consists of messages from local nodes to the coordinator, while *upstream communication* consists of messages from the coordinator to local nodes. We do not consider a multicast capability, although our work can be adapted to such settings.

2.3 Safe functions

Our starting point relates to the representation of a safe *configuration*, in a monitoring algorithm. The *configuration* of the system of k sites is a is a (kD)-dimensional vector consisting of the *concatenation* of the k local drift vectors, $\boldsymbol{X}_i$. The system is in a safe state as long as $\boldsymbol{E} + \frac{\sum_{i=1}^{k} \boldsymbol{X}_i}{k} = \boldsymbol{S} \in A$.

To guarantee that a configuration is safe, FGM employs a real function $\phi : \mathbb{R}^D \to \mathbb{R}$, depending on A, $\boldsymbol{E}$ and k. Each site tracks its ϕ-value, $\phi(\boldsymbol{X}_i)$, as $\boldsymbol{X}_i$ is updated. System safety is guaranteed by tracking the sign of the *sum* $\psi = \sum_{i=1}^{k} \phi(\boldsymbol{X}_i)$. In particular, we need to guarantee that $\psi \leq 0$ implies $\boldsymbol{S} \in A$.

Definition 2.1 (($A, \boldsymbol{E}, k$)-safe function). A function $\phi : \mathbb{R}^D \to \mathbb{R}$ is *safe* for admissible region $A \subseteq \mathbb{R}^D$, vector $\boldsymbol{E} \in \mathbb{R}^D$, and $k \geq 1$, if, $\phi(\boldsymbol{0}) < 0$ and, for all $\boldsymbol{X}_i \in \mathbb{R}^D$, $i = 1, \ldots, k$,

$$\sum_{i=1}^{k} \phi(\boldsymbol{X}_i) \leq 0 \implies \boldsymbol{E} + \frac{\sum_{i=1}^{k} \boldsymbol{X}_i}{k} \in A.$$

Much of previous work on distributed monitoring has proposed safe functions for specific problems. Since we indend to explore the FGM in terms of its generality, we are interested in properties of safe functions as a class.

First, note that safety is preserved under common pointwise operations: positive scaling, addition and pointwise supremum. Consequently, $(A, \boldsymbol{E}, k)$ safety is monotone under pointwise dominance, i.e., if ϕ is $(A, \boldsymbol{E}, k)$-safe and $\forall \boldsymbol{x}, \phi(\boldsymbol{x}) \leq \phi'(\boldsymbol{x})$, then ϕ' is also $(A, \boldsymbol{E}, k)$-safe.

In fact, more can be said about addition and pointwise supremum of safe functions; they can be employed to *compose* safe functions for an admissible region A, defined by a set-algebraic expression over some family of admissible regions $\{A_i\}$, when a safe function ϕ_i for each A_i is available.

Theorem 2.2. *Let ϕ_i be $(A_i, \boldsymbol{E}, k)$-safe, for each $i \in I$ respectively. Then,*

- $\sup_{i \in I} \phi_i$ *is* $(\bigcap_{i \in I} A_i, \boldsymbol{E}, k)$*-safe, and*
- $\sum_{i \in I} \phi_i$ *is* $(\bigcup_{i \in I} A_i, \boldsymbol{E}, k)$*-safe, provided that I is finite.*

On the dependence on k, note that, for any divisor k' of k, a k-safe function is also k'-safe , and 1-safe in particular. Therefore a necessary condition for k-safety is that the 0-sublevel $L(\phi) = \{\boldsymbol{x} | \phi(\boldsymbol{x}) \leq 0\}$, shifted by $\boldsymbol{E}$, be a subset of A:

$$\boldsymbol{E} + L(\phi) \subseteq A.$$

2.3.1 Safe functions and convexity. Intuitively, we can "improve" a safe function ϕ by finding a function $\phi' \leq \phi$ that is still safe; ϕ' is improved in the sense that the set of configurations where $\psi \leq 0$ is larger for ϕ' than for ϕ.

In this respect, of particular interest are functions that are safe for all k (and fixed A, $\boldsymbol{E}$). We denote such functions as $(A, \boldsymbol{E})$-safe. The salient property of $(A, \boldsymbol{E})$-safe functions is that they can always be "improved" into dominated, *convex* safe functions.

Theorem 2.3. *If a function ϕ is $(A, \boldsymbol{E})$-safe, there exists a* ***convex*** *function $\zeta \leq \phi$ which is also $(A, \boldsymbol{E})$-safe.*

Proof. Ommitted due to space constraints. □

Convex safe functions are appealing because they admit a very simple criterion for $(A, \boldsymbol{E})$-safety.

Lemma 2.4. *A convex function ζ with $\zeta(\boldsymbol{0}) < 0$ is $(A, \boldsymbol{E})$-safe, if and only if, $\boldsymbol{E} + L(\zeta) \subseteq A$.*

Proof. We have already seen that $\boldsymbol{E}+L(\zeta) \subseteq A$ is necessary for safety. Sufficiency follows directly from the convexity of ζ. □

2.3.2 Quality of safe functions. Consider the set $C \subseteq \mathbb{R}^{kD}$ of safe configurations of a monitoring problem

$$C = \left\{(\boldsymbol{X}_1, \ldots, \boldsymbol{X}_k) \mid \boldsymbol{E} + \frac{1}{k}\sum_{i=1}^{k} \boldsymbol{X}_i \in A\right\} \tag{4}$$

The FGM protocol under-approximates this set by the set of configurations, where $\psi \leq 0$. Call this the *quiescent region* $\boldsymbol{Q}_\phi$, which is determined by the choice of safe function ϕ. We would like to characterize ϕ so that $\boldsymbol{Q}_\phi$ is as large (inclusion-wise) as possible, in order to improve the approximation of C. Such a characterization is possible if we restrict our attention to $(A, \boldsymbol{E})$-safe (and by virtue of the above theorem, convex) functions.

Intuitively, the issue that we examine can be presented with an example: assume that $A = \{\boldsymbol{x} | \|\boldsymbol{x}\| \leq 1\}$ is the unit ball, and take $\boldsymbol{E} = \boldsymbol{0}$. It is easy to see that both convex functions $\|\boldsymbol{x}\| - 1$ and $\|\boldsymbol{x}\|^2 - 1$ are suitable safe functions. However, the former choice is superior to the latter, when the size of the quiescent region is taken into account. To see this, note that $(1/2)(\|\boldsymbol{x}\|^2 - 1)$ strictly dominates $\|\boldsymbol{x}\| - 1$; therefore, a configuration say, $(\boldsymbol{0}, \boldsymbol{p})$ with $\|\boldsymbol{p}\| = \sqrt{3}$ is quiescent for $\|\boldsymbol{x}\| - 1$ but not for $\|\boldsymbol{x}\|^2 - 1$.

It turns out that safe functions that are best, are those that are *level-minimal*, that is, they do not strictly dominate any function with equal level set.

Theorem 2.5. *A $(A, \boldsymbol{E})$-safe function ϕ has maximal quiescent region, among all $(A, \boldsymbol{E})$-safe functions, for every k, iff,*

- *ϕ is convex*
- *$L(\phi)$ is a maximal convex subset of A, and*
- *ϕ is level-minimal*

Proof. Ommitted due to space constraints. □

The above results highlight the centrality of convexity in the monitoring problem; starting from very broad principles, we have shown that convexity enters in a natural way from the definition of safety, and furthermore, we have formally identified the requirement of level minimality, in order to maximize the quiescent region in FGM.

We will return to the issue of safe functions, with respect to the communication costs they entail, after we present the FGM distributed protocol.

2.4 The basic FGM protocol

The FGM protocol works in rounds. Monitoring the threshold condition

$$\sum_{i=1}^{k} \phi(\boldsymbol{X}_i) \leq 0, \tag{5}$$

over the duration of the round is performed along the lines of the algorithm in [9].

At the beginning of a round, the coordinator knows the current state of the system $\boldsymbol{E} = \boldsymbol{S}$. It selects an $(A, \boldsymbol{E}, k)$-safe function ϕ. At each point in time, let $\psi = \sum_{i=1}^{k} \phi(\boldsymbol{X}_i)$. The round's steps are:

(1) At the beginning of a round, the coordinator ships ϕ to every site (it is sufficient to ship vector $\boldsymbol{E}$, since A can then be determined from it). Local sites initialize their drift vectors to $\boldsymbol{0}$. With these settings, initially it is $\psi = k\phi(\boldsymbol{0})$.

(2) Then, the coordinator initiates a number of subrounds, to be described below. At the end of all subrounds, $\psi > \epsilon_\psi k\phi(\mathbf{0})$, for some small ϵ_ψ (Note that ϵ_ψ is not related to the desired accuracy for the monitored query, ε, but only to the desired quantization for monitoring ψ. We have used $\epsilon_\psi = 0.01$ in our experiments).
(3) Finally, the coordinator ends the round by collecting all drift vectors and updating $\boldsymbol{E}$.

2.4.1 Execution of subrounds. The goal of each subround is to monitor the condition $\psi \leq 0$ coarsely, with a precision of roughly θ, performing as little communication as possible. Subrounds are executed as follows:

(1) At the beginning of a subround, the coordinator knows the value of ψ. It computes the subround's *quantum* $\theta = -\psi/(2k)$, and ships θ to each local site. Also, the coordinator initializes a counter $c = 0$. Each local site records its initial value $z_i = \phi(\boldsymbol{X_i})$, where $2k\theta = -\sum_{i=0}^{k} z_i$. Also, each local site initializes a counter $c_i = 0$.
(2) Each local site i maintains its local drift vector $\boldsymbol{X}_i$, as it processes stream updates. When $\boldsymbol{X_i}$ is updated, site i updates its counter

$$c_i := \max\{c_i, \lfloor \frac{\phi(\boldsymbol{X}_i) - z_i}{\theta} \rfloor\}.$$

If this update increases the counter, the local site sends a message to the coordinator, with the increase to c_i.
(3) When the coordinator receives a message with a counter increment from some site, it adds the increment to its global counter c. If the global counter c exceeds k, the coordinator finishes the subround by collecting all $\phi(\boldsymbol{X}_i)$ from all local sites, recomputing ψ. If $\psi \geq \epsilon_\psi k\phi(\mathbf{0})$, the subrounds end, else another subround begins.

The following simple statement guarantees the correctness of the protocol.

Proposition 2.6. *During the execution of a subround, if $c \leq k$ then $\sum_{i=1}^{k} \phi(\boldsymbol{X}_i) < 0$.*

Proof. At each point in time and for any site, it must be

$$\frac{\phi(\boldsymbol{X}_i) - z_i}{\theta} - 1 < \lfloor \frac{\phi(\boldsymbol{X}_i) - z_i}{\theta} \rfloor \leq c_i.$$

Summing both sides, we get $\frac{1}{\theta}\left(\psi + 2k\theta\right) - k < c$, which simplifies to $\sum_{i=1}^{k} \phi(\boldsymbol{X}_i) < (c-k)\theta \leq 0$. □

2.5 Performance analysis

Apart from the communication incurred during the subrounds of the FGM protocol, the communication cost of a round consists of two parts; an upstream cost $\Theta(kD)$ for shipping $\boldsymbol{E}$ to all sites at the beginning of a round, and a downstream cost $O(\min\{kD, \tau\})$, for shipping drift vectors to the coordinator at the end of the round. Here, τ stands for the total number of stream updates processed by all sites during a round—as mentioned before, sites that received few stream updates during a round, can ship them verbatim to the coordinator.

2.5.1 The cost of subrounds. Each subround itself costs only $3k + 1$ one-word messages: k messages to broadcast quantum θ, k messages to collect ζ-values at the end of a subround, and up to $k + 1$ downstream messages carrying counter updates.

The problem of monitoring $\psi \leq 0$ is of course an instance of the non-monotone distributed counting problem. As shown in [9], if ψ is an increasing function of time, then the number of subrounds is at most $\log_2 \frac{1}{\epsilon_\psi}$.

In general, there is no guarantee that ψ will be increasing; therefore we also provide an analysis within the framework of *variability*, as set out in [10]. In this framework, the cost of tracking a non-monotone counter $f(t)$ within accuracy ε is shown to take $O(\frac{k}{\varepsilon}V_f)$ messages, where $V_f(t) = \sum_{\tau=0}^{t} \min\{1, \frac{|\delta f(t)|}{|f(t)|}\}$. They provide space-plus-time lower bounds for the tracking problem that match the communication cost.

In our setting, we are not interested in *tracking* the value of ψ. Still, the definition of variability during a round can be given as follows: let the sequence ψ_n represent the value of ψ at the end of the n-th subround. Also, define the set of values that $\phi(\boldsymbol{X}_i)$ takes, during subround n, as

$$\Phi_{i,n} = \{\phi(\boldsymbol{X}_i(t)) \mid \forall t \text{ during subround } n\}$$

Then the change $\Delta\psi_n$ is

$$\Delta\psi_n = \sum_{i=1}^{k} \sup \Phi_{i,n} - \inf \Phi_{i,n}$$

Finally, the ψ-variability over a round with q subrounds is

$$V = \sum_{n=1}^{q} \frac{|\Delta\psi_n|}{|\psi_n|}.$$

Theorem 2.7. *The communication cost of all subrounds of a round is $O(kV)$ words.*

Proof. Consider the n-th subround, with quantum $\theta = -\psi_{n-1}/2k$. If the counter of site i is c_i, then, at some point during this subround we had

$$\lfloor \frac{\phi(\boldsymbol{X}_i) - z_i}{\theta} \rfloor = c_i.$$

We conclude that, since $c > k$,

$$\Delta\psi_n \geq (k+1)\theta = \frac{k+1}{2k}|\psi_{n-1}| \geq \frac{|\psi_{n-1}|}{2}.$$

On the other hand, $|\psi_n - \psi_{n-1}| \leq \Delta\psi_n$, thus the variability has increased by at least $1/3$ during this subround. Therefore, the total cost of all subrounds is at most $(9k + 3)V$ words. □

At this point, we should report that in our extensive experiments with complex non-monotone functions over sketches and streams created from real data with deletions (using windows), the number of subrounds per round q was always at most 10, and almost always $7 \approx \log_2 \frac{1}{0.01}$. In fact, the total cost $O(kq)$ of all subrounds in a round, was dominated by several orders of magnitude, by the upstream cost $\Theta(kD)$. This good fortune is possibly due to tight monitoring bounds in our experiments, but still, it is an interesting observation, considering how bad the worst-case costs are.

An interesting open question is to relate ψ-variability to the variability of the query function Q, and in particular derive lower bounds based on this concept.

We should also discuss the role of ϵ_ψ. Note that this bound is unrelated to the approximation bound ε of the monitored query Q. It is simply a threshold of accuracy, with which the value of ψ is approximated. In practice, a fixed value of 0.01 seemed to suffice. The choice of ϵ_ψ can be understood as the precision to which $\phi(\boldsymbol{X}_i)$ are evaluated; their values are quantized to $\epsilon_\psi\phi(\mathbf{0})$ absolute error. Selecting a different value depends on the geometry of a particular problem; we omit the details.

2.5.2 Comparison to classic geometric monitoring. Both the FGM protocol and the standard protocol of geometric monitoring (GM) are generally applicable. The two protocols can be rendered comparable; when FGM is used together with convex safe functions, the condition $\phi(\boldsymbol{x}) \leq 0$ is akin to testing membersip in a convex Safe Zone [22]. However, the GM protocol adopts a much stricter safeness condition, equivalent to

$$\max_{i=1,\ldots,k} \phi(\boldsymbol{X}_i) < 0. \tag{6}$$

When the above condition is violated, the GM protocol performs substantial communication (either a partial, or a full synchronization, by flushing the local state vectors). By contrast, the FGM is much more patient; in fact, it is easy to see that, if the two protocols start from the same estimate $\boldsymbol{E}$ at the beginning of a round, as long as safeness condition (6) holds, the first subround of FGM has not yet finished.

Proof. As long as $\phi(\boldsymbol{X}_i) < 0$, it is $1 - \phi(\boldsymbol{X}_i)/\phi(\boldsymbol{0}) < 1$. The quantum of the first FGM subround is $\theta = -\phi(\boldsymbol{0})/2$, therefore, for each site i, it is

$$\lfloor \frac{\phi(\boldsymbol{X}_i) - \phi(\boldsymbol{0})}{-\phi(\boldsymbol{0})/2} \rfloor = \lfloor 2(1 - \frac{\phi(\boldsymbol{X}_i)}{\phi(\boldsymbol{0})}) \rfloor \leq 1,$$

and thus the coordinator has received at most k bits. □

The advantage of FGM over GM becomes more apparent by considering the size of the quiescent regions for these protocols. Each protocol will synchronize (flush local sites) as soon as the system escapes the quiescent region. It is therefore advantageous to admit a quiescent region that better approximates the set of safe configurations C.

Fig. 1 depicts the situation for $D = 1$ and $k = 2$. Without loss of generality, we are assuming $A = [-1, 1]$. The quiescent region for GM is simply $A \times A$, whereas for FGM the region depends on the choice of ϕ. Choosing $\phi(x) = |x|^p - 1$ will be correct for every $p \geq 1$, but naturally the best function is the level-maximal function $|x| - 1$ (i.e., $p = 1$). In fact, it can be seen in Fig. 1 that as p grows, the benefit of FGM over GM decreases; however, FGM will never be inferior to GM.

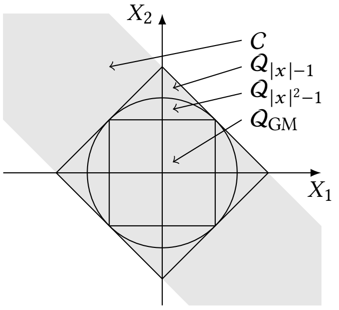


Figure 1: Configuration space for $A = [-1, 1] \subseteq \mathbb{R}$ and $k = 2$, depicting the set of safe configurations C, the FGM quiescent regions $Q_{|x|^p-1}$ for $p = 1, 2$ and the GM quiescent region $Q_{\mathbf{GM}}$.

3 COMPLEXITY RESULTS FOR F_p MOMENTS

We now turn to an analysis of the worst-case communication cost of FGM for F_p moments. In this analysis, it is necessary to introduce monotonicity assumptions about the studied problems; otherwise, even the simplest problems can have very bad worst case complexity. In particular, we assume that all local state vectors and drifts are frequency vectors with nonnegative coefficients; this assumption is compatible with an insert-only stream.

Similar complexity results were obtained by [9], by an algorithm which is similar to FGM; under monotonicity, each round of their algorithm is essentially a round similar to ours, but with carefully selected thresholds. Here, we will strive for a simpler approach.

Since the functions to be monitored are convex, and in fact the $F_p(\boldsymbol{x})$ moment of a frequency vector $\boldsymbol{x}$ is just the norm $\|\boldsymbol{x}\|_p^p$, we use safe functions of the form $\|\boldsymbol{x}+\boldsymbol{E}\|_p - T$. Note that selecting not to raise the norm to p yields better quiescent regions, although it does not make much difference to the asymptotic results under monotonicity.

We begin by examining the effect of a single FGM round. In such a round, we start from some global state $\boldsymbol{E}$ and we allow the protocol to proceed to termination, under an admissible region of the form $\{\boldsymbol{x} | \|\boldsymbol{x}\|_p \leq T\}$, where T depends on whether we are interested in an one-shot query or a continuous one.

Lemma 3.1. *Assume the problem of monitoring admissible region $A = \{\boldsymbol{x} | \|\boldsymbol{x}\|_p \leq T\}$ for $p \geq 1$, starting at some $\boldsymbol{E} \in A$.*

Under monotonicity assumptions, at the end of a single FGM round with safe function $\phi = \|\boldsymbol{x} + \boldsymbol{E}\|_p - T$, the final stream state $\boldsymbol{S}$ will have

$$\|\boldsymbol{S}\|_p^p \geq (1 - \frac{1}{k^{p-1}})\|\boldsymbol{E}\|_p^p + \frac{1}{k^{p-1}}\tilde{T}^p,$$

where $\tilde{T} = T(1 - \epsilon_\psi) + \epsilon_\psi \|\boldsymbol{E}\|_p \approx T$.

Proof. At the end of the round, the value of ψ has become greater than $\epsilon_\psi k \phi(0)$. Therefore, under our safe function, at the end of the round, the coordinator collected drift vectors $\boldsymbol{X}_i$, with

$$\tilde{T} \leq \frac{1}{k} \sum_{i=1}^{k} \|\boldsymbol{X}_i + \boldsymbol{E}\|_p.$$

From Hölder's inequality (thinking of the sum as an inner product with vector $\mathbf{1} = (1, 1, \ldots, 1)$), we get

$$\sum_{i=1}^{k} \|\boldsymbol{X}_i + \boldsymbol{E}\|_p \leq \|\mathbf{1}\|_q \Big(\sum_{i=1}^{k} \|\boldsymbol{X}_i + \boldsymbol{E}\|_p^p \Big)^{1/p},$$

where $q = p/(p-1)$ is the Hölder conjugate of p. Combining the above inequalities by raising to p, and noting that $\|\mathbf{1}\|_q^p = k^{p-1}$, we get

$$\sum_{i=1}^{k} \|\boldsymbol{X}_i + \boldsymbol{E}\|_p^p \geq k\tilde{T}^p. \tag{7}$$

To proceed, first observe that $k\boldsymbol{S} = k\boldsymbol{E} + \sum_{i=1}^{k} \boldsymbol{X}_i$. Consider the following real inequality (over nonnegative numbers):

$$(ke + \sum_{i=1}^{k} x_i)^p = (\sum_{i=1}^{k} (x_i + e))^p \geq \sum_{i=1}^{k} (x_i + e)^p + (k^p - k)e^p.$$

When applied to each coordinate of $\boldsymbol{E}$ and $\boldsymbol{X}_i$, it follows from 7 that

$$k^p \|\boldsymbol{S}\|_p^p \geq k\tilde{T}^p + (k^p - k)\|\boldsymbol{E}\|_p^p.$$

Dividing both sides with k^p finishes the proof. □

Observe from the above lemma that the effect of ϵ_ψ is localized in reducing slightly the threshold T in each round; however, as $\|\boldsymbol{E}\|_p \rightarrow T$, the effect dissappears; this case makes apparent that ϵ_ψ does not affect the accuracy of the monitoring; it only increases (slightly) the number of rounds. We do not discuss ϵ_ψ any further.

We can consider two scenaria for monitoring the F_p norm.

3.0.1 One-shot query. In this scenario, which is exactly the framework of ditributed functional monitoring, we set an initial T and we estimate the number of rounds needed until $\|\boldsymbol{S}\|_p$ exceeds $(1-\varepsilon)T$, starting at $\boldsymbol{E} = \boldsymbol{0}$. By solving a simple linear recurrence, we get the following

THEOREM 3.2. *For safe function $\|\boldsymbol{x}\|_p - T$, the FGM protocol can monitor the F_p moment of a monotone stream in $O(k^{p-1}\log\frac{1}{\varepsilon})$ rounds.*

For the actual communication cost, when using sketches or other summaries for F_p norms, we refer to the discussion of [9].

3.0.2 Continuous query. In this case, the threshold T given to the FGM protocol at each round is set to $(1+\varepsilon)\|\boldsymbol{E}\|_p$, that is, it changes with each round. To describe the communication cost in the continuous setting, we express the number n of rounds as a function of a starting query value Q_0 and an ending query value Q_n. Ommitting the easy details, we have

THEOREM 3.3. *For safe function $\|\boldsymbol{x}\|_p - T$, the FGM protocol can monitor continuously the F_p moment of a monotone stream, as it transitions from value O_0 to Q_n in $O(\frac{k^{p-1}}{\varepsilon}\log\frac{Q_n}{Q_0})$ rounds.*

3.0.3 Discussion. The above complexity results match those of the original paper by Cormode et al. [9]. Our proofs show that they can be obtained without resorting to a special-purpose protocol like the one proposed in [9]. By contrast, the FGM protocol is built along the lines of greedy relaxation: setting a safe zone and letting the protocol iterate to completion. Naturally, there is nothing to forbid a clever coordinator algorithm to set more precise targets, in order to achieve better results.

The real limitation of FGM comes from the fact that in FGM, local nodes are memoryless; once a local drift is transmitted, the node has no memory of its past state. It should not come as a surpsise that better communication complexity can be achieved with stateful nodes. In particular, the results of [29] on very good upper bounds for frequency moments require nodes to retain much information for a long time. On the other hand, this may be quite undesirable from an implementation point of view. Thus, such results are important in the context of communication complexity with unrestricted parties, but arguably not immediately practical.

Another point compares the implementation cost of algorithms; arguably, the algorithms presented in [9] and elsewhere, are harder to adapt to more general setting. To illustrate the point, consider the problem of monitoring, say, the F_2 moment, in a stream allowing deletions as well as insertions. With FGM, it suffices to augment the safe function: starting at state $\boldsymbol{E}$, a good safe function for admissible region $A = \{\boldsymbol{x} | \|\boldsymbol{x}\|_2 \geq (1-\varepsilon)\|\boldsymbol{E}\|\}$ is defined as a half space, tangent to ball A at the projection of $\boldsymbol{E}$ onto A. Then, the safe functions for upper and lower bound of the F_2 moment can be combined via the pointwise-max operation:

$$\phi(\boldsymbol{x}) = \max\{-\varepsilon\|\boldsymbol{E}\| - \boldsymbol{x}\frac{\boldsymbol{E}}{\|\boldsymbol{E}\|}, \|\boldsymbol{x}+\boldsymbol{E}\| - (1+\varepsilon)\|\boldsymbol{E}\|\}.$$

If the above function is employed in an insertion-only setting, it will retain the cost guarantees proved above.

4 ROBUSTNESS UNDER ADVERSE CONDITIONS

We now turn our attention to features of the FGM that allow it to handle gracefully adverse streaming conditions. These conditions can arise from a number of factors, such as:

- Setting the monitoring accuracy ε to a very low value, resulting in tight thresholds for monitoring.
- In the absense of monotonicity, handling local streams which tend to cancel each other (this is a multidimensional version of the problem of non-monotone counter).
- Handling cases where the local stream rates are very uneven (e.g., following a power-law distribution).

To handle the above situations, the FGM protocol offers a number of enhancements; the effect of these enhancements is quite apparent in our experiments, but to our knowledge they do not provide asymptotic improvements in communication cost.

The guiding intuition in the following is the observation that, under a greedy view, it is preferable to have FGM rounds last longer (consuming more stream updates), since then, not only the streams are better summarized in local state vectors, but also, the upstream overhead of shipping $\boldsymbol{E}$ to local sites at the the beginning of a round is paid less often.

4.1 Rebalancing

Our starting point is the observation that $\sum_{i=1}^{k}\phi(\boldsymbol{X}_i) > 0$ does not generally imply that $\phi(\frac{1}{k}\sum_{i=1}^{k}\boldsymbol{X}_i)$ is also positive, or even much different than $\phi(\boldsymbol{0})$, i.e., often, at the end of a round, the global stream state $\boldsymbol{S}$ has not moved significantly far from $\boldsymbol{E}$. Therefore, the current safe function ϕ may still be quite useful, and we would like to avoid the overhead of shipping a new safe function to the sites.

Rebalancing is an important technique in classic Geometric Monitoring. The idea in GM is to flush a subset of the local sites, and then ship them the average of their previous drifts. A straightforward adaptation of the rebalancing method of GM, could benefit FGM. Unfortunately, the method is highly uncertain as to the benefit it provides, versus the added upstream communication overhead (which is a multiple of $O(D)$).

A simple approach to rebalancing, that incurs negligibly small additional upstream communication cost, is to ship to the sites a scaling factor, with which to scale their local drifts. We restrict the discussion to convex safe functions.

In our rebalancing scheme, the coordinator holds an extra state vector, the *balance vector* $\boldsymbol{B}$, which is used to aggregate drift vectors from local sites, without ending the round. At the beginning of a round, the balance vector is set to $\boldsymbol{0}$. During the round, sites update their drift vectors as local stream updates arrive. However, with rebalancing allowed, it is possible for a site to flush its current drift vector to the coordinator, during the round. When a flush occurs, the coordinator updates the balance, by adding $\boldsymbol{X}_i$ to it. After drift vector $\boldsymbol{X}_i$ is flushed, it is reset to $\boldsymbol{0}$.

Therefore, the global drift is always equal to

$$\boldsymbol{B}/k + \frac{1}{k}\sum_{i=1}^{k}\boldsymbol{X}_i. \tag{8}$$

This can be rewritten as

$$\mu \frac{\boldsymbol{B}}{\mu k} + \frac{\lambda}{k} \sum_{i=1}^{k} \frac{\boldsymbol{X}_i}{\lambda} \tag{9}$$

for some $\lambda > 0$, $\mu \geq 0$, with $\lambda + \mu = 1$ (note that we allow $\mu = 0$ when $\boldsymbol{B} = \boldsymbol{0}$, namely at the beginning of a round).

If ϕ is a convex $(A, \boldsymbol{E})$-safe function, known to the sites, we can adapt the safety condition by applying ϕ to Eq. 9. Define

$$\psi = \sum_{i=0}^{k} \lambda \phi(\frac{\boldsymbol{X}_i}{\lambda}) \qquad \text{and} \qquad \psi_B = \begin{cases} (1-\lambda)k\phi(\frac{B}{(1-\lambda)k}) & \lambda < 1, \\ 0 & \lambda = 1. \end{cases}$$

Theorem 4.1. *If ϕ is convex $(A, \boldsymbol{E})$-safe, then, for any $\lambda \in (0, 1]$,*

$$\psi + \psi_B \leq 0 \implies \boldsymbol{E} + \frac{\boldsymbol{B} + \sum_{i=1}^{k} \boldsymbol{X}_i}{k} \in A.$$

Proof. Let $\mu = 1 - \lambda$. By convexity,

$$k\phi(\frac{\boldsymbol{B} + \sum_{i=1}^{k} \boldsymbol{X}_i}{k}) \leq \mu k \phi(\frac{\boldsymbol{B}}{\mu k}) + \lambda k \phi(\frac{\sum_{i=1}^{k} \boldsymbol{X}_i}{\lambda k}) \leq \psi_B + \psi.$$

Since $k\phi$ is $(A, \boldsymbol{E})$-safe, and dominated by $\psi + \psi_B$, the claim follows. □

To monitor condition $\psi + \psi_B \leq 0$, the only modification needed to the FGM algorithm during subrounds, is in the selection of a suitable quantum θ at the beginning of each subround, so that

$$2k\theta = -(\psi + \psi_B). \tag{10}$$

4.1.1 Rebalancing FGM protocol. The extended protocol begins exactly as described in §2.4, with $\lambda = 1$. At the end of all subrounds, it is $\psi > \epsilon_\psi k \phi(\boldsymbol{0})$. Where the basic protocol would start a new round, the rebalancing protocol restores the invariant $\psi + \psi_B \leq 0$ as follows:

1. The coordinator asks some or all of the sites to flush their local drift vectors, and updates $\boldsymbol{B}$. There are many possible heuristics that can be employed to do this as conservatively as possible, dealying flushes and thus giving the opportunity to local streams to summarize their results better.
2. When all drift vectors have been received, the coordinator recomputes ψ_B and ψ, choosing a new value for λ, or failing. The choice of λ is discussed below.
3. If condition $\psi + \psi_B \leq \epsilon_\psi k \phi(\boldsymbol{0})$ is restored, a new subround is started with quantum $\theta = -(\psi + \psi_B)/(2k)$,
4. else, the round finishes and a new round starts by computing the new $\boldsymbol{E}$ and shipping it to all sites.

4.1.2 Selection of λ. The choice of a good λ is a generally dependent on the statistics of the monitored streams. Consider the "ideal" case, where $\boldsymbol{B}$ was shipped back to the sites; then, the sites could instead monitor function $\phi(\boldsymbol{x} + \boldsymbol{B}/k)$ (we would have $\psi_B = 0$). This is "ideal" in the sense that for any $\lambda > 0$,

$$\sum_{i=1}^{k} \phi(\boldsymbol{X}_i + \boldsymbol{B}/k) \leq \sum_{i=1}^{k} \lambda \phi(\boldsymbol{X}_i/\lambda) + \mu k \phi(\boldsymbol{B}/(\mu k)) = \psi + \psi_B. \tag{11}$$

Scaling the input streams. Let $Z = L(\phi)$. Geometrically, the level set of $\phi(\boldsymbol{x} + \boldsymbol{B}/k)$ is $Z - \boldsymbol{B}/k$, that is, it is a shift of Z along the $\boldsymbol{B}$-direction. The new safe zone, by our choice of λ has to be a subset of this set. We could then "scale down" Z to λZ, so that $\lambda Z \subseteq Z - \boldsymbol{B}/k$, and their boundaries touch at a point along the axis of the shift, that is, at $\boldsymbol{B}/(\mu^* k)$, where

$$\mu^* = \inf\{\mu > 0 \mid \phi(\boldsymbol{B}/(\mu k)) = 0\}.$$

This value of μ^* can easily be found iteratively by bisection. Then, $\lambda = 1 - \mu^*$. This heuristic is well-behaved in practice and is the one we have used in our experiments.

4.1.3 Discussion. To assess the effect of rebalancing on round duration, assume that the statistics of the *global stream* are such that the global state vector $\boldsymbol{S}$ maintains a rougly constant "velocity" over the stream data. Under this "statistical inertia" assumption, which is often a realistic approximation of stream statistics, our rebalancing protocol achieves a round duration at least 1/2 of the ideal maximum: if τ stream updates were processed, then processing another τ updates would lead the total drift outside the safety bounds (i.e., outside $L(\phi)$). In such conditions, rebalancing ameliorates the presence of *skew* in the trends and rates of *local* streams.

4.2 Adaptively shipping safe zones to local sites

In order to amortize the upstream cost of a round with communication benefits, it is necessary for a round to last for at least twice this many updates totally; that is, the round must last for at least $2kD$ updates, if the total communication cost of the round is to be better than the naive method. This minimum duration of a round (in terms of local stream updates) may not be not be achievable when overall variability is high.

Another practical issue, even with low variability, is the case where the stream rates of individual sites are highly unequal, e.g., they follow a 98-2 power law. Then, the cost of shipping safe zones to 98% of the sites is probably wasteful; those sites could just forward their local streams to the coordinator, and a protocol should try to save communication on those 2% of the sites which provide 98% of the stream updates.

Many previous protocols for distributed monitoring, including much of the previous work on Geometric Monitoring, do not adapt well to such problematic situations. In this section, we introduce an enhancement the FGM protocol, where such situations are handled within the protocol's basic logic. There are two subproblems addressed by our solution; (a) a systematic way to eliminate the upstream cost of shipping $\boldsymbol{E}$ to selected sites at the beginning of a round, and (b) a cost-based way to select those sites.

4.2.1 Reducing the upstream costs. One simplistic way to avoid shipping $\boldsymbol{E}$ to a site at the beginning of a round, is to put this site into "promiscuous mode", that is, to let it ship all local stream updates to the coordinator, which can then "simulate" the local node, and otherwise execute the protocol as is.

Naturally, this simplistic method will create many small messages, which we would like to avoid. This can be done if we ship to the site a cheaper safe function, such as some function of the form $b(\boldsymbol{x}) = \|\boldsymbol{x}\|_p^q + a$, which takes only 3 words (carrying p, q, a) to tramsmit. To maintain correctness, it is sufficient to guarantee simply that $\phi \leq b$. Given such a function, a site can participate normally in the FGM protocol. Naturally, the fact that this site is not equipped with the full function ϕ may cause it to end subrounds prematurely (sending many bits rapidly) and thus interfering with other sites. Although this is certainly possible, our experiments revealed that, under adverse monitoring conditions, the coordinator will often decide to ship the cheap safe function to *every* site, in which case the interference problem vanishes.

Selecting a function $b \geq \phi$ depends on the analytic properties of ϕ, and can in general be done easily. In general, we should avoid higher-degree functions, as they grow too quickly; this is possible if the degree of ϕ itself is small (which, is important for achieving better quiescent region for ϕ, as discussed previously). In order to keep our exposition simple, we do not discuss this issue in full analytic generality. Instead, we note that a 1-degree requirement can be met, if the safe zone function ϕ is nonexpansive:

$$\forall \boldsymbol{x}, \boldsymbol{y} \in V,\ |\phi(\boldsymbol{x}) - \phi(\boldsymbol{y})| \leq \|\boldsymbol{x} - \boldsymbol{y}\|. \tag{12}$$

This property is well-known in functional analysis, and is also known as *Lipschitz continuity*. It is easy to see that, in this case,

$$|\phi(\boldsymbol{x}) - \phi(\boldsymbol{0})| \leq \|\boldsymbol{x}\|,$$

which implies that

$$\phi(\boldsymbol{x}) \leq \|\boldsymbol{x}\| + \phi(0).$$

An important class of safe functions that are non-expansive are the Signed Distance Functions of convex sets. Also, the gauge functions with bounded level-set (including all norms and semi-norms) can be scaled to be nonexpansive.

Selecting the sites which will use the "cheap" function b is crucial. We propose a solution based on a cost model and some collected statistics, much in the spirit of database query optimization. In the rest of this section, we will ignore the FGM rebalancing protocol, and instead focus on the basic FGM protocol. This is done for the sake of keeping our optimization algorithm simple. However, once the "plan" for a round is selected, the full FGM protocol with rebalancing can be executed for the round.

4.2.2 Modeling the communication cost of a round. Assume that the coordinator is at the beginning of a round, with current estimate $\boldsymbol{E}$ and has selected a non-expansive safe function ϕ. Let $d_i, i = 1, \ldots, k$ be indicator variables; d_i is equal to 1 when the full safe function ϕ is to be shipped to local site i, and 0 if the cheap safe function is to be used. In other words, d_i encodes the optimized "query plan" for the upcoming round. Let $\boldsymbol{d}$ denote the vector of d_i values. Our goal is to select the plan $\boldsymbol{d}$ that maximizes the gain of the round.

Assume that, based on the decision $\boldsymbol{d}$, the length of the next round is going to be τ. Furthermore, assume that a fraction $\gamma_i \tau$ of these updates arrives at local stream i. The benefit of the round in terms of summarizing τ updates in the local state vectors, is

$$g_0 = \tau - \sum_{i=1}^{k} \min\{\gamma_i \tau, D\}, \tag{13}$$

where $\min\{\gamma_i \tau, D\}$ reflects the downstream cost of site i, which will ship $\gamma_i \tau$ raw updates, instead of the D-dimensional drift vector, if $\gamma_i \tau < D$. In addition, the upstream cost of the round is $D \sum_{i=1}^{k} d_i$ (where we assume that the difference in the cost of shipping ϕ vs. b is D). Therefore, we must select $\boldsymbol{d}$ so as maximize the round's *gain*,

$$g = \tau - \sum_{i=1}^{k} \min\{\gamma_i \tau, D\} - D \sum_{i=1}^{k} d_i. \tag{14}$$

The challenge is to predict $\tau(\boldsymbol{d})$, given a choice for $\boldsymbol{d}$. To this end, consider ψ as function of "time" (updates):

$$\psi(t) = \sum_{d_i=1} \phi(\boldsymbol{X_i}(t)) + \sum_{d_i=0} \|\boldsymbol{X}_i(t)\| - \phi(\boldsymbol{0}) \tag{15}$$

The current round can be seen as the transition of the system from a state where $\psi = k\phi(\boldsymbol{0})$ to a state where $\psi = 0$. Of course, this transition will in general follow a complicated, non-linear trajectory in the quiescent region. However, we adopt a simplistic linear estimate. In particular, we model the behaviour of each local stream i by two rates, α_i and β_i, assuming simplistically that

$$\begin{aligned} \phi(\boldsymbol{X}_i(t)) &\approx \phi(\boldsymbol{0}) + |\phi(\boldsymbol{0})|\alpha_i t \quad (16) \\ \|\boldsymbol{X}_i(t)\| + \phi(\boldsymbol{0}) &\approx \phi(\boldsymbol{0}) + |\phi(\boldsymbol{0})|\beta_i t \quad (17) \end{aligned}$$

We shall assume that $0 < \alpha_i < \beta_i$.

Based on this simple-minded model, the prediction of a round's length τ, as a function of $\boldsymbol{d}$, based on Eq. 15, is

$$\tau = \frac{k}{\beta_{\text{tot}} - \boldsymbol{d} \cdot \boldsymbol{\theta}}, \tag{18}$$

where $\beta_{\text{tot}} = \sum_{i=1}^{k} \beta_i$ and $\boldsymbol{\theta}$ is the vector of values $\theta_i = \beta_i - \alpha_i$.

4.2.3 Maximizing the gain of a round. It is required to find the value of $\boldsymbol{d}$ that maximizes the gain g (Eq. 14). An exhaustive search of the solution space would require time $O(2^k)$, which would not scale well to large k. Thankfully, it turns out that a simple greedy algorithm is sufficient to maximize g. The key observation is that $g_0(\tau)$ (from Eq. 13) is non-decreasing in τ. Fix some number $0 \leq n \leq k$. We wish to find a feasible solution $\boldsymbol{d}^*$, with $\sum_i d_i^* = n$, which maximizes g among all solutions $\boldsymbol{d'}$ with $\sum_i d'_i = n$. But since $g(\boldsymbol{d}^*) = g_0(\tau(\boldsymbol{d}^*)) - nD$, and g_0 is non-decreasing in τ, it suffices to maximize $\tau(\boldsymbol{d}^*)$. To do this, simply set $d_i^* = 1$, iff θ_i is among the n largest coordinates of $\boldsymbol{\theta}$ (ties are broken arbitrarily). Now, g can be optimized by comparing among $k+1$ solutions, one for each value of n. Furthermore, since an optimal solution for $n+1$ subsumes an optimal solution for n, the whole computation can be performed in $O(k \log k)$ steps (essentially for sorting vector $\boldsymbol{\theta}$).

4.2.4 Obtaining estimates for local streams. It remains to discuss the estimation of α_i, β_i and γ_i in each round. In this paper, we explored the simplest possible alternative: simply use the data collected at the end of the previous round, to obtain fresh estimates of all three parameters. Since at the end of a round the coordinator has received each drift $\boldsymbol{X}_i$, together with a count of updates to each local stream during the round, all three parameters can be computed directly, more or less from Eqs. 16 and 17.

Some care must be taken, to ensure $0 < \alpha_i < \beta_i$; in particular, Eq. 16 may yield a non-positive value. If this occurs, then simply set α_i to a small positive value (so that θ_i is minimum among the components of vector $\boldsymbol{\theta}$. Also, when $\beta_i = 0$ or $\gamma_i = 0$ (there were no updates to the site in the previous round), simply set $d_i = 0$ and ignore this site in the optimization process.

4.2.5 Discussion. Our estimates for modeling local streams are simple to acquire in practice, but may yield estimates which may not represent well the evolution of the system. After all, predictions are hard, especially about the future! Thankfully, our approach, of estimating τ in order to decide on the next round's plan, is relatively insensitive to the exact value of τ, as it is essentially a based of thresholds, determined by local stream rate predictions, which can be predicted much more accurately.

In practice the algorithm managed to perform quite well, making the FGM protocol quite robust in adverse situations of very high variability, compared to executions that shipped a safe function to every site. Most of the time, the selection of $\boldsymbol{d}$ values either resulted in almost all 1s (when variability was low) or in almost all 0s (during high variability). Also, during periods of medium variability, the algorithm would alternate between these two decisions for a few rounds.

Improving on this algorithm is certainly an interesting problem. On the prediction side, higher-order polynomial models in place of Eqs. (16–17) can in principle be constructed. Whether these more elaborate modeling would benefit the final communication cost in real data settings, remains to be seen.

5 EXPERIMENTAL EVALUATION

We performed an extensive experimental study of the FGM protocol, over a variety of datasets and streaming parameters, with emphasis on validating our claims of resilience to adverse situations. For lack of space, we only present results from the WorldCup dataset [2], which contains log traces of all requests sent to the 1998 World Cup web site, consisting of 33 mirrors spread around the globe and receiving 1.3 billion http requests. Our experiments used only data from day 46, during which 50.3 million requests where received by 27 mirror sites. From this data, we constructed stream records over the schema R(CID, TYPE), where CID is the (anonymized) client address of the http request, and TYPE is the type of file requested (HTML, image etc).

On this stream, we approximately monitored two continuous queries. Both queries operate on Fast-AGMS sketches [8] on the input streams. A Fast-AGMS sketch $\boldsymbol{S}$ is stored as a $d \times w$ matrix $\boldsymbol{S}$ of integer counters, and can be used to estimate join and self-join sizes within accuracy $\Theta(1/\sqrt{w})$ with probability at least $1-2^{-\Theta(d)}$. Each stream update changes the sketch by modifying one cell in each row vector $\boldsymbol{S}[i]$ (totally, d cells) by ±1, according to certain hash functions.

The first query monitors the self-join size of $R \bowtie_{\text{CID}} R$. To estimate this query, an AGMS sketch is used as the state vector to summarize all records. The query function is the self-join size estimate,

$$Q_1(\boldsymbol{S}) = \underset{i=1,\ldots,d}{\text{median}}\{\sum_{j=1}^{w} \boldsymbol{S}[i,j]^2\} = \underset{i=1,\ldots,d}{\text{median}}\{\boldsymbol{S}[i]^2\}.$$

The second query monitors the join size of

$$\sigma_{\text{TYPE=HTML}}(R) \ \bowtie_{\text{CID}} \ \sigma_{\text{TYPE}\neq\text{HTML}}(R).$$

For this query, the state vector consisted of the concatenation of two sketches, $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$. The monitored query function is

$$Q_2(\boldsymbol{S_1 S_2}) = \underset{i=1,\ldots,d}{\text{median}}\{\sum_{j=1}^{w} \boldsymbol{S}_1[i,j]\boldsymbol{S}_2[i,j]\} = \underset{i=1,\ldots,d}{\text{median}}\{\boldsymbol{S}_1[i]\boldsymbol{S}_2[i]\}.$$

Note that query function Q_2 is much more challenging than query Q_1 in terms of variability.

5.1 Experimental setup

We explored the space of four parameters: AGMS sketch size, size of sliding window over the streams, monitoring accuracy ε and the number of sites k.

We evaluated queries Q_1 and Q_2 both in the cash-register model (each record was inserted one at a time), and also in the turnstile model, where we used a time-based sliding window (ranging from 1hr to 4hrs) to generate record deletions. Naturally, the variability of our queries decreases as the time window increases. Also, time-based windows yield higher variability than fixed-size ones. We allowed the monitoring accuracy to vary as $\varepsilon \in [0.02, 0.1]$.

Finally, in order to study the effect of k (the number of sites) on performance, we created synthetic streams by hashing the original 27 local site ids to fewer site ids, for $k \in [2:20]$. Naturally, we also used the original (real) data, for $k = 27$.

Note that, in all experiments presented, the "global" stream was identical, and we simply changed the distribution of the data in time (by sliding windows), and among local streams.

5.1.1 Safe functions employed.

We implemented nonexpansive, convex safe functions for queries Q_1 and Q_2 following the technique of [13]. To monitor query Q_1 for estimate sketch $\boldsymbol{E}$ we need to ensure that $(1-\varepsilon)|Q_1(\boldsymbol{E})| \leq Q_1(\boldsymbol{S}) \leq (1+\varepsilon)|Q_1(\boldsymbol{E})|$.

We can rewrite it compactly (applying properties of the median) as

$$\pm\big(Q_1(\boldsymbol{S}) - T^{\pm}\big) = \underset{i=1,\ldots,d}{\text{median}}\{\pm(\boldsymbol{S}[i]^2 - T^{\pm})\} \leq 0.$$

where, for $\pm \in \{+,-\}$, $T^{\pm} = (1 \pm \varepsilon)|Q_1(\boldsymbol{E})|$, respectively.

The safe function $\phi(\boldsymbol{X})$ we used is *composed* as

$$\phi(\boldsymbol{X}) = \max\big(\phi^{-}(\boldsymbol{X}), \phi^{+}(\boldsymbol{X})\big),$$

where $\phi^{\pm}$ is safe for condition $\pm\big(Q_1(\boldsymbol{S}) - T^{\pm}\big) \leq 0$ respectively. Following the methodology of [13] for the median, we used

$$\phi^{\pm}(\boldsymbol{X}) = \max_{I \in \binom{D^{\pm}}{|D^{\pm}|-(d-1)/2}} \frac{\sum_{i\in I} |\phi_i^{\pm}(\boldsymbol{0})| \cdot \phi_i^{\pm}(\boldsymbol{X}[i])}{\sqrt{\sum_{i\in I} |\phi_i^{\pm}(\boldsymbol{0})|^2}},$$

where $D^{\pm} = \{i \mid 1 \leq i \leq d \text{ and } \pm(\boldsymbol{E}[i]^2 - T^{\pm}) < 0\}$. Note that the notation $I \in \binom{D}{n}$ means "I ranges over all n-subsets of D".

Functions $\phi_i^{\pm}(\boldsymbol{x}), i = 1,\ldots,d$, must be safe for conditions $\pm(\boldsymbol{S}[i]^2 - T^{\pm}) \leq 0$ respectively. We used

$$\phi_i^{+}(\boldsymbol{x}) = \|\boldsymbol{x}+\boldsymbol{E}[i]\|-\sqrt{T^{+}} \quad \text{and} \quad \phi_i^{-}(\boldsymbol{x}) = \sqrt{T^{-}}-\frac{\boldsymbol{E}[i]}{\|\boldsymbol{E}[i]\|}(\boldsymbol{E}[i]+\boldsymbol{x}).$$

The same methodology was applied to derive the safe functions for the Q_2 query; the derivation is very similar to the above, however, the actual formulas for $\phi_i^{\pm}$ for conditions $\pm(\boldsymbol{S}_1[i]\boldsymbol{S}_2[i] - T^{\pm}) \leq 0$ are a bit involved and are omitted due to space constraints; we refer the reader to [13] (Section 6.3) for details, as well as for the justification of the above steps.

5.1.2 Tested protocols.

In order to compare the performance of the FGM protocol to previous work, we implemented a well-studied version of the GM protocol, based on Safe Zones [22], with a rebalancing policy along the lines of [28]. The Safe Zones used where defined using the safe functions of the FGM described above, so as to fairly contrast the inherent communication costs of the GM and FGM protocols.

To study the effect of our cost-based optimizer, we ran versions of FGM with and without it. Overall, the acronyms of the 3 protocols tested are as follows:

Acronym	Protocol
GM	classic GM protocol with rebalancing.
FGM	FGM protocol without cost-based optimizer.
FGM/O	FGM protocol with cost-based optimizer.

5.2 Performance in typical workloads

Our first set of experiments concerns the behaviour of the protocols in a non-adverse scenario (using a 4ht window over the data), monitoring accuracy $\varepsilon = 0.1$, and sketch sizes, $D = 7000$. The results depicted in Figs. 2 and 3 (corresponding to semi-join and join queries) depict this cost as a function of k, both in the turnstile and in the cash-register model.

With respect to communication cost, observe that, as k grows, the FGM protocols exhibit 2–3 times lower communication cost

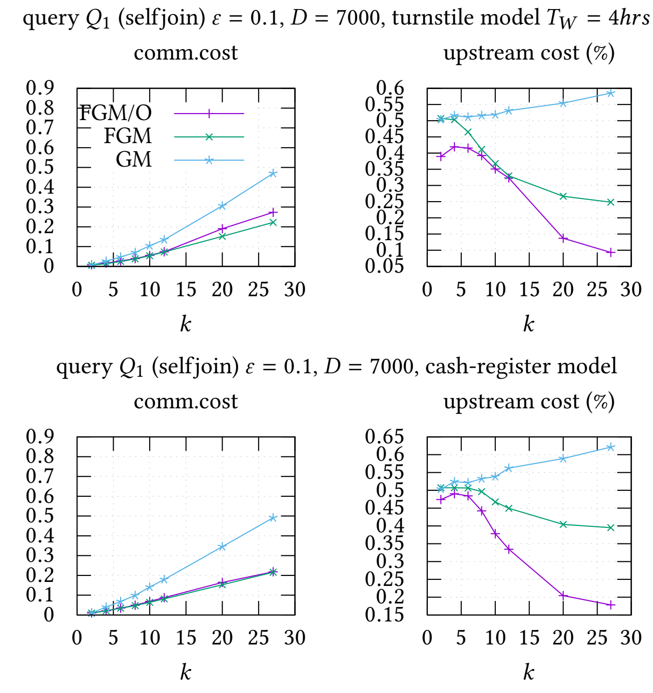


Figure 2: Performance of the GM and FGM protocols, monitoring a self-join query, over k. The top row shows the cost in the tunrstile model (with a window over the streams) and the bottom row show the cost in the cash-register model.

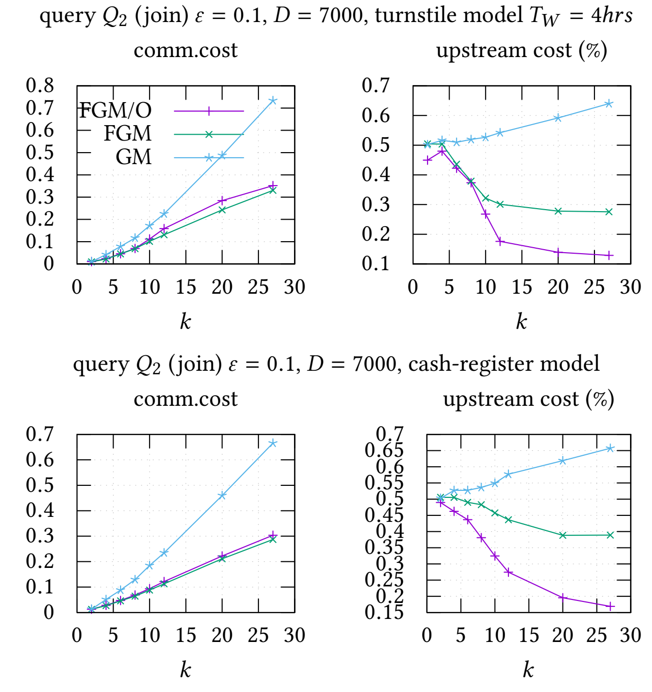


Figure 3: Performance of the GM and FGM protocols, monitoring a join query, over k. The top row shows the cost in the tunrstile model (with a window over the streams) and the bottom row show the cost in the cash-register model.

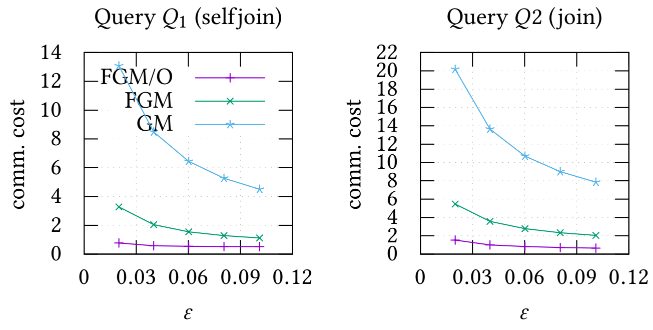


Figure 4: Communication cost for queries Q_1, Q_2, under a difficult workload. Observe that, except for FGM/*O, all protocols exhibit much higher communication cost that the size of the streamed data. Here, $k = 27$, $D = 35000$, $T_W = 1hr$.

than the GM protocols. On the other hand, for small values of k, the difference is not as pronounced.

The graphs on the right side, depicting upsteam communication costs as a percentage of total communication cost, reveals the cause of this behaviour. It is shown that the upstream cost of the standard geometric method grows as a percent of total, as the number of sites increases. This is due to two causes: first, as more sites partake in the monitoring, the strictness of the GM's monitoring condition causes frequent violations of the safety invariant of GM, while most of these violations are false positives. The rebalancing strategy of GM algorithms is unable to overcome this increase (note that, without rebalancing, the GM algorithm's total cost increases even faster, as each false violation would cause a full synchronization).

By contrast, the upstream cost of FGM *decreases* (as a percent of total communication). This is the case both with and without the cost-based optimizer. When k is small, upstream and downstream costs are roughly similar, which is true for the GM as well. As k increases however, the total cost (which increases with k naturally) is dominated by the downstream cost, of shipping data to the coordinator. This is both due to the improved safety condition of FGM, but also to the ovehead-free rebalancing performed.

Note finally the effect of cost-based optimization, which tries to aggressively minimize the upstream cost, even at the expense of downstream cost. Although total cost does not change much, the upstream percentage reduces much further. This is because the cost-based optimizer will decide not to ship safe functions to the sites in many rounds. This choice worsens the quality of summarization at the local nodes, increasing downstream costs, but manages to keep upstream costs low, while achieving good total cost.

5.3 Performance in adverse conditions

We now evaluate the performance of FGM and GM protocols under an adverse scenario, on the real WorldCup dataset, where $k = 27$. We have a large $D = 35,000$, and the stream's window is 1hr, leading to high variability. Fig. 4.

Under these conditions, round lengths are too short to amortize the cost of shipping safe zones to the sites. Therefore, all methods except for FGM/O incur excessive communication costs, in fact several times over the size of the streamed data. This

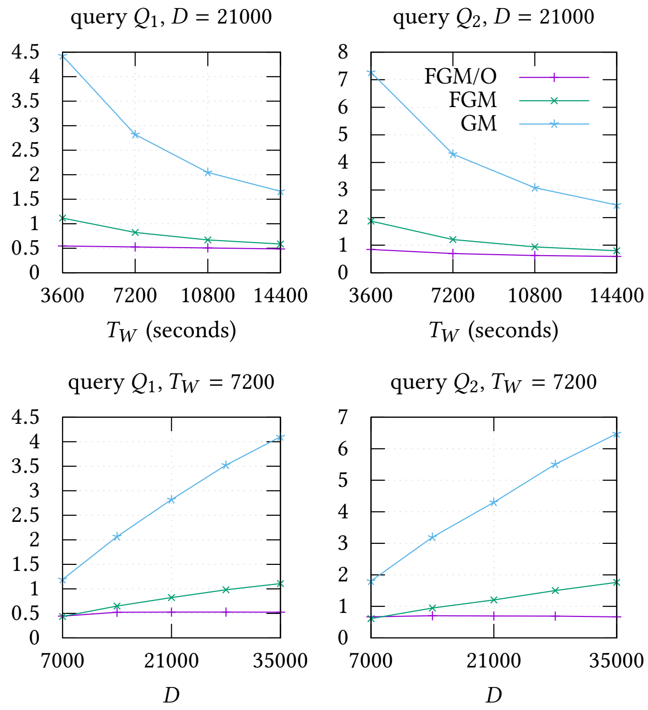


Figure 5: Communication cost for queries Q_1 (left column) and Q_2 (right column), over varying sliding windows (T_w, top row) and sketch size (D, bottom row). In all cases, it was $k = 27$ and $\varepsilon = 0.06$.

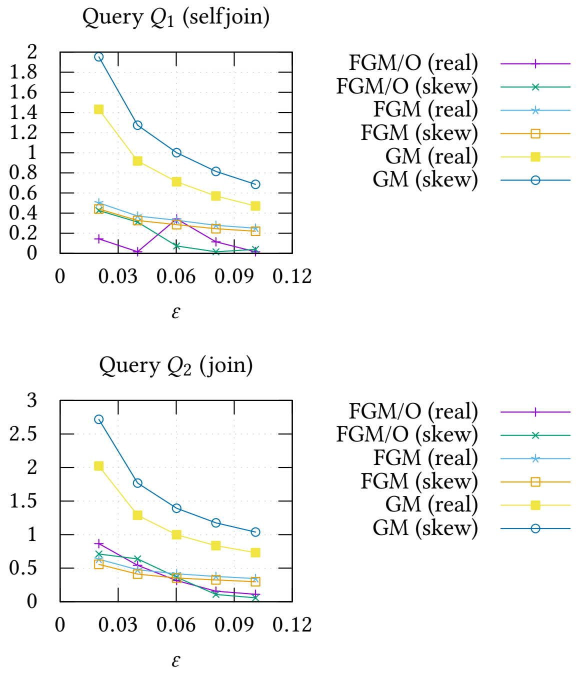


Figure 6: Communication cost for queries Q_1 (top) and Q_2 (bottom), over varying accuracy ε. For each protocol, two curves are shown, one for the real dataset and one for the skewed dataset. For both plots: $k = 27$, $D = 7000$, turnstile model (T_W=4hrs)

is not unexpected; consider that, shipping safe zones to all 27 sites, transmits roughly 3.8 Mbytes of data. Combined with short rounds due to high variability and low values of ε results in excessive overhead.

By contrast, the cost-based optimizer, although it did not deliver significant gains compared to the size of the streamed data, managed to keep the total cost quite low. This was achieved by selecting to avoid the overhead of shipping safe functions in most of the rounds.

5.3.1 Dependence on size of state vectors and on variability. The effect of variability, which decreases as the time window sliding over the stream becomes wider, is quite strong on performance. The top row of plots in Fig. 5 demonstrates this for turnstile queries, where the time window T_W changed from 1 hour to 4 hours. In particular, for the Q_2 function, using the cost model improved performance by two times over FGM (and by 4 over the GM methods), when T_W=1hr.

Similarly strong is the effect of D on performance, as depicted in the bottom row of plots of Fig. 5. In fact, the cost grows linearly with D, except for the case of FGM/O, where the cost-based optimizer switched to the cheap safe functions, achieving a small amount of compression.

5.4 The effect of skew

In order to evaluate the behaviour of our protocols under the presense of skew, we contrast the change in communication when the (real) dataset becomes more skewed. To introduce skew, we constructed a new dataset as follows: we selected 8 sites (out of a total of 27), namely those with local streams of greatest size. Then, we replaced the local stream of one of these sites—which will be referred to as the *hot site*—by the union of all 8 local streams, while the 7 remaining sites received empty local streams. In this *skewed* dataset, one local stream now provides almost half the data to the system, while 7 out of 27 local streams provide no data. However, at each point in time, the *global stream* of the skewed dataset is identical to the global stream of the real dataset.

Fig. 6 depicts the effect of skew on the communication cost. Each protocol was run with both the real and the skewed dataset. Unsuprisingly, the GM protocol's communication cost increases as skew is introduced; this is a well-known weakness of the classic geometric method. The source of the increased cost is a substantial increase in the upstream cost, because of frequent local violations at the hot site.

The FGM prococol without the cost-based optimizer on the other hand, shows resilience in the presence of skew; in fact, the communication cost improves slightly under skew. The key reason is that the ψ-value of the system under the real dataset, is always equal to the ψ-value of the system under the skewed dataset. Therefore, the coordinators in the two systems will perform the exact same number of rounds. The slight improvement is due to a reduction in the downstream cost among the 8 sites; the downstream cost of the hot site has not increased substantially (since the number of rounds remains the same), but the downstream costs of the 7 sites whose local stream vanished has decreased to almost 0 (since these sites will not ship local vectors to the coordinator).

The introduction of the cost-based optimizer is again largely beneficial to the performance. In previous experiments under adverse conditions (e.g., Fig. 4), the benefit of the optimizer was in keeping the upstream cost from becoming too large. In this scenario where skew is introduced, the benefit of the cost-based optimizer materializes more consistently when $\varepsilon \geq 0.05$, where significant benefit to the upstream cost of a round accrues, since the coordinator will undoubtedly choose the cheap safe functions for the 7 sites with empty local streams. Note that, in this scenario, the ψ-values of the constrasted systems are no longer equal (since different optimizer choices affect the actual ψ).

In this experiment, one can also observe the somewhat erratic effect of the cost-based optimizer, due to the crudeness of modeling local stream behaviour (interestingly, in the presense of skew, the behaviour is less erratic). The erratic behaviour is observed in the transition between the two extremes of small and large values of ε; for values of ε around 0.05, it seems that the cost-based optimizer will often be fooled into making sub-optimal choices. However, this is preferable to not using it at all.

Overall, our experimental results demonstrate that the FGM protocol manages to ameliorate the shortcomings of classic GM protocols, both under adverse conditions as well as in the presense of skew in the distributed stream.

6 CONCLUSIONS AND IMPLICATIONS FOR PRACTICE

We have proposed Functional Geometric Monitoring, a novel method for distributed stream monitoring, which offers significant improvements over previous techniques in terms of performance, scalability and robustness. FGM is generally applicable, it can provide worst-case guarantees for problems that were hitherto provided only by problem-specific algorithms, and it is robust in high variability and skew situations, curing an important shortcoming of previous general techniques.

Real-world stream-processing engines are typically customized by providing data-handling code (e.g., mapper/reducer functions in Hadoop, spouts and bolts in STORM, etc), which is independent of distributed execution concerns. The engine orchestrates the distributed execution of this code on a distributed platform, applying complex execution policies (resource allocation, load balancing, networking patterns, failure tolerance, etc).

The salient practical feature of FGM is that it fits this pattern extremely well, as it strictly encapsulates the specifics of monitored queries into data-handling code, namely, routines and data structures—such as sketches—to summarize local streams, and safe function implementations on these summaries. This code is platform-agnostic and an FGM implementation can deploy it on a distributed platform and execute it in a black-box fashion, under any desired execution policy.

Other aspects of FGM are alse important in practice. Although high-quality safe functions for complex query operators can be hard to derive, safe function composition can ease the burden many cases. Furthermore, the FGM protocol is resilient to loss of precision due to computational round-off errors. In addition, since local nodes are memoryless from one round to the next, the FGM protocol is compatible with relatively simple and cheap failure recovery policies.

Acknowledgement. The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825070.

REFERENCES

[1] C. Arackaparambil, J. Brody, and A. Chakrabarti. Functional monitoring without monotonicity. In *ICALP (1)*, 2009.

[2] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *Netwrk. Mag. of Global Internetwkg.*, 14(3):30–37, May 2000.

[3] B. Babcock and C. Olston. Distributed top-k monitoring. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2003. ACM.

[4] S. Burdakis and A. Deligiannakis. Detecting outliers in sensor networks using the geometric approach. In *ICDE*, 2012.

[5] G. Cormode and M. Garofalakis. "Sketching Streams Through the Net: Distributed Approximate Query Tracking". In *Proc. of the 31st Intl. Conference on Very Large Data Bases*, Trondheim, Norway, Sept. 2005.

[6] G. Cormode and M. Garofalakis. "Approximate Continuous Querying over Distributed Streams". *ACM Transactions on Database Systems*, 33(2), June 2008.

[7] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. "Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches". *Foundations and Trends in Databases*, 4(1-3), 2012.

[8] G. Cormode and M. N. Garofalakis. Sketching streams through the net: Distributed approximate query tracking. In *VLDB*, 2005.

[9] G. Cormode, S. Muthukrishnan, and K. Yi. Algorithms for distributed functional monitoring. In *SODA*, 2008.

[10] D. Felber and R. Ostrovsky. Variability in data streams. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, pages 251–260, New York, NY, USA, 2016. ACM.

[11] M. Gabel, D. Keren, and A. Schuster. Anarchists, unite: Practical entropy approximation for distributed streams. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 837–846, New York, NY, USA, 2017. ACM.

[12] M. Garofalakis, D. Keren, and V. Samoladas. "Sketch-based Geometric Monitoring of Distributed Stream Queries". In *Proc. of the 39th Intl. Conference on Very Large Data Bases*, Trento, Italy, Aug. 2013.

[13] M. N. Garofalakis and V. Samoladas. Distributed query monitoring through convex analysis: Towards composable safe zones. In *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy*, pages 14:1–14:18, 2017.

[14] N. Giatrakos, A. Deligiannakis, M. N. Garofalakis, I. Sharfman, and A. Schuster. Prediction-based geometric monitoring over distributed data streams. In *SIGMOD*, 2012.

[15] R. Gupta, K. Ramamritham, and M. K. Mohania. "Ratio threshold queries over distributed data sources". In *Proc. of the 39th Intl. Conference on Very Large Data Bases*, Trento, Italy, Aug. 2013.

[16] S. R. Kashyap, J. Ramamirtham, R. Rastogi, and P. Shukla. Efficient constraint monitoring using adaptive thresholds. In *ICDE*, pages 526–535, 2008.

[17] R. Keralapura, G. Cormode, and J. Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *SIGMOD*, 2006.

[18] D. Keren, G. Sagy, A. Abboud, D. Ben-David, A. Schuster, I. Sharfman, and A. Deligiannakis. "Geometric Monitoring of Heterogeneous Streams". *IEEE Transactions on Knowledge and Data Engineering*, 26(8), Aug. 2014.

[19] D. Keren, I. Sharfman, A. Schuster, and A. Livne. Shape sensitive geometric monitoring. *IEEE Trans. Knowl. Data Eng.*, 24(8), 2012.

[20] A. Lazerson, M. Gabel, D. Keren, and A. Schuster. One for all and all for one: Simultaneous approximation of multiple functions over distributed streams. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 203–214, New York, NY, USA, 2017. ACM.

[21] A. Lazerson, D. Keren, and A. Schuster. Lightweight monitoring of distributed streams. In *Proc. of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1685–1694, New York, NY, USA, 2016. ACM.

[22] A. Lazerson, I. Sharfman, D. Keren, A. Schuster, M. Garofalakis, and V. Samoladas. "Monitoring Distributed Streams using Convex Decompositions". In *Proc. of the 41st Intl. Conference on Very Large Data Bases*, Aug. 2015.

[23] S. Meng, T. Wang, and L. Liu. Monitoring continuous state violation in datacenters: Exploring the time dimension. In *ICDE*, pages 968–979, 2010.

[24] S. Michel, P. Triantafillou, and G. Weikum. Klee: a framework for distributed top-k query algorithms. In *VLDB '05*. VLDB Endowment, 2005.

[25] O. Papapetrou and M. Garofalakis. "Continuous Fragmented Skylines over Distributed Streams". In *Proc. of the 30th Intl. Conference on Data Engineering*, Chicago, Illinois, Apr. 2014.

[26] S. Shah and K. Ramamritham. Handling non-linear polynomial queries over dynamic data. In *ICDE*, 2008.

[27] I. Sharfman, A. Schuster, and D. Keren. "A geometric approach to monitoring threshold functions over distributed data streams". In *SIGMOD*, 2006.

[28] I. Sharfman, A. Schuster, and D. Keren. "A geometric approach to monitoring threshold functions over distributed data streams". *ACM Trans. Database Syst.*, 32(4), 2007.

[29] D. P. Woodruff and Q. Zhang. Tight bounds for distributed functional monitoring. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 941–960, New York, NY, USA, 2012. ACM.

[30] K. Yi and Q. Zhang. Optimal tracking of distributed heavy hitters and quantiles. In *PODS*, 2009.

Efficient Window Aggregation with General Stream Slicing

Jonas Traub[1], Philipp Grulich[2], Alejandro Rodríguez Cuéllar[1], Sebastian Breß[1,2]
Asterios Katsifodimos[3], Tilmann Rabl[1,2], Volker Markl[1,2]

[1]Technische Universität Berlin [2]DFKI GmbH [3]Delft University of Technology

ABSTRACT

Window aggregation is a core operation in data stream processing. Existing aggregation techniques focus on reducing latency, eliminating redundant computations, and minimizing memory usage. However, each technique operates under different assumptions with respect to workload characteristics such as properties of aggregation functions (e.g., invertible, associative), window types (e.g., sliding, sessions), windowing measures (e.g., time- or count-based), and stream (dis)order. Violating the assumptions of a technique can deem it unusable or drastically reduce its performance.

In this paper, we present the first general stream slicing technique for window aggregation. General stream slicing automatically adapts to workload characteristics to improve performance without sacrificing its general applicability. As a prerequisite, we identify workload characteristics which affect the performance and applicability of aggregation techniques. Our experiments show that general stream slicing outperforms alternative concepts by up to one order of magnitude.

1 INTRODUCTION

The need for real-time analysis shifts an increasing number of data analysis tasks from batch to stream processing. To be able to process queries over unbounded data streams, users typically formulate queries that compute aggregates over bounded subsets of a stream, called windows. Examples of such queries on windows are average vehicle speeds per minute, monthly revenue aggregations, or statistics of user behavior for online sessions.

Large computation overlaps caused by sliding windows and multiple concurrent queries lead to redundant computations and inefficiency. Consequently, there is an urgent need for *general* and *efficient* window aggregation in industry [7, 41, 50]. In this paper, we contribute a general solution which not only improves performance but also widens the applicability with respect to window types, time domains, aggregate functions, and out-of-order processing. Our solution is generally applicable to all data flow systems which adopt a tuple-at-a-time processing model (e.g., Apache Storm, Apache Flink, and other Apache Beam-based systems).

To calculate aggregates of overlapping windows, the database community has been working on *aggregation techniques* such as B-Int [3], Pairs [28], Panes [30], RA [42] and Cutty [10]. These techniques compute partial aggregates for overlapping parts of windows and reuse these partial aggregates to compute final aggregates for overlapping windows. We believe that these techniques are not widely adopted in open-source streaming systems for two main reasons: first, the literature on streaming window aggregation is fragmented and, second, every technique has its own assumptions and limitations. As a consequence, it is not clear for researchers and practitioners under which conditions which streaming window aggregation techniques should be used.

General purpose streaming systems require a window operator which is applicable to many types of aggregation workloads. At the same time, the operator should be as efficient as specialized techniques which support selected workloads only.

As our first contribution we identify the workload characteristics which may or may not be supported by existing specialized window aggregation techniques. Those characteristics are: i) *window types* (e.g., sliding, session, tumbling), ii) *windowing measures* (e.g., time or tuple-count), iii) *aggregate functions* (e.g., associative, holistic), and iv) *stream order*. We then conduct an extensive literature survey and classify existing techniques with respect to their underlying concepts and their applicability.

We identify stream slicing as a common denominator on top of which window aggregation can be implemented efficiently. Consequently, our second main contribution is *a general stream slicing technique*. Existing slicing-based techniques do not support complex window types such as session windows [28, 30], do not consider out-of-order processing [10], or limit the type of aggregation functions [10, 28, 30]. With general stream slicing, we provide a single, generally applicable, and highly efficient solution for streaming window aggregation. Our solution inherits the performance of specialized techniques, which use stream slicing, and generalizes stream slicing to support diverse workloads. Because we integrate all workloads into one general solution, we enable computation sharing among all queries with different window types (sliding, sessions, user-defined, etc.) and window measures (e.g., tuple-count or time). General stream slicing is available open source and can be integrated into streaming systems directly as a library[1].

General stream slicing breaks down slicing into three operations on slices, namely `merge`, `split`, and `update`. Specific workload characteristics influence what each operation costs and how often operations are performed. By taking into account the workload characteristics, our slicing technique i) stores the tuples themselves only when it is required which saves memory and ii) minimizes the number of slices that are created, stored, and recomputed. One can extend our techniques with additional aggregations and window types without changing the three core slicing operations. Thus, these core operations may be tuned by system experts while users can still implement custom windows and aggregations.

The contributions of this paper are as follows:

(1) We identify the workload characteristics which impact the applicability and performance limitations of existing aggregation techniques (Section 4).
(2) We contribute *general stream slicing*, a generally applicable and highly efficient solution for streaming window aggregation in dataflow systems (Section 5).
(3) We evaluate the performance implications of different usecase characteristics and show that stream slicing is generally applicable while offering better performance than existing approaches (Section 6).

The remainder of this paper is structured as follows: We first provide background information in Section 2 and present concepts of aggregation techniques in Section 3. We then present our contributions in Section 4, 5, and 6 and discuss related work in Section 7.

[1] Open Source Link: https://github.com/TU-Berlin-DIMA/scotty-window-processor

Series ISSN: 2367-2005

10.5441/002/edbt.2019.10

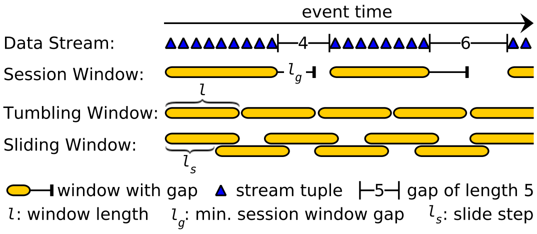


Figure 1: Common Window Types.

2 PRELIMINARIES

Streaming window aggregation involves special terminology with respect to window types, timing, stream order, and data expiration. This section revisits terms and definitions, which are required for the remainder of this paper.

Window Types. A window type refers to the logic based on which systems derive finite windows from a continuous stream. There exist diverse window types ranging from common sliding windows to more complex data-driven windows [17]. We address the diversity of window types with a classification in Section 4.4. For now, we limit the discussion to *tumbling* (or *fixed*), *sliding*, and *session* windows (Figure 1) which we use in subsequent examples. A *tumbling* window splits the time into segments of equal length l. The end of one window marks the beginning of the next window. *Sliding* windows, in addition to the length l, also define a slide step of length l_s. This length determines how often a new window starts. Consecutive windows overlap when $l_s < l$. In this case, tuples may belong to multiple windows. A *session* window typically covers a period of activity followed by a period of inactivity [1]. Thus, a session window times out (ends) if no tuple arrives for some time gap l_g. Typical examples of sessions are taxi trips, browser sessions, and ATM interactions.

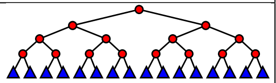
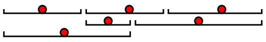
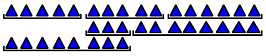
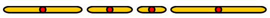
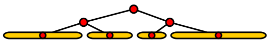
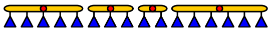
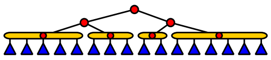
Notion of Time. One can define windows on different measures such as times and tuple-counts. The *event-time* of a tuple is the time when an event was captured and the *processing-time* is the time when an operator processes a tuple [1, 9]. Technically, an event-time is a timestamp stored in the tuple and processing-time refers to a system clock. If not indicated otherwise, we refer to event-time windows in our examples because applications typically define windows on event-time.

Stream Order. Input tuples of a stream are in-order if they arrive chronologically with respect to their event-times, otherwise, they are out-of-order [1, 33]. In practice, streams regularly contain out-of-order tuples because of transmission latencies, network failures, or temporary sensor outages. We differentiate in-order tuples from out-of-order tuples and in-order streams from out-of-order streams. Let a stream S consist of tuples $s_1, s_2, s_3, \ldots$ where the subscripts denote the order in which an operator processes the tuples. Let the event-time of any tuple s_x be $t_e(s_x)$.

- A tuple s_x is *in-order* if $t_e(s_x) \geq t_e(s_y)\ \forall\, y < x$.
- A stream is in-order iff all its tuples are in-order tuples.

Punctuations, Watermarks, and Allowed Lateness. *Punctuations* are annotations embedded in a data stream [47]. Systems use punctuations for different purposes: *low-watermarks* (in short *watermarks*) indicate that no tuple will arrive with a timestamp smaller than the watermark's timestamp [1]. Many systems use watermarks to control how long they wait for out-of-order tuples before they output a window aggregate [2]. *Window punctuations* mark window starts and endings in the stream [14, 20]. The *allowed lateness*, specifies how long systems store window aggregates. If an out-of-order tuple arrives after the watermark, but in the allowed lateness, we output updated aggregates.

Partial Aggregates and Aggregate Sharing. The key idea of partial aggregation is to compute aggregates for subsets of the stream as intermediate results. These intermediate results are *shared* among overlapping windows to prevent repeated computation [3, 28, 51]. In addition, one can compute partial aggregates incrementally when tuples arrive [42]. This reduces the memory footprint if a technique stores few partial aggregates instead of all stream tuples in the allowed lateness. It also reduces the latency because aggregates are pre-computed when windows end.

	Memory Usage	Example
1. Tuple Buffer	|▲|·size(▲)	
2. Aggregate Tree	|▲|·size(▲) +(|▲|−1)·size(●)	
3. Agg. Buckets	|win|·size(●) +|win|·size(␣)	
4. Tuple Buckets	|win|·[avg(▲ per win.) ·size(▲) +size(␣)]	
5. Lazy Slicing	|⬬|·size(⬬)	
6. Eager Slicing	|⬬|·size(⬬) +(|⬬|−1)·size(●)	
7. Lazy Slicing on tuples	|▲|·size(▲) +|⬬|·size(⬬)	
8. Eager Slicing on tuples	|▲| · size(▲) +|⬬|·size(⬬) +(|⬬|−1)·size(●)	

Legend: ▲ Tuple ● Aggregate ⬬ Slice incl. Aggregate ␣ Bucket

Table 1: Memory Usage of Aggregation Techniques.

3 WINDOW AGGREGATION CONCEPTS

In this section, we survey concepts for streaming window aggregation and give an intuition for each solution's memory usage, throughput, and latency. We provide a detailed comparison of all concepts in our experiments. Techniques which support out-of-order streams store values for an *allowed lateness* (see above). In the following discussion, we refer to allowed lateness only. Techniques which do not process out-of-order tuples, store values for the duration of the longest window.

Table 1 provides an overview of all techniques we discuss in the following subsections. We write |▲| for the number of values (i.e., tuples), |⬬| for the number of slices, and |win| for the number of windows in the allowed lateness.

3.1 Tuple Buffer

A tuple buffer (Table 1, Row 1) is a straightforward solution which does not share partial aggregates.

The *throughput* of a tuple buffer is fair as long as there are few or no concurrent windows (i.e., no window overlaps), and there are few or no out-of-order tuples. Window overlaps decrease the throughput because of repeated aggregate computations. Out-of-order tuples decrease the throughput because of memory copy operations which are required for inserting values in the middle of a sorted ring buffer.

The *latency* of a tuple buffer is high because aggregates are computed lazily when windows end. Thus, all aggregate computations contribute to the latency at the window end.

A tuple buffer stores all tuples for the allowed lateness, which is |▲|·size(▲). Thus, the more tuples we process per time, the higher the memory consumption and the higher the memory copy overhead for out-of-order tuples.

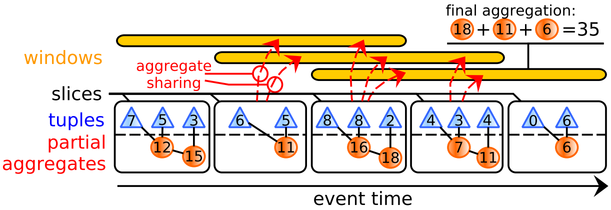


Figure 2: Example Aggregation with Stream Slicing.

3.2 Aggregate Trees

Aggregate trees such as FlatFAT [42] and B-INT [3] store partial aggregates in a tree structure and share them among overlapping windows (Table 1, Row 2). FlatFAT stores a binary tree of partial aggregates on top of stream tuples (leaves) which roughly doubles the memory consumption.

In-order tuples require $\log(|▲|)$ updates of partial aggregates in the tree. Thus, the *throughput* is decreased logarithmically when the number of tuples in the allowed lateness increases. Out-of-order tuples decrease the throughput drastically: they require the same memory copy operation as in tuple buffers. In addition, they cause a rebalancing of the aggregate tree and the respective aggregate updates.

The *latency* of aggregate trees is much lower than for tuple buffers because they can compute final aggregates for windows from pre-computed partial aggregates. Thus, only a few final aggregation steps remain when windows end [39].

3.3 Buckets

Li et al. introduce *Window-ID* (WID) [31–33], a bucket-per-window approach which is adopted by many systems with support for out-of-order processing [1, 2, 9]. Each window is represented by an independent bucket. A system assigns tuples to buckets (i.e., windows) based on event-times, independently from the order in which tuples arrive [33]. Buckets do not utilize aggregate sharing. Instead, they compute aggregates for each bucket independently.

Systems can compute aggregates for buckets incrementally [42]. This leads to very low *latencies* because the final window aggregate is pre-computed when windows end.

We consider two versions of buckets. *Tuple buckets* keep individual tuples in buckets (Table 1, Row 4). This leads to data replication for overlapping buckets. *Aggregate buckets* store partial aggregates in buckets plus some overhead (e.g., start and end times), but no tuples (Table 1, Row 3).

We prefer to store aggregates only in order to save memory. However, some use-cases (e.g., holistic aggregates over count-based windows) require us to keep individual tuples.

Buckets process in-order tuples as fast as out-of-order tuples for most use-cases: they assign the tuple to buckets and incrementally compute the aggregate of these buckets. The throughput bottleneck for buckets are overlapping windows. For example, one sliding window with $l = 20s$ and $l_s = 2s$ results in 10 overlapping windows (i.e., buckets) at any time. This causes 10 aggregation operations for each input tuple.

3.4 Stream Slicing

Slicing techniques divide (i.e., *slice*) a data stream into non-overlapping chunks of data (i.e., *slices*) [28, 30]. The system computes a partial aggregate for each slice. When windows end, the system computes window aggregates from slices.

We show stream slicing with an example in Figure 2. Slicing techniques compute partial aggregates incrementally when tuples arrive (bottom of Figure 2). We show multiple intermediate aggregates per slice to illustrate the workflow.

Partial aggregates (i.e., slices) are shared among overlapping windows which avoids redundant computations. In Figure 2, dashed arrows mark multiple uses of slices. In contrast to aggregate trees and buckets, slicing techniques require just one aggregation operation per tuple because each tuple belongs to exactly one slice. This results in a high *throughput*.

Similar to aggregate trees, the *latency* of stream slicing techniques is low because only a few final aggregation steps are required when a window ends. We consider a lazy and an eager version of stream slicing. The lazy version of stream slicing stores slices including partial aggregates (Table 1, Row 5). The eager version stores a tree of partial aggregates on top of slices to further reduce latencies (Table 1, Row 6). Both variants compute aggregates of slices incrementally when tuples arrive. The term *lazy* refers to the lazy computation of aggregates for combinations of slices.

There are usually many tuples per slice ($|●| \ll |▲|$) which leads to huge memory savings compared to aggregate trees and tuple buffers. Some use-cases such as holistic aggregates over count-based windows require us to keep individual tuples in addition to aggregates (Table 1, Row 7 and 8). In these cases, stream slicing requires more memory than tuple buffers, but saves memory compared to buckets and aggregate trees.

In this paper, we focus on stream slicing because it offers a good combination of high throughputs, low latencies, and memory savings. Moreover, our experiments show that slicing techniques scale to many concurrent windows, high ingestion rates, and high fractions of out-of-order tuples.

4 WORKLOAD CHARACTERIZATION

In this section, we identify workload characteristics which either limit the applicability of aggregation techniques or impact their performance. These characteristics are the basis for subsequent sections in which we generalize stream slicing.

4.1 Characteristic 1: Stream Order

Out-of-order streams increase the complexity of window aggregation because out-of-order tuples can require changes in the past. For example, tuple buffers and aggregate trees process in-order tuples efficiently using a ring buffer (FIFO principle) [42]. Out-of-order tuples break the FIFO principle and require memory copy operations in buffers.

We differentiate whether or not out-of-order processing is required for a use-case. For techniques which support out-of-order processing, we study how the fraction of out-of-order tuples and the delay of such tuples affect the performance.

4.2 Characteristic 2: Aggregation Function

We classify aggregation functions with respect to their algebraic properties. Our notation splits the aggregation in incremental steps and is consistent with related works [10, 42]. We write input values as lower case letters, the operation which adds a value to an aggregate as $\oplus$, and the operation which removes a value from an aggregate as $\ominus$. We first adopt three algebraic properties used by Tangwongsan et al. [42]. These properties focus on the incremental computation of aggregates:

(1) Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z) \ \ \forall \, x, y, z$
(2) Invertibility: $(x \oplus y) \ominus y = x \ \ \forall \, x, y$
(3) Commutativity: $x \oplus y = y \oplus x \ \ \forall \, x, y$

Stream slicing requires *associative* aggregate functions because it computes partial aggregates per slice which are shared among windows. This requirement is inherent for all techniques which share partial aggregates [3, 10, 28, 30, 42]. Our general slicing approach does not require invertibility or commutativity, but exploits these properties if possible to increase performance.

We further adopt the classification of aggregations in *distributive*, *algebraic*, and *holistic* [16]. Aggregations such as `sum`, `min`, and `max` are *distributive*. Their partial aggregates equal the final aggregates of partials and have a constant size. An aggregation is *algebraic* if its partial aggregates can be summarized in an intermediate result of fixed size. The final aggregate is computed from this intermediate result. The remainder of aggregations, which have an unbounded size of partial aggregates, is *holistic*.

4.3 Characteristic 3: Windowing Measure

Windows can be specified using different measures (also called *time domains* [8] or `WATTR` [31]). For example, a tumbling window can have a length of 5 minutes (time-measure), or a length of 10 tuples (count-measure). To simplify the presentation, we refer to *timestamps* in the rest of the paper. However, bear in mind that a timestamp can actually be a point in time, a tuple count, or any other monotonically increasing measure [10]:

- **Time-Based Measures:** Common time-based measures are *event-time* and *processing-time* as introduced in Section 2.
- **Arbitrary Advancing Measures** are a generalization of event-times. Typically, it is irrelevant for a stream processor if *"timestamps"* actually represent a time or another advancing measure. Examples of other advancing measures are transaction counters in a database, kilometers driven by a car, and invoice numbers.
- **Count-Based Measures** (also called *tuple-based* [31] or *tuple-driven* [8]) refer to a tuple counter. For example, a window can start at the 100th and end at the 200th tuple of a stream. Count-based measures cause challenges when combined with out-of-order processing: If tuples are ordered with respect to their event-times and a tuple arrives out-of-order, it changes the count of all other tuples which have a greater event-time. This changes the aggregates of all count-based windows which start or end after the out-of-order tuple.

If we process multiple queries which use different window-measures, timestamps are represented as vectors which contain multiple measures as dimensions. This representations allows for slicing the stream with respect to multiple dimensions (i.e., measures) while slices are still shared among all queries [10].

4.4 Characteristic 4: Window Type

We classify window types with respect to the *context* (or *state*) which is required to know where windows start and end. We adopt the classification in context free (CF), forward-context aware (FCA), and forward-context free (FCF) introduced by Li et al. [31]. Here we present those classes along with the most common window types belonging to those classes.

- **Context Free (CF).** A window type is context free if one can tell all start and end timestamps of windows without processing any tuples. Common *sliding* and *tumbling* windows are context free because we can compute all start and end timestamps a priori based on the parameters l and l_s.
- **Forward Context Free (FCF).** Windows are forward context free, if one can tell all start and end timestamps of windows up to any timestamp t, once all tuples up to this timestamp t have been processed. An example are *punctuation-based* windows where punctuations mark start and end timestamps [14]. Once we processed all tuples up to t (including out-of-order tuples), we also processed all punctuations before t and, thus, we know all start and end positions up to t.
- **Forward Context Aware (FCA).** The remaining window types are forward context aware. Such window types require us to process tuples after a timestamp t in order to know all window start and end timestamps before t. An example of such windows are *Multi-Measure Windows* which define their start and end timestamps on different measures. For example, *output the last 10 tuples (count-measure) every 5 seconds (time-measure)* is forward context aware: we need to process tuples up to a window end in order to compute the window begin.

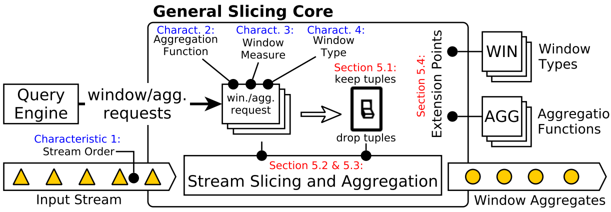


Figure 3: Architecture of General Stream Slicing

5 GENERAL STREAM SLICING

We now present our general stream slicing technique which supports high-performance aggregation for multiple queries with diverse workload characteristics. General stream slicing replaces alternative operators for window aggregation without changing their input or output semantics. Our technique minimizes the number of partial aggregates (saving memory), reduces the final aggregation steps when windows end (reducing latency), and avoids redundant computation for overlapping windows (increasing throughput). The main idea behind our technique is to exploit workload characteristics (Section 4) and to automatically adapt aggregation strategies. Such adaptivity is a highly desired feature of an aggregation framework: current non-adaptive techniques fail to support multiple window types, process in-order streams only, cannot share aggregates among windows defined on different measures, lack support for holistic aggregations, or incur dramatically reduced performance in exchange for being generally applicable.

Approach Overview. Figure 3 depicts an overview of our general slicing and aggregation technique. Users specify their queries in a high-level language such as a flavor of stream SQL or a functional API. The query translator observes the characteristics of a query (i.e., window type, aggregate function, and window measure) as well as the characteristics of input streams (in-order vs. out-of-order streams) and forwards them to our aggregator. Once those characteristics are given to our aggregator, our general slicing technique adapts automatically to the given workload characteristics.

More specifically, general slicing detects if individual tuples need to be kept in memory (to ensure generality) or if they can be dropped after computing partial aggregates (to improve performance). We further discuss this in Section 5.1. Queries can be added or removed from the aggregator and due to that, the workload characteristics can change. To this end, our aggregator adapts when one adds or removes queries. Weather we need to keep tuples in memory or not solely depends on workload characteristics. Thus, there is no need to adapt on changes in the input data streams such as a changing ratio of out-of-order tuples. When processing input tuples, the stream slicing component automatically decides when it needs to apply our three fundamental slicing operations: merge, split, and update (discussed in Section 5.2). General slicing has extension points that can be used to implement user-defined window types and aggregations (discussed in Section 5.4).

5.1 Storing Tuples vs. Partial Aggregates

General aggregation techniques [3, 42] achieve generality by storing all input tuples and by computing high-level partial aggregates. Specialized techniques, on the other hand, only store (partial) aggregates. A general slicing technique needs to decide when to store what, according to workload characteristics of each of the queries

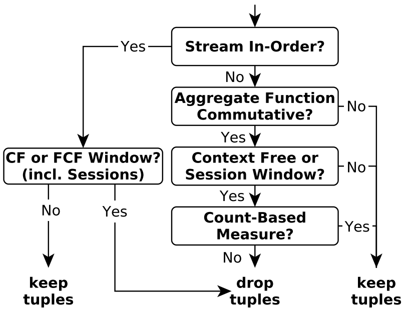

Stream In-Order?
Yes
No
Aggregate Function Commutative?
No
Yes
CF or FCF Window? (incl. Sessions)
Context Free or Session Window?
No
Yes
Count-Based Measure?
Yes
No
No
Yes
keep tuples
drop tuples
keep tuples

Figure 4: Decision Tree - Which workload characteristics require storing individual tuples in memory?

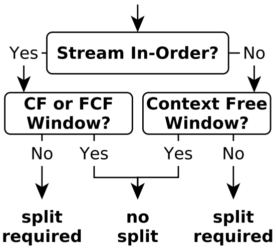


Figure 5: Decision Tree. Are splits required?

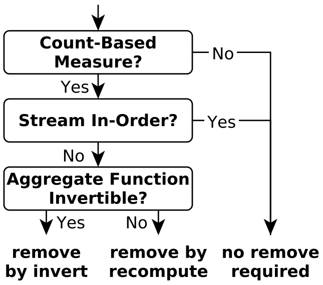


Figure 6: Decision Tree. How to remove tuples?

that it serves. In this section, we discuss how we match the performance of specialized techniques, by choosing on-the-fly whether to keep tuples for a workload or to store partial aggregates only.

For example, consider an aggregation function which is non-commutative ($\exists x, y : x \oplus y \neq y \oplus x$) defined over an unordered stream. When an out-of-order tuple arrives, we need to recompute aggregates from the source tuples in order to retain the correct order of the aggregation. Thus, one would have to store the actual tuples for possible later use. Storing all tuples for the whole duration of the allowed lateness requires more memory but allows for computing arbitrary windows from stored tuples. The decision tree in Figure 4 summarizes when storing source tuples is required depending on different workload characteristics.

In-order Streams. For in-order streams, we drop tuples for all context free and forward context free windows but must keep tuples if we process forward context aware windows. For such windows, forward context leads to additional window start or end timestamps. Thus, we must be able to compute partial aggregates for arbitrary timestamp ranges from the original stored tuples.

Out-of-order Streams. For out-of-order streams, we need to keep tuples if at least one of the following conditions is true:

(1) The aggregation function is non-commutative.
An out-of-order tuple changes the order of the incremental aggregation, which forces us to recompute the aggregate using source tuples. For in-order processing, the commutativity of aggregation functions is irrelevant because tuples are always aggregated in-order. Thus, there is no need to store source tuples in addition to partial aggregates.

(2) The window is neither context free nor a session window.
In combination with out-of-order tuples, all context aware windows require tuples to be stored. This is because out-of-order tuples change backward context which can lead to additional window start or end timestamps. Such additional start and end timestamps require to split slices and to recompute the respective partial aggregates from the original tuples. Session windows are an exception, because they are context aware, but never require recomputing aggregates [46].

(3) The query uses a count-based window measure.
An out-of-order tuple changes the count of all succeeding tuples. Thus, the last tuple of each count-based window shifts to a succeeding window.

5.2 Slice Management

Stream slicing is the fundamental concept that allows us to build partial aggregates and share them among concurrently running queries and overlapping windows. In this section, we introduce three fundamental operations which we can perform on slices.

Slice Metadata. A slice stores its start timestamp ($t_{\texttt{start}}$), its end timestamp (t_{end}), and the timestamp of the first ($t_{\texttt{first}}$) and last tuple it contains ($t_{\texttt{last}}$). Note that the timestamps of the first and last tuples do not need to conincide with the start and end timestamps of a slice. For instance, consider a slice A that starts at $t_{\texttt{start}}(A) = 1$ and ends at $t_{\text{end}}(A) = 10$ but the first (earliest) tuple contained is timestamped as $t_{\texttt{first}}(A) = 2$ and its last/latest one as $t_{\texttt{last}}(A) = 9$. We remind the reader that the *timestamp* can refer not only to actual time, but to any measure presented in Section 4.3.

We identify three fundamental operations which we perform on stream slices. These operations are *i*) *merging* of two slices into one, *ii*) *splitting* one slice into two, and *iii*) *updating* the state of a slice (i.e., aggregate and metadata updates). In the following paragraphs, we discuss `merge`, `split`, and `update` as well as the impact of our workload characteristics on each operation. We use upper case letters to name slices and corresponding lower case letters for slice aggregates.

Merge. Merging two slices A and B happens in three steps:

1. Update the end of A such that $t_{\text{end}}(A) \leftarrow t_{\text{end}}(B)$.
2. Update the aggregate of A such that $a \leftarrow a \oplus b$.
3. Delete slice B, which is now merged into A.

Steps one and three have a constant computational cost. The complexity of the second step ($a \leftarrow a \oplus b$) depends on the type of aggregate function. For instance, the cost is constant for algebraic and distributive functions such as `sum`, `min`, and `avg` because they require just a few basic arithmetic operations. Holistic functions such as quantiles can be more complex to compute. Except from the type of aggregation function, no other workload characteristics impact the complexity of the merge operation. However, stream order and window types influence when and *how often* we merge slices. We discuss this influence in Section 5.3.

Split. Splitting a slice A at timestamp t requires three steps:

1. Add slice B: $t_{\texttt{start}}(B) \leftarrow t + 1$ and $t_{\text{end}}(B) \leftarrow t_{\text{end}}(A)$.
2. Update the end of A such that $t_{\text{end}}(A) \leftarrow t$.
3. Recompute the aggregates of A and B.

Note that splitting slices is an expensive operation because it requires recomputing slice aggregates from scratch. Moreover, if splitting is required, we need to keep individual tuples in memory to enable the recomputation.

We show in Figure 5 when split operations are required. For in-order streams, only forward context aware (FCA) windows require split operations. For such windows, we split slices according to a window's start and end timestamp as soon as we process the required forward context. In out-of-order data streams, all context aware windows require split operations because out-of-order tuples possibly contain relevant backward context. We never split slices for context free windows such as tumbling and sliding ones.

Update. Updating a slice can involve adding in-order tuples, adding out-of-order tuples, removing tuples, or changing metadata ($t_{\texttt{start}}$, t_{end}, $t_{\texttt{first}}$, and $t_{\texttt{last}}$).

Metadata changes are simple assignments of new values to the existing variables. Adding a tuple to a slice requires one incremental aggregation step ($\oplus$), with the exception of processing out-of-order tuples with a non-commutative aggregation function.

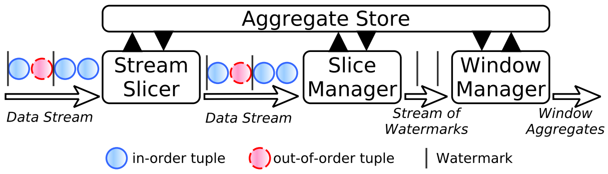


Figure 7: The Stream Slicing and Aggregation Process

For this, we recompute the aggregate of the slice from scratch to retain the order of aggregation steps.

For some workloads we need to remove tuples from slices. We show in Figure 6 when and how we remove tuples from slices. Generally, a remove operation is required only if a window is defined on a count-based measure and if we process out-of-order tuples. An out-of-order tuple changes the count of all succeeding tuples. This requires us to shift the last tuple of each slice one slice further starting at the slice of the out-of-order tuple. If the aggregation function is invertible, we exploit this property by performing an incremental update. Otherwise, we have to recompute the slice aggregate from scratch. If the out-of-order tuple has a small delay, such that it still belongs to the latest slice, we can simply add the tuple without performing a remove operation.

5.3 Processing Input Tuples

The stream slicing and aggregation logic (bottom of Figure 3) consists of four components which we show in Figure 7. The Aggregate Store is our shared data structure which is accessed by the Stream Slicer to create new slices, by the Slice Manager to update slices, and by the Window Manager to compute window aggregates.

The input stream can contain in-order tuples, out-of-order tuples, and watermarks. Note that in-order tuples can either arrive from an in-order stream (i.e., one that is guaranteed to never contain an out-of-order tuple) or from an out-of-order stream (i.e., one that does not guarantee in-order arrival). If the the stream is in-order (i.e., all tuples are in-order tuples), there is no need to ingest watermarks. Instead, we output windows directly since there is no need to wait for potentially delayed tuples.

Step 1 - The Stream Slicer. The Stream Slicer initializes new slices on-the-fly when in-order tuples arrive [28]. In an in-order stream, it is sufficient to start slices when windows start [10]. In an out-of-order stream, we also need to start slices when windows end to allow for updating the last slice of windows later on with out-of-order tuples. We always cache the timestamp of the next upcoming window edge and compare in-order tuples with this timestamp. As soon as the timestamp of a tuple exceeds the cached timestamp, we start a new slice and cache the timestamp of the next edge. This is highly efficient because the majority of tuples do not end a slice and require just one comparison of timestamps.

The Stream Slicer does not process out-of-order tuples and watermarks but forwards them directly to the Slice Manager. This is possible because the slices for out-of-order tuples have already been initialized by previous in-order tuples.

Step 2 - The Slice Manager. The Slice Manager is responsible for triggering all `split`, `merge`, and `update` operations on slices.

First, the Slice Manager checks whether a `merge` or `split` operation is required. We always merge and split slices such that all slice edges match window edges and vice versa. This guarantees that we maintain the minimum possible number of slices [10, 46].

In an out-of-order stream, context aware windows can cause `merge`s or `split`s. In an in-order stream, only forward context aware windows can cause these operations. Context free windows never require `merge` or `split` operations, as the window edges are known in advance and slices never need to change.

In-order tuples can be part of the forward context which indicates window start or end timestamps earlier in the stream. When processing forward context aware windows, we check if the new tuple changes the context such that it introduces or removes window start or end timestamps. In such case, we perform the required `merge` and `split` operation to match the new slice and window edges. Out-of-order tuples can change forward and backward context such that a `merge` operation or `split` operation are required.

If the new context causes new window edges and, thus, `merge` or `split` operations, we notify the Window Manager which outputs window aggregates up to the current watermark.

Finally, the Slice Manager adds the new tuple to its slice and updates the slice aggregate accordingly. In-order tuples always belong to the current slice and are added with an incremental aggregate update [42]. For out-of-order tuples, we look up the slice which covers the timestamp of the out-of-order tuple and add the tuple to this slice. For commutative aggregation functions, we add the new tuple with an incremental aggregate update. For non-commutative aggregation functions, we need to recompute the aggregate from individual tuples to retain the correct order.

Step 3 - The Window Manager. The Window Manager computes the final aggregates for windows from slice aggregates.

When processing an in-order stream, the Window Manager checks if the tuple it processes is the last tuple of a window. Therefore, each tuple can be seen as a watermark which has the timestamp of the tuple. If a window ended, the window manager computes and outputs the window aggregate (final aggregation step).

For out-of-order streams, we wait for the watermark (see Section 2) before we output results of windows which ended before a watermark.

The Slice Manager notifies the Windows Manager when it performs `split`, `merge`, or `update` operation on slices. Upon such notification, the Window Manager performs two operations:

(1) If an out-of-order tuple arrives within the allowed lateness but after the watermark, the tuple possibly changes aggregates of windows which were output before. Thus, the Window Manager outputs updates for these window aggregates.
(2) If a tuple changes the context of context aware windows such that new windows end before the current watermark, the window manager computes and outputs the respective aggregates.

Parallelization. We parallelize stream processing with key partitioning which is the common approach used in stream processing systems [21] such as Flink [9], Spark [4], and Storm [44]. Key partitioning enables intra-node as well as inter-node parallelism and, thus, results in good scalability. Since our generic window aggregation is a drop in replacement for the window aggregation operator, the input and output semantics of the operator remains unchanged. Thus, neither the query interface nor optimizations unrelated to window aggregations are affected.

5.4 User-Defined Windows and Aggregations

Our architecture decouples the general logic of stream slicing from the concrete implementation of window types and aggregation functions. This makes it easy to add window types and aggregation functions as no changes are required in the slicing logic. In this section, we describe how we implement aggregation functions and window types.

5.4.1 *Implementing Aggregation Functions.*

We adopt the same approach of incremental aggregation introduced by Tangwongsan et al. [42]. Each aggregation type consists of three functions: `lift`, `combine`, and `lower`. In addition, aggregations may implement an invert function. We now discuss the concept behind these functions, and refer the reader to the original paper for an overview of different aggregations and their implementation.

Lift. The lift function transforms a tuple to a partial aggregate. For example, consider an average computation. If a tuple $\langle t, v \rangle$ contains its timestamp t and a value v, the lift function will transform it to $\langle \text{sum} \leftarrow v, \text{count} \leftarrow 1 \rangle$, which is the partial aggregate of that one tuple.
Combine. The combine function ($\oplus$) computes the combined aggregate from partial aggregates. Each incremental aggregation step results in one call of the combine function.
Lower. The lower function transforms a partial aggregate to a final aggregate. In our example, the lower function computes the average from sum and count: $\langle \text{sum}, \text{count} \rangle \mapsto \text{sum/count}$.
Invert. The optional invert function removes one partial aggregate from another with an incremental operation.

In this work, we consider holistic aggregation functions which have an unbounded size of partial aggregates. A widely used holistic function is the computation of quantiles. For instance, windowed quantiles are the basis for billing models of content delivery networks and transit-ISPs [13, 23]. For quantile computations, we sort tuples in slices to speed up succeeding merge operations and apply run length encoding to save memory [37].

5.4.2 Implementing Different Window Types. We use a common interface for the in-order slicing logic of all windows. We extend this interface with additional methods for context-aware windows. One can add additional window types by implementing the respective interface.

Context Free Windows. The slicing logic for context free windows depends on in-order tuples only. When a tuple is processed, the slicing core initializes all slices up to the timestamp of that tuple. Our interface for context free windows has two methods: The first method has the signature `long getNextEdge(long timestamp)`. It receives a timestamp as parameter and returns the next window edge (begin or end timestamp) after this timestamp. We use this method to retrieve the next window edge for on-the-fly stream slicing (Step 1 in Section 5.3). For example, a tumbling window with length l would return $timestamp + l - (timestamp \bmod l)$.

The second method triggers the final window aggregation according to a watermark and has the following signature:
`void triggerWin(Callback c, long prevWM, long currWM)`.
The Window Manager calls this method when it processes a watermark. c is a callback object, prevWM is the timestamp of the previous watermark and currWM is the timestamp of the current watermark. The method reports all windows which ended between prevWM and currWM by calling `c.triggerWin(long startTime, long endTime)`. This callback to the Window Manager triggers the computation and output of the final window aggregate.
Context Aware Windows. Context aware windows use the same interface as context free windows to trigger the initialization of slices when processing in-order tuples. In addition, context aware windows require to keep a state (i.e., context) in order to derive window start and end timestamps when processing out-of-order tuples. We initialize context aware windows with a pointer to the Aggregate Store. This prevents redundancies among the state of the shared aggregator and the window state. When the Slice Manager processes a tuple, it notifies context aware windows by calling `window.notifyContext(callbackObj, tuple)`. This method can then add and remove window start and end timestamps through the callback object and the Slice Manager splits and merges slices as required to match window start and end timestamps. We detect whether or not a window is context aware based on the interface which is implemented by the window specification. We provide examples for different context free and context aware window implementations in our open source repository.

6 EVALUATION

In this section, we evaluate the performance of general stream slicing and compare stream slicing with alternative techniques introduced in Section 3.

6.1 Experimental Setup

Setup. We implement all techniques on Apache Flink v1.3. We run our experiments on a VM with 6 GB main memory and 8 processing cores with 2.6 GHz.
Metrics. In our experiments, we report throughput, latency, and memory consumption. We measure throughput as in the Yahoo Streaming Benchmark implementation for Apache Flink [12, 48]. We determine latencies with the JMH benchmarking suite [35]. JMH provides precise latency measurements on JVM-based systems. We use the *ObjectSizeCalculator* of Nashorn to determine memory footprints [36].
Baselines. We compare an eager and a lazy version of general stream slicing with non-slicing techniques from Section 3: As representative for aggregate trees, we implement FlatFAT [42]. For the buckets technique, we use the implementation of Apache Flink [9]. For tuple buffers, we use an implementation based on a ring buffer array. We also include Pairs [28] and Cutty [10] as specialized slicing techniques where possible.
Data. We replay real-world sensor data from a football match [34] and from manufacturing machines [25]. The original data sets track the position of the football with 2000 and the machine states with 100 updates per second. We generate additional tuples based on the original data to simulate higher ingestion rates. We add 5 gaps per minute to separate sessions. This is representative for the ball possession moving from one player to another. If not indicated differently, we show results for the football data. The results for other data sets are almost identical because the performance depends on workload characteristics rather than data characteristics.
Queries. We base our queries (i.e., window length, slide steps, etc.) on the workload of a live-visualization dashboard which is built for the football data we use [45]. If not indicated differently, we use the sum aggregation in Sections 6.2 and Section 6.3. In Section 6.4, we use the M4 aggregation technique [26] to compress the data stream for visualization. M4 computes four algebraic aggregates per window (i.e., minimum, maximum, first and last value of each window). We show in Section 6.3.2 how the performance differs among diverse aggregation functions. Because we do not change the input and output semantics of the window and aggregation operation, there is no impact on upstream or downstream operations. We ensure that windowing and aggregation are the bottleneck and, thus, we measure the performance of aggregation techniques.

We do not alternate between tumbling and sliding windows because they lead to identical performance: For example, 20 concurrent tumbling window queries cause 20 concurrent windows (1 window for each query at any time). This is equivalent to a single sliding window with a window length of 20 seconds and and a slide step of one second (again 20 concurrent windows). In the following, we refer to *concurrent windows* instead of *concurrent tumbling window queries. Sliding window queries* yield identical results if they imply the same number of *concurrent windows.*
Structure. We split our evaluation in three parts. First, we compare stream slicing and alternative approaches with respect to their throughput, latency, and memory footprint (Section 6.2). Second, we study the impact of each workload characteristic introduced in Section 4 (Section 6.3). Third, we integrate general slicing in

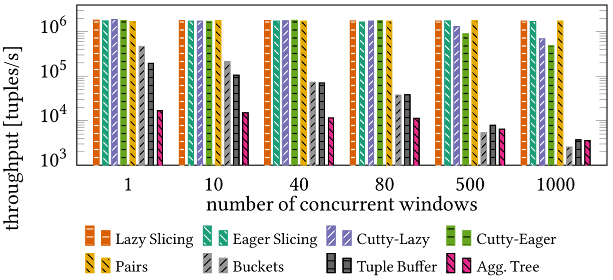
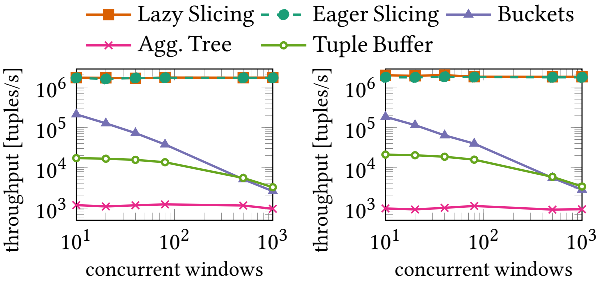


Figure 8: In-order Processing with Context Free Windows.



(a) Football data set [34]. **(b) Machine data set [25].**

Figure 9: Increasing the number of concurrent windows including 20% out-of-order tuples and session windows.

Apache Flink and show the performance gain for a concrete application (Section 6.4). Sections 6.2 and 6.3 focus on the performance per operator instance. Section 6.4 studies the parallelization.

6.2 Stream Slicing Compared to Alternatives

We now compare stream slicing with alternative techniques discussed in Section 3. We first study the throughput for in-order processing on context-free windows in Section 6.2.1. Our goal is to understand the performance of stream slicing compared to alternative techniques, including specialized slicing techniques. In Section 6.2.2, we evaluate how the throughput changes in the presence of out-of-order tuples and context-aware windows. In Section 6.2.3, we evaluate the memory footprint and in Section 6.2.4 the latency of different techniques.

6.2.1 Throughput.

Workload. We execute multiple concurrent tumbling window queries with equally distributed lengths from 1 to 20 seconds. These window lengths are representative of window aggregations which facilitate plotting line charts at different zoom levels (Application of Section 6.3). We chose Pairs [28] and Cutty [10] as example slicing techniques because Pairs is one of the first and most cited techniques and Cutty offers a high generality with respect to window types.

Results. We show our results in Figure 8. All three slicing techniques process millions of tuples per second and scale to large numbers of concurrent windows.

Buckets achieves orders of magnitude less throughput than Slicing techniques and does not scale to large numbers of concurrent windows. The reason is that we must assign each tuple to all concurrent buckets (i.e., windows). Thus, tuples belong to up to 1000 buckets causing 1000 redundant aggregation steps per tuple. In contrast, slicing techniques always assign tuples to exactly one slice. Similar to buckets, the tuple buffer causes redundant aggregation steps for each window as we compute each window independently. Aggregate Trees show a throughput which is orders of magnitude smaller than the one of slicing techniques. This is because each tuple requires several updates in the tree.

Summary. We observe that slicing techniques outperform alternative concepts with respect to throughput and scale to large numbers of concurrent windows.

6.2.2 Throughput under Constraints. We now analyze the throughput under constraints, i.e., including out-of-order tuples and context-aware windows.

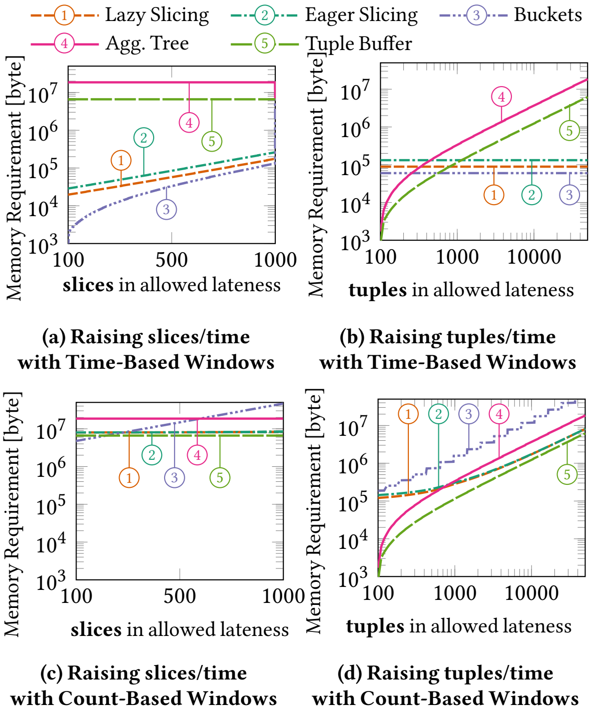
Workload. The workload remains the same as before but we add a time-based session window (l_g = 1sec.) as representative for a context-aware window. We add 20% out-of-order tuples with random delays between 0 and 2 seconds.

Results. We show the results in Figure 9. Slicing techniques achieve an order of magnitude higher throughput than alternative techniques which do not use stream slicing. Moreover, slicing scales to large numbers of concurrent windows with almost constant throughput. This is because the per-tuple complexity remains constant: we assign each tuple to exactly one slice. Lazy Slicing has the highest throughput (1.7 Million tuples/s) because it uses stream slicing and does not compute an aggregate tree. Eager Slicing achieves slightly lower throughput than Lazy Slicing. This is due to out-of-order tuples which cause updates in the aggregate tree. Buckets show the same performance decrease as in the previous experiment. The performance decrease for the Tuple Buffer is intensified due to out-of-order inserts in the ring buffer array. Aggregate Trees process less than 1500 tuples/s with 20% out-of-order tuples. This is because out-of-order tuples require expensive leaf inserts in the aggregate tree (rebalance and update of inner nodes). Eager slicing seldom faces this issue because it stores slices instead of tuples in the aggregate tree. The majority of out-of-order tuples falls in an existing slice which prevents rebalancing. We exemplary show our results on two different datasets for this experiment. Because the performance depends on workload characteristics rather than data characteristics, the results are almost identical. We omit similar results for different data sets in the following experiments and focus on the impact of workload characteristics.

Summary. For workloads including out-of-order tuples and context-aware windows, we observe that general stream slicing outperforms alternative concepts with respect to throughput and scales to large numbers of concurrent windows.

6.2.3 Memory Consumption. We now study the memory consumption of different techniques with four plots: In Figures 10a and 10c, we vary the number of slices in the allowed lateness and fix the number of tuples in the allowed lateness to 50 thousand. In Figures 10b and 10d, we vary the number of tuples and fix the number of slices to 500. We experimentally compare time-based and count-based windows. Our measurements include all memory required for storing partial aggregates and metadata such as the start and end times of slices.

Results for Time-Based Windows. Figures 10a and 10b show the memory consumption for time-based windows, which do not require to store individual tuples. For Stream Slicing and Buckets, the memory footprint increases linearly with the number of slices in the allowed lateness. The memory footprint is independent from the number of tuples. The opposite holds for Tuple Buffers and Aggregate Trees. Slicing techniques store just one partial aggregate per slice, while buckets store one partial aggregate per window. Tuple Buffers and Aggregate Trees store each tuple individually.



(a) Raising slices/time with Time-Based Windows

(b) Raising tuples/time with Time-Based Windows

(c) Raising slices/time with Count-Based Windows

(d) Raising tuples/time with Count-Based Windows

Figure 10: Memory Experiments with Unordered Streams.

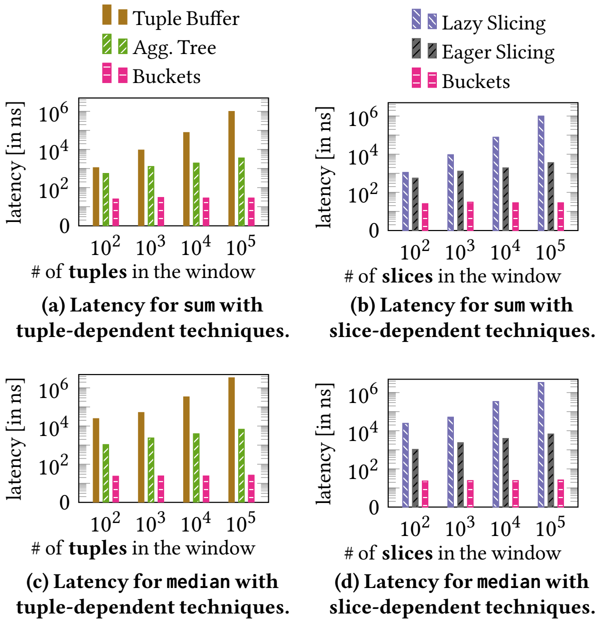
Results for Count-Based Windows. Figures 10c and 10d show the memory consumption for count-based windows, which require individual tuples to be stored. The experiment setup is the same as in Figures 10a and 10b.

The memory consumption of all techniques increases with the number of tuples in the allowed lateness because we need to store all tuples for processing count-based windows on out-of-order streams (Figure 10d). Starting from 1000 tuples in the allowed lateness, the memory consumed by tuples dominates the overall memory requirement. Accordingly, all curves become linear and parallel. Buckets show a stair shape because of the underlying hash map implementation [49]. Slicing techniques start at roughly 10^5 byte which is the space required to store 500 slices. The memory footprint of buckets also increases with the number of slices because more slices correspond to more window buckets (Figure 10c). Each bucket stores all tuples it contains which leads to duplicated tuples for overlapping buckets.

Summary. When we can drop individual tuples and store partial aggregates only (Figure 10a and 10b), the memory consumptions of slicing and buckets depends only on the number of slices in the allowed lateness. In this case, stream slicing and buckets scale to high ingestion rates with almost constant memory utilization. If we need to keep individual tuples (Figure 10c and 10d), storing tuples dominates the memory consumption.

6.2.4 Latency. The output latency for window aggregates depends on the aggregation technique, the number of entries (tuples or slices) which are stored, and the aggregation function. In Figure 11, we show the latency for different situations.

Distributive and Algebraic Aggregation. For the sum aggregation (Figure 11a), Lazy Slicing and Tuple Buffer exhibit up to 1ms latency for 10^5 entries (no matter if 10^5 tuples or 10^5 slices). Eager Slicing and Aggregate Trees show latencies below 5μs. Buckets achieve latencies below 30ns. Lazy aggregation has higher latencies because it computes final aggregates upon request. Eager Aggregation uses precomputed partial aggregates from an aggregate tree which reduces the latency. Buckets pre-compute the final aggregate of each window and store aggregates in a hash map which leads to the lowest latency.



(a) Latency for sum with tuple-dependent techniques.

(b) Latency for sum with slice-dependent techniques.

(c) Latency for median with tuple-dependent techniques.

(d) Latency for median with slice-dependent techniques.

Figure 11: Output Latency of Aggregate Stores

Holistic Aggregation. The latencies for the holistic median aggregation (Figure 11c) are in the same order of magnitude and follow the same trends. Buckets exhibit the same latencies as before because they precompute the aggregate for each bucket. Thus, a more complex holistic aggregation decreases the throughput but does not increase the latency. The latency of slicing techniques increases for the median aggregation because we combine partial aggregates to final aggregates when windows end. This combine step is more expensive for holistic aggregates than for algebraic ones.

Summary. We observe a trade-off between throughput and latency. Lazy aggregation has the highest throughput and the highest latency. Eager aggregation has a lower throughput but achieves microsecond latencies. Buckets provide latencies in the order of nanoseconds but have an order of magnitude less throughput.

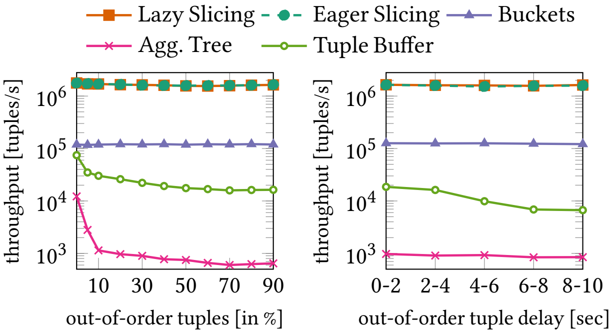
6.3 Studying Workload Characteristics

We measure the impact of the workload characteristics from Section 4 on the performance of general slicing. For comparison, we also show the best alternative techniques.

6.3.1 Impact of Stream Order. In this experiment, we investigate the impact of the amount of out-of-order tuples and the impact of the delay of out-of-order tuples on throughput (Figure 12). We use the same setup as for the throughput experiments in Section 6.2.2 with 20 concurrent windows.

Out-of-order Performance. In Figure 12a, we increase the fraction of out-of-order tuples. Slicing and Buckets process out-of-order tuples as fast as in-order tuples. The throughput of the other techniques decreases when processing more out-of-order tuples.

Slicing techniques process out-of-order tuples efficiently because they perform only one slice update per out-of-order tuple. Eager slicing also updates its aggregate tree. This update has a low overhead because there are just a few hundred slices in the allowed lateness and, accordingly, there are just a few tree levels



(a) Increasing the fraction of out-of-order tuples.

(b) Increasing the delay of out-of-order tuples.

Figure 12: Impact of Stream Order on the Throughput.

which require updates. Aggregate Trees on tuples have a much larger number of tree levels because they store tuples instead of slices as leaf nodes.

Buckets have a constant throughput as in the previous experiments. Tuple Buffers and Aggregate Trees exhibit a throughput decay when processing out-of-order tuples. Tuple Buffers require expensive out-of-order inserts in the sorted buffer array. Aggregate Trees require inserting past leaf nodes in the aggregate tree. This causes a rebalancing of the tree and the respective recomputation of aggregates. Eager Slicing seldom faces this issue (see Section 6.2.2).

Delay Robustness. In Figure 12b, we increase the delay of out-of-order tuples. We use equally distributed random delays within the ranges specified on the horizontal axis.

All techniques except Tuple Buffers are robust against increasing delays. Slicing techniques always update one slice when they process a tuple. Small delays can sightly increase the throughput compared to longer delays if out-of-order tuples still belong to the most recent slice. In this case, we require no lookup operations to find the correct slice. The throughput of Buckets is independent of the delay because Flink stores buckets in a hashmap. The throughput of the tuple buffer decreases with increasing delay of out-or-order tuples, because the lookup and update costs in the sorted buffer array increase.

Summary. Stream slicing and Buckets scale with constant throughput to large fractions of out-of-order tuples and are robust against high delays of these tuples.

6.3.2 Impact of Aggregation Functions.

We now study the throughput of different aggregation functions using the same setup as before (20 concurrent windows, 20% out-of-order tuples, delays between 0 and 2 seconds) in Figure 13. We differentiate time-based and count-based windows to show the impact of invertibility. We implement the same aggregation functions as Tangwongsang et al. [42]. The original publication provides a discussion of these functions and an overview of their algebraic properties. We additionally study the median and the 90-percentile as examples for holistic aggregation. Moreover, we study a naive version of the sum aggregation which does not use the invertibility property. This allows for making a deduction with respect to not invertible aggregations in general.

Time-Based Windows. For time-based windows, the throughput is similar for all algebraic and distributive aggregations with small differences due to different computational complexities of the aggregations. Holistic aggregations (median and 90-percentile) show a much lower throughput because they require to keep all tuples in memory and have a higher complexity.

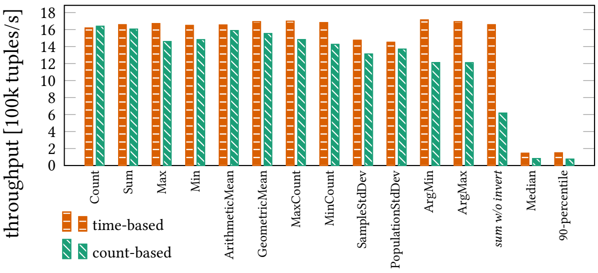
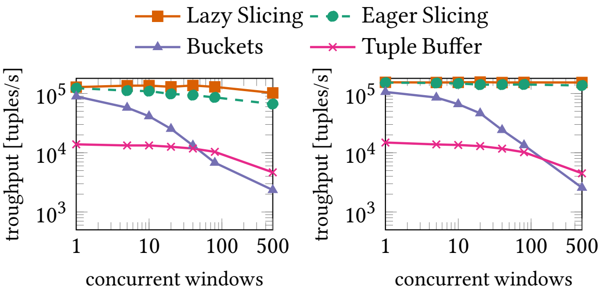


Figure 13: Impact of Aggregation Types on Throughput.



(a) Football data set [34].

(b) Machine data set [25].

Figure 14: Throughput for Median Aggregation.

Count-Based Windows. We observe lower throughputs than for time-based windows, which is because of out-of-order tuples. For count-based windows, an out-of-order tuple changes the sequence id (count) of all later tuples. Thus, we need to shift the last tuple of each slice to the next slice. This operation has low overhead for invertible aggregations because we can subtract and add tuples from aggregates. The operation is costly for not invertible aggregations because it requires the recomputation of the slice aggregate. Time-based windows do not require an invert operation because out-of-order tuples only change the sequence id (count) of later tuples but not the timestamps.

Impact of invertibility. There is a big difference between the performance for different not invertible aggregations on count-based windows. Although `Min`, `Max`, `MinCount`, `MaxCount`, `ArgMin`, and `ArgMax` are not invertible, they have a small throughput decay compared to time-based windows (Figure 13). This is because most invert operations do not affect the aggregate and, thus, do not require a recomputation. For example, it is unlikely that the tuple we shift to the next slice is the maximum of the slice. If the maximum remains unchanged, `max`, `MaxCount`, and `ArgMax` do not require a recomputation. In contrast, the `sum w/o invert` function shows the performance decay for a not invertible function which always requires a recomputation when removing tuples.

Impact of Holistic Aggregations. In Figure 13, we observe that holistic aggregations have a much lower throughput than algebraic and distributive aggregations. In Figure 14, we show that stream slicing still outperforms alternative approaches for these aggregations. The reason is that stream slicing prevents redundant computations for overlapping windows by sorting values within slices and by applying run length encoding. In contrast, Buckets and Tuple Buffer compute each window independently. The machine data set shows slightly higher throughputs because the aggregated column has only 37 distinct values compared to 84232 distinct values in the football dataset. Fewer distinct values increase the savings achieved by run length encoding. Aggregate trees (not shown) can hardly compute holistic aggregates. They maintain partial aggregates for all inner nodes of a large tree which is extremely expensive for holistic aggregations.

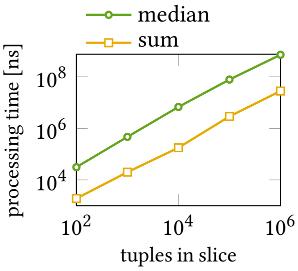


Figure 15: Processing Time for Recomputing Aggregates.

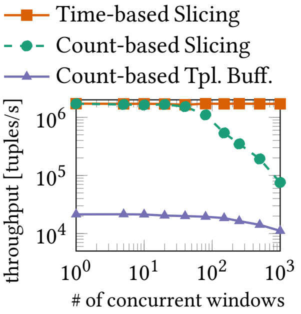


Figure 16: The Impact of Different Window Measures.

Summary. On time-based windows, stream slicing performs diverse distributive and algebraic aggregations with similarly high throughputs. Considering count-based windows and out-of-order tuples, invertible aggregations lead to higher throughputs than not invertible ones.

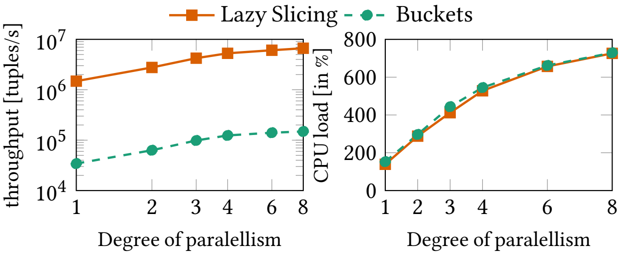
6.3.3 Impact of Window Types. The window type impacts the throughput if we process context-aware windows because these windows potentially require split operations. Note that context aware windows cover arbitrary user-defined windows which makes it impossible to provide a general statement on the throughput for all these windows. Thus, we evaluate the time required to recompute aggregates for slices of different sizes when a split operation is performed (Figure 15). Given a context aware window, one can estimate the throughput decay based on the number of split operations required and the time required for recomputing aggregates after splits. We show the sum aggregation as representative for an algebraic function and the median as example for a holistic function.

The processing time for the recomputation of an aggregate increases linearly with the number of tuples contained in the aggregate. If split operations are required to process a context aware window, a system should monitor the overhead caused by split operations and adjust the maximum size of slices accordingly. Smaller slices require more memory and cause repeated aggregate computation when calculating final aggregates for windows. In exchange, the aggregates of smaller slices are cheaper to recompute when we split slices.

6.3.4 Impact of Window Measures. We compare different window measures in Figure 16. We use the same setup as before (20% out-of-order tuples with delays between 0 and 2 seconds).

Time-Based Windows. For time-based windows, the throughput is independent from the number of concurrent windows as discussed in our throughput analysis in Section 6.2.2. The throughput for arbitrary advancing measures is the same as for time-based measures because they are processed identically [10].

Count-Based Windows. The throughput for count-based windows is almost constant for up to 40 concurrent windows and decays linearly for larger numbers. For up to 40 concurrent windows, most slices are larger than the delay of tuples. Thus, out-of-order tuples still belong to the current slice and require no slice updates. The more windows we add, the smaller our slices become. Thus, out-of-order tuples require an increasing number of updates for shifting tuples between slices which reduces the throughput. Tuple buffers are the fastest alternative to Slicing in our experiment. For 1000 concurrent windows, slicing is still an order of magnitude faster than tuple buffers.



(a) Throughput. **(b) Processor Load.**

Figure 17: Parallelizing the workload of a live-visualization dashboard (80 concurrent windows per operator instance).

Summary. The throughput of time-based windows stays constant whereas the throughput of count-based windows decreases with a growing number of concurrent windows.

6.4 Parallel Stream Slicing

In this experiment, we study stream slicing on the example of our dashboard application [45] which uses the M4 aggregation [26]. We vary the degree of parallelism to show the scalability with respect to the number of cores. We compare Lazy Slicing with Buckets which are used in Flink.

Results. In Figure 17, we increase the number of parallel operator instances of the windowing operation (degree of parallelism). The throughput scales linearly up to a degree of parallelism of four (Figure 17a). Up to this degree, each parallel operator instance runs on a dedicated core with other tasks (data source operator, writing outputs, operating system overhead, etc.) running on the remaining four cores. For higher degrees of parallelism the throughput and the CPU load increase logarithmically, approaching the full 800% CPU utilization (Figure 17b). Slicing achieves an order of magnitude higher throughput than buckets, because it prevents assigning tuples to multiple buckets (cf. Section 6.2.1). The memory consumption scaled linearly with the degree of parallelism for both techniques.

Summary. We conclude that stream slicing and buckets scale linearly with the number of cores for our application.

7 RELATED WORK

Optimizing Window Aggregations. Our general slicing techniques utilizes features of existing techniques such as on-the-fly slicing [28], incremental aggregation [42], window grouping [18, 19], and user-defined windows [10]. However, general stream slicing offers a unique combination of generality and performance. We base our general slicing implementation on a specialized technique which we presented earlier as a poster [46]. One can extend other slicing techniques based on this paper to reach similar generality and performance. Existing slicing techniques such as Pairs [28] and Panes [30] are limited to tumbling and sliding windows. Cutty can process user-defined window types, but does not support out-of-order processing [10]. Several publications optimize sliding window aggregations focusing on different aspects such as incremental aggregation [6, 15, 42] or worst-case constant time aggregation [43]. Hirzel et al. conclude that one needs to decide on a concrete algorithm based on the aggregation, window type, latency requirements, stream order, and sharing requirements because each specialized algorithm addresses a different set of requirements [22]. Instead of alternating between different algorithms, we provide a single solution which is generally applicable and allows for adding aggregation functions and window types without changing the core of our technique.

Stream Processing in Batches. In contrast to our techniques, which adopts a tuple-at-a-time processing approach, several works split streams in batches of data which they process in parallel [5, 27, 52]. SABER introduces *window fragments* to decouple *slide* and *range* of sliding windows from the batch size [27]. However, in contrast to our work, SABER does not consider aggregate sharing among queries. Balkesen et al. use panes to share aggregates among overlapping windows [5]. None of these works addresses the general applicability with respect to workload characteristics.

Complementary Techniques. Weaving optimizes execution plans to reduce the overall computation costs for concurrent window aggregate queries [18, 19, 38]. We use a similar approach to fuse window aggregation queries when window edges match. This optimization is orthogonal to the generalization of slicing which is the focus of this paper. Huebsch et al. study multiple query optimization when aggregating several data streams which arrive at different nodes [24]. General stream slicing complements this work with an increased per-node performance. Truviso proposes an alternative technique based on independent stream partitions to correct outputs when tuples arrive after the watermark [29]. While our work focuses on slicing streams and computing partial aggregations for slices, recent publications of Shein et al. further accelerate the final aggregation step which is required when windows end [39, 40]. Trill [11] is an analytics system that supports streaming, historical, and exploratory queries in the same system. Trill supports incremental aggregation and performs aggregations on snapshots, the state of the window at a certain point in time. Zeuch et al. [53] integrate stream slicing in a lock-free window aggregation operator to optimize throughput on modern hardware.

8 CONCLUSION

Stream slicing is a technique for streaming window aggregation which provides high throughputs and low latencies with a small memory footprint. In this paper, we contribute a generalization of stream slicing with respect to four key workload characteristics: Stream (dis)order, aggregation types, window types, and window measures. Our general slicing technique dynamically adapts to these characteristics, for example, by exploiting the invertibility of an aggregation or the absence of out-of-order tuples.

Our experimental evaluation reveals that general slicing is highly efficient without limiting generality. It scales to a large number of concurrent windows, and consistently outperforms state-of-the-art techniques in terms of throughput. Furthermore, it efficiently supports application scenarios with large fractions of out-of-order tuples, tuples with high delays, time-based and count-based window measures, context-aware windowing, and holistic aggregation functions. Finally, we observed that the throughput of general slicing scales linearly with the number of processing cores.

Acknowledgements: This work was funded by the EU project E2Data (780245), Delft Data Science, and the German Ministry for Education and Research as BBDC I (01IS14013A) and BBDC II (01IS18025A).

REFERENCES

[1] Tyler Akidau et al. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. In *PVLDB*.
[2] Apache Beam. 2018. An advanced unified programming model. https://beam.apache.org/ (project website).
[3] Arvind Arasu et al. 2004. Resource sharing in continuous sliding-window aggregates. *VLDB*.
[4] Michael Armbrust et al. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *SIGMOD*. ACM, 601–613.
[5] Cagri Balkesen et al. 2011. Scalable data partitioning techniques for parallel sliding window processing over data streams. In *DMSN*.
[6] Pramod Bhatotia et al. 2014. Slider: incremental sliding window analytics. In *ACM/IFIP/USENIX Middleware*.
[7] Brice Bingman. 2018. Poor performance with Sliding Time Windows. In *Flink Jira Issues (issues.apache.org/jira/browse/FLINK-6990)*.
[8] Irina Botan et al. 2010. SECRET: a model for analysis of the execution semantics of stream processing systems. In *PVDLB*.
[9] Paris Carbone et al. 2015. Apache Flink: Stream and batch processing in a single engine. *IEEE CS* (2015).
[10] Paris Carbone et al. 2016. Cutty: Aggregate Sharing for User-Defined Windows. In *CIKM*.
[11] Badrish Chandramouli et al. 2014. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *PVLDB* 8, 4 (2014), 401–412.
[12] Sanket Chintapalli et al. 2016. Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In *IPDPS*.
[13] Xenofontas Dimitropoulos et al. 2009. On the 95-percentile billing method. In *PAM*.
[14] Buğra Gedik. 2014. Generic windowing support for extensible stream processing systems. *SPE* (2014).
[15] Thanaa Ghanem et al. 2007. Incremental evaluation of sliding-window queries over data streams. In *TKDE*.
[16] Jim Gray et al. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *DMKDFD* (1997).
[17] Michael Grossniklaus et al. 2016. Frames: data-driven windows. In *DEBS*.
[18] Shenoda Guirguis et al. 2011. Optimized processing of multiple aggregate continuous queries. In *CIKM*.
[19] Shenoda Guirguis et al. 2012. Three-Level Processing of Multiple Aggregate Continuous Queries. In *IEEE ICDE*.
[20] Martin Hirzel et al. 2009. SPL stream processing language specification. *IBM Research Report* (2009).
[21] Martin Hirzel et al. 2014. A Catalog of Stream Processing Optimizations. *Comput. Surveys* 46 (2014).
[22] Martin Hirzel et al. 2017. Sliding-Window Aggregation Algorithms: Tutorial. In *DEBS*.
[23] Kartik Hosanagar et al. 2008. Service adoption and pricing of content delivery network (CDN) services. *INFORMS MSCIAM* (2008).
[24] Ryan Huebsch et al. 2007. Sharing aggregate computation for distributed queries. In *SIGMOD*.
[25] Zbigniew Jerzak et al. 2012. The DEBS 2012 grand challenge. In *DEBS*.
[26] Uwe Jugel et al. 2014. M4: a visualization-oriented time series data aggregation. *PVLDB* (2014).
[27] Alexandros Koliousis et al. 2016. SABER: Window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD*.
[28] Sailesh Krishnamurthy et al. 2006. On-the-fly sharing for streamed aggregation. In *SIGMOD*.
[29] Sailesh Krishnamurthy et al. 2010. Continuous analytics over discontinuous streams. In *SIGMOD*.
[30] Jin Li et al. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. In *SIGMOD Record*.
[31] Jin Li et al. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*.
[32] Jin Li et al. 2008. AdaptWID: An adaptive, memory-efficient window aggregation implementation. *IICOFX* (2008).
[33] Jin Li et al. 2008. Out-of-order processing: a new architecture for high-performance stream systems. In *PVLDB*.
[34] Christopher Mutschler et al. 2013. The DEBS 2013 grand challenge. In *DEBS*.
[35] OpenJDK. 2018. JMH Benchmarking Suite Project Website. goo.gl/TPQSDw.
[36] OpenJDK. 2018. Nashorn Project, ObjectSizeCalculator. goo.gl/962BfX.
[37] David Salomon. 2007. *Variable-length codes for data compression*. Springer Science & Business Media.
[38] Anatoli U. Shein et al. 2015. F1: Accelerating the Optimization of Aggregate Continuous Queries. In *CIKM*.
[39] Anatoli U Shein et al. 2017. Flatfit: Accelerated incremental sliding-window aggregation for real-time analytics. In *SSDBM*.
[40] Anatoli U. Shein et al. 2018. SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation. In *EDBT*.
[41] Leo Syinchwun. 2016. Lightweight Event Time Window. In *Flink Jira Issues (issues.apache.org/jira/browse/FLINK-5387)*.
[42] Kanat Tangwongsan et al. 2015. General incremental sliding-window aggregation. In *PVLDB*.
[43] Kanat Tangwongsan et al. 2017. Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time. In *DEBS*.
[44] Ankit Toshniwal et al. 2014. Storm@ twitter. In *SIGMOD*.
[45] Jonas Traub et al. 2017. I2: Interactive Real-Time Visualization for Streaming Data.. In *EDBT*.
[46] Jonas Traub et al. 2018. Scotty: Efficient Window Aggregation for out-of-order Stream Processing. In *ICDE*.
[47] Peter Tucker et al. 2003. Exploiting punctuation semantics in continuous data streams. In *TKDE*.
[48] Kostas Tzoumas et al. 2015. High-throughput, low-latency, and exactly-once stream processing with Apache Flink. (2015). goo.gl/QdTkEq.
[49] Mikhail Vorontsov. 2013. Memory consumption of popular Java data types - part 2. *Java Performance Tuning Guide* (2013). goo.gl/7CuJtf.
[50] Jark Wu. 2017. Improve performance of Sliding Time Window with pane optimization. In *Flink Jira Issues (issues.apache.org/jira/browse/FLINK-7001)*.
[51] Yuan Yu et al. 2009. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SIGOPS*.
[52] Matei Zaharia et al. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*. ACM.
[53] Steffen Zeuch et al. 2019. Analyzing Efficient Stream Processing on Modern Hardware. In *PVLDB*.

An Efficient Sliding Window Approach for Approximate Entity Extraction with Synonyms

Jin Wang
University of California, Los Angeles
jinwang@cs.ucla.edu

Chunbin Lin
Amazon AWS
lichunbi@amazon.com

Mingda Li
University of California, Los Angeles
limingda@cs.ucla.edu

Carlo Zaniolo
University of California, Los Angeles
zaniolo@cs.ucla.edu

ABSTRACT

Dictionary-based entity extraction from documents is an important task for several real applications. To improve the effectiveness of extraction, many previous studies focused on the problem of approximate dictionary-based entity extraction, which aims at finding all substrings in documents that are similar to pre-defined entities in the reference entity table. However, these studies only consider syntactical similarity metrics, such as Jaccard and edit distance. In the real-world scenarios, there are many cases where syntactically different strings can express the same meaning. For example, *MIT* and *Massachusetts Institute of Technology* refer to the same object but they have a very low value of syntactic similarity. Existing approximate entity extraction work fails to identify such kind of semantic similarity and will definitely suffer from low recall.

In this paper, we come up with the new problem of approximate dictionary-based entity extraction with synonyms and propose an end-to-end framework Aeetes to solve it. We propose a new similarity measure Asymmetric Rule-based Jaccard (JACCAR) to combine the synonym rules with syntactic similarity metrics and capture the semantic similarity expressed in the synonyms. We propose a clustered index structure and several pruning techniques to reduce the filter cost so as to improve the overall performance. Experimental results on three real world datasets demonstrate the effectiveness of Aeetes. Besides, Aeetes also achieves high performance in efficiency and outperforms the state-of-the-art method by one to two orders of magnitude.

1 INTRODUCTION

Dictionary-based entity extraction [11] identifies all substrings from a document that match predefined entities in a reference entity table i.e. the dictionary. Compared with other kinds information extraction approaches, such as rule-based, machine learning and hybrid ones, dictionary-based entity extraction is good at utilizing extra domain knowledge encoded in the dictionary [24]. Therefore, it has been widely adopted in many real world applications that required Named entity recognition (NER), such as *academic search*, *document classification*, and *code auto-debugging*.

A typical application scenario is the *product analysis and reporting system* [10]. These systems maintain a list of well-defined products and require to find the mentions of product names in the online acquired documents. More precisely, these systems receive many consumer reviews, then they extract the substrings that mentioned reference product names from those reviews. Such mentions of referenced entities serve as a crucial signal for further analyzing the review documents, such sentiment analysis, opinion mining and recommendation. High-quality extraction of such mentions will significantly improve the effectiveness of these systems. Furthermore, the large volume of documents such systems receive turns improving the efficiency of extraction into a critical requirement.

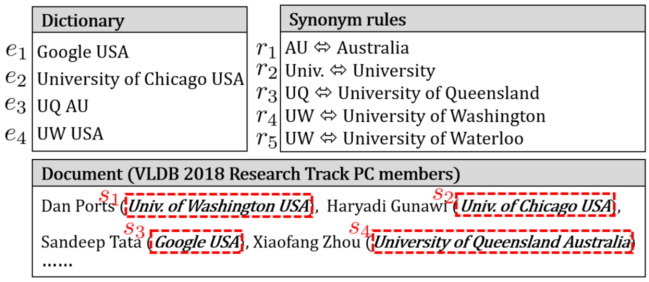

Dictionary
e_1 Google USA
e_2 University of Chicago USA
e_3 UQ AU
e_4 UW USA
Synonym rules
r_1 AU ⇔ Australia
r_2 Univ. ⇔ University
r_3 UQ ⇔ University of Queensland
r_4 UW ⇔ University of Washington
r_5 UW ⇔ University of Waterloo
Document (VLDB 2018 Research Track PC members)
Dan Ports (s_1 Univ. of Washington USA), Haryadi Gunawi (s_2 Univ. of Chicago USA),
Sandeep Tata (s_3 Google USA), Xiaofang Zhou (s_4 University of Queensland Australia)
......

Figure 1: Example of institution name extraction.

To provide high-quality entity extraction results and improve the recall, some prior work [12, 13, 35] studied the problem of Approximate dictionary-based Entity Extraction (AEE). As entities in the dictionary are represented as strings, they employ syntactic similarity functions (e.g. Jaccard and Edit Distance) to measure the similarities between entities and substrings from a document. The goal is to find not only the exactly matched substrings but also those *similar* to entities in the dictionary.

Though prior work has achieved significant degree of success in identifying the syntactic similar substrings from documents, they would still miss some substrings that are semantically similar to entities. In many cases, syntactically different strings can have very close semantic meaning. For example, consider a substring *"Mitochondrial Disease"* in a biomedical document and an entity *"Oxidative Phosphorylation Deficiency"* in the dictionary. Prior studies on AEE problems [13, 35] would fail in identifying this substring since they have very low similarity score under any syntactic similarity metric. However, *"Mitochondrial Disease"* and *"Oxidative Phosphorylation Deficiency"* actually refer to the same disease and they are expected to be included in the result. Therefore, it is necessary to propose a new framework to take both syntactic similarity and semantics carried by synonyms into consideration.

We can capture such synonyms by applying synonym rules on the basis of syntactic similarity. A `synonym rule` r is a pair of strings with the form ⟨ *lhs*⇔ *rhs*⟩ that express the same semantics. Here both `lhs` and `rhs` are token sequences. For example ⟨ *Big Apple*⇔ *New York*⟩ is a synonym rule as *"Big Apple"* is actually a

Series ISSN: 2367-2005

10.5441/002/edbt.2019.11

nickname for *"New York"*. Example 1.1 shows a real-life scenario demonstrating the effectiveness of applying synonym rules in entity extraction.

Example 1.1. Figure 1 provides an example document which contains PVLDB 2018 PC members. The dictionary includes a list of institution names, and the synonym rule table contains a list of synonym rules. The exact match approach only finds s_3 as it is the only one having an exact match in the dictionary. AEE based approaches like Faerie [13] can find s_3 and s_2 but misses s_1 and s_4. By applying rules, we can find all similar results s_1, s_2, s_3 and s_4.

As motivated above, applying synonym rules can significantly improve the effectiveness of approximate entity extraction. In this paper, we formally introduce the problem of Approximate Entity Extraction with Synonym (AEES) from dictionaries.

Though the application of synonym rules could improve effectiveness, it also brings significant challenges in computational performance. To address this issue, we study and propose new solutions for the AEES problem, along with techniques that optimize the performance. In fact, we propose an end-to-end framework called **A**pproximat**e** Dictionary-based **Ent**ity **E**xtraction with **S**ynonyms (Aeetes) that effectively and efficiently solves the AEES problem. We first propose a new similarity metric Asymmetric Rule-based Jaccard (JACCAR) to evaluate the similarity between substrings in documents and entities in the dictionary by considering both syntactic similarity and semantic relevance brought by synonyms. By properly designing the similarity measure, we can reduce the overhead of applying the synonym rules and capture the rich semantics at the same time. To support efficient extraction, we devise a clustered inverted index structure which enables skipping dissimilar entities when traversing the index. We also apply efficient sliding-window based pruning techniques to accelerate the filtering process by leveraging the overlaps between adjacent substrings in the document. We evaluate our proposed methods on three popular datasets with real application scenarios. Experimental results demonstrate the superiority of our method in both effectiveness and efficiency.

To summarize, we make the following contributions.

- We identify and formally define a new problem dictionary-based Approximate Entity Extraction from documents with Synonyms. And we propose an end-to-end framework Aeetes to efficiently address the problem.
- We devise a clustered index structure and several pruning techniques to improve the performance. Specifically, we proposed a dynamic prefix maintenance algorithm and a lazy candidate generation method to take advantage of the shared computation between substrings in a document so as to reduce the filter cost.
- We conduct an extensive empirical evaluation on three real-world datasets to evaluate the efficiency and effectiveness of the proposed algorithms. Experimental results demonstrate the effectiveness of our method. In addition, our method also achieved good efficiency: it outperforms the baseline methods by one to two orders of magnitude in extraction time.

The rest of this paper is organized as follows. We formalize the problem of AEES and introduce the overall framework in Section 2. We propose a clustered inverted index structure in Section 3. We devise the sliding window based filter techniques in Section 4. We make necessary discussions about some important issues in Section 5. The experimental results are reported in Section 6. We summarize the related work in Section 7. Finally, conclusions are made in Section 8.

2 PRELIMINARY

In this section, we first define some basic terminology to describe our work (Section 2.1). We then formulate the AEES problem and justify its definition (Section 2.2). Finally we provide an overview of our framework (Section 2.3).

2.1 Basic Terminology

Entity. An `entity` e is modeled as a token sequence, i.e, $e = e[1], ..., e[|e|]$ where $|e|$ is the number of tokens in e. For example, the entity e_2 =*"Purdue University USA"* in Figure 1 has three tokens and $e_2[3]$ = *"USA"*. We use $e[i, j]$ to denote the subsequence of tokens $e[i], ..., e[j]$ of e.

Applicable synonym rule. Given an entity e, a synonym rule r is an `applicable rule` for e if either *lhs* or *rhs* is a subsequence of e. In some cases, if two applicable rules have overlapping tokens and cannot be applied simultaneously, we call them `conflict rules`. For example, r_4 and r_5 are conflict rules as they have an overlapping token "UW". In order to generate derived entities, we need to obtain the optimal set of non-conflict rules, which includes all possible combinations. Unfortunately, finding the optimal set of non-conflict rules requires exponential time. To improve the performance, we propose a greedy algorithm to select the set of non-conflict rules whose cardinality is as large as possible (details in Section 5). We use $\mathcal{A}(e)$ to denote the sets of non-conflict rules of the entity e.

Derived entity. Given an entity e and one applicable rule r (⟨ *lhs*⇔ *rhs*⟩), without the loss of generality, we assume `lhs` is a subsequence of e. Applying r to e means replacing the `lhs` in the subsequence of e with *rhs* in r. The i^{th} new generated entity e^i is called `derived entity` of e. And e is called `origin entity` of e^i. And the set of all derived entities of e is denoted as $\mathcal{D}(e)$.

According to the previous study [3], we get a derived entity e^i of e by applying rules in a subset of $\mathcal{A}(e)$. In this process, each original token is rewritten by at most one rule [1]. Similar to previous studies [3, 29], different combination of rules in $\mathcal{A}(e)$ will result in different different derived entities. Following this routine, we can get $\mathcal{D}(e)$ by enumerating the combination of applicable rules. The cardinality of $|\mathcal{D}(e)|$ is $O(2^n)$ where $|\mathcal{A}(e)| = n$.

Consider the example data in Figure 1 again. For the entity e_3=*"UQ AU"* in the dictionary, the applicable rules $\mathcal{A}(e_3) = \{r_1, r_3\}$. Thus, $\mathcal{D}(e_4)$ can be calculated as following: {*"UQ AU"*, *"University of Queensland AU"*, *"UQ Australia"*, *"University of Queensland Australia"*}.

For a dictionary of entities $\mathcal{E}_0$, we can generate the `derived dictionary` $\mathcal{E} = \bigcup_{e \in \mathcal{E}_0} \mathcal{D}(e)$.

2.2 Problem Formulation

For the problem of approximate string join with synonyms (ASJS), previous studies have already defined some synonym-based similarity metrics, such as *JaccT* [3], *SExpand* [19] and *pkduck* [29]. In the problem setting of ASJS, we know the threshold in off-line step and need to deal with synonym rules and two collections of strings in the on-line step. So the above similarity metrics apply

[1] As shown in [3], if a new generated token is allowed to apply rules again, then it becomes a non-deterministic problem.

the synonym rules on both strings during the join processing. Suppose the lengths of two strings involved in join is S_1 and S_2, then the search space for joining the two strings is $O(2^{S_1} \cdot 2^{S_2})$

However, for our AEES problem, we obtain the entity dictionary and synonym rules in the off-line step but need to deal with the documents and threshold in the on-line step. Moreover, the length of a document will be much larger than that of entities. Unlike ASJS which computes the similarity between two strings, AEES aims at identifying `substrings` from a document that are similar to entities. Therefore, applying rules onto documents in the on-line step will be too expensive for the AEES problem. Suppose the size of a document is D and the length of an entity is e, if we directly use the similarity metrics of ASJS problem, the search space for extracting e would be $O(2^D \cdot 2^e)$. While r and s will always be similar in ASJS problem, in AEES problem the value of D is usually 10-20 times larger than e. Therefore such a complexity is not acceptable.

Based on the above discussion, we devise our asymmetric similarity metric JACCAR (for Asymmetric Rule-based Jaccard). Unlike previous studies on the ASJS problem, we only apply the synonym rules on the entities to generate derived entities in the off-line step. In the on-line extraction step, instead of applying rules on substrings in the document, we just compute the similarity between the substrings and all derived entities which have been generated in the off-line step. Here we use Jaccard to evaluate the syntactic similarity. Our techniques can also be easily extended to other similarity metrics, such as Overlap, Cosine and Dice. To verify the JACCAR value between an entity e and a substring s from the document, we first find $\mathcal{A}(e)$ and generate all derived entities of e. Then for each $e^i \in \mathcal{D}(e)$, we calculate the value of JAC (e^i, s). Finally we select the maximum JAC (e^i, s) as the value of JACCAR (e,s). The detailed definition is formalized in Definition 2.1.

Definition 2.1 (Asymmetric Rule-based Jaccard). Given an entity e in the dictionary and a substring s in the document, let $\mathcal{D}(e)$ be the full set of derived entities of e by applying rules in $\mathcal{A}(e)$. Then JACCAR(e, s)) is computed as follows:

$$\text{JACCAR}(e, s) = \max_{e^i \in \mathcal{D}(e)} \text{JAC}(e^i, s) \tag{1}$$

We want to highlight that the main difference between previous synonym-based similarity metrics for ASJS and JACCAR is that *previous approaches apply synonyms on both records that are involved into the join process; while* JaccAR *only applies synonyms on the entities in the dictionary*. Recall the above time complexity, by using JACCAR instead of similarity metrics for ASJS problem, we can reduce the time complexity from $O(2^D \cdot 2^e)$ to $O(D \cdot 2^e)$. The intuition behind JACCAR is that some rules have the same lhs/rhs, which might lead to potentially dissimilar derived entities. In order to identify a similar substring, we should focus on the derived entity that is generated by the set of synonym rules that is related to the context of substrings. By selecting the derived entity with the largest syntactic similarity, we can reach the goal of using the proper set of synonym rules to extract similar substrings from a document. JACCAR can achieve such a goal by avoiding the synonym rules which would decrease the similarity and applying those which increases the similarity.

Using the definition of Asymmetric Rule-based Jaccard, we can now characterize the AEES problem by Definition 2.2 below. Following previous studies of ASJS [3, 19], it is safe to assume that the set of synonym rules are given ahead of time. As many studies can effectively discover synonyms [2], our work can be seamlessly integrated with them. Although a recent study [29] supports discovering rules from dataset, it can only deal with abbreviations while our work here needs to support the general case of synonyms.

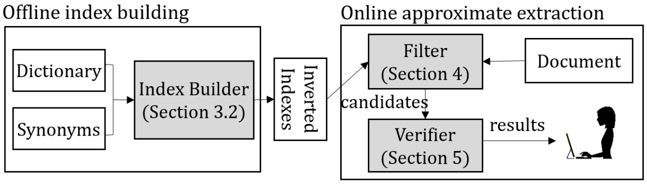


Figure 2: Architecture of Aeetes.

Definition 2.2 (AEES). Given a dictionary of entities $\mathcal{E}_0$, a set of synonym rules $\mathcal{R}$, a document d, and a threshold of Asymmetric Rule-based Jaccard τ, the goal of AEES is to return all the (e, s) pairs where s is a substring of d and $e \in \mathcal{E}_0$ such that $\text{JACCAR}(e, s) \geq \tau$.

Consider the dataset in Figure 1 again. Assume the threshold value is 0.9, then AEES returns the following pairs as results: (e_2, s_2), (e_1, s_3), (e_3, s_4). The JACCAR scores of the above three pairs are all 1.0.

2.3 Overview of Framework

As shown in Figure 2, Aeetes is an end-to-end framework which consists of two stages: off-line preprocessing and on-line extraction. The whole process is displayed in Algorithm 1. In the off-line preprocessing stage, we first find applicable synonym rules for each entity in the dictionary. Then we apply them to entities and generate the derived dictionary (line: 3). Next we create a clustered inverted index for the derived entities, which will be explained later in Section 3 (line: 4).

In the on-line extraction stage, we have a similarity threshold τ and a document d as input, and the goal is to extract all similar substrings from d. To this end, we propose a filter-and-verification strategy. In the filter step, if a derived entity e^i is similar to a substring $s \in d$, we will regard its corresponding origin entity e as the candidate of s (line: 5). In this way, we can adopt the filter techniques of Jaccard to get candidates for JACCAR. We propose effective pruning techniques in Section 4 to collect such candidates. In the verification phase, we verify the real value of JACCAR for all candidates (lines: 6-9).

3 INDEX CONSTRUCTION

In this section, we first review the length and prefix filter techniques, which serves as the cornerstone of our approaches (Section 3.1). Then we devise a clustered inverted index to facilitate the filter techniques (Section 3.2).

3.1 Filtering Techniques Revisit

In order to improve the performance of overall framework, we need to employ effective filtering techniques. As the length of a document is much larger than that of an entity, we should be able to exactly locate mentions of entities in the documents and avoid enumerating dissimilar candidates. To describe the candidate substrings obtained from the document, we use the following terminologies in this paper. Given a document d, we denote a

[2] These are introduced in Section 7, whereas generalizations of this approach are further discussed in Section 5.

Algorithm 1: Aeetes ($\mathcal{E}_0$, $\mathcal{R}$, d, τ)

```
Input: $\mathcal{E}_0$: The dictionary of entities; $\mathcal{R}$: The set of synonym
       rules; $d$: The given Docuemnt; $\tau$: The threshold of
       Asymmetric Rule-based Jaccard
Output: $\mathcal{H} = \{< e, s > | e \in \mathcal{E} \wedge s \in d \wedge \text{JACCAR}(e, s) \geq \tau\}$
begin
    Initialize $\mathcal{H}$ as $\emptyset$;
    Generate a derived dictionary $\mathcal{E}$ using $\mathcal{E}_0$ and $\mathcal{R}$;
    Construct the inverted index for all derived entities in $\mathcal{E}$;
    Traverse substrings $s \in d$ and the inverted index, generate
    a candidate set $C$ of $< s, e >$ pairs;
    for each pair $< s, e > \in C$ do
        Verify the value of JACCAR$(s, e)$;
        if JACCAR$(s, e) \geq \tau$ then
            Add $< s, e >$ into $\mathcal{H}$;
    return $\mathcal{H}$;
end
```

substring with start position p and length l as $\mathcal{W}_p^l$. A token $t \in d$ is a valid token if there exists a derived entity $e_i^j \in \mathcal{E}$ containing t. Otherwise it is an invalid token. We call a group of substrings with the same start position p in the document as a window, denoted as $\mathcal{W}_p$. Suppose the maximum and minimum length of substrings in $\mathcal{W}_p$ is l_{max} and l_{min} respectively, this window can further be denoted as $\mathcal{W}_p(l_{min}, l_{max})$.

One primary goal of the filter step is to prune dissimilar candidates. To this end, we employ the state-of-the-art filtering techniques: length filter [25] and prefix filter [9].

Length filter. The basic idea is that if two strings have a large difference between their lengths, they cannot be similar. Specifically, given two strings e, s and a threshold τ, if $|s| < \lfloor |e| * \tau \rfloor$ or $|s| > \lceil \frac{|e|}{\tau} \rceil$, then we have JAC$(s, e) < \tau$.

Suppose in the derived dictionary $\mathcal{E}$, the minimum and maximum lengths of derived entities are denoted as $|e|_\perp = \min\{|e| \, |e \in \mathcal{E}\}$ and $|e|_\top = \max\{|e| \, |e \in \mathcal{E}\}$, respectively. Then given a threshold τ, we can safely claim that only the substring $s \in d$ whose length $|s|$ is within the range $[\mathcal{E}_\perp \, \mathcal{E}_\top]$ could be similar to the entities in the dictionary where $\mathcal{E}_\perp = \lfloor |e|_\perp * \tau \rfloor$, $\mathcal{E}_\top = \lceil \frac{|e|_\top}{\tau} \rceil$.

Prefix filter. It first fixes a global order $\mathcal{O}$ for all tokens from the dataset (details in next subsection). Then for a string s, we sort all its tokens according to $\mathcal{O}$ and use $\mathcal{P}_\tau^s$ to denote the τ-prefix of string s. Specifically, for Jaccard similarity, we can filter out dissimilar strings using Lemma 3.1.

LEMMA 3.1 (PREFIX FILTER [9]). *Given two strings e, s and a threshold τ, the length of $\mathcal{P}_\tau^s$ ($\mathcal{P}_\tau^e$) is $\lfloor (1 - \tau)|s| + 1 \rfloor$ ($\lfloor (1 - \tau)|e| + 1 \rfloor$). If $\mathcal{P}_\tau^s \cap \mathcal{P}_\tau^e = \emptyset$, then we have* JAC$(s, e) < \tau$.

3.2 Index structure

With the help of length filter and prefix filter, we can check quickly whether a substring $s \in d$ is similar to an entity $e \in \mathcal{E}_0$. However, enumerating s with all the entities one by one is time consuming due to the huge number of derived entities.

To accelerate this process, we build a *clustered inverted index* for entities in the derived dictionary. The inverted index of t, denoted as $\mathcal{L}[t]$, is a list of (e^i, pos) pairs where e^i is the identifier of a derived entity containing the token t and pos is the position of t in the ordered derived entity. For all tokens in the derived entities, we assign a global order $\mathcal{O}$ among them. Then for one derived entity, we sort its tokens by the global order and pos is the

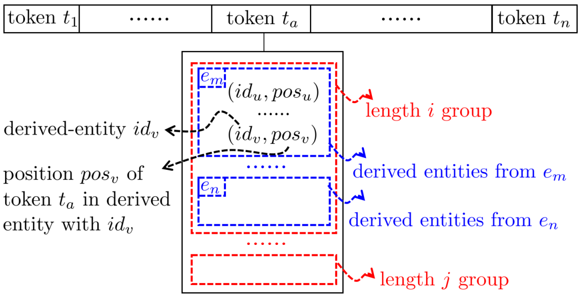
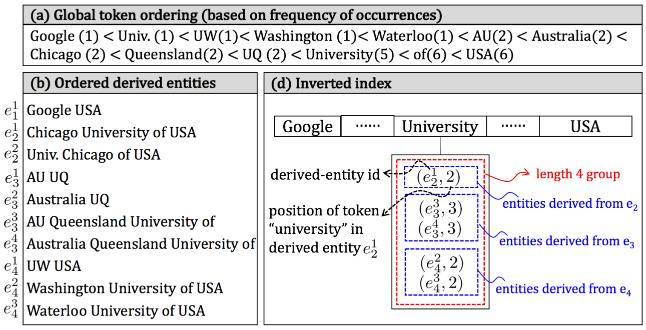


Figure 3: Index structure.



(a) Global token ordering (based on frequency of occurrences)
Google (1) < Univ. (1) < UW(1)< Washington (1)< Waterloo(1) < AU(2) < Australia(2) < Chicago (2) < Queensland(2) < UQ (2) < University(5) < of(6) < USA(6)
(b) Ordered derived entities
e_1^1 Google USA
e_2^1 Chicago University of USA
e_2^2 Univ. Chicago of USA
e_3^1 AU UQ
e_3^2 Australia UQ
e_3^3 AU Queensland University of
e_3^4 Australia Queensland University of
e_4^1 UW USA
e_4^2 Washington University of USA
e_4^3 Waterloo University of USA
(d) Inverted index
Google
......
University
......
USA
derived-entity id
$(e_2^1, 2)$
position of token "university" in derived entity e_2^1
$(e_3^3, 3)$
$(e_3^4, 3)$
$(e_4^2, 2)$
$(e_4^3, 2)$
length 4 group
entities derived from e_2
entities derived from e_3
entities derived from e_4

Figure 4: Example of index structure.

position of t in e^i under this order. Here as is same with previous studies [5, 36], we use the ascending order of token frequency as the global order $\mathcal{O}$. It is natural to deal with invalid tokens: in the on-line extraction stage, if a token $t \in d$ is an invalid token, we will regard its frequency as 0. With the help of pos, we can prune dissimilar entities with the prefix filter.

According to a recent experimental survey [21], the main overhead in set-based similarity queries comes from the filter cost. To reduce such overhead caused by traversing inverted index, we can skip some dissimilar entries by leveraging length filter: in each $\mathcal{L}[t]$, we group all the (e^i, pos) pairs by the length $l = |e^i|$. And such a group of derived entities is denoted as $\mathcal{L}_l[t]$. Then when scanning $\mathcal{L}[t]$ for $t \in s$, if l and $|s|$ does not satisfy the condition of length filter, we can skip $\mathcal{L}_l[t]$ in batch to reduce the number of accessed entries in the inverted index.

In addition, by leveraging the relationship between origin and derived entities, we can further cluster $e^i \in \mathcal{D}(e)$ within each group $\mathcal{L}_l[t]$ according to their original entity. Here we denote the group of entries with origin entity e and length l as $\mathcal{L}_l^e$. When looking for candidate entities for a substring s, if a derived entity e^i is identified as a candidate, *we will regard its origin entity e rather than the derived entity itself* as the candidate of s. If an origin entity e has already been regarded as the candidate of s, we can skip $\mathcal{L}_l^e[t]$ in batch when traversing $\mathcal{L}[t]$ where $t \in s$. Figure 3 visualizes the index structure.

Example 3.2. To have a better understanding of the index structure, we show the index (in Figure 4) built for the example data in Figure 1. As we can see, the token t="*University*" appears in five derived entities e_2^1, e_3^3, e_3^4, e_4^2, and e_4^3. And the positions of "*University*" in the corresponding ordered derived entities are 2, 3, 3, 2, and 2. Therefore, we store five (id, pos) pairs, i.e., $(e_2^1, 2)$, $(e_3^3, 3)$, $(e_3^4, 3)$, $(e_4^2, 2)$ and $(e_4^3, 2)$, in the inverted list $\mathcal{L}[University]$. The five pairs are organized into three groups (see the blue boxes) based on their original entities. For instance, $(e_3^3, 3)$ and $(e_3^4, 3)$ are

```
Algorithm 2: Index Construction($\mathcal{E}_0$, $\mathcal{R}$)
   Input: $\mathcal{E}_0$: The dictionary of entities; $\mathcal{R}$: The set of synonym
          rules.
   Output: $\mathcal{CI}$: The Clustered Inverted Index of entities
1  begin
2      Generate a derived dictionary $\mathcal{E}$ using $\mathcal{E}_0$ and $\mathcal{R}$;
3      Initilize $\mathcal{CI} = \emptyset$ and obtain the global order $O$;
4      foreach derived entitiy $e' \in \mathcal{E}$ do
5          $l \leftarrow |e'|$, $e \leftarrow$ origin entity of $e'$;
6          foreach token $t \in e'$ do
7              Add the pair $(e', pos)$ into the corresponding
               group in inverted list $\mathcal{L}_l^e[t]$;
8      foreach $\mathcal{L}[t]$ do
9          $\mathcal{CI} = \mathcal{CI} \cup \mathcal{L}[t]$
10     return $\mathcal{CI}$;
11 end
```

grouped together as both e_3^3 and e_3^4 derived entities are from the same original entity e_3. In addition, they are further clustered into a group based on their lengths (see the red boxes). In this example, these five pairs are grouped into a length-4 group as the length of the derived entities are 4.

Algorithm 2 gives the details of constructing an inverted index. It first applies synonyms to the entities in the dictionary to get a derived dictionary (line 2). Then for each token t in the derived entities, it stores a list of (e', pos) pairs where e' is the identifier of a derived entity containing t and pos is the position of t in this derived entity according to the global order O (line: 4-7). The (e', pos) pairs in each list are organized into groups based on the length l and their corresponding origin entity e (line 5). Finally, we aggregate all the inverted lists and get the clustered index (line: 9).

4 SLIDING WINDOW BASED FILTERING

Based on the discussion in Section 3, we can come up with a straightforward solution for the AEES problem: we slide the window $\mathcal{W}_p(\mathcal{E}_\perp, \mathcal{E}_\top) \in d$ from the beginning position of document d. For each window, we enumerate the substrings and obtain the prefix of each substring. Next, we recognize the valid tokens from the prefix for each substring and scan the corresponding inverted lists to obtain the candidates. Finally, we verify the candidates and return all truly similar pairs.

Although we can prune many dissimilar substrings by directly applying length filter and prefix filter with the help of inverted index, the straightforward method needs to compute the prefix for a large number of substrings. Thus it would lead to low performance. In this section, we propose a sliding window based filtering mechanism to efficiently collect the candidates from a given document. To improve the overall efficiency, we devise effective techniques based on the idea of sharing computations between substrings and windows. We first devise a dynamic (incremental) prefix computation technique to take advantage of the overlaps between adjacent substrings and windows in Section 4.1. Next we further propose a lazy strategy to avoid redundant visits on the inverted index in Section 4.2.

4.1 Dynamic Prefix Computation by Shared Computation

We have the following observations w.r.t the windows and substrings. On the one hand, for two substrings in the same window with different length i.e. $\mathcal{W}_p^{l_i}$ and $\mathcal{W}_p^{l_j}(\mathcal{E}_\perp \leq l_i < l_j \leq \mathcal{E}_\top)$, they share l_i common tokens. On the other hand, two adjacent windows $\mathcal{W}_p(\mathcal{E}_\perp, \mathcal{E}_\top)$ and $\mathcal{W}_{p+1}(\mathcal{E}_\perp, \mathcal{E}_\top)$ share $\mathcal{E}_\top - 1$ common tokens. This is very likely that there is a large portion of common tokens between the prefixes of $\mathcal{W}_p^{l_i}$ and $\mathcal{W}_p^{l_j}$ and those of $\mathcal{W}_p^{l_i}$ and $\mathcal{W}_{p+1}^{l_i}$.

Motivated by these observations, we can improve the performance of the straightforward solution by dynamically computing the prefix for substrings in the document. Here we use $\mathcal{P}_\tau^{p,l}$ to denote the set of tokens of the τ-prefix(i.e., prefix of length $\lfloor(1 - \tau) * l + 1\rfloor$) of substring $\mathcal{W}_p^l$. Then we can obtain the prefix of one substring by utilizing that of a previous one. Specifically, for a given window $\mathcal{W}_p$, we first directly obtain $\mathcal{P}_\tau^{p,0}$ and then incrementally compute $\mathcal{P}_\tau^{p,l}$ on the basis of $\mathcal{P}_\tau^{p,l-1}$. Then for each substring $\mathcal{W}_p^l \in \mathcal{W}_p$, we scan the inverted lists of the valid tokens and collect the candidate entities. Similarly, for a substring $\mathcal{W}_{p+1}^l \in \mathcal{W}_{p+1}$, we can obtain its prefix $\mathcal{P}_\tau^{p+1,l}$ from $\mathcal{P}_\tau^{p,l}$. Then we can collect the candidate entities for each substring in $\mathcal{W}_{p+1}$ with the same way above.

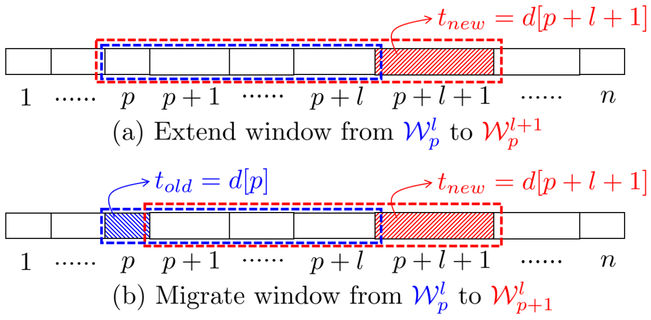
To reach this goal, we propose two operations Window Extend and Window Migrate to dynamically compute the prefix of substrings and collect candidate pairs for the above two scenarios.

Window Extend This operation allows us to obtain $\mathcal{P}_\tau^{p,l+1}$ from $\mathcal{P}_\tau^{p,l}$. Figure 5(a) gives an example of extending the window $\mathcal{W}_p^l$ to $\mathcal{W}_p^{l+1}$. As shown, when performing Window Extend, the length of the substring increases by 1. In this case, the length of the τ-prefix of $\mathcal{W}_p^{l+1}$ (i.e. $\lfloor(1-\tau)*(l+1)+1\rfloor$) can either increase by 1 or stay the same compared with the length of the τ-prefix of $\mathcal{W}_p^l$ (i.e. $\lfloor(1 - \tau) * l + 1\rfloor$). Then we can perform maintenance on the prefix accordingly:

- If the length of τ-prefix stays the same, we need to check whether the newly added token $d[p + l + 1]$ will replace a token in $\mathcal{P}_\tau^{p,l}$. If so, we need to replace a lowest ranked token $t \in \mathcal{W}_p^l$ with $d[p+l+1]$ in the new prefix. Otherwise, there is no change in the prefix i.e. $\mathcal{P}_\tau^{p,l+1} = \mathcal{P}_\tau^{p,l}$.
- If the length of τ-prefix increases by 1, then we need to discuss whether the newly added token $d[p + l + 1]$ belongs to the new prefix $\mathcal{P}_\tau^{p,l+1}$. If so, we can just have $\mathcal{P}_\tau^{p,l+1} = \mathcal{P}_\tau^{p,l} \cup d[p + l + 1]$. Otherwise, we should find a token $t \in \mathcal{W}_p^l$ and $t \notin \mathcal{P}_\tau^{p,l}$ with the highest rank. Then we have $\mathcal{P}_\tau^{p,l+1} = \mathcal{P}_\tau^{p,l} \cup t$.

Example 4.1. Assume $\tau = 0.8$, when extending window from $\mathcal{W}_3^3$ to $\mathcal{W}_3^4$ (see Figure 6(a)), $|\mathcal{P}_\tau^{3,3}| = \lfloor(1 - 0.8) * 3 + 1\rfloor = 1$ and $|\mathcal{P}_\tau^{3,4}| = \lfloor(1 - 0.8) * 4 + 1\rfloor = 1$. So the length of τ-prefix stays the same. $\mathcal{P}_\tau^{3,3} = \{t_4\}$ as t_4 has the highest rank in window $\mathcal{W}_3^3$. The rank of new token t_6 in window $\mathcal{W}_3^4$ is 18, so t_6 will not replace a token in $\mathcal{P}_\tau^{3,3}$, so $\mathcal{P}_\tau^{3,4} = \mathcal{P}_\tau^{3,3} = \{t_4\}$. If the rank of t_6 is 2 instead of 18, then t_6 will replace a token in $\mathcal{P}_\tau^{3,3}$. In this case, $\mathcal{P}_\tau^{3,4} = \mathcal{P}_\tau^{3,3} - \{t_4\} \cup \{t_6\} = \{t_6\}$.

Now let's see the example in Figure 6(b) where we extend window from $\mathcal{W}_3^4$ to $\mathcal{W}_3^5$. The length of $\mathcal{P}_\tau^{3,4}$ is $\lfloor(1 - 0.8) * 4 + 1\rfloor = 1$ and $\mathcal{P}_\tau^{3,4} = \{t_4\}$. But the length of $\mathcal{P}_\tau^{3,5}$ now is $\lfloor(1 - 0.8) * 5 + 1\rfloor = 2$. The newly added token t_7 with rank 2 is in $\mathcal{P}_\tau^{3,5}$, so $\mathcal{P}_\tau^{3,5} = \mathcal{P}_\tau^{3,4} \cup \{t_7\} = \{t_4, t_7\}$. If the rank of t_7 is 10 instead of 2, then t_7 should not be a token in $\mathcal{P}_\tau^{3,5}$. In this case,

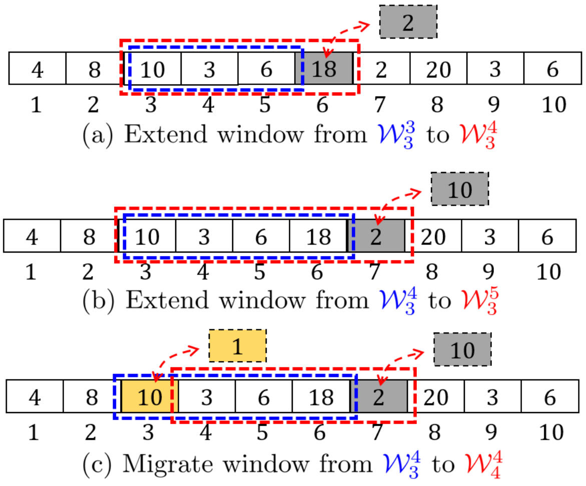


$t_{new} = d[p+l+1]$
1 …… p $p+1$ …… $p+l$ $p+l+1$ …… n
$t_{old} = d[p]$
$t_{new} = d[p+l+1]$
1 …… p $p+1$ …… $p+l$ $p+l+1$ …… n

(a) Extend window from $\mathcal{W}_p^l$ to $\mathcal{W}_p^{l+1}$

(b) Migrate window from $\mathcal{W}_p^l$ to $\mathcal{W}_{p+1}^l$

Figure 5: Example of window extend and window migrate.



(a) Extend window from $\mathcal{W}_3^3$ to $\mathcal{W}_3^4$

(b) Extend window from $\mathcal{W}_3^4$ to $\mathcal{W}_3^5$

(c) Migrate window from $\mathcal{W}_3^4$ to $\mathcal{W}_4^4$

Figure 6: Example of the Window Extend operator and the Window Migrate operator. Values in the boxes are the ranks of tokens according to a global order. Value k means the ranking of the token is top-k.

$\mathcal{P}_\tau^{3,5} = \mathcal{P}_\tau^{3,4} \cup \{t_5\} = \{t_4, t_5\}$ as t_5 has the highest rank, $t_5 \in \mathcal{W}_3^4$ and $t_5 \notin \mathcal{P}_\tau^{3,4}$.

Window Migrate This operation allows us to obtain the prefix of $\mathcal{W}_{p+1}^l$ from that of $\mathcal{W}_p^l$. Figure 5(b) shows an example of migrating the window $\mathcal{W}_p^l$ to $\mathcal{W}_{p+1}^l$. We can see that when performing Window Migrate, the length of substring will stay the same. In this case, for a substring $\mathcal{W}_{p+1}^l$, the token $t_{old} = d[p]$ will be removed and the token $t_{new} = d[p+1+l]$ will be inserted. Then we discuss the maintenance of prefix according to t_{old}: .

- If $t_{old} \notin \mathcal{P}_\tau^{p,l}$, it makes no influence on the prefix, we only need to check whether t_{new} will replace a token in $\mathcal{P}_\tau^{p,l}$ to form $\mathcal{P}_\tau^{p+1,l}$. If so, we need to replace the lowest ranked token $t \in \mathcal{W}_p^l$ with t_{new} in $\mathcal{P}_\tau^{p+1,l}$. Otherwise, we have $\mathcal{P}_\tau^{p,l+1} = \mathcal{P}_\tau^{p,l}$.
- If $t_{old} \in \mathcal{P}_\tau^{p,l}$, we still need to check whether t_{new} will appear in $\mathcal{P}_\tau^{p+1,l}$ in a similar way. If so, we need to replace t_{old} with t_{new} to generate $\mathcal{P}_\tau^{p+1,l}$; Otherwise, we need to replace t_{old} with a token t s.t. $t \in \mathcal{W}_p^l$ and $t \notin \mathcal{P}_\tau^{p,l}$ with the highest rank.

Example 4.2. Consider the case when migrating window from $\mathcal{W}_3^4$ to $\mathcal{W}_4^4$ (see Figure 6(c)), both $\mathcal{W}_3^4$ and $\mathcal{W}_4^4$ have the length $\lfloor(1-0.8)*3+1\rfloor = 1$ (assume $\tau = 0.8$). $\mathcal{P}_\tau^{3,4} = \{t_4\}$ as t_4 has the highest rank in window $\mathcal{W}_3^4$. After this migration, we have $t_{old} = t_3$ (with rank 10) and $t_{new} = t_7$ (with rank 2). Then we know $t_{old} \notin \mathcal{P}_\tau^{3,4}$ and t_{new} will replace a token in $\mathcal{P}_\tau^{3,4}$, thus we have $\mathcal{P}_\tau^{4,4} = \mathcal{P}_\tau^{3,4} - \{t_4\} \cup \{t_7\} = \{t_7\}$. If the rank of t_7 is 10 rather than 2 (see the blocks with gray color), then t_{new} will not replace a token in $\mathcal{P}_\tau^{3,4}$ and $\mathcal{P}_\tau^{4,4} = \mathcal{P}_\tau^{3,4} = \{t_4\}$.

Further, if the rank of t_3 is 1 instead of 10 (see the blocks with yellow color), then $\mathcal{P}_\tau^{3,4} = \{t_3\}$ as now t_3 has the highest rank in the window $\mathcal{W}_3^4$. Then we know $t_{old} \in \mathcal{P}_\tau^{3,4}$ and t_{new} (with rank 2) will occur in $\mathcal{P}_\tau^{4,4}$, so $\mathcal{P}_\tau^{4,4} = \mathcal{P}_\tau^{3,4} - \{t_{old}\} \cup \{t_{new}\} = \{t_7\}$. If the rank of t_7 is 10 rather than 2 (see the blocks with gray color), then we need to replace t_{old} with t_4 as t_4 has the highest rank such that $t_4 \in \mathcal{W}_3^4$ and $t_4 \notin \mathcal{P}_\tau^{3,4}$. Therefore, $\mathcal{P}_\tau^{4,4} = \{t_4\}$.

We show the steps of candidate generation with dynamic prefix computation in Algorithm 3. To implement the operations defined above, we need to use some helper functions. First, we define the function *ScanInvetedIndex* which takes the inverted index and $\mathcal{P}_\tau^{p,l}$ as input and return all the candidate entities of $\mathcal{W}_p^l$ by visiting the inverted indexes that are corresponding to valid tokens in $\mathcal{P}_\tau^{p,l}$. For a valid token $t \in \mathcal{P}_\tau^{p,l}$, we can obtain the candidates from the inverted index $\mathcal{L}[t]$. Note that since we have grouped all items in the inverted index by length, for a group $\mathcal{L}_{l_e}[t] \in \mathcal{L}[t]$, if l_e and l does not satisfy the length filter, we can skip $\mathcal{L}_{l_e}[t]$ in batch. Similarly, if the position of t is beyond the τ-prefix of a derived entity e^i, we can also discard e^i.

Then we devise the function *ExtCandGeneration* to support the Window Extend operation. It first derives the prefix of the current substring from the prefix of previous substring according to the above description; then it obtains the candidates for the current substring. Similarly, we also devise the *MigCandGeneration* function to support Window Migrate operation. Due to the space limitation, we omit their pseudo codes here.

Algorithm 3: Candidate Generation($\mathcal{CI}$, d, τ)

```
Input: $\mathcal{CI}$: The Inverted Index; $d$: The given Document; $\tau$:
       The threshold of Asymmetric Rule-based Jaccard
Output: $C$: The set of candidate pairs
1  begin
2      Initialize $C \leftarrow \emptyset, p \leftarrow 1$;
3      Obtain the prefix $\mathcal{P}_\tau^{p,\mathcal{E}_\perp}$;
4      $C = C \cup$ ScanInvetedIndex($\mathcal{CI}, \mathcal{P}_\tau^{p,\mathcal{E}_\perp}$);
5      for $len \in [\mathcal{E}_\perp + 1, \mathcal{E}_\top]$ do
6          $C_p^{len}, \mathcal{P}_\tau^{p,len} \leftarrow$ ExtCandGeneration($\mathcal{P}_\tau^{p,len-1}$);
7          $C = C \cup C_{len}$;
8      while $p < |d| - \mathcal{E}_\perp$ do
9          if $C_{p-1} \neq \emptyset$ then
10             for $len \in [\mathcal{E}_\perp, \mathcal{E}_\top]$ do
11                 $C = C \cup$ MigCandGeneration($\mathcal{P}_\tau^{p-1,len}$);
12         else
13             Obtain the candidates of $C_p$ in the same way of
               line 6 to line 7;
14         $p \leftarrow p + 1$;
15     Perform Window Extend on the last window with length
       $|d| - \mathcal{E}_\perp + 1$ to collect candidates;
16     return $C$;
17 end
```

The whole process of the algorithm is as follows. We first initialize the candidate set C and start from the first position of the document (line: 2). Here we denote the candidates from window $\mathcal{W}_p$ as C_p. For the first window, we perform Window Extend and collect all the candidates of substrings $\mathcal{W}_0^l$ (line: 4-line: 7). Next we enumerate the start position of the window and look at the previous window. If the previous window has candidate substrings, we will perform Window Migrate on the previous window to obtain the prefix for each substring in the current window and then collect the candidates for each substring (line: 9). Otherwise, we will obtain the prefix of each substring with Window Extend (line: 12). Such processes are repeated until we reach the end of the document. Finally, we return all the candidates and send them to the verification step (line: 16).

Example 4.3. Consider the example data in Figure 1 again. We have $\mathcal{E}_\perp = 1$ and $\mathcal{E}_\top = 5$. Assume document d has 1000 tokens. The total number of the function calls of ExtCandGeneration and MigCandGeneration is $1000 \times (\mathcal{E}_\top - \mathcal{E}_\perp) = 4000$. Notice that, both ExtCandGeneration and MigCandGeneration compute the prefix incrementally. However, the straightforward method needs to compute the prefix for $500,500$ substrings, as it requires to enumerate all possible substrings and compute the prefix for each of them independently.

4.2 Lazy Candidate Generation

With the dynamic prefix computation method, we avoid obtaining the prefix of each substring from scratch. However, there is still room for improvement. We can see that in Algorithm 3, we need to traverse the inverted indexes and generate candidates for all the valid tokens after obtaining the prefix. As substrings within the same window could have a large overlap in their prefix, they could also share many valid tokens. Moreover, a valid token t is also likely to appear anywhere within the same document. Even if t belongs to disjoint substrings, the candidate entities are still from the same inverted index $\mathcal{L}(t)$.

To further utilize such overlaps, we come up with a lazy strategy for candidate generation. The basic idea is that after we compute the prefix of a substring, we will not traverse the inverted indexes to obtain candidates immediately. Instead, we just collect the valid tokens for each substring and construct a global set of valid tokens $\mathcal{T}$. Finally, we will postpone the visits to inverted indexes after we finish recognizing all the valid tokens and corresponding substrings. In this way, *for each valid token $t \in \mathcal{T}$, we only need to traverse its associated inverted index $\mathcal{L}_t$ once during the whole process of extraction for one document.*

The difficulty of implementing the lazy strategy is two-fold. The first problem, i.e. the one of large space overhead required is discussed next, whereas the second one, i.e. the one related to different substring lengths is discussed later. Since we do not collect candidates immediately for each substring, we need to maintain the valid tokens for each substring. As the number of substrings is rather large, there will be heavy space overhead. To solve this problem, we take advantage of the relationship between substrings within the same window. Here we denote the valid token set of a substring $\mathcal{W}_p^l$ as $\Phi_p(l)$. For one window $\mathcal{W}_p$, we only keep the full contents of $\Phi_p(\mathcal{E}_\perp)$. To obtain $\Phi_p(l), l > \mathcal{E}_\perp$, we utilize a light-weighted structure `delta valid token`, which is represented as a tuple $< t, \circ >$. Here t is the valid token that is different from the previous substring; $\circ$ is a symbol to denote the operation on t. If $\circ$ is $+$ $(-)$, it means we need to insert t into(remove t from) the previous valid token set. We denote the set of delta valid tokens of substring $\mathcal{W}_p^l$ as $\Delta\phi(l)$. And then we have:

$$\Phi_p(l+1) = \Phi_p(l) \biguplus \Delta\phi(l) \tag{2}$$

where $\uplus$ means applying all operations of the corresponding token in $\Delta\Phi(l)$ on the given valid token set. If $\Delta\phi(l) = \emptyset$, it means that $\Phi_p(l+1) = \Phi_p(l)$. Then we can obtain all valid tokens and the corresponding candidate substrings of window $\mathcal{W}_p$ as:

$$\Psi(p) = \bigcup_{l \in [\mathcal{E}_\perp, \mathcal{E}_\top]} < \mathcal{W}_p^l, \Phi_p(\mathcal{E}_\perp) \biguplus \Sigma_{i=\mathcal{E}_\perp}^{l} \Delta\phi_p(i) > \tag{3}$$

Example 4.4. Consider the example document in Figure 6 again. Assume we have $\mathcal{E}_\perp = 1$, $\mathcal{E}_\top = 4$ and $\tau = 0.6$. $\Phi_3(1) = \{t_3\}$ as t_3 is the only valid token in $\mathcal{P}_\tau^{3,1}$. Then $\Phi_3(2) = \Phi_3(1) \uplus \{< t_3, - >, < t_4, + >\}$ as t_3 is not a valid token for $\mathcal{W}_3^2$ but t_4 is. Similarly, we have $\Phi_3(3) = \Phi_3(2) \uplus \{< t_5, + >\}$ and $\Phi_3(4) = \Phi_3(3)$. Therefore, according to Equation 3, the valid tokens and the corresponding candidate substrings of window $\mathcal{W}_3^4$ can be expressed as:

$$\Psi(3) = < \mathcal{W}_3^1, \{t_3\} > \bigcup < \mathcal{W}_3^2, \{t_4\} > \bigcup \\ < \mathcal{W}_3^3, \{t_4, t_5\} > \bigcup < \mathcal{W}_3^4, \{t_4, t_5\} >$$

The second issue follows from the fact that the candidate substrings have different lengths. For one valid token $t \in \mathcal{T}$, it might belong to multiple $\Phi_p(l)$ with different values of l. Then we should be able to identify $\mathcal{W}_p^l$ with different l by scanning $\mathcal{L}[t]$ only once. To reach this goal, we propose an effective data structure to connect the candidate substrings and list of entities. Specifically, after moving to the end of the document using Window Extend and Window Migrate, we collect the first valid token set $\Phi_p(\mathcal{E}_\perp)$ and delta valid token sets $\Delta\phi(l)$ for all windows $\mathcal{W}_p$. Next we obtain $\Psi(p)$ using Equation 3 and construct an inverted index $\mathcal{I}$ for candidate substrings. Here a substring $\mathcal{W}_p^l \in \mathcal{I}[t]$ means that $\mathcal{W}_p^l$ is a candidate for entities in $\mathcal{L}[t]$. Then to meet the condition of length and prefix filter, we also group $\mathcal{W}_p^l \in \mathcal{I}[t]$ by length l, denoted as $\mathcal{I}_l[t]$. For substrings $s \in \mathcal{I}_l[t]$, only the entities in groups $\mathcal{L}_{|e|}[t]$ s.t. $|e| \in [\lfloor l * \tau \rfloor, \lceil \frac{l}{\tau} \rceil]$ can be candidate of s. In this way, we can obtain the entity for all $\mathcal{W}_p^l$ with $t \in \Phi_p(l)$ by scanning $\mathcal{L}[t]$ only once. Then for a candidate substring $\mathcal{W}_p^l$, the set of entities can be obtained by $\bigcup_{t \in \mathcal{W}_p^l} \mathcal{L}_{|e|}[t]$.

Algorithm 4 demonstrates the steps of lazy candidate generation. We first collect $\Phi_p(0)$ and $\Delta\phi_p(l)$ for each window using the same method in Algorithm 3. We then initialize the global token dictionary and inverted index for substrings (line: 3). But unlike Algorithm 3, here we only track the valid tokens for each substring instead of generating the candidates. Next, we generate the valid token set for each substring using Equation 2 (line: 4). And we can collect all the valid tokens and their corresponding substrings from them (line: 5- 7). With such information, we can build a mapping between the groups with different lengths $|e|$ in the inverted index and the candidate substrings with different lengths l s.t. $\lfloor |e| * \tau \rfloor \leq l \leq \lceil \frac{|e|}{\tau} \rceil$(line: 9). Then we scan the inverted list only once and collect the candidates. Finally, the entities for a candidate substring can be obtained from the union of the inverted indexes of all its valid tokens (line: 11). We summarize the correctness of Lazy Candidate Generation in Theorem 4.5.

THEOREM 4.5 (CORRECTNESS). *The Lazy Candidate Generation method will not involve any false negative.*

Algorithm 4: Lazy Candidate Generation($\mathcal{CI}$, d, τ)

```
Input: $\mathcal{CI}$: The Inverted Index; $d$: The Document; $\tau$: The
       threshold of JAC
Output: $C$: The candidates
begin
    Use similar methods in Algorithm 3 to generate $\Phi_p(0)$
    and $\Delta\phi_p(l)$ for each window $\mathcal{W}_p$.
    Initialize $\mathcal{T}$ and $\mathcal{I}$;
    Generate all valid token set $\phi_l(p)$ using Equation 2;
    foreach $\phi_p(l)$ do
        collect the valid tokens, update $\mathcal{T}$;
        Construct the inverted index for substrings $\mathcal{I}$;
    foreach $t \in \mathcal{T}$ do
        Map entities in $\mathcal{L}_{|e|}[t]$ with candidate substrings in
        $\mathcal{I}_l[t]$ s.t. length filter;
        Scan $\mathcal{L}[t]$, obtain candidates for each $l$;
        $C = C \cup < \mathcal{W}_p^l, \bigcup_{t \in \mathcal{W}_p^l} \mathcal{L}_{|e|}[t] >$;
    return $C$;
end
```

5 DISCUSSION

In this section, we discuss about the scope as well as the generalization of our work.

Gathering Synonym Rules We first discuss about the way to obtain synonym rules. In our work, we make an assumption that the set of synonyms are known ahead of time. But it will not influence the generalization of our Aeetes framework. For a collection of documents, there are multiple sources of the synonyms rules. We list some of them below:

- Firstly, the synonym rules can come from common sense as well as knowledge bases. For example, we can use the synonyms provide by WordNet [3] as the input of Aeetes. The common sense knowledge can also provide rich source of synonyms, such as the abbreviation of institutes, address and locations used in our DBWorld and USJob datasets.
- Secondly, some domain specific applications provided the set of synonyms. For example, in the PubMed dataset, the synonyms are created by domain experts, which are very important information in understanding the medical publications. Therefore, performing AEES on such kinds of applications is with great values of practical application.
- Lastly, we can also discover the synonyms from documents with existing systems. For example, the output of [7] and [22] can be utilized directly as the input of our framework.

There are also some previous studies about data transformation and learning by example, which are summarized in Section 7. These studies are orthogonal to our work as they focused on detecting high-quality set of rules while our problem is about how to perform approximate entity extraction with predefined synonym rules. The synonyms discovered by them can also be used as the input of our framework.

Generation of Non-conflict Rule Set Given an entity e, let $\mathcal{A}_c(e)$ be the set of complete applicable rules of e and lhs_i be the left-hand of the rule r_i ($\langle$ $lhs_i \Leftrightarrow rhs_i \rangle$). Without loss of generality, we assume the lhs of an applicable rule is a subsequence of the entity. Two rules r_i and r_j are conflict rules if $lhs_i \cap lhs_j \neq \emptyset$. Our goal is to choose a non-conflict set $\mathcal{A}(e) \subseteq \mathcal{A}_c(e)$ such that (i) all rules in $\mathcal{A}(e)$ are non-conflict and (ii) the cardinality of $\mathcal{A}(e)$ is as large as possible.

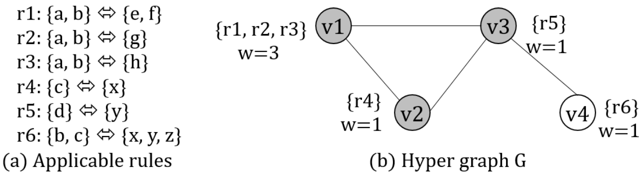


Figure 7: Hypergraph for the applicable rules of the entity $\{a, b, c, d\}$.

The non-conflict rule set can be obtained in the following way:

(1) First build a hypergraph $G = (V, E, W)$ for $\mathcal{A}_c(e)$. Each vertex $v \in V$ corresponds to a set of applicable rules whose left-hands are the same. The number of rules in the vertex v is the weight w of v. There is an edge $e \in E$ between two vertices whose rules are non-conflict.
(2) With such a hypergraph, we can obtain the set of non-conflict rules by finding the maximum weighted clique in the hypergraph G.

Example 5.1. Consider the set of complete applicable rules $\mathcal{A}_c(e)$ for the entity $e = \{a, b, c, d\}$ in Figure 7(a). The corresponding hypergraph G is shown in Figure 7(b). For instance, v_1 contains $\{r_1, r_2, r_3\}$ as they have identical left-hands, i.e., $\{a, b\}$. There is an edge between v_1 and v_2 since $\{a, b\} \cap \{c\} = \emptyset$. In this hypergraph, $\{v_1, v_2, v_3\}$ is the maximal weighted clique. Therefore, the final non-conflict applicable rules $\mathcal{A}(e) = \{r_1, r2, r_3, r_4, r_5\}$.

Unfortunately, finding the maximal weighted clique is a well-known NP-Complete problem. In order to efficiently find $\mathcal{A}(e)$ with large enough cardinality, we adopt a greedy algorithm with the following steps. Firstly, we choose the vertex v^* with maximal weight as a start point. Next we pick the next vertex v with the maximal weight among the unseen vertices such that adding v to the current clique is still a clique. Then we repeat step 2 until no more vertex can be added into the result set.

Example 5.2. Consider the set of complete applicable rules $\mathcal{A}_c(e)$ for entity $e = \{a, b, c, d\}$ in Figure 7 again. The greedy algorithm first chooses v_1 as it has the maximal weight. Then the algorithm picks v_2 since , it is still a clique after adding v_2. Similarly, v_3 is also added. Finally, the clique is $\{v_1, v_2, v_3\}$ and the corresponding non-conflict applicable rules are $\mathcal{A}(e) = \{r_1, r2, r_3, r_4, r_5\}$. Here the greedy algorithm achieves the optimal result.

6 EXPERIMENTS

6.1 Environment and Settings

In this section, we evaluated the effectiveness and efficiency of all proposed algorithms on three real-life datasets:

- PubMed. It is a medical publication dataset. We selected 100, 000 paper abstracts as documents and keywords from 10, 000, 000 titles as entities to construct the dictionary. In addition, we collect 50, 476 synonym Mesh [4] (Medical Subject Headings)[5] term pairs, which are provided by the domain experts.

[3] https://wordnet.princeton.edu/

[4] https://www.ncbi.nlm.nih.gov/mesh

[5] Mesh is the NLM controlled vocabulary thesaurus used for indexing articles for PubMed.

- DBWorld. We collected 1,000 message contents as documents and keywords in 1,414 titles as entities in the dictionary. We also gather 1076 synonym rules including conference names, university names and country names which are common sense knowledge.
- USJob. We chosen 22,000 job descriptions as documents, and 1,000,000 keywords in the job titles as entities in the dictionary. In addition, we collected 24,305 synonyms including the abbreviations of company names, state names and the different names of job positions.

Table 1 gives the statistics of datasets, including the average number of tokens in documents (avg $|d|$), average number of tokens in entities (avg $|e|$), and average number of applicable rules on one entity (avg $|\mathcal{A}(e)|$).

Table 1: Dataset statistics.

	# docs	# entities	# synonyms	avg $\|d\|$	avg $\|e\|$	avg $\|\mathcal{A}(e)\|$
PubMed	8,091	370,836	24,732	187.81	3.04	2.42
DBWorld	1,414	113,288	1,076	795.89	2.04	3.24
USJob	22,000	1,000,000	24,305	322.51	6.92	22.7

All experiments were conducted on a server with an Intel(R) Xeon(R) CPU processor (2.7 GHz), 16 GB RAM, running Ubuntu 14.04.1. All the algorithms were implemented in C++ and compiled with GCC 4.8.4.

6.2 Evaluation of Effectiveness

First, we evaluate the effectiveness of our metric JACCAR by comparing with the state-of-the-art syntactic similarity metrics.

Ground truth For our task, there is no ground truth on these datasets. Borrowing the idea from previous studies on ASJS problem [19, 29], we manually create the ground truths as following: We marked 100 substrings in the documents of each dataset such that each of the marked substrings has one matched entity in the origin entity dictionary. Each pair of marked substring and the corresponding entity is a ground truth. For example, (*Primary Hypertrophic Osteoarthropathy*, *Idiopathic Hypertrophic Osteoarthropathy*), (*Univ. of California Berkeley USA*, *UC Berkeley USA*), and (*SDE in FB*, *software development engineer in Facebook*) are ground truths in PubMed, DBWorld and USJob dataset respectively.

Baseline methods To demonstrate the effectiveness of applying synonym rules, we compare JACCAR with two state-of-the-art syntactic similarity metrics: (i) Jaccard, which is the original Jaccard similarity; and (ii) Fuzzy Jaccard (FJ), which is proposed in [32]. As they are just based on syntactic similarity, they cannot make use of the synonym rules.

We evaluate the effectiveness of all the similarity measures by testing the *Precision* (short for "P"), *Recall* (short for "R"), and *F-measure* (short for "F"), where F-measure = $\frac{2 \times P \times R}{P+R}$ on the three datasets.

Results Table 2 reports the precision, recall and the F-measure of all similarity measures on three datasets. We have the following observations. Firstly, JACCAR obtains higher F-measure scores than Jaccard and FJ. The reason is that JACCAR can use synonym rules to detect the substrings that are semantically similar to the entities, which indicates the advantage of integrating syntactic metrics with synonyms in the entity extraction problem. Secondly, FJ has higher Precision scores than Jaccard as FJ can identify tokens with minor typos. However, Jaccard has higher Recall scores than FJ because FJ may increase the similarity score for substrings that are not valid ground truth.

We present a sample of ground truth for each dataset in Figure 8 to perform a case study on the quality of three similarity metrics. We can see that in PubMed, both Jaccard and FJ are equal to 0. This is because the ground truth substring has no common (or similar) tokens with the entity. JACCAR =1.0 as JACCAR can apply the second synonym to the entity. In DBWorld, FJ has higher similarity score than Jaccard. The reason is that Jaccard can only find three common tokens "The University of " between the substring and the entity. But FJ can get extra benefit by identifying "Aukland" in the document is similar to "Auckland" in the entity (as their edit-distance is only 1). JACCAR achieves the highest score as JACCAR can apply the first synonym on the entity to obtain two more common tokens, i.e., "New Zealand". Similarly, in USJob, FJ has a higher score than Jaccard and JACCAR achieves the highest score.

6.3 Evaluation of Efficiency

Next we look at the efficiency of proposed techniques. We use the average extraction time per document as the main metric for evaluation.

End-to-end performance First we report the end-to-end performance. As there is no previous study on the AEES problem, we extend Faerie [13], which reports the best performance in AEE task, and propose FaerieR to serve as the baseline method. In order to let FaerieR handle the AEES problem, for each dataset we perform a preprocessing by using all applicable synonym rules to all entities in the dictionary so as to construct a derived dictionary. Then we use such a derived dictionary as the input of Faerie. After that, we conduct post-processing to recognize the pairs of origin entity and substrings in the document. For FaerieR, we omit the preprocessing and post-processing time and only report the extraction time by the original Faerie framework. For the implementation of Faerie, we use the code obtained from the original authors.

We compare the overall performance of Aeetes and FaerieR with different threshold values ranging from 0.7 to 0.9 on all three datasets. As shown in Figure 9, Aeetes outperforms FaerieR by one to two orders of magnitudes. The main reason is that we proposed a series of pruning strategies to avoid duplication computation came from applying synonyms and the overlaps in documents.

In the experiment, we observe that the bottleneck of memory usage is index size. And we report it for Aeetes and FaerieR as following. In PubMed, the index sizes of Aeetes and FaerieR are 10.6 MB and 6.9 MB, respectively. In DBWorld, the index sizes of Aeetes and FaerieR are 4.2 MB and 1.9 MB, respectively. While in USJob, the results are 113.2 MB and 54.3 MB, respectively. We can see that compared with FaerieR, the clustered inverted index of Aeetes has around twice larger size than FaerieR. The main reason is that we need to record the group relation for clustered index and use hashing tables to accelerate the query processing. But Aeetes can achieve much better performance by proper designing the search algorithm and utilizing the memory space smartly.

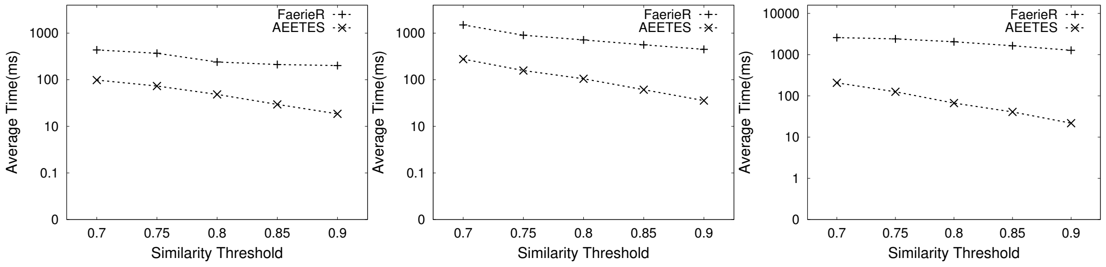
Optimizations Techniques We evaluate the filtering techniques proposed in Section 4. We implement four methods: Simple is the straightforward method to directly apply length and prefix filter by enumerating substrings; Skip is the method that adopts the

Table 2: Quality of similarity measures (P: short for Precision, R: short for Recall, F: short for F-measure).

θ	PubMed									DBWorld									USJob								
	Jaccard			FJ			JaccAR			Jaccard			FJ			JaccAR			Jaccard			FJ			JaccAR		
	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
0.7	0.24	0.73	0.36	0.35	0.73	0.47	0.96	0.89	0.92	0.24	0.67	0.35	0.39	0.68	0.50	0.92	0.91	0.92	0.20	0.71	0.31	0.37	0.77	0.50	0.94	0.94	0.94
0.8	0.14	0.88	0.24	0.34	0.77	0.47	0.95	0.93	0.94	0.24	0.89	0.38	0.36	0.84	0.50	0.90	0.94	**0.92**	0.20	0.83	0.32	0.29	0.85	0.43	0.92	0.97	0.94
0.9	0.12	0.92	0.21	0.28	0.85	0.42	0.95	0.98	**0.96**	0.23	0.92	0.37	0.35	0.90	0.50	0.88	0.93	0.90	0.18	0.90	0.30	0.25	0.86	0.39	0.92	0.98	**0.95**

PubMed	DBWorld	US Job
Entity: moschcowitz disease **Synonyms**: *moschcowitz disease ⇔ familial thrombotic thrombocytopenia purpura* *moschcowitz disease ⇔ thrombotic thrombocytopenic purpura* **Document**: "... the diagnostic challenge of acquired **thrombotic thrombocytopenic purpura** in children ..."	**Entity**: The University of Auckland NZ **Synonyms**: *NZ ⇔ New Zealand* *University ⇔ Univ.* **Document**: "... Gillian Dobbie, **The University of Aukland New Zealand**. Walid Aref, Purdue Univ. USA. Guoliang Li, Tsinghua University. Holger Pirk, MIT ..."	**Entity**: amazon database administrator **Synonyms**: *amazon ⇔ amzn* *database ⇔ databases* **Document**: "... position as a **databases administrator in amzn**. This position will give someone the ability to have input in to systems design ..."
Jaccard = 0.0 | FJ = 0.0 | **JaccAR = 1.0**	Jaccard = 0.38 | FJ = 0.54 | **JaccAR = 0.71**	Jaccard = 0.17 | FJ = 0.37 | **JaccAR = 0.75**

Figure 8: Three examples to illustrate the quality of similarity measures. The substrings with red font in the documents are marked as ground truth results.



(a) Pubmed (b) DB World (c) US Job

Figure 9: End-to-end performance

clustered inverted index to skip dissimilar groups; Dynamic is the method that dynamically computes the prefix for substrings in a document; Lazy is the method that generates candidates in a lazy manner.

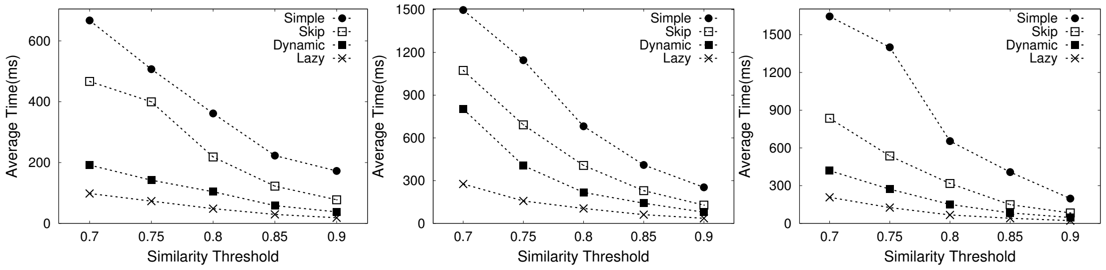
The results of the average extraction time are shown in Figure 10. We can see that Lazy achieves the best results. This is because it only needs to scan the inverted index of each token once. Although it requires some extra processing to allocate the candidate substrings with entities, the overhead can be very slight with proper implementation. The performance of Dynamic ranks second as it can dynamically maintain the prefix and does not need to compute from scratch. The reason it has worse performance than Lazy is that if a valid token exists in different windows, it needs to scan the same inverted index multiple times, which leads to heavy filter cost. Skip performs better than Simple as it utilizes the clustered inverted index to avoid visiting groups of entities that do not satisfy the requirement of length filter.

To further demonstrate the effect of filter techniques, we report in Figure 11 the average number of accessed entries in the inverted indexes, which provides a good metric to evaluate the filter cost. We can see that the results are consistent with those in Figure 10. For example, on the PubMed dataset when $\tau = 0.8$, Simple needs to access $326,631$ inverted index entries per document; Skip reduces the number to $126,895$; while the numbers of Dynamic and Lazy are $16,002$ and $6,120$, respectively.

Scalability Finally, we evaluate the scalability of Aeetes. In Figure 12, we vary the number of entities in each dataset and test the average search time for different threshold values. We can see that as the number of entities increased, Aeetes scales very well and achieves near linear scalability. For example, on the USJob dataset for $\tau = 0.75$, when the number of entities ranges from $200,000$ to $1,000,000$, the average running time is 43.26, 48.82, 62.71, 80.43 and 125.52 ms respectively.

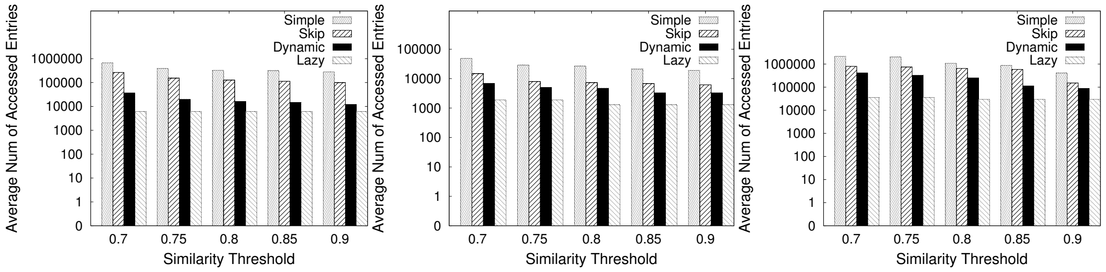
7 RELATED WORK

Approximate dictionary-based Entity Extraction Previous studies focusing on this problem only consider syntactic similarity. Chakrabarti et al. [8] proposed a hash-based method for membership checking. Wang et al. [35] proposed a neighborhood generation-based method for AEE with edit distance constraint, while Deng et al. [12] proposed a trie-based framework to improve the performance. Deng et al. [13] proposed Faerie, an all-purposed framework to support multiple kinds of similarity metrics in AEE problem. Wang et al. [34] addressed the local similarity search problem, which is a variant of AEE problem but with more limitations. All above methods only support syntactic similarity and cannot take synonyms into consideration.



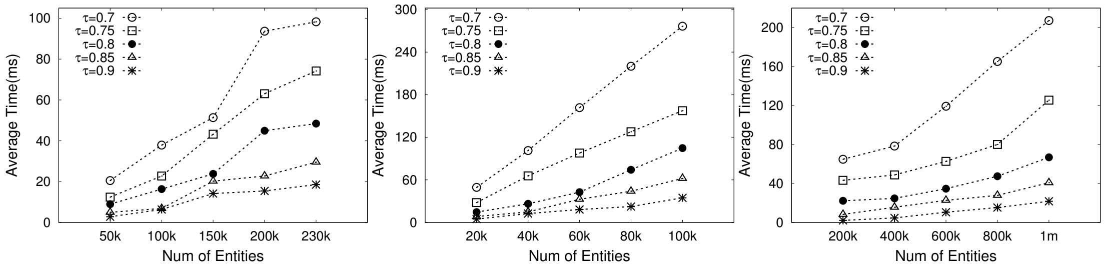
(a) Pubmed (b) DB World (c) US Job

Figure 10: Effect of Filtering techniques: Query Time



(a) Pubmed (b) DB World (c) US Job

Figure 11: Effect of Filtering techniques: Number of Accessed Entries



(a) Pubmed (b) DB World (c) US Job

Figure 12: Scalability: varying number of entities.

String Transformation and Synonym Discovery In some scenarios, the synonym rules may not exist, and it is impractical for a human to manually create a large set of rules. To solve this problem, some previous studies learn the rules using both supervised and unsupervised techniques. Arasu et al. [4] learned general syntactic rules from a small set of given matched records. Singh et al. improved the performance by leveraging language models [27] and semi-supervised learning [26]. Abedjan at al. [1] proposed the DataXFormer system to discover general transformations from web corpus. Singh et al. [28] addressed the same problem using program synthesis rules. Chakrabarti et al. [7] proposed novel similarity functions for synonym discovery from web scale data while He et al. [16] focused on finding synonyms in web tables. Qu et al. [22] discovered synonyms from text corpus with the help of knowledge base.

Entity Matching Entity matching has been a popular topic for decades. An extensive survey is conducted in [14]. Bilenko et al. [6] treated entity matching as a classification problem and proposed machine learning based solutions. Argrawal et al. [2] improved the quality of entity matching by considering errors in words and proposed efficient indexing techniques to improve performance. Wang et al. [33] proposed a learning-based framework to automatically learn the rules for entity matching. Firmani et al. [15] adopted a graph based model to develop an on-line framework for entity matching. Verroios et al. [30] integrated human ratings into entity matching and designed a crowdsourcing framework. Lin et al. [18] proposed a novel ranking mechanism to investigate the combinations of multiple attributes. Such studies mainly worked on collections of entities, while our problem requires to recognize approximate matching entities from documents. It could be an interesting direction of the future work to extend our framework to support other semantic similarity functions proposed here.

String Similarity Query Processing Approximate dictionary-based Entity Extraction (AEE) is a typical application in the field of string similarity query processing. There are also many studies on string similarity queries. Most of them only support syntactic similarity metrics. Among them some are designed for token-based similarity metrics, i.e. Jaccard, Cosine and Overlap, such as [9, 23, 36, 39, 40]; Others are designed for character-based similarity metrics i.e. edit distance, [17, 31, 37, 38]. Wang et al. [32]

combined above two categories of similarity metrics and proposed an efficient framework to support string similarity join. Some previous works tried to support synonym rules in the problem of string similarity join. They proposed some similarity functions to integrate the semantic of synonym rules into Jaccard similarities, such as *JaccT* [3], *SExpand* [19, 20] and *pkduck* [29]. However, they cannot be applied in the AEES problem as we have discussed in Section 2.

8 CONCLUSION AND FUTURE WORK

In this paper, we formally introduced the important problem of Approximate dictionary-based Entity Extraction with Synonyms and proposed an end-to-end framework Aeetes as the solution. We proposed a new similarity metrics to combine syntactic similarity metrics with synonyms and avoid the large overhead of on-line processing documents. We then designed and implemented a filter-and-verification strategy to improve the efficiency. Specifically, for the filtering step, we proposed a dynamic prefix computing mechanism and a lazy candidate generation method to reduce the filter cost. Experimental results on real world dataset demonstrated both the efficiency and effectiveness of our proposed framework.

For future work, we will (i) devise techniques to improve the verification step; (ii) extend our framework to support character-based similarity functions such as Edit Distance for tolerating typos in documents; (ii) support weighted synonym rules by assigning different weights to different rules; and (iii) integrate our techniques into open-source database systems.

REFERENCES

[1] Z. Abedjan, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. Dataxformer: A robust transformation discovery system. In *ICDE*, pages 1134–1145, 2016.

[2] P. Agrawal, A. Arasu, and R. Kaushik. On indexing error-tolerant set containment. In *SIGMOD*, pages 927–938, 2010.

[3] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *ICDE*, pages 40–49, 2008.

[4] A. Arasu, S. Chaudhuri, and R. Kaushik. Learning string transformations from examples. *PVLDB*, 2(1):514–525, 2009.

[5] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[6] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *SIGKDD*, pages 39–48, 2003.

[7] K. Chakrabarti, S. Chaudhuri, T. Cheng, and D. Xin. A framework for robust discovery of entity synonyms. In *KDD*, pages 1384–1392, 2012.

[8] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin. An efficient filter for approximate membership checking. In *SIGMOD*, pages 805–818, 2008.

[9] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

[10] S. Chaudhuri, V. Ganti, and D. Xin. Mining document collections to facilitate accurate approximate entity matching. *PVLDB*, 2(1):395–406, 2009.

[11] W. W. Cohen and S. Sarawagi. Exploiting dictionaries in named entity extraction: combining semi-markov extraction processes and data integration methods. In *SIGKDD*, pages 89–98, 2004.

[12] D. Deng, G. Li, and J. Feng. An efficient trie-based method for approximate entity extraction with edit-distance constraints. In *ICDE*, pages 762–773, 2012.

[13] D. Deng, G. Li, J. Feng, Y. Duan, and Z. Gong. A unified framework for approximate dictionary-based entity extraction. *VLDB J.*, 24(1):143–167, 2015.

[14] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.

[15] D. Firmani, B. Saha, and D. Srivastava. Online entity resolution using an oracle. *PVLDB*, 9(5):384–395, 2016.

[16] Y. He, K. Chakrabarti, T. Cheng, and T. Tylenda. Automatic discovery of attribute synonyms using query logs and table corpora. In *WWW*, pages 1429–1439, 2016.

[17] G. Li, D. Deng, J. Wang, and J. Feng. PASS-JOIN: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[18] C. Lin, J. Lu, Z. Wei, J. Wang, and X. Xiao. Optimal algorithms for selecting top-k combinations of attributes: theory and applications. *VLDB J.*, 27(1):27–52, 2018.

[19] J. Lu, C. Lin, W. Wang, C. Li, and H. Wang. String similarity measures and joins with synonyms. In *SIGMOD*, pages 373–384, 2013.

[20] J. Lu, C. Lin, W. Wang, C. Li, and X. Xiao. Boosting the quality of approximate string matching by synonyms. *ACM Trans. Database Syst.*, 40(3):15:1–15:42, 2015.

[21] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *PVLDB*, 9(9):636–647, 2016.

[22] M. Qu, X. Ren, and J. Han. Automatic synonym discovery with knowledge bases. In *KDD*, pages 997–1005, 2017.

[23] C. Rong, C. Lin, Y. N. Silva, J. Wang, W. Lu, and X. Du. Fast and scalable distributed set similarity joins for big data analytics. In *ICDE*, pages 1059–1070, 2017.

[24] S. Sarawagi. Information extraction. *Foundations and Trends in Databases*, 1(3):261–377, 2008.

[25] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, pages 743–754, 2004.

[26] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *PVLDB*, 9(10):816–827, 2016.

[27] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5(8):740–751, 2012.

[28] R. Singh, V. V. Meduri, A. K. Elmagarmid, S. Madden, P. Papotti, J. Quiané-Ruiz, A. Solar-Lezama, and N. Tang. Synthesizing entity matching rules by examples. *PVLDB*, 11(2):189–202, 2017.

[29] W. Tao, D. Deng, and M. Stonebraker. Approximate string joins with abbreviations. *PVLDB*, 11(1):53–65, 2017.

[30] V. Verroios, H. Garcia-Molina, and Y. Papakonstantinou. Waldo: An adaptive human interface for crowd entity resolution. In *SIGMOD*, pages 1133–1148, 2017.

[31] J. Wang, G. Li, D. Deng, Y. Zhang, and J. Feng. Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search. In *ICDE*, pages 519–530, 2015.

[32] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *ICDE*, pages 458–469, 2011.

[33] J. Wang, G. Li, J. X. Yu, and J. Feng. Entity matching: How similar is similar. *PVLDB*, 4(10):622–633, 2011.

[34] P. Wang, C. Xiao, J. Qin, W. Wang, X. Zhang, and Y. Ishikawa. Local similarity search for unstructured text. In *SIGMOD*, pages 1991–2005, 2016.

[35] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD*, pages 759–770, 2009.

[36] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[37] X. Yang, B. Wang, and C. Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD*, pages 353–364, 2008.

[38] X. Yang, Y. Wang, B. Wang, and W. Wang. Local filtering: Improving the performance of approximate queries on string collections. In *SIGMOD*, pages 377–392, 2015.

[39] Y. Zhang, X. Li, J. Wang, Y. Zhang, C. Xing, and X. Yuan. An efficient framework for exact set similarity search using tree structure indexes. In *ICDE*, pages 759–770, 2017.

[40] Y. Zhang, J. Wu, J. Wang, and C. Xing. A transformation-based framework for knn set similarity search. *IEEE Trans. Knowl. Data Eng.*, 2019.

Attendance Maximization for Successful Social Event Planning

Nikos Bikakis
Athens Univ. of Econ. & Bus.
Greece

Vana Kalogeraki
Athens Univ. of Econ. & Bus.
Greece

Dimitrios Gunopulos
University of Athens
Greece

ABSTRACT

Social event planning has received a great deal of attention in recent years where various entities, such as event planners and marketing companies, organizations, venues, or users in Event-based Social Networks, organize numerous social events (e.g., festivals, conferences, promotion parties). Recent studies show that "attendance" is the most common metric used to capture the success of social events, since the number of attendees has great impact on the event's expected gains (e.g., revenue, artist/brand publicity). In this work, we study the *Social Event Scheduling* (SES) problem which aims at identifying and assigning social events to appropriate time slots, so that the number of events attendees is maximized. We show that, even in highly restricted instances, the SES problem is NP-hard to be approximated over a factor. To solve the SES problem, we design three efficient and scalable algorithms. These algorithms exploit several novel schemes that we design. We conduct extensive experiments using several real and synthetic datasets, and demonstrate that the proposed algorithms perform on average half the computations compared to the existing solution and, in several cases, are 3-5 times faster.

1 INTRODUCTION

The event planning industry has grown enormously in the past decade, with large *event planning* and *marketing* companies (e.g., MKG, GPJ), organizing and managing a variety of social events (e.g., multi-themed festivals, promotion parties, conferences). In addition to companies, social events are also organized by *venues* (e.g., theaters, night clubs), *organizations* (e.g., ACM, TED), as well as *users* in *Event-based Social Networks* (e.g., Meetup, Eventbrite, Plancast).

The *Event Marketing Benchmark Report 2017*,[1] where marketing decision-makers from large organizations participate, indicates that "*attendance*" *is the most common metric used to measure the success of social events*, since the number of attendees has a great influence on the event's expected gains (e.g., revenue, artist publicity). Therefore, achieving maximum attendance is the *organizers first challenge*, as also indicated in the *Event Marketing Trends 2018* study.[2]

Examples of events organization include large festivals and conferences where a large number of (multi-themed) events are organized over several stages and sessions attracting several thousands of people. For example, *Summerfest Festival* has performances from over 800 bands, attracting more than 800K people each year. Beyond music concerts, numerous multi-themed events take place, ranging from art-makings and theatrical performances to fitness activities and parties. In such scenarios, successful event planning is extremely challenging, since various factors need to be taken into account, such as the *large number of events* and available *time slots*, the *diversity of events' themes* and *user interests*, the presence of *overlapped events*, the *available resources* (e.g., available stages), etc.

Assume the following scenario. On Monday two events are scheduled to take place during a festival: (1) a *rock concept* from 19:00 to 22:00; and (2) a *fashion show* between 19:00 and 21:00. Additionally, from 18:00 to 20:00 a music concert of a *rock singer* is taking place in a nearby (competing) venue. Consider that Alice enjoys listening to *rock music*, and is a *fashion lover*. Although Alice is interested in all three events, she is only able to attend one of them.

In this work, we study the *Social Event Scheduling* (SES) problem [4]. *Given a set of candidate events, a set of time intervals and a set of users, SES assigns events to time intervals, in order to maximize the overall number of participants.* The assignments are determined by considering several events' and users' factors, such as user preferences and habits, events' spatiotemporal conflicts, etc.

Recently, several studies have been published examining the problem of assigning *users* (i.e., participants) to a set of *pre-scheduled events* in Event-based Social Networks [6, 12, 26–29, 31]. The objective in these works is to find the *user-event assignments that maximizes the satisfaction of the users*. Here, in contrast to existing works, we study a substantially different problem. Briefly, instead of assigning users to events, we assign *events* to *time intervals*. The objective here is to find the *event-time assignments that maximize the number of event's attendees*. More or less, the SES problem studies the "satisfaction" (e.g., revenue, publicity) of the entities involved in event organization (e.g., organizer, artist, sponsors, services' providers). In other words, SES is an "*organizer-oriented*" problem, while the existing works are "*participant-oriented*". Overall, the objective, the solution and the setting of the SES problem are substantially different from the related works.

The SES problem was recently introduced in [4] where a greedy algorithm was proposed. In the proposed solution, in each assignment selection, the algorithm recomputes (i.e., updates) the scores for a large number of assignments. Additionally, in each selection the algorithm has to examine (e.g., check for validity) all the assignments. The aforementioned result to poor performance of this solution. In this work, we design three efficient and scalable algorithms which are implemented on top of the following novel schemes. First, we propose an *incremental updating* scheme in which a reduced number of score computations are performed in an incremental manner. Further, we design an *assignment organization* scheme which significantly reduces the number of assignments that are examined. Finally, an *assignment selection policy* is proposed, minimizing the impact of performing a part of the required score computations, on the quality of the results. In our extensive experiments, we illustrate that the proposed algorithms perform about half the computations and, in several cases, are 3-5× faster compared to the method proposed in [4].

[1] www.certain.com/blog/certain-presents-the-event-marketing-benchmark-report-spring-2017
[2] https://welcome.bizzabo.com/event-marketing-2018

Series ISSN: 2367-2005

10.5441/002/edbt.2019.12

Event	Location
e_1	Stage 1
e_2	Stage 1
e_3	Room A
e_4	Stage 2

(a) Candidate Events

$t_1 = \langle$Friday 8–11pm$\rangle$

$t_2 = \langle$Sat. 6–9pm$\rangle$

(b) Time Intervals

c_1 $\langle$Friday 6–9pm$\rangle$, $t_{c_1} = t_1$

c_2 $\langle$Sat. 8–10pm$\rangle$, $t_{c_2} = t_2$

(c) Competing Events

	Event Interest				Comp. Ev. Interest		Activ. Prob.	
User	e_1	e_2	e_3	e_4	c_1	c_2	t_1	t_2
u_1	0.9	0.3	0	0.6	0.8	0.3	0.8	0.5
u_2	0.2	0.6	0.1	0.6	0.4	0.7	0.5	0.7

(d) Users

Figure 1: Running example (4 Candidate events, 2 Intervals, 2 Competing events, 2 Users)

Further, examining the theoretical aspects of the SES problem, we study its approximation, showing that even in highly restricted instances, it is NP-hard to be approximated over a factor larger than $(1 - \epsilon)$.

Contributions. The main contributions of this work are summarized as follows: (1) We show that the SES problem is NP-hard to be approximated over a factor larger than $(1 - \epsilon)$. (2) We design three efficient and scalable approximation algorithms. These algorithms outperform the existing algorithm by exploiting a series of schemes that we develop. (3) We conduct an detailed experimental analysis using several real and synthetic datasets.

2 SOCIAL EVENT SCHEDULING PROBLEM

In this section we first define the *Social Event Scheduling* (SES) problem; and then we study its approximation. In what follows, we present a simple example that introduces the main entities involved in SES problem.

Example 1. [*Running Example*] Figure 1 outlines our running example involving four *candidate events* (e_1–e_4), two *time intervals* (t_1, t_2), two *competing events* (c_1, c_2), and *two users* (u_1, u_2).

The *location* of each *candidate event* is presented in Figure 1a We notice that both e_1 and e_2 are going to be hosted at Stage 1. Hence, these events cannot be scheduled to take place during the same time period. Figure 1c presents the *competing events* along with the time periods during which these are scheduled to take place. For example, c_1 is schedule to take place on Friday between 6:00 and 9:00pm (at a nearby competing venue). Further, in Figure 1b we observe that there is the candidate *time interval* t_1 defined on the same day between 8:00 and 11:00pm. Thus, due to overlapping time periods, a user cannot attend both c_1 and a candidate event that will be possibly scheduled to take place during t_1.

Finally, Figure 1d shows, for each user, the *interest values* (i.e., affinity) for the events, as well as the *social activity probability* (e.g., based on user habits) during the time periods defined by the two intervals. For example, u_1 has high social activity probability (equals to 0.8) at t_1, since Friday night u_1 does not work and usually goes out and participates in social activities.

2.1 Problem Definition

In our problem, we assume an ***organizer*** (e.g., company, venue) managing the events' organization. Each organizer possesses a number of ***available resources*** $\theta \in \mathbb{R}^+$. These are abstractions used to refer to staff, materials, budget or any other means related to event organization.

Further, let $\mathcal{T}$ be a set of ***candidate time intervals***, representing time periods that are available for organizing events.

Assume a set $\mathcal{E}$ of available events to be scheduled, referred as ***candidate events***. Each $e \in \mathcal{E}$ is associated with a ***location*** ℓ_e representing the place (e.g., a stage) that is going to host the event. Further, each event e requires a specific amount of resources $\xi_e \in \mathbb{R}_0^+$ for its organization, referred as ***required resources***.

An ***assignment*** α_e^t denotes that the candidate event $e \in \mathcal{E}$ is scheduled to take place at $t \in \mathcal{T}$. An event ***schedule*** $\mathcal{S}$ is a set of assignments, where there exist no two assignments referring to the same event. Given a schedule $\mathcal{S}$, we denote as $\mathcal{E}(\mathcal{S})$ the set of all candidate events that are scheduled by $\mathcal{S}$, i.e., $\mathcal{E}(\mathcal{S}) = \{e \mid a_e^t \in \mathcal{S}\}$; and $\mathcal{E}_t(\mathcal{S})$ the candidate events that are scheduled by $\mathcal{S}$ to take place at t (i.e., assigned to t). Further, for a candidate event $e \in \mathcal{E}(\mathcal{S})$, we denote as $t_e(\mathcal{S})$ the time interval on which $\mathcal{S}$ assigns e.

A *schedule* $\mathcal{S}$ is said to be ***feasible*** if the following constraints are satisfied: (1) $\forall t \in \mathcal{T}$ holds that $\nexists e_i, e_j \in \mathcal{E}_t(\mathcal{S})$ with $\ell_{e_i} = \ell_{e_j}$ (***location constraint***); and (2) $\forall t \in \mathcal{T}$ holds that $\sum_{\forall e \in \mathcal{E}_t(\mathcal{S})} \xi_e \leq \theta$ (***resources constraint***). In analogy, an *assignment* α_e^t is said to be ***feasible*** if the aforementioned constraints hold for t. Further, we call ***valid assignment***, an assignment α_e^t when the assignment is *feasible* and $e \notin \mathcal{E}(\mathcal{S})$.

Let C be a set of ***competing events***, with $C \cap \mathcal{E} = \varnothing$. As competing events we define events that have already been scheduled by third parties, and will possibly attract potential attendees of the candidate events. Based on its scheduled time, each competing event $c \in C$ is associated with a time interval $t_c \in \mathcal{T}$. Further, as C_t we denote the competing events that are associated with the time interval t.

Consider a set of users $\mathcal{U}$, for each ***user*** $u \in \mathcal{U}$ and event $h \in \mathcal{E} \cup C$, there is a function $\mu\colon \mathcal{U} \times (\mathcal{E} \cup C) \to [0, 1]$, denoted as $\mu_{u,h}$, that models the ***interest*** of user u over h. The interest value (i.e., affinity) can be estimated by considering a large number of factors (e.g., preferences, social connections).

Moreover, we define the ***social activity probability*** σ_u^t, representing the probability of user u participating in a social activity at t. This probability can be estimated by examining the user's past behavior (e.g., number of check-ins).

Assume a user u and a candidate event $e \in \mathcal{E}$ that is scheduled by $\mathcal{S}$ to take place at time interval t; $\rho_{u,e}^t$ denotes the ***probability of u attending e at t***. Considering the *Luce's choice theory* [17], the probability $\rho_{u,e}^t$ is influenced by the social activity probability σ of u at t, and the interest μ of u over e, C_t and $\mathcal{E}_t(\mathcal{S})$. We define the *probability of u attending e at t* as:

$$\rho_{u,e}^t = \sigma_u^t \, \frac{\mu_{u,e}}{\sum\limits_{\forall c \in C_t} \mu_{u,c} + \sum\limits_{\forall p \in \mathcal{E}_t(\mathcal{S})} \mu_{u,p}} \tag{1}$$

Furthermore, considering all users $\mathcal{U}$, we define the ***expected attendance*** for an event e scheduled to take place at t as:

$$\omega_e^t = \sum_{\forall u \in \mathcal{U}} \rho_{u,e}^t \tag{2}$$

The ***total utility*** for a schedule $\mathcal{S}$, denoted as $\Omega(\mathcal{S})$, is computed by considering the expected attendance over all scheduled events:

$$\Omega(\mathcal{S}) = \sum_{\forall e \in \mathcal{E}(\mathcal{S})} \omega_e^{t_e(\mathcal{S})} \tag{3}$$

The *Social Event Scheduling* (SES) problem is defined as follows: [3]

Social Event Scheduling Problem (SES). Given an positive *integer* k, a set of candidate *time intervals* $\mathcal{T}$; a set of *competing events* C; a set of *candidate events* $\mathcal{E}$; and a set of *users* $\mathcal{U}$; our goal is to find a *feasible schedule* $\mathcal{S}_k$ that determines how to assign k candidate events such that the *total utility* Ω is maximized; i.e., $\mathcal{S}_k = \arg\max \Omega(\mathcal{S})$ and $|\mathcal{S}| = k$.

Note that, by performing trivial modifications to the algorithms proposed here, additional factors and constraints to those defined in SES, can be easily handled. For example, include event's organization cost/fee (to define a "profit-oriented" version of the SES problem), associate events with duration, or define weights over the users (e.g., based on their influence).

2.2 Approximation Hardness

Here, we show that even in highly restricted instances the SES problem is NP-hard to be approximated over a factor. Therefore, SES does not admit a *Polynomial Time Approximation Scheme* (PTAS).

Theorem 1. There exists an $\epsilon > 0$ such that it is NP-hard to approximate the SES problem within a factor larger than $(1 - \epsilon)$. Thus, SES does not admit a PTAS. [4]

PROOF SKETCH. In our proof we reduce the 3-Bounded 3-Dimensional Matching problem (3DM-3) [10] to a restricted instance of SES. The following is an instance of the 3DM-3 problem. Given a set $T \subseteq X \times Y \times Z$, with $|X| = |Y| = |Z| = n$, $|T| = m$ and with each element of $X \cup Y \cup Z$ appearing at most 3 times as a coordinate of an element of T. A matching in T is a subset $M \subseteq T$, such that no elements in M agree in any coordinate. In our proof, we exploit the following result: [10] showed that in 3DM-3 there exists an $\epsilon_0 > 0$ such that it is NP-hard to decide whether an instance has a matching of size n or if every matching has size at most $(1 - \epsilon_0)n$.

Consider the following *associations between* 3DM-3 and SES: edges g in T to time intervals; and elements in X, Y, Z to candidate events, with required resources $\xi = 1$. Let the aforementioned candidate events form a set E_1 (i.e., $|E_1| = 3n$).

Further, in the proof, we consider the following *restricted instance of* SES: (1) The *available resources* are three. (2) There are no *location constraints*. (3) There is only one *competing event* in each time interval. (4) The *social activity probability* is the same for each user and time interval. (5) The *users* are as many as the candidate events. (6) There is a set E_2 that contains $m - n$ additional (w.r.t. E_1) candidate events, with $\xi = 3$. Thus, the candidate events $\mathcal{E}$ in the restricted instance is $\mathcal{E} = E_1 \cup E_2$, with $E_1 \cap E_2 = \varnothing$. (7) Regarding the *interest function* we assume two disjoint sets of users $|U_1| = 3n$ and $|U_2| = m - n$, a well as the following: ($7a$) Each user $u_1 \in U_1$ likes only one event $e_1 \in E_1$ (as a result, each e_1 is liked only by one user u_1), with $\mu_{u_1,e_1} = 0.25$. ($7b$) Regarding the *competing events* and the users U_1 we have the following. Fix a positive constant $\delta < \frac{1}{12}$. Let $u_p \in U_1$ the user that likes the event e_p, where e_p corresponds to the element y_p in 3DM-3. Then, if y_p is included in the edge g_t (i.e., $y_p \in g_t$), the interest of u_p in the competing event c that appears in the interval t (which in 3DM-3 corresponds to edge g_t) is $\mu_{u_p,c} = 0.25(0.75 - \delta)/(0.25 + \delta)$ and 0.75, otherwise. ($7c$) Each user $u_2 \in U_2$ likes only one event $e_2 \in E_2$ (as a result, each e_2 is liked only by one user u_2), with $\mu_{u_2,e_2} = 0.75$. ($7d$) For each competing event c and user $u_2 \in U_2$, we have $\mu_{u_2,c} = 0$.

We can verify that, for each matching in 3DM-3, we can obtain a schedule in SES with total utility $3(0.25 + \delta)$, by assigning 3 events of E_1 in a same interval. Then, if 3DM-3 has a matching of size n, we can verify that the total utility in SES is $3n(0.25 + \delta) + m - n$. Otherwise, if every matching has size at most $(1 - \epsilon_0)n$, the total utility in SES is $1 - \frac{\epsilon_0 - 12\delta\epsilon_0}{12\delta + 3} < 1 - \frac{1}{3}\epsilon_0$. ■

3 ALGORITHMS

SES is known to be strongly NP-hard, even in highly restricted instances [4]. Due to its hardness, it is computationally prohibitive to find an optimal solution even in small problem sizes. Particularly, in the worst case, we have to enumerate an exponential number of possible assignments, where each assignment requires always $|\mathcal{U}|$ computations. For example, the greedy algorithm proposed in [4], in several cases in our experiments, took more than 5 hours to solve the problem in the default parameters setting, while more than 31 hours in larger settings. To this end, to cope with the hardness of the SES problem we design three efficient and scalable approximation algorithms which perform about half the computations and, in several cases, are 3-5 times faster compared to the method proposed in [4].

3.1 Existing Solution

Here, we outline the previously proposed algorithm. First, we define the assignment score. Given a schedule $\mathcal{S}$ and an assignment α_r^t, as ***assignment score*** (also referred as ***score***) of an assignment α_r^t, denoted as $\alpha_r^t.S$, we define the *gain* in the expected attendance by including α_r^t in $\mathcal{S}$. The assignment score (based on Eq. 2) is defined as:

$$\alpha_r^t.S \;=\; \sum_{\substack{\forall e_j \in \\ \mathcal{E}_t(\mathcal{S}) \cup \{r\}}} \omega'^{\,t}_{e_j} \;-\; \sum_{\substack{\forall e_i \in \\ \mathcal{E}_t(\mathcal{S})}} \omega^t_{e_i} \tag{4}$$

Given a set of assignments, the term ***top assignment*** refers to the assignment with the largest score.

In [4], a simple greedy algorithm is outlined, referred here as ALG. This method starts by initially generating assignments between all pairs of events and intervals. Then, in each iteration, the assignment with the largest score (i.e., top assignment) is selected. After selecting an assignment, a subset of the assignment's scores need to be updated. Recall that, the assignment's score is defined w.r.t. the events assigned in the assignment's interval (Eq. 4). Hence, when an assignment α_e^t is selected, then the scores of the assignments referring to interval t need to be recomputed (updated). The time complexity of ALG is $O(|\mathcal{U}||C| + |\mathcal{E}||\mathcal{T}||\mathcal{U}| + k|\mathcal{E}||\mathcal{T}| + k|\mathcal{E}||\mathcal{U}| - k^2|\mathcal{T}| - k^2|\mathcal{U}|)$; and the space complexity is $O(|\mathcal{E}||\mathcal{T}|)$.

Example 2. [*ALG Algorithm*] Based on our running example, Figure 2 outlines the execution of the ALG algorithm. In this, as well as in the rest of the examples, we assume that $k = 3$. That is, three out of four events have to be scheduled. Each row represents the selection of a single assignment. Rows include the assignment scores (Eq. 4), as well as the selected assignment (presented in bold red

[3] Several of the problem's involved factors (e.g., user interest, activity and attendance probability) can be computed using event-based mining methods, e.g., [5, 7, 13, 19, 21, 36, 37, 40, 41]. However, this is beyond the scope of this work.

[4] Due to lack of space, we only include proof sketches, while in simple cases, the proof sketch is also omitted.

	$\alpha_{e_1}^{t_1}$	$\alpha_{e_2}^{t_1}$	$\alpha_{e_3}^{t_1}$	$\alpha_{e_4}^{t_1}$	$\alpha_{e_1}^{t_2}$	$\alpha_{e_2}^{t_2}$	$\alpha_{e_3}^{t_2}$	$\alpha_{e_4}^{t_2}$		**Select**	**Update**
①	0.59	0.52	0.10	~~0.64~~	0.53	0.57	0.09	**0.66**	→	$\alpha_{e_4}^{t_2}$	$\alpha_{e_1}^{t_2}$ $\alpha_{e_2}^{t_2}$ $\alpha_{e_3}^{t_2}$
②	**0.59**	0.52	0.10	–	~~0.34~~	0.16	0.03	–	→	$\alpha_{e_1}^{t_1}$	$\alpha_{e_3}^{t_1}$
③	–	~~0.52~~	0.05	–	–	**0.16**	0.03	–	→	$\alpha_{e_2}^{t_2}$	–

Figure 2: ALG algorithm example

font) and the assignments that have to be updated after the selection. Initially (i.e., ① selection), the algorithm selects the assignment with the largest score (i.e., $\alpha_{e_4}^{t_2}$). Thus, after this selection the assignments referring to e_4 have to be omitted (marked with /), and the assignments referring to t_2 have to be updated. After the second selection, the algorithm has to update only $\alpha_{e_3}^{t_1}$ since $\alpha_{e_2}^{t_1}$ is no longer feasible (marked with ×) due to location constraint. Note that, for the sake of simplicity, the resources constraint has been omitted from the running example. Finally, the schedule contains $\alpha_{e_4}^{t_2}$, $\alpha_{e_1}^{t_1}$ and $\alpha_{e_2}^{t_2}$.

3.2 Incremental Updating Algorithm (INC)

The ALG algorithm proposed in [4], has the following shortcoming: (1) each time ALG selects an assignment, it has to recompute (i.e., update) from scratch all the scores for all the assignments associated with the selected assignment's interval. This process is referred to assignment updating or simply as updates; and (2) in each step, ALG has to examine (and traverse) all the available assignments in order to perform its tasks (e.g., select assignment, perform updates).

Considering the aforementioned issues, we design the *Incremental Updating algorithm* (INC). Regarding the first issue, INC exploits an *incremental updating scheme*, performing incremental assignment updates. Incremental updating allows INC, to provide the same solution as ALG, while, in each step, INC performs only a part of the updates (i.e., score computations). Regarding the second issue, INC attempts to reduce the number of assignments that should be examined in each step, i.e., search space. To this end, we devise an *assignment organization* that takes into account the incremental updating scheme. In several cases in our experiments, INC *is more than three times faster* than the existing algorithm.

Essentially, INC follows a similar assignment selection process to ALG, selecting the top assignment in each step, in a greedy fashion. However, in INC the assignments' update process has been designed based on the introduced incremental updating scheme.

3.2.1 Incremental Updating

In the proposed scheme, the updates are computed in an incremental manner, where after each assignment selection only a part of the updates are performed. As a result, during the algorithm execution, some of the assignments may not be up-to-date.

An assignment is denoted as ***updated***, if its score has been computed by considering all the (previously) selected assignments, and ***not updated*** otherwise. In analogy, a set of assignment is referred as *updated*, when all its assignments are updated, and ***partially updated***, otherwise.

The basic idea of our scheme is that we can determine a subset of the not updated assignments that have to be updated before each selection. First we show that, from the available assignments $\mathcal{A}$, we can find a set $\mathcal{B} \subseteq \mathcal{A}$ which includes the next algorithm selection χ. Then, we also show that the not updated assignments included in $\mathcal{B}$ are the only not updated assignments that have to be updated in order to find χ.

Assignments Sorted by Score ("+" / "–" : *Updated/Not updated*)										**Select**	**Φ**	**Update**
①	$\boldsymbol{\alpha_{e_4}^{t_2+}}$	$\cancel{\alpha_{e_4}^{t_1+}}$	$\alpha_{e_1}^{t_1+}$	$\alpha_{e_2}^{t_2+}$	$\alpha_{e_1}^{t_2+}$	$\alpha_{e_2}^{t_1+}$	$\alpha_{e_3}^{t_1+}$	$\alpha_{e_3}^{t_2+}$	→	$\alpha_{e_4}^{t_2}$	$\alpha_{e_1}^{t_1}.S$	–
②	–	–	$\boldsymbol{\alpha_{e_1}^{t_1+}}$	$\alpha_{e_2}^{t_2-}$	$\cancel{\alpha_{e_1}^{t_2-}}$	$\alpha_{e_2}^{t_1+}$	$\alpha_{e_3}^{t_1+}$	$\alpha_{e_3}^{t_2-}$	→	$\alpha_{e_1}^{t_1}$	∅	$\alpha_{e_2}^{t_2}$
③	–	–	–	$\boldsymbol{\alpha_{e_2}^{t_2+}}$	–	$\xcancel{\alpha_{e_2}^{t_1+}}$	$\alpha_{e_3}^{t_1-}$	$\alpha_{e_3}^{t_2-}$	→	$\alpha_{e_2}^{t_2}$	$\alpha_{e_3}^{t_2}.S$	–

Figure 3: Incremental updating scheme example

In order to specify $\mathcal{B}$, we use a numeric *bound* Φ. As shown next, the value of Φ is the score of the *top*, *updated* and *valid* assignment of $\mathcal{A}$.

Proposition 1. Let Φ be the score of the *top*, *updated* and *valid* assignment of the available assignments $\mathcal{A}$. Then, the next selected assignment χ is one of the assignments that in $\mathcal{A}$ have score larger or equal to Φ; i.e., $\chi \in \mathcal{B}$, where $\mathcal{B} = \{\alpha_e^t \in \mathcal{A} \mid \alpha_e^t.S \geq \Phi\}$.

Proof Sketch. First, we show that the score of a not updated assignment is always larger or equal to the score of the assignment resulted by its update. Note that the proof for this is not trivial for arbitrary numbers of candidate and competing events. Based on the aforementioned, the not updated assignments of $\mathcal{A}$, having score lower than Φ, also have score lower than Φ in $\mathcal{A}'$, where $\mathcal{A}'$ be the set of assignments resulting from $\mathcal{A}$ by updating its not updated assignments. Further, the score of each updated assignment of $\mathcal{A}$ remains the same in $\mathcal{A}'$. So, both the updated and the not updated assignments of $\mathcal{A}$ have scores lower than Φ; their scores in $\mathcal{A}'$ also remain lower than Φ. Thus, the Proposition 1 holds. ■

Based on Proposition 1, since χ is included in $\mathcal{B}$, we can easily verify that, χ is the *top* assignment of $\mathcal{B}'$, where $\mathcal{B}'$ results from $\mathcal{B}$ by updating its not updated assignments. Thus, in order to find χ, we have to update the not updated assignments of $\mathcal{B}$. Based on the aforementioned, the following corollary describes the incremental updating process.

Corollary 1. In each step, in order to select the next assignment, only the not updated assignments having score larger or equal to Φ have to be updated.

Example 3. [*Incremental Updating Scheme*] Figure 3 illustrates the utilization of the incremental updating scheme. For clarity of presentation, we omit the assignment scores since these are the same as in Example 2. To better understand the procedure, in each row the assignments are presented in descending order, based on their score. The +/- notation is used to denote that the assignment is *updated*, or *not updated*, respectively. After the first selection, Φ is equal to $\alpha_{e_1}^{t_1}.S$ (i.e., top, updated and valid assignment), and all the assignments referring to t_2 change to *not updated*. Further, since all the not updated assignments have score lower than Φ, none of the assignments have to be updated. Then (② selection), after selecting $\alpha_{e_1}^{t_1}$, all the assignments become not updated; so Φ becomes unavailable. Next, the algorithm updates $\alpha_{e_2}^{t_2}$ and sets Φ equal to its score (0.16). In the last selection, since the current Φ is larger than the scores (0.10 and 0.9) of the not updated assignments $\alpha_{e_3}^{t_1}$ and $\alpha_{e_3}^{t_2}$, the algorithm does not have to update it. Compared to the ALG algorithm (Example 2) which performs four updates, our scheme performs only one.

3.2.2 Assignments Organization over Incremental Updating

In each step, the algorithm needs to examine and traverse all the available assignments, in order to perform the following main tasks: (1) select the top assignment; (2) perform updates; and (3) maintain the bound. In order to accomplish these tasks, for each of the available assignments, the algorithm has to perform numerous computations. Indicatively, it has to check validity constraints, compare scores, consider bounds and possibly compute the new assignment score, etc.

Given the above, we introduce an *interval-based assignment organization* that incorporates with our incremental updating scheme. This organization attempts to reduce the number of assignments that are accessed and examined, i.e., *search space*. Using our organization, in most cases in our experiments, INC *examines slightly more than half assignments* compared to the existing algorithm.

Search Space Reduction in Assignment Updates. Here, we describe how we reduce the assignments that should be examined in order to perform the updates. An interval-based assignment organization allows to access at the interval-level, only the assignments that should be examined for update. Adopting this organization in a simple (not incremental) updating process, like in ALG, in each step, the algorithm needs only to access the assignments of one interval in order to performs the updates. On the other hand, in the incremental updating setting, several intervals become partially updated during the execution. In this scenario, in order to identify the assignments that need to be updated, we have to examine all the assignments included in partially updated intervals. As a result, in our setting, a simple interval-based organization will not be effective, since it will allow to skip accessing only the updated intervals.

Beyond ignoring only the updated intervals, in order to further reduce the search space, we have to be able to identify (and skip) the partially updated intervals whose assignments are not going to be updated. In our organization scheme, this is addressed by defining a score over each interval. Particularly, for each interval $t \in \mathcal{T}$, a value M_t is defined, where M_t is equal to the score of the *top*, *updated* and *valid* assignment of interval t. Exploiting M_t, we can directly identify the partially updated intervals that have to be accessed through the updating process. Particularly, it is easy to verify that we have to access all the partially updated intervals $t \in \mathcal{T}$ for which $M_t \leq \Phi$.

Search Space Reduction in Assignment Selection and Bound Maintenance. The organization described so far allows to reduce the search space during the assignment updating process. However, after performing the updates, the algorithm has to accomplish also the tasks of selecting the next assignment and maintaining the bound Φ, which in turn enforce the examination of all the available assignments. In what follows, we show how to perform all the tasks without examining any further assignments beyond the ones examined during the updating process.

The intuition is that, in each step, only a subset of the assignments is updated, while the rest remain the same as in the previous step, referred here as *static*. Therefore, it is reasonable to assume that the algorithm is able to accomplish all of its tasks by utilizing "information" previously captured from the static assignments $\mathcal{W}$. So, if this "information" is known, then, after performing the updates, we can ignore $\mathcal{W}$ (i.e., avoid access). As shown next, this "information" can be captured by two numeric values I_χ and I_Φ determined from $\mathcal{W}$. Briefly, I_χ is exploited to specify the next selected assignment χ, and I_Φ to compute the new Φ.

Proposition 2. Given a set of static assignments $\mathcal{W}$. Let I_χ and I_Φ be the scores of the *first* and the *second larger top, updated* and *valid* assignment of $\mathcal{W}$, respectively. Then, if I_χ and I_Φ are know, the algorithm can ignore $\mathcal{W}$.

Algorithm 1. INC (k, $\mathcal{T}$, $\mathcal{E}$, $\mathcal{C}$, $\mathcal{U}$)

```
Input: $k$: number of scheduled events; $\mathcal{T}$: time intervals;
       $\mathcal{E}$: candidate events; $\mathcal{C}$: competing events; $\mathcal{U}$: users;
Output: $\mathcal{S}$: feasible schedule containing $k$ assignments
Variables: $\mathcal{L}_i$: assignment list for interval $i$; $\Phi$: bound; $\mathcal{M}$: top, valid and updated assign list
1  $\mathcal{S} \leftarrow \varnothing$;  $\mathcal{M} \leftarrow \varnothing$;  $\mathcal{L}_i \leftarrow \varnothing$,  $\mathcal{L}_i.U \leftarrow$ updated $1 \leq i \leq |\mathcal{T}|$
2  foreach $(e, t) \in \mathcal{E} \times \mathcal{T}$ do                                  //generate assignments
3      compute $\alpha_e^t.S$;  $\alpha_e^t.U \leftarrow$ updated;                                 //by Eq. 4
4      insert $\alpha_e^t$ into $\mathcal{L}_t$                                              //initialize assignment lists
5      $\mathcal{M}_{[t]} \leftarrow$ getBetterAssgn($\mathcal{M}_{[t]}$, $\alpha_e^t$)   //initialize $\mathcal{M}$ with the top assignment from each interval
6  while $|\mathcal{S}| < k$ do
7      $\alpha_{e_p}^{t_p} \leftarrow$ getTopAssgn($\mathcal{M}$)                     //select the top, valid & updated assignment
8      insert $\alpha_{e_p}^{t_p}$ into $\mathcal{S}$                                        //insert into schedule
9      remove $\alpha_{e_p}^{t_p}$ from $\mathcal{L}_{t_p}$;  $\mathcal{L}_{t_p}.U \leftarrow$ prtl updated;          //update $\mathcal{L}_{t_p}$ status
10     $\alpha_e^t.U \leftarrow$ prtl updated, $\forall \alpha_e^t \in \mathcal{L}_{t_p}$
11     foreach $\alpha_e^t \in \mathcal{M}$ do                        //update $\mathcal{M}$ list based on selected assignment
12         if $t = t_p$ then
13             $\alpha_e^t \leftarrow \varnothing$                                            //i.e., $\alpha_e^t.S \leftarrow -\infty$
14         else if $e = e_p$ then
15             $\alpha_e^t \leftarrow$ getTopAssgn($\mathcal{L}_i$)
16     $\Phi \leftarrow$ score of top $\mathcal{M}$                                          //set bound
17     for $i = 1$ to $|\mathcal{T}|$ do                                             //update assignments
18         if $\mathcal{L}_i.U =$ prtl updated and $\mathcal{M}_{[i]}.S \leq \Phi$ then            //check for updates
19             foreach $\alpha_e^t \in \mathcal{L}_i$ do
20                 if $\alpha_e^t$ is not valid then
21                     remove $\alpha_e^t$ from $\mathcal{L}_i$
22                 else if $\alpha_e^t.U =$ prtl updated and $\alpha_e^t.S \geq \Phi$ then
23                     compute new $\alpha_e^t.S$;  $\alpha_e^t.U \leftarrow$ updated;                  //by Eq. 4
24                     $\mathcal{M}_{[i]} \leftarrow$ getBetterAssgn($\mathcal{M}_{[i]}$, $\alpha_e^t$)          //update top assignment
25                     $\Phi \leftarrow$ getBetterAssgn($\Phi$, $\alpha_e^t.S$)                       //update bound
26             if all $\alpha_e^t \in \mathcal{L}_i$ is updated then $\mathcal{L}_i.U =$ updated;
27 return $\mathcal{S}$
```

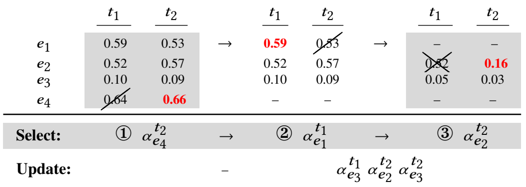
In our interval-based scheme implementation, the static assignments $\mathcal{W}$ correspond to a set of static intervals $\mathcal{T}_\mathcal{W} \subseteq \mathcal{T}$. Both I_χ and I_Φ can be be directly computed based on the values of M_t of the static intervals $\mathcal{T}_\mathcal{W}$. Particularly, $I_\chi = \max\limits_{\forall t \in \mathcal{T}_\mathcal{W}} M_t$ and $I_\phi = \max\limits_{\forall t \in \{\mathcal{T}_\mathcal{W} \setminus t_\chi\}} M_t$, where t_χ is the interval of I_χ. Therefore, based on Proposition 2, in each step, the algorithm needs to access only intervals that have been updated (i.e., subset of partially updated intervals).

Assignment Organization Summary. To sum up, the presented organization allows: (1) the reduction of the assignments that are examined during the updating process; and (2) skipping the examination of any further assignments beyond the updated ones.

3.2.3 INC Algorithm Description & Analysis

Algorithm Description. Algorithm 1 describes the INC algorithm; INC receives the same inputs as ALG. Additionally, INC employs $|\mathcal{T}|$ lists, with each list $\mathcal{L}_i$ filled with the assignments of interval i. Further, each assignment α_e^t and list $\mathcal{L}_i$, use a flag U (denoted as $\alpha_e^t.U$) to define its update status. Finally, the algorithm uses a list $\mathcal{M}$ that holds the top, valid and updated assignments of each interval. Initially, like ALG, INC calculates the scores for all possible assignments (*loop in line* 2). At the same time, the assignments are inserted into the corresponding list $\mathcal{L}_i$ (*line* 3-4). Note that, the *getBetterAssgn* function returns the assignment with the larger score.

Then, at the beginning of each iteration (*line* 6), the algorithm selects the top assignment from $\mathcal{M}$, and inserts it into schedule (*lines* 7-8). Also, the algorithm has to revise the information related to update status (*lines* 9-10). After the assignment's selection phase, the algorithm performs score updates. Initially, the bound Φ is defined by the top updated assignment (*line* 16). Then, the algorithm traverses the lists $\mathcal{L}_i$, using as upper bound the $\mathcal{M}_{[i]}.S$, to identify the lists that have to be checked for updates (*line* 18). From the verified lists, the



	t_1	t_2		t_1	t_2		t_1	t_2
e_1	0.59	0.53	→	**0.59**	~~0.53~~	→	–	–
e_2	0.52	0.57		0.52	0.57		~~0.52~~	**0.16**
e_3	0.10	0.09		0.10	0.09		0.05	0.03
e_4	~~0.64~~	**0.66**		–	–		–	–
Select:	① $\alpha_{e_4}^{t_2}$		→	② $\alpha_{e_1}^{t_1}$		→	③ $\alpha_{e_2}^{t_2}$	
Update:			–		$\alpha_{e_3}^{t_1}$ $\alpha_{e_2}^{t_2}$ $\alpha_{e_3}^{t_2}$			

Figure 4: HOR algorithm example

Algorithm 2. HOR (k, $\mathcal{T}$, $\mathcal{E}$, C, $\mathcal{U}$)

```
Input: $k$: number of scheduled events;  $\mathcal{T}$: time intervals;
       $\mathcal{E}$: candidate events;  $C$: competing events;  $\mathcal{U}$: users;
Output: $\mathcal{S}$: feasible schedule containing $k$ assignments
Variables: $\mathcal{L}_i$: assignment lists for interval $i$;    $\mathcal{M}$: top assignments list
1  $\mathcal{S} \leftarrow \varnothing$;  $\mathcal{M} \leftarrow \varnothing$;
2  while $|\mathcal{S}| < k$ do
3      $\mathcal{L}_i \leftarrow \varnothing$;   $1 \le i \le |\mathcal{T}|$
4      foreach $(e, t) \in \{\mathcal{E} \backslash \mathcal{E}(\mathcal{S})\} \times \mathcal{T}$ do          //generate assignments
5          if $\alpha_e^t$ is valid then
6              compute $\alpha_e^t.S$;                                  //by Eq. 4
7              insert $\alpha_e^t$ into $\mathcal{L}_t$                          //initialize assignment lists
8              $\mathcal{M}_{[t]} \leftarrow$ getBetterAssgn($\mathcal{M}_{[t]}$, $\alpha_e^t$)  //insert into $\mathcal{M}$ the top assignment from each interval
9      while $\mathcal{M} \neq \varnothing$ and $|\mathcal{S}| < k$ do                //select assignments from $\mathcal{M}$
10         $\alpha_{e_p}^{t_p} \leftarrow$ popTopAssgn($\mathcal{M}$)
11         if $e_p \notin \mathcal{S}$ then
12             insert $\alpha_{e_p}^{t_p}$ into $\mathcal{S}$                            //insert into schedule
13         else          //assignment is invalid; select the top valid from $\mathcal{L}_{t_p}$ and insert it into $\mathcal{M}$
14             insert the top assignment $\alpha_{e_i}^{t_p}$ from $\mathcal{L}_{t_p}$ into $\mathcal{M}$, s.t. $e_i \notin \mathcal{S}$
15 return $\mathcal{S}$
```

algorithm performs incremental updates (*lines* 22-23), updating also $\mathcal{M}$ and Φ (*lines* 24-25).

INC vs. ALG Solution. The following proposition states that both INC and ALG, return the same solutions.

Proposition 3. INC and ALG always return the same solution.

Complexity Analysis. The cost in the first loop (*line* 2) is $O(|\mathcal{E}||\mathcal{T}||\mathcal{U}|)$. Note that, each assignment score (Eq. 4) is computed in $O(|\mathcal{U}|)$. Then, the second loop (*line* 6) performs k iterations. The overall cost for the getTopAssgn operation (*line* 7) is $O(\sum_{i=0}^{k-1} |\mathcal{T}|)$. Further, the loop in *line* 11 performs $|\mathcal{T}|$ iterations, whereas in the worst case, in $|\mathcal{T}| - 1$ of these iterations, INC performs a getTopAssgn operation which costs $O(|\mathcal{E}| - (i + 1))$, with $0 \le i \le k - 1$. Thus, the overall cost for this getTopAssgn operation is $O(\sum_{i=0}^{k-1}(|\mathcal{T}| - 1)(|\mathcal{E}| - i - 1))$. Next (*line* 17), in the worst case, all the not updated assignments are updated (same as ALG). Note that, in the *Best case*, INC does not perform any computations for assignment updates, while in ALG, in every case, the cost for updates is $O(|\mathcal{U}| \sum_{i=0}^{k-2} |\mathcal{E}| - i - 1)$.

Hence, in the worst case, the overall cost of INC is same as ALG; i.e., $O(|\mathcal{U}||C| + |\mathcal{E}||\mathcal{T}||\mathcal{U}| + k|\mathcal{E}||\mathcal{T}| + k|\mathcal{E}||\mathcal{U}| - k^2|\mathcal{T}| - k^2|\mathcal{U}|)$. Finally, the space complexity is $O(|\mathcal{E}||\mathcal{T}| + |\mathcal{T}|)$.

3.3 Horizontal Assignment Algorithm (HOR)

In this section we propose the *Horizontal Assignment algorithm* (HOR), which in general, is more efficient than the ALG and INC algorithms and in most cases in practice provides same solutions. The goal of HOR is twofold. First, to reduce the number of updates by performing only a part of the required updates; and, at the same time, minimize the impact of not regular updates, in the solution quality. In HOR both of these issues are realized by the policy that are employed to select the assignments. In our experiments, in several cases HOR *is around 5 and 3 times faster* than ALG and INC, respectively. Also, in more than 70% of our experiments, HOR reports the same solution as ALG, while in the rest cases the difference is marginal.

Horizontal Selection Policy. The key idea of HOR is that it adopts a selection policy, named *horizontal selection policy*, that selects assignments in a "horizontal" fashion. In this policy, in each iteration the algorithm selects a set of assignments consisting of one assignment from each interval. Particularly, the top assignment from each interval is selected. This way, essentially, a layer of assignments is generated in each iteration. For example, consider the scenario where $k > |\mathcal{T}|$ (and the assignments are feasible in all cases). In the first iteration, HOR will assign one event in each interval; equally, in the n^{th} iteration, n events will have been assigned in each interval.[5]

This policy allows HOR to avoid performing updates after each assignment selection. This holds, since, in each iteration at most one assignment per interval is selected. Thus, during an iteration, when an assignment is selected for an interval, the algorithm stops examining the selection of further assignments for this interval. As a result, there is no need to perform any updates until the end of each iteration, where the scores for all the assignments have to recomputed.

In what follows we outline the intuition behind the horizontal selection policy. Considering that the users' attendance is shared between the events that take place during the same or overlapping time intervals. The horizontal policy assigns the same number of events to each interval, ignoring the possibility that it may be preferable to assign a different number of events to some intervals.

Example 4. [*HOR Algorithm*] Figure 4 outlines the execution of the HOR algorithm, presenting assignments following an interval-based organization. Initially, HOR selects the assignment with the largest score. Since the first selected assignment refers to t_2, in the next selection, HOR will select the top assignment from t_1. After selecting assignments from both intervals, HOR has to update all the available assignments in order to perform the third selection. Therefore, HOR performs three updates, whereas it finds the same schedule as ALG/ INC.

Algorithm Description. Algorithm 2 presents the pseudocode of HOR. Note that, since the horizontal selection policy performs selections at the interval-level, we implement interval-based assignment organization. Finally, similarly to INC, HOR uses the $|\mathcal{T}|$ lists $\mathcal{L}_i$ and the list $\mathcal{M}$. At the beginning of each iteration the algorithm calculates the scores for all possible assignments (*loop in line* 4) and initializes $\mathcal{M}$ (*line* 8). In the next phase (*line* 9), the algorithm selects the assignments based on $\mathcal{M}$. Particularly, in each step, the top valid assignment from $\mathcal{M}$ is selected.

3.3.1 HOR Algorithm Analysis

ALG vs. HOR Score Computations Analysis. Here we study the number of score computations, comparing the ALG and the HOR algorithms. The following proposition specifies the cases where HOR performs less score computations than ALG.

Proposition 4. HOR performs less score computations than ALG when $k \le |\mathcal{T}|$ or $|\mathcal{E}| < \frac{k}{2}(3|\mathcal{T}| + 1)$.

PROOF SKETCH. In case that $k \le |\mathcal{T}|$, HOR computes only the scores for the initial assignments (i.e., $|\mathcal{T}||\mathcal{E}|$) without performing any updates. In case that $k > |\mathcal{T}|$, HOR computes the same initial assignments, as well as the scores for $\sum_{i=0}^{(k/|\mathcal{T}|)-1} |\mathcal{T}|(|\mathcal{E}| - i|\mathcal{T}| - |\mathcal{T}|)$ updates. On the other hand, in the ALG algorithm, in any case, we

[5] With an exception in the last iteration l, in which if $k \bmod |\mathcal{T}| \neq 0$, then, $|\mathcal{T}| - (k \bmod |\mathcal{T}|)$ intervals will have $l - 1$ events.

have to compute the same number of initial assignments as in HOR, as well as $k|\mathcal{E}| + \frac{k}{2} - \frac{k^2}{2} - |\mathcal{E}|$ updates. ■

From Proposition 4 we can infer that in "rational/typical" (i.e., real-world) scenarios, HOR perform fewer computations than ALG. Particularly, even in cases that $k > |\mathcal{T}|$, it should also hold that $|\mathcal{E}| \geq \frac{k}{2}(3|\mathcal{T}| + 1)$ in order for HOR to perform more computations. Considering the setting of our problem, the second relation is difficult to hold in practice. For example, consider the scenario where $|\mathcal{T}| = 10$ and $k = 20$. Then, in order for HOR to perform more computations, it should hold that $|\mathcal{E}| \geq 301$, which seems unrealistic due to the noticeable difference between the number of scheduled ($k = 10$) and candidate events ($|\mathcal{E}| \geq 301$).

Worst Case w.r.t. k and $|\mathcal{T}|$. Considering the horizontal selection policy, beyond the size of the input (e.g., $|\mathcal{E}|$, $|\mathcal{U}|$, k), the number of computations in HOR is also influenced by the ratio between parameters k and $|\mathcal{T}|$. During the execution, HOR performs $\lceil k/|\mathcal{T}| \rceil$ iterations. At the beginning of each iteration, it computes the scores according to which $|\mathcal{T}|$ assignments are selected. In the last iteration, if $k \bmod |\mathcal{T}| \neq 0$, then only $k \bmod |\mathcal{T}|$ assignments need to be selected, while the algorithm has already performed the computations that are required to select $|\mathcal{T}|$ assignments. Thus, in this case, HOR has performed more computations than the ones required for selecting its assignments. For example, assume that we have $|\mathcal{T}| = 10$ and $k = 11$. In this case, the score computations performed by HOR are the same as in the case that we have $k = 20$.

The case in which the difference between the number of computed assignment selections and the number of the selections that need to be performed is maximized, is referred as *worst case w.r.t. k and T*. Note that, even in the worst case, in our experiments, HOR outperforms INC in several cases.

Proposition 5. In HOR, the worst case w.r.t. k and $|\mathcal{T}|$ occurs when $k > |T|$ and $k \bmod |\mathcal{T}| = 1$.

Complexity Analysis. In the first while loop (*line* 2), HOR performs $\lceil k/|\mathcal{T}| \rceil$ iterations. In each iteration, in the worst case, there are $|\mathcal{E}| - i|\mathcal{T}|$ available events, where $0 \leq i \leq \frac{k}{|\mathcal{T}|}$. Thus, in each iteration, it computes $|\mathcal{T}|(|\mathcal{E}| - i|\mathcal{T}|)$ assignments (*line* 6). The overall cost for computing the assignments is $O(|\mathcal{U}| \sum_{i=0}^{k/|\mathcal{T}|} |\mathcal{T}|(|\mathcal{E}| - i|\mathcal{T}|))$. Further, in each iteration of the first loop (*line* 2), the nested loop (*line* 9) performs $|\mathcal{T}|$ iterations (referred as nested iterations). In each nested iteration, in the worst case, HOR performs two popTopAssgn operations (*lines* 10 & 14). The cost for the first and the second popTopAssgn operations is $O(|\mathcal{T}|)$ and $O(|\mathcal{E}| - i|\mathcal{T}|)$, respectively. Hence, in sum, the cost for popTopAssgn operations is $O(\sum_{i=0}^{k/|\mathcal{T}|} |\mathcal{T}|(|\mathcal{T}| + |\mathcal{E}| - i|\mathcal{T}|))$. The worst case can occur when all the assignments contained in $\mathcal{M}$ refer to the same event.

Thus, the overall cost of HOR is $O(|\mathcal{U}||C| + |\mathcal{E}||\mathcal{T}||\mathcal{U}| + k|\mathcal{E}||\mathcal{U}| + |\mathcal{T}|^2 - k|\mathcal{T}||\mathcal{U}| - k^2|\mathcal{U}|)$. Finally, the space complexity is $O(|\mathcal{E}||\mathcal{T}| + |\mathcal{T}|)$.

3.4 Horizontal Assignment with Incremental Updating Algorithm (HOR-I)

This section introduces the *Horizontal Assignment with Incremental Updating algorithm* (HOR-I). HOR-I combines the basic concepts from the INC and HOR algorithms, in order to further reduce the computations performed by HOR. Particularly, HOR-I adopts an incremental updating scheme, similar to INC (Sect. 3.2.1), as well as the horizontal selection policy employed by HOR (Sect. 3.3). Note that, in several cases, in our experiment HOR-I *performs about half computations and is up to two times faster compared to* HOR.

Algorithm 3. HOR-I (k, $\mathcal{T}$, $\mathcal{E}$, C, $\mathcal{U}$)

```
Input: $k$: number of scheduled events;  $\mathcal{T}$: time intervals;
       $\mathcal{E}$: candidate events;  $C$: competing events;  $\mathcal{U}$: users;
Output: $\mathcal{S}$: feasible schedule containing $k$ assignments
Variables: $\mathcal{L}_i$: assignment lists for interval $i$;   $\Phi$: bound; $\mathcal{M}$: top assignments list

1  $\mathcal{S} \leftarrow \varnothing$;  $\mathcal{M} \leftarrow \varnothing$;  $\mathcal{L}_i \leftarrow \varnothing \;\; 1 \leq i \leq |\mathcal{T}|$
2  while $|\mathcal{S}| < k$ do
3      if $\mathcal{S} = \varnothing$ then                                         //first iteration
4          foreach $(e, t) \in \{\mathcal{E} \backslash \mathcal{E}(\mathcal{S})\} \times \mathcal{T}$ do                //generate assignments
5              compute $\alpha_e^t.S$;   $\alpha_e^t.U \leftarrow$ updated;                      //by Eq. 4
6              insert $\alpha_e^t$ into $\mathcal{L}_t$                           //initialize assignment lists
7              $\mathcal{M}_{[t]} \leftarrow$ getBetterAssgn($\mathcal{M}_{[t]}$, $\alpha_e^t$)
8      else                                            //incremental assignments updating
9          for $i = 1$ to $|\mathcal{T}|$ do
10             $\Phi \leftarrow 0$                                              //initialize bound
11             foreach $\alpha_e^t \in \mathcal{L}_i$ do
12                 if $\alpha_e^t$ is valid then
13                     if $\alpha_e^t.S \geq \Phi$ then
14                         compute new $\alpha_e^t.S$;   $\alpha_e^t.U \leftarrow$ updated;
15                         $\Phi \leftarrow$ getBetterAssgn($\Phi$, $\alpha_e^t.S$)              //update bound
16                     else
17                         $\alpha_e^t.U \leftarrow$ prtl updated                      //partially updated
18                 else
19                     remove $\alpha_e^t$ from $\mathcal{L}_i$
20             $\mathcal{M}_{[i]} \leftarrow \Phi$                                  //update top assignment
21     while $\mathcal{M} \neq \varnothing$ and $|\mathcal{S}| < k$ do                  //select assignments from $\mathcal{M}$
22         $\alpha_{e_p}^{t_p} \leftarrow$ popTopAssgn($\mathcal{M}$)
23         if $e_p \notin \mathcal{S}$ then
24             insert $\alpha_{e_p}^{t_p}$ into $\mathcal{S}$                              //insert into schedule
25         else           //select the top, valid & updated from $\mathcal{L}_{t_p}$ and insert it into $\mathcal{M}$
26             $\alpha_{e_p}^{t_p} \leftarrow$ top & updated assignment from $\mathcal{L}_{t_p}$, s.t. $e_p \notin \mathcal{S}$
27             if $\alpha_{e_p}^{t_p} = \varnothing$ and $\exists \alpha_e^t \in \mathcal{L}_{t_p}$ s.t. $\alpha_e^t$ is valid then
28                     ·                                   //incremental updates in interval $p$
29                     ·  Same as lines 10 to 20, with $i = p$
30                     ·
31 return $\mathcal{S}$
```

Recall that, at the beginning of each iteration, HOR calculates the scores for all available assignments. Particularly, in the first iteration, the algorithm generates the assignments and calculates their (initial) scores, while in each of the following iterations the scores for all the assignments are updated. On the other hand, after the first iteration, HOR-I instead of updating all the assignments, uses an incremental updating scheme. This way, in each iteration, a reduced number of updates are performed.

Note that since the updates are performed after the first iteration, it is obvious that HOR-I is identical to HOR in cases where only one iteration is required (i.e., $k \leq |\mathcal{T}|$).

Example 5. [*HOR-I Algorithm*] The difference between HOR-I and HOR example (Example 4), appears at the third selection, where from t_2 only the $\alpha_{e_2}^{t_2}$ is updated. This happens because after updating $\alpha_{e_2}^{t_2}$, its score (0.16) is the current bound for this interval. Then, when checking $\alpha_{e_3}^{t_2}$ for update, its score (0.09) is lower than the bound, so there is no need to update it. Hence, HOR-I performs two of the three updates performed by HOR.

Algorithm Description. Algorithm 3 presents HOR-I; HOR-I uses the same structures as HOR, as well as a bound Φ. At the first iteration (*loop in line* 4), as is the case with HOR, HOR-I generates the assignments and initializes $\mathcal{M}$. In the next iterations (*loop in line* 9), it performs incremental updates for each interval, determining a different bound Φ for each interval. Then, similarly to HOR, HOR-I performs the assignment selection based on $\mathcal{M}$ (*loop in line* 21). In contrast to HOR, HOR-I has also to examine the update status of the assignments. In case that there is not a valid and updated assignment left on an interval (*lines* 27-30), HOR-I has to perform incremental updates in this interval.

3.4.1 HOR-I Algorithm Analysis

HOR-I Solution & Worst Case w.r.t. k and $|\mathcal{T}|$. The following propositions state that both HOR-I and HOR return the same solutions and also have the same worst case w.r.t. k and $|\mathcal{T}|$.

Proposition 6. HOR-I and HOR always return the same solution.

Proposition 7. In HOR-I, the worst case w.r.t. k and $|\mathcal{T}|$ occurs when $k > |T|$ and $k \bmod |\mathcal{T}| = 1$.

Complexity Analysis. In the worst case, the computation cost in HOR-I, is the same as HOR. Particularity, in the worst case, the bound employed by HOR-I cannot prevent any of the assignment updates (*line* 9). This case arises, when the assignments in each interval list $\mathcal{L}_i$ are sorted in ascending order by its score, and there are no assignments having the same score. Thus, the computation cost for HOR-I is $O(|\mathcal{U}||C| + |\mathcal{E}||\mathcal{T}||\mathcal{U}| + k|\mathcal{E}||\mathcal{U}| + |\mathcal{T}|^2 - k|\mathcal{T}||\mathcal{U}| - k^2|\mathcal{U}|)$. Note that, in the *Best case* HOR-I does not perform any computations for updates, while in HOR, in any case (where $k > |\mathcal{T}|$), the cost for updates is $O(\sum_{i=0}^{(k/|\mathcal{T}|)-1} |\mathcal{U}||\mathcal{T}|(|\mathcal{E}| - i|\mathcal{T}| - |\mathcal{T}|))$. Finally, the space complexity is $O(|\mathcal{E}||\mathcal{T}| + |\mathcal{T}|)$.

4 EXPERIMENTAL ANALYSIS

4.1 Setup

Datasets. In our experimental evaluation we present the results from four datasets, *two real* and *two synthetic*. The *first real* is the *Meetup dataset* (Meetup) from [21], which contains data from California, and is the dataset used in [4]. We follow the same approach as in [4, 26–28, 31], in order to define the interest of a user to an event. After preprocessing, we have the Meetup dataset containing 42,444 users and about 16K events.

The next *real dataset* (Concerts) which is the largest, is related to music and provided from Yahoo! ("Music user ratings of musical tracks, albums, artists and genres dataset"). Concerts is used to demonstrate the scenario of music festival organization. Particularly, Concerts contains data for several music entities (i.e., tracks, albums, artists, genres), as well as ratings of users over these entities. In this dataset, we consider albums to represent the events (i.e., music concerts). We select the albums that are associated with at least one genre, which results to 89K albums. Further, as users we select the users that have rated at least 10 genres, which result to 379,391 users.

In order to compute user interest over the albums, we consider the users ratings over the music genres, as well as the genres that are associated with the music albums. Let a user u, R_u denote the set of ratings r_i over genres, where $r_i \in [0, 1]$ is the rating over the i genre. Also, let G_a be a set of genres associated with a music album a. Here, we define the interest of a user u over the album a as $(\sum_{\forall g \in G_a} r_g)/|G_a|$, where $r_g = 1$ if the genre g is not specified in R_u. Note that, similar results are reported using alternative methods, such as setting $r_g = 0$ for genres not specified in R_u, or considering only the common user-album genres.

Finally, regarding *synthetic datasets* (Table 1), we generate the users' *interest values* for the events, following the *three distribution types* examined in the related literature [12, 26–28, 31]: *Uniform* (Unf), *Normal* (Nrm) and *Zipfian* (Zip). Note that, for brevity, the results for the Normal distribution are not presented here since they are similar to Uniform. Further for Zipfian, we present only the results with parameter equal to 2 which are similar to those of 1 and 3.

Table 1: Parameters

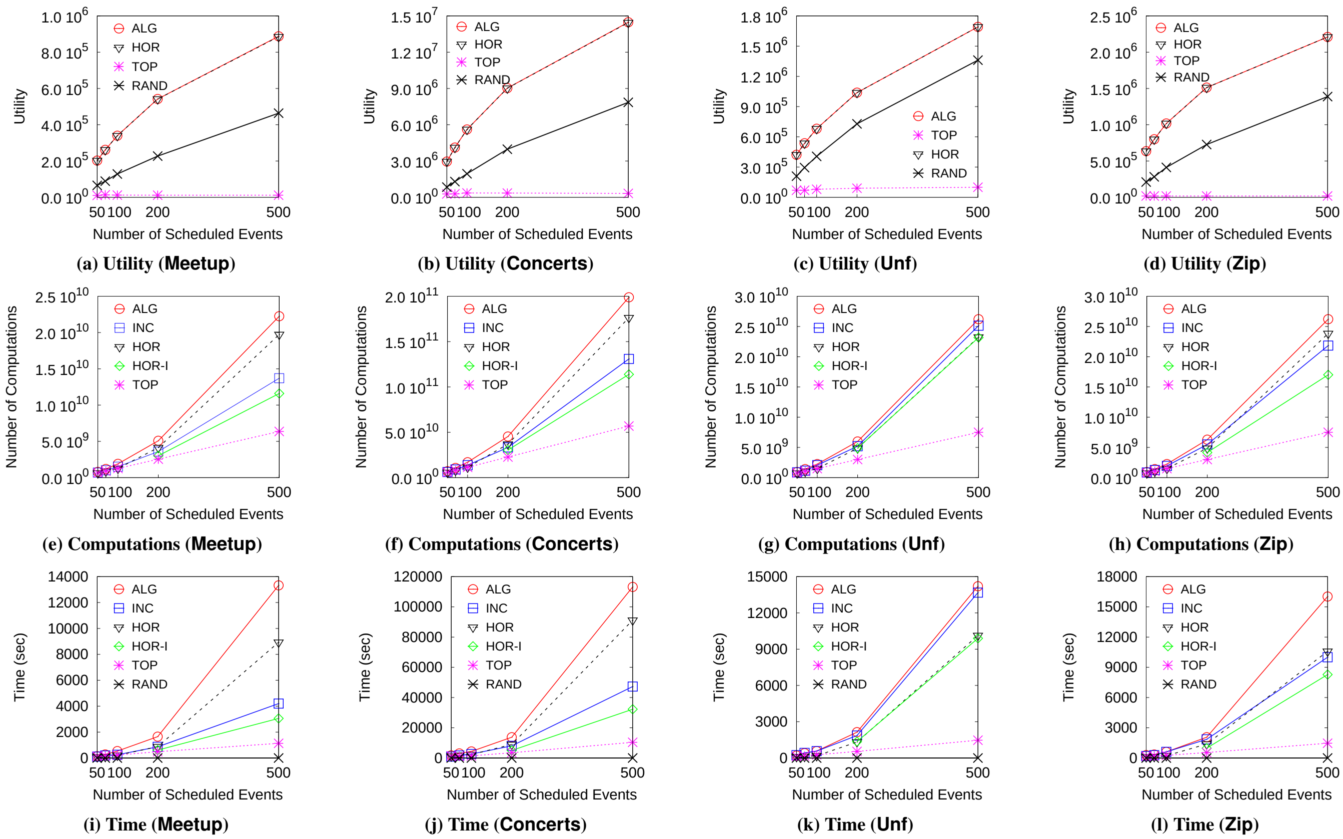
Description (Parameter)	Values
Synthetic & Real Datasets	
Num of scheduled events (k)	50, 70, **100**, 200, 500
Num of candidate events ($\|\mathcal{E}\|$)	k, $\mathbf{2k}$, $3k$, $5k$, $10k$
Num of time intervals ($\|\mathcal{T}\|$)	$\frac{k}{5}$, $\frac{k}{2}$, k, $\mathbf{\frac{3k}{2}}$, $2k$, $3k$
Competing events per interval	Uniform: [1, 4], [1, 8], **[1, 16]**, [1, 32], [1, 64]
Num of available locations	5, 10, **25**, 50, 70
Num of available resources (θ)	10, **20**, 30, 50, 100
Num of required resources per event (ξ_e)	Uniform: $[1, \frac{\theta}{4}]$, $\mathbf{[1, \frac{\theta}{3}]}$, $[1, \frac{\theta}{2}]$, $[1, \frac{3\theta}{4}]$, $[1, \theta]$
Distribution of social activity probability (σ_u^t)	**Uniform**, Normal (0.5, 0.25)
Synthetic Datasets	
Num of users ($\|\mathcal{U}\|$)	10K, **50K**, 100K, 500K, 1M
Distribution of interest ($\mu_{u,e}$)	Uniform, Normal (0.5, 0.25), Zipfian: 1, 2, 3

Parameters. Table 1 summarizes the parameters that we vary and the range of values examined; default values are presented in bold.

Adopting the same setting as in the related works [4, 12, 26–28, 31], we set the the default and maximum value of the of *scheduled events* k, to 100 and 500, respectively. In order to select the values for the number of *competing events per interval*, we analyze the two Meetup datasets used in our evaluation [21]. Particularly, we are interested in the number of events taking place during overlapped time intervals. As event interval we consider the period spanning from one hour before to two hours after the event's scheduled time. From the analysis, we found that, on average, 8.1 events are taking place during overlapping intervals. Therefore, in the default setting the number of competing events per interval is selected by a uniform distribution having 8.1 as mean value. Further, we vary the mean value from 2 to 32 (Table 1). In our experiments, the reported results are similar to the default setting, with the utility score being slightly lower for larger numbers of competing events, as expected (results are omitted due to lack of space).

In order to select the default and the examined values for the *number of available events' locations*, we consider the percentage of pairs of events that are spatio-temporally conflicting, as specified in [26]. Also, we vary the *number of required resources* for each event, as well as the number of *available resources* (Table 1). Here, as resources we consider agents (i.e., organizer's staff). In the aforementioned experiment, the methods are marginally affected by the examined parameters. Thus, due to lack of space, the results are omitted. Finally, regarding the *social activity probability*, we use Uniform and Normal distribution. Note that, the results for Normal distribution are not presented here, since they are the same as in Uniform.

Methods. In our evaluation we study the three proposed algorithms (INC, HOR, HOR-I), as well as the ALG algorithm proposed in [4]. Further, we include the baselines used in [4]. The first, denoted as TOP, computes the assignment scores for all the events and selects the events with top-k score values. Since TOP computes the scores only once, TOP is always performing the minimum number of computations. The second, RAND assigns events to intervals, randomly. Note that, since the objective, the solution and the setting of our problem are substantially different (see Sect. 1) from the related works [6, 12, 26–29, 31], the existing methods cannot be used to solve the SES problem.



(a) Utility (Meetup) **(b) Utility (Concerts)** **(c) Utility (Unf)** **(d) Utility (Zip)**

(e) Computations (Meetup) **(f) Computations (Concerts)** **(g) Computations (Unf)** **(h) Computations (Zip)**

(i) Time (Meetup) **(j) Time (Concerts)** **(k) Time (Unf)** **(l) Time (Zip)**

Figure 5: Varying the number of scheduled events k

Metrics & Implementation. In each experiment, we measure: (1) the *total utility score*; (2) the *execution time*; and (3) the *number of computations* for assignment scores ($|\mathcal{U}|$ per assignment score). All algorithms were written in C++ and the experiments were performed on an 2.67GHz Intel Xeon E5640 with 32GB of RAM.

4.2 Results

Recall that, the HOR and HOR-I algorithms return the same solutions (i.e., equal utilities); the same also holds for the ALG and INC algorithms. Hence, only the former utility plots are presented. Further, in cases where $k < |\mathcal{T}|$, the HOR-I algorithm is identical to HOR (Sect. 3.4); thus, in these cases only the HOR results are included in the plots.

4.2.1 Effect of the Number of Scheduled Events

In the first experiment, we study the effect of varying the number of scheduled events k.

Utility. In terms of utility (Fig. 5a–5d), we observe that, in all cases, our HOR method has the same utility score as the ALG (details are presented in Sect. 4.2.8). Further, the difference between RAND and the other methods increases, as k increases. This is reasonable considering the fact that the larger the k, the larger the number of "better", compared to random, selected assignments.

Regarding the Unf dataset (Fig. 5c), we observe the following. First, the difference between the random and the other methods is the smallest one, compared to the other datasets. Second, the difference between the methods is roughly the same for all k values. The reason for the aforementioned is that the uniform distribution results to utility values being very close, for all assignments. Thus, an effective assignment selection cannot significantly improve the overall utility.

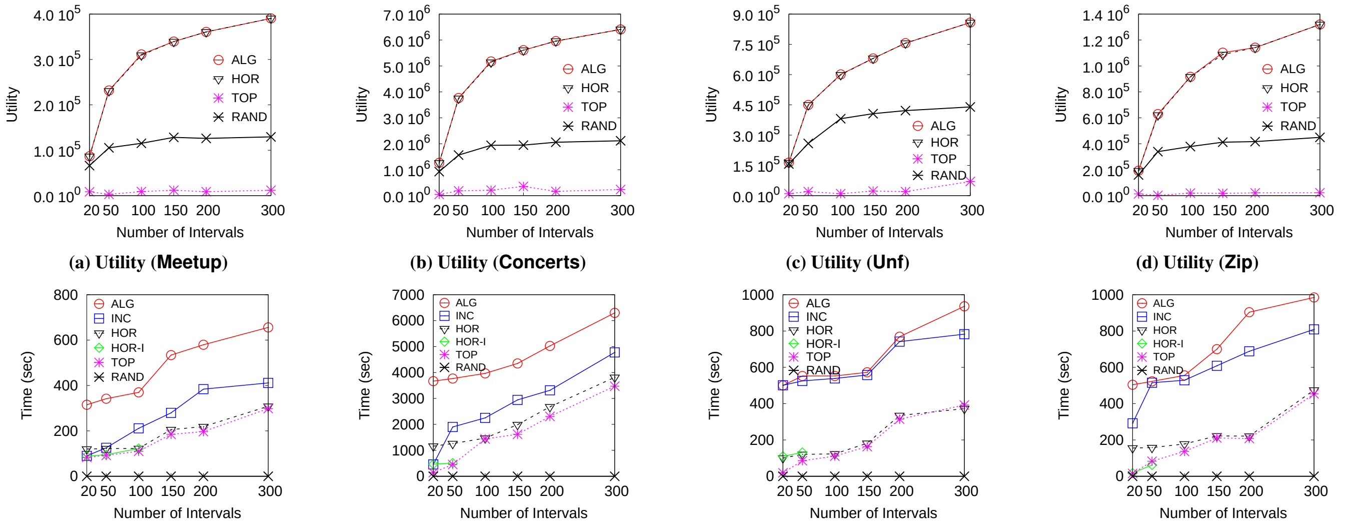
Finally, we can observe that TOP reports considerably low utility scores in all cases (which is also observed in the following experiments). The reason is that TOP assigns the events to a small number of intervals. This results to a large number of parallel events which "share" assigned interval's utility.

Computations. Regarding the number of computations (Fig. 5e–5h), we should mention that, the computations that are performed due to updates increases with k, while the number of initially computed scores is the same for all k. Thus, the difference between the ALG and our methods increases with k. Overall, we can observe that, in all cases, ALG reports the larger number of computations, while HOR-I the lower (excluding the TOP baseline).

Regarding our methods, comparing HOR with HOR-I, we can observe that the difference between our HOR versions increases with k, with HOR-I performing noticeably less computations compared to HOR for large k. An exception is reported in Unf dataset (Fig. 5g), in which all bound-based methods (INC, HOR-I) report poor performance (as explained later).

Further, comparing the HOR with INC, for $k < 200$, HOR performs less computations than INC. However, in the remaining cases where $k \geq 200$, INC outperforms HOR (with an exception in Unf). The reason why INC performs better than HOR for $k \geq 200$ is that, in these cases HOR performs update computations; while, for the cases where $k \leq |\mathcal{T}|$ only the initial computations are performed.

In the Unf dataset (Fig. 5g), we can observe that the bound-based methods (i.e., INC, HOR-I) demonstrate poor performance, with HOR-I performing same as HOR, and INC performing worse than both of them. The reason lies to the uniform distribution, where, as previously stated, the scores are very close for all assignments. As



(a) Utility (Meetup)

(b) Utility (Concerts)

(c) Utility (Unf)

(d) Utility (Zip)

(e) Time (Meetup)

(f) Time (Concerts)

(g) Time (Unf)

(h) Time (Zip)

Figure 6: Varying the number of time intervals $|\mathcal{T}|$

a result, the values of the bounds are larger than a small number of assignments' scores. Hence, a small number of score updates can be avoided by exploiting bounds.

Time. In terms of execution time (Fig. 5i–5l), we can observe that time is determined by the number of computations performed. HOR-I outperforms the other methods in all cases with HOR-I being around 4 times faster than ALG (for large k in real datasets).

4.2.2 Effect of the Number of Time Intervals

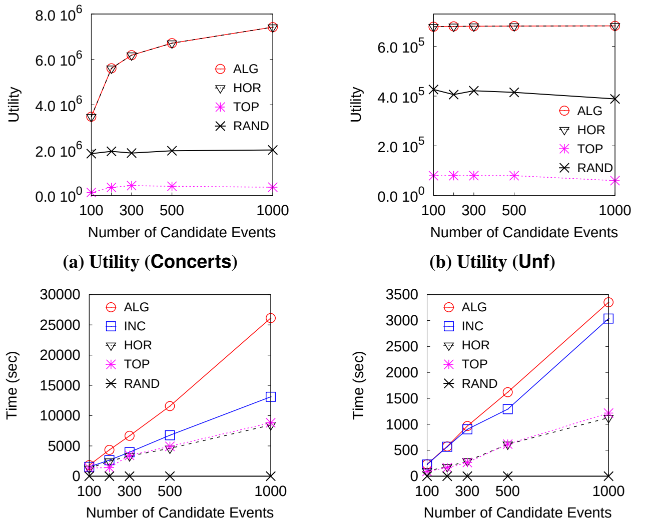
In this experiment (Fig. 6), we vary the number of time intervals $|\mathcal{T}|$. Due to lack of space, for this and the following experiments, the plots presenting the number of computations are omitted.

Utility. Regarding utility (Fig. 6a –6d), similarly to the previous experiment, our HOR algorithm performs the same as the ALG. We observe that, as the number of intervals increases, the utility of all methods increases too. This happens since the increase of available intervals results to a smaller number of events assigned in the same interval, as well as to a larger number of candidate assignments. The former results to the assignment scores (in general) being larger in cases where fewer parallel events take place. The latter offers more options, which possibly result to better assignments.

Time. As for execution time (Fig. 6e–6h), excluding TOP, the HOR-I is the most efficient in the cases which differs from HOR (i.e., $|\mathcal{T}| < 100$); while in the rest cases, HOR is the most efficient. Notice that, in general, HOR performs very close to TOP. Overall, HOR and HOR-I are about 2 to 4 times faster than the ALG, and around 5 times faster for a small number of intervals. Finally, as explained in the previous experiment, we can observe that, also in this experiment, the bound-based methods (i.e., INC, HOR-I) are less effective in Unf.

4.2.3 Effect of the Number of Candidate Events

We next study the effect of varying the number of candidate events $|\mathcal{E}|$. Note that, in this experiment, since $k < |\mathcal{T}|$, HOR-I is identical to HOR. Due to lack of space, in this experiment, the plots for the Meetup and Zip are not presented, since they are similar to Concerts.



(a) Utility (Concerts)

(b) Utility (Unf)

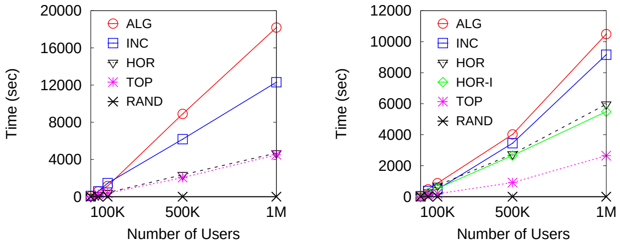
(c) Time (Concerts)

(d) Time (Unf)

Figure 7: Varying the number of candidate events $|\mathcal{E}|$

Utility. Also in this experiment (Fig. 7a–7b) our HOR method has the same utility score as the ALG in all cases. We observe that the utility of ALG and HOR increases with $|\mathcal{E}|$ (with an exception in Unf). On the other hand, for RAND it is either stable or is decreasing. This happens since the increase of $|\mathcal{E}|$ results to more candidate assignments. So, there are more options for the ALG and HOR methods, while for RAND it is less possible to select "good" assignments. Notice that, in the Unf case (Fig. 7b), the utility for the non-random methods remains stable. The reason is that increasing the number of "similar" events (as previously explained) cannot result to better assignments.

Time. Also in this experiment (Fig. 7c–7d), our HOR method outperforms the other, with INC having noticeably bad performance in Unf, compared to HOR (as in the previous experiments). Further, the difference between ALG and our methods increases with $|\mathcal{E}|$, due to the increasing number of update computations. Overall, in general HOR is around 3 to 4 times faster than ALG, and up to 5 times faster in Zip dataset specifically.



(a) Time ($|\mathcal{T}| = 150$) **(b) Time ($|\mathcal{T}| = 65$)**

Figure 8: Varying the number of users $|\mathcal{U}|$ (Unf Dataset)

4.2.4 Effect of the Number of Users

We then study the effect of varying the number of users (Fig. 8). Results for the Zip dataset are omitted since they are similar to the ones reported for Unf. Here, the HOR-I algorithm cannot be defined with the default parameters setting ($k = 100, \mathcal{T} = 150$). Hence, in order to also study HOR-I, we examine a supplementary experiment (Fig. 8b), where $\mathcal{T} = 65$. Note that, this setting ($k = 100, \mathcal{T} = 65$) corresponds to the average case for the HOR and HOR-I algorithms (w.r.t. the relation between k and $|\mathcal{T}|$; see Sect. 3.3.1 & 3.4.1).

In terms of utility (the plot is omitted due to lack of space), as expected, the utility increases with the number of users. The HOR and ALG methods have the same utility scores in all cases. Regarding performance, in the first experiment (Fig. 8a), HOR performs increasingly better than INC and ALG, as the number of users increases. In the second experiment (Fig. 8b), for larger numbers of users (i.e., $|\mathcal{U}| > 100$K), INC performs close to ALG. On the other hand, HOR and HOR-I outperforms INC, with the difference increases with $|\mathcal{U}|$. Overall, in the first experiment, HOR is around 3 to 4 times faster than ALG; in the second one, HOR and HOR-I are around 2 times faster than ALG.

4.2.5 Effect of the Number of Available Locations

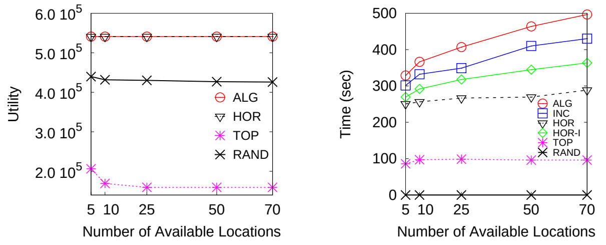
In this experiment we vary the number of available locations of each candidate event (Fig. 9). The results correspond to the Unf dataset; though, similar results are reported in all datasets. We can observe, that the utility score (Fig. 9a) remains almost unaffected for the ALG and HOR methods, while TOP and RAND perform slightly better in 5 locations. This is expected, since, as the number of locations decreases, the number of feasible assignments decreases too. Regarding the execution time (Fig. 9b), in all methods, increases with number of locations. This is due to the fact that the number of feasible assignments (as well as the computations) increases too.

4.2.6 HOR & HOR-I Worst Case w.r.t. k and $|\mathcal{T}|$

Here, we consider the setting that corresponds to the worst case w.r.t. k and $|\mathcal{T}|$ for the HOR and HOR-I algorithms (Sect. 3.3.1 & 3.4.1). Thus, for $k = 100$, the worst case corresponds to $|\mathcal{T}| = 99$. Fig. 10a presents the execution time for all datasets. We can observe that even in the worst case, HOR-I outperforms all methods in all cases (excluding the TOP). Also, in synthetic datasets, where the INC demonstrates poor performance, HOR is more efficient.

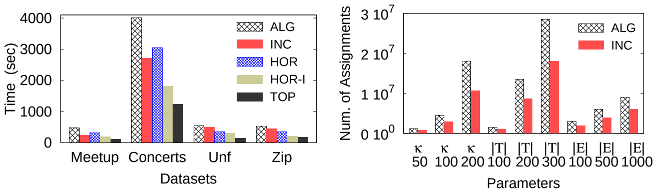
4.2.7 Search Space

In this experiment (Fig. 10b) we study the effectiveness of the proposed assignment organization (Sect. 3.2.2). We measure the number of assignments examined by the ALG and our INC algorithm, varying the main parameters (k, $|\mathcal{T}|$, $|\mathcal{E}|$). In all cases, INC accesses noticeable less assignments. Also, in each parameter, the differences between INC and ALG increases in large parameter values. Overall, in most cases, INC examines slightly more than half assignments that ALG accesses.



(a) Utility vs. Locations **(b) Time vs. Locations**

Figure 9: Varying the number of locations (Unf, $|\mathcal{T}| = 65$)



(a) HOR & HOR-I worst case **(b) ALG & INC search space**

Figure 10: HOR/-I worst case and ALG/INC search space

4.2.8 Summary

In what follows, we summarize our findings: (1) *Between the datasets used*, with the exception of Unf, all the methods report similar results. In the Unf dataset, the bound-based methods (INC & HOR-I) demonstrate lower performance than in other datasets.

(2) *Regarding utility score*, in all cases, our HOR (and HOR-I) algorithm achieves almost the same utility score as ALG. Particularly, in more than 70% of the performed experiments (including the experiments omitted from the manuscript), HOR and ALG report the same utility scores, while in the rest of the cases, the difference in utility is on average 0.008%; with the largest difference being 1.3%. Recall that, the INC algorithm always return the same solution as ALG.

(3) *Comparing* ALG [4] *with our methods*, in several cases: (*i*) our HOR and HOR-I methods are around 5× faster and perform less than half of the computations. (*ii*) INC is more than 3× faster, performs less than half of the computations.

(4) *Comparing our methods*: (*i*) HOR-I is always faster than the other methods. In several cases, HOR-I is around 3 and 2 times faster than INC and HOR, respectively. (*ii*) HOR outperforms INC in terms of time and computations, with some exceptions, in cases where $k > |\mathcal{T}|$. Overall, in several cases, HOR is around 3× faster than INC.

5 RELATED WORK

Event Management & Mining. The SES problem studied in this work was recently introduced in [4], where a simple greedy algorithm was proposed. Compared to [4], here we show that SES is hard to be approximated over a factor and we design three efficient and scalable algorithms which perform on average half the computations compared the method presented in [4] and, in most cases, are 3 to 5 times faster (more details in Sect. 1).

Recently, a number of studies have been proposed in the context of event-participant planing. These works examine the problem of finding assignments between a set of users and a set of pre-scheduled events. The determined assignments aim to maximize the satisfaction of the users while satisfying several constraints. Particularly, [12] assigns one event to each user, based on her interests and social relations. [27] finds an user-event arrangement by assigning users to events. The latter work is extended in [28], where the online setting of the problem is examined. A similar user-event arrangement problem is defined in a more advanced setting [26], where more factors are considered (e.g., complex spatio-temporal factors, travel cost). This work is extended in [6], in which participation lower bounds on event and potential changes induced either by event organizer or by users (e.g., changes on event location) are considered. In the same context, [31] tries to maximize the satisfaction of the least satisfied user. In an online scenario, [29] exploits the user feedback (i.e., accept or reject the assigned events) in order to adaptively learn user interests. This work tries to maximize the number of accepted assigned events. Compared to our work, as discussed in Sect. 1, the objective, the solution and the setting of our problem substantially differ from the aforementioned approaches.

In a different context, [8, 9] attempt to find influential event organizers and promoters from online social networks. [25] studies the influence of early respondents in online event scheduling process. Further, a number of works [5, 7, 36, 41] analyze several factors form (Event-based) Social Networks data in order to study user attendance and provide event recommendations. Our work studies a different problem compared to the aforementioned approaches. However, some of the aforementioned methods can be exploited in our problem to estimate the user attendance probability.

Assignment & Matching Problems. The problem studied shares common characteristics with the Generalized assignment (GAP) and Multiple knapsack (MKP) problems. Particularly, our problem is a generalized case of the GAP and MKP problems with identical bin capacities [18]. A major difference of SES compared to GAP and MKP is that in SES the expected attendance (resp. profit) of assigning an event (resp. item) to an interval (resp. bin) is determined based on the other events assigned to this interval. Also, beyond the event and interval entities which are also considered in the aforementioned problems, in our problem further core entities are involved (e.g., users, organizer, competing events). Additionally, assignment/matching problems (similar to bipartite matching) have been studied in spatial context [16, 30, 32, 35]. In general, the main differences of these works compared to SES, are the same as the ones that hold in GAP and MKP problems (see above).

Recommender Systems. Numerous approaches have been proposed in the context of location and event recommendations. Particularly, several works recommend events to users [13, 19, 21, 37, 40], while others offer location-based recommendations [2, 11, 14, 15, 34, 38, 39]. Further, in a more general setting, approaches have been proposed for recommending locations or items to groups of people (i.e., group recommendations) [1, 3, 20, 22–24, 33]. Compared to our work, the aforementioned approaches study a different problem, that is, recommending objects (e.g., venues, events) to users.

6 CONCLUSIONS

This paper studied the *Social Event Scheduling* (SES) problem, which assigns a set of events to time intervals, so that the number of attendees is maximized. We showed that SES is NP-hard to be approximated over a factor, and we proposed three efficient and scalable algorithms. The proposed algorithms are evaluated over several real and synthetic datasets, outperforming the existing solution three to five times in several of cases.

Acknowledgment. This research has been financed by the European Union through the FP7 ERC IDEAS 308019 NGHCS project, the Horizon2020 688380 VaVeL project, and a Google Faculty award.

REFERENCES

[1] S. Amer-Yahia, S. B. Roy, A. Chawla, G. Das, and C. Yu. Group Recommendation: Semantics and Efficiency. *PVLDB*, 2(1), 2009.

[2] J. Bao, Y. Zheng, D. Wilkie, and M. F. Mokbel. Recommendations in Location-based Social Networks: A Survey. *GeoInformatica*, 19(3), 2015.

[3] N. Bikakis, K. Benouaret, and D. Sacharidis. Finding Desirable Objects under Group Categorical Preferences. *KAIS*, 49(1), 2016.

[4] N. Bikakis, V. Kalogeraki, and D. Gunopulos. Social Event Scheduling. In *ICDE*, 2018.

[5] I. Boutsis, S. Karanikolaou, and V. Kalogeraki. Personalized Event Recommendations Using Social Networks. In *MDM*, 2015.

[6] Y. Cheng, Y. Yuan, L. Chen, C. G. Giraud-Carrier, and G. Wang. Complex Event-participant Planning and Its Incremental Variant. In *ICDE*, 2017.

[7] R. Du, Z. Yu, T. Mei, Z. Wang, Z. Wang, B. Guo Predicting activity attendance in event-based social networks: Content, context and social influence. *UbiComp* 2014

[8] K. Feng, G. Cong, S. S. Bhowmick, and S. Ma. In Search of Influential Event Organizers in Online Social Networks. In *SIGMOD*, 2014.

[9] K. Han, Y. He, X. Xiao, S. Tang, F. Gui, C. Xu, and J. Luo. Budget-Constrained Organization of Influential Social Events. In *ICDE*, 2018.

[10] V. Kann. Maximum Bounded 3-Dimensional Matching Is Max Snp-complete. *Inf. Process. Lett.*, 37(1), 1991.

[11] T. K. Lee, S. Kim, M. Balduini, D. Dell'Aglio, I. Celino, Y. Huang, V. Tresp, and E. D. Valle. Location-based Mobile Recommendations by Hybrid Reasoning on Social Media Streams. In *JIST*, 2013.

[12] K. Li, W. Lu, S. Bhagat, L. V. Lakshmanan, and C. Yu. On Social Event Organization. In *KDD*, 2014.

[13] X. Liu, Q. He, Y. Tian, W. Lee, J. McPherson, and J. Han. Event-based Social Networks: Linking the Online and Offline Social Worlds. In *KDD*, 2012.

[14] X. Liu, Y. Liu, K. Aberer, and C. Miao. Personalized Point-of-interest Recommendation by Mining Users' Preference Transition. In *CIKM*, 2013.

[15] Y. Liu, T. Pham, G. Cong, Q. Yuan. An Experimental Evaluation of Point-of-interest Recommendation in Location-based Social Networks. *PVLDB*, 2017.

[16] C. Long, R. C. Wong, P. S. Yu, and M. Jiang. On Optimal Worst-case Matching. In *SIGMOD*, 2013.

[17] R. D. Luce. *Individual Choice Behavior: A theoretical analysis*. Wiley, 1959.

[18] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1990.

[19] E. Minkov, B. Charrow, J. Ledlie, S. J. Teller, and T. S. Jaakkola. Collaborative Future Event Recommendation. In *CIKM*, 2010.

[20] E. Ntoutsi, K. Stefanidis, K. Nørvåg, and H. Kriegel. Fast Group Recommendations by Applying User Clustering. In *ER*, 2012.

[21] T. N. Pham, X. Li, G. Cong, and Z. Zhang. A General Graph-based Model for Recommendation in Event-based Social Networks. In *ICDE*, 2015.

[22] S. Qi, N. Mamoulis, E. Pitoura, and P. Tsaparas. Recommending Packages to Groups. In *ICDM*, 2016.

[23] Y. Qian, Z. Lu, N. Mamoulis, and D. W. Cheung. P-LAG: Location-aware Group Recommendation for Passive Users. In *SSTD*, 2017.

[24] E. Quintarelli, E. Rabosio, and L. Tanca. Recommending New Items to Ephemeral Groups Using Contextual User Influence. In *ACM Recsys*, 2016.

[25] D. Romero, K. Reinecke, and L. Robert. The Influence of Early Respondents: Information Cascade Effects in Online Event Scheduling. In *WSDM*, 2017.

[26] J. She, Y. Tong, and L. Chen. Utility-aware Social Event-participant Planning. In *SIGMOD*, 2015.

[27] J. She, Y. Tong, L. Chen, and C. C. Cao. Conflict-aware Event-participant Arrangement. In *ICDE*, 2015.

[28] J. She, Y. Tong, L. Chen, and C. C. Cao. Conflict-aware Event-participant Arrangement and Its Variant for Online Setting. *TKDE*, 28(9), 2016.

[29] J. She, Y. Tong, L. Chen, and T. Song. Feedback-aware Social Event-participant Arrangement. In *SIGMOD*, 2017.

[30] Y. Tong, J. She, B. Ding, L. Chen, T. Wo, and K. Xu. Online Minimum Matching in Real-time Spatial Data: Experiments and Analysis. *PVLDB*, 2016.

[31] Y. Tong, J. She, and R. Meng. Bottleneck-aware Arrangement Over Event-based Social Networks: The Max-min Approach. *WWWJ*, 19(6), 2015.

[32] L. H. U, M. L. Yiu, K. Mouratidis, and N. Mamoulis. Capacity Constrained Assignment in Spatial Databases. In *SIGMOD*, 2008.

[33] H. Wang, G. Li, and J. Feng. Group-based Personalized Location Recommendation on Social Networks. In *APWeb*, 2014.

[34] W. Wang, H. Yin, S. W. Sadiq, L. Chen, M. Xie, and X. Zhou. SPORE: a Sequential Personalized Spatial Item Recommender System. In *ICDE*, 2016.

[35] R. C. Wong, Y. Tao, A. W. Fu, and X. Xiao. On Efficient Spatial Matching. In *VLDB*, 2007.

[36] T. Xu, H. Zhong, H. Zhu, H. Xiong, E. Chen, G. Liu. Exploring the Impact of Dynamic Mutual Influence on Social Event Participation. In *SDM*, 2015.

[37] H. Yin, Q. V. H. Nguyen, Z. Huang, and X. Zhou. Joint Event-partner Recommendation in Event-based Social Networks. In *ICDE*, 2018.

[38] H. Yin, Y. Sun, B. Cui, Z. Hu, and L. Chen. LCARS: a Location-content-aware Recommender System. In *KDD*, 2013.

[39] H. Yin, X. Zhou, B. Cui, H. Wang, K. Zheng, and N. Q. V. Hung. Adapting to User Interest Drift for Poi Recommendation. *TKDE*, 28(10), 2016.

[40] W. Zhang, J. Wang, and W. Feng. Combining Latent Factor Model with Location Features for Event-based Group Recommendation. In *KDD*, 2013.

[41] X. Zhang, J. Zhao, and G. Cao. Who Will Attend? - Predicting Event Attendance in Event-based Social Network. In *MDM*, 2015.

GroupTravel: Customizing Travel Packages for Groups

Sihem Amer-Yahia[1], Shady Elbassuoni[2], Behrooz Omidvar-Tehrani[1],
Ria Mae Borromeo[3], Mehrdad Farokhnejad[1]
[1]Univ. Grenoble Alpes, CNRS, LIG (France), [2]The American University in Beirut (Lebanon),
[3]The University of the Philippines (Philippines)
[1]firstname.lastname@univ-grenoble-alpes.fr, [2]se58@aub.edu.lb, [3]riamae@gmail.com

ABSTRACT

We present GroupTravel, a framework that generates customized travel packages (TPs) for a group of individuals. GroupTravel implements different consensus functions proposed in group recommendation to reach agreement among members. Given a group whose members provide a travel query, GroupTravel returns k Composite Items (CIs) of Points Of Interest (POIs) that are *valid*, *representative*, *cohesive* and *personalized*. Validity is achieved by satisfying the query expressed by the group. Representativity ensures good coverage of a city. Cohesiveness reflects geographic proximity of POIs forming a CI. Personalization is achieved by choosing POIs that best match the travel preferences of group members. Additionally, group members can interact with generated TPs to customize them. With extensive synthetic experiments and user studies, we examine the benefit of personalization and the impact of different group consensus on user satisfaction. We also show that providing the ability to interact with TPs and reflecting that in the consensus yields better TPs.

1 INTRODUCTION

The ability to generate a travel package (TP) that best fits a traveler's profile is a longstanding problem that has been studied for years (e.g., [1–3]). In this work, we develop GroupTravel, a framework that generates TPs for a group of individuals traveling together. A TP is a set of k Composite Items (CIs), each of which is formed by Points of Interest (POIs) in a city. GroupTravel personalizes TPs based on a group profile that is computed as an aggregation of its members' preferences. GroupTravel allows travelers in a group to further customize the proposed TP via interaction. GroupTravel extends recent work that focused on a single traveler at a time [4] to reach consensus between multiple travelers [5, 6].

Travel Packages as Composite Items. A TP is a set of k CIs. CIs are useful in planning a city tour, selecting books for a reading club, or organizing a movie rating contest [1, 7–15]. Each CI satisfies a query that specifies desired POI categories and a budget constraint [13]. An objective function is defined to build a TP containing k *valid*, *representative*, *cohesive*, and *personalized* CIs. Validity ensures that each CI satisfies the query. Representativity enforces that the k CIs "cover" the city. Cohesiveness forms CIs containing geographically close POIs. Finally, personalization ensures that the CIs contain POIs that match the members' travel preferences. We use a fuzzy clustering algorithm to find the best k CIs forming a TP.

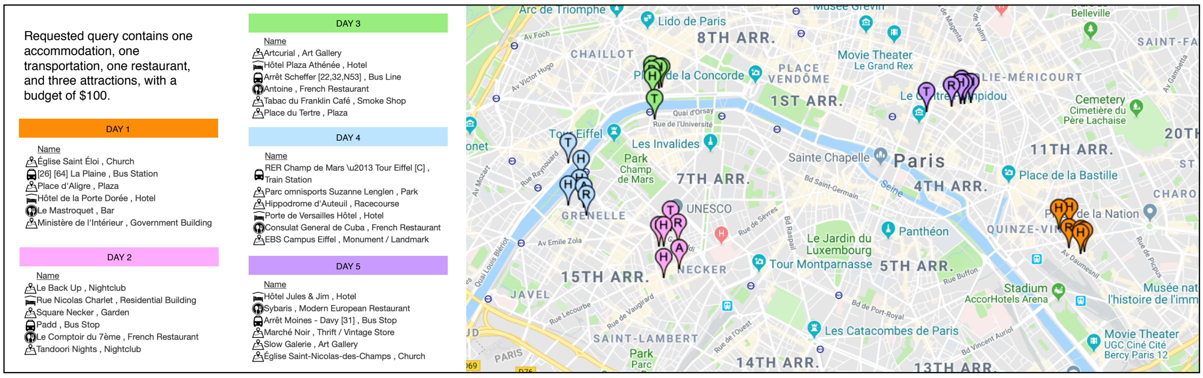
For example, a group wishing to visit Paris may request a TP consisting of five CIs, one per day. The group specifies a query which dictates CI validity: each CI must contain an accommodation, a restaurant, three attractions and one transportation mode, and such that the overall cost of visiting POIs in a CI is no more than $100. Figure 1 shows the TP returned by GroupTravel. Each CI contains a set of co-located POIs that can be visited in one day. In addition, the TP formed by the set of 5 CIs, provides a good coverage of Paris and the POIs in each CI match the group members' travel preferences.

In this paper, we make three contributions. We formalize the problem of building a TP for a group of travelers. We define how group members can interact with the generated TP to further customize it. We run synthetic experiments and user studies to validate GroupTravel's effectiveness in generating a satisfying TP for groups.

First contribution: TPs for groups. GroupTravel takes as input a query and individual travel profiles. It outputs a *personalized* TP for the group. The preference of a group for an item must *reflect the degree to which the item is preferred by all group members*. The group preference must also *capture the level at which members disagree or agree with each other*. All other conditions being equal, an item that draws high agreement should have a higher score than an item with a lower overall group agreement. The different ways of aggregating *group preference* and *group disagreement* result in different *group consensus methods* ranging from *average preference*, to *least misery* and *disagreement-based* methods [16–18]. We leverage those definitions to generate a group travel profile from individual preferences.

Second contribution: interactive TPs. In [4], we examined the benefits of letting a user interact with a TP to customize it. The rationale is that even though the k CIs are valid, representative, cohesive, and personalized, a user may still want to intervene after seeing the travel options available in a city. We showed that providing interaction primitives to the user enabled expressing additional contextual preferences such as exploring some neighborhoods in a city or requesting a new CI which contains a specific POI. In this paper, we examine the benefit of interactivity in GroupTravel, i.e., for a group of individuals traveling together. To achieve that, we define the impact of each operation on a TP and on the profile of a group.

Figure 2 illustrates the flow of GroupTravel. Given a group of travelers and a consensus function, a group profile is generated from individual profiles. Our fuzzy clustering algorithm admits a geographic region (e.g., a city), a query and the group profile. It generates a TP that is shown to the travelers who can modify CIs, delete CIs, or generate new CIs. This interaction is reflected in the group's profile by updating the overall group preferences according to the



Requested query contains one accommodation, one transportation, one restaurant, and three attractions, with a budget of $100.

DAY 1

Name
Église Saint Éloi , Church
[26] [64] La Plaine , Bus Station
Place d'Aligre , Plaza
Hôtel de la Porte Dorée , Hotel
Le Mastroquet , Bar
Ministère de l'Intérieur , Government Building

DAY 2

Name
Le Back Up , Nightclub
Rue Nicolas Charlet , Residential Building
Square Necker , Garden
Padd , Bus Stop
Le Comptoir du 7ème , French Restaurant
Tandoori Nights , Nightclub

DAY 3

Name
Artcurial , Art Gallery
Hôtel Plaza Athénée , Hotel
Arrêt Scheffer [22,32,N53] , Bus Line
Antoine , French Restaurant
Tabac du Franklin Café , Smoke Shop
Place du Tertre , Plaza

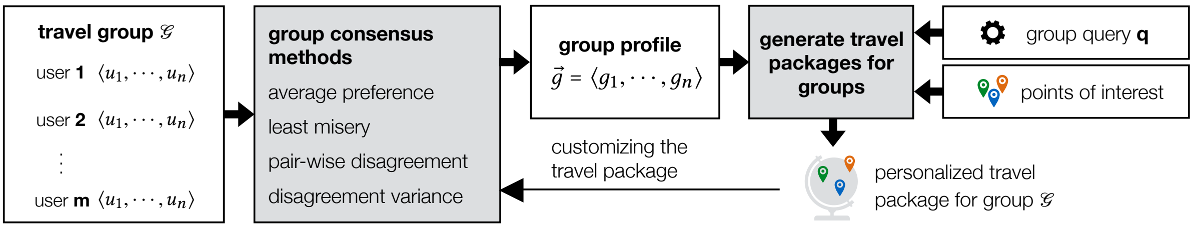
DAY 4

Name
RER Champ de Mars \u2013 Tour Eiffel [C] , Train Station
Parc omnisports Suzanne Lenglen , Park
Hippodrome d'Auteuil , Racecourse
Porte de Versailles Hôtel , Hotel
Consulat General de Cuba , French Restaurant
EBS Campus Eiffel , Monument / Landmark

DAY 5

Name
Hôtel Jules & Jim , Hotel
Sybaris , Modern European Restaurant
Arrêt Moines - Davy [31] , Bus Stop
Marché Noir , Thrift / Vintage Store
Slow Galerie , Art Gallery
Église Saint-Nicolas-des-Champs , Church

Figure 1: A 5-day travel package (TP) in Paris consisting of 5 Composite Items (CIs) of POIs for the group query $\langle 1\ \mathit{accommodation}, 1\ \mathit{transportation}, 1\ \mathit{restaurant}, 3\ \mathit{attractions}, \$100\rangle$. **Letters A, T, R, and H on POIs represent categories of accommodation, transportation, restaurant, and attraction, respectively.**



travel group $\mathcal{G}$
user 1 $\langle u_1, \cdots, u_n\rangle$
user 2 $\langle u_1, \cdots, u_n\rangle$
⋮
user m $\langle u_1, \cdots, u_n\rangle$
group consensus methods
average preference
least misery
pair-wise disagreement
disagreement variance
group profile
$\vec{g} = \langle g_1, \cdots, g_n\rangle$
generate travel packages for groups
group query q
points of interest
customizing the travel package
personalized travel package for group $\mathcal{G}$

Figure 2: GroupTravel framework

requested changes. The new group profile can then be used to generate other TPs in the same or in a different city and test the "robustness" of the updated profile across cities.

Third contribution: experiments. The purpose of our experiments is two-fold: (1) study the utility of consensus functions from group recommendation in the context of GroupTravel, and (2) examine the benefit of interactive customization for groups. We run two extensive sets of experiments. In the first, we generate synthetic data to examine the relationship between group characteristics (group size, agreement between members, and consensus methods) and optimization dimensions (representativity, cohesiveness, and personalization). In the second, we run an extensive user study with real users from Figure-Eight[1] and Amazon Mechanical Turk[2]. Our study evaluates the usefulness of GroupTravel by asking actual users about their satisfaction with TPs before and after customization.

Our findings extend previous work for single travelers [4] and in group recommendation [5, 6]. In particular, we find that customization makes travel profiles more robust. Additionally, we find that disagreement-based consensus performs best in terms of all optimization dimensions, and for all different group variants (uniform and non-uniform as well as small, medium and large). Least misery, on the other hand, is more successful at satisfying the median user in larger groups with diverse tastes.

We also observe that TPs for non-uniform groups are more cohesive than TPs for uniform groups. This result generalizes previous work where a tension between personalization and cohesiveness was observed for individual users: the more personalized a TP is, the less likely it is to be cohesive, and vice versa [4]. Non-uniform groups contain members with diverse preferences. This diversity dilutes personalization (the aggregated profile expresses lower preferences than individual profiles). Given that, cohesiveness is likely to be higher for non-uniform groups. Similarly, the cohesiveness of uniform groups increases with group size, while their personalization decreases.

Our user study validates our objective function by showing that personalized TPs perform well and are liked better than non-personalized and random TPs. We also find that TPs obtained using average preference and least misery are best for uniform groups, whereas TPs obtained using disagreement-based methods are best for non-uniform groups. Similarly, incorporating inter-member disagreements is shown to be the best way to reach a consensus within diverse groups.

The paper is organized as follows. In Section 2, we describe our data model. Our approach for building group travel packages and interacting with them is described in Section 3. Experiments are reported in Section 4. The related work is reviewed in Section 5. We conclude with a summary of our work and a discussion of future work in Section 6.

[1] *http://www.figure-eight.com/*
[2] *https://www.mturk.com/*

i.id	*i.name*	*i.cat*	*i.coordinates*	*i.type*	*i.tags*	*i.cost*
1	Le Burgundy	*acco*	⟨48.8679, 2.3256⟩	hotel	*luxury suites cognac champagne bar gastronomic restaurant spa*	3.00
2	The Bicycle Store	*trans*	⟨48.8642, 2.3658⟩	bike shop	*accessoires vélo beach cruiser bicycle paris fixed gear*	2.71
3	Un Zèbre à Montmartre	*rest*	⟨48.886, 2.3348⟩	french	*bankers bar brunch café comedy fireplace frat hipsters liquor margaritas*	3.20
4	Les Arts Décoratifs	*attr*	⟨48.8632, 2.3334⟩	museum	*arts contemporary decorative exhibition fashion gallery mode modern museum*	3.86

Table 1: Sample Points Of Interest in Paris

2 DATA MODEL

2.1 Items

Our travel packages are built using Points Of Interest (POIs) in a city. Table 1 shows sample POIs in Paris. In our experiments, we use the TourPedia dataset.[3] It consists of POIs in eight cities which are divided into four main categories (*cat* for short): (1) accommodation (*acco*), (2) transportation ($trans$), (3) restaurant ($rest$) and (4) attraction ($attr$). Each POI or item i has a unique id, a name, a longitude and a latitude. To be able to set the rest of the attributes for the items in our dataset, we augment it with additional information extracted from Foursquare.[4] Using the Foursquare API, we retrieved the type of each item i. For instance, in the case of an accommodation item i, $i.type$ will be set to either a hotel, a hostel, a motel, a college residence hall, etc. Similarly, for a transportation item i, we set $i.type$ to its transportation mode which can be a tram station, a train station, a car rental, a bike rental, and so on. To set $i.tags$ for an item i, we retrieved all the tags provided by users on Foursquare for item i. Finally, there are may ways of setting the cost of visiting an item i or $i.cost$ including declarative data.

2.2 User Profile

Our goal is to provide a group of users with personalized travel packages. To do that, we build a group travel profile which captures the preferences of group members for different types of POIs. We start by defining a single-user profile and then explain how we aggregate the profiles of different users to generate a group profile.

Each user u is associated with a profile for each POI category c (i.e., *acco*, *trans*, *rest* or *attr*), which is a vector defined as follows:

$$\vec{u} = \langle u_1, \cdots, u_n \rangle$$

where n is the number of different POI types in category c and each $u_j, 1 \leq j \leq n$ is a score between 0 and 1.

To simplify our notation, c does not appear in $\vec{u}$. One way to set the vector $\vec{u}$ is to ask the user to state her preferences for the different types of POIs. In case the POI types are not sufficient to capture all the dimensions of travel preferences for users, we can try to learn these other dimensions from the data. For accommodation and transportation, the types are well-defined (e.g., Bicycle, Bus, Tram for transportation, and Hotel, Hostel, Resort for accommodation). For restaurants and attractions, we leverage their tags to capture information such as cuisine and ambiance for restaurants, or type and entrance fee for attractions. Particularly, we rely on Latent Dirichlet Allocation (LDA) applied to tags to identify latent topics for restaurants and attractions [19]. This results in several types such as "art gallery, museum, library" and "garden, park, event hall" for attractions, and "Japanese, sushi" and "beer, wine, bistro" for restaurants.

To set the individual components of the vector $\vec{u}$, we do the following. For the case of transportation and accommodation, we ask the user to provide a rating r_j between 0 and 5 for each accommodation or transportation type s_j. Similarly, for restaurants and attractions, we ask the user to provide a rating r_j between 0 and 5 for each latent topic s_j where each topic is represented by representative tags. Finally, we set the score u_j in the user profile as the normalized rating over all types or topics, i.e.,

$$u_j = \frac{r_j}{\sum_{k=1}^{n} r_k}$$

2.3 Group Profile

Similar to users, a group of users $\mathcal{G}$ is associated with a group profile for each POI category c (i.e., *acco*, *trans*, *rest* or *attr*), which is a vector defined as follows:

$$\vec{g} = \langle g_1, \cdots, g_n \rangle$$

where n is the number of different POI types in category c, and each $g_j, 1 \leq j \leq n$ is a score between 0 and 1. The value g_j reflects the preference of group $\mathcal{G}$ for a POI type by aggregating the preferences of group members.

To compute each g_j, we need to aggregate the preferences u_j of each user $u \in \mathcal{G}$. To do so, we leverage *consensus functions* that were previously proposed in the context of group recommendation [17, 18]. A consensus function is used to aggregate two components: *group preference* and *group disagreement*. Intuitively, to compute g_j, we need to *reflect the degree to which the j^{th} POI type in a given category is preferred by all group members*, and *capture the level at which members disagree or agree with each other about the j^{th} POI type in a given category.* All other conditions being equal, a POI that draws high agreement should have a higher score than a POI with a lower overall group agreement. We revisit the definitions we introduced in [16] to compute group consensus as a combination of group preference and group disagreement.

Group preference. The degree to which the j^{th} POI type in a given category is preferred by all group members is denoted p_j, and is computed using one of two common preference aggregation functions:

[3] *http://tour-pedia.org/about/*

[4] *https://foursquare.com/*

(1) *Average Preference:* $p_j = \frac{1}{|\mathcal{G}|} \sum_{u \in \mathcal{G}} u_j$

(2) *Least-Misery Preference:* $p_j = \min_{u \in \mathcal{G}} u_j$

Group disagreement. The level at which members disagree or agree with each other about the j^{th} POI type in a given category is denoted d_j, and is computed using one of two common disagreement computation functions:

(1) *Average Pair-wise Disagreement:*
$d_j = \frac{2}{|\mathcal{G}|(|\mathcal{G}|-1)} \sum_{u,v \in \mathcal{G}} (|u_j - v_j|)$,

(2) *Disagreement Variance:*
$d_j = \frac{1}{|\mathcal{G}|} \sum_{u \in \mathcal{G}} (u_j - \mu_j)^2$ where $\mu_j = \frac{1}{|\mathcal{G}|} \sum_{u \in \mathcal{G}} u_j$

The average pair-wise disagreement function computes the average of pair-wise differences in individual preferences for the j^{th} POI type among group members, while the variance disagreement function computes the mathematical variance of individual preferences. Intuitively, the closer the preferences between users u and v, the lower their disagreement.

Group consensus. We are now ready to compute a single group consensus score g_j for the j^{th} POI type in a given category. We do that by combining group preference and disagreement as follows:

$$g_j = w_1 \times p_j + w_2 \times (1 - d_j)$$

where $0 \leq w_1, w_2 \leq 1$, $w_1 + w_2 = 1$, and they specify the relative importance of preference and disagreement in the overall group consensus, respectively. We hence have four possible consensus functions that combine preference and disagreement to compute a single score g_j in the group profile.

Example. Consider a family (a couple with three kids) which forms a travel group $\mathcal{G}$ of size 4. Their preferences for visiting museums are 0.8, 1.0, 0.6, and 0.2, for the father, mother, the teenage child, and the kid, respectively, where 1.0 reflects the highest preference. Using average preference method, the group preference for this POI type is $p = 0.65$. However the group preference towards museums gets as low as 0.2 using the least misery method. Least misery favors the most unhappy user in the group, hence the preference of the kid dominates others'. On the other hand, the average pair-wise disagreement between $\mathcal{G}$'s members is $d = 0.43$. Also, the disagreement variance is $d = 0.088$. Given $w_1 = 0.5$ (hence $w_2 = 0.5$), $\mathcal{G}$'s consensus for museums is $g = 0.61$ by considering average preference and average pair-wise disagreement as the group preference and group disagreement components, respectively.

3 BUILDING GROUP TRAVEL PACKAGES

In this section, we define and solve the problem of building personalized travel packages for groups. We start by introducing Composite Items (CIs) and Travel Packages (TPs), and formulate building travel packages as a fuzzy clustering problem. We then discuss how groups can interact with travel packages to customize them using GroupTravel.

3.1 Composite Items

A Composite Item is a set of POIs of different categories. To be able to define what constitutes a CI, we rely on a group query which is a vector defined as follows:

$$\vec{q} = \langle \#c_1, \cdots, \#c_m, B \rangle$$

where m is the number of POI categories (4 in our dataset), $\#c_j, 1 \leq j \leq m$ specifies the number of items for POI category c_j, and B is a total budget.

A query indicates which categories of POIs, and how many of them, should constitute a CI. For example, the query $\vec{q} = \langle 1\ acco, 1\ trans, 2\ rest, 1\ attr, \$120 \rangle$ represents a CI with 1 accommodation, 1 transportation, 2 restaurants and 1 attraction for a daily budget of $120.

The query is used to define valid CIs as follows. Given a set of items $\mathcal{I}$ and a query $\vec{q} = \langle \#c_1, \cdots, \#c_m, B \rangle$, a valid $CI \subseteq \mathcal{I}$ is a set of items such that (1) their categories correspond to the requested categories in the group query, and (2) the total budget of items forming the CI is at most B, i.e.,

$$\begin{cases} (i)\ \forall j \in \{1, \cdots, m\}, \ \sum_{i \in CI} \mathbb{1}(i.cat, c_j) = \#c_j \\ (ii)\ \sum_{i \in CI} i.cost \leq B \end{cases}$$

where $\mathbb{1}$ is an indicator function which is equal to 1 if the category of item i is c_j and 0 otherwise. We refer to the set of valid CIs as $\mathcal{V}$.

3.2 Travel Packages

We are now ready to define the notion of a group travel package and formulate building travel packages as a fuzzy clustering problem. Given a group $\mathcal{G}$, a set of items $\mathcal{I}$, and a query $\vec{q}$, we define a group travel package as a set of k Composite Items $\mathsf{TP} = \{CI_1, CI_2, \cdots, CI_k\}$ where each $CI_j \subseteq \mathcal{I}$ is a valid Composite Item.

A travel package is formed by *valid* and *cohesive* CIs that are *representative* of the set of available items in the city. The validity of a CI is expressed in terms of a query $\vec{q}$ as defined in Section 3.1. Its cohesiveness must reflect how close the items forming a CI are to each other. The intuition is that each CI represents things to do in a given area of a city and must thus have POIs that are geographically close to each other. Finally, the representativity of a travel package serves the purpose of providing a good coverage of the city [13]. KFC, the algorithm that solves that problem in [13], relies on fuzzy clustering to position k centroids that "cover" the whole dataset. CIs are then formed in the vicinity of these centroids, which ensures that they provide a good summary of the dataset. In the context of this work, we may want to see a given item in different CIs. For example, a user's hotel could belong to multiple CIs. The same applies to a museum if the user wants to go back to the museum (as is the case for the "Louvre museum" in Paris that requires more than one visit). Contrary to *hard* clustering, *fuzzy* clustering allows each data point to participate in multiple clusters [20]. Thus, KFC is a natural choice for us to generate travel packages.

To be able to generate CIs, we define an item vector for each POI i as follows:

$$\vec{i} = \langle i_1, \cdots, i_n \rangle$$

where n is the number of types for the POI category that item i belongs to, and each $i_j, 1 \leq j \leq n$ is a score between 0 and 1. The item vector $\vec{i}$ is set based on the category of the item i. For accommodation and transportation items (i.e., $i.cat = acco$ or $i.cat = trans$), we set i_j as follows:

$$i_j = \begin{cases} 1, & \text{if } i.type = t_j \\ 0, & \text{otherwise} \end{cases}$$

where t_j is the j^{th} type in the category that item i belongs to. For restaurants and attractions, the item vector $\vec{i}$ is set to the topic distribution vector for item i obtained from applying LDA.

To generate a personalized travel package TP for a group $\mathcal{G}$, we optimize the following objective function:

$$\begin{aligned} &\underset{M,W}{argmax}\ \alpha \sum_{j=1}^{k} \sum_{i \in \mathcal{I}} w_{ij}^{f}\ (1 - Euclidean(i, \mu_j)) + \\ &\sum_{j=1}^{k} \max_{CI_j \in \mathcal{V}} \Bigg(\beta \sum_{i \in CI_j} (1 - Euclidean(i, \mu_j)) + \\ &\qquad\qquad \gamma \sum_{i \in CI_j} Cosine(\vec{i}, \vec{g}) \Bigg) \\ &s.t.\ \forall i \in \mathcal{I},\ \sum_{j=1}^{k} w_{ij} = 1 \end{aligned} \tag{1}$$

In Equation 1, we use a normalized geographic Euclidean distance between two items, and Cosine similarity between an item vector and the group profile vector for the category the item belongs to. Euclidean distance is an an approximation of Haversine calculations on a spherical space (to measure the distance in miles/kilometers between two latitudes and longitudes) with Equirectangular calculations on a Euclidean space to gain performance. This approximation makes sense for short distances within a city as we have experimentally observed that our performance gain is 30x with only 0.1% of precision loss. To obtain a normalized Euclidean distance, we divide all distance values by the largest observed distance value. $M = \{\mu_1, \mu_2, \cdots, \mu_k\}$ is a set of k centroids, W is a weight matrix of size $|\mathcal{I}| \times k$ which contains the w_{ij} weights indicating which item belongs to which cluster. α and β are user-dependent parameters controlling the weight of the optimization objectives, and $f \leq 1$ is the weighting exponent used in fuzzy clustering.

The two components of the objective function are inherited from KFC [13] and capture cohesiveness and representativity by choosing items i that are close to the centroid μ_j of each of the k clusters, where closeness is based on the geographic distance. Those components serve to identify cluster centroids μ_j that are representative of the complete dataset, while ensuring that the centroids obtained are close to some valid CI (i.e., $\in \mathcal{V}$). Maximizing the sum of the similarities of all items in a CI to its centroid additionally ensures the cohesiveness of the valid CI considered.

The last component, weighted with γ, captures personalization by comparing the similarity of the group's profile vector $\vec{g}$ to the item vector $\vec{i}$. This allows the algorithm to focus on producing CIs that are valid, cohesive and that contain personalized items that matter to the group, rather than any items.

3.3 Customizing Travel Packages

In order to customize travel packages, we provide group members with a GUI where all the CIs forming a travel package are displayed on an interactive map of the city. We define five atomic operations to allow groups to refine their preferences and produce customized TPs. Our operations are:

1. REMOVE(i,CI): remove POI i from Composite Item CI.
2. ADD(i,CI): add POI i to a Composite Item CI. The user can filter the POIs by category and type and the closest items to CI satisfying the user filter are displayed for the user to choose from.
3. REPLACE(i,CI): replace POI i in CI with another POI. In that case, the system recommends to the user the closest POI j in terms of geographic distance and such that $i.cat = j.cat$.
4. GENERATE(RECTANGL$E(x, y, w, h)$): generate a new CI that is centered in the area enclosed by a rectangle whose upper-left point is (x, y), and with width w and height h. The generated CI is both *valid* and *cohesive*.

Using the above set of operations, group members can customize the generated travel package until they are satisfied with it. For example, a member can drop or add a set of POIs in a given CI. She can also replace POIs with others that the system recommends to ensure that the CI remains as cohesive as possible. Finally, a group member can completely delete a CI by iteratively removing items in that CI until it is empty. Similarly, a group member can generate a new CI by selecting an area in the map. The group interactions with the CIs provide us with additional information about the group travel preferences. Particularly, they are useful in refining the group travel profile.

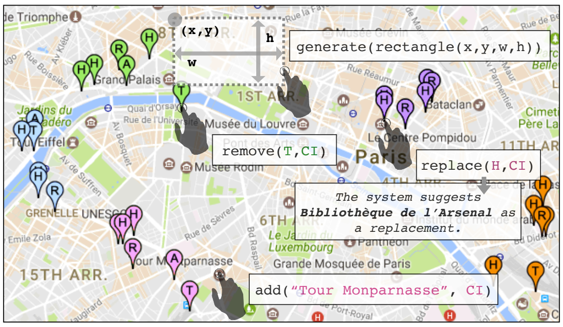


Figure 3: Customization operators

Figure 3 illustrates examples of customization operators in Paris. For instance, a REMOVE operator is requested to discard a bus stop in the area of "Invalides". It is also requested to ADD "Monparnasse tower" to the travel package as an attraction. In response to a REPLACE operator, the system suggests "Arsenal library" to replace "Pompidou library". Also a GENERATE operation is requested by defining an area from "L'église de la Madeleine" to "Palais Royal", where a potential attraction POI is "Place Vendôme".

Refining the group profile. The interactions of group members with the provided CIs serve as implicit feedback that can be used to update the group's travel profile. This refinement serves two purposes: *(1) make the group profile robust so that fewer interactions will be needed in the future including in other cities, (2) build long-lasting profiles for non-ephemeral groups.* We define two strategies for updating the group profile: individual and batch strategies. The *individual strategy* was defined for single travelers [4]. It first refines each group member profile based on that member's interactions with the TP, if that member customized the TP. It then aggregates all individual profiles into a new group profile. The *batch strategy* gathers interactions performed by all group members and directly refines the group profile. We describe the batch strategy that is a direct adaptation of the individual strategy.

Let $\vec{g}$ be the current group profile for POI category c. Furthermore, assume a group member added a set of POIs I^+ that belong to category c. Also, assume a group member removed a set of POIs I^- that belong to category c. Now the group vector $\vec{g}$ for category c can be updated as follows:

$$\vec{g} = \vec{g} + \vec{g}^+ - \vec{g}^-$$

where

$$\vec{g}^+ = \frac{1}{|I^+|} \sum_{i \in I^+} \vec{i}$$

and $\vec{i}$ is the item vector of item i as defined in Section 3.2.

The value $\vec{g}^-$ will be set the exact same way as $\vec{g}^+$ by replacing I^+ with I^- above. Finally, if any of the components of the updated vector $\vec{g}$ falls below 0, the value of this component will be set to 0.

4 EXPERIMENTS

We provide two sets of experiments. First, we generate synthetic data to examine the relationship between group characteristics (group size, uniformity, and consensus methods) and optimization dimensions (representativity, cohesiveness, and personalization). As we do not recruit real participants in this experiment, we focus on dissecting the objective function of GroupTravel. In the second experiment, we describe our user study which evaluates the usefulness of GroupTravel by asking group members about their satisfaction with the generated travel packages, before and after customization.

4.1 Setup

Group composition. We build groups by aggregating profiles of individual users. In our synthetic experiment, user profiles are generated at random. In our user study, user profiles capture the travel preferences of the participants.

We form different groups by varying their *size* (the number of users in a group) and *uniformity*. Intuitively, a group is more uniform if its members have similar preferences with respect to POI types. The uniformity of a group $\mathcal{G}$ is a value between 0 and 1 and is computed as the average pairwise Cosine similarity between profile vectors of all $\mathcal{G}$'s members, i.e.,

$$uniformity(\mathcal{G}) = \frac{2}{|\mathcal{G}||\mathcal{G}-1|} \sum_{u,v \in \mathcal{G}} Cosine(\vec{u}, \vec{v})$$

We consider three categories of group sizes, i.e., *small* groups having 5 members, *medium* groups having 10 members, and *large* groups having 100 members. We also consider two categories of group uniformity, i.e., *uniform* groups having a uniformity value larger then 0.85, and *non-uniform* groups having a uniformity value smaller than 0.20.

Group consensus. Recall from Section 2.3 that for a group $\mathcal{G}$, we aggregate the individual user profiles to generate a group profile by applying a group consensus as follows:

$$g_j = w_1 \times p_j + w_2 \times (1 - d_j)$$

where p_j represents a group preference for POI type j and d_j represents group disagreement for POI type j. We employ the following variants of group consensus in our experiments:

- *Average preference*, where p_j is *average preference* and $w_1 = 1.0$ (i.e., group disagreement is not considered).
- *Least misery*, where p_j is *least misery* and $w_1 = 1.0$ (i.e., again group disagreement is not considered).
- *Average preference with average disagreement*, where p_j is *average preference*, d_j is *average pair-wise disagreement* and $w_1 = 0.5$. Hereinafter, we call this method "pair-wise disagreement" for simplicity.
- *Average preference with disagreement variance*, where p_j is *average preference*, d_j is *disagreement variance* and $w_1 = 0.5$. Hereinafter, we call this method "disagreement variance" for simplicity.

4.2 Optimization dimensions

Once a TP is computed for a group, we measure each component of our optimization objective, representativity, cohesiveness and personalization (Section 3.2).

Representativity measures the collective coverage of POIs in a TP over a region of interest, e.g., a city. The farther CIs in a TP are from each other, the higher the TP's representativity. Representativity is measured as follows:

$$representativity(\mathsf{TP}) = \sum_{l=1}^{k} \sum_{j=l}^{k} Euclidean(\mu_l, \mu_j) \qquad (2)$$

where μ_l is the centroid of the composite item C_l, and $Euclidean(\mu_l, \mu_j)$ measures the Euclidean distance between the centroids μ_l and μ_j. Recall from Section 3.2 that the Euclidean distance is an an approximation of Haversine calculations.

Cohesiveness measures the geographical compactness of CIs in a TP, i.e., how close the POIs in a CI are to each other. It is measured as follows:

$$cohesiveness(\mathsf{TP}) = \mathcal{S} - \left(\sum_{CI \in \mathsf{TP}} \sum_{i,j \in CI} Euclidean(i, j) \right) \qquad (3)$$

where $Euclidean(i, j)$ measures the Euclidean distance between the geographical coordinates of POIs i and j. The constant $\mathcal{S}$ defines the maximum possible sum of distances over CIs in a TP. In our synthetic experiment, we set $\mathcal{S} = 221.79$ as the largest observed value for aggregated distances.

While representativity and cohesiveness evaluate TPs in a geographical domain, personalization evaluates them in terms of preferences (using the profile vector $\vec{g}$ for the group $\mathcal{G}$). Personalization is measured as follows:

$$personalization(\mathsf{TP}, \mathcal{G}) = \sum_{CI \in \mathsf{TP}} \sum_{i \in CI} Cosine(\vec{i}, \vec{g}) \qquad (4)$$

		average preference			**least misery**			**pair-wise disagreement**			**disagreement variance**		
		R	C	P	R	C	P	R	C	P	R	C	P
uniform groups	**small**	100%	69%	95%	38%	0%	74%	100%	74%	99%	99%	79%	100%
	medium	94%	70%	94%	75%	57%	73%	95%	77%	98%	96%	80%	98%
	large	85%	73%	72%	76%	76%	68%	96%	87%	97%	97%	84%	96%
non-uniform groups	**small**	17%	89%	75%	21%	76%	07%	98%	94%	98%	97%	90%	98%
	medium	25%	90%	83%	14%	76%	7%	98%	98%	99%	98%	94%	98%
	large	32%	96%	98%	13%	79%	00%	95%	100%	96%	95%	96%	93%

Table 2: Synthetic experiment for travel groups. Optimization dimensions are abbreviated as R for representativity, C for cohesiveness, and P for personalization.

4.3 Synthetic Data Experiment

Our goal in the synthetic data experiment is to examine the relationship between group characteristics and our optimization dimensions. Preferences of real people will be verified later in the user study (Section 4.4). Groups are characterized by their *uniformity*, i.e., similarity between members, *size*, and the *consensus function* used to aggregate individual preferences.

4.3.1 Setup. We describe how we generate user and group profiles and other settings in the synthetic experiments.

Group profiles. We generate user profiles in an independent roll-and-dice process. Each profile is a vector whose cells contain preference values for different types of POIs. We assign a random value between 0 and 1 to each cell in the user profile vector. A group $\mathcal{G}$ is a matrix of $|\mathcal{G}|$ user profiles, where $|\mathcal{G}| \in \{5, 10, 100\}$. For each combination of group size and group uniformity (uniform and non-uniform), we generated 100 different random groups. For each generated group, we computed a group profile using the four different consensus methods. As a result, we obtained 2400 distinct group profiles in total.

Query and objective function. We generate a TP for each group profile. Each TP contains exactly 5 CIs that are valid with respect to a default query, i.e., $\langle 1$ *acco*, 1 *trans*, 1 *rest*, 3 *attr*$\rangle$. Also, we specify *an infinite budget* to ensure that all popular POIs are included in the CIs. Regarding the weights in our objective function (Equation 1), we always set $\gamma = 1.0$ for personalization, and we assign random values to α and β in the range $[0, 1]$ for representativity and cohesiveness, respectively, in order to prevent bias towards an optimization objective.

Optimization dimensions. For the TPs generated for group profiles, we report representativity, cohesiveness, and personalization as defined in Equations 2, 3, and 4, respectively. The values obtained for all dimensions are normalized in the range $[0, 1]$ in min-max style:

$$normalized\_value(o) = \frac{value(o) - min(o)}{max(o) - min(o)}$$

where $min(o)$ and $max(o)$ are the smallest and highest values of an optimization dimension o, respectively. Before normalization, the values of representativity, cohesiveness, and personalization were spread in the ranges $[0.03, 41.39]$, $[19.29, 221.79]$, and $[0.01, 0.16]$, respectively.

Validation of observations. We validate all our observations on optimization dimensions in terms of statistical significance using the One-way ANOVA procedure, with the $\mathcal{F}$-measure of MSB/MSE[5] and the significance level of $p = 0.05$. ANOVA results are reported as $\mathcal{F}(n, k) = x$ given $p < 0.05$, where n and k are the first and second degrees of freedom, respectively, and x is the value obtained for the $\mathcal{F}$-measure. Fully-independent generation of user and group profiles ensures that the $\mathcal{F}$-measure captures truly significant results.

We also compute the Pearson correlation coefficient (PCC) to validate linear correlations between attributes. PCC has a value between $+1$ and -1, where $+1$ reflects a totally positive linear correlation, 0 means no linear correlation, and -1 represents a totally negative linear correlation.

4.3.2 Summary of results. Table 2 reports the values of the optimization dimensions averaged over 100 generated groups. Overall, we observe that disagreement-based consensus functions, whether pair-wise or variance, perform best in terms of all optimization dimensions, and for all different group variants. Least misery appears to be the worst aggregation method. We also observe that TPs for non-uniform groups are more cohesive than uniform groups. However, the cohesiveness of uniform groups increases with group size, while their personalization decreases. We also note that there is a tension between personalization and cohesiveness where more personalized TPs are less likely to be cohesive.

Additionally, we report the similarity between the TP of a group and its median user (Table 3). Overall, we observe that the similarity decreases in larger groups. For non-uniform groups, the best similarity values are achieved using least misery, while for uniform groups, disagreement-based methods are superior.

4.3.3 Interpretation of results. We discuss the influence of consensus functions, group uniformity, and group size, on the optimization dimensions and the agreement between individuals and groups.

Influence of consensus functions. We observe in Table 2 that TPs are generally more personalized when their associated group profile is built using a disagreement-based consensus (variance disagreement and pair-wise disagreement). Least misery is the worst consensus method for personalization. This shows that optimizing towards one single group member is not an effective personalization strategy. Incorporating inter-member disagreements is therefore the

[5] *MSB: Mean Square Between, MSE: Mean Square Error*

		average preference			least misery			pair-wise disagreement			disagreement variance		
		R	C	P	R	C	P	R	C	P	R	C	P
uniform groups	**small**	99%	31%	93%	38%	98%	75%	98%	26%	99%	98%	20%	99%
	medium	86%	55%	85%	93%	66%	75%	87%	47%	99%	98%	45%	99%
	large	99%	71%	94%	98%	68%	71%	98%	57%	99%	97%	61%	99%
non-uniform groups	**small**	15%	73%	59%	83%	76%	64%	13%	74%	38%	14%	72%	37%
	medium	21%	67%	28%	83%	81%	88%	21%	59%	14%	21%	63%	14%
	large	5%	26%	2%	48%	44%	54%	2%	22%	1%	2%	23%	5%

Table 3: Agreement between median users and groups, where the value 100% represents the highest degree of agreement. The symbol R stands for representativity, C for cohesiveness, and P for personalization.

best way to obtain POIs that satisfy everyone in a group, regardless of group uniformity. We also observe that average preference and disagreement-based methods result in similar representativity values, validating that fuzzy clustering achieves good representativity overall by strategically placing centroids on a map.

Influence of group uniformity. We observe in Table 2 that TPs for non-uniform groups are always more cohesive than TPs for uniform groups. This result generalizes previous work where a tension between personalization and cohesiveness was observed for single-member groups: the more personalized a TP is, the less likely it is to be cohesive [4]. Non-uniform groups contain members with diverse preferences. This diversity makes personalization weaker (the aggregated profile expresses lower preferences than individual profiles). Given that, cohesiveness is likely to be higher for non-uniform groups.

Influence of group size. In Table 2, we observe that regardless of the consensus function, cohesiveness increases as uniform groups grow in size. The values of PCC are $+0.98$, $+0.73$, $+0.73$, and $+0.99$ for average preference, least misery, variance disagreement, and pair-wise disagreement, respectively. As groups grow in size, their uniformity decreases yielding a weaker personalization effect, which in turn favors cohesiveness. We also observe an inverse correlation between personalization and group size for uniform groups. The values of PCC are -0.99, -0.99, -0.89, and -0.89 for average preference, least misery, variance disagreement, and pair-wise disagreement, respectively. That is explained by the fact that in larger groups, the preferences of individuals fade out and personalization decreases. This is also compatible with the previous single-user study [4].

Agreement between individuals and groups. Table 3 reports the similarity of individuals (the median user in each group) and groups they belong to. The goal is to measure the *sacrifice of individuals when joining groups.* For this aim, we compute a median profile for each of the 600 previously generated groups. The sum of Cosine values between the profile of the median user u and all other members of u's group is the highest. We generate a TP for each median user and compute the optimization dimensions for that TP. Table 3 reports the similarity between the values of optimization dimensions of a group and its median user. The higher that value, the better it is from that median user's perspective.

A general observation is that group size and group uniformity play an important role. In large groups, preferences of individuals fade out and returned TPs are farther from the the median user's preferences. Concerning cohesiveness, the highest similarity is obtained with least misery. It is also the case for personalization. Least misery yields higher similarity between the median user and the group for non-uniform groups. Both findings are consistent with previous work on group recommendation [5, 6], where least misery is more successful at satisfying the median user in larger groups with diverse tastes. For uniform groups, disagreement-based methods are best for personalization.

4.4 User Study

The goal of our user study is to observe how GroupTravel helps users obtain and refine a TP when traveling with others. The study consists of two parts. First, we focus on personalization aspects of GroupTravel and compare personalized and non-personalized TPs together. Second, we shed light on customization and observe how enabling interaction with TPs and their refinement can help improve the group travel profile, which consequently means more satisfactory TPs.

4.4.1 Setup. We recruited 3000 participants for our user study. To ensure diversity, we gathered 2000 of them from Figure-Eight platform[6], and the remaining 1000 from Amazon Mechanical Turk[7]. After pruning profiles with invalid email addresses and/or identifiers, we retained 90.1% and 96.6% of the participants in the aforesaid platforms, respectively. We based our choice for the number of study participants on Equation 5, that uses the central limit theorem [21].

$$Sample\ size = \frac{\frac{z^2 \times p(1-p)}{e^2}}{1 + (\frac{z^2 \times p(1-p)}{e^2 N})} \tag{5}$$

We describe the parameters in Equation 5 as follows.

- $N = 200,000$ is the population size, i.e., the number of contributors on Figure-Eight and Amazon Mechanical Turk platforms [22].
- $e = 3\%$ is the margin of error, i.e., the percentage of deviation in result in the sample size compared with the total population.
- $z = 95\%$ is the confidence level, i.e., if the job is repeated 100 times, 95 times out of 100 the result would lie within the margin of error.

[6] *http://www.figure-eight.com/*

[7] *https://www.mturk.com/*

		random	non personalized	average preference	least misery	pair-wise disagreement	disagreement variance
uniform	small	3.42	3.59	3.54	3.53	3.77	3.65
	medium	3.43	3.48	3.69	3.47	3.56	3.65
	large	3.52	3.58	3.72	3.62	3.78	3.70
non-uniform	small	3.01	2.68	3.28	3.28	3.23	3.19
	medium	3.01	2.94	3.17	3.19	3.21	3.26
	large	3.05	3.00	3.14	2.84	3.09	3.12

Table 4: Independent evaluation of user study

- $p = 50\%$ is the percentage value, i.e., the expected result value of the experiment. It is advised to put it at 50% when the result is not known.

Our sample size rounded up to at least 1062 participants based on the above formula. We recruited almost three times more participants to allow flexibility in forming groups and account for contributors who might quit the study before fully completing it.

We built travel profiles for the recruited participants by asking them to state their preferences on POI categories using Google Forms[8]. We then used the generated user profiles to build groups with varying characteristics, i.e., size and uniformity. For uniform groups, we generated 5 groups of each size (small, medium, and large). We gathered assessments from all members of small and medium groups, and from 30 random members for large groups. For non-uniform groups, we generated 3 groups for each size, and gathered assessments from all members of small and medium groups, and between 19 and 30 members for large groups. Each participant was paid \$0.01 for profile collection and \$0.50 for evaluating TPs.

4.4.2 Summary of results. Regarding personalization, we observe that participants liked personalized TPs more than non-personalized and random TPs. We also observe that TPs associated with average preference and least misery are the best performers for uniform groups, and TPs associated with disagreement-based methods are highly appreciated by members of non-uniform groups. Regarding customization, we noticed the supremacy of the batch strategy over the individual strategy in almost all cases.

4.4.3 Exploring personalization. In this part of the study, we aim to evaluate how satisfied users are with personalized TPs. We build personalized packages in the city of Paris. Similarly to the synthetic data experiment, each TP contains exactly 5 CIs that are valid with respect to a default query, i.e., ⟨1 *acco*, 1 *trans*, 1 *rest*, 3 *attr*⟩. Also, we specify *an infinite budget* to ensure that all popular POIs are included in the CIs. We conduct two evaluations, *independent* and *comparative*.

Independent evaluation. We asked members of the formed groups to evaluate TPs. The TPs under evaluation were either *non-personalized*, or *personalized* with one of the four group consensus methods. Non-personalized TPs were generated by setting the weight of the personalization dimension to 0 in the objective function. In addition, to filter undesired participants, we injected a random TP which included *invalid* CIs, and discarded input from participants who preferred that TP (23 participants). For each TP out of the 6 TPs to be evaluated (random, non-personalized, and personalized with the four different consensus methods), we asked the remaining 326 participants to indicate their interest in visiting POIs in the TP *with other members of the group*, using a score between 1 and 5. A score of 1 means that there are very few POIs that the participant is interested in, and a score of 5 means that the participant is interested in almost all of the POIs. To prevent bias, we did not share with participants any details about the characteristics and members of the group they are involved in.

Table 4 illustrates the results of our independent evaluation. The average interest of participants who were not filtered out is reported for groups with different sizes and uniformity categories. The results validate our objective function (Equation 1), because they show that personalized TPs perform well and are liked better than non-personalized and random TPs. We also observe that scores for uniform groups remain fairly stable as groups become larger. However for non-uniform groups, scores decrease by group size. This is in-line with our findings in the synthetic experiment where the preferences of individual members in non-uniform groups fade out as groups grow in size, resulting in less-personalized TPs.

Comparative evaluation. We also presented the participants of our study with a pair of TPs among the 6 aforementioned TPs, and asked them to choose the one that they prefer the most, and to state a reason behind their choice. Table 5 reports results for each pair-wise comparison in terms of the percentage of supremacy. For instance, for small uniform groups, **AVTP** is preferred over **LMTP** in 48% of the time implying that **LMTP** is preferred over **AVTP** in 52% of the time. We observe that TPs associated with average preference and least misery (**AVTP** and **LMTP** in Table 5) are winners for uniform groups, whereas TPs associated with disagreement-based methods (**ADTP** and **DVTP**) are winners for non-uniform groups. This finding is in-line with previous work on group recommendation [5, 6], where recommendations aggregated with either average preference or least misery are preferred for uniform groups where preferences are homogeneous. Similarly, incorporating inter-member disagreements is shown to be the best way to reach a consensus within diverse groups.

We also reviewed the statements that the participants provided to justify their choices. First we focused on cases where personalized TPs are not preferred. In these cases, participants often justified their choice only as a tie-breaker: *"I like this TP a little more"*, *"I think this TP is a bit better"*. For uniform groups, participants mentioned that they prefer TPs with average preference and least misery because those

[8] *https://docs.google.com/forms*

		AVTP vs.				**LMTP** vs.			**ADTP** vs.		**DVTP** vs.
		LMTP	**ADTP**	**DVTP**	**NPTP**	**ADTP**	**DVTP**	**NPTP**	**DVTP**	**NPTP**	**NPTP**
uniform	**small**	48%	56%	64%	48%	56%	56%	64%	36%	36%	40%
	medium	56%	42%	42%	54%	62%	56%	70%	42%	56%	42%
	large	47%	52%	44%	50%	51%	54%	55%	46%	52%	55%
non-uniform	**small**	27%	27%	66%	60%	40%	73%	40%	60%	60%	47%
	medium	43%	73%	46%	53%	73%	66%	67%	46%	43%	67%
	large	54%	42%	48%	49%	51%	36%	42%	64%	57%	48%

Table 5: Comparative evaluation of user study. AVTP, LMTP, ADTP, and DVTP refer to personalized TPs obtained with average preference, least misery, average disagreement, and disagreement variance, respectively, and NPTP refers to the non-personalized TP.

TPs reflect their personal preferences better: *"this TP seems to me more interesting for my taste"*, *"with this TP, I can move around the city more"*. We observe the same type of statements for non-uniform groups in case of disagreement-based methods, e.g., *"there are types of places in this TP that I want to visit"*.

4.4.4 *Exploring customization.*

In this experiment, we aim to validate the benefit of customization by allowing group members to interact with travel packages. More precisely, we want to assess if interacting with personalized TPs will refine the group profile in such a way that subsequent TPs are more relevant. To do so, we displayed personalized TPs on the map of Paris and asked participants to interact with the CIs forming those TPs by adding, removing, replacing POIs or generating new CIs (see the interface in Figure 3). We then refined the group profile based on the interactions of all group members. We compare the *individual* and *batch* strategies defined in Section 3.3.

With the refined group profile in Paris using either strategy, we built a customized travel package in a comparable city, namely Barcelona. We then asked our participants to evaluate the generated TP for Barcelona both in independent and comparative evaluations. Similar to the personalization study case in Section 4.4.3, to filter undesired participants, we injected a random TP which included *invalid* CIs, and discarded input from participants who preferred that TP. Participants were asked to rate the TPs in a scale of 1 to 5. A total of 18 workers participated in this study. That allowed us to build one uniform group with 11 members and one non-uniform group with 7 members. We recruited workers with an approval rate superior to 90%.

Independent evaluation. Participants in each group were asked to evaluate three different TPs in Barcelona: the first one was non-personalized, the second personalized and customized using the individual strategy, and the third personalized and customized with the batch strategy. Table 6 reports average ratings. Results are comparable across TPs. Overall, all travelers were equally satisfied with the POIs in all the TPs.

Comparative evaluation. Participants in each group were asked to compare a pair of TPs built from a non-personalized TP, a personalized and customized TP using the individual strategy, and a personalized and customized TP with the batch strategy. Table 6 reports results for each pair-wise comparison in terms of the percentage of supremacy. For instance, the batch strategy is preferred over the individual one in 82% of the time for uniform groups, implying that the individual strategy is preferred over the batch one only 18% of the time. The batch strategy is by far the best. That is particularly true for uniform groups. That is in-lined with the independent study. The intuition is that refining group profiles directly yields better TPs.

5 RELATED WORK

5.1 Itineraries and Personalization

The extraction of travel itineraries from Flickr photos was first proposed in [1] and their personalization in [2] to build customized city tours. Tailored itineraries are extracted from Flickr using an objective function that combines POI popularity with the actual user preferences over POI categories. This approach is not directly applicable to ours, since the personalization is merely a filtering of extracted trajectories. In our case, it is the POI composition itself that is personalized using the query and travel profile. That makes our problem computationally more challenging. Another difference is that unlike itineraries, POIs forming a CI are not ordered, and hence, their generation relies on a different model and algorithm (clustering instead of graph traversal). Finally, in this work we are also interested in generating travel packages for groups of users traveling together rather than just a single user.

The idea of learning travel packages was recently explored in [3]. This work proposed learning topics conditioned on both the tourists and the intrinsic features (i.e., locations and travel seasons) of landscapes. As a result, preferences on which landscapes to visit, in which season, and how to travel from one point to another (transportation modes), are extracted. This work does not propose interactive refinement of one's travel preferences and does not support group travel.

5.2 Composite Items

Composite retrieval was studied with different semantics in recent work [1, 7–10, 13–15, 23]. Most existing algorithms rely on a two-stage process that decouples the query (e.g., a CI must contain one museum and 2 restaurants) from the optimization goal (e.g., each CI is a set of close landmarks in a city). In [13], it was shown that an integrated approach produces better representative CIs than a two-stage approach. We hence build on that and extend it to build personalized CIs for groups.

TP type	uniform (11 members)	non-uniform (7 members)
individual	3.45	3.69
batch	2.91	3.8
non-personalized	3.37	3.83

Table 6: Independent evaluation of customized travel packages.

	batch vs.		individual vs.
	individual	non-personalized	non-personalized
uniform	82%	63%	54%
non-uniform	72%	57%	14%

Table 7: Comparative evaluation of customized travel packages.

5.3 Recommendation and Interactivity

Out of the multitude of itinerary recommendation approaches [24–27], only a few are interactive. In [24], a user provides feedback on the next set of POIs to visit, the system then recommends the best itineraries and further suggests new POIs, with optimal utility, to solicit feedback for. In GroupTravel, the system recommends POIs to replace those unwanted by a user or to form a CI with some selected POIs. We do not assume any prior knowledge about a city. Additionally, we focus on enabling group-based interaction with TPs.

MOBI is a collaborative itinerary planning framework [28]. Each user provides preferences and constraints in the form of "I want {at most, at least, exactly} [number] {activities, hours} of $\{cat_1, cat_2, \cdots, cat_n\}$", which resembles our query. Users interact with proposed itineraries and are told which constraints remain to be satisfied. In our work, users intervene in a second stage to refine the package. We have shown that helping bootstrap travel package construction is preferred as it induces fewer interactions.

Finally, in our previous work [4], we studied the benefit of interactivity in the generation of customized travel packages for a single user. We found that while personalization helps the selection of relevant POIs to include in a travel package, customization is necessary to allow users to customize their travel packages as they explore the alternatives the city has to offer. Customization has also been shown to help refine users' profiles based on their interactions with travel packages. In this current work, we extend the approaches and techniques we proposed for generating customized travel packages to support group travel.

6 CONCLUSION

We develop GroupTravel, a framework that generates travel package for groups. GroupTravel aggregates individual preferences into a single group preference using consensus methods developed for group recommendation. GroupTravel relies on a fuzzy clustering algorithm to generate k valid, representative, cohesive and personalized composite items that form a travel package. Travelers can interact with the generated travel package to further customize it. We run extensive synthetic and real data experiments and show that our findings are consistent with previous work in generating travel packages for single users [4] and in group recommendation [5, 6].

This work opens several research directions. One immediate challenge is to incorporate different collaboration models into the primitives used to interact with the TPs. We are examining different models such as the star model where a designated traveler moderates all requests from others in the same group, the sequential model where a TP is customized in a pipeline fashion, and a hybrid model where different primitives are requested in parallel by different travelers. This additional expressiveness raises new algorithmic questions and new ways of conducting user studies at scale.

REFERENCES

[1] Munmun De Choudhury, Moran Feldman, Sihem Amer-Yahia, Nadav Golbandi, Ronny Lempel, and Cong Yu. Automatic construction of travel itineraries using social breadcrumbs. In *HT*, pages 35–44, 2010.

[2] Igo Ramalho Brilhante, José Antônio Fernandes de Macêdo, Franco Maria Nardini, Raffaele Perego, and Chiara Renso. Where shall we go today?: planning touristic tours with tripbuilder. In *CIKM'13, San Francisco, CA, USA*, pages 757–762, 2013.

[3] Zhiwen Yu, Huang Xu, Zhe Yang, and Bin Guo. Personalized travel package with multi-point-of-interest recommendation based on crowdsourced user footprints. *IEEE Trans. Human-Machine Systems*, 46(1):151–158, 2016.

[4] Manish Singh, Ria Mae Borromeo, Anas Hosami, Sihem Amer-Yahia, and Shady Elbassuoni. Customizing travel packages with interactive composite items. In *2017 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2017, Tokyo, Japan, October 19-21, 2017*, pages 137–145, 2017.

[5] Sihem Amer-Yahia, Behrooz Omidvar Tehrani, Senjuti Basu Roy, and Nafiseh Shabib. Group recommendation with temporal affinities. In *EDBT*, pages 421–432, 2015.

[6] Sihem Amer-Yahia, Senjuti Basu Roy, Ashish Chawlat, Gautam Das, and Cong Yu. Group recommendation: Semantics and efficiency. *Proceedings of the VLDB Endowment*, 2(1):754–765, 2009.

[7] Sihem Amer-Yahia, Francesco Bonchi, Carlos Castillo, Esteban Feuerstein, Isabel Méndez-Díaz, and Paula Zabala. Composite retrieval of diverse and complementary bundles. *IEEE Trans. Knowl. Data Eng.*, 26(11):2662–2675, 2014.

[8] Albert Angel, Surajit Chaudhuri, Gautam Das, and Nick Koudas. Ranking objects based on relationships and fixed associations. In *EDBT 2009*, pages 910–921, 2009.

[9] Horatiu Bota, Ke Zhou, Joemon M. Jose, and Mounia Lalmas. Composite retrieval of heterogeneous web search. In *WWW 2014*, pages 119–130, 2014.

[10] Alexander Brodsky, Sylvia Morgan Henshaw, and Jon Whittle. Card: a decision-guidance framework and application for recommending composite alternatives. In *RecSys*, pages 171–178, 2008.

[11] Adrian Graham, Hector Garcia-Molina, Andreas Paepcke, and Terry Winograd. Time as essence for photo browsing through personal digital libraries. In *ACM/IEEE JCDL*, pages 326–335, 2002.

[12] Alexander Jaffe, Mor Naaman, Tamir Tassa, and Marc Davis. Generating summaries and visualization for large collections of geo-referenced photographs. In *SIGMM MIR*, pages 89–98, 2006.

[13] Vincent Leroy, Sihem Amer-Yahia, Éric Gaussier, and Seyed Hamid Mirisaee. Building representative composite items. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015*, pages 1421–1430, 2015.

[14] Senjuti Basu Roy, Sihem Amer-Yahia, Ashish Chawla, Gautam Das, and Cong Yu. Constructing and exploring composite items. In *SIGMOD Conference*, pages 843–854, 2010.

[15] Min Xie, Laks V.S. Lakshmanan, and Peter T. Wood. Breaking out of the box of recommendations: From items to packages. In *RecSys*, 2010.

[16] Sihem Amer-Yahia, Senjuti Basu Roy, Ashish Chawla, Gautam Das, and Cong Yu. Group recommendation: Semantics and efficiency. *PVLDB*, 2(1):754–765, 2009.

[17] Mark O'Connor, Dan Cosley, Joseph A. Konstan, and John Riedl. Polylens: a recommender system for groups of users. In *ECSCW*, 2001.

[18] Anthony Jameson and Barry Smyth. Recommendation to groups. In *The adaptive web*, pages 596–627. Springer, 2007.

[19] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Lda. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[20] James C. Bezdek, Robert Ehrlich, and William Full. FCM: The Fuzzy c-Means Clustering Algorithm. *Computers & Geosciences*, 10(2-3):191–203, 1984.

[21] SurveyMonkey. Calculating the number of respondents you need. https://help.surveymonkey.com/articles/en_US/kb/How-many-respondents-do-I-need.

[22] Djellel Difallah, Elena Filatova, and Panos Ipeirotis. Demographics and dynamics of mechanical turk workers. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pages 135–143. ACM, 2018.

[23] Sihem Amer-Yahia and Senjuti Basu Roy. Interactive exploration of composite items. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018.*, pages 513–516, 2018.

[24] Senjuti Basu Roy, Gautam Das, Sihem Amer-Yahia, and Cong Yu. Interactive itinerary planning. In *ICDE 2011*, pages 15–26, 2011.

[25] Igo Ramalho Brilhante, José Antônio Fernandes de Macêdo, Franco Maria Nardini, Raffaele Perego, and Chiara Renso. Planning sightseeing tours using crowdsensed trajectories. *SIGSPATIAL Special*, 7(1):59–66, 2015.

[26] Kwan Hui Lim, Jeffrey Chan, Christopher Leckie, and Shanika Karunasekera. Towards next generation touring: Personalized group tours. In *ICAPS*, pages 412–420, 2016.

[27] Dawei Chen, Cheng Soon Ong, and Lexing Xie. Learning points and routes to recommend trajectories. In *CIKM*, pages 2227–2232, 2016.

[28] Haoqi Zhang, Edith Law, Rob Miller, Krzysztof Gajos, David C. Parkes, and Eric Horvitz. Human computation tasks with global constraints. In *CHI*, pages 217–226, 2012.

SEP2P: Secure and Efficient P2P Personal Data Processing

Julien Loudet[1,2,3]
[1]Cozy Cloud, France
julien@cozycloud.cc

Iulian Sandu-Popa[3,2]
[2]INRIA Saclay, France
<fname.lname>@inria.fr

Luc Bouganim[2,3]
[3]University of Versailles, France
<fname.lname>@uvsq.fr

ABSTRACT

Personal Data Management Systems are flourishing allowing an individual to integrate all her personal data in a single place and use it for her benefit and for the benefit of the community. This leads to a significant paradigm shift since personal data become massively distributed. In this context, an important issue needed to be addressed is: how can users/applications execute queries and computations over this massively distributed data in a secure and efficient way, relying exclusively on peer-to-peer (P2P) interactions? In this paper, we motivate and study the feasibility of such a pure P2P personal data management system and provide efficient and scalable mechanisms to reduce the data leakage to its minimum with covert adversaries. In particular, we show that data processing tasks can be assigned to nodes in a verifiable random way, which cannot be influenced by malicious colluding nodes. Then, we propose a generic solution which largely minimizes the verification cost. Our experimental evaluation shows that the proposed protocols lead to minimal private information leakage, while the cost of the security mechanisms remains very low even with a large number of colluding corrupted nodes. Finally, we illustrate our generic protocol proposal on three data-oriented use-cases, namely, participatory sensing, targeted data diffusion and more general distributed aggregative queries.

1 INTRODUCTION

The time of individualized management and control over one's personal data is upon us. Thanks to smart disclosure initiatives (e.g., BlueButton [9] and GreenButton in US, MesInfos [16] in France, Midata [25] in UK) and new regulations (e.g., the Europe's new General Data Protection Regulation [27]), users can access their personal data from the companies or government agencies that collected them. Concurrently, Personal Data Management System (PDMS) solutions are flourishing [4] both in the academic (e.g., Personal Data Servers [1], Personal Information Management Systems, Personal Data Stores [14], Personal Clouds [20]) and industry [12, 26, 33]. Their goal is to offer a data platform allowing users to easily store and manage into a single place data directly generated by user devices (e.g., quantified-self data, smart home data, photos, etc.) and data resulting from user interactions (e.g., user preferences, social interaction data, health, bank, telecom, etc.). Users can then leverage the power of their PDMS to benefit from their personal data for their own good and in the interest of the community. Thus, the PDMS paradigm holds the promise of unlocking new innovative usages.

Let us consider three emblematic distributed applications based on large user communities which could greatly benefit from the PDMS paradigm: (1) mobile participatory sensing apps [36], in which mobile users produce sensed geo-localized data (e.g., traffic, air quality, noise, health conditions) to compute spatially aggregated statistics benefiting the whole community; (2) subscription-based or profile-based data diffusion apps [38], in which PDMS users provide preferences or exhibit profiles in order to selectively receive pertinent information; and (3) distributed query processing over the personal data of large sets of individuals [37], in which users contribute with their personal data and issue queries over the globally contributed data (e.g., computing recommendations, participative studies).

However, these exciting perspectives should not eclipse the security issues raised by the PDMS paradigm. Indeed, each PDMS can store potentially the entire digital life of its owner, thereby proportionally increasing the impact of a leakage. Hence, centralizing all users' data into powerful servers is risky since these data servers become highly desirable targets for attackers: huge amounts of personal data belonging to millions of individuals could be leaked or lost as illustrated by the recent massive attacks (e.g., Facebook, Yahoo or Equifax). Besides, such a centralized solution makes little sense in the PDMS context in which data is naturally distributed at the users' side [19].

Alternatively, recent works [4, 14, 20, 33] propose to let the user data distributed on personal trustworthy platforms under users' control. Such platforms can be built thanks to the combination of (1) a Trusted Execution Environment (TEE) (i.e., secure hardware such as smart cards [1] or secure micro-controllers [4, 5, 20], ARM TrustZone [18], or Intel SGX [29]) and (2) specific software (e.g., minimal Trusted Computing Base and information flow control [22, 29]). In this paper, we follow this approach and consider that a PDMS is a dedicated personal device that the user possesses and is secured thanks to TEE hardware.

In addition, as in many academic and commercial approaches [33], we assume that the PDMS personal device offers a rather good connectivity and availability like, for instance, home-cloud solutions [4, 12, 26, 33] (e.g., a set-top box or a plug computer [4]). Thus, PDMSs can establish peer-to-peer (P2P) connections with other PDMSs, and can be used as data processor in order to provide part of the processing required in distributed applications. Hence, our objective is to study solutions based on a full distribution of PDMSs (called nodes interchangeably) which can act as data sources and data processors and communicate in a peer-to-peer fashion. We discard solutions requiring recentralizing the distributed personal data during its processing, since this would dynamically create a personal data concentration leading to a similar risk as with centralized servers.

Incorporating TEEs greatly increases the protection against malicious PDMS owners. However, since no security measure can be considered as unbreakable, we cannot exclude having some corrupted nodes in the system and, even worse, those corrupted nodes can collude and might very well be undistinguishable from honest nodes, acting as covert adversaries [7]. Also, since data processing relies exclusively on PDMS nodes, and given the very high scale of the distribution which disqualifies secure multi-party computation (MPC) protocols [31], sensitive data leaks are unavoidable in the presence of corrupted nodes, i.e., some data

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

might be disclosed whenever a corrupted node is selected as a data processor.

The goal of this paper is to assess the feasibility of building a secure and efficient data processing system over a fully distributed network of PDMS housing covert adversaries. To achieve it we provide mechanisms to reduce the data leakage to its minimum, and make the following contributions:

(1) We propose a P2P architecture of PDMSs, called SEP2P (for Secure and Efficient P2P), based on classical Distributed Hash Tables (DHT) and analyze potential data leakages of data sources and data processors. We show that (i) data tasks should be assigned to nodes in a *verifiable random* way, i.e., the assignment cannot be influenced by malicious colluding nodes; and (ii) any data-oriented task, whether it is storage or computation, should be *atomic*, i.e., reduced to a maximum such that it minimizes the quantity of sensitive data accessible by the task.

(2) We focus on the verifiable random assignment problem and propose a generic solution (i.e., independent of the distributed computation tasks) which largely minimizes the verification cost (e.g., 8 asymmetric crypto-operations with a SEP2P network of 1M nodes of which 10K are colluding corrupted nodes).

(3) We experimentally evaluate the quality and efficiency of the proposed protocols. The verifiable random assignment protocol leads to minimal private information leakage, i.e., linear with the number of corrupted nodes, while the cost of the security mechanisms remains very low even with a large number of colluding corrupted nodes.

(4) We address the task atomicity subproblem by providing sketches of solutions for the three classes of applications indicated above. We do not propose full solutions since task atomicity is dependent on the considered class of distributed computation and as such needs to be studied in detail.

Sections 2 to 5 present these four contributions respectively. We finally discuss the related work in Section 6 and conclude the paper in Section 7.

2 SEP2P ARCHITECTURAL DESIGN

2.1 Base System Architecture

SEP2P is a peer-to-peer system and only relies on the PDMS nodes to enable the aforementioned applications. Consequently, each node may play several roles for SEP2P applications:

Node role 1. Each node is a potential **data source**. For instance, producing sensed geo-localized data about the local traffic speed, or sharing grades used to compute recommendations.

Node role 2. Given the fully-decentralized nature of SEP2P, each node is a potential **data processor**, also called **actor**, providing part of the required processing.

Node role 3. The initiator of a distributed processing is called the **triggering node (*T*)**. *T* could be any node with participatory sensing applications, or the query issuer in distributed query or data diffusion applications.

2.2 Efficient P2P Data Processing

Relying on a fully-distributed system induces several problems, e.g., integrating new nodes, maintaining a coherent global state, making nodes that do not know each other interact, handling churn, maintaining some metadata. It thus requires a communication overlay allowing for efficient node discovery, data indexing and search. Fortunately, these problems have already been extensively studied in the literature and the *Distributed Hash Tables* (DHTs) appear to be the solution reaching consensus.

Background 1. A **distributed hash table (DHT)** [23, 30, 34] in a P2P network offers an optimized solution to the problem of locating the node(s) storing a specific data item. The DHT offers a basic interface allowing any node of the network to store data, i.e., $store(key, value)$, or to search for certain data, i.e., $lookup(key) \rightarrow value$. DHTs proposals [23, 30, 34] share the concepts of keyspace or **DHT virtual space** (e.g., a 224 bits string obtained by hashing the key or the node ID), space partitioning (mapping space partitions to nodes, using generally a distance function), and overlay network (set of routing tables and strategies allowing reaching a node, given its node ID). For instance, the virtual space is represented as a multi-dimensional space in CAN [30], as a ring in Chord [34] or as a binary tree in Kademlia [23] and is uniformly divided among the nodes in the network. Thus, each node is responsible for the indexing of all the $(key, value)$ pairs where the key falls in the subspace it manages. Both the data storage and the lookup operations are thus fully distributed in a DHT. DHTs have interesting properties: uniform repartition of the data, scalability, fault tolerance and do not require any central coordination.

Hence, SEP2P leverages the classical DHT techniques as a basis for communication efficiency and scalability.

2.3 Security Considerations

In this paper, we use the terminology of ARM [35] to designate the three attack levels on a PDMS node, i.e., *hack*, *shack* and *lab attacks*. A *hack attack* is a software attack in which the attacker (the PDMS owner or remote attacker) downloads code on the device to control it. A *shack attack* is a low-budget hardware attack, i.e., using basic equipment and knowledge. Finally, a *lab attack* is the most advanced, comprehensive and invasive hardware attack for which the attacker has access to laboratory equipment, can perform reverse engineering of a device and monitor analog signals. Note that shack and lab attacks require a physical access to the device and that TEEs are designed to at least resist hack and shack attacks.

Our threat model considers three security assumptions:

Assumption 1. *Each PDMS is locally secured by using TEE-like technology flourishing nowadays* (e.g., [18, 20, 29]). This assumption is reasonable considering that a PDMS is supposed to store the entire digital life of its owner. A major security feature of TEE technology is to provide isolation, i.e., strong guarantees that the local computation inside the TEE cannot be spied upon, even in the presence of an untrusted computational environment. Hence, to break to confidentiality barrier of a TEE, a lab attack is mandatory. This has an important consequence: *an attacker cannot conduct a successful attack on a remote node, i.e., not under her possession.*

Assumption 2. *Each PDMS device is supplied with a trustworthy certificate attesting that it is a genuine PDMS.* Without this assumption, an attacker can easily emulate nodes in the network, and conduct a Sybil attack [11], mastering a large proportion of nodes (e.g., playing the role of data processor nodes), thus defeating any countermeasure. Note that this does not require an online PKI (the certificate can be attached to the hardware device and not to the device owner).

Assumption 3. *Corrupted nodes by a lab attack behave like covert adversaries*, i.e., they derive from the protocol to obtain private information only if they cannot be detected [7], as detected malicious behavior leads to an exclusion from the system.

2.4 Threat Model

The above considered assumptions already offer a certain level of security at the node and system levels. Yet, no hardware security can be described as unbreakable. Therefore, our threat model considers that an attacker (e.g., one or several colluding malicious users) can possess several PDMSs and conduct lab attacks on these devices, thus mastering several corrupted nodes which can collude. For simplicity, we will call them *colluding nodes*.

It is important to notice that the worst-case attack is represented by the maximum number of colluding nodes in the system (i.e., controlled by a single entity). Corrupting few nodes can lead to some private data disclosure, but this will be very limited in a well-designed system with a large number of nodes. Therefore, an attacker needs to increase the collusion range to fully benefit from the attack (i.e., access a significant amount of private data).

Thereby, the remaining question is: how many colluding nodes could an attacker control in the system? The main difficulty for an attacker is that colluding nodes must remain indistinguishable from honest nodes (see Assumption 3). Since PDMSs are associated to "real" individuals (e.g., by delivering the device only to real users proving their identity), collusions between individuals remains possible (hidden groups) but such collusions cannot scale without being minimally advertised, hence breaking the indistinguishability mentioned above. Thus, wide collusions are extremely difficult to build since it requires significant organization between a very large number of users, which in practice requires an extremely powerful attacker as well as extreme discretion, and are thus the equivalent of a state-size attack. Finally, note that considering a large proportion of colluding nodes (e.g., 10%) is vain as it would inexorably lead to large disclosure whatever the protocol having a reasonable overhead (e.g., outside the MPC scope). Hence, in this paper we consider that a very powerful attacker could control up to a small percentage (e.g., 1%) of the nodes, which corresponds to a wide collusion requiring a lab attack on these nodes as well as a highly organized collusion between the owners of those nodes.

What does the system protect? The objective of SEP2P is to offer the maximum possible confidentiality protection of the user private data under the above considered threat model. Many other issues related to statistical databases (e.g., inferences from results, determining the authorized queries, query replay, fake data injection, etc.) or to network security (e.g., message drop/delay, routing table poisoning [39]) are complementary to this work and fall outside of the scope of this paper. Similarly, we leave aside the problems related to the attestation and integrity of the code executing distributed computations (e.g., against corrupted nodes that maliciously modify the computation results).

2.5 SEP2P Requirements

Given the considered threat model, we derive in this section the requirements that a SEP2P must address to protect the data privacy of the users. Since we cannot exclude having colluding nodes in the system and since the colluding nodes behave like covert adversaries, private information leakage is unavoidable. Under these conditions, the best countermeasures one can take are: (i) *minimize the risk* of a data leakage, i.e., reduce at most the probability of a leakage to happen; and (ii) *minimize the impact* of a data leakage, i.e., reduce at most the leakage size. Obviously, these countermeasures should not generate overheads that render the system unpractical. This leads to:

Requirement 1 (security). Random actor selection. Ensure that colluding nodes cannot influence the selection of the data processor nodes.

Requirement 2 (security). Task atomicity. Data tasks should be atomic, i.e., reduced to a maximum such that it minimizes the required sensitive data to execute the task.

Requirement 3 (efficiency). Security overheads. Minimize the number of costly operations, e.g., cryptographic signature verifications or communication overhead, and ensure system scalability with an increasing number of nodes or colluding nodes.

The task atomicity requirement is similar to the principle of compartmentalization in information security, which consists in limiting the information access to the minimum amount allowing an entity to execute a certain task. Typically, a node can execute a subtask without knowing the purpose or the scope of the global task. Dividing a given distributed computation in atomic tasks obviously depends on the precise definition of that computation. Hence, we restrict our analysis in Section 5 to sketches of solutions for the three application classes considered in this paper.

Independently of the distributed protocol chosen to implement some given application, the system must delegate the data-oriented tasks to randomly selected nodes. Therefore, the random selection protocol is generic and constitutes the security basis of any distributed protocol in our system. However, given the considered threat model, it is challenging to design an actor selection protocol that is both secure and efficient. Section 3 addresses this problem while section 4 evaluates the proposed solution.

3 SECURE ACTOR SELECTION

Let us first detail some useful classical cryptographic tools focusing on the properties used in our protocol.

Background 2. A **cryptographic hash function** [24] is a one-way function that maps a data of arbitrary size to a fixed size bit string (e.g., 224 bits) and is resistant to collision. An interesting property of hash functions is that output distribution is uniform. In the following, *hash*() refers to cryptographic hash.

Background 3. A **cryptographic signature** [24] can be used by a node n to prove that a data d was produced by n (authentication) and has not been altered (integrity). The signature is produced by encrypting $hash(d)$ using the private key of n. Any node can verify the signature by decrypting it using the public key of n and comparing the result with $hash(d)$. The signature includes the signer public key certificate, $cert_n$ (see Assumption 2).

We consider a system of N nodes, in which we want to randomly select A actors, despite wide collusion attacks from C colluding nodes. The main notations are summarized in Table 1.

3.1 Effectiveness, Cost and Optimal Bounds

Ideally, we would want to ensure that all A actors are honest, but this is impossible, since colluding nodes are indistinguishable from honest nodes. Therefore, the best achievable protection is obtained when actors are randomly selected and the selection cannot be influenced by C colluding nodes, i.e., the average number of corrupted selected actors in the ideal case is $A_{ideal_C} = A \times C/N$

N	Total number of nodes in the SEP2P system
A	Number of actor nodes (data processors)
C	Maximum number of colluding nodes ($C \geq 1$)
A_C	Average number of corrupted actors for a given protocol
A_{ideal_C}	Average number of corrupted actors for an ideal protocol
T	Triggering node (starting the execution)
k	Security degree
α	Security threshold
S	Execution Setter node, computing actor list
R_i, rs_i	DHT region R_i of size rs_i

Table 1: Main notations for Sections 3.1 and 3.2

($A_{ideal_C} > 0$). Thus, the impact of a collusion attack remains proportional with the number of colluding nodes, which is the best situation given our context. This guarantees that the attacker cannot obtain more private information than what she can passively get from observing the information randomly reaching its colluding nodes.

The following definitions quantify the security effectiveness and security cost of an actor selection protocol.

Definition 1. The **security effectiveness** of an actor selection protocol is defined as the ratio between A_{ideal_C} and the average number of corrupted selected actors for the measured protocol (A_C), i.e., security effectiveness = A_{ideal_C}/A_C. The security effectiveness has maximum value (i.e., 1) when $A_C = A_{ideal_C}$ and minimum value (i.e. C/N) when all the actors are corrupted.

Definition 2. A **verifier node** is a node who needs verifying the actor list before delivering sensitive data, e.g., a data source.

Definition 3. The **security cost** of an actor selection protocol is defined as the number of asymmetric cryptographic operations, e.g., signature verification, required by verifier nodes to check the selected actor list.

Note that the security cost considers only the verification of the actor list and not the cost of building the list. The rationale is that the verification cost has a larger impact on the overall performance since the number of verifier nodes can be high in a large distributed system: data sources need to verify the actor list before delivering their data. Other performance related issues (cost of the actor list generation, load balancing, maintenance costs) are discussed in Section 3.6 and 4.

Optimal bounds. The best possible case one could expect in terms of security effectiveness and cost in our context can be achieved using an idealized trusted server that knows all the nodes and provides a different random actor list for each system computation. This ideal solution reaches a maximal security effectiveness and a security cost of 1, since any verifier node must only check the signature of the trusted entity.

Evidently, this solution in not acceptable since it represents a highly desirable target for attackers, i.e., a central point of attack and contradicts the fully distributed nature of SEP2P. Therefore, we need distributed solutions relying only on the nodes. To underline the existing tension between security effectiveness and cost, we discuss two basic distributed protocols for the actor selection, focusing either on the security cost or on the security effectiveness. To simplify the protocols description, we initially assume a full mesh network overlay, i.e., each node knows the complete list of nodes in the system and its evolution over time.

Baseline cost-optimal protocol. The triggering node (T) selects randomly the actors. The security effectiveness is minimal: $A_C = min(A, C)$ since T may be corrupted (which is the case when any node can trigger a computation). There is thus no necessity to provide any signature: the security cost is 0.

Baseline security-optimal protocol. Proposing an optimal protocol in terms of security is challenging in a decentralized architecture (without any supporting trusted party) and considering covert adversaries. This conjunction leads to a situation where no single node in the system can claim to securely provide a list of actors (the provider itself can be corrupted). The work in [8] proposes the CSAR protocol which provides a secure way to generate a verifiable random value under the condition that there is at least one honest node participating in the distributed protocol. Applying to our context, we can ensure generating a real random value only if there are at least $C + 1$ participating nodes. Also, once we obtain a verifiable random value, we can derive up to A random values by repeatedly hashing the initial value $A - 1$ times. The final step is to map the set of A random values to the nodes. This can be easily done, e.g., by sorting the nodes on their public key and associating the random value to a rank in the sorted list. This protocol has an optimal security effectiveness, i.e., 1, since the actors are guaranteed to be selected randomly. On the other hand, checking the CSAR results requires one signature verification per participant. Thus, the security cost is $C + 1$ asymmetric cryptographic operations per verifier node. Since C can be large, such a solution cannot scale with large systems and wide collusion attackers as it would lead to an extreme verification cost.

Moreover, to achieve these security bounds, both protocols require a full mesh network overlay which is also extremely costly to maintain in practice, especially for large networks. This contradicts the efficiency and scalability requirement formulated in Section 2.5. Using a DHT overlay instead of a full mesh solves the problem of communication efficiency/scalability. However, this will impact the optimal bounds of both protocols. For the first protocol, the security cost increases from 0 to up to A since a verifier node which does not "know" any of the actors has to verify their certificates to be sure that the actors are genuine PDMSs (to avoid Sybil attacks). Similarly, for the second protocol, the security cost increases to $2(C + 1) + A$ for the same reason, i.e., checking that participant and selected actors are genuine PDMSs. Even worse, the optimal security effectiveness can no longer be guaranteed since with a DHT, there is no secure way of associating the random values to the nodes unless using secure DHT techniques [39] with a large impact on performance.

3.2 Overview of the Proposed Solution

To address all these problems, we propose a protocol that reaches maximal security effectiveness at a verification cost of $2k$. k is called the *security degree* and is very small. Also, our protocol builds directly on a classical, efficient DHT overlay without requiring any modifications. We describe some important features in SEP2P which make this possible and then sketch the protocol.

Imposed and uniform distribution of node location: the node ID, used when inserting a node in the DHT, is imposed in SEP2P, in a way that leads to a uniformly distributed node location in the DHT virtual space. Consequently, colluding nodes are also evenly distributed in the DHT, thus avoiding spatial clusters. We use extensively this property to drastically reduce the cost of security by taking localized decisions (see below), i.e., limited to the nodes situated in "small" regions in the virtual space. Achieving imposed node location is easy, based on the public key of the certificate of each node. We compute a cryptographic hash of this key, which is, by construction, uniformly distributed, and use this hash for insertion in the DHT virtual space. The advantages of using the public key are (i) its uniqueness; and (ii) the node location can be checked with a single signature verification.

Probabilistic guarantees: Given the imposed, uniform node location which applies indistinctly to honest and colluding nodes, we can have probabilistic guarantees on the maximum number of colluding nodes in a DHT subspace of a given size, called *DHT region* hereafter. We can compute the probability of having **at least** k **colluding nodes** (see Section 3.3) and choose the DHT region size such that the probability is very close to 0. In our context, we want to have a probability smaller than α, the security threshold. The main idea is to set α so that the probability of having k colluding nodes in the same region becomes so low that we can consider that it "never happens", e.g., $\alpha = 10^{-6}$ (see Section 4.1). Such a guarantee is used in the protocol sketched below and then detailed in the following subsection.

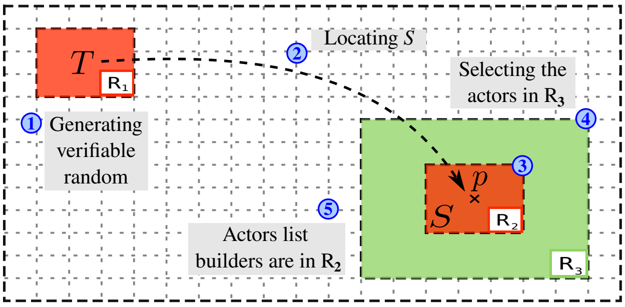

T
R₁
Locating S
2
Selecting the actors in R₃
4
1 Generating verifiable random
3
p
S
R₂
5
Actors list builders are in R₂
R₃

Figure 1: Sketch of verifiable selection

Sketch of verifiable selection protocol of A ***actors (see Figure 1)***

(1) Run a distributed protocol inspired from CSAR [8] to generate a **verifiable random value**, i.e., proven to have been truly randomly generated by k nodes if at least one is honest (see Section 3.4). The k nodes are selected in a DHT region R_1, centered on the triggering node (T), whose region size rs_1 is set such that we have probabilistic guarantees to "never" (probability $< \alpha$) have k or more colluding nodes, i.e., at least one of the k nodes is honest.

(2) Map the hash of that random value into coordinates to define a location p in the DHT virtual space and contact through the DHT the node, called **execution Setter** (S), managing this location.

(3) S then selects k nodes (the actor list builders) in a region R_2, centered on p, using probabilistic guarantees, such that we "never" have k or more colluding nodes. Given the uniform distribution of the node on the virtual space, we have $rs_2 = rs_1$.

(4) Each actor list builder then selects A nodes in a region R_3, centered on p, whose size rs_3 is such that R_3 includes at least A nodes with high probability (see Section 3.6 and Section 4.3 for rs_3 tuning).

(5) Run a **distributed verifiable selection protocol** in the spirit of [8] such that the k nodes selected in (3) can: (i) check the validity of the random value generated in (1); (ii) build the actor list securely; (iii) sign both the random value and the list of A actors. This step is detailed in Section 3.5.

The result is a list of A actors that is signed by k nodes, among which at least one is honest. Doing so reduces the verification cost to $2k$ asymmetric cryptographic operations: k to check the certificate of the k list builders, verifying that they belong to region R_2, centered on p; and k to check each builder signature.

3.3 Providing Probabilistic Guarantees

To generate verifiable random values or validate the query actor selection, SEP2P employs distributed computations between a small subset of the nodes thanks to the notion of node legitimacy and probabilistic guarantees defined below using the notations in Table 2.

$kpub_n$	Public key of node n
$cert_n$	Trustworthy certificate of node n
$sign_n$	Signature by node n (includes $cert_n$)
TL_i	execution Trigger Legitimate node i
RND_i	Random number generated by TL_i
$(V)\,RND_T$	(Verifiable) random generated by T
SL_j	execution Setter Legitimate node j
RND_S	Random generated by S
CL_j	Partial candidate list of legitimate nodes w.r.t. R_3
CL	Candidate List of legitimate nodes
$(V)\,AL$	(Verifiable) Actor List

Table 2: Main notations for Sections 3.3 – 3.5

Definition 4. Legitimate nodes. Given a region R in the virtual space of a DHT, for any node i we say that node i is legitimate w.r.t. R iff $hash(kpub_i) \in R$.

To be able to provide probabilistic guarantees as explained in Section 3.2, we need to estimate the number of nodes in a region:

Lemma. Let R be a DHT region of size rs in a virtual space of a DHT of total size 1 (i.e., normalized) and let N be the total number of network nodes with a uniform distribution of the node location in the virtual space. The probability, PL, of having at least m legitimate nodes in R is:

$$PL(\geq_m, N, rs) = \sum_{i=m}^{N} \binom{N}{i} \cdot rs^i \cdot (1-rs)^{N-i} \qquad (1)$$

Proof (sketch): Let us consider a partition of the N nodes into two subsets containing i and $N - i$ nodes. Since the distribution of nodes is uniform in space, the probability of having the i nodes inside R and the $N - i$ nodes outside R is $rs^i \cdot (1 - rs)^{N-i}$ and there are $\binom{N}{i}$ possible combinations of generating this node partitioning. The probability of having at least m nodes in R is equal to the probability of having exactly m nodes plus the probability of having exactly $m+1$ plus... the probability of having N, which leads to the equation in (1).

Application to colluding nodes: Let $C < N$ be the maximum number of colluding nodes. We can apply formula 1 to compute the probability, PC of having at least k colluding nodes in R:

$$PC(\geq_k, C, rs) = \sum_{i=k}^{C} \binom{C}{i} \cdot rs^i \cdot (1-rs)^{C-i} \qquad (2)$$

We can notice that this probability only depends on C. It does not depend on the region center since we have a uniform distribution of the nodes on the virtual space.

3.4 Verifiable Random Generation

Our goal is to generate a random value, using k nodes and to guarantee that none of the k nodes can choose the final computed random value (or any of its bits). Any node in the system should be able to check the validity of this random value (i.e., to have proofs that it has been correctly generated). This is possible as soon as at least one of the k nodes is honest, this guarantee being obtained thanks to equation (2) by choosing the adequate size for the DHT region R and by using k legitimate nodes w.r.t. R.

A node T wanting to generate a verifiable random, selecting a region of size rs_1 with $PC(rs_1) < \alpha$ centered on itself, executes:

Verifiable random number generation protocol

(1) T contacts any k legitimates nodes TL_i ($i \in [1, k]$) w.r.t. R_1.
(2) Each TL_i sends $hash(RND_i)$ to T, where RND_i is a random number (on the same domain as the hash function, e.g., 224 bits) TL_i generates.
(3) Once T has received the k hashes, it sends back the list L of hashes to the TL_is; $L = (hash(RND_i))_{i \in [1,k]}$.
(4) Each TL_i checks that $hash(RND_i) \in L$, and, in the positive case, returns $sign_i(L)$ and RND_i.
(5) T gathers the k messages and builds the verifiable random: $VRND_T = (cert_T, (sign_i(L), RND_i)_{i \in [1,k]})$.

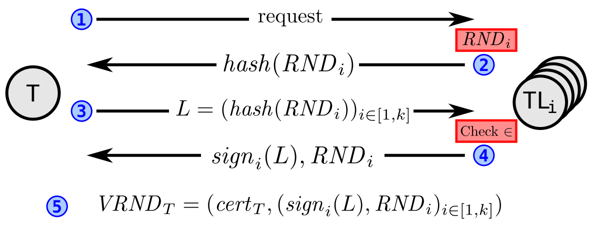

1
request
RND_i
hash(RND_i)
2
T
TL_i
3
L = (hash(RND_i))_{i∈[1,k]}
Check ∈
sign_i(L), RND_i
4
5
VRND_T = (cert_T, (sign_i(L), RND_i)_{i∈[1,k]})

Figure 2: Verifiable random

The above random generation protocol is adapted from [8] which includes a formal proof. Note that the protocol in [8] does not include the notion of node legitimacy and thus needs $C + 1$ participating nodes instead of k. Intuitively, the nodes commit on their selected random value by sending its hash (Step 2), and all the hash values are known by each of the k nodes before providing the final signature (Step 4). Therefore, an attacker controlling $k - 1$ TL_i nodes cannot influence the final random value since these nodes cannot change their random values (committed at Step 2). Thus, the correct random value of a single honest node is enough to obtain a truly random final value RND_T.

To obtain and check the verifiable random value, any node must: (i) check $cert_T$ and compute L by hashing all RND_i; (ii) for $i \in [1, k]$, check $cert_i$, check the legitimacy of TL_i using $cert_T$ and validate $sign_i(L)$. The final random value is $RND_T = RND_1 \oplus RND_2 \oplus \cdots \oplus RND_k$.

In (i), we verify that T is a genuine PDMS, retrieve the center of the region R_1 and compute L, both being necessary for the next verification; (ii) starts by confirming that each TL_i is genuine, then it ensures that they are legitimate w.r.t the location of T and R_1, after which it confirms the hash list by checking the signatures, and finally, it computes RND_T.

3.5 Distributed Secure Selection Protocol

The main goal of the proposed protocol is to select the A actors such that this selection cannot be influenced by colluding nodes.

Definition 5. The **execution Setter** (S) is chosen randomly based on a verifiable random generated by T. Its role is to coordinate the selection of the computation actors and to setup the execution by sending the appropriate information to each actor.

In the following, we assume that each node n in SEP2P keeps a node cache, called $cache_n$, of the IP address and certificate of legitimate nodes w.r.t. a region of size rs_3 centered on node n location. The cache size and the cache maintenance cost are discussed in Section 3.6 and evaluated in Section 4.3.

SEP2P distributed secure actor selection protocol

(1) Generates the verifiable random $VRND_T$ (see Section 3.4).
(2) Maps $hash(RND_T)$ into coordinates and contact S through the DHT.
(3) S contacts any k legitimates nodes w.r.t. R_2, SL_j ($j \in [1, k]$) and sends to each $VRND_T$ (see Section 3.4).
(4) Each SL_j sends $hash(RND_j \| CL_j)$ to S, where RND_j is a random number SL_j generates, and CL_j is the set of nodes from $Cache_j$ which are legitimate w.r.t. R_3.
(5) Once S has received the k hashes, it sends back the list L_1 of hashes to all SL_j; $L_1 = (hash(RND_j \| CL_j))_{j \in [1,k]}$.
(6) Each SL_j checks that its own $hash(RND_j \| CL_j) \in L_1$ and, in the positive case, returns RND_j and CL_j.
(7) S gathers the k messages and sends to all SL_j the list $L_2 = ((RND_j, CL_j)_{j \in [1,k]})$.
(8) Each SL_j does the following:
    (a) Checks $VRND_T$ and computes RND_T (see Section 3.4).
    (b) Checks that each (RND_j, CL_j) from L_2 is consistent with the corresponding $hash(RND_j \| CL_j)$ from L_1.
    (c) Computes the union, after removing possible duplicates, of all CL_j to obtain a candidate list of legitimate nodes CL.
    (d) Computes the $RND_S = RND_1 \oplus RND_2 \oplus \cdots \oplus RND_k$.
    (e) Sorts CL on $kpub_n \oplus RND_S$ (where $kpub_n$ is the public key of a node $n \in CL$) and selects the A first candidates to build the actor list AL.
    (f) Checks the legitimacy of AL nodes w.r.t. R_3.
    (g) Signs (RND_T, AL) and sends it to S.
(9) S gathers k results and builds the verifiable actor lists: $VAL = (RND_T, AL, (sign_j(RND_T, AL)))_{j \in [1,k]}$.

The goal of **steps 1 and 2** is to displace the DHT region, where actors will be selected, from T to S with three benefits: (1) T is likely to be corrupted (as any node is allowed to trigger a computation) while S is chosen randomly using the verifiable random protocol; (2) it distributes the potential leaks in a different region for each computation; (3) it balances the load on the whole SEP2P network thus improving the overall performance.

Steps 3 to 6 are similar to steps 1 to 4 of the verifiable random protocol, except that the signature by SL_j is delayed to Step 8.g. Delaying the signature allows SL_js to check and attest the validity of $VRND_T$ (step 8.a). The protocol cost is increased (since k nodes verify $VRND_T$) but the verifying cost is reduced accordingly since having k SL_js signing RND_T (step 8.g) means that it is correct (remind that at least one of the k SL_js is honest).

Steps (8.b) to (8.e) are dedicated to the actor list building (AL) based on the candidate list (CL) and deserve a more detailed explanation: in our context, in order to securely build the actor list, the k participants first have to agree on a common basis and then execute, in parallel, a procedure that is unpredictable and gives identical results to all participants. Since it is unpredictable we are certain that the inputs cannot be manipulated beforehand so as to influence the rest of the procedure. Since it gives identical results for all actor list builders, and since at least one node is honest, we are sure that no colluding node can alter the results. By sorting the nodes in CL using a verifiable random number and the public keys of the nodes fulfills both requirements: the random number takes care of the unpredictability, while the commitment of each SL_j on their intermediary lists in step 4, coupled with the XOR operation on the public keys of CL nodes, is a simple yet effective way of producing identical results.

In **steps 8.f and 8.g**, k SL_js check the validity of the result, i.e., that any actor of AL belongs to R_3 and attest it by signing the results. Note that this check is not necessary for any actor n in AL that was found in k CL_j since this fact attests that at least one honest node possesses n in its $Cache_j$. Assuming $Cache_j$ contains only genuine nodes (we say that $Cache_j$ is valid - see Section 3.6) and since $rs_3 > rs_2$, most of the actors in AL will be found in k CL_j, thus diminishing drastically the actor list building cost. Actually the validity of $Cache_j$ is necessary to ensure that a colluding node selected as SL cannot hide honest nodes with the hope of having a larger proportion of colluding nodes in AL. Indeed, at least one of the SL is honest and will provide its full $Cache_j$ that will be thus included in CL. We can observe that $Cache_j$ can be actually seen as the relevant part (for node j) of a full mesh network, which offers its benefits without paying the whole maintenance cost.

To check the verifiable actor list (VAL), any verifier node must do: for $j \in [1, k]$, check $cert_j$, check the legitimacy of SL_j using RND_T and validate $sign_j(AL)$. Thus, the verifying cost is limited to k certificate verifications and k signature verifications, i.e., $2k$ asymmetric crypto-operations. We show in Section 4 that k is generally lower than 6.

3.6 Protocol Implementation Details

In this section we discuss a few important implementation issues of the proposed actor selection protocol.

Despite the uniform distribution of nodes on the DHT virtual space, there is no absolute guarantee of not having **sparse DHT regions**. This can have two negative impacts on the SEP2P protocol: during the selection of k TLs in R_1 (or k SLs in R_2) and A actors in R_3. Both cases exhibit interesting trade-offs:

Choosing R_1 (or R_2) region size: on the one hand, a small rs leads to a smaller k value, which in turn reduces the protocol verification cost. On the other hand, setting rs too small can lead to situations in which nodes have less than k legitimate nodes in their R region and as such cannot participate in the actor selection protocol (as triggering node or execution setter) which is problematic. For this reason, in SEP2P we provide a table of couples (k_i, rs_i), named **k-table**, which allows any node to find k_i legitimate nodes in the region of associated rs_i size. The k-table is computed thanks to PL and PC (equations (1) and (2)) to ensure that whatever the couple chosen, the probability of having k or more colluding nodes remains equal. The largest k of the k-table corresponds to the region size allowing any node to find those legitimate nodes with a very high probability, i.e., 1-α, while lower values allow to reduce the security cost in denser network regions. Thus, the k-table optimizes the overall cost of the SEP2P protocol and warrants that any node can be selected as triggering node or execution setter.

Choosing R_3 region size: Choosing a too small rs_3 has a negative impact on the system performance. If the SLs cannot find enough nodes in R_3, they can attest it (e.g., in Step 8.c in SEP2P protocol) and S can use the k signatures to displace the actor selection to another region (e.g., selected by rehashing the initial RND_T). This mechanism allows the protocol to be executed successfully even if some network regions are sparser. However, there are two drawbacks. First, the cost of the actor selection increases since (part of) SEP2P protocol must be executed twice (or more times). Second, this also introduces an unbalance in the system load since the sparse regions cannot fully take part in data processing. Finally, setting rs_3 to very large values (see Section 4.3) is not an option since the maintenance cost of the cache increases proportionally when nodes join or leave the network.

Joining the network and $Cache_j$ validity: Due to space limitation, we only sketch the joining procedure in the case of a Chord DHT (leaving the network can be easily deduced). As mentioned above, any node must maintain a consistent node cache despite the natural evolution of the network. Thus, a node joining the network must ask its successors and predecessors (Chord DHT) to provide their node cache attested by k legitimate nodes in a region of size rs_1 centered on their location. The new node can then make the union of these caches and keep only legitimate nodes w.r.t R_3 centered on its location. The resulting cache contains only genuine nodes and is thus valid since it has been attested by at least k nodes in a region of size rs_1 centered on the successors or predecessors of the new node (a recurrence proof can be established).

Reusing an actor list: If there is no mechanism that prevents an attacker from reusing an actor list, then she only has to keep generating such lists until she obtains one she deems satisfactory. To counter this behavior, we put in place two mechanisms: (i) a timestamp and (ii) a limit to the number of triggered executions a node can make. With (i) we prevent any node from reusing an actor list: TLs and SLs add a timestamp to their signatures which will respectively be checked by the SLs and the data sources. If the timestamp is too distant, the computation is cancelled. Enforcing (ii) is possible thanks to the node cache and the k-table: the TLs solicited by T first check if T chose the smallest possible number of TLs (as their node cache contains, by construction, R_1 centered on T, they are capable of judging), thus forcing T to choose the same TLs. They then only have to monitor and limit the number of queries T does in a given amount of time.

Failures and disconnections: In the most complex case of node failures (i.e., unexpected disconnection) of a TL, SL or S, either RND_T or AL cannot be computed and the protocol must be restarted (i.e., T generates a new RND_T). However, the probability of failures during the execution of the secure actor selection being low in our context, such restarts do not lead to severe execution limitations as mentioned above. The case of "graceful" disconnections is easier: we can safely force nodes involved in the actor selection process to remain online until its completion, thus avoiding the restarts. If a node, selected as actor wants to disconnect (or fails), the impact will be mainly on the result quality since part of the results will be missing.

4 EXPERIMENTAL EVALUATION

This section evaluates the effectiveness, efficiency, scalability and robustness of the SEP2P actor selection protocol.

4.1 Experimental Setting

Reference methods. To better underline our contributions and to provide a comparison basis, we implemented three strategies in addition to the SEP2P actor selection protocol. We discarded the baseline cost-optimal and security-optimal protocols from the evaluation since the former does not provide any security while the latter is much too costly and not scalable (w.r.t. N and C) to be used in practice. Hence, we used for comparison more advanced actor selection strategies based on these protocols but using our verifiable random generation protocol with k participants (see Section 3.4).

The first two strategies use the verifiable random to designate the execution Setter (S) which freely chooses the actor list (as in the cost-optimal protocol). These strategies differ only in the verification process. The first one, ES.NAV (for Execution Setter, No Actor Verification) requires verifying the legitimacy of S but not of the actors. The second one, ES.AV requires, in addition, to verify that actors are genuine PDMSs. ES.AV is expected to provide better security effectiveness than ES.NAV at a higher verification cost. The third strategy, M.Hash (for Multiple Hash) is derived from the security optimal protocol, but uses a DHT instead of a full mesh network. Verifiers must check that actors are genuine PDMSs and that they are "near" the random values determined by the initial verifiable random, hashed as many times as there are actors.

Strategy	**Description**
ES.NAV	Execution Setter with No Actor Verification
ES.AV	Execution Setter with Actor Verification
M.Hash	Multiple Hash (with Actor Verification)
SEP2P	Proposed protocol (Section 3.5)

Param.	**Description**	Values(**default**)
N	Number of nodes	10K; **100K**; 10M
$C\%$	% of colluding nodes	0.001%; 0.01%; 0.1%; **1%**; (10%)
A	Number of actors	8; 16; **32**; 64; 128; 256
α	Security threshold	10^{-4}; $\mathbf{10^{-6}}$; 10^{-10}
$\mid Cache_j \mid$	Node cache size	**48** or varying from 8 to 32K
MTBF	Mean time betw. failure	from **1h** to 5 days

Metrics	**Unit(s) & comments**
Security effectiveness	Ratio (1 = ideal, C/N = worst)
Verification cost	Number of asymmetric crypto-operations
Latency of setup cost	Number of exchanged messages and number of asymmetric crypto-operations (per minute for the maintenance overhead)
Total work setup cost	
Maintenance overhead	
Security degree (k)	Ratio (1 = ideal, C/N = worst)

Table 3: Strategies, parameters and metrics

Simulation platform. We identified all the parameters that may impact the security and efficiency of the proposed strategies and considered all the metrics (see Table 3) that are worth evaluating to analyze the strengths and weaknesses of the proposed strategies, i.e., security effectiveness and cost, setup cost, scalability, robustness w.r.t. failure or disconnections. Let us note that a real implementation of the SEP2P distributed system is not very useful if we consider the above listed objectives of the evaluation. Also, measuring the scalability for very large systems (e.g., 10M nodes) with many parameters is practically impossible. Therefore, as in most of the works on distributed systems [30, 34], we base our evaluation on a simulator and objective metrics. That is, the latency is measured as the number of asymmetric crypto-operations and exchanged messages between peers instead of absolute time values. This allows for a more precise assessment of the system performance than time latency, which can greatly vary in our context because of the node heterogeneity (e.g., TEE resources or network performance).

Our simulator is built on top of a DHT network. Currently, we implemented Chord and CAN as DHT overlays and use Chord for the results presented in this paper. The simulator allows to force choosing a given Execution Setter (by artificially fixing the RND_T value). We used this feature to obtain the exhaustive set of cases for a given network setting, each node being the Execution Setter, and then capture the average, maximum and standard deviation values for our metrics. The parameters and metrics of the simulator are described in Table 3. Values in bold are the default choices and their tuning is discussed throughout the Section. Note that (1) the verification cost is given by verifier node; (2) the latency indicates the "duration" of the protocol executed in parallel; (3) the total work indicates the cumulative number of cryptographic operations and communications during the execution of a protocol.

Security threshold value: Generating several networks and varying the security threshold α, we experimentally observed that for $\alpha = 10^{-4}$, an attacker never controls k or more nodes. However, given the importance of this parameter for the system security, we set $\alpha = 10^{-6}$ and show in Figure 6 the impact of choosing $\alpha = 10^{-10}$ on a small ($10K$) and large ($10M$) network. Indeed, if an attacker could master by chance k colluding nodes in a region of size $rs_1 = rs_2$, then she could completely circumvent the security mechanism of SEP2P since, for example, she can obtain k signatures from these regrouped colluding nodes for an actor list of her willing. Note that increasing α reduces the probability accordingly but increases the verification cost in a logarithmic way (as discussed below in Section 4.3).

4.2 Security Effectiveness versus Efficiency

Figure 3 represents the security effectiveness (Y axis) versus the verification cost (X axis) for the four measured strategies and with $C\%$ varying from 0.001% to 10%. Note that the value of 10% is not realistic: it would lead to large disclosure even with an optimal random actor selection protocol, and as mentioned in Section 2.4, is equivalent to state-size attack. We have however run the simulation with 10% to understand its impact on the security effectiveness and cost.

Security effectiveness: SEP2P achieves an ideal security effectiveness, i.e. as good as a trusted server, independently of the number of colluding nodes. Indeed, the selection of actors is truly random, thus providing the same results as the ideal case. In addition, the verification cost ($2k$) is also very low (4 to 8 asymmetric crypto-operations for $C\% \leq 1\%$). Not surprisingly ES.NAV has the same verification cost than SEP2P, but the cost of ES.AV or M.Hash is much larger ($2k + A + 1$ and $2k + A$ respectively) since both must check the certificate of each actor in the list. This check allows ES.AV to have better security effectiveness than ES.NAV when C is very small ($C < A$). With respect to security effectiveness, ES.NAV, ES.AV and M.Hash are far from offering an adequate protection. Let us explain the cause for the poor security effectiveness: while RND_T value is correctly chosen, an attacker mastering a corrupted node located "sufficiently near" from $hash(RND_T)$ can claim to be the Execution Setter and then select a list of actors including a maximum number of colluding nodes. Here, "sufficiently near" means that it satisfies the check made by the verifiers. Note that we tuned the system parameters such that we can be "sure" to have always a node sufficiently near of any random value to allow executing the actor selection protocol for any RND_T. The same problem arrives with M.Hash for each new random destination, thus explaining the poor security effectiveness. Hence, increasing the number of verifications or selecting each actor in a different network region does not solve the intrinsic limitation of these strategies. Note also that this behavior does not affect SEP2P. Indeed, even if the Execution Setter is a corrupted node, it cannot influence the actor list selection since it is done by k SLs (S only routes the messages between the SLs).

Setup costs: Figures 4 and 5 show the setup costs (Y axis in log scale) in terms of asymmetric crypto-operations and exchanged messages respectively, once more with respect to the verification cost (X axis). Curves with empty symbols represent latency while plain symbols represent total work. The results show that SEP2P is the slowest in latency and has the higher total setup cost for crypto-operations. These "bad" results are the consequence of two design choices: (1) to increase the security effectiveness, we

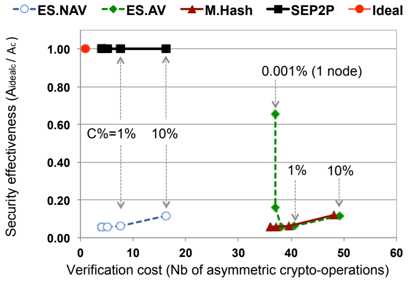


Figure 3: Sec. Effectiveness vs Verification

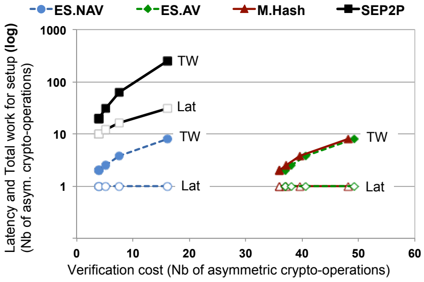


Figure 4: Setup asymmetric crypto-costs

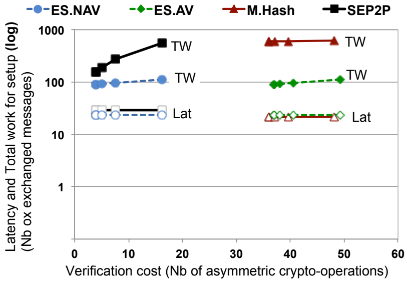


Figure 5: Setup communication costs

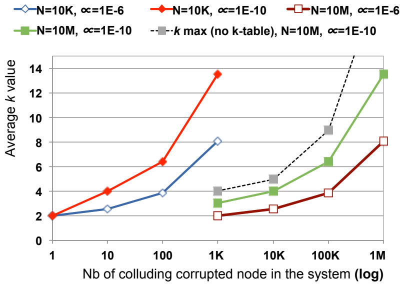


Figure 6: k versus C (N and $\propto$ vary)

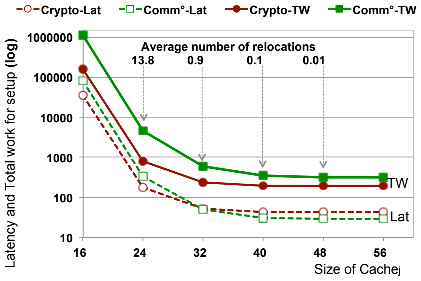


Figure 7: Setup costs varying R3 size

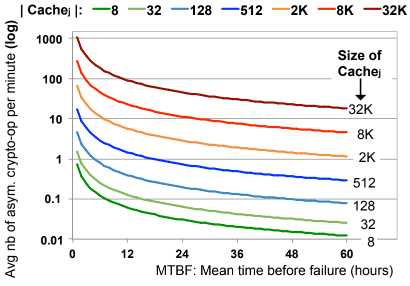


Figure 8: Maintenance overheads

run our protocol on k SL nodes thus increasing the total setup cost; and (2) we voluntarily make most of the checks during the setup (e.g., checking the actor certificates or verifying their availability) in order to reduce, as much as possible, the subsequent verification cost. Since this verification process will potentially be performed by a (very) large set of nodes (e.g., data sources), it is in our best interest to reduce it to avoid overloading the entire system. Figures 4 and 5 illustrate this aspect: our non-optimal setup cost is balanced by an optimized verification cost (and ideal disclosure in Figure 3). Note also that most operations are done in parallel (either by k TLs or SLs), thus leading to a reasonable setup latency (around 20 crypto-operations and 30 exchanged messages). We can also note in Figure 5 that M.Hash achieves the worst total work for setup (exchanged messages), because of the A routings in the DHT. Finally, we can remark the almost identical latency of ES.NAV, ES.AV and M.Hash on both metrics. Indeed, they all run the same initial protocol to compute RND_T. With respect to communication, the results are also identical because all DHT routings for M.Hash are done in parallel.

4.3 Scalability and Robustness

We now concentrate on SEP2P to study its scalability and its robustness to node failure.

Scalability: To study the scalability, we compute the averaged k value varying C and N. Indeed, k is the main factor in the verification cost, setup latency and total work (since everything is done k times). As seen in Section 3.6, depending on C and N, we can compute a k-table which gives several increasing values of k with increasing region size. We have considered small (10K) to very large (10M) networks and four values for $C\%$, leading to eight different SEP2P network configurations. For each configuration, we have computed, for each node, the minimal value for k with respect to the k-table and then averaged the results. Figure 6 shows the average k (Y axis) versus the $C\%$ (X Axis in log scale) for several network size considering two values for α: 10^{-6} and 10^{-10}. We also plot on the same figure the value of k without k-tables (the grey curve) to highlight the benefit brought by k-tables (only shown for the large network with $\alpha = 10^{-10}$). This figure offers many insights. (1) **SEP2P is highly scalable w.r.t. N:** Indeed, k values are identical for small and large networks independently of α if we consider the percentage of colluding nodes and not the absolute value (e.g., 1% colluding nodes is equivalent to absolute values of $C = 100$ and $C = 100K$ for the small and large networks). Indeed, scaling N and C in the same proportion leads to reduce $rs_1 = rs_2$ size accordingly. Note that with a single corrupted node, the k optimization is useless ($k = C + 1$ in that case) regardless of the α value. **(2) k increases slowly when $C\% < 1\%$:** k remains smaller than 6 even with $\alpha = 10^{-10}$. For $N = 10M$ and $C\% = 1\%$, the k-optimization reduces the number of participants in the verifiable random generation from $100K$ to 6. **(3) α has a small influences on k:** increasing α by four orders of magnitude increases k from 1 unit (e.g., $1K$ colluding nodes for $N = 10M$) to 5 units (e.g., $1K$ colluding nodes for $N = 10K$ or $1M$ colluding nodes for $N = 10M$). **(4) the k-table optimization is important:** k-tables allow reducing k by 1 unit up to 9 units (for 10% colluding nodes).

Number of actors: We also studied the impact of the variation of the number of actors. Overall, this results in a linear increase in the total work in terms of communications as the k SLs must check for the availability of A legitimate nodes to construct their respective CLs. For the sake of brevity, we omit here these results.

Node cache size: We now focus on adapting the node cache size to the maximum number of required actors. Our goal is to evaluate the impact of the cache size on the global performances. To do so we take a reference network with $N = 100K$, $C\% = 1\%$ and $A = 32$ and vary the average cache size on the whole network (we compute $rs3$ easily dividing the cache size by N). Figure 7 shows the results (Y axis in log-scale). For each cache size, we simulated an execution on each node of the network and computed the average values for our metrics. Our measures show that with a very small cache, the probability of relocating the actor selection process is high (the SLs do not find enough legitimate nodes in their cache w.r.t. R_3), which then leads to an increased latency and total work. Choosing a cache size greater than A, the query is almost never relocated (see Figure 7), giving better performances. This would lead to choose the largest possible cache. However, constructing such a cache also means maintaining it.

Maintenance costs: We also evaluated the impact of the cache size in the presence of node disconnections and, more generally, the impact of disconnections. To observe it, we simulated disconnections and measured their cost depending on the size of the node cache ($Cache_j$) using the default values for C, N, α and resulting k. We then considered those costs as a baseline and computed the global impact in a network where nodes disconnect (and reconnect) every x hours (mean time before failure or MTBF). We represent this cost in terms of asymmetric cryptographic operations (see Figure 8 - Y axis in log scale). The number of exchanged messages is not shown because graphs are very similar. We also computed these metrics for large node cache sizes (up to $32K$) to confirm that full mesh networks cannot be an alternative to DHT. Our results show that an overestimated cache is excessively costly even with an MTBF of 5 days: it consumes a large portion of the overall computing power of the entire system just to maintain it up to date. With small MTBFs, the network would be probably not maintainable. Since the number of actors for a computation is likely to be relatively small (e.g., few hundred, see Section 5), we can safely set the node cache size around 512 which leads to a reasonable maintenance cost (less than 1 signature per node per minute on average for MTBF = 1 day) and never trigger relocations (see Figure 7).

5 TASK ATOMICITY

5.1 Proposed Use Cases

We now focus on requirement 2, illustrating task atomicity on the use cases proposed in Section 1.

Use case 1: Mobile participatory sensing is used in many smart city applications for urban monitoring such as traffic monitoring (e.g., Waze or Navigon), evaluating the quality of road infrastructures, finding available parking spaces or noise mapping [36]. In these scenarios, the community members act as mobile probes and contribute to spatial aggregate statistics (density, averaged measures by location and time, spatial interpolation [36]) which in turn, benefit the whole community. As an alternative to the classical centralized architecture, the distributed PDMS paradigm increases the privacy guarantees for the users, thus encouraging their participation. A mobile user can generate sensing data (e.g., using her smartphone or vehicular systems) which is securely transmitted and recorded into her PDMS (e.g., a home box). This way each PDMS becomes a potential data source in the system. These data can then be aggregated by a small subset of data processor nodes to produce the required spatial aggregate statistics, which can be broadcasted to all the participating nodes.

Use case 2: Users can **subscribe to information flows based on their preference or user profile** (e.g., RSS feeds, specific product promotions or ads, etc.). A user profile can be represented by a set of concepts associating metadata terms (e.g., location, age, occupation, income, etc.) to values specific to each user. These associations are traditionally stored at a publication server to allow targeting the interested nodes. Instead, we propose to distributively store and index those profiles in SEP2P, thus greatly improving users' privacy. We call a concept index, an index associating for each concept the list of node addresses having this concept. Storing and searching this concept index is straightforward with a DHT. Each node does a $store(concept, IPaddress)$ for each concept in its profile. To find all the nodes matching a certain target profile (e.g., a logical expression of concepts), a DHT search is launched for each concept in the profile. Then, a set of randomly selected data processors are used to pick up the scattered pieces of the concept index, apply the logical expression of the target profile and compute the matching target nodes (*TN*), i.e., their IP addresses. Finally, the information is sent to the selected targets.

Use case 3: We consider **queries over the personal data contributed by a large set of individuals**, e.g., to compute recommendations, make participative studies. To achieve a high degree of pertinence and avoid flooding the system, such queries should target only a specific subset of the nodes, i.e., the nodes exposing a given user profile. Query examples are numerous, e.g., get the top-10 ranked movies by academics from Paris, or find the average number of sick leave days of pilots in their forties. The query processing is done in two steps which roughly correspond to the use case 2 combined with use case 1. First, the relevant subset of nodes, which match the query profile, must be discovered (use case 2). Then, the selected subset of target nodes become data sources which supply the required data (e.g., number of sick leave days) to compute the query result (use case 1). The main differences are that only the selected nodes provide data and that the result is transmitted only to the querier node and not to the entire system.

5.2 Detailed Node Roles

From the above description, we can define new node roles:

Node role 4. A **metadata indexer** (*MI*) stores part of the metadata shared by the nodes, allowing pertinent and efficient distributed data processing.

Node role 5. A **target finder** (*TF*), applies a logical expression on its input to produce a list of target nodes.

Node role 6. A **data aggregator** (*DA*) applies an aggregative function to its input and produces partially aggregated results.

Node role 7. A **main data aggregator** (*MDA*) aggregates its input and produces the final result.

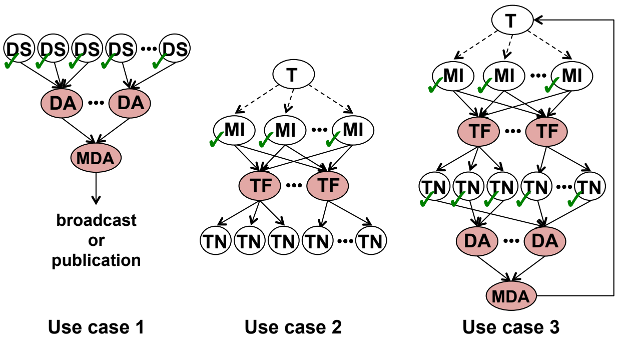


Figure 9: Distributed execution plans for the use cases

These roles allow designing distributed execution plans for the three use cases as shown in Figure 9. The nodes that must be chosen using the SEP2P protocol are shown in pink, and we used the symbol √ to denote that a node is a verifier (as specified in Section 3.6). This must be done each time a node discloses some sensitive data, thus on data sources and metadata indexers.

5.3 Towards Task Atomicity

The node roles and DEP proposed above already provide some task compartmentalization dividing the whole processing in tasks. However, much more can be done to minimize the impact of data leakage. In this section we present a few methods to achieve task atomicity. Our objective is mainly to show that task atomicity can be indeed performed and that it can significantly improve the system security when used in conjunction with the secure random actor selection. Given the space limitation, a detailed study of task atomicity is left for future work.

Metadata index protection: The concept index design already exhibits some form of task atomicity: (1) it is evenly distributed among all the nodes using the DHT mechanisms; (2) the imposed location of nodes in the DHT (see Section 3.2) leads to a randomized association between concepts and *MI* nodes. Nevertheless, a single corrupted node could disclose all the index information it owns. Further security improvements can be obtained by splitting each concept into s shares using the Shamir's secret sharing technique [32] which requires knowing at least p ($p \leq s$) shares to reconstruct the secret. Disclosing a single concept will now require p colluding nodes randomly selected.

User data protection: We consider here sensed data in use case 1 or the result of queries performed on a single PDMS in use case 2. Considering several *DA*s already reduces the impact of potential data leakage by a corrupted *DA* node. A simple way to reduce further this impact is to realize the aggregation on anonymized data (e.g., average traffic speed without user identity) or data without semantics (e.g., averaging data, a salary for instance, without knowing its meaning) or even encrypted data (with deterministic encryption). Note also that aggregation is continuous in the mobile sensing use case and that selected *DA* node will change at each iteration.

User identity protection: User's PDMS actively participate in the DEP either by receiving information (use case 2) or queries (use case 3) or by sending information (use cases 1 and 3). They thus communicate with *DA* nodes or receive messages from *TF* nodes, both being potentially corrupted. The reception / transmission task should be "isolated" to make one more step towards task atomicity. This can be achieved using the notion of proxy-forwarder that we illustrate for the *TN-DA* communication in the use case 3. The *TN* (which is actually a data source) must transmit its local result (e.g., number of sick leave days) to the *DA* node. *TN* can choose randomly any node *P* in the system and send the data, encrypted with the public key of the *DA* (known from the Verifiable Actor List). *P* will receive this data and transmit it to the *DA*. Thus, *DA* will have the data without knowing the sender, while *P* will know the sender but not the data. Note that (1) *TN* has good reasons to choose randomly *P* since it is the most interested in protecting its data; (2) the probability that both *DA* and *P* to be colluding nodes is extremely low ($\approx (C/N)^2$); and (3) we could use several proxies, thus mimicking anonymization network techniques (e.g., Tor).

6 RELATED WORK

DHT security. Several works focus on DHT security [40] considering the following attacks: (i) Sybil attack: an attacker generates numerous false DHT nodes to outnumber the honest nodes. Introducing an (offline) certificate authority, is deemed to be among the most effective defenses against the Sybil attack [11]. (ii) Routing table poisoning (eclipse attack): an attacker attempts to control most of the neighbors of honest nodes to isolate them. According to [40] the best strategy against such attacks is to constrain the DHT node identifiers. Again, using a central authority to provide verifiable identifiers is the simplest yet most effective way of achieving this goal [34]. (iii) Routing and storage attacks: Sybil and eclipse attacks do not directly impact the DHT, they are mainly necessary means for future attacks, like various denials of service (DoS). For instance, the objectives might be to prevent a lookup request from reaching its destination, denying the existence of a valid key, or impersonating another node to deliver false data. These DoS attacks are usually classified as routing and storage attacks and most of the mechanisms employed to negate them are based on redundancy at the storage and routing levels [40]. Thus, none of these works consider the secure and efficient actor selection for distributed processing as in SEP2P.

Secure Multi-party Computation and differential privacy. Cryptographic protocols have been proposed to protect the users' privacy in distributed computations with a focus on data confidentiality enforcement in personal data aggregation. Examples of computations related to this work are personal time-series clustering [2], kNN similarity queries [17], and location-based aggregate statistics [28]. However, MPC raises major scalability issues which in practice limit such protocols to specific types of computations [31].

Although it yields interesting results in privacy protection [15], differential privacy generally requires a central trusted aggregating node and ad-hoc adaptations depending on the targeted queries. As we search to provide a generic framework and exclude having a central actor to avoid a single point of failure, both requirements cannot be met by differential privacy. Even though local differential privacy [13] tries to address our first requirement, the solutions offered until now are still not generic, while the pertinence or the quality of the results may still be problematic with some applications [13]. Also, differential privacy exhibits intrinsic limitations with applications requiring continuous data flow aggregation (e.g., such as mobile participatory sensing) because of temporal correlation between consecutive data batches [10].

Distributed data aggregation using secure hardware. To overcome the limitations of MPC or differential privacy, several works propose using secure hardware at the user-side. Several secure protocols have been proposed for SQL aggregation [37], spatio-temporal aggregation [36], top-*k* full-text search [21], or privacy-preserving data publishing [3]. SEP2P also considers a secure PDMS at the user-side but our attack model considers having many colluding nodes. Moreover, the focus in SEP2P is on the secure and efficient random node selection. Differently, existing work focus on data aggregation or publishing and consider that all the nodes in the network participate in the protocol with their data being thus complementary to SEP2P.

Secure server-centric approaches. The above cited solutions are based on fully-distributed (P2P) or hybrid architectures. Alternatively, one could envision a solution based on a secured centralized server [6]. However, this raises important issues. First, users are exposed to sophisticated attacks, whose cost-benefit is high on a centralized database. Second, centralizing all users' data into one powerful server makes little sense in the PDMS context in which data is naturally distributed at the users' side. Hence, users might be reluctant to use such a massively centralized data service. Finally, new legislation such as the European GDPR [27] may hinder the development of such centralized solutions.

7 CONCLUSION

Personal Data Management Systems arrive at a rapid pace allowing users to share their personal data within large P2P communities. While the benefits are unquestionable, the important risks of private personal data leakage and misuse represent a major obstacle on the way of the massive adoption of such systems. This paper is one of the first efforts to deal with this important and challenging issue. To this end, we proposed SEP2P, a fully-distributed P2P system laying the foundation for secure, efficient and scalable execution of distributed computations. By considering a realistic threat model, we analyzed the fundamental security and efficiency requirements of such a distributed system. We showed that the secure selection of random actor nodes is the basis of security for any distributed computation. Then, we proposed secure and highly efficient protocols to address the actor selection problem. Our simulation-based experimental evaluation indicates that our protocol leads to minimal private information leakage, i.e., increasing linearly with the number of colluding nodes. At the same time, the cost of the security mechanisms depends only on the maximum number of colluding nodes and remains very low even with wide collusion attacks.

This work opens the way for several interesting research problems. In particular, to further minimize the impact of a private data leakage, the random actor selection needs complemented with task atomicity, i.e., decompose the computation process such that it minimizes the amount of sensitive data the processor nodes have access to. To underline this requirement, we discussed in this paper three types of representative applications in the PDMS context and provided sketches of solutions to achieve task atomicity. Certainly, this problem deserves a deeper look and constitutes our main objective as future work.

Acknowledgment. This research is partially supported by the ANR PersSoCloud grant ANR-16-CE39-0014.

REFERENCES

[1] Tristan Allard, Nicolas Anciaux, Luc Bouganim, Yanli Guo, Lionel Le Folgoc, Benjamin Nguyen, Philippe Pucheral, Indrajit Ray, Indrakshi Ray, and Shaoyi Yin. 2010. Secure personal data servers: a vision paper. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 25–35.

[2] Tristan Allard, Georges Hébrail, Florent Masseglia, and Esther Pacitti. 2015. Chiaroscuro: Transparency and privacy for massive personal time-series clustering. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 779–794.

[3] Tristan Allard, Benjamin Nguyen, and Philippe Pucheral. 2014. METAP: revisiting Privacy-Preserving Data Publishing using secure devices. *Distributed and Parallel Databases* 32, 2 (2014), 191–244.

[4] Nicolas Anciaux, Philippe Bonnet, Luc Bouganim, Benjamin Nguyen, Philippe Pucheral, Iulian Sandu Popa, and Guillaume Scerri. 2019. Personal Data Management Systems: The security and functionality standpoint. *Information Systems* 80 (2019), 13 – 35.

[5] Nicolas Anciaux, Luc Bouganim, Philippe Pucheral, Yanli Guo, Lionel Le Folgoc, and Shaoyi Yin. 2014. MILo-DB: a personal, secure and portable database machine. *Distributed and Parallel Databases* 32, 1 (2014), 37–63.

[6] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. 2013. Secure database-as-a-service with cipherbase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 1033–1036.

[7] Yonatan Aumann and Yehuda Lindell. 2007. Security against covert adversaries: Efficient protocols for realistic adversaries. In *Theory of Cryptography Conference*. Springer, 137–156.

[8] Michael Backes, Peter Druschel, Andreas Haeberlen, and Dominique Unruh. 2009. CSAR: A Practical and Provable Technique to Make Randomized Systems Accountable.. In *NDSS*, Vol. 9. 341–353.

[9] Blue Button. 2010. Find Your Health Data. (2010). Retrieved October 12, 2018 from https://www.healthit.gov/topic/health-it-initiatives/blue-button

[10] Yang Cao, Masatoshi Yoshikawa, Yonghui Xiao, and Li Xiong. 2017. Quantifying Differential Privacy under Temporal Correlations. In *33rd IEEE International Conference on Data Engineering, ICDE 2017*. 821–832.

[11] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S Wallach. 2002. Secure routing for structured peer-to-peer overlay networks. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 299–314.

[12] Cozy Cloud. 2013. Your digital home. (2013). Retrieved October 12, 2018 from https://cozy.io/en

[13] Graham Cormode, Somesh Jha, Tejas Kulkarni, Ninghui Li, Divesh Srivastava, and Tianhao Wang. 2018. Privacy at Scale: Local Differential Privacy in Practice. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018*. 1655–1658.

[14] Yves-Alexandre de Montjoye, Erez Shmueli, Samuel S Wang, and Alex Sandy Pentland. 2014. openpds: Protecting the privacy of metadata through safeanswers. *PloS one* 9, 7 (2014), e98790.

[15] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. 2017. Collecting Telemetry Data Privately. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*. 3574–3583.

[16] Fing. 2013. The mesinfos project explores the self data concept in france. (July 2013). Retrieved October 12, 2018 from http://mesinfos.fing.org/english

[17] Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Antoine Rault, François Taïani, and Jingjing Wang. 2015. Hide & Share: Landmark-based Similarity for Private KNN Computation. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE, 263–274.

[18] Javier González, Michael Hölzl, Peter Riedl, Philippe Bonnet, and René Mayrhofer. 2014. A practical hardware-assisted approach to customize trusted boot for mobile devices. In *International Conference on Information Security*. Springer, 542–554.

[19] Anne-Marie Kermarrec and François Taïani. 2015. Want to scale in centralized systems? Think P2P. *J. Internet Services and Applications* 6, 1 (2015), 16:1–16:12.

[20] Saliha Lallali, Nicolas Anciaux, Iulian Sandu Popa, and Philippe Pucheral. 2017. Supporting secure keyword search in the personal cloud. *Information Systems* 72 (2017), 1–26.

[21] Thi Bao Thu Le, Nicolas Anciaux, Sébastien Gilloton, Saliha Lallali, Philippe Pucheral, Iulian Sandu Popa, and Chao Chen. 2016. Distributed secure search in the personal cloud. In *19th International Conference on Extending Database Technology (EDBT 2016)*. 652–655.

[22] Sangmin Lee, Edmund L Wong, Deepak Goel, Mike Dahlin, and Vitaly Shmatikov. 2013. πBox: A Platform for Privacy-Preserving Apps.. In *NSDI*. 501–514.

[23] Petar Maymounkov and David Mazieres. 2002. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*. Springer, 53–65.

[24] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. 1996. *Handbook of Applied Cryptography*. CRC Press.

[25] MiData. 2011. The midata vision of consumer empowerment. (2011). Retrieved October 12, 2018 from https://www.gov.uk/government/news/the-midata-vision-of-consumer-empowerment

[26] Nextcloud. 2016. Protecting your data. (Jun 2016). Retrieved October 12, 2018 from https://nextcloud.com

[27] European Parliament. 2016. General Data Protection Regulation. Law. (27 April 2016). Retrieved October 12, 2018 from https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679

[28] Raluca Ada Popa, Andrew J Blumberg, Hari Balakrishnan, and Frank H Li. 2011. Privacy and accountability for location-based aggregate statistics. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 653–666.

[29] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A Secure Database using SGX. In *EnclaveDB: A Secure Database using SGX*. IEEE, 0.

[30] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. 2001. *A scalable content-addressable network*. Vol. 31. ACM.

[31] Eyad Saleh, Ahmad Alsa'deh, Ahmad Kayed, and Christoph Meinel. 2016. Processing over encrypted data: between theory and practice. *ACM SIGMOD Record* 45, 3 (2016), 5–16.

[32] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[33] Solid. 2018. Solid empowers users and organizations to separate their data from the applications that use it. (2018). Retrieved October 12, 2018 from https://solid.inrupt.com/

[34] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review* 31, 4 (2001), 149–160.

[35] ARM Security Technology. 2008. *Building a Secure System using TrustZone Technology*. Technical Report. ARM.

[36] Dai Hai Ton That, Iulian Sandu Popa, Karine Zeitouni, and Cristian Borcea. 2016. PAMPAS: Privacy-Aware Mobile Participatory Sensing Using Secure Probes. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*. ACM, 4.

[37] Quoc-Cuong To, Benjamin Nguyen, and Philippe Pucheral. 2016. Private and scalable execution of SQL aggregates on a secure decentralized architecture. *ACM Transactions on Database Systems (TODS)* 41, 3 (2016), 16.

[38] J. C. Tomàs, B. Amann, N. Travers, and D. Vodislav. 2011. RoSeS: a continuous query processor for large-scale RSS filtering and aggregation. In *Proc. of the 20th ACM Conf. on Information and Knowledge Management*. 2549–2552.

[39] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. 2011. A survey of DHT security techniques. *ACM Computing Surveys (CSUR)* 43, 2 (2011), 8.

[40] Qiyan Wang and Nikita Borisov. 2012. Octopus: A secure and anonymous DHT lookup. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*. IEEE, 325–334.

Indexing Trajectories for Travel-Time Histogram Retrieval

Robert Waury
Aalborg University
Aalborg, Denmark
rwaury@cs.aau.dk

Christian S. Jensen
Aalborg University
Aalborg, Denmark
csj@cs.aau.dk

Satoshi Koide
Nagoya University
Nagoya, Japan
koide@db.is.i.nagoya-u.ac.jp

Yoshiharu Ishikawa
Nagoya University
Nagoya, Japan
ishikawa@i.nagoya-u.ac.jp

Chuan Xiao
Nagoya University
Nagoya, Japan
chuanx@nagoya-u.jp

ABSTRACT

A key service in vehicular transportation is routing according to estimated travel times. With the availability of massive volumes of vehicle trajectory data, it has become increasingly feasible to estimate travel times, which are typically modeled as probability distributions in the form of histograms. An earlier study shows that use of a carefully selected, context-dependent subset of available trajectories when estimating a travel-time histogram along a user-specified path can significantly improve the accuracy of the estimates. This selection of trajectories cannot occur in a pre-processing step, but must occur online—it must be integrated into the routing itself. It is then a key challenge to be able to select very efficiently the "right" subset of trajectories that offer the best accuracy when the cost of a route is to be assessed. To address this challenge, we propose a solution that applies novel indexing to all available trajectories and that then is capable of selecting the most relevant trajectories and of computing a travel-time distribution based on these trajectories. Specifically, the solution utilizes an in-memory trajectory index and a greedy algorithm to identify and retrieve the relevant trajectories. The paper reports on an extensive empirical study with a large real-world GPS data set that offers insight into the accuracy and efficiency of the proposed solution. The study shows that the proposed online selection of trajectories can be performed efficiently and is able to provide highly accurate travel-time distributions.

1 INTRODUCTION

Vehicular transportation is an important global phenomenon that impacts the lives of virtually all of us. We rely on it for mobility, and we are affected by congestion, accidents, and air and noise pollution. Its influence can be expected to continue into the foreseeable future. For example, in the European Union alone, more than 75% of all freight transport and more than 80% of passenger transport rely on the road networks [8]. The availability of high-resolution GPS trajectories allows for reliable map-matching to a road network. The resulting trajectories are called *network-constrained trajectories* (NCT) and can be used to obtain travel-time estimates for paths in the network, thus making transportation more predictable, safe, and environmentally friendly.

When using such a data set, the most straightforward approach to computing a travel-time estimate for a path is to compute a real-valued estimate for each segment in the path and then sum up these to obtain an estimate for the full path. This approach can be refined by collecting travel-time histograms for each segment and then combine them by means of convolution to obtain a travel-time histogram for the full path. This improves the accuracy of estimates since travel times are better modeled as distributions than real valued. Further, the distributions often do not follow a parameterized distribution, e.g., normal or uniform, and are therefore better estimated with histograms. This segment level approach can also be extended to computing different histograms for different times of day, e.g., the 96 15-minute intervals of the day, to account for changing congestion throughout the day. These histograms can be used as edge weights by routing algorithms to compute better results. All of the above approaches, however, only consider travel-time estimates at the segment level. These approaches fail to take into account factors like the times it takes to pass through intersections, going straight or turning left or right, which are hard to model accurately. An earlier study [26] shows that travel-time estimates for a given path can be improved considerably when they are computed from trajectories that strictly follow the path, as opposed to computing them from segment-level estimates. This type of path-based estimate relies on efficiently processing *strict path queries* (SPQ) as proposed by Krogh et al. [14], which is a query on a trajectory set that only returns trajectories which traversed a given path without detours.

We propose a system that can compute time-varying and personal travel-time histograms for any path in a network based on a large trajectory set. It would be infeasible and impractical to pre-compute and store these time-varying and personal weights for any path in a network before routing occurs. For example, given even a moderately sized road network of a million segments, for all 15-minute windows, nearly a 100 million histograms would be needed to just cover every single segment, with the storage requirements increasing dramatically when considering larger path lengths. We therefore obtain the weights for a path on-the-fly by expressing them as a series of SPQs, which we can efficiently process using our in-memory NCT index. If any of these sub-queries fails to retrieve a sufficient number of matching trajectories, we apply a greedy algorithm that relaxes the SPQ's predicates until the retrieved trajectory set has a specified cardinality. Since performance is crucial in our setting, we also implement a cardinality estimator for SPQs to prevent unnecessary index traversals. We also show that carefully choosing the initial set of SPQs increases the accuracy of the path weights and increases the performance of the query. We perform extensive experiments using a real-world trajectory data set containing 1.4 million trajectories from Northern Denmark, which shows that our approach is suitable for real-time applications.

Series ISSN: 2367-2005 10.5441/002/edbt.2019.15

The main contributions of this paper are the following:

- An adapted NCT index that supports efficient computation of travel-time histograms for SPQs.
- A greedy algorithm that enables efficient processing of any SPQ in periodic time intervals.
- A cardinality estimator for SPQs.
- A detailed analysis of the accuracy and performance of the solution and its components.

The rest of the paper is structured as follows. Section 2 provides an overview of prior work, preliminaries, and a detailed problem description. Section 3 describes the query processing method, while Section 4 details the construction and use of the NCT index. Section 5 outlines the experimental setup and the evaluation metrics. Section 6 reports on the results of the experiments, and Section 7 concludes.

2 PROBLEM FORMULATION

This section provides an overview of prior work, preliminaries, and a problem definition.

2.1 Related Work

We review approaches to travel-time estimation and then, we review network-constrained trajectory indexing with a focus on indexes supporting SPQs.

2.1.1 Travel-Time Estimation. Earlier studies on travel-time estimation compute histograms for single segments [15], which still requires to model turn costs [27], or for short pre-defined paths with considerable traffic [4], which are then convolved at query time. In our approach, travel-times are computed for sub-paths instead of only for individual segments. This approach implicitly handles turn costs within sub-paths, and turn costs only need to be modeled explicitly in-between sub-paths if applicable. Other approaches based on tensor decomposition [25], support vector regression [28], variance-entropy-based clustering [29], or deep neural networks [24] have also been proposed. But they either do not provide travel-time distributions or do not provide estimates for specific paths but only for origin-destination pairs.

2.1.2 NCT Indexing. Several indexes for network-constrained trajectories based on R-trees [5, 6, 9] or B+-trees [20] have been proposed, but they are often only optimized for range queries or nearest neighbor queries.

Two indexes have been proposed to support strict path queries, NETTRA [14] and the SNT-index [12]. NETTRA is a disk-based index designed to answer SPQs with minimal I/O and also supports efficient updates of the index, but may return false positives due to hash collisions. The SNT-index uses data structures adapted from string matching to efficiently identify matching trajectories. This index was originally designed to retrieve all matching trajectory IDs in a given time interval that fulfill the SPQ requirement. We extend it to accommodate travel-time retrieval as well.

2.2 Network Graph & Trajectories

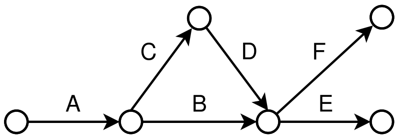


Figure 1: Example Road Network

Table 1: Example of $\mathcal{F}$ and Function *estimateTT*

e	*c*	*z*	*sl*	*l*	*estimateTT*
A	*motorway*	*rural*	110	900	29.5 s
B	*primary*	*city*	50	120	8.6 s
C	*secondary*	*city*	30	40	4.8 s
D	*secondary*	*city*	30	80	9.6 s
E	*primary*	*city*	50	100	7.2 s
F	*primary*	*rural*	80	800	36.0 s

A spatial network is modeled as a directed graph $G = (V, E, \mathcal{F})$, where V is a vertex set, $E \subseteq V \times V$ is a set of edges that represent road segments, and $\mathcal{F} : E \rightarrow Cat \times Z \times SL \times L$ is a set of functions, where Cat is the set of road categories, Z is the set of different types of zones the segments are located in, SL is the set of speed limits in kilometers per hour (or $\frac{1000}{3600}$ meters per second), and L is the set of segment lengths in meters. From this we can derive the function $estimateTT(e_i) = 3.6 \frac{\mathcal{F}(e_i).l}{\mathcal{F}(e_i).sl}$ that returns the traversal time in seconds if the segment is traversed at the speed limit. This function is used as a fallback so that we can return a result even if no data is available for a segment. Every edge $e \in E$ has a category that captures the road type of the segment it represents and a zone type describing its location. Figure 1 shows the graph representation of the road network we use in examples. Table 1 shows the mapping of each segment to categories $c \in Cat = \{motorway, primary, secondary\}$ and zones $z \in Z = \{city, rural\}$.

A traversable sequence of segments $P = \langle e_0, e_1, \ldots, e_{l-1} \rangle$ is called a path, with $|P| = l$. A sub-path $\langle e_i, \ldots, e_{j-1} \rangle$, with $0 \leq i < j \leq l$, of P is denoted as $P[i, j)$. The set of trajectories is given as $\mathcal{T} \subseteq \mathcal{D} \times \mathcal{U} \times \mathcal{S}$, where $\mathcal{D}$ is the set of all trajectory ids, $\mathcal{U}$ is the set of all drivers. Further, $\mathcal{S} : \mathbb{N}_l \rightarrow E \times \mathcal{TS} \times \mathcal{C}$ is the domain of functions from the set consisting of the first l natural numbers to the range of triples consisting of an edge $e \in E$, a timestamp $t \in \mathcal{TS}$, and a time duration $TT \in \mathcal{C}$. This domain of functions encodes finite sequences of length l.

A trajectory $tr \in \mathcal{T}$ of a user u with the id d is therefore denoted as (d, u, s), where $s \in \mathcal{S}$ is a sequence of 3-tuples:

$$s = \langle (e_0, t_0, TT_0), (e_1, t_1, TT_1), \ldots, (e_{l-1}, t_{l-1}, TT_{l-1}) \rangle,$$

where $t_0, .., t_{l-1}$ are the timestamps when a segment was entered with $\forall i \forall j (i < j \Rightarrow t_i < t_j)$, $TT_i > 0$ is the duration of the traversal of e_i, and l is the number of segments traversed.

The path of trajectory tr is called P_{tr}, and its starting time is $tr.t_0$. The duration function $Dur(tr, P) = TT_0 + TT_1 + \ldots + TT_{l-1}$ returns the sum of all segment traversal times a_{tr}^P of a path P by a trajectory. If a trajectory path P_{tr} does not contain P as a sub-path , $Dur(tr, P)$ is undefined. A trajectory set in our example road network from Figure 1 is shown below:

$$tr_0 : (0, u_1) \rightarrow \langle (A, 0, 3), (B, 3, 4), (E, 7, 4) \rangle$$
$$tr_1 : (1, u_2) \rightarrow \langle (A, 2, 4), (C, 6, 2), (D, 8, 4), (E, 12, 5) \rangle$$
$$tr_2 : (2, u_2) \rightarrow \langle (A, 4, 3), (B, 7, 3), (F, 10, 6) \rangle$$
$$tr_3 : (3, u_1) \rightarrow \langle (A, 6, 3), (B, 9, 3), (E, 12, 4) \rangle$$

2.3 Travel-Time Query

To address the shortcomings of the segment-level approach, we employ the strict path query $Q = spq(P, I, f, \beta)$ that returns a travel-time histogram H. The histogram can be derived from the traversal times of the set of trajectories $\mathcal{T}^P \subseteq \mathcal{T}$ that traverse path P without stops or detours in the time interval I, and fulfill additional filter predicates f:

$$\mathcal{T}^P = \{tr \in \mathcal{T} | \exists i, j \, (P_{tr}[i, j) = P \land tr.s.t_i \in I \land f(tr))\},$$

where $I = [t_s, t_e)$ denotes a temporal predicate with a size $\alpha = t_e - t_s$ and β is a cardinality requirement for $\mathcal{T}^P$, i.e., we only proceed if $|\mathcal{T}^P| \geq \beta$. If β is omitted all eligible trajectories are retrieved. The temporal predicate can either cover a fixed time interval, e.g., all trajectories from December 1st 2017 until May 1st 2018, or a periodic time-of-day interval denoted as $I^R = \langle \ldots, \left[t_s - 24\ hours, t_e - 24\ hours\right), \left[t_s, t_e\right), \left[t_s + 24\ hours, t_e + 24\ hours\right), \ldots, \left[t_s + n\ (24\ \ hours), t_e + n\ (24\ \ hours)\right)\rangle$, e.g., all trajectories from 8:00 until 8:30 on every day. Parameter f is an additional non-temporal filter predicate that trajectories in $\mathcal{T}^P$ have to fulfill, e.g., being from a specified driver.

Using such a query Q for a typical trip path, which can consist of dozens of segments, may not return a sufficient number of trajectories to derive accurate travel-time estimations. To address this problem, we split Q into k sub-queries $\langle Q_1, Q_2, \ldots, Q_k \rangle = \langle spq(P_1, I_1, f_1, \beta), spq(P_2, I_2, f_2, \beta), \ldots, spq(P_k, I_k, f_k, \beta) \rangle$ that return the trajectory sets $\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k\}$, where P_i are sub-paths that partition P. These can then be used to compute a set of k histograms $\{H_1, H_2, \ldots, H_k\}$ if $\forall i \, |\mathcal{T}_i| \geq \beta$. Their convolution we call $H = H_1 * H_2 * \ldots * H_k$, where $*$ is the discrete convolution operator and H is a travel-time histogram that covers the full path P. The intuition behind partitioning into k sub-queries is, that different sub-paths often provide better estimates with different predicates, e.g., user predicates mainly improve accuracy outside of cities [26]. Another advantage of partitioning the query is the increased number of eligible trajectories.

How this partitioning into sub-queries is performed and how the sub-queries are processed is discussed in Sections 3 and 4.

An example query for our example trajectory set could be $Q = spq(\langle A, B, E \rangle, [0, 15), u = u_1, 2)$. This would return $\mathcal{T}^P = \{tr_0, tr_3\}$ yielding a histogram with $H = \{[10, 11)\colon 1; [11, 12)\colon 1\}$ since $Dur(tr_0, \langle A, B, E \rangle) = 11$ and $Dur(tr_3, \langle A, B, E \rangle) = 10$. But if a larger cardinality is required, Q could be split into two queries $Q_1 = spq(\langle A, B \rangle, [0, 15), \emptyset, 3)$ and $Q_2 = spq(\langle E \rangle, [0, 15), \emptyset, 3)$ that yield the histograms $H_1 = \{[6, 7)\colon 2; [7, 8)\colon 1\}$ and $H_2 = \{[4, 5)\colon 2; [5, 6)\colon 1\}$, from which the convolution $H = \{[10, 11)\colon 4; [11, 12)\colon 4; [12, 13)\colon 1\}$ can be obtained.

3 QUERY PROCESSING

This section describes the architecture of the system, the processing of travel-time queries, and the greedy algorithm used for relaxing sub-query predicates.

3.1 Architecture

Figure 2 shows the overall system architecture, where boxes with dotted lines indicate pre-existing components, dashed lines indicate modified components, and solid lines indicate new components. At first, a GPS data set is map-matched off-line to trajectories and loaded into the modified SNT-index consisting of a collection of temporal indexes and a spatial index.

Once the trajectory set is loaded, a user is able to dispatch a strict path query Q to the `Sub-query Module` where the query is initially partitioned into k sub-queries by the `Query Partitioner` according to a simple heuristic called π, e.g., sub-paths of a fixed length, or sub-paths that have the same segment category. Each sub-query is then assigned temporal and trajectory filter predicates. Next, the `Cardinality Estimator` uses the `Histogram Store` and the `SNT-Index` to estimate the cardinality $\hat{\beta}$ of the trajectory set $\mathcal{T}_i$ returned by the sub-query $spq(P_i, I_i, f_i, \beta)$. If $\hat{\beta}$ is smaller than the desired cardinality β, the sub-query is modified by the `Sub-query Splitter` using a splitting function σ that relaxes the predicates. If the sub-query's cardinality estimate meets the requirement, it is dispatched to the index, and a trajectory set $\mathcal{T}_i$ is obtained. If $|\mathcal{T}_i| \geq \beta$, it is forwarded to the `Histogram Builder`. If the cardinality is still below the threshold, it is modified again by the `Sub-query Splitter`.

Once all trajectory sets $\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k\}$ are obtained, their travel-time sets $\{X_1, X_2, \ldots, X_k\}$ are extracted, with $X_i = \{Dur(tr, P_i) | tr \in \mathcal{T}_i\}$. From those, a set of histograms $\mathcal{H} = \{H_1, H_2, \ldots, H_k\}$ is computed, and they are convolved into a single histogram $H = H_1 * H_2 * \ldots * H_k$ that estimates the travel-time distribution for the complete path P.

3.2 Partitioning Methods

For the initial partitioning of queries, we propose five different methods. We use the query $Q = spq(P, I, f, \beta)$ with path $P = \langle A, C, D, E \rangle$ from the network in Figure 1 as example. The initial periodic time interval I_i^R is identical for all sub-queries and is always chosen so that $t_e - t_s = \alpha_{min}$, where α_{min} is the minimum time interval size, which is chosen by the system. The predicate f is also initially identical for all sub-queries but may be modified by the splitting method (cf. Section 3.3).

3.2.1 Regular (π_p). The regular partitioning creates sub-queries for paths of length p, i.e., every query is partitioned into $k = \lceil \frac{l}{p} \rceil$ sub-queries, i.e., the sub-queries $\pi_p(Q) = \langle spq(P[0, p), I_1^R, f_1, \beta), spq(P[p, 2p), I_2^R, f_2, \beta), \ldots, spq(P[p\lfloor \frac{l}{p} \rfloor, l), I_k^R, f_k, \beta) \rangle$ are created. In our experiments we chose π_1, π_2 and π_3, which for our example path yield the paths $\langle \langle A \rangle, \langle C \rangle, \langle D \rangle, \langle E \rangle \rangle$, $\langle \langle A, C \rangle, \langle D, E \rangle \rangle$, and $\langle \langle A, C, D \rangle, \langle E \rangle \rangle$, respectively.

3.2.2 Segment Category (π_C). The segment type partitioning creates partitions of sub-paths with identical segment categories, i.e., two neighboring segments e_i and e_{i+1} are split unless $\mathcal{F}(e_i).c = \mathcal{F}(e_{i+1}).c$. For our example query, this results in the sub-paths $\langle \langle A \rangle, \langle C, D \rangle, \langle E \rangle \rangle$.

3.2.3 Zone Type (π_Z). The zone type partitioning creates partitions of sub-paths within the same zone type, i.e., two neighbouring segments e_i and e_{i+1} are split unless $\mathcal{F}(e_i).z = \mathcal{F}(e_{i+1}).z$. For our example query, this results in the sub-paths $\langle \langle A \rangle, \langle C, D, E \rangle \rangle$.

3.2.4 Zone Type & Segment Category (π_{ZC}). The zone type and segment category partitioning creates partitions of sub-paths within the same zone type and segment category combination, i.e., two neighboring segments e_i and e_{i+1} are split unless $\mathcal{F}(e_i).z = \mathcal{F}(e_{i+1}).z \land \mathcal{F}(e_i).c = \mathcal{F}(e_{i+1}).c$. For our example query, this results in the sub-paths $\langle \langle A \rangle, \langle C, D \rangle, \langle E \rangle \rangle$.

3.2.5 None (π_N). No initial partitioning is attempted, and the query is processed according to one of the splitting strategies described below. For our example query, this results in the single sub-path $\langle \langle A, C, D, E \rangle \rangle$.

3.3 Splitting Methods

If a sub-query $spq(P, I, f, \beta)$ does not return the desired cardinality, it is modified by a splitting function σ described in Procedure 1 that takes a query and the list of time interval sizes $A = \langle \alpha_1, \ldots, \alpha_n \rangle$, with $\forall i \forall j \, (i < j \Rightarrow \alpha_i < \alpha_j)$, and $\alpha_1 = \alpha_{min}$ and $\alpha_n = \alpha_{max}$ as arguments.

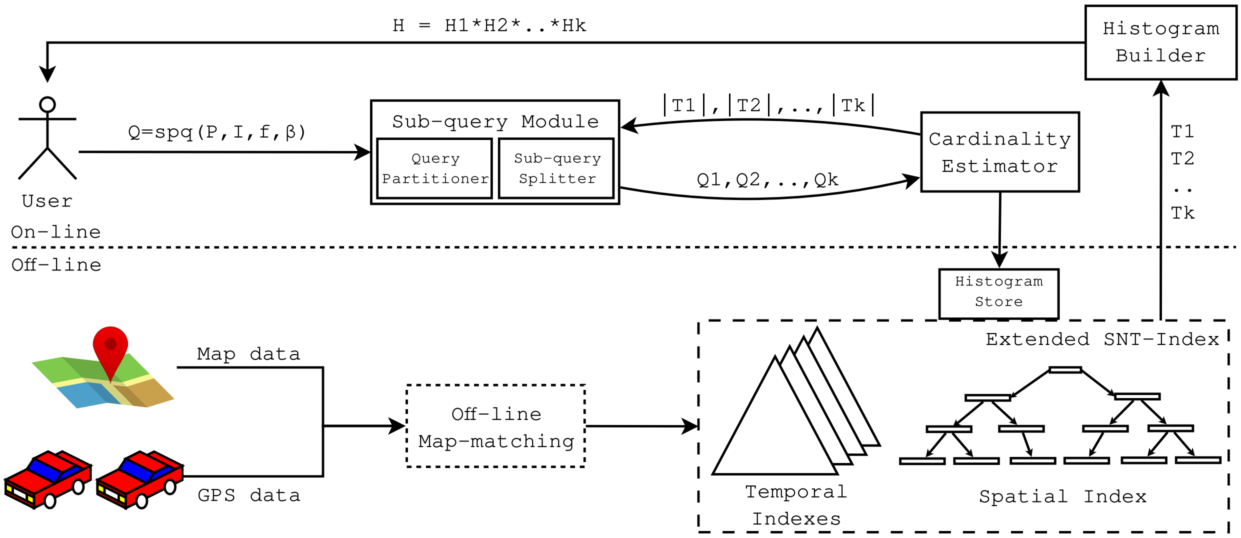


Figure 2: Overall Architecture

At first the procedure tries to increase the sample size by increasing the size of the time interval for the path by choosing the next largest size from the list A and widening the periodic interval with $widen([t_s, t_e)^R, \alpha_{i+1}) = [t_s - \frac{\alpha_{i+1}-\alpha_i}{2}, t_e + \frac{\alpha_{i+1}-\alpha_i}{2})^R$. After A has been exhausted, the path is split, and two new sub-queries with the smallest allowed time interval size α_{min} are created.

We propose two types of splitting and again use the path $P = \langle A, C, D, E\rangle$ in examples.

- σ_R Regular splitting cuts the path in half, i.e., $P_1 = P[0, \lfloor \frac{l}{2} \rfloor)$ and $P_2 = P[\lfloor \frac{l}{2} \rfloor, l)$, so splitting the example path P results in $P_1 = \langle A, C\rangle$ and $P_2 = \langle D, E\rangle$.
- σ_L Longest prefix splitting creates two sub-paths $P_1 = P[0, m)$ and $P_2 = P[m, l)$, with $1 \leq m < l$, where the maximum value for m for which $|\mathcal{T}^{P_1}| \geq \beta$ holds is chosen.

If a sub-path cannot be split further, any non-temporal filter predicates are dropped (Line 10). As a fallback, all temporal filters and the β parameter are dropped as well, i.e., for a single segment, all available trajectories are considered in the fixed time interval $[0, t_{max})$ (Line 12).

Procedure 1 Modify a sub-query *spq* to increase sample size (σ):

```
Input: Sub-query $spq(P, I, f, \beta)$, time interval sizes $A$
Output: a sequence of sub-queries $\langle Q_1, \ldots, Q_k\rangle$
1: $\alpha_i \leftarrow t_e - t_s$
2: if $\alpha_i < \alpha_{max}$ then
3:     $I'^R \leftarrow widen(I^R, \alpha_{i+1})$
4:     return $\langle spq(P, I'^R, f, \beta)\rangle$
5: else if $|P| > 1$ then
6:     $m \leftarrow split(P)$
7:     $I'^R \leftarrow shrink(I^R, \alpha_{min})$
8:     return $\langle spq(P[0, m), I'^R, f, \beta), spq(P[m, l), I'^R, f, \beta)\rangle$
9: else if $f \neq \emptyset$ then
10:    return $\langle spq(P, I^R, \emptyset, \beta)\rangle$
11: else
12:    return $\langle spq(P, [0, t_{max}), \emptyset)\rangle$
13: end if
```

4 THE INDEX

This section describes the SNT-index and how we adapt and optimize it to support travel-time queries using an example trajectory set.

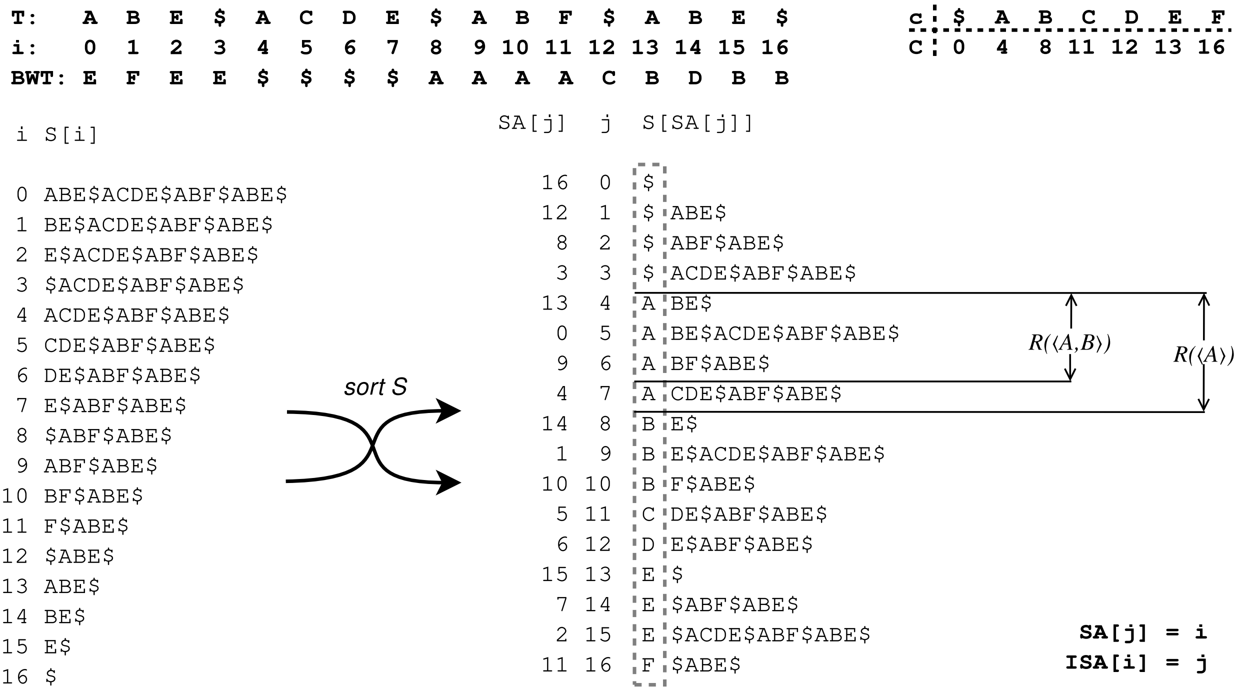
4.1 SNT-Index

Koide et al. [12] proposed the SNT-index for strict path queries using the FM-index as a spatial index and a forest of B+-trees as a temporal index. The advantage of the FM-index over R-tree-based methods is that by representing the trajectory set $\mathcal{T}$ as a string T and adapting a method from substring matching, evaluating spatial queries is only dependent on the size of the spatial network ($|E|$) and not on the size of the trajectory set ($|\mathcal{T}|$). In addition, it can be established from just the FM-index whether a given path is traversed at all, often saving a costly temporal index traversal. While the original index returns a set of trajectory ids given the query $spq(P, I)$, where P is the path and I is a time interval, our index returns the traversal times of the trajectories for P, which can be stored in a histogram. Sections 4.1.1 and 4.1.2 recap the previously described SNT-index and the remaining section describes our modifications to it to facilitate the efficient retrieval of travel-times.

4.1.1 The Spatial FM-Index. For our example we are indexing the trajectory set $\mathcal{T} = \{tr_0, tr_1, tr_2, tr_3\}$ introduced earlier.

To index the trajectories, we first need to compute the trajectory string T from the alphabet $\Sigma = E \cup \{\$\}$ where the symbol \$ denotes the end of a trajectory and where $\forall e \in E\, (e > \$)$ and $T = P_{tr_0}\$P_{tr_1}\$\ldots\$P_{tr_{n-1}}\$, \forall tr \in \mathcal{T}$. With our example trajectory set, this yields the trajectory string $T = ABE\$ACDE\$ABF\$ABE\$$.

From this trajectory string, we compute an array S of all suffixes of T, where $S[i] = T[i, n)$, where $0 \leq i < n = |T|$. These suffixes are then sorted lexicographically to obtain the *suffix array SA* as shown in Figure 3, where $SA[j]$ contains the index of the j-th smallest suffix. From SA, we can then compute the *inverse suffix array ISA* where $SA[j] = i$ and $ISA[i] = j$ [17]. Every substring (or in our case, subpath) P of length l therefore has a range of ISA values $R(P) = [st, ed)$ that is defined as $R(P) = \{i \mid S[SA[i]]\,[0, l) = P\}$, e.g., the *ISA* range of the path $\langle A\rangle$ is $R(\langle A\rangle) = [4, 8)$ since four trajectories contain A and they



T:	A	B	E	$	A	C	D	E	$	A	B	F	$	A	B	E	$
i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
BWT:	E	F	E	E	$	$	$	$	A	A	A	A	C	B	D	B	B

c	$	A	B	C	D	E	F
C	0	4	8	11	12	13	16

i	S[i]
0	ABE$ACDE$ABF$ABE$
1	BE$ACDE$ABF$ABE$
2	E$ACDE$ABF$ABE$
3	$ACDE$ABF$ABE$
4	ACDE$ABF$ABE$
5	CDE$ABF$ABE$
6	DE$ABF$ABE$
7	E$ABF$ABE$
8	$ABF$ABE$
9	ABF$ABE$
10	BF$ABE$
11	FABE
12	ABE
13	ABE$
14	BE$
15	E$
16	$

SA[j]	j	S[SA[j]]	
16	0	$	
12	1	$	ABE$
8	2	$	ABF$ABE$
3	3	$	ACDE$ABF$ABE$
13	4	A	BE$
0	5	A	BE$ACDE$ABF$ABE$
9	6	A	BF$ABE$
4	7	A	CDE$ABF$ABE$
14	8	B	E$
1	9	B	E$ACDE$ABF$ABE$
10	10	B	FABE
5	11	C	DE$ABF$ABE$
6	12	D	E$ABF$ABE$
15	13	E	$
7	14	E	$ABF$ABE$
2	15	E	$ACDE$ABF$ABE$
11	16	F	ABE

SA[j] = i
ISA[i] = j

Figure 3: The Suffix Array and Burrows-Wheeler-Transform

Procedure 2 Calculate ISA range $[st, ed)$ for a path P of length l (*getISARange*):

```
Input: Burrows-Wheeler transform $T_{bwt}$ of the trajectory string
    $T$, symbol counts $C$, path $P = p_0 \ldots p_{l-1}$
Output: ISA range $[st, ed)$ that matches $P$
 1: $c \leftarrow p_{l-1}$
 2: $st \leftarrow C[c]$
 3: $ed \leftarrow C[c+1]$
 4: for $i \leftarrow 2$ to $l$ do
 5:     $c \leftarrow p_{l-i}$
 6:     $st \leftarrow C[c] + rank_c(T_{bwt}, st)$
 7:     $ed \leftarrow C[c] + rank_c(T_{bwt}, ed)$
 8:     if $st \geq ed$ then
 9:         return $[0, 0)$
10:     end if
11: end for
12: return $[st, ed)$
```

appear at the start of the suffixes $S[SA[st]]$ to $S[SA[ed-1]]$, and the range for $R(\langle A, B \rangle)$ is $[4, 7)$ as only three trajectories traverse this path.

The ISA range of a path can be obtained efficiently from two data structures that comprise the FM-index:

- C an array that stores the number of lexicographically smaller characters in the trajectory string for every member of the alphabet Σ, e.g., $C['B'] = 8$ since there exist 8 characters in T that are lexicographically before $'B'$.
- T_{bwt} the Burrows-Wheeler transform [3] of the trajectory string T that is defined as $T_{bwt}[i] = T[SA[i]-1]$, with $0 \leq i < |T|$. For our example, this yields the string *EFEE$$$$AAAA–CBDBB*.

We define the rank operation $rank_c(T_{bwt}, i)$ that counts the occurrences of the character c in $T_{bwt}[0, i)$. As an example of computing the ISA range, we compute the range for the path $P = \langle A, B \rangle$ as described in Procedure 2. At first, the segment $c \leftarrow B$ is set in Line 1, and $st \leftarrow 8$ and $ed \leftarrow 11$ are initialized in Lines 2 and 3. For the first (and in this case the only) iteration of the loop from Line 4 to 11, $c \leftarrow A$, $st \leftarrow 4 + rank_A(T_{bwt}, 8)$, and $ed \leftarrow 4 + rank_A(T_{bwt}, 11)$, which yields the ISA range $[4, 7)$, since the ranks are 0 and 3, respectively.

The Burrows-Wheeler transform is stored in a wavelet tree to enable rank queries in $O(log|\Sigma|)$ time [10]. Therefore, obtaining the ISA range $[st, ed)$ of any path P can be performed in $O(|P|\ log\ |\Sigma|)$ time, which does not depend on the size of T.

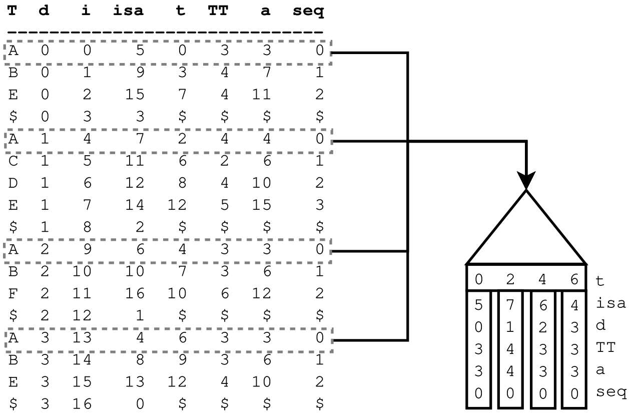
4.1.2 The Temporal Indexes. The temporal indexes $F = \{\Phi_e | e \in E\}$ contain a B+-tree for every segment in the network. Each tree indexes the records $r \in \Phi$ by the timestamp t when a trajectory entered the segment. A leaf node entry r for a timestamp t contains the ISA index (*isa*) and the trajectory identifier (d).

The original SNT-index is only capable of retrieving the trajectory ids, which would then have to be processed in turn to obtain the traversal times of the query path.

4.1.3 Extensions to the SNT-Index. To support travel-time histogram construction directly using the SNT-index, we add the following information to each leaf node in a temporal index:

- The traversal time TT of the segment in seconds.
- The sequence number *seq* of the segment in the trajectory.
- The sum of the travel-times $a_{seq} = \sum_{i=0}^{seq} TT_i$ from the start of the trajectory and up to and including the segment.

Figure 4 shows the contents of the temporal index Φ_A of segment A for our example trajectory set where each leaf is a record r, mapping a timestamp t to a tuple (isa, d, TT, a, seq). Furthermore, we add an associative container U that maps every trajectory id d to its respective user id u to check the filter predicate f. With those fields, we can build a hash table during the scan of the index of the first segment with the trajectory id and sequence number as the key (d, seq) and the aggregate of the preceding segment of the trajectory $(a_0 - TT_0)$, as value as described in Procedure 3. The sequence number is included to guard against



T	d	i	isa	t	TT	a	seq
A	0	0	5	0	3	3	0
B	0	1	9	3	4	7	1
E	0	2	15	7	4	11	2
$	0	3	3	$	$	$	$
A	1	4	7	2	4	4	0
C	1	5	11	6	2	6	1
D	1	6	12	8	4	10	2
E	1	7	14	12	5	15	3
$	1	8	2	$	$	$	$
A	2	9	6	4	3	3	0
B	2	10	10	7	3	6	1
F	2	11	16	10	6	12	2
$	2	12	1	$	$	$	$
A	3	13	4	6	3	3	0
B	3	14	8	9	3	6	1
E	3	15	13	12	4	10	2
$	3	16	0	$	$	$	$

0	2	4	6	t
5	7	6	4	isa
0	1	2	3	d
3	4	3	3	TT
3	4	3	3	a
0	0	0	0	seq

Figure 4: Extended Temporal Index

trajectories with circular paths. The spatial filtering is performed with the ISA range $[st, ed)$ obtained from Procedure 2 during the index scan in Line 3. The filter predicate f can be evaluated in constant time with the associative container U. The cardinality parameter β is used to reduce the processing time since not all eligible trajectories are necessary to obtain a good estimate, and the *buildMap* procedure terminates as soon as β trajectories are found (Line 6). When scanning the temporal index of the last segment in the query, we can obtain the traversal time of the query path by $a_{l-1} - (a_0 - TT_0)$ as described in Procedure 4.

4.2 Travel-Time Query

When used together, the previous three procedures make it possible to obtain the set of travel times for any path, as shown in Procedure 5, to answer the sub-query $spq(P, I, f, \beta)$. To obtain all trajectories that traversed a path P during a given time interval I, an ISA range is first obtained from the FM-index in Line 1. If a non-empty range is returned, a range scan on the index of the first (Line 6) and last segment (Line 11) of the path are performed for I and filtered by the ISA index in the leafs. If no matching trajectories exist or no periodic time interval with more than β trajectories is found (Line 7) the query returns the empty set. If the sub-query provided by Procedure 1 has a fixed time interval, the query is processed regardless of β. If that still yields no trajectories, an estimate based on the speed limit of the segment is provided (Line 13).

Procedure 6 shows how a full query is partitioned and processed. For longer trips, the periodic interval I_i^R is adapted with the shift-and-enlarge procedure (Line 4) suggested by Dai et al. [4], that shifts the beginning of the interval t_s by the sum of all previous minimums $S_i = \sum_{j=1}^{j=i-1} H_j^{min}$ and enlarges it by the sum of all previous ranges $R_i = \sum_{j=1}^{j=i-1}(H_j^{max} - H_j^{min})$.

4.3 Optimizations

4.3.1 CSS-Trees. The cache sensitive search tree (CSS-tree) proposed by Rao and Ross is a low memory overhead pointerless index that speeds up searches in sorted arrays [21]. In our system, we use it as an append-only replacement for the temporal B+-tree forest (cf. Section 4.1.2) to speed up Procedures 3 and 4 and to reduce memory consumption. Furthermore, its ability to efficiently compute the size of a key range in logarithmic time is used to improve the accuracy of the cardinality estimator (cf. Section 4.4). The CSS-tree is optimized to reduce the number of cache misses during a search by using the processor's cache line size as its node size. Since it indexes sorted arrays, only appends can be performed efficiently. We deem this an acceptable trade off because inserting additional trajectories would also require a re-computation of the entire FM-index, making the index mostly suited for batch updates.

Procedure 3 Create a mapping of trajectory identifier and sequence number (d, seq) to an antecedent travel time *diff* for the first β trajectories matching the predicates (*buildMap*):

```
Input: Temporal index $\Phi$ of the first segment of the query path,
    ISA range $[st, ed)$, time interval $I$, predicate $f$, and cardinality
    parameter $\beta$
Output: a mapping of $(d, seq)$ to $(a - TT)$
 1: $M \leftarrow \emptyset$
 2: for all $r \in \Phi$ do
 3:     if $r.t \in I \wedge st \leq r.isa < ed \wedge f(r.d)$ then
 4:         $diff \leftarrow r.a - r.TT$
 5:         $M \leftarrow M \cup \{(r.d, r.seq) \rightarrow diff\}$
 6:         if $|M| \geq \beta$ then
 7:             return $M$
 8:         end if
 9:     end if
10: end for
11: return $M$
```

Procedure 4 Compute the travel times for all eligible trajectories over the path identified in the *buildMap* function (*probeMap*):

```
Input: Temporal index $\Phi$ of the last segment of the query path,
    path length $l$, and probe table $M$
Output: a list of travel times $X$
 1: $X \leftarrow \emptyset$
 2: for all $r \in \Phi$ do
 3:     $b \leftarrow M[(r.d, r.seq + 1 - l)]$
 4:     if $b \neq \emptyset$ then
 5:         $X \leftarrow X \cup \{r.a - b.diff\}$
 6:     end if
 7: end for
 8: return $X$
```

Procedure 5 Retrieve all travel-times $X = \langle x_0, ..., x_{\beta-1} \rangle$ of trajectories in I that meet predicate f for a path P (*getTravelTimes*):

```
Input: Burrows-Wheeler transform of the trajectory string $T_{bwt}$,
    temporal indexes $F$, symbol counts $C$, path $P = p_0 ... p_{l-1}$,
    time interval $I$, predicate $f$, and cardinality parameter $\beta$
Output: a set of travel-times $X$
 1: $[st, ed) \leftarrow getISARange(T_{bwt}, C, P)$
 2: if $st \geq ed$ then
 3:     return $\emptyset$
 4: end if
 5: $\Phi_0 \leftarrow F[p_0]$
 6: $M \leftarrow buildMap(\Phi_0, [st, ed), I, f, \beta)$
 7: if $|M| < \beta$ and $isPeriodic(I)$ then
 8:     return $\emptyset$
 9: end if
10: $\Phi_{l-1} \leftarrow F[p_{l-1}]$
11: $X \leftarrow probeMap(\Phi_{l-1}, l, M)$
12: if $X = \emptyset$ and $|P| = 1$ then
13:     return $\{estimateTT(p_0)\}$
14: end if
15: return $X$
```

Procedure 6 Compute a histogram H for the query $spq(P, I, f, \beta)$ (*tripQuery*):

```
Input: Burrows-Wheeler transform of the trajectory string $T_{bwt}$,
    temporal indexes $F$, symbol counts $C$, query $spq(P, I, f, \beta)$,
    time interval sizes $A$, partitioning method $\pi$, and splitting
    method $\sigma$
Output: a histogram $H$
$\langle Q_1, \ldots, Q_k \rangle \leftarrow \pi(Q); \mathcal{H} \leftarrow \emptyset$
for all $Q_i \in \langle Q_1, \ldots, Q_k \rangle$ do
    if isPeriodic($I_i$) and $i > 1$ then
        $I_i \leftarrow [t_s + S_i, t_e + R_i)^R$
    end if
    $X_i \leftarrow$ getTravelTimes($P_i, I_i, f_i, \beta$)
    if $X_i \neq \emptyset$ then
        $\mathcal{H} \leftarrow \mathcal{H} \cup$ createHistogram($X_i$)
    else
        $\langle Q_{i+1}, \ldots, Q_k \rangle \leftarrow \langle Q_{i+1}, \ldots, Q_k \rangle \cup \sigma(Q_i)$
    end if
end for
$H \leftarrow H_1$
for all $i > 1 \wedge H_i \in \mathcal{H}$ do
    $H \leftarrow H * H_i$
end for
return $H$
```

4.3.2 Temporal Partitioning. Temporal partitioning of the SNT-index was originally proposed here [13], but not evaluated. It allows more efficient updates to the index without necessitating a complete re-computation of the FM-index which does not efficiently support updates or appends. Partitioning requires to split the trajectory set $\mathcal{T}$ into $\mathcal{T}_1, \ldots, \mathcal{T}_W$, where $\forall i < j \; (\nexists tr_i \in \mathcal{T}_i \; (\forall tr_j \in \mathcal{T}_j \; (tr_i.t_0 \geq tr_j.t_0)))$. From those trajectory sets, W trajectory strings $T^1, \ldots, T^W$ are then computed, and Procedure 2 is modified return a collection of ISA ranges from the Burrows-Wheeler transforms $T^1_{bwt}, \ldots, T^W_{bwt}$ using separate segment counters $C^1, \ldots, C^W$. Temporal partitioning also requires adding the partition identifier w to every leaf in the temporal indexes since every partition's FM-index can return a different ISA range for the same path.

4.4 Cardinality Estimator

Cardinality estimators are widely used in DBMSs to improve query plans. In our case, we want to avoid costly scans of our temporal indexes if the required sample size β cannot be met. We require a function $card(Q)$ that returns an estimate $\hat{\beta}$ for the cardinality of the return trajectory set $\mathcal{T}$ and if $\hat{\beta} < \beta$, we apply the split function σ to Q without running a costly query. The cardinality estimator relies on a time-of-day histogram for every segment and fast computation of the ISA range $[st, ed)$, which is enabled by Procedure 2. The exact count of all trajectories traversing a path $c_P = ed - st$ is efficiently retrieved. After that, the selectivity of the temporal filters needs to be estimated. The easiest way is to assume a uniform distribution throughout the day and to divide the size of a periodic interval by the length of the day, which yields the time-of-day selectivity:

$$sel_{tod} = sel(P, I^R = [t_s, t_e)^R) = \frac{t_e - t_s}{24 \; hours} \tag{1}$$

The uniformity assumption, however, usually does not hold so, the selectivity estimate can be improved by maintaining a time-of-day histogram H_e for every segment e. Then the selectivity can be estimated using the following formula:

$$sel(P, I^R = [t_s, t_e)^R) = \frac{B\big(H_{e_0}, [t_s, t_e)\big)}{B\big(H_{e_0}, [0, 24 \; hours)\big)}, \tag{2}$$

where $B\big(H, [t_s, t_e)\big)$ counts the elements of all buckets in H in the range $[t_s, t_e)$. In addition to being constrained by the time-of-day a user might wish to limit the query to a certain time frame, e.g., only considering trajectories within the past year. The selectivity can be estimated naively with the following formula:

$$sel_{tf} = sel(P, I = [t_s, t_e)) = \frac{t_e - t_s}{F[e_0]_{max} - F[e_0]_{min}}, \tag{3}$$

where $F[e_0]_{min}$ and $F[e_0]_{max}$ are the earliest and latest traversal times of segment e_0. When using the CSS-tree, the number of entries for which $t_s \leq t < t_e$ can be obtained exactly in logarithmic time and sel_{tf} can be computed exactly. To compute the selectivity of user predicates sel_u, we use the default of $\frac{1}{10}$ suggested by Selinger et al. [22]. To obtain the estimate for a query, we combine these selectivity factors to obtain our estimate $\hat{\beta} = sel_{tod} * sel_{tf} * sel_u * c_P$.

We define five different modes for the cardinality estimator:

ISA only uses the size of the ISA range c_P as estimate $\hat{\beta}$

BT-Fast uses formulas 1 and 3 to estimate the selectivity

BT-Acc uses formulas 2 and 3 to estimate the selectivity

CSS-Fast uses formula 1 and a fast lookup in the CSS-tree to estimate the selectivity

CSS-Acc uses formula 2 and a fast lookup in the CSS-tree to estimate the selectivity

5 EXPERIMENTAL SETUP

This section describes the data set and quality metrics we use to evaluate our system.

5.1 Datasets

5.1.1 OpenStreetMap. Our network graph is based on the OpenStreetMap data of the road network of Northern Denmark, which contains around 750,000 road segments. When converted to a spatial network graph, this graph has around 1.46 million directed edges [19]. Each edge represents a direction on a segment and has one of 17 different segment categories. This categorization is available for all OpenStreetMap maps and makes segment category-based partitioning possible for other map-matched trajectory datasets as well. The OpenStreetMap data also includes the speed limits for many segments, which we use as a fallback if no trajectory data is available. If the speed limit is not known, we use the median of all known speed limits of its segment category.

5.1.2 Zone Dataset. To distinguish rural and urban areas, we use the zoning map published by the Danish Business Authority [7] that consists of 4,259 zone geometries, each of which assigns one of three categories to an area:

- *city*: segments within city limits
- *rural*: segments in rural areas
- *summer house*: segments in areas zoned for summer house usage

A spatial join is used to assign a zone type to every segment in the map. A fourth category that we call *ambiguous* is assigned to segments located in more than one zone type.

5.1.3 ITSP Dataset. The "ITS Platform" dataset contains over 1.1 billion GPS points sampled at 1 Hertz collected from 458 vehicles in Aalborg and the surrounding region during the period from May 2012 to December 2014 [1].

In a preprocessing step, the GPS points are map-matched [18] to obtain in excess of 79 million segment traversals that form around 1.4 million trajectories, where a new trajectory is created if more than a 180 seconds have elapsed since the last GPS point. The map-matching algorithm also discards GPS points at the beginning and end of a trip if too few points are matched to the start and end segments of the trajectory. This is done so the durations of the segment traversals are meaningful. Each GPS record contains the trajectory ID, the vehicle ID, a segment ID, the time and date the segment was entered (minute resolution), and the time on segment (second resolution). Since the cars in our dataset are privately owned, we treat the vehicle ID as the user ID. The segment IDs are derived from the unique mapping of OpenStreetMap segment key and the driving direction. The time on a segment is also computed during the preprocessing step.

5.2 Query

We derive our query set $\mathcal{Q}$ from a random sample $\mathcal{T}_S \subset \mathcal{T}$ from our trajectory set:

$$\mathcal{Q} = \{spq(P_{tr}, I_{tr}, f, \beta) | tr \in \mathcal{T}_S\},$$

with either $f = \{u = tr.u\}$ or $f = \emptyset$ if no user filters are used and different values of β being used in the experiments. For the time interval I_{tr}, either the periodic time interval $I^R_{tr} = [tr.s.t_0 - \frac{\alpha_{min}}{2}, tr.s.t_0 + \frac{\alpha_{min}}{2})^R$ or or the fixed time interval $I_{tr} = [0, tr.s.t_0)$ is used.

For the interval size we use the values 15 *min*, 30 *min*, 45 *min*, 60 *min*, 90 *min*, and 120 *min*.

5.3 Accuracy Metric

5.3.1 sMAPE. To evaluate the accuracy of the retrieved traversal times, we use the symmetric mean absolute percentage error [2] of the sum of the means of all sub-paths.

$$\text{sMAPE} = \frac{100\%}{|\mathcal{Q}|} \sum_{i=1}^{|\mathcal{Q}|} \frac{|\sum_{j=1}^{k} \bar{X}_j - a_{tr_i}|}{\frac{1}{2}(\sum_{j=1}^{k} \bar{X}_j + a_{tr_i})},$$

where k is the respective number of sub-queries of each query $Q \in \mathcal{Q}$ and $\bar{X}_j$ is the travel-time mean retrieved with the sub-query.

5.3.2 Weighted Error. The weighted error, which we derive from sMAPE, considers the accuracy of the sub-query results and weighs them according to their fraction of the path length.

$$wE = \frac{100\%}{|\mathcal{Q}|} \sum_{i=1}^{|\mathcal{Q}|} \sum_{j=1}^{k} w_j \frac{|\bar{X}_j - a^{P_j}_{tr_i}|}{\frac{1}{2}(\bar{X}_j + a^{P_j}_{tr_i})},$$

with $w_j = \frac{\sum^{e \in P_j} \mathcal{F}(e).l}{\sum^{e \in P} \mathcal{F}(e).l}$, where P is the query path and P_j is the sub-query path.

5.3.3 Log-Likelihood. To evaluate the quality of the histograms, we compute the average log-likelihood of the travel-times a_{tr_i} with a discrete probability density function derived from the result histogram H_i.

For each trajectory with a result histogram H with a bucket width h, we compute the average log-likelihood $log\mathcal{L}$:

$$\frac{1}{|\mathcal{Q}|} \sum_{i=1}^{|\mathcal{Q}|} log\mathcal{L}(a_{tr_i}, H_i),$$

where the likelihood $\mathcal{L}(x, H)$ is defined by the discrete probability density function

$$p_H(x) = \gamma f(x, H) + (1 - \gamma)U(x),$$

where $U(x)$ is a uniform distribution defined for $[t_{min}, t_{max})$, $0 < \gamma < 1$ and

$$f(x, H) = \frac{B\big(H, [\lfloor \frac{x}{h} \rfloor, \lfloor \frac{x}{h} \rfloor + h)\big)}{B(H, [t_{min}, t_{max}))}.$$

The smoothing with the uniform distribution $U(x)$ is performed so that $p_H(x)\ \forall x \in [t_{min}, t_{max})$ never reaches zero.

5.3.4 Q-Error. To evaluate the accuracy of the cardinality estimator, we use the q-error proposed by Moerkotte et al. [16]. To estimate the quality of our cardinality estimate $\hat{\beta}$, we compare it to the actual cardinality of the retrieved trajectory set $n = |\mathcal{T}|$. For every estimate, we obtain the q-error $q = max(\hat{\beta}'/n', n'/\hat{\beta}')$ with $n' = max(n, 1)$ and $\hat{\beta}' = max(\hat{\beta}, 1)$. This is done to handle estimations for empty sets as proposed by Stefanoni et al. [23]. The q-error shows the difference in orders of magnitude between the real cardinality and the estimate.

6 EVALUATION

This section reports on the experimental results. For all experiments, a query set $\mathcal{Q}$ is generated from the trajectory set $|\mathcal{T}_S| = 6{,}942$, which is a random 1% sample of all trajectories in $\mathcal{T}$ that occur after the 8th of September 2013, the median of the timestamps in the ITSP data set. This is to ensure that every query has more than a year of trajectory data available. On average, the paths of the query set have a length of 13.7 kilometers, consist of 55 segments, and last 800 seconds.

In our study we evaluate three types of queries:

Temporal Filters that use a periodic time interval and no user filter ($spq(P_{tr}, I^R_{tr}, \emptyset, \beta)$)

User Filters that use a periodic time interval and a user filter ($spq(P_{tr}, I^R_{tr}, \{u = tr.u\}, \beta)$)

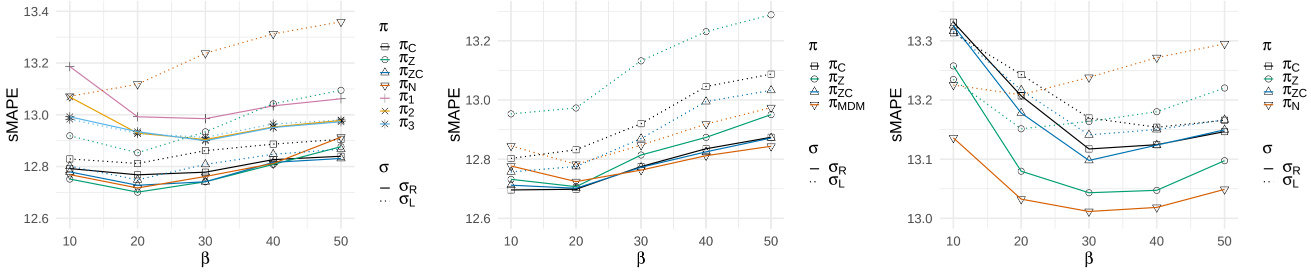
SPQ Only that use a fixed time interval and no user filter ($spq(P_{tr}, [0, t_{max}), \emptyset, \beta)$)

6.1 Qualitative Assessment

Figures 5 to 8 show the results of accuracy measured with sMAPE, the weighted error, and the log-likelihood and the average sub-query length. The figures show the results for different types of partitioning and splitting methods and filter predicates.

The regular partitioning method π_p (cf. Section 3.2.1) is used as a baseline with $p = 1, 2,$ and 3 because they are the sub-path lengths for which histograms can still be pre-computed at a reasonable overhead and because no known histogram-based methods perform better. For the user filter queries, we also evaluate the π_{MDM} method that partitions queries like π_C but only applies user filters to sub-queries with paths on main roads like motorways or other major roads connecting cities. This partitioning method is derived from the results of a previous study of travel-time estimation methods [26].

Figure 5a shows the average error for seven different partitioning methods with temporal filters. Here, π_1 performs worst, followed π_2 and π_3, and they achieve their highest accuracy at $\beta = 30$. If only the speed limits are used to estimate the travel time, sMAPE is 34.3% and if all available trajectories for each segment are used, the error is 13.8%. The partitioning methods based on the segment category and/or zone (π_C, π_Z, and π_{ZC})

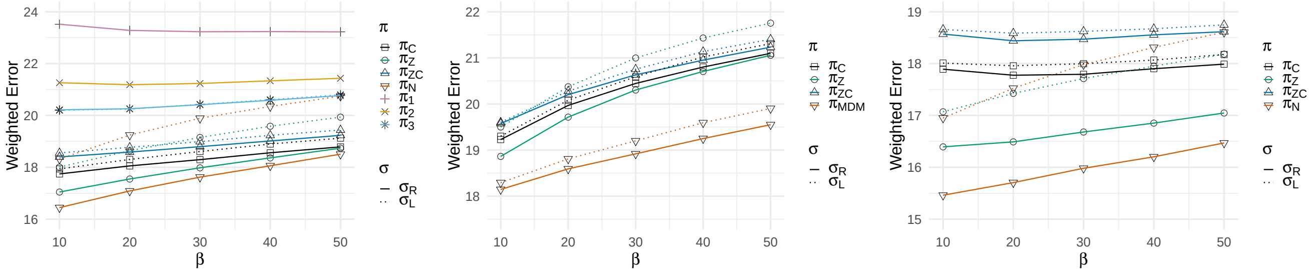


(a) Temporal Filters

(b) User Filters

(c) SPQ Only

Figure 5: sMAPE

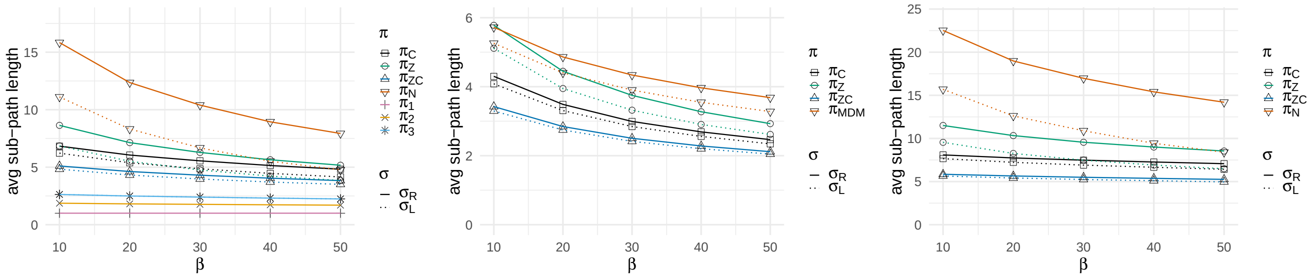


(a) Temporal Filters

(b) User Filters

(c) SPQ Only

Figure 6: Weighted Error

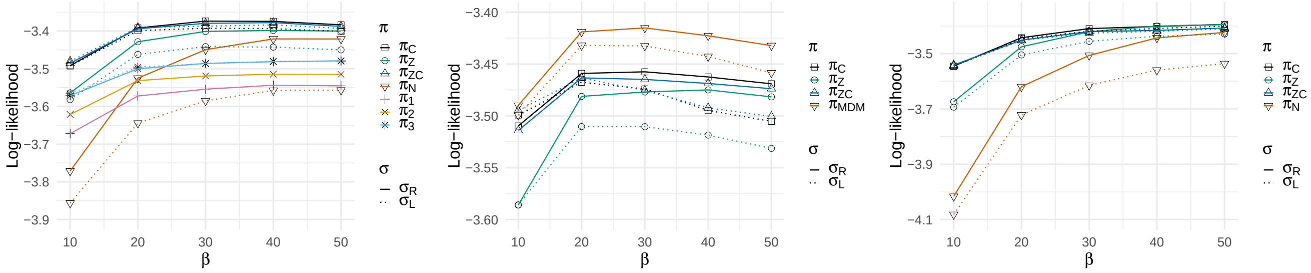


(a) Temporal Filters

(b) User Filters

(c) SPQ Only

Figure 7: Sub-query Path Length



(a) Temporal Filters

(b) User Filters

(c) SPQ Only

Figure 8: Log-Likelihood

together with π_N achieve very similar accuracy. Here, the accuracy peaks at $\beta = 20$. Category-based partitioning is the most stable in terms of accuracy, and zone-based partitioning provides the overall best result. The queries using user filters shown in Figure 5b perform equally well, but with the exception of π_Z do not degrade as much with higher values of β and also obtain their lowest error at $\beta = 20$ and exhibit very similar accuracy to the queries without user filters. The SPQ Only methods in Figure 5c methods did not manage to outperform the baseline because it does not use periodic intervals that can observe changing congestion, e.g., longer travel-times during rush hours. In nearly all cases with regular splitting (σ_R) achieves considerably better accuracy than longest prefix splitting (σ_L). In most cases

A similar picture to the sMAPE results can be seen for the temporal filter queries in Figure 6a, with π_N having the lowest weighted error. If only the speed limits are used to make an estimate, the weighted error is 36.9%, and if all available trajectories for each segment are used, the error is 24.0%. For the user filter queries in Figure 6b, only π_{MDM} manages to consistently outperform the baseline. The SPQ only queries shown in Figure 6c show the lowest error with the coarsest partitioning methods. The low error of the SPQ only methods is due to sub-query results being weighted according to the length of the sub-paths and not relative to the share of travel time. Estimates for paths on long segments with high speed limits, e.g., motorways, exhibit already low estimation errors and also tend to improve the most when custom predicates are used [26]. In all cases, σ_L has a higher error than σ_R. Figure 7 shows the average path lengths of the final sub-queries. We can see that there is an inverse relationship between the weighted error and the sub-query paths. We can also see that π_Z provides the coarsest partitioning with the exception of π_N, which initially provides none.

Figures 8a to 8c show the average log-likelihood with $f(x, H)$ derived form a histogram with a bucket size of $h = 10s$ and different values for β and $\gamma = 0.99$. The queries with only temporal filters and π_Z and π_{ZC} return the most accurate histograms, and the coarser the partitioning method the less accurate the histograms are with low sample sizes. Among the User Filter queries π_{MDM} consistently outperforms the other three partitioning methods. The queries run with π_N do not even outperform the baseline for $\beta < 30$. In all cases, σ_L performs worse than σ_R. We evaluated several values for γ (from 0.90 to 0.99) but the qualitative results did not change.

6.2 Efficiency

The index is implemented in C++17 and compiled with g++ 7.2.0 with the `-O3 -march=native` flags. For the performance test, the SNT-index with a CSS-forest and only a single partition is used. The FM-index is implemented using `sdsl-lite`'s integer-alphabet Huffman-shaped wavelet tree implementation, and the suffix array is computed with Yuta Mori's `sais-lite` library [17]. The performance test ran on a server with AMD Opteron 6376 processors and 512 GiB RAM. For the processing time, the average runtime in milliseconds of 6,942 queries is reported in Figure 9.

The temporal filter queries shown in Figure 9a perform very similar to the baseline, with π_C and π_{ZC} being slightly faster than regular partitioning. The combination of π_C and σ_L has been omitted in the figure for reasons of scaling since the results are in the range of 50 to 65 ms. In Figure 9b, it can be seen that the average user filter query takes around 4 to 5 times longer than the temporal filter queries; and with π_{MDM}, queries only take around twice as long since it applies non-temporal predicates only selectively. SPQ only queries have much lower processing times than do the other two query types, and all consistently outperform the baseline. The reason for their low processing times can be seen in Figure 7c and Procedure 5. Since their sub-queries tend to cover comparatively long paths, SPQ only queries need to perform considerably fewer temporal index scans than the other query types, which need to be split into more sub-queries to fulfill the cardinality requirements. The average runtime of π_N with σ_L, which is between 30 to 35 ms, has been omitted in Figure 9c. In all cases σ_L performed poorly in comparison to σ_R.

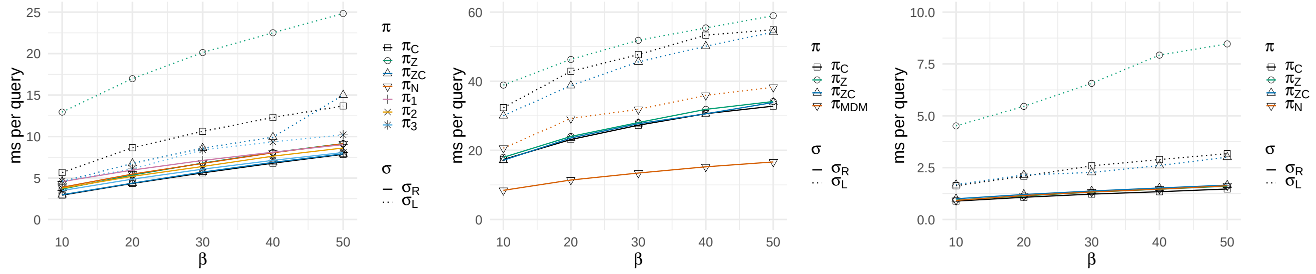
6.3 Temporal Partitioning

Figure 10 shows the effect of the temporal partitioning defined in Section 4.3.2 on memory consumption and setup time. The figures show the results for partition sizes of 7, 30, 90, and 365 days, resulting in a 138, 33, 11, and 3 partitions, respectively. We also examine the case where only one partition is used (FULL). Where applicable, the performance of the index with a B+-forest (BT) is reported as well. For the in-memory B+-trees, we use the `btree_multimap` from Google's `cpp-btree` library [11]. Figure 10a shows the memory consumption of the different index components, where *Forest* is the memory consumption of the CSS-tree or B+-tree forest, respectively. The size of the forest is not impacted by different partition sizes, but if the partition feature is removed from the index, the memory saved in the tree leafs by omitting the partition identifier w is around 300 MiB for our data set. We can also see that the in-memory B+-tree forest has slightly higher memory requirements than the CSS-forest. The associative container U used to enable user filtering (user) is also not affected by the partitioning and takes up around 65 MB for our data set. The two data structures that comprise the FM-index, the wavelet tree (WT) and the segment counter (C), are affected considerably by partitioning. The segment counter grows linearly with the number of partitions from less than 6 MB to nearly 600 MB since a separate segment count needs to be maintained for every wavelet tree. The compression rate of the wavelet tree degrades considerably with smaller trajectory strings, which for the 7 day partitioning are only a few MBs per partition as opposed to several hundred in a single partition and it grows from around 280 MB to over 4 GB. The memory requirements of the time-of-day histograms required for the cardinality estimator are affected considerably if a histogram is maintained for every non-empty partition for every segment, and the memory required for the histograms soon exceeds the amount required for the index. Figure 10b shows the memory consumption for three different bucket sizes h (1, 5, and 10 minutes).

The setup times for the index shown in Figure 10c are not significantly affected by the different partition sizes or tree types and always remain between 425 and 475 seconds. For the setup, the trajectory and map data are loaded from disk.

6.4 Cardinality Estimator

Figure 10 shows the results for the cardinality estimator. In all cases the results for partitioning method π_Z with regular splitting and $\beta = 20$ are shown. Figure 11a shows the q-error of the five different cardinality estimator modes. Here, 5,000 queries are run, after which their cardinalities n are compared with estimate $\hat{\beta}$. The simplest estimate using just the ISA range is on average off by an order of magnitude. The four other modes provide considerably more reliable estimates, with the histogram

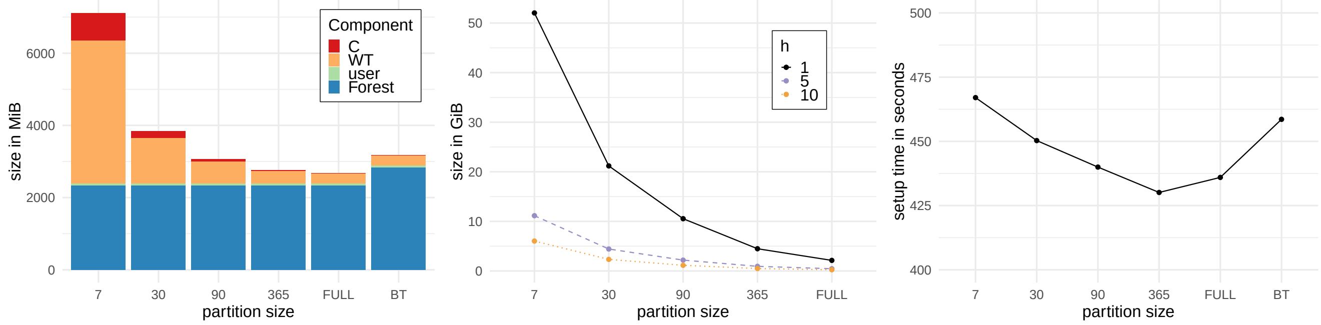


(a) Temporal Filters

(b) User Filters

(c) SPQ Only

Figure 9: Processing Time

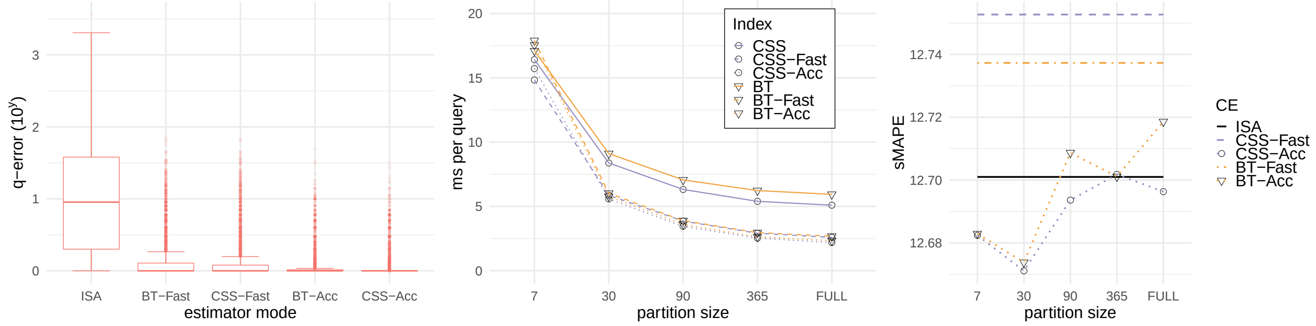


(a) Index Memory Consumption

(b) Histogram Memory Consumption

(c) Setup Time

Figure 10: Temporal Partitioning



(a) Q-Error

(b) Runtime

(c) Effect on Accuracy

Figure 11: Cardinality Estimator

based methods performing better than the fast ones and the CSS-tree based methods performing slightly better than their B+-tree counterparts.

Since the selectivity estimates of the estimators might underestimate cardinalities of queries, a query might be split despite covering a sufficient sample size. This may affect the quality of the overall travel-time estimate. Figure 11c, however, shows that the effects on quality are minuscule compared to the baseline (ISA) and might even yield slight improvements in accuracy.

Figure 11b shows that partitioning as well as using the cardinality estimators can impact performance significantly. For single, yearly, and quarterly partitions, the query performance changes little, and use of the cardinality estimator reduces query processing times by around 50%. For smaller partitions, however, the effects of using the cardinality estimator diminish; and with weekly partitioning, the B+-tree version of the index performs worse with the estimators. The histogram-based CSS-tree version (CSS-Acc) performs worse than the fast version (CSS-Fast), which is most likely due to the amount of time-of-day histograms that have to be scanned to obtain the selectivity sel_{tod}.

6.5 Implications

Overall our data shows that after a certain β is reached no significant gains in accuracy are obtained by increasing it further indicating smaller result sets obtained from fewer SPQs of long paths provide more accurate estimates than larger result sets obtained with short paths. One can also see that evaluating non-temporal predicates comes with a considerable overhead and for the user predicates provides no improvement in quality over the purely temporal methods. If such methods are however applied selectively (e.g. π_{MDM}) the performance overhead is mitigated and the accuracy improves. The naive regular splitting method does not only achieve better accuracy but also has a considerably shorter runtime, making it more suitable for a real-time queries. The CSS-tree version of the index is as least as fast or faster than the B+-tree-based version, but the improvements become less noticeable when using the index in conjunction with a cardinality estimator. CSS-trees reduce the memory consumption of the index as well and improve the accuracy of the cardinality estimator with their efficient range lookups. We have also shown that temporal partitioning of the index is viable in some cases, but that using time-of-day histograms to estimate the selectivity of periodic time intervals, despite slight improvements in estimator accuracy and query performance, is hardly worth the memory overhead for the evaluated data set. Additional experiments with larger data sets may offer additional insight into this trade off, but no larger trajectory data sets with user information were available to us. Our results show that modifications aimed at improving query performance often also improve the accuracy of the estimates.

7 CONCLUSIONS & OUTLOOK

Travel-time estimations in road networks can be improved considerable by utilizing large NCT data sets not only to provide estimates on a segment level, but also for full paths in the network. To our knowledge no current system supports these path-based estimations which cannot rely on pre-computations. We therefore propose a system that computes travel-time estimations based on trajectories selected at runtime and is able to improve upon the accuracy of existing histogram-based methods by expressing them as a series of strict path queries and adapting their predicates automatically to ensure accurate estimates. The SPQs are processed by our adapted SNT-index which is able to retrieve the traversal times for any path directly from the index. We have shown that the queries can be processed fast enough for real-time applications by utilizing specialized in-memory data structures and cardinality estimators tailored to SPQs. We evaluate our system with a large real-world trajectory data set and find that optimizing queries for performance is not preclusive of accuracy.

Our proposed system leaves several avenues of future work. The current greedy algorithm used for identifying a suitable partitioning and splitting of an SPQ is based on fairly simple heuristics and could be augmented by more sophisticated machine learning methods to improve accuracy of estimations. Approaches that use different values of the parameter β for each sub-query, e.g., smaller sample size requirements in rural zones, could be evaluated. While the processing time of single query might not considerably improve through parallelization the overall query throughput of the system most likely could, making it suitable for online routing applications that support a large number of users. Our approach also does not fully address the issue of data sparseness apart from providing relaxing the predicates if their selectivity is too low. Several approaches to solving the problem of data sparseness have been suggested [25, 30] and could be combined with our system to provide time-dependent travel-time estimates for paths where data is sparse.

ACKNOWLEDGMENTS

This research was supported in part by a grant from the Obel Family Foundation and by the DiCyPS project.

REFERENCES

[1] Aalborg University. 2018. ITS Platform. http://www.itsplatform.dk/.

[2] J Scott Armstrong. 1985. *Long-Range Forecasting: From Crystal Ball to Computer* (2 ed.). John Wiley & Sons, Inc.

[3] Michael Burrows and David J Wheeler. 1994. *A Block-sorting Lossless Data Compression Algorithm.* Technical Report. Digital Equipment Corporation.

[4] Jian Dai, Bin Yang, Chenjuan Guo, Christian S Jensen, and Jilin Hu. 2016. Path Cost Distribution Estimation Using Trajectory Data. *Proceedings of the VLDB Endowment* 10, 3 (2016), 85–96.

[5] Victor Teixeira De Almeida and Ralf Hartmut Güting. 2005. Indexing the Trajectories of Moving Objects in Networks. *GeoInformatica* 9, 1 (2005), 33–60.

[6] Zhiming Ding. 2008. UTR-tree: An Index Structure for the Full Uncertain Trajectories of Network-Constrained Moving Objects. In *9th IEEE International Conference on Mobile Data Management.* IEEE, 33–40.

[7] Erhvervsstyrelsen. 2018. zonekort. http://kort.plandata.dk/spatialmap.

[8] Eurostat. 2018. Transport, volume and modal split. https://ec.europa.eu/eurostat/web/transport/data/main-tables.

[9] Elias Frentzos. 2003. Indexing Objects Moving on Fixed Networks. In *8th International Symposium on Spatial and Temporal Databases.* Springer, 289–305.

[10] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From Theory to Practice: Plug and Play with Succinct Data Structures. In *13th International Symposium on Experimental Algorithms.* Springer, 326–337.

[11] Google Inc. 2011. cpp-btree. https://code.google.com/archive/p/cpp-btree/.

[12] Satoshi Koide, Yukihiro Tadokoro, and Takayoshi Yoshimura. 2015. SNT-index: Spatio-temporal index for vehicular trajectories on a road network based on substring matching. In *1st International ACM SIGSPATIAL Workshop on Smart Cities and Urban Analytics.* ACM, 1–8.

[13] Satoshi Koide, Yukihiro Tadokoro, Takayoshi Yoshimura, Chuan Xiao, and Yoshiharu Ishikawa. 2018. Enhanced Indexing and Querying of Trajectories in Road Networks via String Algorithms. *ACM Transactions on Spatial Algorithms and Systems* 4, 1 (2018), 1–41.

[14] Benjamin Krogh, Nikos Pelekis, Yannis Theodoridis, and Kristian Torp. 2014. Path-based Queries on Trajectory Data. In *22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* ACM, 341–350.

[15] Yu Ma, Bin Yang, and Christian S Jensen. 2014. Enabling Time-Dependent Uncertain Eco-Weights For Road Networks. In *Workshop on Managing and Mining Enriched Geo-Spatial Data.* ACM, 1–6.

[16] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proceedings of the VLDB Endowment* 2, 1 (2009), 982–993.

[17] Yuta Mori. 2008. SAIS: An implementation of the induced sorting algorithm.

[18] Paul Newson and John Krumm. 2009. Hidden Markov Map Matching Through Noise and Sparseness. In *17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* ACM, 336–343.

[19] OpenStreetMap contributors. 2014. Planet dump retrieved from https://planet.osm.org. https://www.openstreetmap.org.

[20] Iulian Sandu Popa, Karine Zeitouni, Vincent Oria, Dominique Barth, and Sandrine Vial. 2010. PARINET: A Tunable Access Method for in-Network Trajectories. In *26th IEEE International Conference on Data Engineering.* IEEE, 177–188.

[21] Jun Rao and Kenneth A Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *25th International Conference on Very Large Databases.* 78–89.

[22] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access Path Selection in a Relational Database Management System. In *ACM SIGMOD International Conference on Management of Data.* ACM, 23–34.

[23] Giorgio Stefanoni, Boris Motik, and Egor V Kostylev. 2018. Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation. In *27th International World Wide Web Conference.* 1043–1052.

[24] Dong Wang, Junbo Zhang, Wei Cao, Jian Li, and Yu Zheng. 2018. When Will You Arrive? Estimating Travel Time Based on Deep Neural Networks. In *32nd AAAI Conference on Artificial Intelligence.* 2500–2507.

[25] Yilun Wang, Yu Zheng, and Yexiang Xue. 2014. Travel Time Estimation of a Path using Sparse Trajectories. In *20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM, 25–34.

[26] Robert Waury, Christian S Jensen, and Kristian Torp. 2018. Adaptive Travel-Time Estimation: A Case for Custom Predicate Selection. In *19th IEEE International Conference on Mobile Data Management.* IEEE, 96–105.

[27] Stephan Winter. 2002. Modeling Costs of Turns in Route Planning. *GeoInformatica* 6, 4 (2002), 345–361.

[28] Chun-Hsin Wu, Jan-Ming Ho, and Der-Tsai Lee. 2004. Travel-Time Prediction With Support Vector Regression. *IEEE Transactions on Intelligent Transportation Systems* 5, 4 (2004), 276–281.

[29] Jing Yuan, Yu Zheng, Chengyang Zhang, Wenlei Xie, Xing Xie, Guangzhong Sun, and Yan Huang. 2010. T-Drive: Driving Directions Based on Taxi Trajectories. In *18th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* ACM, 99–108.

[30] Fangfang Zheng and Henk Van Zuylen. 2013. Urban link travel time estimation based on sparse probe vehicle data. *Transportation Research Part C: Emerging Technologies* 31 (2013), 145–157.

BB-Tree: A practical and efficient main-memory index structure for multidimensional workloads

Stefan Sprenger
Humboldt-Universität zu Berlin
Berlin, Germany
sprengsz@informatik.hu-berlin.de

Patrick Schäfer
Humboldt-Universität zu Berlin
Berlin, Germany
schaefpa@informatik.hu-berlin.de

Ulf Leser
Humboldt-Universität zu Berlin
Berlin, Germany
leser@informatik.hu-berlin.de

ABSTRACT

We present the BB-Tree, a fast and space-efficient index structure for processing multidimensional read/write workloads in main memory. The BB-Tree uses a k-ary search tree for pruning and searching while keeping all data in leaf nodes. It linearizes the inner search tree and manages it in a cache-optimized array, creating the need for occasional re-organizations when data changes. To reduce the frequency of such re-organizations, the BB-Tree introduces a novel architecture for leaf nodes, called bubble buckets, which can automatically morph between different representations depending on their fill degree and are thus able to buffer a large number of insertions or deletions in-place. We compare the BB-Tree to scanning, main-memory variants of the R*-tree, the kd-tree, and the VA-file, and the recent PH-tree using different multidimensional workloads over real and synthetic data sets. The BB-Tree is the fastest access method for range queries up to a selectivity of around 20% (after which it is only beaten by scanning), the fastest method in read/write workloads, and achieves an exact-match query performance similar to that of the best point access method. In addition, it is the most space-efficient method of all considered index structures. We also describe a parallel range query operator and show that it scales with the number of physical cores.

1 INTRODUCTION

Many data sets are inherently multidimensional, with typical dimensionalities ranging from two to a few dozen. We give three examples: (1) Precision Medicine is based on the comparison of a patient's mutational landscape to that of background populations and disease cohorts. Each mutation is a multidimensional object [13], with dimensions like genomic location, type of the mutation, prevalence in a population, functional impact, etc. Oncologists query such data seeking for mutations with certain properties to discover commonalities across diseases or treatment results [21]. (2) In modern machine surveillance, a battery of sensors measure multiple properties of parts of engines, such as temperature, vibration, electric currents, humidity, accelerations in all three spatial dimensions, etc. Analyzing such data for specific events or situations often induces executing series of multidimensional range queries (MDRQ) [20]. (3) In data warehousing, commercially relevant events are described by multiple, often hierarchically organized dimensions, leading to the famous OLAP cube [11]. Slicing such a cube, i.e., selecting (aggregated) events based on values of certain dimensions, often boils down to MDRQ, for instance selecting all sales in a certain date and price range.

Searching multidimensional data can be sped up by using multidimensional index structures (MDIS). MDIS can support different types of queries; in this work we focus on range queries over all (complete-match MDRQ) or a subset of the dimensions of a data space (partial-match MDRQ). MDIS are different from one-dimensional index structures because they cannot exploit a natural sort order in the data. Especially partial-match queries require MDIS to treat all dimensions equally, which typically is achieved by building and maintaining some, often hierarchical, structure on top of the data [26]. Navigation of such a structure necessitates inefficient random access patterns, which quickly leads to the situation that MDIS are outperformed by scans when queries are less selective, irrespective of whether data are held on disk [34] or in main memory [28]. The aim of this research is thus to create an MDIS that can efficiently support exact-match and range queries, that has low memory overhead, that performs gracefully in mixed read/write workloads, that is robust against the dimensionality of the data (up to a certain point) and that is faster than scans even for less selective queries.

In this paper, we present the BB-Tree, a novel main-memory MDIS which fulfills these requirements. Conceptually, a BB-Tree is an *almost-balanced* k-ary search tree, where inner nodes recursively split the data space into k partitions according to a delimiter dimension and $k-1$ delimiter values. Data objects are stored in leaf nodes (buckets). When too many data points are inserted (or deleted) and buckets overflow (or underflow), the structure is rebuilt to achieve a balance that is beneficial regarding the depths of leaves. Within this general and well-known layout, the BB-Tree combines a number of advanced techniques that yield its superior performance.

As the main contribution, BB-Trees introduce elastic buckets, called *bubble buckets* (BB), that can efficiently handle strongly fluctuating bucket fill degrees and that significantly reduce the frequency of index rebuilds. BB automatically morph between different representations, depending on their number of stored data objects. We distinguish between *regular* and *super* BB. Regular BB can hold up to b_{max} data objects and are implemented using arrays. Super BB are composites and consist of a routing node and a set of up to k regular BB, hence, they locally add a further level to the tree. BB can dynamically grow and shrink: Overflowing regular BB let them morph into super BB, and underflowing super BB let them morph back into regular BB. Both operations leave the rest of the BB-Tree unchanged. Since overflows create k new leaf buckets, a BB can cater for a rather large number of inserts. Eventually, the tree must be rebuilt when a super BB overflows. In workloads with *hammered* inserts, i.e., series of insertions into the same small region of the space, BB help to significantly reduce the number of rebuilds and thus greatly improve the performance of writes with only minimal influence on query performance though a locally slightly deeper tree.

Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.16

BB help to keep the inner search tree (IST) of the BB-Tree stable over long periods of data changes, which enables an adaptation of the inner nodes to cache lines, the basic unit of data transfers between main memory and on-die CPU caches. We always choose k depending on how many delimiter values fit into one cache line to improve cache line utilization. For instance, when implementing delimiter values with four-byte floats and running on a machine with 64-byte cache lines, k is set to 17. Furthermore, we store the inner nodes of the BB-Tree in a flat and static array to avoid pointer chasing during search, to decrease random access patterns, and thus to reduce cache misses, especially at the last cache level. Typically, such an optimization either makes the index structure completely static [15, 27] or creates the need to manage delta stores [19, 23]. In contrast, BB-Trees manage all changes in-place and also can, due to the separation between the IST and the leaves and due to the concept of BB, manage a large number of updates without index rebuilds. Additionally, eliminating pointers improves space efficiency.

We furthermore describe and evaluate a special technique for the parallelization of MDIS queries, which effectively avoids complicated data partitioning. In the parallel range query operator, search queries are evaluated by first navigating the IST to determine all buckets that may hold matching data objects. This step is performed by a single thread as the tree, due to its high fan out, is quite low even for very large data sets. In the next step, which strongly dominates the runtime of queries, all qualifying BB are scanned in parallel. As a result, the performance of the parallel range query operator scales with the number of physical cores.

In a comprehensive evaluation, we compared the BB-Tree to sequential and parallel scans and to four other MDIS, namely the recent PH-tree [35], and main-memory adapted variants of the R*-tree [12], the kd-tree [4], and the VA-file [34]. We used different real and synthetic data sets of different sizes with dimensionalities between three and 100. We evaluated complete-match and partial-match range and exact-match queries, and also considered mixed read/write workloads. The BB-Tree is the fastest method for range queries up to a selectivity of around 20% after which it is only outperformed by a scan. For exact-match queries, the BB-Tree is almost as fast as the best point access method, the PH-tree; for more than ten dimensions it even shows a superior performance. It is the fastest MDIS regarding insertions and the overall fastest method regarding deletions. Its performance is virtually unaffected by the dimensionality of the data. The BB-Tree has the best space efficiency among all MDIS, an important property for in-memory data structures.

A preliminary version of this work will appear in [29]. Here, we extend [29] by dynamic updates, parallel execution of range queries and provide extended experiments.

2 RELATED WORK

We structure our discussion of related work into two parts: (1) Main-memory indexing and (2) multidimensional indexing.

Main-Memory Indexing. The recent focus on main-memory database systems led to the development of several main-memory index structures. A popular example is the adaptive radix tree (ART) [18], designed for efficient execution of exact-match queries especially over longer keys. However, ART does not support multidimensional data and executing range queries requires a costly traversal of its radix tree. The cache-sensitive skip list [30] is a main-memory index focussing on range queries. It uses a CPU-friendly data layout aligned with the sizes of cache lines, a technique that was previously suggested by other one-dimensional index structures, such as FAST [15] or the KISS-Tree [17]. Schlegel et al. [27] showed how to linearize k-ary search trees in a read-only in-memory setting, a technique we also use for the BB-Tree. We introduced bubble buckets to handle updates efficiently. None of the aforementioned index structures is able to index multidimensional data for partial-match range queries.

Multidimensional Indexing. MDIS have been researched for decades, leading to a multitude of different methods [10].

One of the most popular MDIS is the kd-tree [4], which organizes multidimensional point objects in a binary search tree by splitting the data space at each node using one of the dimensions as delimiter. It is integrated into several mature database systems, e. g., PostgreSQL[1]. K-D-B-trees [25] combine the concepts of B-trees [2] and kd-trees to optimize I/O behaviour. Quadtrees [9] are similar to kd-trees, but split the space in all dimensions at each node, which is less efficient for high dimensionalities. The Vector Approximation-file (VA-file) [34] is a mixture between an MDIS and a sequential scan that divides the space into cells of equal size using hash functions to allow for efficient pruning. All of these approaches were originally developed for disk-based data storage but can be adapted to main-memory settings [28]. Only the K-D-B-tree keeps its structure balanced when data changes, whereas the VA-file is an essentially immutable index. A recent main-memory based MDIS is the PH-tree [35], which integrates the concepts of PATRICIA-tries and hypercubes.

The R-tree [12] is probably the most prominent access method for handling spatially extended objects, but is also frequently used for storing point objects [14]. It uses minimum bounding rectangles (MBR) to represent all objects belonging to a certain subtree. These MBR are used for pruning. The R*-tree [3] is an R-tree variant that improves partitioning by aggressively reinserting data objects leading to a more efficient search performance. It is employed by several database systems to manage spatial data, e. g., SQLite[2]. PR-trees [1] optimize I/O in disk-based systems, and X-trees [5] target data of very high dimensionality. The CR-tree [16] is an R-tree variant that compresses inner nodes to pack more entries into MBR, which increases space and cache efficiency. Recently, Qi et al. [24] proposed a novel R-tree packing technique that provides asymptotically optimal I/O search complexity. However, their technique is restricted to static data and requires a complete reconstruction of the index with every update. Accordingly, the latter three approaches are not directly relevant for our work.

There also exist a number of interesting works, which are further away from our own research. Wang et al. [32, 33] exploit the characteristics of observational data, e. g., immutability, continuous dimension values, and append-only insertions, to achieve high query efficiency. In contrast, the BB-Tree is a general-purpose MDIS that supports updates in any order. ELF [6] executes multi-column selection predicates on in-memory data, but requires delta stores to handle updates.

3 THE BB-TREE INDEX STRUCTURE

In a nutshell, a BB-Tree is a main-memory optimized MDIS for point data. It combines the pruning power of an *almost-balanced* k-ary search tree with the efficiency of scans in main memory.

[1] https://www.postgresql.org/docs/9.6/static/spgist.html
[2] https://sqlite.org/rtree.html

Notation	Description
n	Size of the data set.
m	Dimensionality of the data set.
h	Tree height.
k	Inner nodes split the space into k subparts according to a delimiter dimension and $k-1$ delimiter values.
t	Number of available hardware threads.
B_{match}	Number of bubble buckets that need to be scanned to evaluate a certain range query.

Parameter	Description
b_{max}	Capacity of a regular bubble bucket.
$R_{samples}$	When reorganizing the inner search tree, we use $R_{samples}\%$ of all data as samples.

Table 1: Frequently used notations and input parameters.

The inner search tree (IST) is linearized and stored in a cache-optimized yet immutable array. Data objects are stored in special leaf nodes, the bubble buckets (BB), which are able to digest a large number of insertions or deletions without affecting the IST and without hurting the tree balance considerably. Nevertheless, in case of long series of hammered inserts or deletions, the BB-Tree must be rebuilt to keep its balance. In this case, the structure of the novel tree is determined using random sampling. In the following, we describe the different components and techniques of the BB-Tree in detail; Table 1 summarizes our notation.

3.1 Data Organization

A BB-Tree consists of two components: A k-ary search tree and a set of bubble buckets. Inner nodes of the search tree recursively split the data space into k disjoint subsets according to a delimiter dimension and $k-1$ delimiter values. All data are kept in BB, which initially hold up to b_{max} m-dimensional data objects, but can dynamically expand and shrink to cope with varying number of objects in the region they represent. When searching in a BB-Tree, the inner nodes are navigated to reduce the data space. Once all certainly irrelevant regions (or subtrees) have been pruned, the remaining BB are scanned to determine the true query results.

Inner search tree. The entire IST is implemented as a single, immutable array. This has several advantages: (1) Cache lines are the basic unit for transferring data between main memory and CPU caches. By choosing an appropriate value for k, the BB-Tree tailors its inner nodes to the individual cache line size of the CPU, which increases cache line utilization and reduces the amount of data transferred through the cache hierarchy. (2) Using a single dense array for representing a balanced IST makes pointers superfluous. Array indexes of child nodes are calculated in constant time based on the current tree level and the fan out k. The BB-Tree linearizes inner nodes in a breadth-first order, which reduces memory pressure and increases cache efficiency during traversals. (3) By using an array representation, searching a specific delimiter value within an inner node (as necessary during searching) can be efficiently implemented using a binary search, which requires only $log_2(k-1)$ instead of $(k-1)$ comparisons. Note that all these accesses occur within a single cache line, which means that they do not produce any cache miss.

Leaf nodes. All data objects are stored in bubble buckets. The elasticity property of BB is described in Section 3.2. For now, we assume that every BB has a maximum capacity of b_{max} objects. b_{max} is an important parameter of the BB-Tree as it determines, at query time, the balance between time spent in tree searching, resulting in pruning, and time spent in scanning, producing the query results. A high value leads to large leaf nodes storing more objects, which in turn requires less inner nodes and thus a less deep tree. Such a structure is preferable for less selective query workloads: More work is put on scans where the comparison of the data objects with the query lead to many matches, whereas less time is spent in pruning which, for less selective queries, is not effective anyway. In contrast, a lower BB capacity results in smaller leaf nodes and a deeper tree structure, which is beneficial for highly selective queries as more time is invested in successful pruning and less in scans producing almost no matches.

Delimiter values. For a good search performance, it is crucial that delimiters allow to prune subtrees and non-relevant BB as early as possible. When being rebuilt, BB-Trees choose their delimiter dimensions in the order of the number of distinct values of a dimension, moving dimensions with a high cardinality and thus a presumably higher pruning power to the top. If a dimension has more than k different values, delimiter values are determined such that each subtree features a roughly equal number of objects. If the number of inner node levels, h, is smaller than the dimensionality of the data set, m, BB-Trees thus omit the dimensions with the smallest cardinalities in the IST. Note that this scenario is quite frequent due to the high fan out of the tree which makes the BB-Tree rather flat even for very large data sets. As an example, assume a BB-Tree over one Billion objects, a fan out of $k = 17$, a b_{max} value of 1,000, and a fill degree of 50%. Adressing the resulting two million BB requires only six IST levels. Thus, low-cardinality dimensions, which are anyway problematic in terms of pruning power, do not clutter the tree. On the other hand, if h is larger than m, we employ dimensions multiple times as delimiters in a round-robin fashion. This scenario occurs especially for data sets with a dimensionality between two and four (depending on the number of objects). A special case occurs when low-cardinality dimensions are used as delimiters (see Section 3.5).

Example. Figure 1 illustrates a BB-Tree with $k = 3$, $h = 2$ (two tree levels), and nine BB managing three-dimensional data objects (buckets three to six are not displayed). Each (regular) BB can hold up to $b_{max} = 4$ data objects. Individual data objects are identified by tids. At the first level, the shown BB-Tree splits the data space into $k = 3$ partitions according to the first dimension. All data objects having a value of three or less in this dimension are held in the left subtree. All data objects having a value of seven or less, but larger than three, in this dimension are held in the middle subtree; all other data objects can be found in the right subtree. At the next level, the data space is recursively split according to the second dimension. Note that this example uses two dimensions as delimiter, although $m = 3$. Given a fan out of $k = 3$, two tree levels are sufficient for distinguishing between nine BB. Figure 2 illustrates the linearization of the IST. We link the linearized IST with the corresponding BB by maintaining an array of pointers, where entry i references the i-th BB.

SIMD. Although processing inner nodes with Single Instruction Multiple Data (SIMD) instructions sounds promising at first glance, especially because the tree is linearized and packed into a dense array, we were not able to obtain any performance benefits

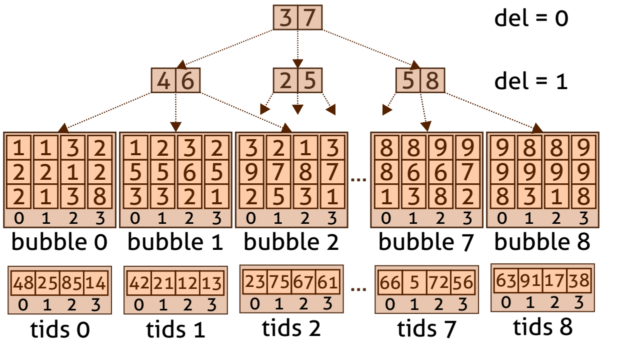


Figure 1: A BB-Tree ($k = 3$, $b_{max} = 4$) of height $h = 2$ managing $n = 36$ data objects of dimensionality $m = 3$; buckets three to six are omitted.

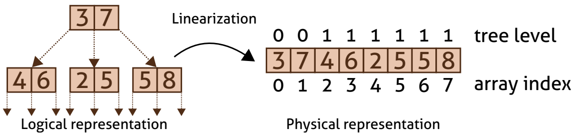


Figure 2: Linearized storage of the inner search tree.

through SIMD parallelism. Compared to a binary search, scanning inner nodes with SIMD instructions does not save many comparisons yet incurs overhead. For instance, when employing 16 delimiter values, which perfectly fit into one 64-byte cache line, a binary search requires $log_2(16) = 4$ comparisons, while a SIMD search on 256-bit registers needs two comparisons (or four comparisons if only 128-bit SIMD registers are available). These small savings are outweighed by the overhead induced by SIMD scans, especially data transfers between regular and vector registers [7], and the necessary scalar evaluation of the results of SIMD instructions.

3.2 Bubble Buckets

Until now, we described the BB-Tree as a static index structure and omitted the treatment of overflowing or underflowing leaf buckets. We lift this restriction and describe two techniques to cope with changing data, namely bubble buckets (this section) and index rebuilds (next section).

All leaf nodes are implemented as elastic bubble buckets. There exist two types of BB: A regular BB is implemented as a C++ *std::vector*, which is a dynamically growing and shrinking array, and takes inserts up to its maximum capacity b_{max}. In contrast, a super BB locally adds a further level to the tree. It consists of an inner node and a set of k regular nodes. The inner node holds a delimiter dimension and a set of delimiter values. As usual, super BB employ the dimension that has the largest number of distinct values as delimiter, and the $k - 1$ delimiter values are chosen such that the data objects are as evenly distributed as possible among the k regular child BB. Super BB morph into regular BB upon underflow, and regular BB morph into super BB upon overflow.

Inserts. The complete procedure for inserting objects is as follows: We first traverse over the inner nodes to determine the bucket that is responsible for the new object. If the chosen BB is a regular BB and has free space, we insert the object and are done. If there is no free space, we morph the BB into a super BB, and insert the data object; this also happens when the chosen BB already is a super BB. To insert into a super BB, we first check whether the super BB currently contains less than $k * b_{max}$ objects. If this is the case, we determine the appropriate child BB, which must be a regular BB, and insert the object; otherwise we reorganize the index.

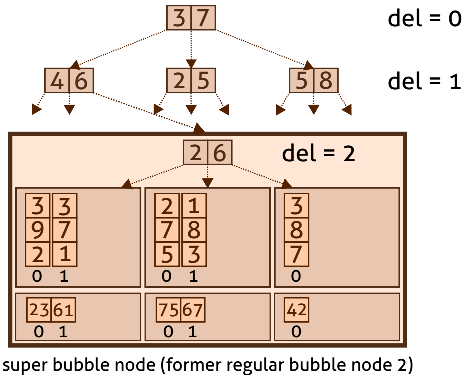


Figure 3: Assuming that a new data object (3 8 7) with tid 42 gets inserted into the BB-Tree from Figure 1, the second BB morphs into a super BB that consists of k regular nodes; dimension two is employed as delimiter.

BB can thus accommodate up to $k * b_{max}$ inserts into the same region before the index needs a rebuild. If objects are deleted during insert-heavy workloads, this period gets even longer. Within this time, the IST of the BB-Tree is stable, and the local depth is increased by at most one. However, to keep the algorithms simple we currently do not balance the size of the child nodes of a super BB, which, in theory, could lead to cases where all inserts accumulate in one child node. This would for instance happen when objects of an one-dimensional data set are inserted in a certain sort order; for such situations, other index structures are more appropriate, such as [18].

Deletes. When deleting an object, we first search the IST to determine the responsible BB and delete the object there. In the case of a regular BB, no further processing is performed. This implies that a BB-Tree may have empty leaf buckets; however, due to the dynamic size of their implementation, the memory overhead is minimal. We nevertheless rebuild the index when more than 10% of all BB are empty to get rid of superfluous inner nodes. If we delete from a child of a super BB, this bucket checks the total number of objects it contains and morphs into a regular BB in case the number is smaller than $p * b_{max}$. In the default setting, we set $p = 0.5$ to prevent pathological cases of constantly morphing BB when the b_{max}-th object is inserted and deleted iteratively.

Example. Consider again the BB-Tree from Figure 1. When we insert a new data object (3 8 7), the second bucket overflows and morphs into the super BB shown in Figure 3. Here, the super BB uses the third dimension as delimiter.

3.3 Building and Reorganizing a BB-Tree

A BB-Tree is initialized with one regular BB. After b_{max} objects have been inserted, this regular BB morphs into a super BB. With more inserts, this super BB eventually overflows, triggering a rebuild of the index. All operations, except for the very first, operate on a BB-Tree that was the result of an index rebuild.

Such a rebuild consists of four steps. First, we determine how many regular BB are needed to hold the current data, while leaving capacity for new inserts. From this number, we also derive the necessary number of levels of the IST. By default, we set the number of BB to $n/(10\% * b_{max})$ allowing each node to ingest further $90\% * b_{max}$ data objects until it overflows. This parameter may be changed if the expected workload consists of many inserts (lower value, less rebuilds, deeper tree) or few inserts (higher value, less deep tree). Second, we randomly sample $R_{samples} * n$ data objects as representatives of the whole data set. By scanning the sample data, we estimate the cardinality of each dimension. Dimensions are sorted by cardinality and assigned to the h IST levels in descending order. Third, we recursively determine the delimiter values for the inner nodes. Using the sample data, we compute an equi-depth histogram with k buckets, reflecting the distribution of the dimension values of the current level. Using that histogram, we obtain $k-1$ delimiter values such that the data are divided into k partitions of rougly equal size. In the case of a delimiter value, which occurs with a much higher frequency than the others, the derived partitions will be of unequal size. Clearly, this procedure fails for low-cardinality dimensions containing less than k distinct values. In the last step, all objects are inserted into their respective BB.

Obviously, index reorganization is an expensive operation. A random sample must be determined which is scanned multiple times, a new IST is constructed, and data objects must be moved to new locations. We chose pragmatic and fast methods for these steps, which come at certain drawbacks. First, equally splitting a subtree by one dimension is not always possible, which, in the worst case, may lead to an unbalanced BB-Tree (see Section 3.4), where subtrees at the same depth contain an unequal number of data objects. Second, we globally assign dimensions to tree levels, which again can lead to imbalances when dimensions are strongly correlated. Third, we compute the IST structure only on a sample. If the sample is small, the tree is found quickly yet might not optimally represent the data. Contrary, if the sample is large, building the tree needs more time yet probably leads to a better tree structure. We make two notes regarding these issues. First, they are shared by most other updateable MDIS. For instance, the structure of kd-trees strongly depends on the order of the insertions. The K-D-B-tree turns kd-trees into balanced search trees, but at the price of complicated and slow update operations. Second, though we cannot give formal guarantees, for the data sets we used in our evaluation, we never observed any notable imbalance. We are thus confident that unbalanced BB-Trees, which are possible in theory, remain very rare in practice.

3.4 Search Algorithms

BB-Trees focus on partial- and complete-match range queries, but also support exact-match queries.

All search queries have in common that they first exploit the linearized inner nodes to efficiently find those BB that may hold data objects relevant for the search query while pruning all others. This step is followed by sequential scans over all candidate BB to determine the data objects matching the query. Evaluation of search queries may have to follow multiple paths through the tree: Partial-match range queries must consider multiple paths whenever a node splits on a dimension which is not part of the query. Complete-match range queries have to consider multiple paths when the range covers more than one subtree. Even exact-match queries need to consider multiple paths when a low-cardinality dimension (with less than k distinct values) is used as delimiter.

Assuming the usual case, where the navigation of the IST results in only one candidate BB, exact-match queries have a complexity of $O(h * log(k) + b_{max} * m)$: They first perform h times a binary search within inner nodes and eventually scan one BB holding up to b_{max} objects, where a comparison between the query and a data object requires m value comparisons. The tree height (h) depends on the number of stored data objects, the BB capacity and the fan out: $h = log_k(n/b_{max})$. Note that the tree height is increased by one in case the search leads to a super BB. The complexity of exact-match queries is dominated by the scans of leaf nodes, calling for small values of b_{max}. On the other hand, every change of a level during IST traversal may result in a cache miss, making these operations more costly in practice.

Note that the worst-case complexity of the BB-Tree is linear in n. First, this is trivially the case when queries select all indexed objects. Second, this case occurs when less than B_{max} objects are indexed, as these are all stored in one bucket preventing pruning. Third, the worst case also applies, when BB-Trees consist of only one super BB and most data objects are inserted into the same child BB due to a inaptly-chosen delimiter dimension. However, once the super BB overflows, the data objects are distributed to multiple regular BB and the search complexity (for queries requiring only one BB) converges to the formulas given above.

For less selective range queries, scans are more attractive as more of their comparisons actually lead to matches, without any tree navigation in-between. However, determining an optimal b_{max} value would only be possible if all queries had the same, a-priori known selectivity across the entire data space, an assumption that is rather impractical. In practice, every setting of b_{max} implements an expectation on the average selectivities of queries in the future workload. In our evaluation, we will show that our default value leads to a performance that is almost on-par with MDIS specialized in exact-match queries while clearly outperforming all competitors for MDRQ.

3.5 Low-Cardinality Dimensions

We consider a dimension of a data set to have a low cardinality when its number of distinct values is smaller than k. Low-cardinality dimensions are common in real-world data sets and challenge MDIS because they make partitioning generally hard and equal partitions impossible. The problem is less severe for the BB-Tree, as it sorts delimiter dimensions by the number of distinct values, which usually keeps low-cardinality dimensions completely out of the IST. However, if a data set contains less than h dimensions and these dimensions have low cardinalities, a low-cardinality dimension will be chosen as delimiter dimension of an inner node, making it impossible to find distinct delimiter values such that the data is split into k subparts of equal size. In the worst case, where a delimiter dimension features only one distinct value, the IST loses all its pruning power and searching the BB-Tree degenerates to a sequential scan over all leaf nodes.

4 PARALLEL EVALUATION OF RANGE QUERIES

The parallel range query operator uses multiple threads to execute search queries and consists of two phases.

The first phase navigates the inner nodes with a single thread to determine the candidate BB. Traversing over the linearized inner nodes in parallel is complicated, as an optimal scheme would require solving a non-trivial load balancing problem due to different pruning effects in different subtrees. At the same

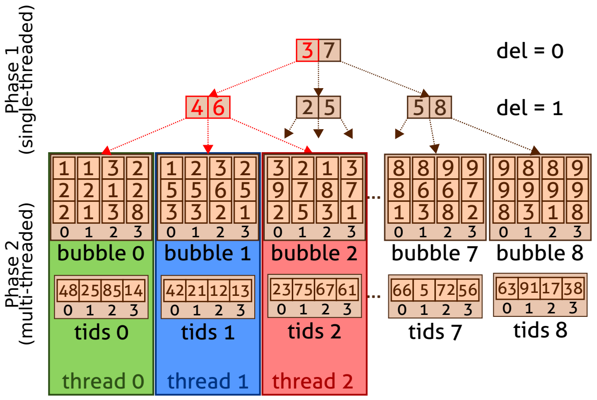


Figure 4: Parallel evaluation of an exemplary range query defined by lower boundary $[1, 0, 3]$ **and upper boundary** $[3, 7, 6]$**.**

time, there is only little to gain as the tree usually is rather flat due to its high fan out.

In the second phase, all candidate BB are scanned in parallel. Let B_{match} denote the number of candidate BB that have been determined by the first phase (super BB are considered as multiple BB). If $B_{match} \geq t$, we divide the candidate BB into t partitions of size B_{match}/t. Each partition is processed by a distinct thread using the same algorithm as in the regular, single-threaded BB-Tree. Hence, if $B_{match} = t$, we obtain the perfect degree of parallelism. If $B_{match} < t$, some threads would remain idle when sticking to the one-thread-per-bucket assignment. In this case, we assign multiple threads to single BB allowing to divide the data objects of one BB into partitions and scan these partitions in parallel.

Example. Figure 4 illustrates the parallel evaluation of a range query in a BB-Tree. The shown search query retrieves all data objects that match the lower boundary $[1, 0, 3]$ and the upper boundary $[3, 7, 6]$. The first, single-threaded phase of the query execution determines that the first three BB may hold data objects matching the range boundaries (the search path is marked in red). In the second, multi-threaded phase, these BB are searched concurrently with one CPU thread per bucket assuming that our imaginary machine features three threads ($B_{match} = t$).

5 EVALUATION

In a comprehensive evaluation, we compare the BB-Tree with state-of-the-art approaches to general-purpose indexing of multidimensional data by executing synthetic and real-world query workloads over synthetic and real-world data sets. Specifically, we aim to answer the following questions: (1) Does the performance of the BB-Tree depend on data set- or workload-specific characteristics, e. g., data dimensionality, data skew, or query selectivity (see Sections 5.4 to 5.7)? (2) How does the BB-Tree perform on mixed workloads that contain both reads and writes (see Section 5.9)? (3) What is the effect of parallelization (see Section 5.10)? (4) How efficient does the BB-Tree utilize memory space (see Section 5.11)?

5.1 Experimental Setup

Hardware. We executed all experiments on a server equipped with two Intel Xeon E5-2620 CPUs (2 GHz clock rate, 64-byte cache lines, six cores, 12 hardware threads) and 32 GB of RAM. In total, the machine features 12 cores and 24 hardware threads. Most experiments are single-threaded, some experiments investigate the parallel range query operator and therefore make use of multiple threads.

Methodology. In our evaluation, the competitors are completely kept in main memory. All data sets are inserted in random order. All experiments measure the average execution time of an operation, e. g., range query. We run experiments three times and present the arithmetic mean.

Competitors. We compare the BB-Tree with multiple approaches to general-purpose multidimensional indexing: the kd-tree [4], the PH-tree [35], the R*-tree [3], the VA-file [34] and the sequential scan [28]. Section 2 provides a brief description of the competitors; for more details we refer the interested reader to the original papers or surveys, like [10], or benchmarks, like [28]. For the R*-tree, we used an open-source, main memory implementation (https://libspatialindex.github.io/) and relied on the default configuration, but slightly adjusted the node capacities such that nodes are aligned to cache lines. For the PH-tree, which is a main-memory MDIS by design, we used a publicly available implementation shared by the authors (https://github.com/tzaeschke/phtree-1). For the kd-tree, the VA-file and the sequential scan, we used our own implementations based on the original publications, but adapted them to main-memory storage following techniques described in [28]. Most contestants, including the BB-Tree, use 32-bit floating-point values to manage dimension data. The R*-tree implementation uses 64-bit floating-point values; the PH-tree implementation uses 64-bit integer values. We evaluated the BB-Tree with $k = 17$, because $k - 1 = 16$ four-byte floating-point values fit into one cache line of the evaluation machine. Based on the observations described in Section 5.3, we empirically set $b_{max} = 2,500$. For reorganization, we use $R_{samples} = 10\%$ of all objects as samples to estimate the current data distribution.

Software. All software was implemented in C++ and was compiled with GCC using optimization flag *-O3*. We measured hardware performance counters with PAPI (http://icl.cs.utk.edu/papi/) and space consumption with valgrind (http://valgrind.org/). For the parallel range query operator, we used an open-source thread pool library (https://github.com/vit-vit/CTPL) to enable the reuse of POSIX threads. Our implementation of the BB-Tree is freely available (https://www2.informatik.hu-berlin.de/~sprengsz/bb-tree/).

5.2 Data Sets and Workloads

We evaluate the competitors on four data sets. Table 2 provides the number of data objects (n), the dimensionality (m), the number of distinct values per dimension (for UNIFORM and CLUST, we provide averages over all dimensions), and the raw size of each data set. We primarily use synthetic workloads. Unless noted otherwise, we generate synthetic range queries by randomly choosing two objects from the data set and, for each dimension, we use the smaller (larger) value of both objects as lower (upper) boundary. For GENOMIC, we execute a realistic workload, the Genomic Multidimensional Range Query Benchmark (GMRQB) [28], consisting of eight partial- and complete-match MDRQ templates of varying selectivity.

Data Set	n	m	Distinct Values per Dimension	Raw Size
UNIFORM	10k	5	10k (avg)	0.19MB
	100k	5	95k (avg)	1.91MB
	1M	5	632k (avg)	19.07MB
	10M	5-100	1M (avg)	190.74MB-3,814.7MB
	10M	5	4-64	190.74MB
CLUST	10k	5	10k (avg)	0.19MB
	100k	5	95k (avg)	1.91MB
	1M	5	632k (avg)	19.07MB
	10M	5	1M (avg)	190.74MB
POWER	10k	3	10k; 1k; 1k	0.11MB
	100k	3	100k; 2k; 2k	1.14MB
	1M	3	1M; 4k; 5k	11.44MB
	10M	3	10M; 6k; 8k	114.44MB
GENOMIC	10k-10M	19	1-63,883	0.72MB-724.79MB

Table 2: Data sets used in our experiments.

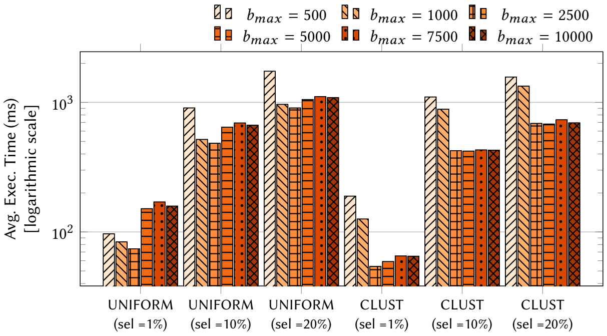


Figure 5: Performance of BB-Trees with different BB capacities (B_{max}) when executing range queries with varying selectivities (1%, 10%, and 20%) (n=10M, m=5, UNIFORM/CLUST).

Uniform Data (UNIFORM). Synthetic data facilitates experiments with arbitrary data set sizes, dimensionalities and query selectivities. We generate uniformly distributed data objects within the domain $[0, 1]$.

Clustered Data (CLUST). The five-dimensional data set CLUST features up to 20 clusters. We used a generator provided by Müller et al. [22] to generate CLUST within the domain $[0, 1]$. Within clusters, data are uniformly distributed.

Sensor Data (POWER). POWER is obtained from the DEBS 2012 challenge (http://debs.org/?p=38) and resembles real-world sensor data of hi-tech manufacturing equipment. As in previous studies [28, 33], we index three dimensions.

Genomic Data (GENOMIC). GENOMIC consists of real-world genomic variant data obtained from the 1000 Genomes Project [31]. We transformed the raw data, originally provided as text files, into 19-dimensional data objects, which can be indexed by the competitors (some attributes were provided as text and had to be converted into floating-point values). As shown in Table 2, the dimensions of GENOMIC have a highly varying cardinality, ranging from one to 63,883 distinct values. We previously defined the Genomic Multidimensional Range Query Benchmark (GMRQB) in [28], which consists of eight realistic partial- and complete-match MDRQ templates restricting between two and 19 dimensions of the data space. The templates are instantiated with concrete values obtained from the 1000 Genomes Project data and have an average selectivity between 10.76% and 0.00001%.

5.3 Impact of Bubble Bucket Capacities

The capacity of BB, as defined by b_{max}, controls the ratio between index probing (navigation of IST) and scanning (evaluation of leaf nodes) when searching in BB-Trees. While small BB put more work on index probing, large BB increase the relative time spent on scanning. As shown in previous work [28, 34], index probing is beneficial for highly selective queries and scanning is superior for less selective workloads.

We study the impact of b_{max} on the performance of range queries with varying selectivities (1%, 10%, 20%) when applied to ten million five-dimensional objects from UNIFORM and CLUST. Our goal is to find a pragmatic configuration providing a robust performance for a wide range of query selectivities and data distributions. Figure 5 shows the results.

For uniform distributions, this experiment confirms that small (large) capacities are beneficial for highly (less) selective queries. While small BB capacities are more efficient than large BB capacities for queries with an average selectivity of 1%, they become less efficient with increasing query selectivity (10% and 20%). For the selectivies considered here, BB holding up to 2,500 objects provide the best performance.

Clustered data lead to a less optimal partitioning, which lessens the pruning power of the IST and puts more work on scanning. As a result, small BB capacities become less efficient, even for small selectivities, because less BB can be pruned. Also for clustered data, BB with a maximum capacity of 2,500 objects provide either the best performance or are on a par with other configurations.

Taking the results of this experiment into account, we set $b_{max} = 2,500$ for all following experiments.

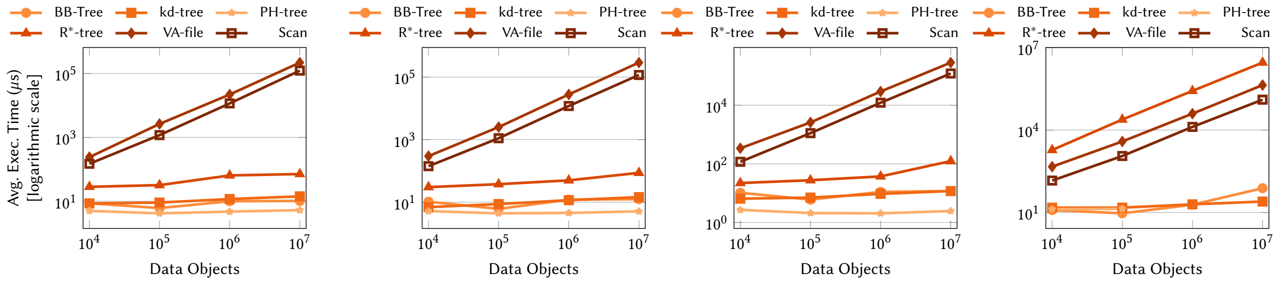
5.4 Exact-Match Queries

Figure 6 shows the average execution time of exact-match queries for the four data sets depending on the number of data objects. We execute n exact-match queries on the contestants given that n denotes the data set size. Each exact-match query retrieves a randomly-chosen, existing data object. To manage 10^4 objects, the BB-Tree does not need an IST, but employs one super BB consisting of $k = 17$ regular nodes ($b_{max} = 2,500$). In general, for exact-match queries, the performance of the BB-Tree is very similar to that of the kd-tree and the PH-tree. It clearly outperforms the R*-tree, the VA-file and the sequential scan for all data sets, often by multiple orders of magnitude. For the largest instance of GENOMIC (10^7 objects), inner nodes at the lowest tree level feature duplicate delimiter values, which require scanning multiple candidate BB and result in minor performance drops.

Typically, MDIS achieve a better exact-match query performance than sequential scans, because they can prune large parts of the data while scans need to consider all data. Although the BB-Tree needs to scan over data objects stored in BB it achieves a very competitive performance. The BB-Tree exploits the linearized inner nodes to effectively reduce the amount of data to consider for query evaluation.

5.5 Insertions and Deletions

Figure 7 presents the average time a contestant needs to ingest a data object. The shown results include the reorganizations of the BB-Tree; in a real system, we would advise to handle rebuilds in background jobs, which strongly increases insert performance. We do not insert entire data sets at once, but load object by



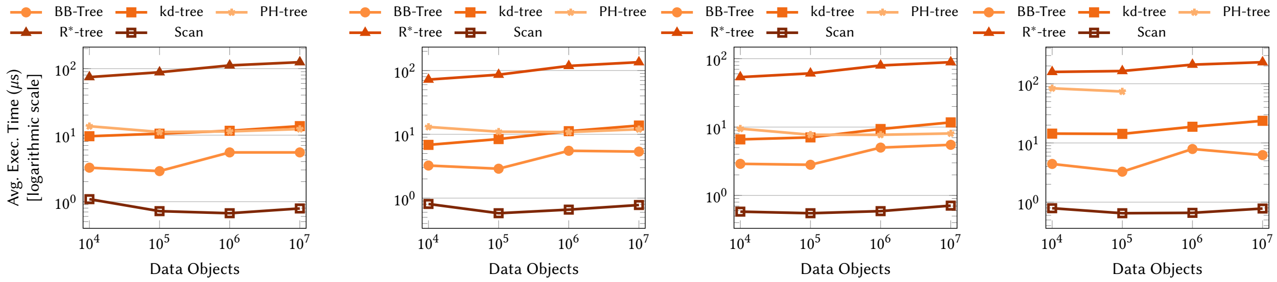
(a) UNIFORM (m=5)

(b) CLUST (m=5)

(c) POWER (m=3)

(d) GENOMIC (m=19)

Figure 6: Performance of exact-match queries on synthetic and real-world data depending on the number of data objects.



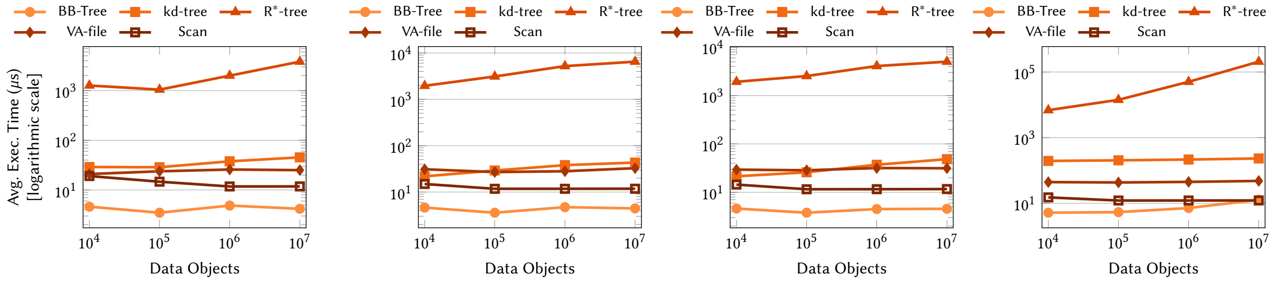
(a) UNIFORM (m=5)

(b) CLUST (m=5)

(c) POWER (m=3)

(d) GENOMIC (m=19)

Figure 7: Performance of insert operations on synthetic and real-world data depending on the number of data objects.



(a) UNIFORM (m=5)

(b) CLUST (m=5)

(c) POWER (m=3)

(d) GENOMIC (m=19)

Figure 8: Performance of delete operations on synthetic and real-world data depending on the number of data objects.

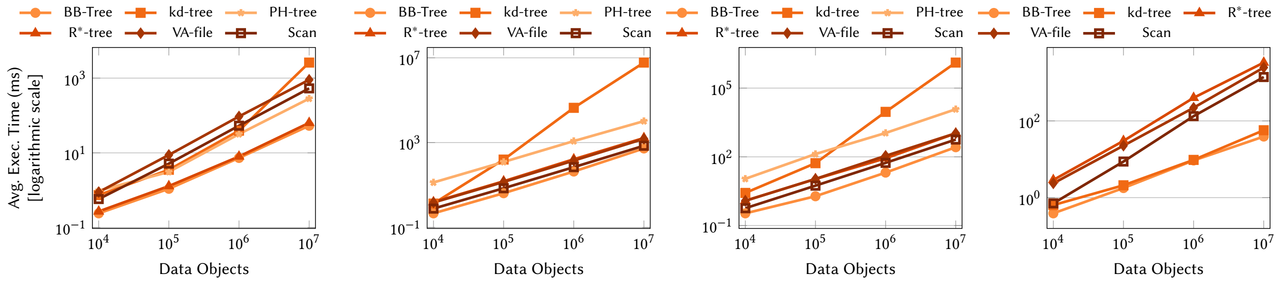
object. Thus, this experiment does not include the VA-file, which supports only bulk inserts, because it requires to know the data distribution beforehand. For instances of GENOMIC with more than 10^5 objects, the space needs of the PH-tree exceeded the available 32 GB of memory.

The scan achieves the highest insert performance, because it implements inserts by appending new data objects to a dynamic array and does not require to deal with node overflows, like the R*-tree. Notably, the BB-Tree shows a better insert performance than the kd-tree and the PH-tree and clearly outperforms the R*-tree. The concept of elastic BB effectively reduces the frequency of rebalancing operations. When dynamically inserting ten million data objects, regardless of the data set, the BB-Tree needed only three reorganizations, which took 6.55s on average (standard deviation σ = 8.75s). Smaller data sets require even less reorganizations.

Figure 8 shows the average time needed for deleting an object from the four data sets depending on data set size. The used implementation of the PH-tree did not provide a delete operator. The delete performance of the BB-Tree correlates with its exact-match query performance, because it first locates the to-be-deleted data object and then removes it from the corresponding BB. The BB-Tree outperforms all other competitors in deleting data objects, even scans, except for the largest instance of GENOMIC.

5.6 Range Queries

Figure 9 shows the average execution time of complete-match MDRQ that we generated by randomly choosing two objects from the data set. Depending on the data distribution, the obtained MDRQ objects have a varying average selectivity; UNIFORM: 0.4% (σ = 0.9%), CLUST: 19.8% (σ = 19.7%), POWER: 12.6% (σ = 13.1%), GENOMIC: 0.2% (σ = 0.2%). For CLUST, one range query may span multiple clusters, therefore average selectivities



(a) UNIFORM (m=5) **(b) CLUST (m=5)** **(c) POWER (m=3)** **(d) GENOMIC (m=19)**

Figure 9: Performance of synthetic complete-match range queries on synthetic and real-world data depending on the number of data objects. Due to the technique used to generate MDRQ (see Section 5.2), average selectivities are as follows: UNIFORM: 0.4% (σ = 0.9%), CLUST: 19.8% (σ = 19.7%), POWER: 12.6% (σ = 13.1%), GENOMIC: 0.2% (σ = 0.2%).

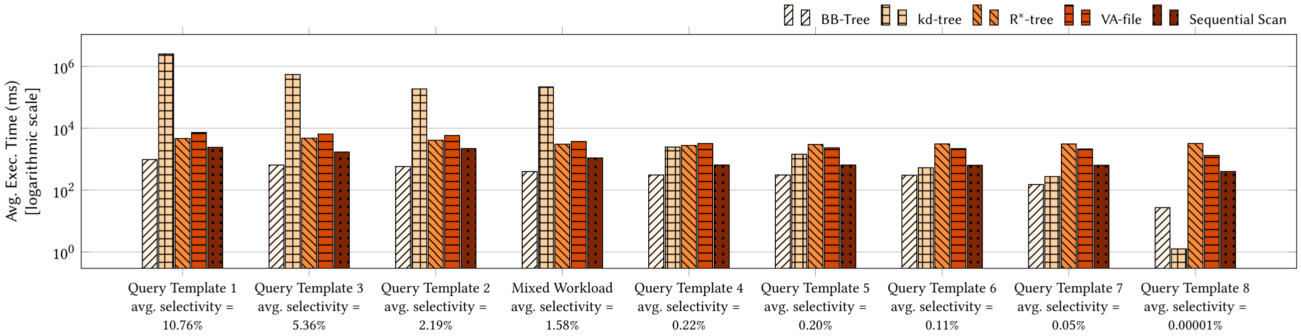


Figure 10: Performance of eight realistic MDRQ templates, including a mixed workload, from GMRQB; query templates are ordered by selectivity (n=10M, m=19, GENOMIC).

are higher than for UNIFORM although both data sets have an identical size and both are generated within $[0,1]$. The BB-Tree achieves the best overall performance and outperforms the other contestants, sometimes by up to three orders of magnitude. For UNIFORM, the R*-tree shows a performance similar to that of the BB-Tree. For GENOMIC, the kd-tree performs similar to the BB-Tree. We omit the PH-tree for all range query experiments on GENOMIC, because the implementation given by the authors crashed with failing C++ assertions.

Figure 10 presents the average execution time of the GMRQB on ten million data objects from GENOMIC. Note that most query templates, except query templates 7 and 8, have their first selection predicate in the second level of the BB-tree, which means that the GMRQB workload is rather unfavorable for our index. Nonetheless, BB-Trees consistently achieve the best performance for query templates 1-7, which are partial-match MDRQ querying 5.81 dimensions on average (σ = 4.11), and a mixed workload consisting of all query templates randomly mixed together. Only for query template 8, which resembles an exact-match query as it selects a single data object on average, they are beaten by kd-trees. For data of high(er) dimensionality, such as GENOMIC, R*-trees lose their pruning power and show a worse performance than scans.

Figure 11 shows the performance of range queries on ten million objects from UNIFORM depending on query selectivity. We omit the kd-tree because, compared to the other competitors, its execution time was orders of magnitude higher for queries selecting more than 1% of the data. The BB-Tree outperforms all other MDIS regardless of the query selectivity. It also beats the scan for queries with a selectivity of up to 20%. For less selective queries, the performance of the BB-Tree remains close to that of a scan. Furthermore, the BB-Tree achieves a very high cache efficiency, almost as good as that of a sequential scan, and follows most predicted branches leading to few pipeline flushes (see Table 3).

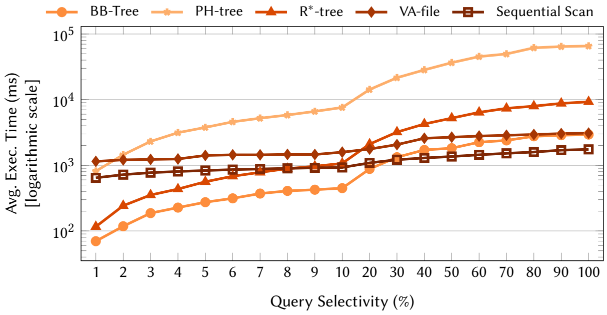


Figure 11: Performance of range queries depending on query selectivity; kd-tree is omitted as its performance decreases severely for less selective queries, strongly impairing the readability of the figure (n=10M, m=5, UNIFORM).

5.7 Impact of Dimensionality

We measure the performance of exact-match and complete-match range queries on ten million data objects from UNIFORM depending on data set dimensionality. We generate complete-match range queries with an average selectivity of 1% (σ = 0.7%). With a

	BB-Tree	kd-tree	PH-tree	R*-tree	VA-file	Scan
CPU Cycles	**164M**	8,306M	1,908M	252M	2,934M	1,582M
LLC Accesses	1.0M	824M	1.2M	2.5M	1.8M	**0.5M**
LLC Misses	0.7M	0.9M	0.8M	0.5M	1.6M	**0.3M**
TLB Misses	0.3M	1.0M	0.3M	0.3M	0.2M	**0.1M**
Branch Mispr.	**0.1M**	0.7M	3M	0.2M	10M	7M

Table 3: Performance counters per range query (1% selectivity;n=10M, m=5, UNIFORM).

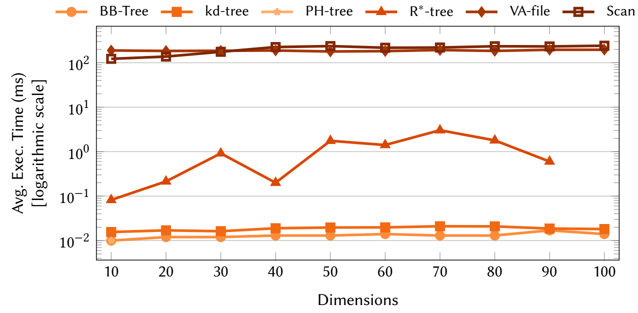


Figure 12: Performance of exact-match range queries (average selectivity = 1%, σ = 0.7%) depending on dimensionality (n=10M, UNIFORM).

growing dimensionality, this results in very low single-dimension selectivities posing serious challenges to MDIS because pruning becomes less useful. For instance, when running complete-match MDRQ with an overall selectivity of 1% on 100-dimensional uniformly distributed data, where dimensions are not correlated, single-dimension selectivities are approximately 95.50%, as $0.955^{100} \approx 0.01$.

Figure 12 shows the runtimes of exact-match queries for dimensionalities between ten and 100[3]. For such workloads, all methods except the R*-tree are mostly unaffected by the dimensionality of the data space. Similarly, Figure 13 shows the runtimes of complete-match MDRQ depending on the dimensionality. All methods show a performance degradation roughly proportional to the dimensionality of the data space, starting at a dimensionality of 20, because an increasing number of dimensions has to be compared when evaluating queries. The slow-down is more pronounced for lower dimensionalities.

We also executed workloads on instances of CLUST featuring five and ten dimensions (data not shown). All competitors behave very similar as for UNIFORM: Exact-match queries are almost unaffected by the dimensionality of the data space, whereas range queries degrade noteably.

5.8 Low-Cardinality Dimensions

Low-cardinality dimensions are challenging for BB-Trees because they make it impossible to find k different delimiter values, which limits the pruning power of the IST. We first study this effect using range queries applied to ten million five-dimensional data objects from UNIFORM with different moderately low cardinalties for all dimensions. Results are shown in Figure 14. At these cardinalities, none of the competitors is affected severely as the differences only correspond to the different query selectivities. Note that in the cases of eight and 16 distinct values per dimension, the data space includes duplicate data objects which is not supported by the PH-tree; therefore, we omit this method in this experiment.

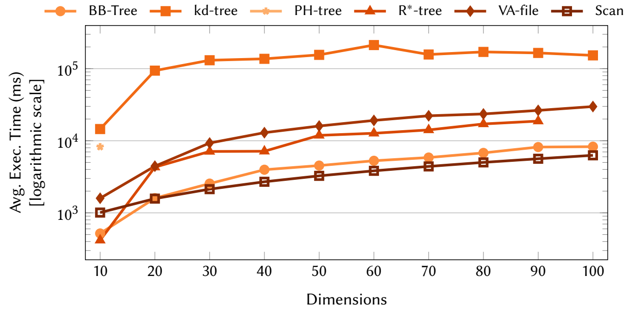


Figure 13: Performance of complete-match range queries (average selectivity = 1%, σ = 0.7%) depending on dimensionality (n=10M, UNIFORM).

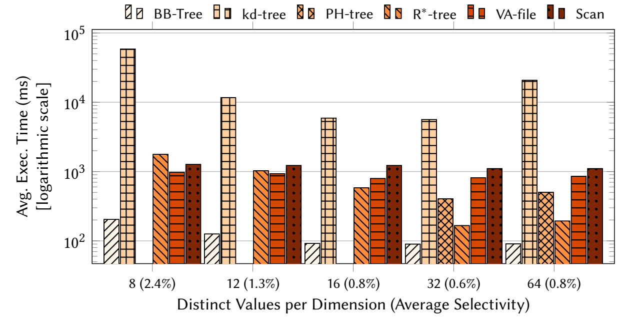


Figure 14: Performance of MDRQ with a varying selectivity depending on number of distinct values per dimension (n=10M, m=5, UNIFORM).

Next, we performed an experiment with extremely low cardinalities (between two and 12) yet used data of higher dimensionality. Figure 15 shows the performance of range queries with a selectivity of 0.00002% (σ = 0.0%), when applied to ten million 50-dimensional objects from UNIFORM. The PH-tree had to be omitted because it produced incorrect results. This experiment shows that the performance of most MDIS drops considerably with lower cardinalities, whereas scans and VA-files are much less effected. However, for such low cardinalities other index structures, like bitmaps [8], are probably a better choice anyway.

5.9 Mixed Workload

In most applications, MDIS are loaded in bulk before running large batches of search queries. Once built, inserts and deletes rarely happen. This experiment studies the contestants when running such workloads on data from GENOMIC. We use ten million data objects, of which we first insert 9,999,900[4]. Next, we run 100 inserts, 100 deletes, 2,800 exact-match queries and 7,000 range queries in random order. For inserts, we use objects, which were not bulk loaded. For exact-match queries and deletes, we randomly choose objects from the data set. This may result in queries asking for previously deleted data objects. For range queries, we

[3] Note that the space requirements of the PH-tree exceeded the available 32 GB of main memory for dimensionalities higher than ten. Similarly, the R*-tree ran out of space for 100 dimensions.

[4] For the VA-file, we insert all data objects at the beginning of the workload, because it only supports bulk inserts.

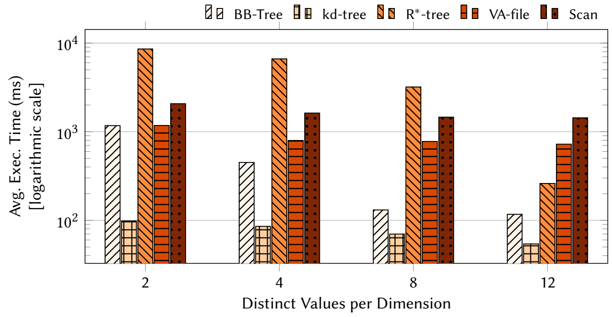


Figure 15: Performance of MDRQ with a selectivity of 0.00002% (σ = 0.0%) depending on number of distinct values per dimension; PH-tree is omitted (n=10M, m=50, UNIFORM).

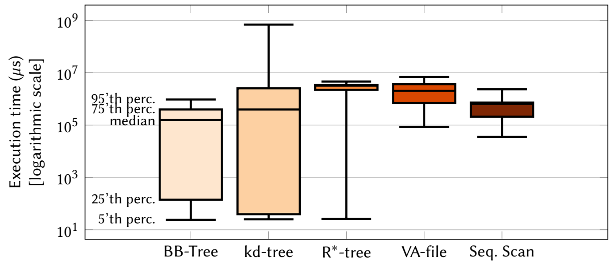


Figure 16: Execution times of single queries (inserts, deletes, exact-match and range queries) from a mixed workload in random order; bulk insert is not included; PH-tree ran out of memory (n=10M, m=19, GENOMIC).

	Bulk Insert (s)	Average/Minimum/Maximum exec. time (ms)
BB-Tree	54.7s	**262.66ms** / 0.005ms / **1,866.73ms**
kd-tree	236.7s	128,735.5ms / 0.011ms / 4,842,752ms
PH-tree		Ran out of memory.
R*-tree	2,316s	2,735.16ms / 0.008ms / 7,735.76ms
VA-file	38.7s	2,704.8ms / 0.004ms / 8,148.82ms
Seq. Scan	**7.8s**	809.83ms / **0.002ms** / 3,117.46ms

Table 4: (1) Total execution time of the bulk insert and (2) average, minimum and maximum execution time of the remaining queries of the mixed workload.

use the mixed workload from GMRQB (avg. sel.=1.58%, σ = 3.58%), which consists of all query templates randomly mixed together. Once again the PH-tree ran out of memory.

Figure 16 summarizes the runtimes. It does not include the bulk insert, because this would focus too much on inserts (ten million inserts vs. 9,900 search queries and deletions). For most contestants, insertions are the fastest operation, which would move all other operations out of the 95'th percentile. Table 4 shows the runtime of the bulk insert and summarizes the execution times of the remaining 10,000 queries. The BB-Tree achieves the highest performance in most cases. Only for the bulk insert, it is outperformed by the scan and the VA-file. The results show that the BB-Tree combines high search performance with fast inserts and deletes.

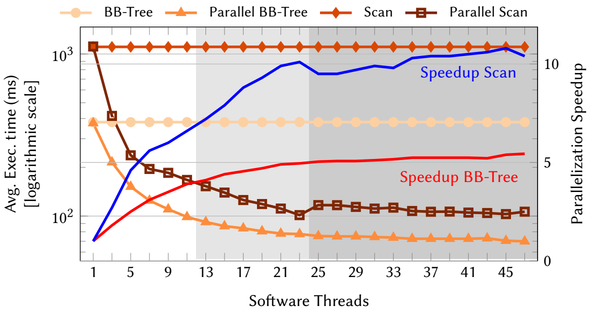


Figure 17: Performance of the mixed workload from GMRQB (avg. selectivity=1.58%) depending on the number of used software threads (n=10M, m=19, GENOMIC).

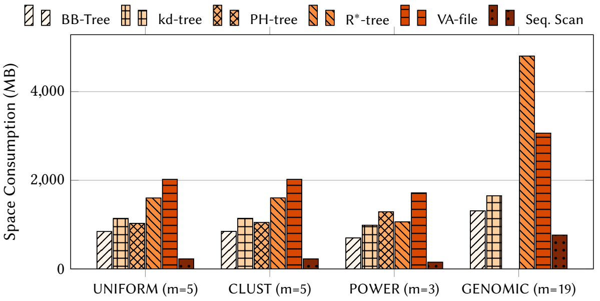


Figure 18: Space consumption of the competitors (n=10M).

5.10 Parallel Evaluation of Range Queries

Figure 17 shows the performance of the parallel range query operator when executing the mixed workload from GMRQB over ten million objects from GENOMIC depending on the number of used software threads. We compare results to a single-threaded BB-Tree, a single-threaded scan, and a parallel scan, which (1) divides the data objects into t partitions, (2) concurrently scans each partition with one thread, and (3) concatenates the results of the individual partitions. The performance of the parallel range query operator of the BB-Tree improves with the number of used threads up to a barrier established by the number of available physical cores (12 on our evaluation machine). Hyper-threading provides only few benefits for the mostly compute-bound BB-Tree, but is useful for memory-bound applications, like scans. Using moderately more threads than supported by the hardware (> 24), does neither provide benefits nor disadvantages. Scanning (up to 10.9X speed-up) benefits more from multi-threading than the BB-Tree (up to 5.5X speedup), because (a) the parallel scan leverages hyper-threading and (b) scan-based MDRQ can be completely parallelized while BB-Trees navigate the IST with a single thread. Nonetheless, BB-Trees outperform scans regardless of the number of used threads.

5.11 Space Consumption

Figure 18 shows the space consumption of the contestants when storing ten million data objects of the data sets used in the evaluation. For GENOMIC, the PH-tree required more than the available 32 GB of main memory. The BB-Tree achieves a high space efficiency, which is mainly enabled by the linearization of its inner nodes. Compared to the other MDIS, it requires the smallest index overhead over the scan.

6 CONCLUSIONS

We presented the BB-Tree as a fast and space-efficient means for storing and querying multidimensional data in main memory. It supports complete- and partial-match range queries, exact-match queries, and dynamic updates. We compared the BB-Tree with state-of-the-art MDIS using different synthetic and real-world workloads over different synthetic and real-world data sets with three to 100 dimensions. The BB-Tree beats all competitors in executing range queries up to a selectivity of 20%; for less selective queries it is only outperformed by a scan. It executes exact-match queries almost as fast as the best competitor, the PH-tree; for higher dimensionalities it even provides the best performance. The BB-Tree achieves the best insert and delete performance. We also presented a parallel variant that accelerates range queries almost linearly with the number of available CPU cores. Of course, BB-Trees are pure main-memory data structures; if data does not fit in memory, disk-based MDIS should be used like the original R*-Tree [12] or the original VA-File [34]. In future work, we intend to support nearest neighbor search and concurrent execution of search queries.

7 ACKNOWLEDGMENTS

Stefan Sprenger is funded by the Deutsche Forschungsgemeinschaft through graduate school SOAMED (GRK 1651).

REFERENCES

[1] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. 2008. The priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Trans. Algorithms* 4, 1 (2008), 9:1–9:30.

[2] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indices. *Acta Inf.* 1 (1972), 173–189.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *Proc. of the ACM SIGMOD International Conference on Management of Data* (1990).

[4] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* (1975).

[5] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. 1996. The X-tree : An Index Structure for High-Dimensional Data. *Proc. of the 22th International Conference on Very Large Data Bases* (1996).

[6] David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. 2017. Accelerating Multi-Column Selection Predicates in Main-Memory - The Elf Approach. In *33rd IEEE International Conference on Data Engineering*. 647–658.

[7] David Broneske and Martin Schäler. 2017. Single Instruction Multiple Data - Not Everything is a Nail for this Hammer. In *FADS@VLDB*.

[8] Chee Yong Chan and Yannis E. Ioannidis. 1998. Bitmap Index Design and Evaluation. In *Proc. of the ACM SIGMOD International Conference on Management of Data*. 355–366.

[9] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.* 4 (1974), 1–9.

[10] Volker Gaede and Oliver Günther. 1998. Multidimensional Access Methods. *ACM Comput. Surv.* 30, 2 (1998), 170–231.

[11] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Min. Knowl. Discov.* 1, 1 (1997), 29–53.

[12] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD, Proc. of Annual Meeting*. 47–57.

[13] Jörg Hakenberg, Wei-Yi Cheng, Philippe E. Thomas, Ying-Chih Wang, Andrew V. Uzilov, and Rong Chen. 2016. Integrating 400 million variants from 80,000 human samples with extensive annotations: towards a knowledge base to analyze disease cohorts. *BMC Bioinformatics* 17 (2016), 24.

[14] Kothuri Venkata Ravi Kanth, Siva Ravada, and Daniel Abugov. 2002. Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. *Proc. of the ACM SIGMOD International Conference on Management of Data* (2002).

[15] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. *Proc. of the ACM SIGMOD International Conference on Management of Data* (2010).

[16] Kihong Kim, Sang Kyun Cha, and Keunjoo Kwon. 2001. Optimizing Multidimensional Index Trees for Main Memory Access. In *Proc. of the ACM SIGMOD International Conference on Management of Data*. 139–150.

[17] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. 2012. KISS-Tree: smart latch-free in-memory indexing on modern architectures. *Proc. of the Eighth International Workshop on Data Management on New Hardware* (2012).

[18] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. *29th IEEE International Conference on Data Engineering* (2013).

[19] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. *29th IEEE International Conference on Data Engineering* (2013).

[20] Xin Li, Young-Jin Kim, Ramesh Govindan, and Wei Hong. 2003. Multidimensional range queries in sensor networks. In *Proc. of the 1st International Conference on Embedded Networked Sensor Systems*. 63–75.

[21] Astrid Lievre, Jean-Baptiste Bachet, Delphine Le Corre, Valerie Boige, Bruno Landi, Jean-François Emile, Jean-François Côté, Gorana Tomasic, Christophe Penna, Michel Ducreux, et al. 2006. KRAS mutation status is predictive of response to cetuximab therapy in colorectal cancer. *Cancer research* 66, 8 (2006), 3992–3995.

[22] Emmanuel Müller, Stephan Günnemann, Ira Assent, and Thomas Seidl. 2009. Evaluating Clustering in Subspace Projections of High Dimensional Data. *PVLDB* 2, 1 (2009), 1270–1281.

[23] Hasso Plattner. 2009. A common database approach for OLTP and OLAP using an in-memory column database. *Proc. of the ACM SIGMOD International Conference on Management of Data* (2009).

[24] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. 2018. Theoretically Optimal and Empirically Efficient R-trees with Strong Parallelizability. *PVLDB* 11, 5 (2018), 621–634.

[25] John T. Robinson. 1981. The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes. *Proc. of the ACM SIGMOD International Conference on Management of Data* (1981).

[26] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures.* Morgan Kaufmann.

[27] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. 2009. k-ary search on modern processors. *Proc. of the Fifth International Workshop on Data Management on New Hardware* (2009).

[28] Stefan Sprenger, Patrick Schäfer, and Ulf Leser. 2018. Multidimensional range queries on modern hardware. *Proc. of the 30th International Conference on Scientific and Statistical Database Management* (2018).

[29] Stefan Sprenger, Patrick Schäfer, and Ulf Leser. 2019. BB-Tree: A practical and efficient main-memory index structure for multidimensional workloads. *35th IEEE International Conference on Data Engineering* (2019).

[30] Stefan Sprenger, Steffen Zeuch, and Ulf Leser. 2016. Cache-Sensitive Skip List: Efficient Range Queries on Modern CPUs. *4th International Workshop on In-Memory Data Management and Analytics* (2016).

[31] The 1000 Genomes Project Consortium. 2015. A global reference for human genetic variation. *Nature* 526, 7571 (2015), 68–74.

[32] Sheng Wang, David Maier, and Beng Chin Ooi. 2014. Lightweight Indexing of Observational Data in Log-Structured Storage. *PVLDB* 7, 7 (2014), 529–540.

[33] Sheng Wang, David Maier, and Beng Chin Ooi. 2016. Fast and Adaptive Indexing of Multi-Dimensional Observational Data. *PVLDB* 9, 14 (2016), 1683–1694.

[34] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. *Proc. of the 24rd International Conference on Very Large Data Bases* (1998).

[35] Tilmann Zäschke, Christoph Zimmerli, and Moira C. Norrie. 2014. The PH-tree: a space-efficient storage structure and multi-dimensional index. *Proc. of the ACM SIGMOD International Conference on Management of Data* (2014).

Semantic and Influence aware k-Representative Queries over Social Streams

Yanhao Wang
National University of Singapore
yanhao90@comp.nus.edu.sg

Yuchen Li
Singapore Management University
yuchenli@smu.edu.sg

Kian-Lee Tan
National University of Singapore
tankl@comp.nus.edu.sg

ABSTRACT

Massive volumes of data continuously generated on social platforms have become an important information source for users. A primary method to obtain fresh and valuable information from social streams is *social search*. Although there have been extensive studies on social search, existing methods only focus on the *relevance* of query results but ignore the *representativeness*. In this paper, we propose a novel Semantic and Influence aware k-Representative (k-SIR) query for social streams based on topic modeling. Specifically, we consider that both user queries and elements are represented as vectors in the topic space. A k-SIR query retrieves a set of k elements with the maximum *representativeness* over the sliding window at query time w.r.t. the query vector. The representativeness of an element set comprises both semantic and influence scores computed by the topic model. Subsequently, we design two approximation algorithms, namely Multi-Topic ThresholdStream (MTTS) and Multi-Topic ThresholdDescend (MTTD), to process k-SIR queries in real-time. Both algorithms leverage the ranked lists maintained on each topic for k-SIR processing with theoretical guarantees. Extensive experiments on real-world datasets demonstrate the effectiveness of k-SIR query compared with existing methods as well as the efficiency and scalability of our proposed algorithms for k-SIR processing.

1 INTRODUCTION

Enormous amount of data is being continuously generated by web users on social platforms at an unprecedented rate. For example, around 650 million tweets are posted by 330 million users on Twitter per day. Such user generated data can be modeled as continuous social streams, which are key sources of fresh and valuable information. Nevertheless, social streams are extremely overwhelming for their huge volumes and high velocities. It is impractical for users to consume social data in its raw form. Therefore, *social search* [7–9, 17, 19, 28, 33, 37, 39] has become the primary approach to facilitating users on finding their interested content from massive social streams.

Existing search methods for social data can be categorized into keyword-based approaches and topic-based approaches based on how they measure the relevance between queries and elements. Keyword-based approaches [7–9, 17, 28, 33, 37] adopt the *textual relevance* (e.g., TF-IDF and BM25) for evaluation. However, they merely capture the *syntactic* correlation but ignore the *semantic* correlation. Considering the tweets in Figure 1, if a query "soccer" is issued, no results will be found because none of the tweets contains the term "soccer". It is noted that the words like "as-roma" and "LFC" are semantically relevant to "soccer". Therefore, elements such as e_1, e_2 are relevant to the query but missing from the result. Thus, overlooking the semantic meanings of user queries may degrade the result quality, especially against social data where lexical variation is prevalent [14].

To overcome this issue, topic-based approaches [19, 39] project user queries and elements into the same latent space defined by a probabilistic topic model [5]. Consequently, queries and elements are both represented as vectors and their relevance is computed by similarity measures for vectors (e.g., cosine distance) in the topic space. Although topic-based approaches can better capture the *semantic* correlation between queries and elements, they focus on the *relevance* of results but neglect the *representativeness*. Typically, they retrieve top-k elements that are the most coherent with the query as the result. Such results may not be *representative* in the sense of *information coverage* and *social influence*. First, users are more satisfied with the results that achieve an extensive coverage of information on query topics than the ones that provide limited information. For example, a top-2 query on topic θ_1 in Figure 2 returns $\{e_3, e_4\}$ as the result. Nevertheless, compared with e_4, e_6 can provide richer information to complement the news reported by e_3. Therefore, in addition to *relevance*, it is essential to consider *information coverage* to improve the result quality. Second, *influence* is another key characteristic to measure the *representativeness* of social data. Existing methods for social search [7, 8, 19, 37] have taken into account the influences of elements for scoring and ranking. These methods simply use the influences of authors (e.g., PageRank [24] scores) or the retweet/share count to compute the influence scores. Such a naïve integration of influence is topic-unaware and may lead to undesired query results. For example, e_6 in Figure 1, which is mostly related to θ_1, may appear in the result for a query on θ_2 because of its high retweet count. In addition, they do not consider that the influences of elements evolve over time, when previously trending contents may become outdated and new posts continuously emerge. Hence, incorporating a topic-aware and time-critical influence metric is imperative to capture recently trending elements.

To tackle the problems of existing search methods, we define a novel Semantic and Influence aware k-Representative (k-SIR) query for social streams based on topic modeling [5]. Specifically, a k-SIR query retrieves a set of k elements from the active elements corresponding to the sliding window W_t at the query time t. The result set collectively achieves the maximum representativeness score w.r.t. the query vector $\mathbf{x}$, each dimension of which indicates the degree of interest on a topic. We advocate the representativeness score of an element set to be a weighted sum of its semantic and influence scores on each topic. We adopt a weighted word coverage model to compute the semantic score so as to achieve the best information preservation, where the weight of a word is evaluated based on its information entropy [31, 42]. The influence score is computed by a probabilistic coverage model where the influence probabilities are topic-aware. In addition, we restrict the influences within the sliding window W_t so that the recently trending elements can be selected.

Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

ID	Tweet	Retweets
e_1	@asroma win but it's @LFC joining @realmadrid in the #UCL final	3154
e_2	#OnThisDay in 1993, @ManUtd were crowned the first #PL champion	1476
e_3	@Cavs defeats @Raptors 128-110 and leads the series 2-0 in #NBAPlayoffs	2706
e_4	LeBron is great! #NBAPlayoffs	2
e_5	Congratulations to @LFC reaching #UCL Final!! #YNWA	2167
e_6	LeBron is the 1st player with 40+ points 14+ assists in an #NBAPlayoffs game	3489
e_7	Hope this post inspires us to win #PL champions again in 2018-19	4
e_8	Schedule for #PL and #NBAPlayoffs tonight	25

Figure 1: A list of exemplar tweets

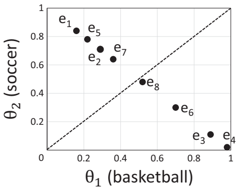
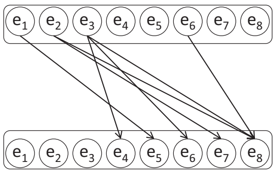


Figure 2: Topic distribution



e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8

Figure 3: References

The challenges of real-time k-SIR processing are two-fold. First, the k-SIR query is *NP-hard*. Second, it is highly *dynamic*, i.e., the results vary with query vectors and evolve quickly over time. Due to the submodularity of the scoring function, existing submodular maximization algorithms, e.g., CELF [16] and SieveStreaming [2], can provide approximation results for k-SIR queries with theoretical guarantees. However, existing algorithms need to evaluate all active elements at least once for a single query and often take several seconds to process one k-SIR query as shown in our experiments. To support real-time k-SIR processing over social streams, we maintain the ranked lists to sort the active elements on each topic by topic-wise representativeness score. We first devise the Multi-Topic ThresholdStream (MTTS) algorithm for k-SIR processing. Specifically, to prune unnecessary evaluations, MTTS sequentially retrieves elements from the ranked lists in decreasing order of their scores w.r.t. the query vector and can be terminated early whenever possible. Theoretically, it provides $(\frac{1}{2} - \varepsilon)$-approximation results for k-SIR queries and evaluates each active element at most once. Furthermore, we propose the Multi-Topic ThresholdDescend (MTTD) algorithm to improve upon MTTS. MTTD maintains the elements retrieved from ranked lists in a buffer and permits to evaluate an element more than once to improve the result quality. Consequently, it achieves a better $(1 - \frac{1}{e} - \varepsilon)$-approximation but has a higher worst-case time complexity than MTTS. Despite this, MTTD shows better empirical efficiency and result quality than those of MTTS.

Finally, we conduct extensive experiments on three real-world datasets to evaluate the effectiveness of k-SIR as well as the efficiency and scalability of MTTS and MTTD. The results of a user study and quantitative analysis demonstrate that k-SIR achieves significant improvements over existing methods in terms of *information coverage* and *social influence*. In addition, MTTS and MTTD achieve up to 124x and 390x speedups over the baselines for k-SIR processing with at most 5% and 1% losses in quality.

Our contributions in this work are summarized as follows.

- We define the k-SIR query to retrieve representative elements over social streams where both semantic and influence scores are considered. (Section 3)
- We propose MTTS and MTTD to process k-SIR queries in real-time with theoretical guarantees. (Section 4)
- We conduct extensive experiments to demonstrate the effectiveness of k-SIR as well as the efficiency and scalability of our proposed algorithms for k-SIR processing. (Section 5)

2 RELATED WORK

Search Methods for Social Streams. Many methods have been proposed for searching on social streams. Here, we categorize existing methods into two types: keyword-based approaches and topic-based approaches.

Keyword-based approaches [7–9, 17, 28, 33, 37, 40] typically define *top-k queries* to retrieve k elements with the highest scores as the results where the scoring functions combine the *relevance* to query keywords (measured by TF-IDF or BM25) with other contexts such as *freshness* [17, 28, 33, 37], *influence* [8, 37], and *diversity* [9]. They also design different indices to support instant updates and efficient top-k query processing. However, keyword queries are substantially different from the k-SIR query and thus keyword-based methods cannot be trivially adapted to process k-SIR queries based on topic modeling.

As the metrics for textual relevance cannot fully represent the semantic relevance between user interest and text, recent work [19, 39] introduces topic models [5] into social search, where user queries and elements are modeled as vectors in the topic space. The relevance between a query and an element is measured by cosine similarity. They define *top-k relevance query* to retrieve k most relevant elements to a query vector. However, existing methods typically consider the *relevance* of results but ignore the *representativeness*. Therefore, the algorithms in [19, 39] cannot be used to process k-SIR queries that emphasize the *representativeness* of results.

Social Stream Summarization. There have been extensive studies on social stream summarization [1, 4, 23, 25–27, 29, 36] : the problem of extracting a set of *representative* elements from social streams. Shou et al. [27, 36] propose a framework for social stream summarization based on dynamic clustering. Ren et al. [26] focus on the personalized summarization problem that takes users' interests into account. Olariu [23] devise a graph-based approach to abstractive social summarization. Bian et al. [4] study the multimedia summarization problem on social streams. Ren et al. [25] investigate the multi-view opinion summarization of social streams. Agarwal and Ramamritham [1] propose a graph-based method for contextual summarization of social event streams. Nguyen et al. [31] consider maintaining a sketch for a social stream to best preserve the latent topic distribution.

However, the above approaches cannot be applied to ad-hoc query processing because they (1) do not provide the query interface and (2) are not efficient enough. For each query, they need to filter out irrelevant elements and invoke a new instance of the summarization algorithm to acquire the result, which often takes dozens of seconds or even minutes. Therefore, it is unrealistic to deploy a summarization method on a social platform for ad-hoc queries since thousands of users could submit different queries at the same time and each query should be processed in real-time.

Submodular Maximization. Submodular maximization has attracted a lot of research interest recently for its theoretical significance and wide applications. The standard approaches to submodular maximization with a cardinality constraint are the greedy heuristic [22] and its improved version CELF [16], both of which are $(1 - \frac{1}{e})$-approximate. Badanidiyuru and Vondrak [3] propose several approximation algorithms for submodular maximization with general constraints. Kumar et al. [15] and Badanidiyuru et al. [2] study the submodular maximization problem in the distributed and streaming settings. Epasto et al. [12] and

Wang et al. [34] further investigate submodular maximization in the sliding window model. However, the above algorithms do not utilize any indices for acceleration and thus they are much less efficient for k-SIR processing than MTTS and MTTD proposed in this paper.

3 PROBLEM FORMULATION

3.1 Data Model

Social Element. A social element e is represented as a triple $\langle ts, doc, ref \rangle$, where $e.ts$ is the timestamp when e is posted, $e.doc$ is the textual content of e denoted by a bag of words drawn from a vocabulary $\mathcal{V}$ indexed by $\{1, \ldots, m\}$ ($m = |\mathcal{V}|$), and $e.ref$ is the set of elements referred to by e. Given two elements e and e' ($e'.ts < e.ts$), if e refers to e', i.e., $e' \in e.ref$, we say e' influences e, which is denoted as $e' \rightsquigarrow e$. In this way, the attribute ref captures the influence relationships between social elements [30, 35]. If e is totally original, we set $e.ref = \emptyset$. For example, tweets on Twitter shown in Table 1 are typical social elements and the propagation of hashtags can be modeled as references [18, 30]. Note that the *influence relationships* vary for different types of elements, e.g., "cite" between academic papers and "comment" on Reddit can also be modeled as references.

Social Stream. We consider social elements arrive continuously as a data stream. A social stream E comprises a sequence of elements indexed by $\{1, 2, 3, \ldots\}$. Elements are ordered by timestamps and multiple elements with the same timestamp may arrive in an arbitrary manner. Furthermore, social streams are time-sensitive: elements posted or referred to recently are more important and interesting to users than older ones. To capture the *freshness* of social streams, we adopt the well-recognized time-based sliding window [11] model. Given the window length T, a sliding window W_t at time t comprises the elements from time $t - T + 1$ to T, i.e., $W_t = \{e \in E | e.ts \in [t - T + 1, t]\}$. The set of active elements A_t at time t includes not only the elements in W_t but also the elements referred to by any element in W_t, i.e., $A_t = W_t \cup \{e' \in E | e \in W_t \wedge e' \in e.ref\}$. We use $n_t = |A_t|$ to denote the number of active elements at time t.

Topic Model. We use probabilistic topic models [5] such as LDA [6] and BTM [38] to measure the (semantic and influential) representativeness of elements and the preferences of users. A topic model $\Theta = \{\theta_1, \ldots, \theta_z\}$ consisting of z topics is trained from the corpus $\mathcal{E} = \{e.doc | e \in E\}$ and the vocabulary $\mathcal{V}$. Each topic θ_i is a multinomial distribution over the words in $\mathcal{V}$, where $p_i(w)$ is the probability of a word w distributed on θ_i and $\sum_{w \in \mathcal{V}} p_i(w) = 1$. The topic distribution of an element e is a multinomial distribution over the topics in Θ, where $p_i(e)$ is the probability that $e.doc$ is generated from θ_i and $\sum_{i=1}^{z} p_i(e) = 1$.

The selection of appropriate topic models is orthogonal to our problem. In this work, we consider any probabilistic topic model can be used as a black-box oracle to provide $p_i(w), \forall w \in \mathcal{V}$ and $p_i(e), \forall e \in E$. Note that the evolution of topic distribution is typically much slower than the speed of social stream [38, 41]. In practice, we assume that the topic distribution remains stable for a period of time. We need to retrain the topic model from recent elements when it is outdated due to concept drift.

3.2 Query Definition

Query Vector. Given a topic model Θ of z topics, we use a z-dimensional vector $\mathbf{x} = \{x_1, \ldots, x_z\}$ to denote a user's preference on topics. Formally, $\mathbf{x} \in [0, 1]^z$ and, x_i indicates the user's degree of interest on θ_i. W.l.o.g., $\mathbf{x}$ is normalized to $\sum_{i=1}^{z} x_i = 1$. Since it is impractical for users to provide the query vectors directly for their lack of knowledge about the topic model Θ, we design a scheme to transform the standard *query-by-keyword* [17] paradigm in our case: the keywords provided by a user is treated as a pseudo-document and the query vector is inferred from its distribution over the topics in Θ. Note that other query paradigms can also be supported, e.g., the *query-by-document* [39] paradigm where a document is provided as a query and the *personalized search* [19] where the query vector is inferred from a user's recent posts.

Definition of Representativeness. Given a set of elements S and a query vector $\mathbf{x}$, the *representativeness* of S w.r.t. $\mathbf{x}$ at time t is defined by a function $f(\cdot, \cdot) : 2^{|E|} \times [0, 1]^z \rightarrow \mathbb{R}_{\geq 0}$ that maps any subset of E to a nonnegative score w.r.t. a query vector. Formally, we have

$$f(S, \mathbf{x}) = \sum_{i=1}^{z} x_i \cdot f_i(S) \tag{1}$$

where $f_i(S)$ is the score of S on topic θ_i. Intuitively, the overall score of S w.r.t. $\mathbf{x}$ is the weighted sum of its scores on each topic. The score $f_i(S)$ on θ_i is defined as a linear combination of its semantic and influence scores. Formally,

$$f_i(S) = \lambda \cdot \mathcal{R}_i(S) + \frac{1 - \lambda}{\eta} \cdot \mathcal{I}_{i,t}(S) \tag{2}$$

where $\mathcal{R}_i(S)$ is the semantic score of S on θ_i, $\mathcal{I}_{i,t}(S)$ is the influence score of S on θ_i at time t, $\lambda \in [0, 1]$ specifies the trade-off between semantic and influence scores, and $\eta > 0$ adjusts the ranges of $\mathcal{R}_i(\cdot)$ and $\mathcal{I}_{i,t}(\cdot)$ to the same scale. Next, we will introduce how to compute the semantic and influence scores based on the topic model Θ respectively.

Topic-specific Semantic Score. Given a topic θ_i, we define the semantic score of a set of elements by the *weighted word coverage* model. We first define the weight of a word w in $e.doc$ on θ_i. According to the generative process of topic models [5], the probability $p_i(w, e)$ that $w \in e.doc$ is generated from θ_i is denoted as $p_i(w, e) = p_i(w) \cdot p_i(e)$. Following [31, 42], the weight $\sigma_i(w, e)$ of w in $e.doc$ on θ_i can be defined by its frequency and information entropy, i.e., $\sigma_i(w, e) = -\gamma(w, e) \cdot p_i(w, e) \cdot \log p_i(w, e)$, where $\gamma(w, e)$ is the frequency of w in $e.doc$. Then, the semantic score of e on θ_i is the sum of the weights of distinct words in $e.doc$, i.e., $\mathcal{R}_i(e) = \sum_{w \in V_e} \sigma_i(w, e)$ where V_e is the set of distinct words in $e.doc$. We extend the definition of semantic score to an element set by handling the word overlaps. Given a set S and a word w, if w appears in more than one element of S, its weight is computed only once for the element e with the maximum $\sigma_i(w, e)$. Formally, the semantic score of S on θ_i is defined by

$$\mathcal{R}_i(S) = \sum_{w \in V_S} \max_{e \in S} \sigma_i(w, e) \tag{3}$$

where $V_S = \cup_{e \in S} V_e$. Equation 3 aims to select a set of elements to maximally cover the important words on θ_i so as to best preserve the information of θ_i. Additionally, it implicitly captures the diversity issue because adding highly similar elements to S brings little increase in $\mathcal{R}_i(S)$.

Example 3.1. Table 1 gives a social stream extracted from the tweets in Figure 1 and a topic model on the vocabulary of elements in the stream. We demonstrate how to compute the semantic score $\mathcal{R}_2(S)$ where $S = \{e_2, e_7\}$ on θ_2. The frequency of each word in any element is 1. The set of words in S is $V_S = \{w_4, w_9, w_{11}\}$. The word w_9 only appears in e_2. Its weight is $\sigma_2(w_9, e_2) = 0.15$. The words w_4, w_{11} appear in both elements. As $\sigma_2(w_4, e_2) = 0.18 > \sigma_2(w_4, e_7) = 0.17$ and $\sigma_2(w_{11}, e_2) = 0.20 >$

Table 1: Example for social stream and topic model

(a) Elements extracted from tweets in Figure 1

Elem ID	Time	Words	θ_1	θ_2	References
e_1	1	$w_1, w_6, w_8, w_{14}, w_{16}$	0.2	0.8	$\varnothing$
e_2	2	w_4, w_9, w_{11}	0.26	0.74	$\varnothing$
e_3	3	w_3, w_5, w_{10}, w_{13}	0.89	0.11	$\varnothing$
e_4	4	w_7, w_{10}	1	0	e_3
e_5	5	w_6, w_8, w_{16}	0.29	0.71	e_1
e_6	6	w_2, w_7, w_{10}, w_{12}	0.7	0.3	e_3
e_7	7	w_4, w_{11}	0.33	0.67	e_2
e_8	8	w_{10}, w_{11}, w_{15}	0.51	0.49	e_2, e_3, e_6

(b) Topic-Word distribution – I

Word ID	Word	θ_1	θ_2
w_1	asroma	0	0.03
w_2	assist	0.06	0.04
w_3	cavs	0.09	0
w_4	champion	0.1	0.09
w_5	defeat	0.05	0.04
w_6	final	0.11	0.12
w_7	lebron	0.12	0
w_8	lfc	0	0.06

(c) Topic-Word distribution – II

Word ID	Word	θ_1	θ_2
w_9	manutd	0	0.07
w_{10}	nbaplayoffs	0.11	0
w_{11}	pl	0	0.11
w_{12}	point	0.15	0.14
w_{13}	raptors	0.08	0
w_{14}	realmadrid	0	0.07
w_{15}	schedule	0.13	0.12
w_{16}	ucl	0	0.11

$\sigma_2(w_{11}, e_7) = 0.19$, $\sigma_2(w_4, e_2)$ and $\sigma_2(w_{11}, e_2)$ are the weights of w_4 and w_{11} for S. Finally, we sum up the weights of each word in V_S and get $\mathcal{R}_2(S) = 0.53$. In this example, e_7 has no contribution to the semantic score because all words in e_7 are covered by e_2.

Topic-specific Time-critical Influence Score. Given a topic θ_i and two elements $e', e \in E$ ($e' \in e.ref$), the probability of influence propagation from e' to e on θ_i is defined by $p_i(e' \rightsquigarrow e) = p_i(e') \cdot p_i(e)$. Furthermore, the probability of influence propagation from a set of elements S to e on θ_i is defined by $p_i(S \rightsquigarrow e) = 1 - \prod_{e' \in S \cap e.ref} \big(1 - p_i(e' \rightsquigarrow e)\big)$. We assume the influences from different precedents to e are independent of each other and adopt the *probabilistic coverage* model to compute the influence probability from a set of elements to an element. To select recently trending elements, we define the influence score in the sliding window model where only the references observed within W_t are considered. Let $I_t(e') = \{e | e' \in e.ref \wedge e \in W_t\}$ be the set of elements influenced by e' at time t and $I_t(S) = \cup_{e' \in S} I_t(e')$ be the set of elements influenced by S at time t. The influence score of S on θ_i at time t is defined by

$$\mathcal{I}_{i,t}(S) = \sum_{e \in I_t(S)} p_i(S \rightsquigarrow e) \tag{4}$$

Equation 4 tends to select a set of influential elements on θ_i at time t. The value of $\mathcal{I}_{i,t}(S)$ will increase greatly only if an element e is added to S such that e is relevant to θ_i itself and e is referred to by many elements on θ_i within W_t.

Example 3.2. We compute the influence score $\mathcal{I}_{2,8}(S)$ of $S = \{e_2, e_3\}$ in Table 1 on θ_2 at time $t = 8$. We consider the window length $T = 4$ and $W_t = \{e_5, e_6, e_7, e_8\}$. $I_8(S)$ at time 8 is $\{e_6, e_7, e_8\}$ and e_4 expires at time 8. First, $p_2(S \rightsquigarrow e_6) = p_2(e_3 \rightsquigarrow e_6) = 0.03$. Similarly, $p_2(S \rightsquigarrow e_7) = p_2(e_2 \rightsquigarrow e_7) = 0.50$. For e_8, we have $p_2(S \rightsquigarrow e_8) = 1 - \big(1 - p_2(e_2 \rightsquigarrow e_8)\big) \cdot \big(1 - p_2(e_3 \rightsquigarrow e_8)\big) = 0.40$. Finally, we acquire $\mathcal{I}_{2,8}(S) = 0.03 + 0.5 + 0.4 = 0.93$. We can see, although e_3 is referred to by several elements, its influence score on θ_2 is low because e_3 and the elements referring to it are mostly on θ_1.

Query Definition. We formally define the *Semantic and Influence aware k-Representative* (k-SIR) query to select a set of elements S with the maximum representativeness score w.r.t. a query vector from a social stream. We have two constraints on the result of k-SIR query S: (1) its size is restricted to $k \in \mathbb{Z}^+$, i.e., S contains at most k elements, to avoid overwhelming users with too much information; (2) the elements in S must be active at time t, i.e., $S \subseteq A_t$, to satisfy the freshness requirement. Finally, we define a k-SIR query $q_t(k, \mathbf{x})$ as follows.

Definition 3.3 (k-SIR). Given the set of active elements A_t and a vector $\mathbf{x}$, a k-SIR query $q_t(k, \mathbf{x})$ returns a set of elements $S^* \subseteq A_t$ with a bounded size k such that the scoring function $f(\cdot, \mathbf{x})$ is maximized, i.e., $S^* = \operatorname{argmax}_{S \subseteq A_t : |S| \leq k} f(S, \mathbf{x})$, where S^* is the optimal result for $q_t(k, \mathbf{x})$ and $\texttt{OPT} = f(S^*, \mathbf{x})$ is the optimal representativeness score.

Example 3.4. We consider two k-SIR queries on the social stream in Table 1. We set $\lambda = 0.5$, $\eta = 2$ in Equation 2 and the window length $T = 4$. At time 8, the set of active elements A_t contains all except e_4. Given a k-SIR query $q_8(2, \mathbf{x}_1)$ where $\mathbf{x}_1 = (0.5, 0.5)$ (a user has the same interest on two topics), $S^* = \{e_1, e_3\}$ is the query result and $\texttt{OPT} = f(S^*, \mathbf{x}_1) = 0.65$. We can see e_3, e_1 obtain the highest scores on θ_1, θ_2 respectively and they collectively achieve the maximum score w.r.t. $\mathbf{x}_1$. Given an k-SIR query $q_8(2, \mathbf{x}_2)$ where $\mathbf{x}_2 = (0.1, 0.9)$ (the user prefers θ_2 to θ_1), the query result is $S^* = \{e_1, e_2\}$ and $\texttt{OPT} = 0.94$. e_3 is excluded because it is mostly distributed on θ_1.

3.3 Properties and Challenges

Properties of k-SIR Queries. We first show the *monotonicity* and *submodularity* of the scoring function $f(\cdot, \cdot)$ for k-SIR query by proving that both the semantic function $\mathcal{R}_i(\cdot)$ and the influence function $\mathcal{I}_{i,t}(\cdot)$ are monotone and submodular.

Definition 3.5 (Monotonicity & Submodularity). A function $g(\cdot) : 2^{|E|} \rightarrow \mathbb{R}_{\geq 0}$ on the power set of E is monotone iff $g(S \cup \{e\}) \geq g(S)$ for any $e \in E \setminus S$ and $S \subseteq E$. The function $g(\cdot)$ is submodular iff $g(S \cup \{e\}) - g(S) \geq g(T \cup \{e\}) - g(T)$ for any $S \subseteq T \subseteq E$ and $e \in E \setminus T$.

Lemma 3.6. *$\mathcal{R}_i(\cdot)$ is monotone and submodular for $i \in [1, z]$.*

Lemma 3.7. *$\mathcal{I}_{i,t}(\cdot)$ is monotone and submodular for $i \in [1, z]$ at any time t.*

The proofs are given in Appendices A.1 and A.2.

Given a query vector $\mathbf{x}$, the scoring function $f(\cdot, \mathbf{x})$ is a non-negative linear combination of $\mathcal{R}_i(\cdot)$ and $\mathcal{I}_{i,t}(\cdot)$. Therefore, $f(\cdot, \mathbf{x})$ is *monotone* and *submodular.*

Challenges of k-SIR Queries. In this paper, we consider that the elements arrive continuously over time. We always maintain the set of active elements A_t at any time t. It is required to provide the result for any ad-hoc k-SIR query $q_t(k, \mathbf{x})$ in real-time.

The challenges of processing k-SIR queries in such a scenario are two-fold: (1) *NP-hardness* and (2) *dynamism.* First, the following theorem shows the k-SIR query is *NP-hard.*

Theorem 3.8. *It is NP-hard to obtain the optimal result S^* for any k-SIR query $q_t(k, \mathbf{x})$.*

The *weighted maximum coverage* problem can be reduced to k-SIR query when $\lambda = 1$ in Equation 2. Meanwhile, the *probabilistic coverage* problem is a special case of k-SIR query when $\lambda = 0$ in

Table 2: Frequently Used Notations

Notation	Description
E, e, e_i	$E = \{e_1, \ldots, e_n\}$ is a social stream; e is an arbitrary element in E; e_i is the i-th element in E.
T, W_t, A_t	T is the window length; W_t is the sliding window at time t; A_t is the set of active elements at time t.
Θ, θ_i	Θ is a topic model; θ_i is the i-th topic in Θ.
$\mathbf{x}, x_i$	$\mathbf{x}$ is a z-dimensional vector; x_i is the i-th entry of $\mathbf{x}$.
$\mathcal{R}_i(\cdot), \mathcal{I}_{i,t}(\cdot)$	$\mathcal{R}_i(\cdot)$ is the semantic function on θ_i; $\mathcal{I}_{i,t}(\cdot)$ is the influence function on θ_i at time t.
$f_i(\cdot), f(\cdot,\cdot)$	$f_i(\cdot)$ is the representativeness scoring function on θ_i; $f(\cdot,\cdot)$ is the scoring function w.r.t. a query vector.
$q_t(k, \mathbf{x})$	$q_t(k, \mathbf{x})$ is a k-SIR query at time t with a bounded result size k and a query vector $\mathbf{x}$.
$S^*, \mathtt{OPT}$	S^* is the optimal result for $q_t(k, \mathbf{x})$; $\mathtt{OPT} = f(S^*, \mathbf{x})$ is the optimal representativeness score.
$\delta_i(e), \delta(e, \mathbf{x})$	$\delta_i(e) = f_i(\{e\})$ is the score of e on θ_i; $\delta(e, \mathbf{x}) = f(\{e\}, \mathbf{x})$ is the score of e w.r.t. $\mathbf{x}$.
$\Delta(e\|S)$	$\Delta(e\|S) = f(S \cup \{e\}, \mathbf{x}) - f(S, \mathbf{x})$ is the marginal score gain of adding e to S.
RL_i	RL_i is the ranked list maintained for the elements on topic θ_i.

Equation 2. Because both problems are NP-hard [13], the k-SIR query is NP-hard as well.

In spite of this, existing algorithms for submodular maximization [22] can provide results with constant approximations to the optimal ones for k-SIR queries due to the monotonicity and submodularity of the scoring function. For example, CELF [16] is $(1 - \frac{1}{e})$-approximate for k-SIR queries while SieveStreaming [2] is $(\frac{1}{2} - \varepsilon)$-approximate (for any $\varepsilon > 0$). However, both algorithms cannot fulfill the requirements for real-time k-SIR processing owing to the dynamism of k-SIR queries. The results of k-SIR queries not only vary with query vectors but also evolve over time for the same query vector due to the changes in active elements and the fluctuations in influence scores over the sliding window. To process one k-SIR query $q_t(k, \mathbf{x})$, CELF and SieveStreaming should evaluate $f(\cdot, \mathbf{x})$ for $O(k \cdot n_t)$ and $O(\frac{\log k}{\varepsilon} \cdot n_t)$ times respectively. Empirically, they often take several seconds for one k-SIR query when the window length is 24 hours. To the best of our knowledge, none of the existing algorithms can efficiently process k-SIR queries. Thus, we are motivated to devise novel real-time solutions for k-SIR processing over social streams.

Before moving on to the section for k-SIR processing, we summarize the frequently used notations in Table 2.

4 QUERY PROCESSING

In this section, we introduce the methods to process k-SIR queries over social streams. The architecture is illustrated in Figure 4. At any time t, we maintain (1) **Active Window** to buffer the set of active elements A_t, (2) **Ranked Lists** $\mathsf{RL}_1, \ldots, \mathsf{RL}_z$ to sort the lists of elements on each topic of Θ in descending order of topic-wise representativeness score, and (3) **Query Processor** to leverage the *ranked lists* to process k-SIR queries. In addition, when the topic model is given, the query and topic inferences become rather standard (e.g., Gibbs sampling [21]), and thus we do not discuss these procedures here for space limitations. We consider the query vectors and the topic vectors of elements have been given in advance.

As shown in Figure 4, we process a social stream E in a batch manner. E is partitioned into buckets with equal time length $L \in \mathbb{Z}^+$ and updated at discrete time $L, 2L, \ldots$ until the end time of the stream t_n. When the window slides at time t, a bucket B_t containing the elements between time $t - L + 1$ to t is received. After inferring the topic vector of each $e \in B_t$ with the topic model, we first update the *active window*. The elements in B_t are inserted into the *active window* and the elements referred to by them are updated. Then, the elements that are never referred to by any element after time $t - T + 1$ are discarded from the *active window*. Subsequently, the *ranked list* RL_i on each topic θ_i is maintained for B_t. The detailed procedure for ranked lists maintenance will be presented in Section 4.1.

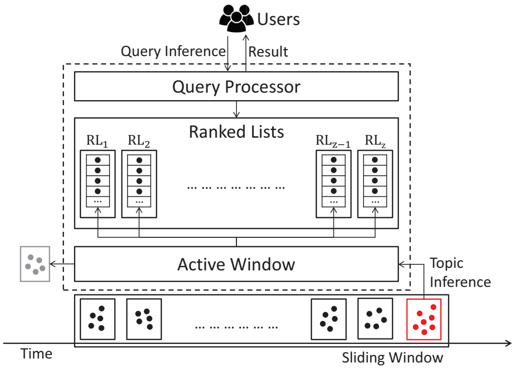


Figure 4: The architecture for k-SIR query processing

Next, let us discuss the mechanism of k-SIR processing. One major drawback of existing submodular maximization methods, e.g., CELF [16] and SieveStreaming [2], on processing k-SIR queries is that they need to evaluate every active element at least once. However, real-world datasets often have two characteristics: (1) The scores of elements are skewed, i.e., only a few elements have high scores. For example, we compute the scores of a sample of tweets w.r.t. a k-SIR query and scale the scores linearly to the range of 0 to 1. The statistics demonstrate that only 0.4% elements have scores of greater than 0.9 while 91% elements have scores of less than 0.1. (2) One element can only be high-ranked in very few topics, i.e., one element is about only one or two topics. In practice, we observe that the average number of topics per element is less than 2. Therefore, most of the elements are not relevant to a specific k-SIR query. We can greatly improve the efficiency by avoiding the evaluations for the elements with very low chances to be included into the query result. To prune these unnecessary evaluations, we leverage the ranked lists to sequentially evaluate the active elements in decreasing order of their scores w.r.t. the query vector. In this way, we can track whether unevaluated elements can still be added to the query result and terminate the evaluations as soon as possible.

Although such a method to traverse the *ranked lists* is similar to the one for top-k query [39], the procedures for maintaining the query results are totally different. A top-k query simply returns k elements with the maximum scores as the result for a k-SIR query. Although the top-k result can be retrieved efficiently from the *ranked lists* using existing methods [39], its quality for k-SIR queries is suboptimal because the word and influence overlaps are ignored. Thus, we will propose the Multi-Topic ThresholdStream (MTTS) and Multi-Topic ThresholdDescend (MTTD) algorithms for k-SIR processing in Sections 4.2 and 4.3. They can return high-quality results with constant approximation guarantees for k-SIR queries while meeting the real-time requirements.

Algorithm 1: Ranked List Maintenance

```
   Input: A social stream $E$, the window length $T$, and the
          bucket length $L$
1  $t \leftarrow 0$, initialize an empty ranked list $\mathsf{RL}_i$ for $i \in [1, z]$;
2  while $t \leq t_n$ do
3      $t \leftarrow t + L, B_t \leftarrow \{e \in E | e.ts \in [t - L + 1, t]\}$;
4      foreach $e \in B_t$ do
5          foreach $i : p_i(e) > 0$ do
6              $\delta_i(e) \leftarrow \mathcal{R}_i(e), t_e \leftarrow e.ts$;
7              create a tuple $\langle \delta_i(e), t_e \rangle$ and insert it into $\mathsf{RL}_i$;
8          foreach $e' \in e.ref$ do
9              foreach $i : p_i(e') > 0 \wedge p_i(e) > 0$ do
10                 $\delta_i(e') \leftarrow f_i(\{e'\}), t_{e'} \leftarrow e.ts$;
11                 adjust the position of $\langle \delta_i(e'), t_{e'} \rangle$ in $\mathsf{RL}_i$;
12     foreach $e$ : $e$ is never referred to after $t - T + 1$ do
13         delete the tuples of $e$ from $\mathsf{RL}_i$ with $p_i(e) > 0$;
```

4.1 Ranked List Maintenance

In this subsection, we introduce the procedure for ranked list maintenance. Generally, a ranked list RL_i keeps a tuple for each active element on topic θ_i. A tuple for element e is denoted as $\langle \delta_i(e), t_e \rangle$ where $\delta_i(e) = f_i(\{e\})$ is the topic-wise representativeness score of e on θ_i and t_e is the timestamp when e is last referred to. All tuples in RL_i are sorted in descending order of topic-wise score.

The algorithmic description of ranked list maintenance over a social stream is presented in Algorithm 1. Initially, an empty ranked list is initialized for each topic θ_i in the topic model Θ (Line 1). At discrete timestamps $t = L, 2L, \ldots$ until t_n, the *ranked lists* are updated according to a bucket of elements B_t. For each element e in B_t, a tuple $\langle \delta_i(e), t_e \rangle$ is created and inserted into RL_i for every topic θ_i with $p_i(e) > 0$ (Lines 4–7). The score $\delta_i(e)$ is $\mathcal{R}_i(e)$ because the elements influenced by e have not been observed yet. The time t_e when e is last referred to is obviously $e.ts$. Subsequently, it recomputes the influence score $\mathcal{I}_{i,t}(e')$ for each parent e' of e. After that, it updates the tuple $\langle \delta_i(e'), t_{e'} \rangle$ by setting $\delta_i(e')$ to $f_i(\{e'\})$ and $t_{e'}$ to $e.ts$. The position of $\langle \delta_i(e'), t_{e'} \rangle$ in RL_i is adjusted according to the updated $\delta_i(e')$ (Lines 8–11). Finally, we delete the tuples for expired elements from RL_i (Lines 12–13).

Complexity Analysis. The cost of evaluating $\delta_i(e)$ for any element e is $O(l)$ where $l = \max_{e \in A_t}(|V_e| + |I_t(e)|)$. Then, the complexity of inserting a tuple into RL_i is $O(\log n_t)$. For each $e' \in e.ref$, the complexity of re-evaluating $\mathcal{I}_{i,t}(e')$ is also $O(l)$. Overall, the complexity of maintaining RL_i for element e is $O\big(\mathcal{P}(l + \log n_t)\big)$ where $\mathcal{P} = \max_{e \in A_t} |e.ref|$. As the tuples for e may appear in $O(z)$ ranked lists, the time complexity of ranked list maintenance for element e is $O\big(z\mathcal{P}(l + \log n_t)\big)$.

Operations for Ranked List Traversal. We need to access the tuples in each *ranked list* RL_i in decreasing order of topic-wise score for k-SIR processing. Two basic operations are defined to traverse the ranked list RL_i: (1) RL_i.first to retrieve the element w.r.t. the first tuple with the maximum topic-wise score from RL_i; (2) RL_i.next to acquire the element w.r.t. the next unvisited tuple in RL_i from the current one. Note that once a tuple for element e has been accessed in one ranked list, the remaining tuples for e in the other lists will be marked as "visited" so as to eliminate duplicate evaluations for e.

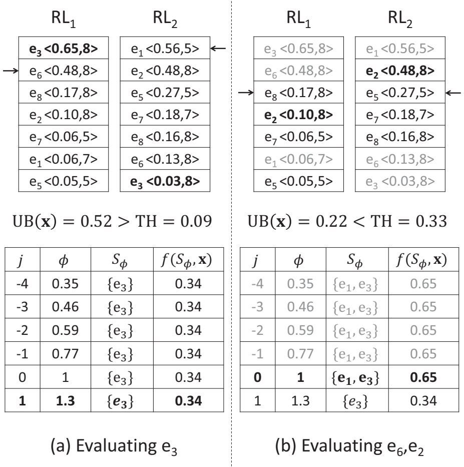
Algorithm 2: Multi-Topic ThresholdStream

```
   Input: The ranked list $\mathsf{RL}_i$ for each $i \in [1, z]$ and a $k$-SIR
          query $q_t(k, \mathbf{x})$
   Parameter: $\varepsilon \in (0, 1)$
   Result: $S_{ts}$ for $q_t(k, \mathbf{x})$
1  $\Phi = \{(1 + \varepsilon)^j | j \in \mathbb{Z}\}$, foreach $\phi \in \Phi$ do $S_\phi \leftarrow \varnothing$;
2  foreach $i \in [1, z] : x_i > 0$ do $e^{(i)} \leftarrow \mathsf{RL}_i$.first;
3  $\delta_{max}, \mathsf{TH} \leftarrow 0$ and $\mathsf{UB}(\mathbf{x}) \leftarrow \sum_{i=1}^{z} x_i \cdot \delta_i(e^{(i)})$;
4  while $\mathsf{UB}(\mathbf{x}) \geq \mathsf{TH}$ do
5      $i^* \leftarrow \mathrm{argmax}_{i \in [1,z]} x_i \cdot \delta_i(e^{(i)}), e \leftarrow e^{(i^*)}$;
6      $\delta(e, \mathbf{x}) \leftarrow \sum_{i=1}^{z} x_i \cdot \delta_i(e)$;
7      if $\delta(e, \mathbf{x}) > \delta_{max}$ then $\delta_{max} \leftarrow \delta(e, \mathbf{x})$;
8      $\Phi = \{(1 + \varepsilon)^j | j \in \mathbb{Z} \wedge \delta_{max} \leq (1 + \varepsilon)^j \leq 2 \cdot k \cdot \delta_{max}\}$;
9      delete $S_\phi$ if $\phi \notin \Phi$;
10     foreach $\phi \in \Phi$ do
11         if $\delta(e, \mathbf{x}) \geq \frac{\phi}{2k} \wedge |S_\phi| < k$ then
12             if $\Delta(e|S_\phi) \geq \frac{\phi}{2k}$ then $S_\phi \leftarrow S_\phi \cup \{e\}$;
13     $e^{(i^*)} \leftarrow \mathsf{RL}_{i^*}$.next;
14     $\mathsf{TH} \leftarrow \min_{\phi \in \Phi : |S_\phi| < k} \frac{\phi}{2k}$, $\mathsf{UB}(\mathbf{x}) \leftarrow \sum_{i=1}^{z} x_i \cdot \delta_i(e^{(i)})$;
15 return $S_{ts} \leftarrow \mathrm{argmax}_{\phi \in \Phi} f(S_\phi, \mathbf{x})$;
```

4.2 Multi-Topic ThresholdStream Algorithm

In this subsection, we present the MTTS algorithm for k-SIR processing. MTTS is built on two key ideas: (1) a thresholding approach [15] to submodular maximization and (2) a ranked list based mechanism for early termination. First, given a k-SIR query, the thresholding approach always tracks its optimal representativeness score OPT. It establishes a sequence of candidates with different *thresholds* within the range of OPT. For any element e, each candidate determines whether to include e independently based on e's marginal gain and its threshold. Second, to prune unnecessary evaluations, MTTS utilizes *ranked lists* to sequentially feed elements to the candidates in decreasing order of score. It continuously checks the minimum threshold for an element to be added to any candidate and the upper-bound score of unevaluated elements. MTTS is terminated when the upper-bound score is lower than the minimum threshold. After termination, the candidate with the maximum score is returned as the result for the k-SIR query.

The algorithmic description of MTTS is presented in Algorithm 2. The initialization phase is shown in Lines 1–3. Given a parameter $\varepsilon \in (0, 1)$, MTTS establishes a geometric progression Φ with common ratio $(1 + \varepsilon)$ to estimate the optimal score OPT for $q_t(k, \mathbf{x})$. Then, it maintains a candidate S_ϕ initializing to $\varnothing$ for each $\phi \in \Phi$. The threshold for S_ϕ is $\frac{\phi}{2k}$. The traversal of ranked lists starts from the first tuple of each list. We use $e^{(i)}$ to denote the element corresponding to the current tuple from RL_i. MTTS keeps 3 variables: (1) δ_{max} to store the maximum score w.r.t. $\mathbf{x}$ among the evaluated elements, (2) TH to maintain the minimum threshold for an element to be added to any candidate, and 3) $\mathsf{UB}(\mathbf{x})$ to track the upper-bound score for any unevaluated element w.r.t. $\mathbf{x}$. Specifically, TH is the threshold $\frac{\phi}{2k}$ of the unfilled candidate S_ϕ (i.e., $|S_\phi| < k$) with the minimum ϕ. We set $\mathsf{TH} = 0$ before the evaluation. If $\delta(e, \mathbf{x}) < \mathsf{TH}$, e can be safely excluded from evaluation. In addition, for any unevaluated element e, it holds that $\delta_i(e) \leq \delta_i(e^{(i)})$ because the tuples in RL_i are sorted



RL_1	RL_2
e_3 <0.65,8>	e_1 <0.56,5> ←
→ e_6 <0.48,8>	e_2 <0.48,8>
e_8 <0.17,8>	e_5 <0.27,5>
e_2 <0.10,8>	e_7 <0.18,7>
e_7 <0.06,5>	e_8 <0.16,8>
e_1 <0.06,7>	e_6 <0.13,8>
e_5 <0.05,5>	**e_3 <0.03,8>**

$\mathsf{UB}(\mathbf{x}) = 0.52 > \mathsf{TH} = 0.09$

j	ϕ	S_ϕ	$f(S_\phi, \mathbf{x})$
-4	0.35	$\{e_3\}$	0.34
-3	0.46	$\{e_3\}$	0.34
-2	0.59	$\{e_3\}$	0.34
-1	0.77	$\{e_3\}$	0.34
0	1	$\{e_3\}$	0.34
1	**1.3**	**$\{e_3\}$**	**0.34**

(a) Evaluating e_3

RL_1	RL_2
e_3 <0.65,8>	e_1 <0.56,5>
e_6 <0.48,8>	**e_2 <0.48,8>**
→ e_8 <0.17,8>	e_5 <0.27,5> ←
e_2 <0.10,8>	e_7 <0.18,7>
e_7 <0.06,5>	e_8 <0.16,8>
e_1 <0.06,7>	e_6 <0.13,8>
e_5 <0.05,5>	e_3 <0.03,8>

$\mathsf{UB}(\mathbf{x}) = 0.22 < \mathsf{TH} = 0.33$

j	ϕ	S_ϕ	$f(S_\phi, \mathbf{x})$
-4	0.35	$\{e_1, e_3\}$	0.65
-3	0.46	$\{e_1, e_3\}$	0.65
-2	0.59	$\{e_1, e_3\}$	0.65
-1	0.77	$\{e_1, e_3\}$	0.65
0	**1**	**$\{e_1, e_3\}$**	**0.65**
1	1.3	$\{e_3\}$	0.34

(b) Evaluating e_6,e_2

Figure 5: Example for k-SIR processing using MTTS.

by topic-wise score. Thus, $\mathsf{UB}(\mathbf{x}) = \sum_{i=1}^{z} x_i \cdot \delta_i(e^{(i)})$ can be used as the upper-bound score of unevaluated elements w.r.t. $\mathbf{x}$.

After the initialization phase, the elements are sequentially retrieved from the ranked lists and evaluated by the candidates according to Lines 4–14. At each iteration, MTTS selects an element $e^{(i^*)}$ with the maximum $x_i \cdot \delta_i(e^{(i)})$ as the next element e for evaluation (Line 5). Subsequently, the candidate maintenance procedure is performed following Lines 6–9. It first computes the score $\delta(e, \mathbf{x})$ of e w.r.t. $\mathbf{x}$. Second, it updates the maximum score δ_{max}. Third, the range of OPT is adjusted to $[\delta_{max}, 2 \cdot k \cdot \delta_{max}]$. Fourth, it deletes the candidates out of the range for OPT. Next, each candidate S_ϕ determines whether to add e independently according to Lines 10–12. If $\delta(e, \mathbf{x}) < \frac{\phi}{2k}$ or S_ϕ has contained k elements, e will be ignored by S_ϕ. Otherwise, the marginal gain $\Delta(e|S_\phi) = f(S_\phi \cup \{e\}, \mathbf{x}) - f(S_\phi, \mathbf{x})$ of adding e to S_ϕ is evaluated. If $\Delta(e|S_\phi)$ reaches $\frac{\phi}{2k}$, e will be added to S_ϕ. Finally, it obtains the next element in RL_{i^*} as $e^{(i^*)}$ and updates TH, $\mathsf{UB}(\mathbf{x})$ accordingly (Lines 13 and 14). The evaluation procedure will be terminated when $\mathsf{UB}(\mathbf{x}) < \mathsf{TH}$ because $\delta(e', \mathbf{x}) \leq \mathsf{UB}(\mathbf{x}) < \mathsf{TH}$ is satisfied for any unevaluated element e', which can be safely pruned. Finally, MTTS returns the candidate with the maximum score as the result for $q_t(k, \mathbf{x})$ (Line 15).

Example 4.1. Following the example in Table 1, we show how MTTS processes a k-SIR query $q_8(2, \mathbf{x})$ where $\mathbf{x} = (0.5, 0.5)$ in Figure 5. We set $\varepsilon = 0.3$ in this example.

First of all, the traversals of RL_1 and RL_2 start from e_3 and e_1 respectively. Initially, we have $\mathsf{UB}(\mathbf{x}) = 0.61$ and $\mathsf{TH} = 0$. Then, the first element to evaluate is e_3 because $x_1 \cdot \delta_1(e_3) = 0.33 > x_2 \cdot \delta_2(e_1) = 0.28$. As $\delta(e_3, \mathbf{x}) = 0.34$, the range of OPT is $[0.34, 1.36]$. We have $0.34 \leq 1.3^{-4} \leq \ldots \leq 1.3^{1} \leq 1.36$ and 6 candidates with $j \in [-4, 1]$ are maintained. e_3 can be added to each of the candidates. After that, e_6 is the next element from RL_1. $\mathsf{UB}(\mathbf{x})$ and TH are updated to 0.52 and 0.09 respectively. The second element to evaluate is e_1 from RL_2. As $\delta(e_1, \mathbf{x}) = 0.31$, the candidate with $j = 1$ directly skips e_1 for $\delta(e_1, \mathbf{x}) < \frac{\phi}{2k} = 0.33$. Other candidates include e_1 as $\Delta(e_1|S_\phi) \geq \frac{\phi}{2k}$. Then, e_2 is the next element from RL_2. $\mathsf{UB}(\mathbf{x})$ decreases to 0.48 while TH increases to 0.33. Subsequently, e_6, e_2 are retrieved but skipped by all candidates. After evaluating e_2, $\mathsf{UB}(\mathbf{x})$ decreases to 0.22 and is lower than TH. Thus, no more evaluation is needed and $S_{ts} = \{e_1, e_3\}$ is returned as the result for $q_8(2, \mathbf{x})$.

The approximation ratio of MTTS is given in Theorem 4.2.

Theorem 4.2. *S_{ts} returned by MTTS is a $(\frac{1}{2} - \varepsilon)$-approximation result for any k-SIR query.*

The proof is given in Appendix A.3.

Complexity Analysis. The number of candidates in MTTS is $O(\frac{\log k}{\varepsilon})$ as the ratio between the lower and upper bounds for OPT is $O(k)$. The complexity of retrieving an element from ranked lists is $O(\log n_t)$. The complexity of evaluating one element for a candidate is $O(ld)$ where $l = \max_{e \in A_t}(|V_e| + |I_t(e)|)$ and d is the number of non-zero entries in the query vector $\mathbf{x}$. Thus, the complexity of MTTS to evaluate one element is $O(\log n_t + \frac{ld \log k}{\varepsilon})$. Overall, the time complexity of MTTS is $O\big(n'_t(\log n_t + \frac{ld \log k}{\varepsilon})\big)$ where n'_t is the number of elements evaluated by MTTS.

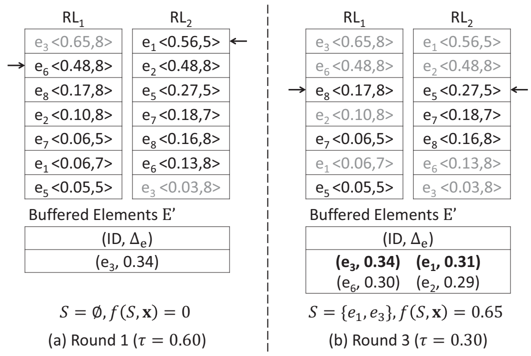
4.3 Multi-Topic ThresholdDescend Algorithm

Although MTTS is efficient for k-SIR processing, its approximation ratio is lower than the the best achievable approximation guarantees, i.e., $(1 - \frac{1}{e})$ [13] for submodular maximization with cardinality constraints. In addition, its result quality is also slightly inferior to that of CELF. In this subsection, we propose the Multi-Topic ThresholdDescend (MTTD) algorithm to improve upon MTTS. Different from MTTS, MTTD maintains only one candidate S from $\varnothing$ to reduce the cost for evaluation. In addition, it buffers the elements that are retrieved from ranked lists but not included into S so that these elements can be evaluated more than once. This can lead to better quality as the chances of missing significant elements are smaller. Specifically, MTTD has multiple rounds of evaluation with decreasing *thresholds*. In the round with threshold τ, each element e with $\delta(e, \mathbf{x}) \geq \tau$ is considered and will be included to S once the marginal gain $\Delta(e|S)$ reaches τ. When S contains k elements or τ is descended to the lower bound, MTTD is terminated and S is returned as the result. Theoretically, the approximation ratio of MTTD is improved to $(1 - \frac{1}{e} - \varepsilon)$ but its worst-case complexity is higher than MTTS. Despite this, the efficiency and result quality of MTTD are both better than MTTS empirically.

The algorithmic description of MTTD is presented in Algorithm 3. In the initialization phase (Lines 1–3), the candidate S and the element buffer E' are both set to $\varnothing$. The traversals of ranked lists are initialized in the same way as MTTS. The initial threshold τ for the first round of evaluation is the upper-bound score for any active element w.r.t. $\mathbf{x}$ and the termination threshold τ' is 0. After initialization, MTTD runs each round of evaluation with threshold τ following Lines 4–11. It first retrieves the set of elements E_τ whose scores potentially reach τ from the ranked lists. The method is shown in the procedure `retrieve`(τ) (Lines 13–19), which generally uses the same idea as MTTS: it traverses each ranked list sequentially in decreasing order of topic-wise scores and continuously adds the element with the maximum $x_i \cdot \delta_i(e^{(i)})$ to E_τ until the upper-bound score $\mathsf{UB}(\mathbf{x})$ is decreased to τ. After adding E_τ to the element buffer E', the evaluation procedure is started (Lines 6–10). It always considers the element $e' \in E' \setminus S$ with the maximum $\Delta_{e'}$. If the marginal gain $\Delta(e'|S)$ of adding e' to S is at least τ, e' will be included into S and deleted from E'. When S has contained k elements, MTTD is directly terminated and S is returned as the result S_{td} for $q_t(k, \mathbf{x})$. The round of evaluation is finished when no elements in E' could achieve a marginal gain of τ. Next, the termination threshold τ' is updated and the threshold τ is descended by $(1 - \varepsilon)$ times for the subsequent round of evaluation. Finally, when τ is lower than

Algorithm 3: Multi-Topic ThresholdDescend

```
    Input: The ranked list $\mathsf{RL}_i$ for each $i \in [1, z]$ and a $k$-SIR
           query $q_t(k, \mathbf{x})$
    Parameter: $\varepsilon \in (0, 1)$
    Result: $S_{td}$ for $q_t(k, \mathbf{x})$
 1  $S, E' \leftarrow \emptyset$;
 2  foreach $i \in [1, z] : x_i > 0$ do $e^{(i)} \leftarrow \mathsf{RL}_i$.first;
 3  $\tau \leftarrow \sum_{i=1}^{z} x_i \cdot \delta_i(e^{(i)}), \tau' \leftarrow 0$;
 4  while $\tau \geq \tau'$ do
 5      $E_\tau \leftarrow \mathtt{retrieve}(\tau), E' \leftarrow E' \cup E_\tau$;
 6      while $\exists e \in E' \setminus S : \Delta_e \geq \tau$ do
 7          $e' \leftarrow \mathrm{argmax}_{e \in E' \setminus S}\, \Delta_e, \Delta_{e'} \leftarrow \Delta(e'|S)$;
 8          if $\Delta_{e'} \geq \tau$ then
 9              $S \leftarrow S \cup \{e'\}, E' \leftarrow E' \setminus \{e'\}$;
10              if $|S| = k$ then return $S_{td} \leftarrow S$;
11      $\tau' \leftarrow f(S, \mathbf{x}) \cdot \frac{\varepsilon}{k}, \tau \leftarrow (1 - \varepsilon)\tau$;
12  return $S_{td} \leftarrow S$;
13  Procedure retrieve($\tau$)
14      $E_\tau \leftarrow \emptyset, \mathsf{UB}(\mathbf{x}) \leftarrow \sum_{i=1}^{z} x_i \cdot \delta_i(e^{(i)})$;
15      while $\mathsf{UB}(\mathbf{x}) \geq \tau$ do
16          $i^* \leftarrow \mathrm{argmax}_{i \in [1,z]}\, x_i \cdot \delta_i(e^{(i)})$;
17          $\Delta_{e^{(i^*)}} \leftarrow \sum_{i=1}^{z} x_i \cdot \delta_i(e^{(i^*)}), E_\tau \leftarrow E_\tau \cup \{e^{(i^*)}\}$;
18          $e^{(i^*)} \leftarrow \mathsf{RL}_{i^*}$.next, $\mathsf{UB}(\mathbf{x}) \leftarrow \sum_{i=1}^{z} x_i \cdot \delta_i(e^{(i)})$;
19      return $E_\tau$;
```



(a) Round 1 ($\tau = 0.60$)

(b) Round 3 ($\tau = 0.30$)

Figure 6: Example for k-SIR processing using MTTD.

τ', no more rounds of evaluations are required. In this case, S is returned as the result S_{td} for $q_t(k, \mathbf{x})$ even though it contains fewer than k elements (Line 12).

Example 4.3. In Figure 6, we illustrate the procedure for MTTD to process a k-SIR query $q_8(2, \mathbf{x})$ where $\mathbf{x} = (0.5, 0.5)$ following the example in Table 1. We also set $\varepsilon = 0.3$ in this example.

First, MTTD initializes the threshold $\tau = 0.60$ for the first round and the termination threshold $\tau' = 0$. The candidate S and the element buffer E' are initialized to $\emptyset$. In Round 1 and 2 with $\tau = 0.60$ and 0.42, MTTD retrieves 3 elements e_3, e_1, e_6 from RL_1 and RL_2 and adds them to E'. However, they are not evaluated in the first two rounds because $\Delta_{e_3} = 0.34$, $\Delta_{e_1} = 0.31$, and $\Delta_{e_6} = 0.30$, all of which are smaller than 0.42. In Round 3 with $\tau = 0.30$, e_2 is added to E'. Then, e_3 is added to S as $\Delta_{e_3} = 0.34 > \tau$. Furthermore, e_1 is also added to S as $\Delta_{e_1} = 0.31 > \tau$. At this time, $S = \{e_1, e_3\}$ has contained two elements. Therefore, MTTD is terminated and no more rounds are needed. $S_{td} = \{e_1, e_3\}$ is returned as the result for $q_8(k, \mathbf{x})$.

Table 3: Statistics of datasets

Dataset	AMiner	Reddit	Twitter
Number of Elements	1.66M	20.2M	14.8M
Vocabulary Size	580K / 71K	2.8M / 88K	3.0M / 68K
Average Length	74.5 / 49.2	24.6 / 8.6	12.6 / 5.1
Average References	3.68	0.85	0.62

The approximation ratio of MTTD is given in Theorem 4.4.

Theorem 4.4. *The result S_{td} returned by MTTD is $(1 - \frac{1}{e} - \varepsilon)$-approximate for any k-SIR query.*

The proof is given in Appendix A.4.

Complexity Analysis. Let τ_0 be the threshold τ of the first round in MTTD. The number of rounds in MTTD is at most $\lceil \log_{1-\varepsilon}(\frac{\tau'}{\tau_0}) \rceil$. Because $\tau' = f(S, \mathbf{x}) \cdot \frac{\varepsilon}{k} \geq \delta_{max} \cdot \frac{\varepsilon}{k}$ and $\tau_0 \leq d \cdot \delta_{max}$, we have $\frac{\tau_0}{\tau'} \leq \frac{kd}{\varepsilon}$ and the number of rounds is $O(\frac{\log(kd)}{\varepsilon^2})$. In each round, it evaluates $O(n''_t)$ elements where n''_t is the number of elements in the buffer E' of MTTD and the evaluation of an element is also $O(ld)$. Here, we use a max-heap for E' and thus it costs $O(\log n''_t)$ to dequeue the top element from E'. In addition, the time for retrieving an element from ranked lists is still $O(\log n_t)$. The complexity for each round is $O\big(n''_t \cdot (ld + \log n_t)\big)$. Therefore, the time complexity of MTTD is $O\big(n''_t \cdot \log(kd) \cdot \varepsilon^{-2} \cdot (ld + \log n_t)\big)$.

5 EXPERIMENTS

In this section, we conduct extensive experiments to verify the effectiveness of k-SIR query as well as the efficiency of MTTS and MTTD for k-SIR processing. We first introduce the experimental setup in Section 5.1. Then, we show the results for the effectiveness of k-SIR query in Section 5.2. Finally, the results for the efficiency and scalability of MTTS and MTTD are reported in Section 5.3.

5.1 Experimental Setup

Dataset. Three real-world datasets used in the experiments are listed as follows.

- **AMiner** [32] is a collection of academic papers published in the ACM Digital Library till 2015. We assign random timestamps to the papers published in the same year.
- **Reddit**[1] is a collection of submissions and comments on Reddit from June 01, 2014 to June 14, 2014.
- **Twitter**[2] consists of the tweets collected via the streaming API from July 14, 2017 to July 26, 2017.

The statistics of the datasets are given in Table 3. In the preprocessing, we remove stop words and noise words from the textual contents of elements. Note that we report the vocabulary size and the average length of elements both before and after the preprocessing.

Topic Model. We use LDA [6] to train topic models on the corpora of *AMiner* and *Reddit*. PLDA [21] is the implementation of LDA for training. For topic training on the corpus of *Twitter*, we use the biterm topic model [38] (BTM) because it is designed for short texts like tweets. The corpus of each dataset consists of $e.doc$ of each element e. To study how the number of topics z affects the performance of compared methods, we train 5 topic models for each dataset with z ranging from 50 to 250. Two Dirichlet priors α, β are set to $\frac{50}{z}$, 0.01 for both LDA and BTM.

[1] https://www.reddit.com/r/datasets

[2] https://developer.twitter.com/en/docs

The pre-trained topic models are loaded into memory and used as a black-box oracle for each compared method.

Compared Methods. We compare the following methods in Section 5.2 to evaluate the effectiveness of k-SIR query.

- **Top-k Keyword Query** (TF-IDF) retrieves k most relevant elements to the query keywords. We adopt the log-normalized TF-IDF weight to vectorize the elements and queries. Cosine similarity is used as the similarity measure between an element and a query.
- **Diversity-aware Top-k Keyword Query** [9] (DIV) considers both *textual relevance* and *result diversity*. Given a query q and a set of elements S, we have $score(q, S) = \lambda \sum_{e \in S} rel(q, e) + (1 - \lambda)div(S)$, where $rel(q, e)$ is the relevance of e to q and $div(S)$ is the average dissimilarity between each pair of elements in S. We set $\lambda = 0.3$ following [9]. A set of k elements S with the maximum $score(q, S)$ is returned as the result for q.
- **Sumblr** [27] is a method for social stream summarization. In our experiments, we use Sumblr for query processing as follows: given a set of keywords, we select the elements that contain at least one keyword as candidates. Then, we run Sumblr on the candidates to generate a summary of k elements as the query result. The parameters for k-means clustering and LexRank are the same as [27].
- **Top-k Relevance Query** [39] (REL) measures the relevance between an element and a query by topic modeling. It returns k elements whose topic vectors have the highest cosine similarities to the query vector as the result.
- **k-SIR Query** retrieves a set of elements S maximizing $f(S, \mathbf{x})$ w.r.t. a query vector $\mathbf{x}$. The results of MTTD are used in the effectiveness tests.

We note that TF-IDF, DIV, and Sumblr are keyword queries while REL and k-SIR use query vectors inferred from topic models. To compare them fairly, the queries are generated as follows: (1) draw the keywords from the vocabulary; (2) acquire a query vector by treating the keywords as a pseudo-document and inferring its topic vector from the topic model. To retrieve the query results, TF-IDF, DIV, and Sumblr receive the keywords while REL and k-SIR receive the query vectors.

The following methods are compared in Section 5.3 to evaluate their efficiency and scalability for k-SIR processing.

- **CELF** [16] is an improved version of the basic greedy algorithm [22]. It is the most common batch algorithm for submodular maximization and acquires $(1 - \frac{1}{e})$-approximation results for k-SIR queries. Note that $(1 - \frac{1}{e})$ is the best approximation ratio for this problem unless P=NP [13].
- **SieveStreaming** [2] is the state-of-the-art streaming algorithm for submodular maximization. It returns $(\frac{1}{2} - \varepsilon)$-approximation results for k-SIR queries.
- **Top-k Representative** retrieves k elements with the highest representativeness scores $\delta(e, \mathbf{x})$ w.r.t. a query vector $\mathbf{x}$ from ranked lists as the result, which is only $\frac{1}{k}$-approximate for k-SIR queries. We compare with it to show that traditional methods for top-k queries cannot work well for k-SIR queries.
- **MTTS** and **MTTD** are our proposed algorithms for k-SIR processing based on ranked lists.

Query and Workload Generation. We generate a k-SIR query as follows: (1) draw 1–5 words randomly from the vocabulary; (2) acquire the query vector by inferring the topic distribution of selected words from the topic model.

Table 4: Parameters in the experiments

Parameter	Setting	Default
the parameter ε in MTTS/MTTD	0.1 to 0.5	0.1
the result size k	5 to 25	10
the number of topics z	50 to 250	50
the window length T	6 hours to 30 hours	24 hours

In an experiment, we feed all elements in a dataset to compared methods in ascending order of timestamp. The active window and ranked lists perform batch-updates for each bucket of elements. Then, the query workload is generated as follows: we generate 10K k-SIR queries for each dataset and assign a random timestamp in range $[1, t_n]$ (t_n is the end time of the stream) to each query. The query results are retrieved at the assigned timestamps.

Parameter Setting. The parameters we examine in the experiments are listed in Table 4. In addition, the factors λ, η in Equation 2 are set to $0.5, 20$ for the *AMiner* and *Reddit* datasets, and $0.5, 200$ for the *Twitter* dataset. The bucket length L is fixed to 15 minutes.

Experimental Environment. All experiments are conducted on a server running Ubuntu 16.04.3 LTS. It has an Intel Xeon E7-4820 1.9GHz processor and 128 GB memory. All compared methods are implemented in Java 8.

5.2 Effectiveness

To evaluate the effectiveness of our k-SIR query, we first conduct a study on users' satisfaction for the results returned by each query method. We follow the methodology and procedure of user study in previous work on social search [9]. The detailed procedure is as follows.

First, we generate 20 queries by selecting 20 trending topics on three datasets (e.g., "*social media analysis*" on *AMiner*, "*NBA*" on *Reddit*, and "*pop music*" on *Twitter*) and use the topical words of each topic as keywords. Second, we process these queries with each method in the default setting and return a set of five elements as the results. Third, we recruit 30 volunteers who are not related to this work and familiar with the query topics to evaluate the result quality of compared methods. For each query, we ask 3 different evaluators to rank the quality of result sets and record the average score on each aspect. Specifically, each evaluator is requested to rank his/her satisfaction for the result sets on two aspects: (1) *representativeness*: the relevance to query topic and the information coverage on the query topic of its entirety (ranking from "the least representative" to "the most representative", mapped to values 1 to 5); (2) *impact*: the number of citations, comments, and retweets of selected elements (ranking from "the lowest impact" to "the highest impact", mapped to values 1 to 5).

The results of the user study are shown in Table 5. Following [9], we measure the agreement between different users by computing the Cohen's linearly weighted kappa [10] for each query on each aspect. The kappa values for *representativeness* are between 0.5 and 0.89 (0.72 on average). The kappa values for *impact* are in the range of 0.56–1.0 (0.79 on average). We observe that k-SIR achieves the highest scores among compared methods on both *representativeness* and *impact* in all datasets. We also collect feedback from users for the reason of dissatisfaction. "Low coverage" is the primary problem for TF-IDF and REL, while "containing irrelevant elements" is the main reason why the results of DIV and Sumblr are unsatisfactory.

Table 5: Results for user study

Method		TF-IDF	DIV	Sumblr	REL	k-SIR
AMiner	Represent.	2.28	1.56	3.72	2.78	**4.67**
	Impact	2.39	1.44	4.01	2.39	**4.78**
Reddit	Represent.	2.05	3.00	3.67	1.95	**4.33**
	Impact	1.80	2.24	3.80	2.33	**4.80**
Twitter	Represent.	1.79	2.38	4.08	2.08	**4.67**
	Impact	1.58	2.25	4.01	2.34	**4.88**

Table 6: Results for quantitative analysis

Method		TF-IDF	DIV	Sumblr	REL	k-SIR
AMiner	Coverage	0.1968	0.1766	0.2140	0.2400	**0.2663**
	Influence	0.0765	0.0777	0.5470	0.1159	**0.8430**
Reddit	Coverage	0.2387	0.2050	0.2419	0.2885	**0.3162**
	Influence	0.0175	0.0107	0.4315	0.0143	**0.5862**
Twitter	Coverage	0.2200	0.2118	0.2213	0.2722	**0.3052**
	Influence	0.0295	0.0296	0.1611	0.1268	**0.6516**

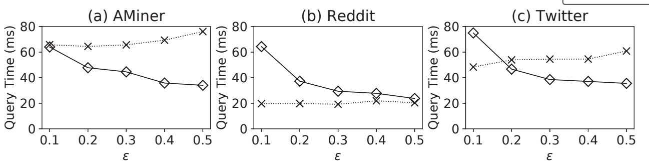


Figure 7: Query time with varying ε

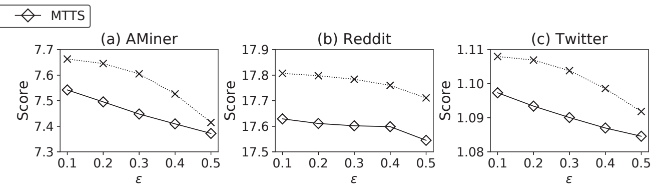


Figure 8: Scores with varying ε

Then, we use two quantitative metrics to evaluate the effectiveness of k-SIR query: (1) *coverage*: do the result sets achieve high information coverage on query topics? Following the metric used in previous studies [2, 20], the coverage score of a result set S w.r.t. a query vector $\mathbf{x}$ is computed by $\sum_{e \in A_t \setminus S} \max_{e' \in S} rel(e, \mathbf{x}) \cdot sim(e, e')$ where $rel(e, \mathbf{x})$ is the relevance of e to $\mathbf{x}$ and $sim(e, e')$ is the similarity of e and e'; (2) *influence*: are the result sets referred by a large number of elements (e.g., citations, comments, retweets, and so on)? We use the total number of elements referring to at least one element in the result set as the influence score. For ease of presentation, the influence scores are linearly scaled to $[0, 1]$ by dividing by the influence score of top-k influential elements. To acquire the results shown in Table 6, we sample the result sets of 1K queries returned by each method and compute the average scores.

We present the quantitative results for the effectiveness of compared methods in Table 6. First, k-SIR outperforms other query methods on *information coverage*, which verifies that our semantic model is able to preserve information on query topics. Second, as only k-SIR and Sumblr account for the influences of elements, they naturally achieve much higher influence scores than other methods. k-SIR further outperforms Sumblr in terms of *influence* because k-SIR directly adopt the number of references for influence computation while Sumblr only considers the PageRank scores of authors.

Overall, the above results have confirmed that k-SIR shows better result quality than existing methods for social search and summarization in terms of *information coverage* and *influence*.

5.3 Efficiency and Scalability

Effect of ε. The average CPU time of MTTS and MTTD to process one k-SIR query (i.e., *query time*) with varying ε is illustrated in Figure 7. MTTS and MTTD show different trends w.r.t. ε. On the one hand, the query time of MTTS drops drastically when ε increases as the number of candidates in MTTS is inversely proportional to ε. On the other hand, MTTD is not sensitive to ε and typically takes slightly more time for a larger ε. This is because a greater ε often leads to a smaller threshold for termination. In this case, more elements are retrieved from ranked lists and evaluated by MTTD, which degrades the query efficiency.

The average scores of the results returned by MTTS and MTTD with varying ε are shown in Figure 8. The scores of both methods decrease when ε increases, which is consistent with the theoretical results of Theorem 4.2 and 4.4. However, both methods show good robustness against ε: compared with CELF, their quality losses are at most 5% even when $\varepsilon = 0.5$.

Effect of result size k. The average query time of compared methods with varying k is presented in Figure 9. In addition, the average ratios between the number of elements evaluated by MTTS/MTTD and the number of active elements are shown in Figure 10. First of all, MTTS and MTTD run at least one order of magnitude faster than CELF and SieveStreaming for k-SIR processing in all datasets. MTTS and MTTD can achieve up to 124x and 390x speedups over the two baselines respectively. Compared with them, MTTS and MTTD can prune most of the unnecessary evaluations (at least 98% as shown in Figure 10) by utilizing the ranked lists. Then, the query time of MTTS and MTTD significantly grows with increasing k. The result can be explained by the ratios of evaluated elements. From Figure 10, we can see the ratio increases near linearly with k. As more elements are evaluated when k increases, the query time naturally rises. Finally, we can see MTTD outperforms MTTS in most cases but the ratio of elements evaluated by MTTD is always higher than MTTS. This is because MTTD only keeps one candidate but MTTS maintains multiple candidates independently. As a result, MTTD reduces the number of evaluations though it retrieves more elements from ranked lists than MTTS.

The average scores of the results returned by MTTS and MTTD with varying k are shown in Figure 11. We can see the result quality of MTTD is always nearly equal (>99%) to CELF for different k. Meanwhile, MTTS can also return results with over 95% representativeness scores compared with CELF. The results of SieveStreaming are inferior to those of CELF, MTTS, and MTTD. Although Top-k Representative shows the best performance among compared methods, its results are of the lowest quality among compared methods. In addition, its result quality degrades dramatically when k increases because the word and influence overlaps are ignored.

Scalability. We evaluate the scalability of MTTS and MTTD with varying the number of topics z and the window length T. The results for query time are illustrated in Figure 12 and 13. The query time of MTTS and MTTD drops when z increases. Because the average number of elements on each topic deceases with increasing z, the number of evaluated elements naturally decreases. However, when $z = 250$ in the *AMiner* dataset, the query time of MTTS and MTTD grows because there are more

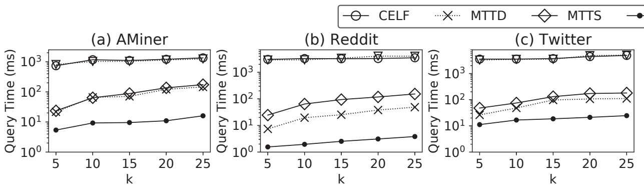


Figure 9: Query time with varying k

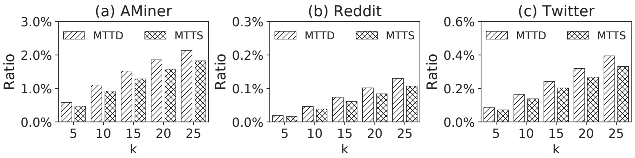


Figure 10: Ratios of evaluated elements with varying k

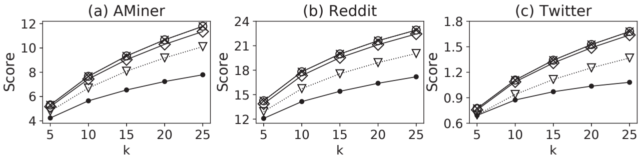


Figure 11: Scores with varying k

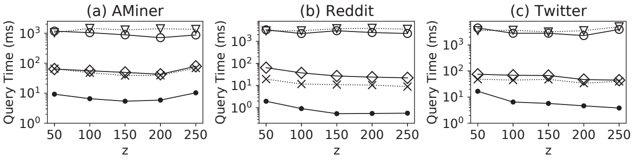


Figure 12: Query time with varying z

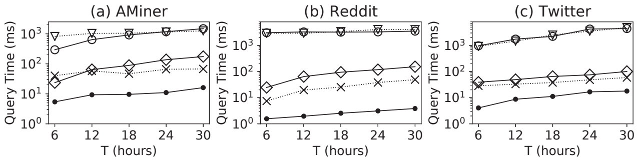


Figure 13: Query time with varying T

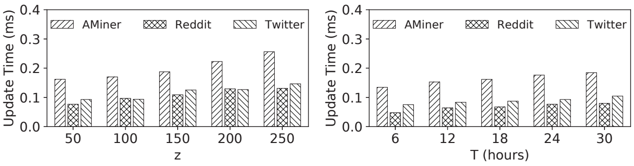


Figure 14: Update time with varying z and T

non-zero entries in the query vectors. The query time of all methods increases with T since there are more active elements. Nevertheless, MTTS and MTTD significantly outperform the baselines in all cases.

The average CPU time elapsed to update the ranked lists per arrival element is shown in Figure 14. We can see it takes more update time when z or T increases. As the number of maintained ranked lists is equal to z and the number of active elements grows with T, the cost for ranked list maintenance inevitably rises with increasing z or T. Nevertheless, the update time is always lower than 0.3ms in all datasets.

Overall, the experimental results show that our proposed methods demonstrate high efficiency and scalability for both ranked list maintenance and k-SIR processing, which can meet the requirements for real-world social streams.

6 CONCLUSION

In this paper, we defined a novel k-SIR query to retrieve a set of k representative elements from a social stream w.r.t. a query vector. We then proposed two algorithms, namely MTTS and MTTD, that leveraged the ranked lists for k-SIR processing over sliding windows. Theoretically, MTTS and MTTD provided $(\frac{1}{2} - \varepsilon)$ and $(1 - \frac{1}{e} - \varepsilon)$ approximation results for k-SIR queries respectively. Finally, we conducted extensive experiments on real-world datasets to demonstrate that (1) the k-SIR query achieved better performance in terms of *information coverage* and *social influence* than existing query methods on social data; (2) MTTS and MTTD had much higher efficiency and scalability than the baselines for k-SIR processing with near-equivalent result quality. In future work, we plan to extend our approach for supporting the incremental updates of topic models over streams.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments to improve this paper. Yuchen Li is supported by the Singapore MOE Tier 1 grant MSS18C001.

A PROOFS OF LEMMAS AND THEOREMS

A.1 Proof of Lemma 3.6

Proof. First, for $e \in E \setminus S$ and $S \subseteq E$, we have $\mathcal{R}_i(S \cup \{e\}) - \mathcal{R}_i(S) \geq \sum_{w \in V_e \setminus V_S} \sigma_i(w, e) \geq 0$ because $-p \cdot \log p \geq 0$ for $p \in [0, 1]$. Thus, $\mathcal{R}_i(\cdot)$ is monotone.

Given any $e \in E \setminus T$ and $S \subseteq T \subseteq E$, we use $\Delta(e|S) = \mathcal{R}_i(S \cup \{e\}) - \mathcal{R}_i(S)$ and $\Delta(e|T) = \mathcal{R}_i(T \cup \{e\}) - \mathcal{R}_i(T)$ to denote the marginal score gains of adding e to S and T.

First, as $S \subseteq T$, $V_S \subseteq V_T$. We divide V_e into three disjoint subsets $V_1 = V_e \setminus V_T$, $V_2 = V_e \cap (V_T \setminus V_S)$ and $V_3 = V_e \cap V_S$. Then, it is obvious that $\Delta(e|\cdot) = \Delta(V_1|\cdot) + \Delta(V_2|\cdot) + \Delta(V_3|\cdot)$ for S and T. For V_1, we have $\Delta(V_1|S) = \Delta(V_1|T) = \sum_{w \in V_1} \sigma_i(w, e)$ because $V_1 \cap V_S = \varnothing$ and $V_1 \cap V_T = \varnothing$. For V_2, we have $\Delta(V_2|S) = \sum_{w \in V_2} \sigma_i(w, e)$ and $\Delta(V_2|T) = \sum_{w \in V_2} \max\big(0, \sigma_i(w, e) - \max_{e' \in T} \sigma_i(w, e')\big)$ as $V_2 \cap V_S = \varnothing$ and $V_2 \subseteq V_T$. Obviously, we can acquire $\Delta(V_2|S) \geq \Delta(V_2|T)$ as well. For V_3, we have $\Delta(V_3|S) = \sum_{w \in V_3} \max\big(0, \sigma_i(w, e) - \max_{e' \in S} \sigma_i(w, e')\big)$ and $\Delta(V_3|T) = \sum_{w \in V_3} \max\big(0, \sigma_i(w, e) - \max_{e' \in T} \sigma_i(w, e')\big)$ because of $V_3 \subseteq V_S \subseteq V_T$. Because $\max_{e' \in S} \sigma_i(w, e') \leq \max_{e' \in T} \sigma_i(w, e')$ for $S \subseteq T$, $\Delta(V_3|S) \geq \Delta(V_3|T)$. According to the above results, we prove $\Delta(e|S) \geq \Delta(e|T)$ and thus $\mathcal{R}_i(\cdot)$ is submodular. □

A.2 Proof of Lemma 3.7

Proof. First, given any $e' \in E \setminus S$ and $S \subseteq E$, for each $e \in I_t(S)$, we have $p_i(S \cup \{e'\} \rightsquigarrow e) - p_i(S \rightsquigarrow e) = 1 - \big(1 - p_i(S \rightsquigarrow e)\big) \cdot \big(1 - p_i(e' \rightsquigarrow e)\big) - p_i(S \rightsquigarrow e) = p_i(e' \rightsquigarrow e) \cdot \big(1 - p_i(S \rightsquigarrow e)\big) \geq 0$ for $p_i(S \rightsquigarrow e) \in [0, 1]$.

Second, given any $S \subseteq T \subseteq E$, for each $e \in I_t(T)$, we have $p_i(S \rightsquigarrow e) \leq p_i(T \rightsquigarrow e)$ for $e.ref \cap I_t(S) \subseteq e.ref \cap I_t(T)$. Therefore, for any $e' \in E \setminus T$, we have $p_i(S \cup \{e'\} \rightsquigarrow e) - p_i(S \rightsquigarrow e) = 1 - \big(1 - p_i(S \rightsquigarrow e)\big) \cdot \big(1 - p_i(e' \rightsquigarrow e)\big) - p_i(S \rightsquigarrow e) = p_i(e' \rightsquigarrow e) \cdot \big(1 - p_i(S \rightsquigarrow e)\big) \geq p_i(e' \rightsquigarrow e) \cdot \big(1 - p_i(T \rightsquigarrow e)\big) = p_i(T \cup \{e'\} \rightsquigarrow e) - p_i(T \rightsquigarrow e)$. Finally, because $\mathcal{I}_{i,t}(S) = \sum_{e \in I_t(S)} p_i(S \rightsquigarrow e)$ and $p_i(\cdot \rightsquigarrow e)$ is monotone and submodular, $\mathcal{I}_{i,t}(\cdot)$ is monotone and submodular as well. □

A.3 Proof of Theorem 4.2

Proof. The sequence of estimations Φ for OPT is in range $[\delta_{max}, 2 \cdot k \cdot \delta_{max}]$. Due to the monotonicity and submodularity of $f(\cdot, \mathbf{x})$, we have $\mathsf{OPT} \in [\delta_{max}, k \cdot \delta_{max}]$. Therefore, there must exist some $\phi \in \Phi$ such that $(1-\varepsilon)\mathsf{OPT} \leq \phi \leq \mathsf{OPT}$.

Next, we discuss two cases for such ϕ and S_ϕ.

Case 1 ($|S_\phi| = k$). For each $e \in S_\phi$, we have $\Delta(e|S') \geq \frac{\phi}{2k}$ where S' is the subset of S_ϕ when e is added. Therefore, $f(S_\phi, \mathbf{x}) \geq k \cdot \frac{\phi}{2k} \geq (\frac{1}{2} - \varepsilon)\mathsf{OPT}$.

Case 2 ($|S_\phi| < k$). For each $e \in S^* \setminus S_\phi$, if e has been evaluated by MTTS, it is excluded from S_ϕ because $\Delta(e|S') < \frac{\phi}{2k}$ where S' is the subset of S_ϕ when e is evaluated; if e has not been evaluated by MTTS, it holds that $\Delta(e|S) \leq \delta(e, \mathbf{x}) < \mathsf{UB}(\mathbf{x}) < \mathsf{TH} \leq \frac{\phi}{2k}$. Thus, $\mathsf{OPT} - f(S_\phi, \mathbf{x}) \leq f(S^* \cup S_\phi, \mathbf{x}) - f(S_\phi, \mathbf{x}) \leq \sum_{e \in S^* \setminus S_\phi} \Delta(e|S_\phi) \leq k \cdot \frac{\phi}{2k} \leq \frac{1}{2} \cdot \mathsf{OPT}$. Equivalently, $f(S_\phi, \mathbf{x}) \geq \frac{1}{2} \cdot \mathsf{OPT}$.

In both cases, we have $f(S_{ts}, \mathbf{x}) \geq f(S_\phi, \mathbf{x}) \geq (\frac{1}{2} - \varepsilon)\mathsf{OPT}$. □

A.4 Proof of Theorem 4.4

Proof. There are two cases when MTTD is terminated. Here, we discuss them separately.

Case 1 **($|S_{td}| = k$).** Let $S_j = \{e_1, \ldots, e_j\}$ ($j \in [1, k]$) be the subset of S_{td} after the first j elements are added and $S_0 = \varnothing$. Assume that e_{j+1} is added to S_j in the round with threshold τ. It holds that $\Delta(e_{j+1}|S_j) \geq \tau$ and $\Delta(e|S_j) < \frac{\tau}{1-\varepsilon}, \forall e \notin S_j \cup \{e_{j+1}\}$. Then, we have $\Delta(e_{j+1}|S_j) \geq (1-\varepsilon)\Delta(e|S_j), \forall e \in S^* \setminus S_j$. By summing up the above inequality for $e \in S^* \setminus S_j$, we have $|S^* \setminus S_j| \cdot \Delta(e_{j+1}|S_j) \geq (1-\varepsilon)\sum_{e \in S^* \setminus S_j} \Delta(e|S_j)$. Thus, we get $\Delta(e_{j+1}|S_j) \geq \frac{1-\varepsilon}{|S^* \setminus S_j|} \cdot \sum_{e \in S^* \setminus S_j} \Delta(e|S_j) \geq \frac{1-\varepsilon}{k} \cdot \sum_{e \in S^* \setminus S_j} \Delta(e|S_j)$. Due to the submodularity of $f(\cdot, \mathbf{x})$, we have $\sum_{e \in S^* \setminus S_j} \Delta(e|S_j) \geq \mathsf{OPT} - f(S_j, \mathbf{x})$. Thus,$\Delta(e_{j+1}|S_j) = f(S_{j+1}, \mathbf{x}) - f(S_j, \mathbf{x}) \geq \frac{1-\varepsilon}{k}(\mathsf{OPT} - f(S_j, \mathbf{x}))$. Equivalently, we acquire $f(S_{j+1}, \mathbf{x}) - \mathsf{OPT} \geq (1 - \frac{1-\varepsilon}{k})(f(S_j, \mathbf{x}) - \mathsf{OPT})$. Substituting S_{j+1} by $S_k, \ldots, S_1$ for k times, we prove $f(S_{td}, \mathbf{x}) = f(S_k, \mathbf{x}) \geq \big(1 - (1 - \frac{1-\varepsilon}{k})^k\big) \cdot \mathsf{OPT} \geq (1 - e^{-(1-\varepsilon)})\mathsf{OPT} \geq (1 - \frac{1}{e} - \varepsilon)\mathsf{OPT}$.

Case 2 **($|S_{td}| < k$).** We have $\Delta(e|S_{td}) < \tau' = f(S_{td}, \mathbf{x}) \cdot \frac{\varepsilon}{k}, \forall e \in S^* \setminus S_{td}$. Therefore, $\mathsf{OPT} - f(S_{td}, \mathbf{x}) \leq \sum_{e \in S^* \setminus S_{td}} \Delta(e|S_{td}) \leq \sum_{e \in S^* \setminus S_{td}} f(S_{td}, \mathbf{x}) \cdot \frac{\varepsilon}{k} \leq \varepsilon \cdot f(S_{td}, \mathbf{x})$. Therefore, we acquire $f(S_{td}, \mathbf{x}) \geq \frac{\mathsf{OPT}}{1+\varepsilon} \geq (1-\varepsilon)\mathsf{OPT}$.

In both cases, $f(S_{td}, \mathbf{x}) \geq (1 - \frac{1}{e} - \varepsilon)\mathsf{OPT}$. □

REFERENCES

[1] Manoj K. Agarwal and Krithi Ramamritham. 2017. Real Time Contextual Summarization of Highly Dynamic Data Streams. In *EDBT*. 168–179.

[2] Ashwinkumar Badanidiyuru, Baharan Mirzasoleiman, Amin Karbasi, and Andreas Krause. 2014. Streaming submodular maximization: massive data summarization on the fly. In *KDD*. 671–680.

[3] Ashwinkumar Badanidiyuru and Jan Vondrak. 2014. Fast algorithms for maximizing submodular functions. In *SODA*. 1497–1514.

[4] Jingwen Bian, Yang Yang, Hanwang Zhang, and Tat-Seng Chua. 2015. Multimedia Summarization for Social Events in Microblog Stream. *IEEE Trans. Multimedia* 17, 2 (2015), 216–228.

[5] David M. Blei. 2012. Probabilistic topic models. *Commun. ACM* 55, 4 (2012), 77–84.

[6] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *JMLR* 3 (2003), 993–1022.

[7] Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy J. Lin. 2012. Earlybird: Real-Time Search at Twitter. In *ICDE*. 1360–1369.

[8] Chun Chen, Feng Li, Beng Chin Ooi, and Sai Wu. 2011. TI: an efficient indexing mechanism for real-time search on tweets. In *SIGMOD*. 649–660.

[9] Lisi Chen and Gao Cong. 2015. Diversity-Aware Top-k Publish/Subscribe for Text Stream. In *SIGMOD*. 347–362.

[10] Jacob Cohen. 1968. Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit. *Psychol. Bull.* 70, 4 (1968), 213–220.

[11] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 2002. Maintaining Stream Statistics over Sliding Windows. *SIAM J. Comput.* 31, 6 (2002), 1794–1813.

[12] Alessandro Epasto, Silvio Lattanzi, Sergei Vassilvitskii, and Morteza Zadimoghaddam. 2017. Submodular Optimization Over Sliding Windows. In *WWW*. 421–430.

[13] Uriel Feige. 1998. A Threshold of ln *n* for Approximating Set Cover. *J. ACM* 45, 4 (1998), 634–652.

[14] Wen Hua, Zhongyuan Wang, Haixun Wang, Kai Zheng, and Xiaofang Zhou. 2015. Short text understanding through lexical-semantic analysis. In *ICDE*. 495–506.

[15] Ravi Kumar, Benjamin Moseley, Sergei Vassilvitskii, and Andrea Vattani. 2013. Fast greedy algorithms in mapreduce and streaming. In *SPAA*. 1–10.

[16] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne M. van Briesen, and Natalie S. Glance. 2007. Cost-effective outbreak detection in networks. In *KDD*. 420–429.

[17] Yuchen Li, Zhifeng Bao, Guoliang Li, and Kian-Lee Tan. 2015. Real time personalized search on social networks. In *ICDE*. 639–650.

[18] Yuchen Li, Ju Fan, Dongxiang Zhang, and Kian-Lee Tan. 2017. Discovering Your Selling Points: Personalized Social Influential Tags Exploration. In *SIGMOD*. 619–634.

[19] Yuchen Li, Dongxiang Zhang, Ziquan Lan, and Kian-Lee Tan. 2016. Context-aware advertisement recommendation for high-speed social news feeding. In *ICDE*. 505–516.

[20] Hui Lin and Jeff A. Bilmes. 2010. Multi-document Summarization via Budgeted Maximization of Submodular Functions. In *NAACL*. 912–920.

[21] Zhiyuan Liu, Yuzhou Zhang, Edward Y. Chang, and Maosong Sun. 2011. PLDA+: Parallel latent dirichlet allocation with data placement and pipeline processing. *ACM Trans. Intell. Syst. Technol.* 2, 3 (2011), 26:1–26:18.

[22] George L. Nemhauser, Laurence A. Wolsey, and Marshall L. Fisher. 1978. An analysis of approximations for maximizing submodular set functions – I. *Math. Program.* 14, 1 (1978), 265–294.

[23] Andrei Olariu. 2014. Efficient Online Summarization of Microblogging Streams. In *EACL*. 236–240.

[24] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report. Stanford InfoLab.

[25] Zhaochun Ren, Oana Inel, Lora Aroyo, and Maarten de Rijke. 2016. Time-aware Multi-Viewpoint Summarization of Multilingual Social Text Streams. In *CIKM*. 387–396.

[26] Zhaochun Ren, Shangsong Liang, Edgar Meij, and Maarten de Rijke. 2013. Personalized time-aware tweets summarization. In *SIGIR*. 513–522.

[27] Lidan Shou, Zhenhua Wang, Ke Chen, and Gang Chen. 2013. Sumblr: continuous summarization of evolving tweet streams. In *SIGIR*. 533–542.

[28] Alexander Shraer, Maxim Gurevich, Marcus Fontoura, and Vanja Josifovski. 2013. Top-k Publish-Subscribe for Social Annotation of News. *PVLDB* 6, 6 (2013), 385–396.

[29] Liangjun Song, Ping Zhang, Zhifeng Bao, and Timos K. Sellis. 2017. Continuous Summarization over Microblog Threads. In *DASFAA*. 511–526.

[30] Karthik Subbian, Charu C. Aggarwal, and Jaideep Srivastava. 2016. Querying and Tracking Influencers in Social Streams. In *WSDM*. 493–502.

[31] Nguyen Thanh Tam, Matthias Weidlich, Duong Chi Thang, Hongzhi Yin, and Nguyen Quoc Viet Hung. 2017. Retaining Data from Streams of Social Platforms with Minimal Regret. In *IJCAI*. 2850–2856.

[32] Jie Tang, Jimeng Sun, Chi Wang, and Zi Yang. 2009. Social influence analysis in large-scale networks. In *KDD*. 807–816.

[33] Leong-Hou U, Junjie Zhang, Kyriakos Mouratidis, and Ye Li. 2017. Continuous Top-k Monitoring on Document Streams. *IEEE Trans. Knowl. Data Eng.* 29, 5 (2017), 991–1003.

[34] Yanhao Wang, , Yuchen Li, and Kian-Lee Tan. 2018. Efficient Representative Subset Selection over Sliding Windows. *IEEE Trans. Knowl. Data Eng.* (2018). https://doi.org/10.1109/TKDE.2018.2854182

[35] Yanhao Wang, Qi Fan, Yuchen Li, and Kian-Lee Tan. 2017. Real-Time Influence Maximization on Dynamic Social Streams. *PVLDB* 10, 7 (2017), 805–816.

[36] Zhenhua Wang, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. 2015. On Summarization and Timeline Generation for Evolutionary Tweet Streams. *IEEE Trans. Knowl. Data Eng.* 27, 5 (2015), 1301–1315.

[37] Lingkun Wu, Wenqing Lin, Xiaokui Xiao, and Yabo Xu. 2013. LSII: An indexing structure for exact real-time search on microblogs. In *ICDE*. 482–493.

[38] Xiaohui Yan, Jiafeng Guo, Yanyan Lan, and Xueqi Cheng. 2013. A biterm topic model for short texts. In *WWW*. 1445–1456.

[39] Dongxiang Zhang, Yuchen Li, Ju Fan, Lianli Gao, Fumin Shen, and Heng Tao Shen. 2017. Processing Long Queries Against Short Text: Top-*k* Advertisement Matching in News Stream Applications. *ACM Trans. Inf. Syst.* 35, 3 (2017), 28:1–28:27.

[40] Dongxiang Zhang, Liqiang Nie, Huanbo Luan, Kian-Lee Tan, Tat-Seng Chua, and Heng Tao Shen. 2017. Compact Indexing and Judicious Searching for Billion-Scale Microblog Retrieval. *ACM Trans. Inf. Syst.* 35, 3 (2017), 27:1–27:24.

[41] Wayne Xin Zhao, Jing Jiang, Jianshu Weng, Jing He, Ee-Peng Lim, Hongfei Yan, and Xiaoming Li. 2011. Comparing Twitter and Traditional Media Using Topic Models. In *ECIR*. 338–349.

[42] Hao Zhuang, Rameez Rahman, Xia Hu, Tian Guo, Pan Hui, and Karl Aberer. 2016. Data Summarization with Social Contexts. In *CIKM*. 397–406.

Comparative Analysis of Content-based Personalized Microblog Recommendations [Experiments and Analysis]

Efi Karra Taniskidou
University of California, Irvine
ekarrata@uci.edu

George Papadakis
University of Athens
gpapadis@di.uoa.gr

George Giannakopoulos
NCSR Demokritos
ggianna@iit.demokritos.gr

Manolis Koubarakis
University of Athens
koubarak@di.uoa.gr

ABSTRACT

Microblogging platforms constitute a popular means of real-time communication and information sharing. They involve such a large volume of user-generated content that their users suffer from an information deluge. To address it, numerous recommendation methods have been proposed to organize the posts a user receives according to her interests. The content-based methods typically build a text-based model for every individual user to capture her tastes and then rank the posts in her timeline according to their similarity with that model. Even though content-based methods have attracted lots of interest in the data management community, there is no comprehensive evaluation of the main factors that affect their performance. These are: *(i)* the representation model that converts an unstructured text into a structured representation that elucidates its characteristics, *(ii)* the source of the microblog posts that compose the user models, and *(iii)* the type of user's posting activity. To cover this gap, we systematically examine the performance of 9 state-of-the-art representation models in combination with 13 representation sources and 3 user types over a large, real dataset from Twitter comprising 60 users. We also consider a wide range of 223 plausible configurations for the representation models in order to assess their robustness with respect to their internal parameters. To facilitate the interpretation of our experimental results, we introduce a novel taxonomy of representation models. Our analysis provides novel insights into the main factors determining the performance of content-based recommendation in microblogs.

1 INTRODUCTION

Microblogging platforms enable the instant communication and interaction between people all over the world. They allow their users to post messages in real-time, often carelessly and ungrammatically, through any electronic device, be it a mobile phone or a personal computer. They also allow for explicit connections between users so as to facilitate the dissemination and consumption of information. These characteristics led to the explosive growth of platforms like Twitter (www.twitter.com), Plurk (www.plurk.com), Sina Weibo (www.weibo.com) and Tencent Weibo (http://t.qq.com).

Their popularity has led to an information deluge: the number of messages that are transmitted on a daily basis on Twitter alone has jumped from 35 million tweets in 2010 to over 500 million in 2017 [29]. Inevitably, their users are constantly overwhelmed with information. As we also show in our experiments, this situation cannot be ameliorated by presenting the new messages in chronological order; the relatedness with users' interests is typically more important than the recency of a post. Equally ineffective is the list of trending topics, where the same messages are presented to all users, irrespective of their personal interests.

A more principled solution to information deluge is offered by *Personalized Microblog Recommendation* (PMR). Its goal is to capture users' preferences so as to direct their attention to the messages that better match their personal interests. A plethora of works actually focuses on *Content-based PMR* [6, 13, 15, 31, 40, 41, 45], which typically operates as follows: first, it builds a document model for every individual post in the training set by extracting features from its textual content. Then, it constructs a user model by assembling the document models that capture the user's preferences. Subsequently, it compares the user model to the models of recommendation candidates (documents) with a similarity measure. The resulting similarity scores are used to rank all candidates in descending order, from the highest to the lowest score, thus placing the most relevant ones at the top positions. Finally, the ranked list is presented to the user.

Content-based PMR is a popular problem that has attracted a lot of attention in the data management community [1, 15, 29–31, 55]. However, the experimental results presented in the plethora of relevant works are not directly comparable, due to the different configurations that are used for several important, yet overlooked parameters.

The core parameter is the *representation model* that is used for converting a set of unstructured texts into a structured representation that reveals their characteristics. The available options range from traditional vector space models [41, 65] to topic models [39, 50]. Also crucial is the *representation source*, i.e., the source of the microblog posts that compose user models. Common choices include the user's tweets [36] together with their retweets [17, 23, 41, 56] as well as the posts of followers [31, 50, 65] and followees [15, 31, 39]. Another decisive factor is the posting activity of a user, i.e., whether she is an information producer or seeker [5, 35]. Other parameters include the novel challenges posed by the short, noisy, multilingual content of microblogs as well as the external information that enriches their textual content, e.g., concepts extracted from Wikipedia [41] or the content of a Web page, whose URL is mentioned in a post [1].

Despite their significance, little effort has been allocated on assessing the impact of these parameters on Content-based PMR. To cover this gap, we perform a thorough experimental analysis that investigates the following questions: *Which representation model is the most effective for recommending short, noisy, multilingual microblog posts? Which is the most efficient one? How robust is the performance of each model with respect to its configuration?*

Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.18

Which representation source yields the best performance? How does the behavior of individual users affect the performance of Content-based MPR? We leave the investigation of external information as a future work, due to the high diversity of proposed approaches, which range from language-specific word embeddings like Glove [49] to self-reported profile information [21].

To investigate the above questions, we focus on Twitter, the most popular microblogging service worldwide, with over 335 million active users per month.[1] We begin with a categorization of the representation sources and the users it involves, based on its special social graph: every user u_1 is allowed to unilaterally follow another user u_2, with u_1 being a *follower* of u_2, and u_2 a *followee* for u_1; if u_2 follows back u_1, the two users are *reciprocally connected.* Then, we list the novel challenges posed by the short, noisy, user-generated tweets in comparison with the long and curated content of traditional documents. We also introduce a taxonomy of representation models that provides insights into their endogenous characteristics. Based on it, we briefly present nine state-of-the-art representation models and apply them to a dataset of 60 real Twitter users (partitioned into three different categories) in combination with 223 parameter configurations, three user types and 13 representation sources. Finally, we discuss the experimental outcomes in detail, interpreting the impact of every parameter on the performance of Content-based PMR.

In short, we make the following contributions:

• We perform the first systematic study for content-based recommendation in microblogging platforms, covering nine representation models, 13 representation sources and three user types. We have publicly released our code along with guidelines for our datasets[2].

• We organize the main representation models according to their functionality in a novel taxonomy with three main categories and two subcategories. In this way, we facilitate the understanding of our experimental results, given that every (sub-)category exhibits different behavior.

• We examine numerous configurations for every representation model, assessing their relative effectiveness, robustness and time efficiency. Our conclusions facilitate their fine-tuning and use in real recommender systems.

The rest of the paper is structured as follows: Section 2 provides background knowledge on Twitter and formally defines the recommendation task we are tackling in this work. In Section 3, we present our taxonomy of representation models and describe the state-of-the-art models we consider. We present the setup of our experiments in Section 4 and their results in Section 5. Section 6 discusses relevant works, while Section 7 concludes the paper along with directions for future work.

2 PRELIMINARIES

Representation Sources. We consider five sources of tweets for modeling the preferences of a Twitter user, u:

(i) The past *retweets* of u, $R(u)$, which are the tweets she has received from her followees and has reposted herself. Apparently, their subjects have captured u's attention so intensely that she decided to share them with her followers.

(ii) All past *tweets* of u except her retweets, $T(u)$. They enclose themes u is interested in chatting or in informing her followers.

(iii) All (re)tweets of u's followees, $E(u) = \bigcup_{u_i \in e(u)}(R(u_i) \cup T(u_i))$, where $e(u) = \{u_1, \ldots, u_k\}$ is the set of her followees. $E(u)$ models a user as an *information seeker*, who actively and explicitly follows accounts providing interesting information [15, 31, 39].

(iv) All (re)tweets of u's followers, $F(u) = \bigcup_{u_i \in f(u)}(R(u_i) \cup T(u_i))$, where $f(u) = \{u_1, \ldots, u_m\}$ stands for the set of her followers. Given that they have actively decided to follow u, due to the interest they find in her posts, $F(u)$ models u as an *information producer* [31, 50, 65].

(v) All (re)tweets of u's reciprocal connections, $C(u)$=$E(u)$∩$F(u)$. Unlike the unilateral following relationship in Twitter, reciprocal connections may indicate users with very high affinity, thus providing valuable information for user modeling.

Note that all these *atomic* representation sources are complementary, as they cover different aspects of the activity of a particular user and her network. For this reason, $T(u)$ is typically combined with $R(u)$ [17, 23, 41, 56], with only rare exceptions like [36], which considers $T(u)$ in isolation. In this work, we consider not only $T(u) \cup R(u)$ (TR for short), but also the seven remaining pairwise combinations, which give rise to the following *composite* representation sources: $T(u) \cup E(u)$, $R(u) \cup E(u)$, $E(u) \cup F(u)$, $T(u) \cup F(u)$, $R(u) \cup F(u)$, $T(u) \cup C(u)$, and $R(u) \cup C(u)$. For simplicity, we denote them by TE, RE, EF, TF, RF, TC and RC, respectively.

Twitter Challenges. Tweets have some special characteristics that distinguish them from other conventional domains and pose major challenges to representation models [17, 44, 48].

(C1) Sparsity. Tweets are short, comprising up to 280 characters for most languages, except for Chinese, Korean and Japanese, where the length limit is 140 characters. As a result, they lack sufficient content for user and document modeling.

(C2) Noise. The real-time nature of Twitter forces users to tweet quickly, without taking into account the frequent grammar errors and misspellings; these are corrected in subsequent tweets.

(C3) Multilingualism. The global popularity of Twitter has led to a high diversity in tweet languages. This renders inapplicable most language-specific pre-processing techniques, such as stemming and lemmatization. Even tokenization becomes difficult: unlike the European ones, many Asian languages, such as Chinese, Japanese, and Korean, do not use spaces or other punctuation in order to distinguish consecutive words.

(C4) Non-standard language. Tweets offer an everyday informal communication, which is unstructured, ungrammatical or simply written in slang; words are abbreviated (e.g., "gn" instead of "goodnight"), or contain emoticons, such as :), hashtags like #edbt and emphatic lengthening (e.g., "yeeees" instead of "yes").

We consider the above challenges when discussing the outcomes of our experimental analysis in Section 5.

User Categories. Twitter users are typically classified into three categories [5, 35]: *(i) Information Producers* (IP) are those users who tweet and retweet more frequently than they receive updates from their followees, (ii) *Information Seekers* (IS) are those users who are less active compared to their followees, and (iii) *Balanced Users* (BU) are those exhibiting a symmetry between the received and the posted messages.

To quantify these categories, we use the ratio of *outgoing* to *incoming tweets.* For a particular user u, the former involve the tweets and retweets she posts, $R(u) \cup T(u)$, while the latter comprise the tweets and retweets of her followees, $E(u)$. Dividing the outgoing with the incoming tweets, we get the *posting ratio.* Apparently, BU is the set of users with a posting ratio close to 1, i.e., $|R(u) \cup T(u)| \simeq |E(u)|$. To ensure significantly different behavior for the other two categories, we define IP as the set of users with a posting ratio higher than 2, thus indicating that they

[1] https://en.wikipedia.org/wiki/Twitter, last accessed on 14 January 2019.
[2] See https://github.com/efikarra/text-models-twitter for more details.

post at least twice as many tweets as they receive. Symmetrically, we define IS as the set of users with a posting ratio lower than 0.5, receiving at least twice as many tweets as those they publish.

Problem Definition. The task of Content-based PMR aims to distinguish a user's incoming messages, R∪T, into *irrelevant* and *relevant* ones. A common assumption in the literature [17, 33, 41, 56], which allows for large-scale evaluations, is that the relevant messages are those that are retweeted by the user that receives them, an action that implicitly indicates her interests – intuitively, a user forwards a tweet to her followers after carefully reading it and appreciating its content.

In this context, Content-based PMR is usually addressed via a *ranking-based recommendation algorithm* [11, 17, 20], which aims to rank relevant posts higher than the irrelevant ones. More formally, let D denote a set of documents, U a set of users and M the representation space, which is common for both users and microblog posts (e.g., M could be the real vector space). Given a user model $UM : U \rightarrow M$ and a document model $DM : D \rightarrow M$, we define this type of recommendation algorithms as follows:

Definition 2.1. **Content-based Personalized Microblog Recommendation** learns a ranking model $RM : M \times M \rightarrow \mathbb{R}$, which, given a user u and a set of testing posts $D_{test}(u) = \{d_1, \ldots, d_k\} \subseteq E(u)$, calculates the ranking scores $RM(UM(u), DM(d_i)), \forall i \in \{1 \ldots k\}$, and returns a list with $D_{test}(u)$ ranked in decreasing score.

We consider a user u as the set of documents that stem from a particular representation source s, i.e., $s(u)$, and we build a different $UM_s(u)$ for each s. Given a set of users U along with a representation source $s = \{s(u) : u \in U\}$ and a set of labeled data $D^s_{tr} = \{D^s_{tr}(u) : u \in U\}$, where $D^s_{tr}(u) = \{(d_i, l_i), d_i \in s(u), l_i \in L\}$ and $D^s_{tr}(u) \cap D_{test}(u) = \emptyset$, the recommendation algorithm is trained as follows for each individual source s: (i) for each $u \in U$, we learn $UM_s(u)$ and $DM(d_i)$ for each $d_i \in D^s_{tr}(u)$, and (ii) we train RM on $\bigcup_{u \in U}\{(UM_s(u), DM(d_i)) : i \in \{1, \ldots |D^s_{tr}(u)|\}$.

As a recommendation algorithm, we employ the one commonly used in the literature for ranking-based PMR [15, 31, 40]: in essence, RM is a similarity function and $RM(UM(u), DM(d_i))$ measures the similarity of document model d_i and user model u.

3 REPRESENTATION MODELS

3.1 Taxonomy

We first present a taxonomy of the state-of-the-art representation models based on their internal functionality, i.e., the way they handle the order of n-grams. Figure 1 presents an overview of their relations, with every edge **A** $\rightarrow$ **B** indicating that model B shares the same n-grams with A, but uses them in a different way when representing users and documents. Based on these relations, we identify three main categories of models:

(i) **Context-agnostic models** disregard the order of n-grams that appear in a document, when building its representation. E.g., the token-based models of this category yield the same representation for the phrases "Bob sues Jim" and "Jim sues Bob".

(ii) **Local context-aware models** take into account the relative order of characters or tokens in the n-grams that lie at the core of their document representations. Yet, they lose contextual information, as they ignore the ordering of n-grams themselves. Continuing our example, the token-based models of this category are able to distinguish the bigram "Bob sues" from "sues Bob", but cannot capture the bigrams that precede or follow it.

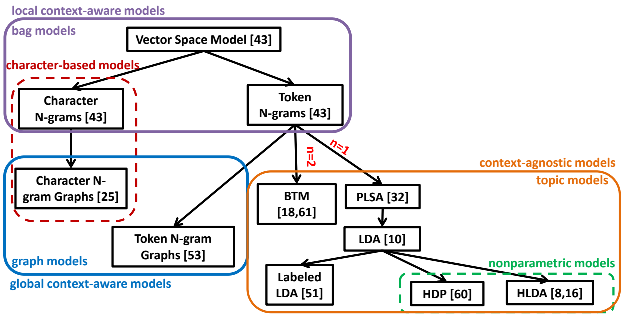


Figure 1: Taxonomy of representation models.

(iii) **Global context-aware models** incorporate into their representations both the relative ordering of tokens or characters in an n-gram and the overall ordering between n-grams in a document. Continuing our example, models of this category distinguish "Bob sues" from "sues Bob" and know that the former is followed by "sues Jim", while the latter is preceded by "Jim sues".

The first category comprises the *topic models* [9]. These are representation models that uncover the latent semantic structure of texts by determining the topics they talk about. The topics and their proportions in a document are considered as the hidden structure of a topic model, which can be discovered by exclusively analyzing the observed data, i.e., the individual tokens (words) in the original text. In general, they regard each document as a mixture of multiple topics, where each topic constitutes a set of co-occurring words. As a result, topic models are typically described in terms of probabilistic modeling, i.e., as generative processes that produce documents, words and users from distributions over the inferred topics [7].

A subcategory of context-agnostic (topic) models pertains to *nonparametric models* [16, 60], which adapt their representations to the structure of the training data. They allow for an unbounded number of parameters that grows with the size of the training data, whereas the parametric models are restricted to a parameter space of fixed size. For example, the nonparametric topic models assume that the number of topics is a-priori unknown, but can be inferred from the documents themselves, while the parametric ones typically receive a fixed number of topics as an input parameter before training.

The category of local context-aware models encompasses the bag models [43], which impose a strict order within n-grams: every n-gram is formed by a specific sequence of characters or tokens and, thus, two n-grams with different sequences are different, even if they involve the same characters or tokens; for example, the bigrams "ab" and "ba" are treated as different. The only exception is the token-based vector model with n=1, which essentially considers individual words; its context-agnostic functionality actually lies at the core of most topic models.

Finally, the category of global context-aware models includes the n-gram graph models, which represent every document as a graph in a language-agnostic way [25]: every node corresponds to an n-gram and edges connect pairs of n-grams co-located within a window of size n, with their weights indicating the co-occurrence frequency. These weighted edges allow graph models to capture global context, going beyond the local context of the bag models that use the same n-grams. Recent works suggest that the graph models outperform their bag counterparts in various tasks [42], which range from Information Retrieval [53] to Document Classification [48].

It should be stressed at this point that the bag and graph models share the second subcategory of our taxonomy: the *character-based models*. These operate at a finer granularity than their token-based counterparts, thus being more robust to noise [48]. For example, consider the words “tweet” and “twete”, where the second one is misspelled; they are considered different in all types of token-based models, but for character bi-gram models, they match in three out of four bigrams. In this way, the character-based models capture more precisely the actual similarity between noisy documents.

In the following, we collectively refer to local and global context-aware models as *context-based models*.

3.2 State-of-the-art Models

We now elaborate on the main representation models that are widely used in the literature. To describe topic models, we use the common notation that is outlined in Table 1, while their generative processes are illustrated in Figures 2(i)-(vi) using *plate diagrams*: shaded nodes correspond to the observed variables, the unshaded ones to the hidden variables, the arrows connect conditionally dependent variables, and finally, the plates denote repeated sampling for the enclosed variables for as many times as the number in the right bottom corner.

Bag Models [43]. There are two types of n-grams, the character and the token ones. These give rise to two types of bag models: the *character n-grams model* (**CN**) and the *token n-grams model* (**TN**). Collectively, they are called *bag* or *vector space models*, because they model a document d_i as a vector with one dimension for every distinct n-gram in a corpus of documents D: $DM(d_i) = (w_{i1}, \dots, w_{im})$, where m stands for the *dimensionality* of D (i.e., the number of distinct n-grams in it), while w_{ij} is the weight of the j^{th} dimension that quantifies the importance of the corresponding n-gram for d_i.

The most common weighting schemes are:

(i) Boolean Frequency (**BF**) assigns binary weights that indicate the absence or presence of the corresponding n-gram in d_i. More formally, $BF(t_j, d_i)$=1 if the n-gram t_j of the j^{th} dimension appears in document d_i, and 0 otherwise.

(ii) Term Frequency (**TF**) sets weights in proportion to the number of times the corresponding n-grams appear in document d_i. More formally, $TF(t_j, d_i)$=f_j/N_{d_i}, where f_j stands for the occurrence frequency of t_j in d_i, while N_{d_i} is the number of n-grams in d_i, normalizing TF so as to mitigate the effect of different document lengths on the weights.

(iii)Term Frequency-Inverse Document Frequency (**TF-IDF**) discounts the TF weight for the most common tokens in the entire corpus D, as they typically correspond to noise (i.e., stop words). Formally, $TF\text{-}IDF(t_j, d_i) = TF(t_j, d_i) \cdot IDF(t_j)$, where $IDF(t_j)$ is the inverse document frequency of the n-gram t_j, i.e., $IDF(t_j) = \log |D|/(|\{d_k \in D : t_j \in d_k\}| + 1)$. In this way, high weights are given to n-grams with high frequency in d_i, but low frequency in D.

To construct the bag model for a specific user u, we aggregate the vectors corresponding to the documents that capture u's interests. The end result is a weighted vector $(a(w_1), ..., a(w_m))$, where $a(w_j)$ is the *aggregation function* that calculates the importance of the j^{th} dimension for u.

The main aggregation functions are: *(i)* the sum of weights, i.e., $a(w_j) = \sum_{d_i \in D} w_{ij}$. *(ii)* the centroid of unit (normalized) document vectors, i.e., $a(w_j) = \frac{1}{|D|} \cdot \sum_{d_i \in D} \frac{w_{ij}}{||DM(d_i)||}$, where $||DM(d_i)||$ is the magnitude of $DM(d_i)$. *(iii)* the Rocchio algorithm, i.e., $a(w_j) = \alpha/|D^p| \cdot \sum_{d_i \in D^p} w_{ij}/||DM(d_i)|| - \beta/|D^n| \cdot \sum_{d_i \in D^n} w_{ij}/||DM(d_i)||$, where D^p and D^n are the sets of positive (relevant) and negative (irrelevant) documents in D, respectively, while $\alpha, \beta \in [0, 1]$ control the relative importance of positive and negative examples, respectively, such that $\alpha + \beta = 1.0$ [43].

Table 1: The notation describing topic models.

Symbol	Meaning
D	the corpus of the input documents
$\|D\|$	the number of input documents
d	an individual document in D
N_d	the number of words in d, i.e., the *document length*
U	the set of users
u	an individual user in U
$N_{d,u}$	the number of words in document d of user u
D_u	the documents posted by a user u
$\|D_u\|$	the number of documents posted by user u
V	the *vocabulary* of D (i.e., the set of words it includes)
$\|V\|$	the number of distinct words in D
w	an individual word in V
$w_{d,n}$	the word at position n in d
Z	the set of latent topics
$\|Z\|$	the number of latent topics
z	an individual topic in Z
$z_{d,n}$	the topic assigned to the word at position n in document d
θ_d	the multinomial distribution of document d over Z, $\{P(z\|d)\}_{z \in Z}$
$\theta_{d,z}$	the probability that topic z appears in document d, $P(z\|d)$
ϕ_z	the multinomial distribution of topic z over V, $\{P(w\|z)\}_{w \in V}$
$\phi_{z,w}$	the probability that word w appears in topic z, $P(w\|z)$
$Dir(\alpha)$	a symmetric Dirichlet distribution parameterized by α

To compare two bag models, $DM(d_i)$ and $DM(d_j)$, one of the following similarity measures is typically used:

(i) Cosine Similarity (**CS**) measures the cosine of the angle of the weighted vectors. Formally, it is equal to their dot product similarity, normalized by the product of their magnitudes: $CS(DM(d_i), DM(d_j)) = \sum_{k=1}^{m} w_{ik} w_{jk}/||DM(d_i)||/||DM(d_j)||$.

(ii) Jaccard Similarity (**JS**) treats the document vectors as sets, with weights higher than (equal to) 0 indicating the presence (absence) of the corresponding n-gram. On this basis, it defines as similarity the ratio between the sizes of set intersection and union: $JS(DM(d_i), DM(d_j))$=$|DM(d_i) \cap DM(d_j)|/|DM(d_i) \cup DM(d_j)|$.

(iii) Generalized Jaccard Similarity (**GJS**) extends JS so that it takes into account the weights associated with every n-gram: $GJS(DM(d_i), DM(d_j))$= $\sum_{k=1}^{m} min(w_{ik}, w_{jk})/\sum_{k=1}^{m} max(w_{ik}, w_{jk})$. Note that for BF weights, GJS is identical with JS.

Graph Models [25, 53]. There are two graph models, one for each type of n-grams, i.e., *token n-gram graphs* (**TNG**) [53] and *character n-gram graphs* (**CNG**) [25]. Both models represent each document d as an undirected graph G_d that contains one vertex for each n-gram in d. An edge connects every pair of vertices/n-grams that co-occur within a window of size n in d. Every edge is weighted according to the co-occurrence frequency of the corresponding n-grams. Thus, the graphs incorporate *contextual information* in the form of n-grams' closeness.

To construct the model for a user u, we merge the graphs of the documents representing u's interests using the update operator, which is described in [26]. To compare graph models, we can use the following graph similarity measures [25]: *(i) Containment Similarity* (**CoS**) estimates the number of edges shared by two graph models, G_i and G_j, regardless of the corresponding weights (i.e., it merely estimates the portion of common n-grams in the original texts). Formally: $CoS(G_i, G_j) = \sum_{e \in G_i} \mu(e, G_j)/min(|G_i|, |G_j|)$, where $|G|$ is the size of graph G, and $\mu(e, G) = 1$ if $e \in G$, or 0 otherwise. *(ii) Value Similarity* (**VS**) extends CoS by considering the weights of common edges. Formally, using w_e^k for the weight of edge e in G_k:

$VS(G_i, G_j) = \sum_{e \in (G_i \cap G_j)} \frac{min(w_e^i, w_e^j)}{max(w_e^i, w_e^j) \cdot max(|G_i|, |G_j|)}$. *(iii) Normalized Value Similarity* (**NS**) extends VS by mitigating the impact of imbalanced graphs, i.e., the cases where the comparison between a large graph with a much smaller one yields similarities close to 0: $NS(G_i, G_j) = \sum_{e \in (G_i \cap G_j)} min(w_e^i, w_e^j)/max(w_e^i, w_e^j)/min(|G_i|, |G_j|)$.

Probabilistic Latent Semantic Analysis (PLSA) [32]. This model assigns a topic z to every observed word w in a document d. Thus, every document is modeled as a distribution over multiple topics, assuming that the observed variables w and d are conditionally independent given the unobserved topic z: $P(w|d, z) = P(w|z)$. For an observed pair (w, d), the joint probability distribution is: $P(w, d) = P(d) \cdot \sum_z P(w, z|d) = P(d) \cdot \sum_z P(z|d) \cdot P(w|z) = P(d) \cdot \sum_z \theta_{d,z} \cdot \phi_{z,w}$, where $\theta_{d,z}$ stands for the probability that a topic z appears in document d, while $\phi_{z,w}$ denotes the probability that a word w appears in topic z (see Table 1).

Figure 2(i) depicts the generative process of PLSA: *(1)* Select a document d with probability $P(d)$. *(2)* For each word position $n \in \{1, \dots, N_d\}$: *(a)* Select a topic $z_{d,n}$ from distribution θ_d. *(b)* Select the word $w_{d,n}$ from distribution $\phi_{z_{d,n}}$. Note that $P(d)$ is the frequency of d in the corpus, thus being uniform in practice.

In total, PLSA should estimate $|D| \cdot |Z| + |Z| \cdot |V|$ parameters: $\theta_d = \{P(z|d)\}_{z \in Z}$ for each $d \in D$ and $\phi_z = \{P(w|z)\}_{w \in V}$ for each $z \in Z$. Consequently, the number of parameters grows linearly with the number of documents, leading to overfitting [10].

Latent Dirichlet Allocation (LDA) [10]. Unlike PLSA, which regards each document d as a list of probabilities θ_d, LDA assigns a random variable of $|Z|$ parameters with a Dirichlet prior to distribution θ_d. In a latter variant, a $|V|$-parameter variable with a Dirichlet prior was also assigned to ϕ_z [28]. The number of topics $|Z|$ is given as a parameter to the model and raises the following issue: the smaller the number of topics is, the broader and more inaccurate is their content, failing to capture the diverse themes discussed in the corpus; in contrast, for large values of $|Z|$, the model is likely to overfit, learning spurious word co-occurrence patterns [58].

Figure 2(ii) shows the generative process of LDA: *(1)* For each topic $z \in Z$, draw a distribution ϕ_z from $Dir(\beta)$. *(2)* For each document $d \in D$: *(a)* Select a distribution θ_d from $Dir(\alpha)$. *(b)* For each word position $n \in \{1, \dots, N_d\}$: *(i)* Draw a topic $z_{d,n}$ from distribution θ_d. *(ii)* Draw the word $w_{d,n}$ from distribution $\phi_{z_{d,n}}$.

Note that the hyperparameters α and β of the Dirichlet priors on θ and ϕ, respectively, distinguish LDA from PLSA. The former denotes the frequency with which a topic is sampled for a document, while the latter shows the frequency of a word in a topic, before actually observing any words in D.

Labeled LDA (LLDA) [51]. This is a supervised variant of LDA that characterizes a corpus D with a set of observed labels Λ. Each document d is modeled as a multinomial distribution of labels from $\Lambda_d \subseteq \Lambda$. Subsequently, each word $w \in d$ is picked from a distribution ϕ_z of some label z contained in Λ_d. Besides the observed labels, LLDA can also use $|Z|$ latent topics for all documents, assigning the labels "Topic 1",..., "Topic $|Z|$" to each document $d \in D$ [50].

Figure 2(iii) presents the generative process of LLDA: *(1)* For each topic $z \in Z$, draw a distribution ϕ_z from $Dir(\beta)$. *(2)* For each document $d \in D$: *(a)* Construct distribution Λ_d by selecting each topic $z \in Z$ as a label based on a Bernoulli distribution with parameter Φ_z. *(b)* Select a multinomial distribution θ_d over Λ_d from $Dir(\alpha)$. *(c)* For each word position $n \in \{1, \dots, N_d\}$: *(i)* Draw a label $z_{d,n}$ from distribution θ_d. *(ii)* Draw the word $w_{d,n}$ from distribution $\phi_{z_{d,n}}$. Note that the prior probability of adding a topic z to $\Lambda_d(\Phi_z)$ is practically superfluous, as Λ_d is observed for each document d.

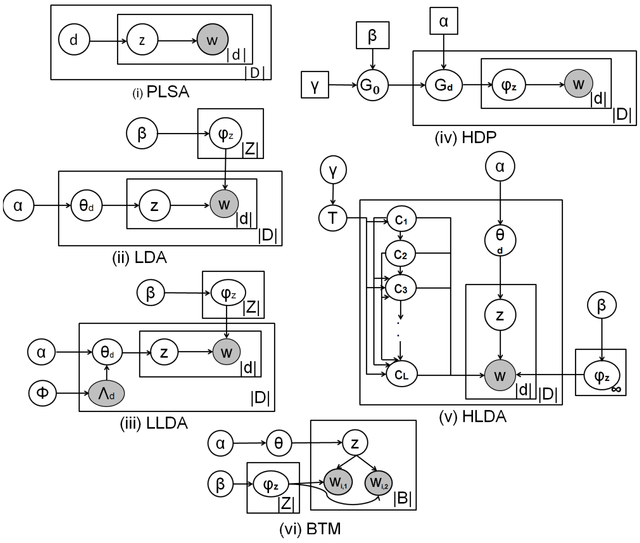


Figure 2: Plate diagrams for: (i) Probabilistic Latent Semantic Analysis (PLSA), (ii) Latent Dirichlet Allocation (LDA), (iii) Labeled LDA (LLDA), (iv) Hierarchical Dirichlet Process (HDP), (v) Hierarchical LDA (HLDA), and (vi) Biterm Topic Model (BTM).

Hierarchical Dirichlet Process (HDP) [60]. This Bayesian nonparametric model is crafted for clustering the observations of a group into mixing components. In PMR, each document corresponds to a group, the words of the document constitute the observations within the group, and the topics comprise the mixing components in the form of distributions over words.

Two are the main properties of HDP: *(i)* The number of mixing components is countably infinite and unknown beforehand. This is achieved by assigning a random variable G_j to the j^{th} group distributed according to $DP(\alpha, G_0)$, where DP stands for a Dirichlet Process, which is a probability distribution over distributions (i.e., samples from a DP are probability distributions themselves). G_0 is the base probability distribution, playing the role of the mean around which distributions are sampled by DP, while α is called concentration parameter and can be thought as an inverse variance. G_0 also follows a $DP(\gamma, H)$. *(ii)* The groups share the same components. This is achieved by linking the DPs of all groups, under the same G_0. More formally, HDP is defined as: $G_0|\gamma, H \sim DP(\gamma, H)$ and $G_j|\alpha, G_0 \sim DP(\alpha, G_0)\ \forall j$.

Figure 2(iv) shows the generative process for 1 hierarchical level: *(1)* Draw G_0 from $DP(\gamma, H)$, where H is a $Dir(\beta)$. G_0 provides an unbounded number of ϕ_z distributions, i.e., topics that can be assigned to any document $d \in D$ [60]. *(2)* For each document $d \in D$: *(a)* Associate a subset of distributions ϕ_z with d, by drawing G_d from $DP(\alpha, G_0)$. *(b)* For each word position $n \in \{1, \dots, N_d\}$: *(i)* Pick a distribution $\phi_{z_{d,n}}$ from G_d. *(ii)* Draw the word $w_{d,n}$ from $\phi_{z_{d,n}}$.

Note that it is straightforward to add more hierarchical levels to HDP, by exploiting its recursive nature. For example, in multiple corpora, where documents are grouped into broader categories, topics shared between categories are revealed and can be compared with topics shared between individual documents [60].

Hierarchical LDA (HLDA) [8, 16]. This model extends LDA by organizing the topics in Z into a hierarchical tree such that every tree node represents a single topic. The broader a topic is, the higher is its level in the tree, with the most specific topics

Table 2: Statistics for each user group in our dataset.

	IS	BU	IP	All Users
Users	20	20	9	60
Outgoing tweets (TR)	47,659	48,836	42,566	192,328
Minimum per user	1,100	766	1,602	766
Mean per user	2,383	2,442	4,730	3,205
Maximum per user	6,406	8,025	17,761	17,761
Retweets (R)	27,344	32,951	38,013	140,649
Minimum per user	840	445	1,198	445
Mean per user	1,367	1,648	4,224	2,344
Maximum per user	2,486	6,814	17,761	17,761
Incoming tweets (E)	390,638	49,566	10,285	484,698
Minimum per user	8,936	696	525	525
Mean per user	19,532	2,478	1,143	8,078
Maximum per user	53,003	7,726	1,985	53,003
Follower's tweets (F)	665,778	166,233	50,330	1,391,579
Minimum per user	1,074	110	348	110
Mean per user	33,289	8,312	5,592	23,193
Maximum per user	144,398	52,318	33,639	447,639

assigned to the leaves. Although the tree levels are fixed, the branching factor is inferred from the data, leading to a nonparametric functionality. Each document is modeled as a path from the root to a leaf and its words are generated by the topics across this path. Hence, every document representation is derived from the topics of a single path, rather than all topics in Z (as in LDA).

Specifically, HLDA is based on the *Chinese Restaurant Process* (CRP), which is a distribution over partitions of integers. CRP assumes the existence of a Chinese restaurant with an infinite number of tables. The first customer selects the first table, while the n^{th} customer selects a table based on the following probabilities [8]: $\mathbf{P_1}$=P(occupied $table_i$|previous customers)=$n_i/(n\text{-}1{+}\gamma)$, $\mathbf{P_2}$=P(next unoccupied $table_i$|previous customers)=$\gamma/(n\text{-}1{+}\gamma)$, where γ is a parameter controlling the possibility for a new customer to sit to an occupied or an empty table, and n_i are the customers already seated on table i. The placement of M customers produces a partition of M integers.

In fact, HLDA relies on the *nested Chinese Restaurant Process* (nCRP), which extends CRP by building a infinite hierarchy of Chinese restaurants. It assumes that there exists an infinite number of Chinese restaurants, each with an infinite number of tables. One of the restaurants is the root and every restaurant's table has a label, pointing to another restaurant – a restaurant cannot be referred by more than one label in the hierarchy. To illustrate the functionality of this process, assume a customer that visits the restaurants for L days. Starting from the root, she forms a path of L restaurants, one per day, following the labels of the tables she has chosen to sit based on P_1 and P_2. The repetition of this process for M customers yields an L-level hierarchy of restaurants.

Figure 2(v) presents the generative process of HLDA, with T denoting an infinite tree drawn from nCRP(γ) (i.e., the infinite set of possible L-level paths): *(1)* For each restaurant z in T, draw a distribution ϕ_z from $Dir(\beta)$. *(2)* For each document $d \in D$: *(a)* Draw an L-level path from T as follows: Let $c_{d,1}$ be the root restaurant. For each level $l \in \{2, \ldots, L\}$ pick a table from restaurant $c_{d,l-1}$ using P_1 and P_2 and set $c_{d,l-1}$ to refer to the restaurant $c_{d,l}$, which is indicated by that table. *(b)* Select a distribution θ_d over $\{1, \ldots, L\}$ from $Dir(\alpha)$. *(c)* For each word position $n \in \{1, \ldots, N_d\}$: *(i)* Draw a level $l_{d,n} \in \{1, \ldots, L\}$ from θ_d. *(ii)* Draw the word $w_{d,n}$ from distribution $\phi_{z_{d,n}}$, where $z_{d,n}$ is the topic corresponding to the restaurant $c_{d,l_{d,n}}$.

Biterm Topic Model (BTM) [18, 61]. At the core of this model lies the notion of *biterm*, which is an unordered pair of words that are located in close distance within a given text. In short documents, close distance means that both words belong to the same document, whereas in longer texts, it means that they co-occur within a window of tokens that is given as a parameter to BTM. Based on this notion, BTM addresses the sparsity in short texts like tweets (Challenge C1) in two ways: *(i)* It models the word co-occurrence for topic learning *explicitly* by considering biterms (rather than *implicitly* as in LDA, where word co-occurrence is captured by drawing a document's words from topics of the same topic distribution θ_d). *(ii)* It considers the entire corpus D as a set of biterms B, extracting word patterns from the entire training set (rather than an individual document). Overall, BTM assumes that the corpus consists of a mixture of topics and directly models the biterm generation from these topics.

Figure 2(vi) shows the generative process of BTM: *(1)* For each topic $z \in Z$, draw a distribution ϕ_z from $Dir(\beta)$. *(2)* For the entire corpus D, select a multinomial distribution θ over Z from $Dir(\alpha)$. *(3)* For each biterm position n in the entire corpus $\{1, \ldots, |B|\}$: *(a)* Draw a topic z_n from θ. *(b)* Draw two words $w_{n,1}$, $w_{n,2}$ from ϕ_{z_n}.

Note that BTM does not contain a generation process for documents. The distribution θ_d for an individual document $d \in D$ is inferred from the formula $P(z/d) = \sum_{b \in d} P(z/b) \cdot P(b/d)$, which presumes that the document-level topic proportions can be derived from the document-level generated biterms [18, 61].

Other models. There is a plethora of topic models in the literature. Most of them lie out of the scope of our experimental analysis, because they encapsulate external or non-textual information. For example, [54] and [12] incorporate temporal information from users' activity, while [21] proposes a representation model called *badges*, which combines the posting activity of Twitter users with self-reported profile information. Other topic models are incompatible with the ranking-based recommendation algorithm for Content-based PMR. For example, Twitter-LDA [64] and Dirichlet Multinomial Mixture Model [47] assign a single topic to every tweet, thus yielding too many ties when ranking document models in decreasing similarity score - all tweets with the same inferred topic are equally similar with the user model.

Using Topic Models. To build a topic model for a particular user u, we average the distributions corresponding to the documents that capture her preferences. To compare user models with document models, we use the cosine similarity. To address the four Twitter challenges, which hamper the functionality of topic models due to the scarcity of word co-occurrence patterns, we apply two different pooling schemes to the training data: *(i)* the aggregation on users, called *User Pooling* (UP), where all tweets posted by the same user are considered as a single document, and *(ii)* the aggregation on hashtags, called *Hashtag Pooling* (HP), where all tweets annotated with the same hashtag form a single document (the tweets without any hashtag are treated as individual documents). We also consider *No Pooling* (NP), where each tweet is considered as an individual document. Finally, we estimate the parameters of all topic models using *Gibbs Sampling* [24], except for PLSA, which uses *Expectation Maximization* [19].

4 EXPERIMENTAL SETUP

All methods were implemented in Java 8 and performed in a server with Intel Xeon E5-4603@2.20 GHz (32 cores) and 120GB RAM, running Ubuntu 14.04.

Table 3: The 10 most frequent languages in our dataset, which collectively cover 1,879,470 tweets (91% of all tweets).

	English	Japanese	Chinese	Portuguese	Thai	French	Korean	German	Indonesian	Spanish
Total Tweets	1,710,919	71,242	35,356	14,416	13,964	12,895	10,220	5,038	4,339	1,081
Relative Frequency	82.71%	3.44%	1.71%	0.70%	0.68%	0.62%	0.49%	0.24%	0.21%	0.05%

Our dataset was derived from a large Twitter corpus that captures almost 30% of all public messages published on Twitter worldwide between June 1, 2009 and December 31, 2009 [62]. Although slightly dated, recent studies have verified that core aspects of the users activity in Twitter remain unchanged over the years (e.g., retweet patterns for individual messages and users) [29]. Most importantly, this dataset can be combined with a publicly available snapshot of the entire social graph of Twitter as of August, 2009 (https://an.kaist.ac.kr/traces/WWW2010.html). Given that every record includes the raw tweet along with the corresponding usernames and timestamps, we can simulate the tweet feed of every user during the second half of 2009 with very high accuracy. To ensure that there is a critical mass of tweets for all representation sources and a representative set of testing documents for every user, we removed from our dataset those users that had less than three followers and less than three followees. We also discarded all users with less than 400 retweets.

From the remaining users, we populated each of the three user types we defined in Section 2 with 20 users. For IS, we selected the 20 users with the lowest posting ratios, for BU the 20 users with the closest to 1 ratios, and for IP, the 20 users with the highest ratios. The difference between the maximum ratio for IS (0.13) and the minimum one for BU (0.76) is large enough to guarantee significantly different behavior. However, the maximum ratio among BU users is 1.16, whereas the minimum one among IP users is 1.20, due to the scarcity of information providers in our dataset. This means that the two categories are too close, a situation that could introduce noise to the experimental results. To ensure distinctive behavior, we placed in the IP group the nine users that have a posting ratio higher than 2. The remaining 11 users with the highest ratios are included in the group All Users, which additionally unites IS, BU and IP to build a large dataset with 2.07 million tweets and 60 users, in total. The technical characteristics of the four resulting user groups appear in Table 2.

Each user has a different train set, which is determined by the representation source; for example, the train set for T contains all tweets of a user, except for her retweets. In contrast, the test set of every user is independent of the representation source and contains her incoming tweets; the retweeted ones constitute the *positive examples* and the rest are the *negative examples* [17, 33, 41, 56]. However, the former are sparse and imbalanced across time in our dataset. Following [17], we retain a reasonable proportion between the two classes for each user by placing the 20% most recent of her retweets in the test set. The earliest tweet in this sample splits each user's timeline in two phases: the *training* and the *testing phase.* Based on this division, we sampled the negative data as follows: for each positive tweet in the test set, we randomly added four negative ones from the testing phase [17]. Accordingly, the train set of every representation source is restricted to all the tweets that fall in the training phase.

Based on this setup, we build the user models as follows: for the bag and graph models, we learn a separate model $UM_s(u)$ for every combination of a user u and a representation source s, using the corresponding train set. For the topic models, we first learn a single model $M(s)$ for each representation source, using the train set of all 60 users. Then, we use $M(s)$ to infer the distributions over topics for the training tweets of u that stem from s. Finally, the user model $UM_s(u)$ is constructed by computing the centroid of u's training vectors/tweets [50].

Note that for all models, we converted the raw text of all training and testing tweets into lowercase. For all token-based models, we tokenized all tweets on white spaces and punctuation, we squeezed repeated letters and we kept together URLs, hashtags, mentions and emoticons. We also removed the 100 most frequent tokens across all training tweets, as they practically correspond to stop words. We did not apply any language-specific pre-processing technique, as our dataset is multilingual.

In more detail, Table 3 presents the 10 most frequent languages in our dataset along with the number of tweets that correspond to them. To identify them, we first cleaned all tweets from hashtags, mentions, URLs and emoticons in order to reduce the noise of non-English tweets. Then, we aggregated the tweets per user (UP) to facilitate language detection. Finally, we automatically detected the prevalent language in every pseudo-document (i.e., user) [46] and assigned all relevant tweets to it. As expected, English is the dominant language, but our corpus is highly multilingual, with 3 Asian languages ranked within the top 5 positions. This multilingual content (Challenge C3) prevents us from boosting the performance of representation models with language-specific pre-processing like stemming, lemmatization and part-of-speech tagging, as is typically done in the literature [4, 17, 65]. Instead, our experimental methodology is language-agnostic.

Performance Measures. We assess the effectiveness of representation models using the *Average Precision* (**AP**) of a user model $UM_s(u)$, which is the average of the Precision-at-n ($P@n$) values for all retweets. Formally [17, 41]: $AP(UM_s(u)) = 1/|R(u)| \cdot \sum_{n=1}^{N} P@n \cdot RT(n)$, where $P@n$ is the proportion of the top-n ranked tweets that have been retweeted, $RT(n)$=1 if n is a retweet and 0 otherwise, N is the size of the test set, and $|R(u)|$ is the total number of retweets in the test set. Thus, AP expresses the performance of a representation model over an individual user. To calculate the performance of a user group U, we define *Mean Average Precision* (**MAP**) as the average AP over all users in U.

To assess the robustness of a representation model with respect to its internal configuration, we consider its *MAP deviation*, i.e., the difference between the maximum and the minimum MAP of the considered parameter configurations over a specific group of users. The lower the MAP deviation, the higher the robustness.

To estimate the time efficiency of representation models, we employ two measures: *(i)* The *training time* (**TTime**) captures the aggregated modeling time that is required for modeling all 60 users. For topic models, this also includes the time that is required for training once the model $M(s)$ from the entire train set. *(ii)* The *testing time* (**ETime**) expresses the total time that is required for processing the test set of all 60 users, i.e., to compare all user models with their testing tweets and to rank the latter in descending order of similarity. For a fair comparison, we do not consider works that parallelize representation models (e.g., [16, 57, 63]), as most models are not adapted to distributed processing.

Parameter Tuning. For each representation model, we tried a wide range of meaningful parameter configurations. They are reported in Tables 4 and 5. In total, we employed 223 different

Table 4: Configurations of the 5 context-agnostic (topic) models. CS, NP, UP and HP stand for cosine similarity, no pooling, user pooling and hashtag pooling, respectively.

	LDA	LLDA	BTM	HDP	HLDA
#Topics	{50,100,150,200}			-	-
#Iterations	{1,000, 2,000}		1,000		
Pooling	{NP, UP, HP}				UP
α	50/#Topics			1.0	{10, 20}
β	0.01			{0.1, 0.5}	
γ	-	-	-	1.0	{0.5, 1.0}
Aggregation function	{centroid, Rocchio}				
Similarity measure	CS				
Fixed parameters	-	-	r=30	-	levels=3
#Configurations	48	48	24	12	16

configurations – excluding those violating the *memory constraint* (the memory consumption should be less than 32GB RAM), or the *time constraint* ($TTime$ should be less than 5 days). As a result, we excluded PLSA from our analysis, since it violated the memory constraint for all configurations we considered.

In total, 9 representation models were tested in our experiments. For LDA, LLDA and BTM, α and β were tuned according to [58]. For LLDA, the number of topics refers to the latent topics assigned to every tweet in addition to the tweet-specific labels. For the latter, we followed [50], using: *(i)* one label for every hashtag that occurs more than 30 times in the training tweets, *(ii)* the question mark, *(iii)* 9 categories of emoticons (i.e., "smile", "frown", "wink", "big grin", "heart", "surprise", "awkward" and "confused"), and *(iv)* the @user label for the training tweets that mention a user as the first word. Most of these labels were quite frequent in our corpus and, thus, we considered 10 variations for them, based on [50]. For example, "frown" produced the labels: :(-0 to :(-9. The only labels with no variations are the hashtag ones and the emoticons "big grin", "heart", "surprise" and "confused".

For HLDA, we did not employ the pooling strategies NP and HP and more than three hierarchical levels, as these configurations violated the time constraint. Hence, we only varied α and γ.

For BTM, we selected 1,000 iterations following [61]. For individual tweets, we set the context window (r), i.e., the maximum distance between two words in a biterm, equal to the size of the tweet itself. For the large pseudo-documents in user and hashtag pooling, we set r=30 based on [61]; for greater values, BTM conveys no significant improvement over LDA, since the larger the distance between the words in a biterm is, the more irrelevant are their topics. Larger window sizes yield higher $TTime$, too.

For bag models, some configuration combinations are invalid: JS is applied only with BF weights, GJS only with TF and TF-IDF, and the character-based n-grams, CN, are not combined with TF-IDF. Also, BF is exclusively coupled with the sum aggregation function, which in this case is equivalent to applying the boolean operator OR among the individual document models. For the Rocchio algorithm, we set α=0.8 and β=0.2, and we used only the CS similarity measure in combination with the TF and TF-IDF weights for those representation sources that contain both positive and negative examples, namely C, E, TE, RE, TC, RC and EF.

Table 5: Configurations of the 4 context-based models. Remember that BF and TF stand for boolean and term frequency weights, respectively, while (G)JS, CoS, VS, and NS denote the (generalized) Jaccard, the containment, the value and the normalized value graph similarities, resp.

	TN	CN	TNG	CNG
n	{1,2,3}	{2,3,4}	{1,2,3}	{2,3,4}
Weighting scheme	{BF,TF,TF-IDF}	{BF,TF}	-	-
Aggregation function	{sum, centroid, Rocchio}		-	-
Similarity measure	{CS, JS, GJS}		{CoS, VS, NS}	
#Configurations	36*	21*	9	9

* excluding invalid configuration combinations

5 EXPERIMENTAL ANALYSIS

Effectiveness & Robustness. To assess the performance of the 9 representation models, we measure their *Mean MAP*, *Min MAP* and *Max MAP*, i.e., their average, minimum and maximum MAP, respectively, over all relevant configurations for a particular combination of a user type and a representation source. The outcomes for All Users are presented in Figure 3, which considers the 5 individual representation sources along with the 3 combinations that achieve the best average performance. Due to space limitations, the remaining 5 combinations along with the figures for IP, BU and IS are only presented in the extended version of the paper [59]. Nevertheless, we consider them in the discussion of our experimental results.

Every diagram also reports the MAP for two baseline methods: (i) the *Chronological Ordering* (CHR), which ranks the test set from the latest tweet (first position) to the earliest one (last position), and (ii) the *Random Ordering* (RAN), which sorts the test set in an arbitrary order. For RAN, we performed 1,000 iterations per user and considered the overall average per user type.

Starting with All Users in Figure 3, we observe that TNG consistently outperforms all other models across all representation sources. Its Mean MAP fluctuates between 0.625 (EF) and 0.784 (T). The second most effective model is TN, whose Mean MAP ranges from 0.541 (EF) to 0.673 (T). The dominance of TNG over TN is statistically significant (p<0.05) and should be attributed to its ability to capture the relations between neighboring token n-grams through the weighted edges that connect them. In this way, TNG incorporates global contextual information into its model and inherently alleviates sparsity (Challenge C1). Instead, TN exclusively captures local contextual information in the form of token sequences. The same holds for Challenges C2 and C4: both models fail to identify misspelled or non-standard token n-grams, but TNG is better suited to capture their patterns by encapsulating their neighborhood.

Regarding *robustness*, TN is more sensitive to its configuration. Its MAP deviation ranges from 0.359 (TF, RF) to 0.444 (EF), while for TNG, it fluctuates between 0.096 (T) and 0.281 (EF). In all cases, the difference between the two models is statistically significant (p<0.05). TNG is superior, because its performance is fine-tuned by just two parameters: the size of n-grams (n) and the similarity measure. TN additionally involves the aggregation function and the weighting scheme, increasing drastically its possible configurations. In general, *every configuration parameter acts as a degree of freedom for a representation model; the higher the overall number of parameters is, the more flexible the model gets and the less robust it is expected to be with respect to its configuration.*

Comparing the character-based instantiations of bag and graph models, we notice that their difference in Mean Map is statistically insignificant. For CNG, it fluctuates between 0.368 (TF) and 0.477 (R), while for CN, it ranges from 0.334 (TF) to 0.436 (RC). This implies that *there is no benefit in considering global contextual*

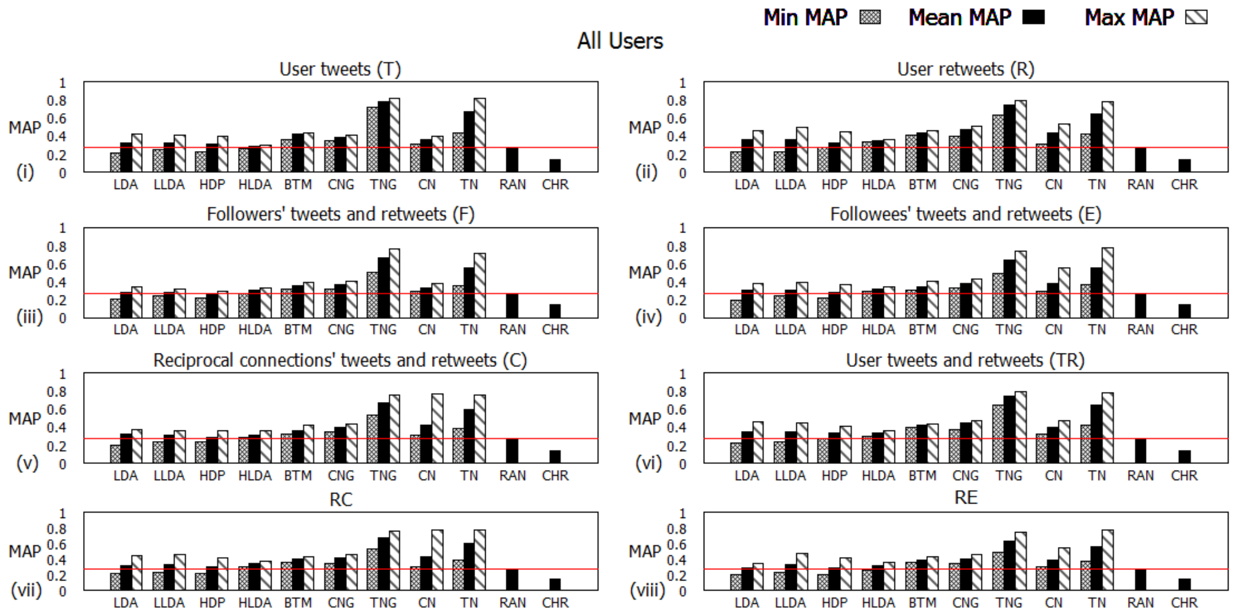


Figure 3: Effectiveness of the nine representation models over All Users in combination with the five individual representation sources and their three best performing pairwise combinations with respect to MAP. Higher bars indicate better performance. The red line corresponds to the performance of the best baseline, RAN.

information for representation models that are based on character n-grams. The strength of these models lies in the local contextual information that is captured in the sequence of characters.

Regarding robustness, the relative sensitivity of character-based models exhibits the same patterns as their token-based counterparts: the bag models are significantly less robust than the graph ones, due to their larger number of parameters and configurations. In more detail, the MAP deviation ranges from 0.061 (T) to 0.114 (RF) for CNG and from 0.077 (T) to 0.476 (RC) for CN.

Concerning the topic models, we observe that BTM consistently achieves the highest effectiveness across all representation sources. Its Mean MAP fluctuates between 0.340 (EF) and 0.434 (R). All other topic models exhibit practically equivalent performance: their highest Mean MAP ranges from 0.337 (for HDP with TR) to 0.360 (for LLDA with R), whereas their lowest Mean MAP fluctuates between 0.265 (for HDP over F) and 0.270 (for LLDA over TF). As a result, HDP, LDA, LLDA and HLDA outperform only CHR to a large extent. This indicates that *recency alone constitutes an inadequate criterion for recommending content in microblogging platforms. Any model that considers the personal preferences of a user offers more accurate suggestions.*

Compared to the second baseline (RAN), HDP, LDA, LLDA and HLDA are more effective, but to a minor extent. The Mean MAP of RAN amounts to 0.270, thus, some configurations of these topic models perform consistently worse than RAN, across all representation sources. Most of these configurations correspond to the absence of pooling (NP), where every tweet is considered as an individual document. In these settings, these four topic models fail to extract distinctive patterns from any representation source, producing noisy user and document models. This suggests that *sparsity is the main obstacle to most topic models.*

Regarding robustness, we can distinguish the 5 topic models into two categories: the first one involves 3 models that are highly sensitive to their configuration, namely LDA, LLDA and HDP. Their MAP deviation starts from 0.119, 0.075, and 0.077, respectively, and raises up to 0.250, 0.264 and 0.211, respectively. These values are extremely high, when compared to their absolute Mean MAP. This means that *extensive fine-tuning is required for successfully applying most topic models to text-based PMR.* In contrast, MAP deviation fluctuates between 0.034 and 0.109 for both HLDA and BTM. For the former, this is probably caused by the limited number of configurations that satisfied our time constraint, while for the latter, it should be attributed to its Twitter-specific functionality.

Discussion. We now discuss the novel insights that can be deduced from our experimental analysis. We start by comparing token- with character-based models. We observe that the former are significantly more effective than the latter for both bag and graph models. At first, this seems counter-intuitive, as CNG and CN are in a better position to address Challenge C1: they extract more features from sparse documents than TNG and TN, respectively [48]. They are also better equipped to address Challenges C2 and C4: by operating at a finer granularity, they can identify similarities even between noisy and non-standard tokens. Yet, the character-based models seem to accumulate noise when aggregating individual tweet models into a user model. Their similarity measures fail to capture distinctive information about the real interests of a user, yielding high scores for many irrelevant, unseen documents. The lower n is, the more intensive is this phenomenon. In fact, most bigrams are shared by both relevant and irrelevant examples, which explains why the poorest performance corresponds to n=2 for both CNG and CN. The only advantage of character-based models over their token-based counterparts is their higher robustness. However, their lower values for MAP deviation are probably caused by their lower absolute values for MAP.

Among the topic models, BTM consistently performs best with respect to effectiveness and robustness. Its superiority stems from the two inherent characteristics that optimize its functionality for the short and noisy documents in Twitter: *(i)* it considers pairs of words (i.e., biterms), instead of individual tokens, and *(ii)* it bypasses sparsity (Challenge C1) by capturing topic patterns at the level of entire corpora, instead of extracting them from individual documents. Compared to context-based models, though, BTM is significantly less effective than the token-based bag and graph models. Its performance is very close to the character-based models, especially CNG, with their difference in terms of effectiveness and robustness being statistically insignificant.

Table 6: Performance of all 13 representation sources over the 4 user types with respect to Min(imum), Mean and Max(imum) MAP across all configurations of the 9 representation models. The rightmost column presents the average performance per user type.

		R	T	E	F	C	TR	RE	RF	RC	TE	TF	TC	EF	Average
All Users	Min MAP	0.225	0.217	0.196	0.208	0.199	0.222	0.205	0.199	0.212	0.201	0.205	0.201	0.198	0.207
	Mean MAP	0.456	0.429	0.392	0.378	0.410	0.448	0.402	0.386	0.427	0.387	0.377	0.406	0.377	0.406
	Max MAP	0.796	0.816	0.771	0.764	0.768	0.797	0.775	0.758	0.783	0.775	0.764	0.776	0.780	0.779
IS	Min MAP	0.193	0.199	0.192	0.191	0.196	0.212	0.199	0.195	0.188	0.200	0.192	0.195	0.195	0.196
	Mean MAP	0.415	0.383	0.351	0.336	0.364	0.407	0.357	0.340	0.374	0.347	0.342	0.357	0.336	0.362
	Max MAP	0.776	0.818	0.733	0.699	0.732	0.790	0.732	0.699	0.733	0.732	0.699	0.733	0.736	0.739
BU	Min MAP	0.212	0.174	0.168	0.192	0.168	0.198	0.168	0.182	0.178	0.167	0.179	0.171	0.175	0.179
	Mean MAP	0.474	0.430	0.400	0.376	0.420	0.453	0.419	0.389	0.443	0.391	0.381	0.406	0.376	0.412
	Max MAP	0.733	0.775	0.747	0.746	0.745	0.753	0.746	0.741	0.753	0.746	0.741	0.743	0.741	0.747
IP	Min MAP	0.243	0.220	0.200	0.198	0.186	0.242	0.200	0.202	0.182	0.189	0.212	0.198	0.208	0.206
	Mean MAP	0.524	0.446	0.460	0.458	0.488	0.493	0.470	0.467	0.497	0.428	0.443	0.444	0.440	0.466
	Max MAP	0.878	0.857	0.854	0.858	0.839	0.854	0.854	0.856	0.854	0.854	0.858	0.857	0.854	0.856

Generally, we can conclude that all topic models fail to improve the token-based bag and graph models in the context of Content-based PMR. Both TN and TNG achieve twice as high Mean MAP, on average, across all representation sources. The poor performance of topic models is caused by two factors: *(i)* they disregard the contextual information that is encapsulated in word ordering, and *(ii)* the Challenges C1 to C4 of Twitter. Indeed, most topic models were originally developed to extract topics from word co-occurrences in individual documents, but the sparse and noisy co-occurrence patterns in the short text of tweets reduce drastically their effectiveness.

User Types. We notice that the *relative* performance of representation models for IP, BU and IS remains practically the same as in Figure 3. Yet, there are significant differences in their *absolute* performance. On average, across all models and representation sources, IP users have higher Mean MAP than IS and BU users by 30% and 13% respectively, while BU surpasses IS by 15%. Compared to All Users, IP increases Mean MAP by 12%, BU by just 1%, while IS decreases it by 11%, on average, across all models and representation sources. These patterns are also demonstrated in the rightmost column of Table 6, which essentially expresses the average values for Min(imum), Mean and Max(imum) MAP per user type across all combinations of representation models, configurations and representation sources.

Therefore, we can conclude that *the more information an individual user produces, the more reliable are the models that represent her interests*, and vice versa: *taciturn users are the most difficult to model.* This should be expected for R and T, since the posting activity of a user increases the content that is available for building their models. However, this pattern applies to the other representation sources, too, because IP users are typically in closer contact with their followees, followers and reciprocal users, who thus can represent their interests effectively. The opposite holds for IS users. In the middle of these two extremes lie BU users, which exhibit a balanced activity in all respects.

Overall, these patterns suggest that the user categories we defined in Section 2 have a real impact on the performance of Content-based PMR. Therefore, they should be taken into account when designing a Content-based PMR approach, as different user types call for different recommendation methods.

Representation Sources. We now examine the relative effectiveness of the five representation sources and their eight pairwise combinations. Table 6 reports the performance for every combination of a user type and a representation source with respect to Min(imum), Mean and Max(imum) MAP over all configurations of the nine representation models.

Starting with individual representation sources, we observe that R consistently achieves the highest Mean MAP across all user types. This suggests that *retweets constitute the most effective means of capturing user preferences under all settings*, as users repost tweets that have attracted their attention and approval.

The second best individual source is T. This applies to all user types, except for IP, where T actually exhibits the worst performance across all individual sources. These patterns suggest that *the tweets of a user offer valuable information for her interests as long as her posting ratio is lower than 2*; such users post their messages thoughtfully, when they have something important to say. Instead, the IP users are hyperactive, posting quite frequently careless and noisy messages that do not reflect their preferences, e.g., by engaging into irrelevant discussions with other users.

Among the remaining individual sources, C achieves the best performance, followed by E and F. This pattern is consistent across all user types, with the differences being statistically significant in most cases. We can deduce, therefore, that *the reciprocal connections in Twitter reflect friendships between users that share common interests to a large extent.* Instead, the one-way connections offer weaker indications of common tastes among users, especially when they are not initiated by the ego user: the followers' posts (F) invariably result in noisy user models.

For the eight pairs of sources, we observe the following patterns: *(i)* All combinations of R with another source X result in higher performance for X, with the difference being statistically significant. Hence, R is able to enrich any representation source with valuable information about user preferences. *(ii)* All combinations of T with another source result in a (usually insignificant) lower performance in all cases but two, namely TF over IS and BU. This means that T typically conveys noisy, irrelevant information. *(iii)* For both R and T, all pairwise combinations degrade their own performance. The only exception is TR, which improves the effectiveness of T.

On the whole, we can conclude that *a user's retweets (R) should be used as the sole information source for building her model.* There is no need to combine it with T.

Time Efficiency. Figures 4(i) and (ii) depict the minimum, average and maximum values for $TTime$ and $ETime$, respectively, for every representation model across all configurations, representation sources and users. The vertical axes are logarithmic, with lower values indicating better performance.

Among the global context-aware models, we observe that on average, TNG is faster than CNG with respect to both $ETime$ and $TTime$ by 1 and 2 orders of magnitude, respectively. Similarly, among the local context-aware models, TN is faster than CN

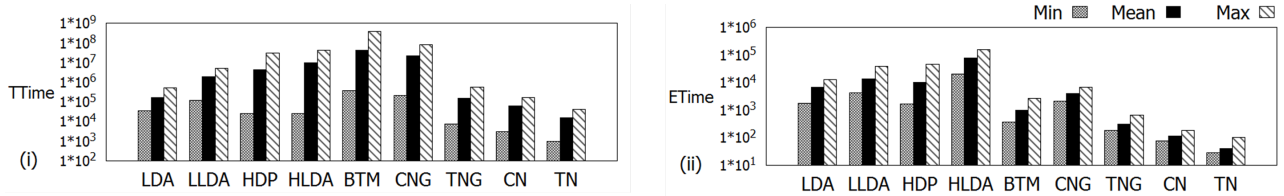


Figure 4: Time efficiency of the 9 representation models with respect to *(i)* Training Time ($TTime$), and *(ii)* Testing Time ($ETime$), on average across all configurations, representation sources and users. Lower bars indicate better performance.

by 4 and 3 times, respectively. The reason in both cases is the relative dimensionality of token- and character-based models. Typically, the latter yield a much larger feature space than the former, depending, of course, on the size of the n-grams – the larger n is, the more character n-grams are extracted from a corpus [48] and the more time-consuming is their processing.

Among the topic models, BTM exhibits the highest $TTime$. The reason is that it operates on the level of biterms, which result in a much higher dimensionality than the individual tokens that lie at the core of the other topic models. However, our analysis does not consider the most time-consuming configurations of the second worst model (HLDA), as they violated the time constraint. In practice, HLDA is expected to be slower than BTM, since its nonparametric nature lies in exploring the possible L-level hierarchical trees during topic inference. On the other extreme, LDA is the fastest topic model, while LLDA is almost 5 times faster than HDP, on average, due to its simpler models.

These patterns are slightly altered in the case of $ETime$. The worst performance by far corresponds to HLDA, due to its nonparametric functionality, while BTM turns out to be the fastest model. Unlike the other topic models, which perform Gibbs sampling for topic inference, BTM simply iterates over the biterms in a document d in order to calculate $P(z|d)$ for each topic $z \in Z$.

Comparing the model categories between them, we observe that the graph models are more time-consuming than their bag counterparts: on average, TNG and CNG are 1 and 2 orders of magnitude slower than TN and CN, respectively, for both $TTime$ and $ETime$. This should be attributed to the contextual information they incorporate in their edges, whose number is much larger in the case of CNG. Similarly, most topic models are at least 1 order of magnitude slower than their base model, TN, for both time measures, due to the time required for topic inference. Overall, *TN is consistently the most efficient representation model, due to the sparsity of tweets and the resulting low dimensionality.*

These patterns are consistent across all user types and representation sources. For the latter, we also observed that the size of the train set affects directly $TTime$ for all representation models: the more tweets are used to build a user model, the more patterns are extracted, degrading time efficiency.

6 RELATED WORK

There has been a bulk of work on recommender systems over the years [2, 3]. Most recent works focus on microblogging platforms and Social Media, employing the bag model in order to suggest new followees [13], URLs [15] and hashtags [40]. Others employ topic models for the same tasks, e.g., hashtag recommendations [27]. In the latter case, emphasis is placed on tackling sparsity through *pooling techniques*, which aggregate short texts that share similar content, or express similar ideas into lengthy pseudo-documents [4, 44]. E.g., Latent Dirichlet Allocation [10] and the Author Topic Model [52] are trained on individual messages and on messages aggregated by user and hashtag in [33].

Content-based PMR has attracted lots of interest in the data management community [14, 16, 22, 55, 63], where many works aim to improve the time efficiency of topic models. [16] parallelizes the training of HLDA through a novel concurrent dynamic matrix and a distributed tree. [14] scales up LDA through the WarpLDA algorithm, which achieves $O(1)$ time complexity per-token and fits the randomly accessed memory per-document in the L3 cache. Along with other state-of-the-art LDA samplers, this work is incorporated into LDA*, a high-end system that scales LDA training to voluminous datasets, using different samplers for various types of documents [63]. Another approach for the massive parallelization of LDA is presented in [57]. Other works facilitate real-time content recommendations in Twitter. This is done either by partitioning the social graph across a cluster in order to detect network motifs in parallel [30], or by holding the entire graph in the main memory of a single server in order to accelerate random walk-based computations on a bipartite interaction graph between users and tweets [55].

On another line of research, external resources are employed in order to augment text representation and improve their performance in various tasks. For short text clustering, Dual Latent Dirichlet Allocation learns topics from both short texts and auxiliary documents [37]. For personalized Twitter stream filtering, tweets can be transformed into RDF triples that describe their author, location and time in order to use ontologies for building user profiles [38]. User profiles can also be enriched with Wikipedia concepts [41] and with concepts from news articles that have been read by the user [34]. These approaches lie out of the scope of our analysis, which focuses on recommendations based on Twitter's internal content.

To the best of our knowledge, no prior work examines systematically Content-based PMR with respect to the aforementioned parameters, considering a wide range of options for each one.

7 CONCLUSIONS

We conclude with the following five observations about Content-based Personalized Microblog Recommendation: *(i)* The token-based vector space model achieves the best balance between effectiveness and time efficiency. In most cases, it offers the second most accurate recommendations, while involving the minimum time requirements both for training a user model and applying it to a test set. On the flip side, it involves four parameters (i.e., degrees of freedom), thus being sensitive to its configuration. *(ii)* The token n-gram graphs achieve the best balance between effectiveness and robustness. Due to the global contextual information they capture, they consistently outperform all other representation models to a significant extent, while exhibiting limited sensitivity to their configuration. Yet, they are slower than the vector space model by an order of magnitude, on average. *(iii)* The character-based models underperform their token-based counterparts, as their similarity measures cannot tackle the noise that piles up when assembling document models into user models. *(iv)* The topic models exhibit much lower effectiveness than the token-based bag models for three reasons: *1)* most of them are not crafted for the sparse, noisy and multilingual content of Twitter, *2)* they depend heavily on their configuration, and *3)* they are context-agnostic, ignoring the sequence of words in documents. Their processing is time-consuming, due to the inference of topics, requiring parallelization techniques to scale to voluminous data [16, 57, 63]. *(v)* All representation models perform best when they are built from the retweets of hyperactive users (information producers).

In the future, we plan to expand our comparative analysis to more recommendation tasks for microblogging platforms.

REFERENCES

[1] F. Abel, Q. Gao, G.-J. Houben, and K. Tao. Analyzing user modeling on twitter for personalized news recommendations. In *UMAP*, pages 1–12, 2011.

[2] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE TKDE*, 17(6):734–749, 2005.

[3] C. C. Aggarwal. *Recommender Systems - The Textbook*. Springer, 2016.

[4] D. Alvarez-Melis and M. Saveski. Topic modeling in twitter: Aggregating tweets by conversations. In *ICWSM*, pages 519–522, 2016.

[5] M. Armentano, D. Godoy, and A. Amandi. Topology-based recommendation of users in micro-blogging communities. *J. Comp. Sci. Tech.*, 27(3):624–634, 2012.

[6] M. Balabanović and Y. Shoham. Fab: content-based, collaborative recommendation. *Communications of the ACM (CACM)*, 40(3):66–72, 1997.

[7] D. M. Blei. Probabilistic topic models. *CACM*, 55(4):77–84, 2012.

[8] D. M. Blei, T. L. Griffiths, M. I. Jordan, and J. B. Tenenbaum. Hierarchical topic models and the nested chinese restaurant process. In *NIPS*, pages 17–24, 2003.

[9] D. M. Blei and J. D. Lafferty. Topic models. *Text mining: classification, clustering, and applications*, 10(71):34, 2009.

[10] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[11] C. Chen, D. Wu, C. Hou, and X. Yuan. Facet-based user modeling in social media for personalized ranking. In *ECIR*, pages 443–448, 2014.

[12] C. Chen, X. Zheng, C. Zhou, and D. Chen. Making recommendations on microblogs through topic modeling. In *WISE Workshops*, pages 252–265, 2013.

[13] J. Chen, W. Geyer, C. Dugan, M. Muller, and I. Guy. Make new friends, but keep the old: recommending people on social networking sites. In *SIGCHI*, pages 201–210, 2009.

[14] J. Chen, K. Li, J. Zhu, and W. Chen. Warplda: a cache efficient O(1) algorithm for latent dirichlet allocation. *PVLDB*, 9(10):744–755, 2016.

[15] J. Chen, R. Nairn, L. Nelson, M. Bernstein, and E. Chi. Short and tweet: experiments on recommending content from information streams. In *SIGCHI*, pages 1185–1194, 2010.

[16] J. Chen, J. Zhu, J. Lu, and S. Liu. Scalable training of hierarchical topic models. *PVLDB*, 11(7):826–839, 2018.

[17] K. Chen, T. Chen, G. Zheng, O. Jin, E. Yao, and Y. Yu. Collaborative personalized tweet recommendation. In *SIGIR*, pages 661–670, 2012.

[18] X. Cheng, X. Yan, Y. Lan, and J. Guo. Btm: Topic modeling over short texts. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):2928–2941, 2014.

[19] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38, 1977.

[20] Y. Duan, L. Jiang, T. Qin, M. Zhou, and H.-Y. Shum. An empirical study on learning to rank of tweets. In *COLING*, pages 295–303, 2010.

[21] K. El-Arini, U. Paquet, R. Herbrich, J. V. Gael, and B. A. y Arcas. Transparent user models for personalization. In *SIGKDD*, pages 678–686, 2012.

[22] A. El-Kishky, Y. Song, C. Wang, C. R. Voss, and J. Han. Scalable topical phrase mining from text corpora. *PVLDB*, 8(3):305–316, 2014.

[23] W. Feng and J. Wang. Retweet or not?: personalized tweet re-ranking. In *WSDM*, pages 577–586, 2013.

[24] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE TPAMI*, (6):721–741, 1984.

[25] G. Giannakopoulos, V. Karkaletsis, G. Vouros, and P. Stamatopoulos. Summarization system evaluation revisited: N-gram graphs. *ACM Transactions on Speech and Language Processing (TSLP)*, 5(3):5, 2008.

[26] G. Giannakopoulos and T. Palpanas. Content and type as orthogonal modeling features: a study on user interest awareness in entity subscription services. *International Journal of Advances on Networks and Services*, 3(2), 2010.

[27] F. Godin, V. Slavkovikj, W. De Neve, B. Schrauwen, and R. Van de Walle. Using topic models for twitter hashtag recommendation. In *WWW (Companion Volume)*, pages 593–596, 2013.

[28] T. L. Griffiths and M. Steyvers. Finding scientific topics. *PNAS*, 101(suppl 1):5228–5235, 2004.

[29] Q. Grossetti, C. Constantin, and et al. An homophily-based approach for fast post recommendation on twitter. In *EDBT*, pages 229–240, 2018.

[30] P. Gupta, V. Satuluri, A. Grewal, S. Gurumurthy, V. Zhabiuk, Q. Li, and J. J. Lin. Real-time twitter recommendation: Online motif detection in large dynamic graphs. *PVLDB*, 7(13):1379–1380, 2014.

[31] J. Hannon, M. Bennett, and B. Smyth. Recommending twitter users to follow using content and collaborative filtering approaches. In *ACM RecSys*, pages 199–206, 2010.

[32] T. Hofmann. Probabilistic latent semantic analysis. In *UAI*, pages 289–296, 1999.

[33] L. Hong and B. D. Davison. Empirical study of topic modeling in twitter. In *Proceedings of the first workshop on social media analytics*, pages 80–88, 2010.

[34] W. IJntema, F. Goossen, F. Frasincar, and F. Hogenboom. Ontology-based news recommendation. In *Proceedings of the EDBT/ICDT Workshops*, 2010.

[35] A. Java, X. Song, T. Finin, and B. Tseng. Why we twitter: understanding microblogging usage and communities. In *SNA-KDD*, pages 118–138, 2007.

[36] M. Jiang, P. Cui, F. Wang, W. Zhu, and S. Yang. Scalable recommendation with social contextual information. *IEEE TKDE*, 26(11):2789–2802, 2014.

[37] O. Jin, N. N. Liu, K. Zhao, Y. Yu, and Q. Yang. Transferring topical knowledge from auxiliary long texts for short text clustering. In *CIKM*, pages 775–784, 2011.

[38] P. Kapanipathi, F. Orlandi, A. P. Sheth, and A. Passant. Personalized filtering of the twitter stream. In *SPIM*, pages 6–13, 2011.

[39] Y. Kim and K. Shim. TWITOBI: A recommendation system for twitter using probabilistic modeling. In *IEEE ICDM*, pages 340–349, 2011.

[40] S. M. Kywe, T.-A. Hoang, E.-P. Lim, and F. Zhu. On recommending hashtags in twitter networks. In *SocInfo*, pages 337–350, 2012.

[41] C. Lu, W. Lam, and Y. Zhang. Twitter user modeling and tweets recommendation based on wikipedia concept graph. In *AAAI Workshops*, pages 33–38, 2012.

[42] F. D. Malliaros, P. Meladianos, and M. Vazirgiannis. Graph-based text representations: Boosting text mining, nlp and information retrieval with graphs. In *WWW Tutorials*, 2018.

[43] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[44] R. Mehrotra, S. Sanner, W. Buntine, and L. Xie. Improving lda topic models for microblogs via tweet pooling and automatic labeling. In *SIGIR*, pages 889–892, 2013.

[45] R. J. Mooney and L. Roy. Content-based book recommending using learning for text categorization. In *ACM Conference on Digital Libraries*, pages 195–204, 2000.

[46] F. R. Nakatani Shuyo, Fabian Kessler and R. Theis. Language detector. https://github.com/optimaize/language-detector, 2018.

[47] K. Nigam, A. K. McCallum, S. Thrun, and T. Mitchell. Text classification from labeled and unlabeled documents using em. *Machine Learning*, 39(2-3), 2000.

[48] G. Papadakis, G. Giannakopoulos, and G. Paliouras. Graph vs. bag representation models for the topic classification of web documents. *World Wide Web*, 19(5):887–920, 2016.

[49] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[50] D. Ramage, S. T. Dumais, and D. J. Liebling. Characterizing microblogs with topic models. In *ICWSM*, 2010.

[51] D. Ramage, D. Hall, R. Nallapati, and C. D. Manning. Labeled lda: A supervised topic model for credit attribution in multi-labeled corpora. In *EMNLP*, pages 248–256, 2009.

[52] M. Rosen-Zvi, T. Griffiths, M. Steyvers, and P. Smyth. The author-topic model for authors and documents. In *UAI*, pages 487–494, 2004.

[53] F. Rousseau and M. Vazirgiannis. Graph-of-word and TW-IDF: new approach to ad hoc IR. In *CIKM*, pages 59–68, 2013.

[54] J. Sang, D. Lu, and C. Xu. A probabilistic framework for temporal user modeling on microblogs. In *ACM CIKM*, pages 961–970, 2015.

[55] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. J. Lin. Graphjet: Real-time content recommendations at twitter. *PVLDB*, 9(13):1281–1292, 2016.

[56] K. Shen, J. Wu, Y. Zhang, Y. Han, X. Yang, L. Song, and X. Gu. Reorder user's tweets. *ACM TIST*, 4(1):6:1–6:17, 2013.

[57] A. J. Smola and S. M. Narayanamurthy. An architecture for parallel topic models. *PVLDB*, 3(1):703–710, 2010.

[58] M. Steyvers and T. Griffiths. Probabilistic topic models. *Handbook of latent semantic analysis*, 427(7):424–440, 2007.

[59] E. K. Taniskidou, G. Papadakis, G. Giannakopoulos, and M. Koubarakis. Compadrative analysis of content-based personalized microblog recommendations - experiments and analysis. *CoRR*, abs/1901.0549, 2019.

[60] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei. Hierarchical dirichlet processes. *J. Amer. Stat. Assoc.*, 101(476):1566–1581, 2006.

[61] X. Yan, J. Guo, Y. Lan, and X. Cheng. A biterm topic model for short texts. In *WWW*, pages 1445–1456, 2013.

[62] J. Yang and J. Leskovec. Patterns of temporal variation in online media. In *WSDM*, pages 177–186, 2011.

[63] L. Yu, B. Cui, C. Zhang, and Y. Shao. Lda*: A robust and large-scale topic modeling system. *PVLDB*, 10(11):1406–1417, 2017.

[64] W. X. Zhao, J. Jiang, J. Weng, J. He, E.-P. Lim, H. Yan, and X. Li. Comparing twitter and traditional media using topic models. In *ECIR*, pages 338–349, 2011.

[65] X. Zhao and K. Tajima. Online retweet recommendation with item count limits. In *IEEE/WIC/ACM WI-IAT*, pages 282–289, 2014.

Crowdsourced Truth Discovery in the Presence of Hierarchies for Knowledge Fusion

Woohwan Jung
Seoul National University
whjung@kdd.snu.ac.kr

Younghoon Kim
Hanyang University
yhkim7951@gmail.com

Kyuseok Shim*
Seoul National University
shim@kdd.snu.ac.kr

ABSTRACT

Existing works for truth discovery in categorical data usually assume that claimed values are mutually exclusive and only one among them is correct. However, many claimed values are not mutually exclusive even for functional predicates due to their hierarchical structures. Thus, we need to consider the hierarchical structure to effectively estimate the trustworthiness of the sources and infer the truths. We propose a probabilistic model to utilize the hierarchical structures and an inference algorithm to find the truths. In addition, in the knowledge fusion, the step of automatically extracting information from unstructured data (e.g., text) generates a lot of false claims. To take advantages of the human cognitive abilities in understanding unstructured data, we utilize crowdsourcing to refine the result of the truth discovery. We propose a task assignment algorithm to maximize the accuracy of the inferred truths. The performance study with real-life datasets confirms the effectiveness of our truth inference and task assignment algorithms.

1 INTRODUCTION

Automatic construction of large-scale knowledge bases is very important for the communities of database and knowledge management. Knowledge fusion (KF) [8] is one of the methods used to automatically construct knowledge bases (a.k.a. knowledge harvesting). It collects the possibly conflicting values of objects from data sources and applies *truth discovery* techniques for resolving the conflicts in the collected values. Since the values are extracted from unstructured or semi-structured data, the collected information exhibits error-prone behavior. The goal of the *truth discovery* used in knowledge fusion is to infer the true value of each object from the noisy observed values retrieved from multiple information sources while simultaneously estimating the reliabilities of the sources. Two potential applications of knowledge fusion are web source trustworthiness estimation and data cleaning [10]. By utilizing truth discovery algorithms, we can evaluate the quality of web sources and find systematic errors in data curation by analyzing the identified wrong values.

Truth discovery with hierarchies: As pointed out in [6, 8, 25], the extracted values can be hierarchically structured. In this case, there may be multiple correct values in the hierarchy for an object even for functional predicates and we can utilize them to find the most specific correct value among the candidate values. For example, consider the three claimed values of 'NY', 'Liberty Island' and 'LA' about the location of the Statue of Liberty in Table 1. Because Liberty Island is an island in NY, 'NY' and 'Liberty Island' do not conflict with each other. Thus, we can conclude that the Statue of Liberty stands on Liberty Island in NY.

Table 1: Locations of tourist attractions

Object	Source	Claimed value
Statue of Liberty	UNESCO	NY
Statue of Liberty	Wikipedia	Liberty Island
Statue of Liberty	Arrangy	LA
Big Ben	Quora	Manchester
Big Ben	tripadvisor	London

We also observed that many sources provide generalized values in the real-life. Figure 1 shows the graph of the generalized accuracy against the accuracy of the sources in the real-life datasets *BirthPlaces* and *Heritages* used for experiments in Section 5. The accuracy and the generalized accuracy of a source are the proportions of the exactly correct values and hierarchically-correct values among all claimed values, respectively. If a source claims exactly correct values without generalization, it is located at the dotted diagonal line in the graph. This graph shows that many sources in real-life datasets claim with generalized values and each source has its own tendency of generalization when claiming values.

Most of the existing methods [7, 9, 30, 38, 39] simply regard the generalized values of a correct value as incorrect. Thus, it causes a problem in estimating the reliabilities of sources. According to [8], 35% of the false negatives in the data fusion task are produced by ignoring such hierarchical structures. Note that there are many publicly available hierarchies such as WordNet [32] and DBpedia [1]. Thus, a truth discovery algorithm to incorporate hierarchies is proposed in [2]. However, it does not consider the different tendencies of generalization and may lead to the degradation of the accuracy. Another drawback is that it needs a threshold to control the granularity of the estimated truth.

We propose a novel probabilistic model to capture the different generalization tendencies shown in Figure 1. Existing probabilistic models [7, 9, 30, 39] basically assume two interpretations of a claimed value (i.e., correct and incorrect). By introducing three interpretations of a claimed value (i.e., exactly correct, hierarchically correct, and incorrect), our proposed model represents the generalization tendency and reliability of the sources.

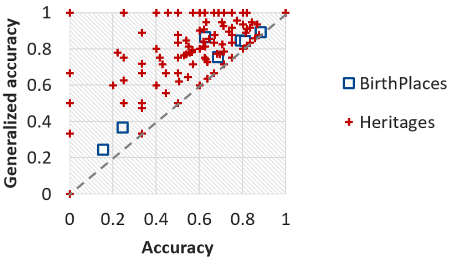


Figure 1: Generalization tendencies of the sources

*Corresponding author

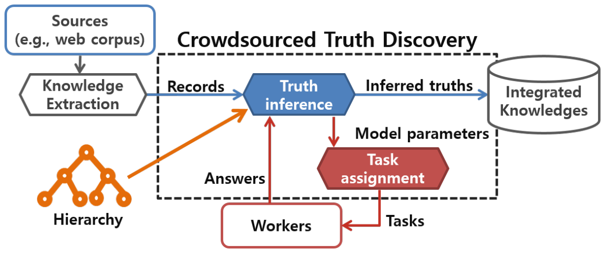


Figure 2: Crowdsourced truth discovery in KF

Crowdsourced truth discovery: It is reported in [8] that upto 96% of the false claims are made by extraction errors rather than by the sources themselves. Since crowdsourcing is an efficient way to utilize human intelligence with low cost, it has been successfully applied in various areas of data integration such as schema matching [12], entity resolution [34], graph alignment [17] and truth discovery [39, 41]. Thus, we utilize crowdsourcing to improve the accuracy of the truth discovery.

It is essential in practice to minimize the cost of crowdsourcing by assigning proper tasks to workers. A popular approach for selecting queries in active learning is *uncertainty sampling* [3, 18, 19, 39]. It asks a query to reduce the uncertainty of the confidences on the candidate values the most. However, it considers only the uncertainty regardless of the accuracy improvement. QASCA algorithm [41] asks a query with the highest accuracy improvement, but measures the improvement without considering the number of collected claimed values. It can be inaccurate since an additional answer may be less informative for an object which already has many records and answers.

Assume that there are two candidate values of an object with equal confidences. If only a few sources provide the claimed values for the object, an additional answer from a crowd worker will significantly change the confidence distribution. Meanwhile, if hundreds of sources already provide the claimed values for the object, the influence of an additional answer is likely to be very little. Thus, we need to consider the number of collected answers as well as the current confidence distribution. Based on the observation, we develop a new method to estimate the increase of accuracy more precisely by considering the number of collected records and answers. We also present an incremental EM algorithm to quickly measure the accuracy improvement and propose a pruning technique to efficiently assign the tasks to workers.

An overview of our truth discovery algorithm: By combining the proposed task assignment and truth inference algorithms, we develop a novel *crowdsourced truth discovery algorithm using hierarchies*. As illustrated in Figure 2, our algorithm consists of two components: *hierarchical truth inference* and *task assignment*. The hierarchical truth inference algorithm finds the correct values from the conflicting values, which are collected from different sources and crowd workers, using hierarchies. The task assignment algorithm distributes objects to the workers who are likely to increase the accuracy of the truth discovery the most. The proposed *crowdsourced truth discovery algorithm* repeatedly alternates the truth inference and task assignment until the budget of crowdsourcing runs out. As discussed in [20], some workers answer slower than others and increase the latency. However, we do not investigate how to reduce the latency in this work since we can utilize the techniques proposed in [13].

Our contributions: The contributions of this paper are summarized below.

- We propose a truth inference algorithm utilizing the hierarchical structures in claimed values. To the best of our knowledge, it is the first work which considers both the reliabilities and the generalization tendencies of the sources.
- To assign a task which will most improve the accuracy, we develop an incremental EM algorithm to estimate the accuracy improvement for a task by considering the number of claimed values as well as the confidence distribution. We also devise an efficient task assignment algorithm for multiple crowd workers based on the quality measure.
- We empirically show that the proposed algorithm outperforms the existing works with extensive experiments on real-life datasets.

2 PRELIMINARIES

In this section, we provide the definitions and the problem formulation of *crowdsourced truth discovery in the presence of hierarchy.*

2.1 Definitions

For the ease of presentation, we assume that we are interested in a single attribute of objects although our algorithms can be easily generalized to find the truths of multiple attributes. Thus, we use ‘the target attribute value of an object’ and ‘the value of an object’ interchangeably.

A *source* is a structured or unstructured database which contains the information on target attribute values for a set of objects. In this paper, a *source* is a certain web page or website and a *worker* represents a human worker in crowdsourcing platforms. The information of an object provided by a source or a worker is called a *claimed value.*

Definition 2.1. A *record* is a data describing the information about an object from a source. A record on an object o from a source s is represented as a triple (o, s, v_o^s) where v_o^s is the claimed value of an object o collected from s. Similarly, if a worker w answers that the truth on an object o is v_o^w, the *answer* is represented as (o, w, v_o^w).

Let S_o be the set of the sources which claimed a value on the object o and V_o be the set of candidate values collected from S_o. Each worker in W_o answers a question about the object o by selecting a value from V_o.

In our problem setting, we assume that we have a hierarchy tree H of the claimed values. If we are interested in an attribute related to locations (e.g., birthplace), H would be a geographical hierarchy with different levels of granularity (e.g., continent, country, city, etc.). We also assume that there is no answer with the value of the root in the hierarchy since it provides no information at all (e.g., Earth as a birthplace). We summarize the notations to be used in the paper in Table 2.

Example 2.2. Consider the records in Table 1. Since the source Wikipedia claims that the location of the Statue of Liberty is Liberty Island, it is represented by v_o^s =‘Liberty Island’ where o =‘Statue of Liberty’ and s =‘Wikipedia’. If a human worker ‘Emma Stone’ answered Big Ben is in London, it is represented by v_o^w =‘London’ where o =‘Big Ben’ and w =‘Emma Stone’.

2.2 Problem Definition

Given a set of objects O and a hierarchy tree H, we define the two subproblems of the crowdsourced truth discovery.

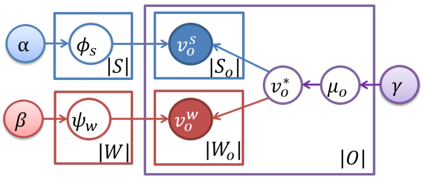


Figure 3: A graphical model for truth inference

Definition 2.3 (Hierarchical truth inference problem). For a set of records R collected from the sources and a set of answers A from the workers, we find the most specific true value v_o^* of each object $o \in O$ among the candidate values in V_o by using the hierarchy H.

Definition 2.4 (Task assignment problem). For each worker w in a set of workers W, we select the top-k objects from O which are likely to increase the overall accuracy of the inferred truths the most by using the hierarchy H.

We present a hierarchical truth inference algorithm in Section 3 and a task assignment algorithm in Section 4.

3 HIERARCHICAL TRUTH INFERENCE

For the hierarchical truth inference, we first model the trustworthiness of sources and workers for a given hierarchy. Then, we propose a probabilistic model to describe the process of generating the set of records and the set of answers based on the trustworthiness modeling. We next develop an inference algorithm to estimate the model parameters and determine the truths.

3.1 Our Generative Model

Our probabilistic graphical model in Figure 3 expresses the conditional dependence (represented by edges) between random variables (represented by nodes). While the previous works [5, 15, 31, 35] assume that all sources and workers have their own reliabilities only, we assume that each source or worker has its generalization tendency as well as reliability. We first describe how sources and workers generate the claimed values based on their trustworthiness. We next present the model for generating the true value. Finally, we provide the detailed generative process of our probabilistic model.

Model for source trustworthiness: For an object o, let v_o^* be the truth and v_o^s be the claimed value reported by a source s. Recall that V_o is the set of candidate values for an object o. Furthermore, we let $G_o(v)$ denote the set of candidate values which are ancestors of a value v except for the root in the hierarchy H.

Table 2: Notations

Symbol	Description
s	A data source
w	A crowd worker
v_o^s	Claimed value from s about o
v_o^w	Claimed value from w about o
R	Set of all records collected from the set of sources S
A	Set of all answers collected from the set of workers W
V_o	Set of candidate values about o
S_o	Set of sources which post information about o
W_o	Set of workers who answered about o
O_s	Set of objects that source s provided a value
O_w	Set of objects that worker w answered to
$G_o(v)$	Set of values in V_o which are ancestors of a value v except the root in the hierarchy H
$D_o(v)$	Set of values in V_o which are descendants of v

There are three relationships between a claimed value v_o^s and the truth v_o^*: (1) $v_o^s = v_o^*$, (2) $v_o^s \in G_o(v_o^*)$ and (3) otherwise. Let $\phi_s = (\phi_{s,1}, \phi_{s,2}, \phi_{s,3})$ be the *trustworthiness distribution* of a source s where $\phi_{s,i}$ is the probability that a claimed value of the source s corresponds to the i-th relationship. In each relationship, a claimed value is generated as follows:

- **Case 1 ($v_o^s = v_o^*$):** The source s provides the exact true value with a probability $\phi_{s,1}$.
- **Case 2 ($v_o^s \in G_o(v_o^*)$):** The source s provides a *generalized true value* v_o^s with a probability $\phi_{s,2}$. In this case, the claimed value is an ancestor of the truth v_o^* in H. We assume that the claimed value is uniformly selected from $G_o(v_o^*)$.
- **Case 3 (*otherwise*):** The source s provides a wrong value v_o^s not even in $G_o(v_o^*)$. The claimed value is uniformly selected among the rest of the candidate values in V_o.

The probability distribution ϕ_s is an initially-unknown model parameter to be estimated in our inference algorithm. Accordingly, the probability of selecting an answer v_o^s among the values in V_o for an object o is represented by

$$P(v_o^s|v_o^*, \phi_s) = \begin{cases} \phi_{s,1} & \text{if } v_o^s = v_o^*, \\ \phi_{s,2}/|G_o(v_o^*)| & \text{if } v_o^s \in G_o(v_o^*), \\ \phi_{s,3}/(|V_o| - |G_o(v_o^*)| - 1) & \text{otherwise.} \end{cases} \quad (1)$$

For the prior of the distribution ϕ_s, we assume that it follows a Dirichlet distribution $Dir(\alpha)$, with a hyperparameter $\alpha = (\alpha_1, \alpha_2, \alpha_3)$, which is the conjugate prior of categorical distributions.

Let O_H be the set of objects who have an ancestor-descendant relationship in their candidate set. In practice, there may exist some objects whose candidate values do not have an ancestor-descendant relationship. In this case, the probability of the second case (i.e., $\phi_{s,2}$) may be underestimated. Thus, if there is no ancestor-descendant relationship between the claimed values about o (i.e., $o \notin O_H$), we assume that a source generates its claimed value v_o^s with the following probability

$$P(v_o^s|v_o^*, \phi_s) = \begin{cases} \phi_{s,1} + \phi_{s,2} & \text{if } v_o^s = v_o^*, \\ \phi_{s,3}/(|V_o| - 1) & \text{otherwise.} \end{cases} \quad (2)$$

Model for worker trustworthiness: Let v_o^w be the claimed value chosen by a worker w among the candidates in V_o for an object o. Similar to the model for source trustworthiness, we also assume the three relationships between a claimed value v_o^w and the truth v_o^*: (1) $v_o^w = v_o^*$, (2) $v_o^w \in G_o(v_o^*)$ and (3) otherwise. Each worker w has its *trustworthiness distribution* $\psi_w = (\psi_{w,1}, \psi_{w,2}, \psi_{w,3})$ where $\psi_{w,i}$ is the probability that an answer of the worker w corresponds to the i-th relationship. We assume that the trustworthiness distribution is generated from $Dir(\beta)$ with a hyperparameter $\beta = (\beta_1, \beta_2, \beta_3)$.

Since it is difficult for the workers to be aware of the correct answer for every object, a worker can refer to web sites to answer the question. In such a case, if there is a widespread misinformation across multiple sources, the worker is also likely to respond with the incorrect information. Similar to [9, 30], we thus exploit the *popularity* of a value in Cases 2 and 3 to consider such dependency between sources and workers.

- **Case 1 ($v_o^w = v_o^*$):** The worker w provides the exact true value with a probability $\psi_{w,1}$.
- **Case 2 ($v_o^w \in G_o(v_o^*)$):** The worker w provides a generalized true value with a probability $\psi_{w,2}$. We assume that

$$f^v_{o,s}=\frac{P(v^s_o|v^*_o=v,\phi_s)\cdot\mu_{o,v}}{\sum_{v'\in V_o}P(v^s_o|v^*_o=v',\phi_s)\cdot\mu_{o,v'}}$$

$$f^v_{o,w}=\frac{P(v^w_o|v^*_o=v,\psi_w)\cdot\mu_{o,v}}{\sum_{v'\in V_o}P(v^w_o|v^*_o=v',\psi_w)\cdot\mu_{o,v'}}$$

$$g^1_{o,s}=\frac{\phi_{s,1}\cdot\mu_{o,v^s_o}}{\sum_{v\in V_o}P(v^s_o|v^*_o=v,\phi_s)\cdot\mu_{o,v}}$$

$$g^2_{o,s}=\begin{cases}\dfrac{\sum_{v\in D_o(v^s_o)}\frac{\phi_{s,2}}{|G_o(v)|}\cdot\mu_{o,v}}{\sum_{v\in V_o}P(v^s_o|v^*_o=v,\phi_s)\cdot\mu_{o,v}} & \text{if } o\in O_H\\ \dfrac{\phi_{s,2}\cdot\mu_{o,v^s_o}}{\sum_{v\in V_o}P(v^s_o|v^*_o=v,\phi_s)\cdot\mu_{o,v}} & \text{otherwise}\end{cases}$$

$$g^3_{o,s}=\frac{\sum_{v\in\neg D_o(v^s_o)}\frac{\phi_{s,3}}{|V_o-G_o(v)|-1}\cdot\mu_{o,v}}{\sum_{v\in V_o}P(v^s_o|v^*_o=v,\phi_s)\cdot\mu_{o,v}}$$

$$g^1_{o,w}=\frac{\psi_{w,1}\cdot\mu_{o,v^w_o}}{\sum_{v\in V_o}P(v^w_o|v^*_o=v,\psi_w)\cdot\mu_{o,v}}$$

$$g^2_{o,w}=\begin{cases}\dfrac{\sum_{v\in D_o(v^w_o)}\psi_{w,2}\cdot Pop_2(v^w_o|v^*_o=v)\cdot\mu_{o,v}}{\sum_{v\in V_o}P(v^w_o|v^*_o=v,\psi_w)\cdot\mu_{o,v}} & \text{if } o\in O_H\\ \dfrac{\psi_{w,2}\cdot\mu_{o,v^w_o}}{\sum_{v\in V_o}P(v^w_o|v^*_o=v,\psi_w)\cdot\mu_{o,v}} & \text{otherwise}\end{cases}$$

$$g^3_{o,w}=\frac{\sum_{v\in\neg D_o(v^w_o)}\psi_{w,3}\cdot Pop_3(v^w_o|v^*_o=v)\cdot\mu_{o,v}}{\sum_{v\in V_o}P(v^w_o|v^*_o=v,\psi_w)\cdot\mu_{o,v}}$$

Figure 4: E-step for the proposed truth inference algorithm

the claimed value v^w_o is selected according to the popularity $Pop_2(v^w_o|v^*_o)=\frac{|\{s|s\in S_o,v^s_o=v\}|}{|\{s|s\in S_o,v^s_o\in G_o(v^*_o)\}|}$ which is the proportion of the records whose claimed value is v^w_o out of the records with generalized values of v^*_o.

- **Case 3** (*otherwise*)**:** The claimed value is selected from the wrong values according to the popularity $Pop_3(v^w_o|v^*_o)=\frac{|\{s|s\in S_o,v^s_o=v\}|}{|\{s|s\in S_o,v^s_o\notin G_o(v^*_o),v^s_o\neq v^*_o\}|}$.

By the above model, the probability of selecting an answer v^w_o for the truth v^*_o of an object o is formulated as

$$P(v^w_o|v^*_o,\psi_w)=\begin{cases}\psi_{w,1} & \text{if } v^w_o=v^*_o,\\ \psi_{w,2}\cdot Pop_2(v^w_o|v^*_o) & \text{if } v^w_o\in G_o(v^*_o),\\ \psi_{w,3}\cdot Pop_3(v^w_o|v^*_o) & \text{otherwise.}\end{cases} \quad (3)$$

Similar to the model for source trustworthiness, if there is no ancestor-descendant relationship in the candidate values of an object o, the probability of selecting a claimed value v^w_o is

$$P(v^w_o|v^*_o,\psi_w)=\begin{cases}\psi_{w,1}+\psi_{w,2} & \text{if } v^w_o=v^*_o,\\ \psi_{w,3}\cdot Pop_3(v^w_o|v^*_o) & \text{otherwise.}\end{cases} \quad (4)$$

Model for truth: We introduce the probability distribution over the candidate answers to determine the truth, called *confidence distribution.* Each object o has a confidence distribution $\mu_o=\{\mu_{o,v}\}_{v\in V_o}$ where $\mu_{o,v}$ is the probability that the value $v\in V_o$ is the true answer for o. We also use a dirichlet prior $Dir(\gamma_o)$ for the confidence distribution μ_o where $\gamma_o=\{\gamma_{o,v}\}_{v\in V_o}$ is a hyperparameter.

Based on the above three models, the generative process of our model works as follows.

Generative process: Given a set of objects O, a set of sources S and a set of workers W, our proposed model assumes the following generative process for the set of records R and the set of answers A:

(1) Draw $\phi_s\sim Dir(\alpha)$ for each source $s\in S$
(2) Draw $\psi_w\sim Dir(\beta)$ for each worker $w\in W$
(3) For each object $o\in O$
  (a) Draw $\mu_o\sim Dir(\gamma_o)$
  (b) Draw a true value $v^*_o\sim Categorical(\mu_o)$
  (c) For each source $s\in S_o$
    (i) Draw a value v^s_o following $P(v^s_o|v^*_o,\phi_s)$
  (d) For each worker $w\in W_o$
    (i) Draw a value v^w_o following $P(v^w_o|v^*_o,\psi_w)$

3.2 Estimation of Model Parameters

We now develop an inference algorithm for the generative model. Let $\Theta=\boldsymbol{\phi}\cup\boldsymbol{\psi}\cup\boldsymbol{\mu}$ be the set of all model parameters where $\boldsymbol{\phi}=\{\phi_s|s\in S\}$, $\boldsymbol{\psi}=\{\psi_w|w\in W\}$ and $\boldsymbol{\mu}=\{\mu_o|o\in O\}$. We propose an EM algorithm to find the maximum a posteriori (MAP) estimate of the parameters in our model.

The maximum a posteriori (MAP) estimator: Recall that $R=\{(o,s,v^s_o)\}$ is the set of records from the sources and $A=\{(o,w,v^w_o)\}$ is the set of answers from the workers. For every object o, each source $s\in S_o$ and each worker $w\in W_o$ generates its claimed values independently. Then, the likelihood of R and A based on our generative model is

$$P(R,A|\Theta)=\prod_{o\in O}\prod_{s\in S_o}P(v^s_o|\phi_s,\mu_o)\cdot\prod_{o\in O}\prod_{w\in W_o}P(v^w_o|\psi_w,\mu_o)$$

where the probability of generating a claimed value by a source or a worker becomes

$$P(v^s_o|\phi_s,\mu_o)=\sum_{v\in V_o}P(v^s_o|\phi_s,v^*_o=v)\cdot\mu_{o,v} \quad (5)$$

$$P(v^w_o|\psi_w,\mu_o)=\sum_{v\in V_o}P(v^w_o|\psi_w,v^*_o=v)\cdot\mu_{o,v}. \quad (6)$$

Consequently, the MAP point estimator is obtained by maximizing the log-posterior as

$$\hat{\Theta}=\arg\max_{\Theta}\{\log P(R,A|\Theta)+\log P(\Theta)\}=\arg\max_{\Theta}\mathbb{F} \quad (7)$$

where the objective function $\mathbb{F}$ is

$$\begin{aligned}\mathbb{F}=&\sum_{o\in O}\sum_{s\in S_o}\log\sum_{v\in V_o}P(v^s_o|\phi_s,v^*_o=v)\cdot\mu_{o,v}\\ &+\sum_{o\in O}\sum_{w\in W_o}\log\sum_{v\in V_o}P(v^w_o|\psi_w,v^*_o=v)\cdot\mu_{o,v}\\ &+\sum_{s\in S}\log p(\phi_s|\alpha)+\sum_{w\in W}\log p(\psi_w|\beta)+\sum_{o\in O}\log p(\mu_o|\gamma_o).\end{aligned} \quad (8)$$

Note that although we assumed that each claimed value is generated independently according to its probability distribution defined in Eq. (5) and (6), the dependencies between sources and workers are already considered in $Pop_2(v^w_o|v^*_o)$ and $Pop_3(v^w_o|v^*_o)$.

The EM algorithm: We introduce a random variable C_v to represent the type of the relationship between the claimed value v and the truth v^*_o. It is defined as follows:

$$C_v=\begin{cases}1 & \text{if } v=v^*_o,\\ 2 & \text{if } v\in G_o(v^*_o),\\ 3 & \text{otherwise.}\end{cases}$$

In the **E-step**, we compute the conditional distributions of the hidden variables $C_{v^s_o}$, $C_{v^w_o}$ and v^*_o under our current estimate of the parameters Θ. Let $f^v_{o,s}$, $f^v_{o,w}$, $g^t_{o,s}$ and $g^t_{o,w}$ denote the conditional probabilities $P(v^*_o=v|v^s_o,\mu_o,\phi_s)$, $P(v^*_o=v|v^w_o,\mu_o,\psi_w)$, $P(C_{v^s_o}=t|\mu_o,\phi_s)$ and $P(C_{v^w_o}=t|\ \mu_o,\psi_w)$, respectively. Using Bayes' rule, we can update the conditional probabilities as shown in Figure 4 where $D_o(v)=\{v'|v\in G_o(v')\wedge v'\in V_o\}$ is the set of descendants of v among the candidate values and $\neg D_o(v)=$

$V_o - D_o(v) - \{v\}$ is the set of candidate values each of which is neither a descendant of the value v nor the v itself.

In the **M-step**, we find the model parameters Θ that maximize our objective function $\mathbb{F}$. We first add Lagrange multipliers to enforce the constraints of model parameters.

$$\mathbb{L} = \mathbb{F} + \sum_{s \in S} \lambda_{\phi,s} \left(1 - \sum_{t=1}^{3} \phi_{s,t}\right) + \sum_{w \in W} \lambda_{\psi,w} \left(1 - \sum_{t=1}^{3} \psi_{w,t}\right) + \sum_{o \in O} \lambda_{\mu,o} \left(1 - \sum_{v \in V_o} \mu_{o,v}\right)$$

We obtain the following equations for updating the model parameters Θ by taking the partial derivative of the Lagrangian $\mathbb{L}$ with respect to each model parameter and setting it to zero:

$$\mu_{o,v} = \frac{\sum_{s \in S_o} f_{o,s}^{v} + \sum_{w \in W_o} f_{o,w}^{v} + \gamma_{o,v} - 1}{|S_o| + |W_o| + \sum_{v' \in V_o} (\gamma_{o,v'} - 1)} \tag{9}$$

$$\phi_{s,t} = \frac{\sum_{o \in O_s} g_{o,s}^{t} + \alpha_t - 1}{|O_s| + \sum_{t'=1}^{3} (\alpha_{t'} - 1)} \tag{10}$$

$$\psi_{w,t} = \frac{\sum_{o \in O_w} g_{o,w}^{t} + \beta_t - 1}{|O_w| + \sum_{t'=1}^{3} (\beta_{t'} - 1)} \tag{11}$$

where O_s and O_w are the sets of objects claimed by s and w, respectively. We infer the truth by choosing the value with the maximum confidence among the candidate values as

$$v_o^* = \arg\max_{v \in V_o} \mu_{o,v}. \tag{12}$$

Extension to numerical data: In the world wide web, numerical data also have an implicit hierarchy due to the significant digits which carry meaning contributing to its measurement resolution. For example, even though the area of Seoul is $605.196km^2$, different websites may represent the area in various forms depending on the significant figures (e.g., $605.2km^2$, $605km^2$). An existing algorithm [21] to handle numerical data utilizes a weighted sum of the claimed values to consider the distribution of the claimed values. However, such method is sensitive to outliers and thus need a proper preprocessing to remove the outliers. To overcome the drawbacks, we generate the underlying hierarchy in the numerical data by assuming that v_d is a descendant of v_a if a value v_a can be obtained by rounding off a value v_d. Then, we can use our TDH algorithm to find the truths in numerical data by taking into account the relationship between the values in the implicit hierarchy. Our algorithm is also robust to the outliers with extremely small or large value since we estimate the truth by selecting the most probable value from the candidate values rather than computing a weighted average of the claimed values.

4 TASK ASSIGNMENT TO WORKERS

In this section, we propose a task assignment method to select the best objects to be assigned to the workers in crowdsourcing systems. We first define a quality measure of tasks called *Expected Accuracy Increase (EAI)* and develop an incremental EM algorithm to quickly estimate the quality measure. Finally, we present an efficient algorithm for assigning the k questions to each worker w in a set of workers W based on the measure.

4.1 The Quality Measure

Given a worker w, our goal is to choose an object to be assigned to the worker w which is likely to increase the accuracy of the estimated truths the most. Thus, we define a quality measure for a pair of worker and an object based on the improvement of the accuracy. As discussed in [41], the improvement of the accuracy by a task can be estimated by using the difference between the highest confidence as follows:

$$(Accuracy\ improvement) = \{\max_v \mu_{o,v|w} - \max_v \mu_{o,v}\}/|O| \tag{13}$$

where $\mu_{o,v|w}$ is the estimated confidence on v if the worker w answers about an object o.

The quality measure used by QASCA: The QASCA[41] algorithm calculates the estimated confidence by using the current confidence distribution and the likelihood of the answer v_o^w given the truth $v_o^* = v$ as

$$\mu_{o,v|w} \propto \mu_{o,v} \cdot p(v_o^w = v' | v_o^* = v)$$

where v' is a sampled claimed value. There are two drawbacks in the quality measure of QASCA. First, since it computes the estimated confidence $\mu_{o,v|w}$ based on a sampled answer $v_o^w = v$, the value of the quality measure is very sensitive to the sampled answer. In addition, QASCA does not consider the number of claimed values collected so far and the estimated confidence $\mu_{o,v|w}$ may not be accurate. For instance, assume that there exist two objects which have identical confidence distributions. If one of the objects already has many collected claimed values, an additional answer is not likely to change the confidence significantly. Thus, task assignment algorithms should select another object who has a smaller number of collected records and answers.

Our quality measure: To avoid the sensitiveness caused by sampling answers, we develop a new quality measure *Expected Accuracy Improvement (EAI)* which is obtained by taking the expectation to Eq. (13). That is,

$$EAI(w,o) = \{E[\max_v \mu_{o,v|w}] - \max_v \mu_{o,v}\}/|O|. \tag{14}$$

By the definition of expectation, $E[\max_v \mu_{o,v|w}]$ becomes

$$E[\max_v \mu_{o,v|w}] = \sum_{v' \in V_o} P(v_o^w = v' | \psi_w, \mu_o) \cdot \max_v \mu_{o,v|v_o^w = v'}. \tag{15}$$

where $\mu_{o,v|v_o^w=v'}$ is the conditional confidence when a worker w answers with v' about the object o.

Since $P(v_o^w = v' | \psi_w, \mu_o)$ can be computed by Eq. (6), to compute $E[\max_v \mu_{o,v|w}]$ by Eq. (15), we need the estimation of the conditional confidence $\mu_{o,v|v_o^w=v'}$ with an additional answer $v_o^w = v'$. Recall that the estimated confidence computed by QASCA may not be accurate because it does not consider the collected records and answers so far. To reduce the error, we use them to compute the conditional confidence $\mu_{o,v|v_o^w=v'}$. We can compute the conditional confidence $\mu_{o,v|v_o^w=v'}$ by applying the EM algorithm in Section 3.2 with the collected records and answers including $v_o^w = v'$. However, since it is computationally expensive, we next develop an *incremental EM algorithm.*

4.2 The Incremental EM Algorithm

Let $\mathbb{F}_{v_o^w=v'}$ be the objective function in Eq. (7) after obtaining an additional answer (o, w, v'). Then, we have

$$\mathbb{F}_{v_o^w=v'} = \mathbb{F} + \log \sum_{v \in V_o} P(v_o^w = v' | \psi_w, v_o^* = v) \cdot \mu_{o,v}$$

by adding the related term of the additional answer (log likelihood of the additional answer) to Eq. (8). Instead of running the iterative EM algorithm in Section 3.2, we incrementally perform a *single EM-step* to speed up for only the additional answer with the current model parameters and the above objective function.

E-step: Since we use the current model parameters, the probabilities of the hidden variables for collected records and answers are not changed. Thus, we only need to compute the conditional probabilities of the hidden variable given the additional answer as

$$f_{o,w|v_o^w=v'}^{v} = \frac{P(v_o^w = v' | v_o^* = v, \psi_w) \cdot \mu_{o,v}}{\sum_{v'' \in V_o} P(v_o^w = v' | v_o^* = v'', \psi_w) \cdot \mu_{o,v''}} \tag{16}$$

based on the equation for $f^{v}_{o,w}$ used at the E-step in Figure 4.

M-step: For the objective function $\mathbb{F}_{v_o^w=v'}$, we obtain the following equation of the M-step for the confidence distribution μ_o with the additional answer $v_o^w = v'$

$$\mu_{o,v|v_o^w=v'} = \frac{\sum_{s\in S_o} f^{v}_{o,s} + \sum_{w'\in W_o} f^{v}_{o,w'} + f^{v}_{o,w|v_o^w=v'} + \gamma_{o,v} - 1}{|S_o| + |W_o| + 1 + \sum_{v''\in V_o}\left(\gamma_{o,v''} - 1\right)}$$

by adding the related terms $f^{v}_{o,w|v_o^w=v'}$ and 1 to the numerator and the denominator of the update equation in Eq. (9), respectively. Let $N_{o,v}$ and D_o be the numerator and the denominator in Eq. (9), respectively. Then, the above equation can be rewritten as

$$\mu_{o,v|v_o^w=v'} = \frac{N_{o,v} + f^{v}_{o,w|v_o^w=v'}}{D_o + 1}. \tag{17}$$

By substituting $f^{v}_{o,w|v_o^w=v'}$ in Eq. (17) with Eq. (16), the conditional confidence becomes

$$\mu_{o,v|v_o^w=v'} = \frac{N_{o,v} + \frac{P(v_o^w=v'|v_o^*=v,\psi_w)\cdot\mu_{o,v}}{\sum_{v''\in V_o} P(v_o^w=v'|v_o^*=v'',\psi_w)\cdot\mu_{o,v''}}}{D_o + 1}. \tag{18}$$

Since $N_{o,v}$ and D_o are proportional to the number of the existing claimed values, the confidence will be changed very little if there are many claimed values already. Thus, we can overcome the second drawback of QASCA. Since $N_{o,v}$s and D_os are repeatedly used to compute $\mu_{o,v|v_o^w=v'}$, our truth inference algorithm keeps $N_{o,v}$s and D_os in main memory to reduce the computation time.

Time complexity analysis: To calculate $E[\max_v \mu_{o,v|w}]$ by Eq. (15), $P(v_o^w = v'|\psi_w, \mu_o)$ is computed $|V_o|$ times and $\mu_{o,v|v_o^w=v'}$ is calculated for every pair of v and v' (i.e., $O(|V_o|^2)$ times). Moreover, computing $P(v_o^w = v'|\psi_w, \mu_o)$ and $\mu_{o,v|v_o^w=v'}$ take $O(|V_o|)$ time. Thus, it takes $O(|V_o|^3)$ time to compute $EAI(w, o)$ by Eq. (14). In reality, $|V_o|$ is very small compared to $|O|$,$|S|$ and $|W|$. In addition, by utilizing the pruning technique in the next section, we can significantly reduce the computation time. Therefore, the task assignment step can be performed within a short time compared to the truth inference. The execution time for each step will be presented in the experiment section.

4.3 The Task Assignment Algorithm

To find the k objects to be assigned to each worker, we need to compute $EAI(w, o)$ for all pairs of w and o. To reduce the number of computing $EAI(w, o)$, we develop a pruning technique by utilizing an upper bound of $EAI(w, o)$.

An upper bound of EAI: We provide the following lemma which allows us to compute an upper bound $U_{EAI}(o)$.

Lemma 4.1. (Upper Bound of Expected Accuracy Increase) *For an object o and a worker w, we have*

$$EAI(w, o) \le U_{EAI}(o) = \frac{1 - \max_v \mu_{o,v}}{|O| \cdot (D_o + 1)}. \tag{19}$$

Proof. From Eq. (18), since $\sum_{v'} P(v_o^w=v'|\psi_w, \mu_o) = 1$, we get

$$\begin{aligned} E[\max_v \mu_{o,v|w}] &= \sum_{v'\in V_o} P(v_o^w=v'|\psi_w, \mu_o) \cdot \max_v \mu_{o,v|v_o^w=v'} \\ &\le \max_{v,v'} \mu_{o,v|v_o^w=v'} \cdot \sum_{v'\in V_o} P(v_o^w=v'|\psi_w, \mu_o) \\ &= \max_{v,v'} \mu_{o,v|v_o^w=v'}. \end{aligned} \tag{20}$$

Moreover, from Eq. (17), we obtain

$$\mu_{o,v|v_o^w=v'} = \frac{N_{o,v} + f^{v}_{o,w|v_o^w=v'}}{D_o + 1} \le \frac{N_{o,v} + 1}{D_o + 1}. \tag{21}$$

Algorithm 1 Task Assignment

```
Input: set of workers $W$, number of questions $k$
 1: Compute the upper bound $U_{EMCI}(o)$ for $o \in O$
 2: $h_{UB} \leftarrow$ BuildMaxHeap($\{\langle U_{EAI}(o), o\rangle | o \in O\}$)
 3: Sort workers in the decreasing order of $\psi_{w,1}$
    (i.e., $\psi_{1,1} \ge \psi_{2,1} \ge \cdots \ge \psi_{|W|,1}$).
 4: for $w$ = 1 to $|W|$ do
 5:     $h_{EAI}[w] \leftarrow$ BuildMinHeap({})
 6: while True do
 7:     $\langle U_{EAI}(o), o\rangle \leftarrow h_{UB}$.extractMax()
 8:     if $h_{EAI}[|W|].size = k$ and $h_{EAI}[w].min > U_{EAI}(o)$ for all $w$ then
 9:         break
10:     for $w$ = 1 to $|W|$ do
11:         if $w$ already answered on $o$ or $h_{EAI}[w].min > U_{EAI}(o)$ then
12:             continue
13:         Compute $EAI(w, o)$
14:         $h_{EAI}[w]$.insert($\langle EAI(w, o), o\rangle$)
15:         if $h_{EAI}[w].size \le k$ then
16:             break
17:         $o \leftarrow h_{EAI}[w]$.extractMin().value()
```

By substituting Eq. (21) for $\mu_{o,v|v_o^w=v'}$ in Eq. (20), we derive

$$E[\max_v \mu_{o,v|w}] \le \max_{v,v'} \mu_{o,v|v_o^w=v'} \le \frac{\max_v N_{o,v} + 1}{D_o + 1}. \tag{22}$$

In addition, by applying Eq. (22) to Eq. (14), we get

$$EAI(w, o) \le (\tfrac{\max_v N_{o,v}+1}{D_o+1} - \max_v \mu_{o,v})/|O|.$$

Since $\mu_{o,v} = \frac{N_{o,v}}{D_o}$, we finally obtain the upper bound of $EAI(w, o)$.

$$\begin{aligned} EAI(w, o) &\le (\tfrac{\max_v N_{o,v}+1}{D_o+1} - \tfrac{\max_v N_{o,v}}{D_o})/|O| \\ &= \frac{1 - \frac{\max_v N_{o,v}}{D_o}}{|O|\cdot(D_o+1)} = \frac{1 - \max_v \mu_{o,v}}{|O|\cdot(D_o+1)} = U_{EAI}(o). \end{aligned}$$

□

We devise an algorithm to assign the best k objects to each available worker in crowdsourcing systems. Since a single answer is sufficient to find the correct value for some objects, we assign an object to only a single worker in each round. If the answer is not sufficient to find the correct value of the object, we assign the object to another worker in the next round.

Our task assignment algorithm sequentially assigns each object to a worker by scanning the objects o with non-increasing order of the upper bound $U_{EAI}(o)$. To allocate an object to a worker, since $\psi_{w,1}$ is the probability of answering the truth, we consider the workers w with non-increasing order of $\psi_{w,1}$. After assigning an object to a worker w, if the number of assigned objects to the worker w exceeds k, we remove the object o with the minimum $EAI(w, o)$ and assign the deleted object to the next worker and perform the same step. While scanning the objects, we stop the assignment if the upperbound $U_{EAI}(o)$ is smaller than the minimum $EAI(w, o')$ among the $EAI(w, o')$s of all assigned objects and each worker has k assigned objects. The reason is that the $EAI(w, o)$ of the remaining objects o can be larger than that of any assigned object.

The pseudocode: It is shown in Algorithm 1. We first compute the upper bound $U_{EAI}(o)$ for every object $o \in O$ by Lemma 4.1 and build a maxheap h_{UB} of all objects by using $U_{EAI}(o)$ as the key to assign the objects to workers in the decreasing order of $U_{EAI}(o)$ (in lines 1-2). The workers are sorted in the decreasing order of $\psi_{w,1}$ to give a higher priority to reliable workers (in line 3). We next initialize a minheap $h_{EAI}[w]$ for each worker w to contain the k assigned objects (in lines 4-6). Then, we repeatedly extract an object from h_{UB} and assign the object to a worker in the sorted order of $\psi_{w,1}$ (in lines 12-18). Before assigning an object o, if the heaps $h_{EAI}[w]$s of all workers are full and the minimum value of $EAI(w, o')$ of the objects o' in all $h_{EAI}[w]$s is larger than the upper bound $U_{EAI}(o)$, we stop immediately.

5 EXPERIMENTS

The experiments are conducted on a computer with Intel i5-7500 CPU and 16GB of main memory.

Datasets: We collected the two real-life datasets publicly available at http://kdd.snu.ac.kr/home/datasets/tdh.php.

BirthPlaces: We crawled 13,510 records about the birthplaces of 6,005 celebrities from 7 websites (sources). For the gold standard data to evaluate the correctness of discovered birthplaces, we used IMDb biography which is available at *http://www.imdb.com.* Moreover, the geographical hierarchy was created by using the IMDb data. For example, if there is a person who was born in 'LA, California, USA', we assigned 'LA' as a child of 'California' and 'California' as a child of 'USA'. The hierarchy contains 4,999 nodes (e.g., countries, cities and etc.) and its height is 5.

Heritages: This is a dataset of the locations of World Heritage Sites provided by UNESCO World Heritage Centre, available at *http://whc.unesco.org*. We queried about the locations of 785 World Heritage Sites with Bing Search API and obtained 4,424 claimed values from 1,577 distinct websites. The hierarchy was created in the same way as we did for *BirthPlaces* and it has 1,027 nodes. The height of this hierarchy tree is 6.

Quality Measures: We use *Accuracy*, *GenAccuracy* and *AvgDistance* to evaluate the truth discovery algorithms. Let t_o be the truth of the object o in the gold standard and v_o^* be the estimated truth by an algorithm. Note that t_o may not exist in the set of candidate values. In this case, the most specific candidate value among the ancestors of the truth is assumed to be t_o. *Accuracy* is the proportion of objects that the algorithm discovers the truth exactly. It is actually used in [10, 39–41] to evaluate truth discovery algorithms.

$$(Accuracy) = \Sigma_{o \in O} I(v_o^* = t_o)/|O|$$

The ancestors of t_o are less informative but still correct values. Thus, we develop an evaluation measure named *GenAccuracy* which is the proportion of objects o whose estimated truth v_o^* is either the truth t_o or an ancestor of the truth.

$$(GenAccuracy) = \Sigma_{o \in O} I(v_o^* \in G_H(t_o) \cup \{t_o\})/|O|$$

Ancestors of the truth have a different level of informativeness depending on the distance to the truth. For example, 'New York' is more informative than 'USA' as the location of the Statue of Liberty. Thus, we utilize another evaluation measure named *AvgDistance* which weights the estimated truth based on the distance from the ground truth. More specifically, it is the average number of edges $d(v_o^*, t_o)$ between the truth t_o and the estimated truth v_o^* in the hierarchy H.

$$(AvgDistance) = \Sigma_{o \in O} d(v_o^*, t_o)/|O|$$

AvgDistance is robust to the case where the ground truth is less specific than the estimated truth. The estimated truth is regarded as a wrong value when we compute *Accuracy* and *GenAccuracy* even though the estimate truth is correct and more specific. Since the distance between the less specific ground truth and the estimated truth is generally small, *AvgDistance* compensates the drawback of *Accuracy* and *GenAccuracy*.

Settings for simulated crowdsourcing: To evaluate the truth discovery algorithms with varying the quality of the answers from workers, we conducted experiments with simulated crowd workers. In our simulation, we assumed that each simulated worker answers a question correctly with its own probability p_w and randomly selects an answer from the candidate values with probability $1-p_w$. We sampled the probability p_w from a uniform distribution ranging from $\pi_p-0.05$ to $\pi_p+0.05$ where the default value of π_p is 0.75. In the experiments, each of 10 worker answers 5 questions for each round.

Table 3: Performance of truth inference algorithms

	Dataset					
	BirthPlaces			*Heritages*		
Algorithm	*Accuracy*	*GenAccuracy*	*AvgDistance*	*Accuracy*	*GenAccuracy*	*AvgDistance*
TDH	**0.8913**	**0.8988**	**0.3151**	**0.7414**	0.8726	**0.5210**
VOTE	0.7900	0.8924	0.4961	0.6892	**0.8994**	0.6382
LCA	0.8834	0.8923	0.3414	0.6930	0.8866	0.6611
DOCS	0.8828	0.8916	0.3409	0.6904	0.8866	0.6599
ASUMS	0.8543	0.8571	0.4573	0.6229	0.7414	1.2000
MDC	0.8263	0.8432	0.5320	0.7254	0.8087	0.6869
ACCU	0.8137	0.8296	0.6063	0.5834	0.7656	1.0637
POPACCU	0.8133	0.8300	0.6070	0.6561	0.8586	0.7554
LFC	0.8085	0.8743	0.4669	0.6803	0.8076	0.8076
CRH	0.8083	0.8271	0.6120	0.6841	0.8828	0.6688

5.1 Implemented Algorithms

We implemented 10 truth inference algorithms and 4 task assignment algorithms in Python for comparative experiments. The truth inference algorithms are referred to as follows:

- TDH: This is our algorithm proposed in Section 3. For the prior distribution $Dir(\alpha)$, we set the hyperparameter $\alpha = (3, 3, 2)$ since correct values are more frequent than wrong values for most of the sources. For the other hyperparameters β and γ, we set every dimension of β and γ to 2.
- ACCU: It is the algorithm proposed in [7] which considers the dependencies between sources to find the truths. The algorithm exploits Bayesian analysis to find the dependencies.
- POPACCU: This denotes the algorithm in [9] which extends ACCU. It computes the distribution of the false values from the records while ACCU assumes that it is uniform.
- LFC: This algorithm is proposed in [31] and utilizes a confusion matrix to model a source's quality.
- CRH: It is proposed in [22] to resolve conflicts in heterogeneous data containing categorical and numerical attributes.
- LCA: It is a probabilistic model proposed in [30]. We select GuessLCA to be compared in this paper which is one of the best performers among the 7 algorithms proposed in [30].
- ASUMS: This is proposed in [2] by adapting an existing method SUMS [29] to hierarchical truth discovery.
- MDC: This denotes the truth discovery method designed for medical diagnose from non-expert crowdsourcing in [24].
- DOCS: This is the state-of-the-art technique presented in [39] that suggests the domain-sensitive worker model.
- VOTE: This is a baseline that selects a value with the highest frequency in the claimed values.

We implemented the following task assignment algorithms.

- *EAI*: This is our proposed algorithm in Section 4.
- *MB*: It is the task assignment algorithm used by DOCS [39].
- *QASCA*: It is a task assignment algorithm proposed in [41].
- *ME*: This is our baseline algorithm which utilizes an uncertainty sampling. It selects an object o^* whose confidence distribution has the maximum entropy. (i.e., $o^* = argmax_{o \in O}$ $(-\sum_{v \in V_o} \mu_{o,v} \cdot \log \mu_{o,v}))$

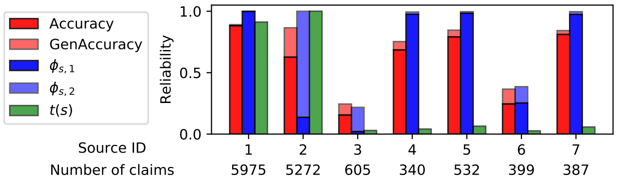


Figure 5: Source reliability distribution in *BirthPlaces*

Note that *EAI* and *MB* are the task assignment algorithms specially designed to work with *TDH* and *DOCS*, respectively. *QASCA* can work with truth inference algorithms based on probabilistic models such as *TDH*, *DOCS*, *LCA*, *ACCU* and *POPACCU*. All the truth inference algorithms can be combined with *ME*.

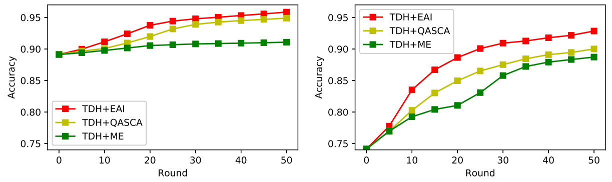
5.2 Truth Inference

We first provide the performances of the truth inference algorithms without using crowdsourcing in Table. 3.

***BirthPlaces*:** Our TDH outperforms all other algorithms in terms of all quality measures since TDH finds the exact truths by utilizing the hierarchical relationships. Since TDH estimates the reliabilities of the sources and workers by considering the hierarchies, it does not underestimate the reliabilities of the sources and workers. Thus, TDH also finds more correct values including the generalized truths. We will discuss the reliability estimation in detail at the end of this section by comparing TDH with ASUMS. LCA is the second-best performer and VOTE shows the lowest *Accuracy* among all compared algorithms. However, in terms of *GenAccuracy*, VOTE performs the second-best. It is because many websites claim the generalized values rather than the most specific value.

***Heritages*:** In terms of *AvgDistance* and *Accuracy*, TDH performs the best among those of the compared algorithms. VOTE shows the highest *GenAccuracy* because many sources provide the generalized truths. In fact, a high *GenAccuracy* with low *Accuracy* and *AvgDistance* can be easily obtained by providing the most general values for the truths. However, such values usually are not informative. Since our algorithm shows much higher *Accuracy* and much lower *AvgDistance* than VOTE, we can see that the estimated truth by TDH is more accurate and precise than the result from VOTE. *Heritages* contains many sources and most of the sources have a few claims. Thus, it is very hard to estimate the reliability of each source accurately. Therefore, most of the compared algorithms show worse performance than VOTE in terms of *AvgDistance*. In particular, ACCU has the lowest *Accuracy*. The reason is that ACCU requires many shared objects between two sources in order to accurately determine the dependency between the sources. The average accuracy of the sources in *Heritages* is 58.0% while that of the sources in *BirthPlaces* is 72.1%. Thus, every algorithm shows a lower *Accuracy* in this dataset than in *BirthPlaces*.

Comparison with ASUMS: Since ASUMS [2] is the only existing algorithm which utilizes hierarchies for truth inference, we show the statistics related to the reliability distributions estimated by TDH and ASUMS for *BirthPlaces* dataset in Figure 5. *Accuracy* and *GenAccuracy* represent the actual reliabilities of each source computed from the ground truths. Recall that $\phi_{s,1}$ and $\phi_{s,2}$ are the estimated probabilities of providing a correct value and a generalized correct value respectively for a source s by our TDH, as defined in Section 3. In addition, $t(s)$ is the estimated reliability of a source s by ASUMS which ignores the generalization level of each source. The reliabilities of the sources 4, 5 and 7 computed by ASUMS (i.e. $t(s)$) are quite different from the actual reliabilities (i.e., *Accuracy*). As we discussed in Section 1, for a pair of sources that provide different claimed values with an ancestor-descendant relationship in a hierarchy, existing methods may assume that one of the claimed values is incorrect. Thus, the reliability of the source with the assumed wrong value tends to become lower by the existing methods. ASUMS suffers from the same problem and underestimates the reliabilities of the sources 4, 5 and 7 which provide a small number of claimed values. Meanwhile, our proposed algorithm TDH accurately estimates the reliabilities of the sources by introducing another class of the claimed values (generalized truth).



(a) *BirthPlaces*

(b) *Heritages*

Figure 6: Evaluation of task assignment algorithms

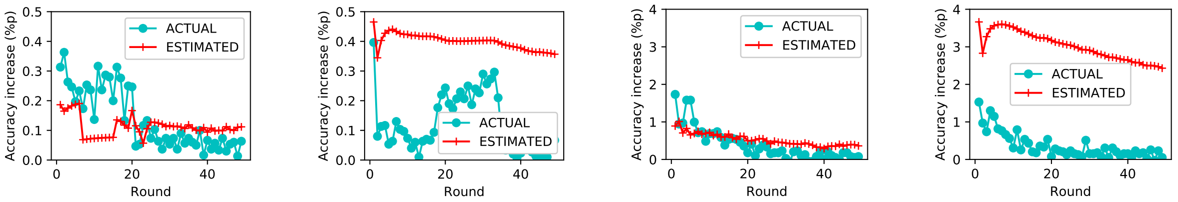
5.3 Task Assignment

Before providing the full comparison of all possible combinations of truth inference algorithms and task assignment algorithms, we first evaluate the task assignment algorithms with our proposed truth inference algorithm. We plotted the average *Accuracy* of the truth discovery algorithms with different task assignment algorithms for every 5 round in Figure 6. The points at the 0-th round represent the *Accuracy* of the algorithms without crowdsourcing. All algorithms show the same *Accuracy* at the beginning since they use the same truth inference algorithm TDH. As the round progresses, the *Accuracy* of TDH+EAI increases faster than those of all other algorithms. The *Accuracy* of TDH+ME is the lowest since ME selects a task based only on the uncertainty without estimating the accuracy improvement by the task.

As discussed in Section 4.1, our task assignment algorithm EAI estimates the accuracy improvement by considering the number of existing claimed values and the confidence distribution whereas QASCA considers the confidence distribution only. We plotted the actual and estimated accuracy improvements by EAI and QASCA in Figure 7. The graphs show that the estimated accuracy improvement by EAI is similar to the actual accuracy improvement while QASCA overestimates the accuracy improvement at every round. On average, the absolute estimation errors

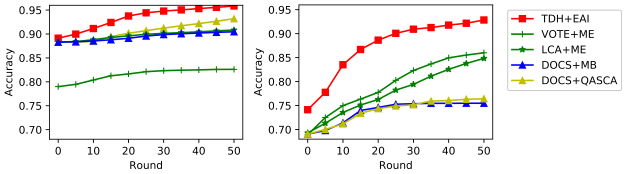
Table 4: *Accuracy* of the algorithms after the 50th round

	BirthPlaces				*Heritages*			
	EAI	MB	QASCA	ME	EAI	MB	QASCA	ME
TDH	**0.9601**	-	**0.9500**	**0.9109**	**0.9304**	-	**0.8999**	**0.8884**
DOCS	-	**0.9052**	0.9341	0.8842	-	**0.7546**	0.7661	0.7631
LCA	-	-	0.8823	0.9089	-	-	0.7136	0.8507
POPACCU	-	-	0.9295	0.8987	-	-	0.7512	0.8336
ACCU	-	-	0.8468	0.8257	-	-	0.5796	0.5896
ASUMS	-	-	-	0.8700	-	-	-	0.7427
CRH	-	-	-	0.9000	-	-	-	0.8459
MDC	-	-	-	0.8254	-	-	-	0.7241
LFC	-	-	-	0.8287	-	-	-	0.7327
VOTE	-	-	-	0.8261	-	-	-	0.8634



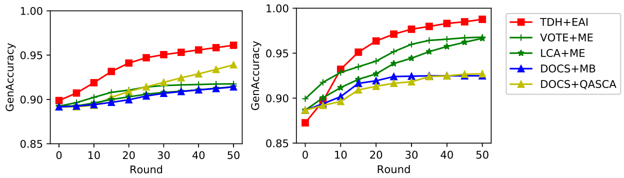
(a) *BirthPlaces*-EAI (b) *BirthPlaces*-QASCA (c) *Heritages*-EAI (d) *Heritages*-QASCA

Figure 7: Actual and estimated accuracy improvement by EAI and QASCA



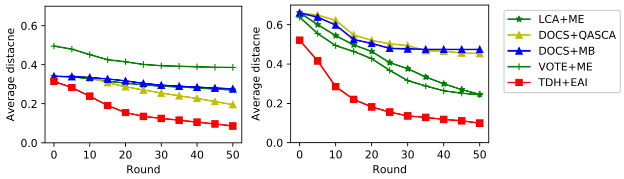
(a) *BirthPlaces* (b) *Heritages*

Figure 8: *Accuracy* with crowdsourced truth discovery



(a) **BirthPlaces** (b) **Heritages**

Figure 9: *GenAccuracy* with crowdsourced truth discovery



(a) **BirthPlaces** (b) **Heritages**

Figure 10: *AvgDistance* with crowdsourced truth discovery

from EAI are 0.08 and 0.26 percentage points (pps) while those errors from QASCA are 0.28 and 2.66 pps in *BirthPlaces* and *Heritages* datasets, respectively. This result confirms that EAI outperforms QASCA by effectively estimating the accuracy improvement. In terms of the other quality measures *GenAccuracy* and *AvgDistance*, our proposed EAI also outperforms the other task assignment algorithms in both datasets. Due to the lack of space, we omit the results with the other quality measures.

5.4 Simulated Crowdsourcing

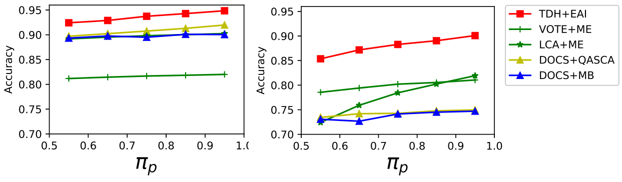
We evaluate the performance of crowdsourced truth discovery algorithms with the simulated crowdsourcing.

For all possible combinations of the implemented truth inference and task assignment algorithms, we show the *Accuracy* after 50 rounds of crowdsourcing in Table 4 where the impossible combinations are denoted by '-'. As expected, TDH+EAI has the highest *Accuracy* in both datasets for all possible combinations. The result also shows that both TDH and EAI contribute to increasing *Accuracy*. The improvement obtained by EAI can be estimated by comparing the result of TDH+EAI to that of the second performer TDH+QASCA. The accuracies of TDH+EAI in *BirthPlaces* and *Heritages* datasets are 1 and 3 percentage points (pps) higher than those of TDH+QASCA, respectively. In addition, for each combined task assignment algorithm, the improvement by TDH can be inferred by comparing the results with those of other truth inference algorithms. In both datasets, TDH shows the highest *Accuracy* among the applicable truth inference algorithms for each task assignment algorithm. For example, TDH+QASCA shows 2.6 and 13 pps higher *Accuracy* in *BirthPlaces* and *Heritages* datasets, respectively, than the second performer DOCS+QASCA among the combinations with QASCA. In the rest of the paper, we report *Accuracy*, *GenAccuracy* and *AvgDistance* of TDH+EAI, DOCS+MB, DOCS+QASCA, LCA+ME and VOTE+ME only since these combinations are the best or the second-best for each task assignment algorithm.

Cost efficiency: We plotted the average *Accuracy* of the tested algorithms for every 5 rounds in Figure 8. TDH+EAI shows the highest *Accuracy* for every round in both datasets. For the *BirthPlaces* dataset, DOCS+QASCA was the next best performer which achieved 0.9341 of *Accuracy* at the 50-th round. Meanwhile, TDH+EAI only needs 17 rounds of crowdsourcing to achieve the same *Accuracy*. Thus, TDH+EAI saved 66% of crowdsourcing cost compared to the second-best performer DOCS+ QASCA. Likewise, TDH+EAI reduced the crowdsourcing cost 74% in *Heritages* dataset compared to the next performer. In terms of *GenAccuracy* and *AvgDistance*, TDH+EAI also outperforms all the other algorithms as plotted in Figure 9 and Figure 10. The results confirm that TDH+EAI is the most efficient as it achieves the best qualities in terms of both *Accuracy* and *GenAccuracy*.

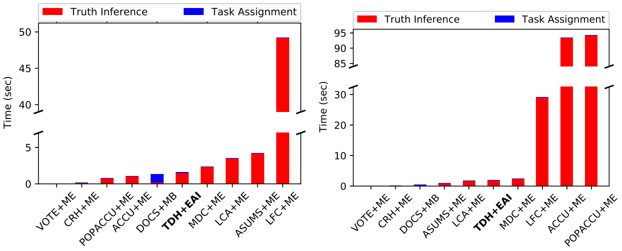
Varying π_p: We plotted the average *Accuracy* of all algorithms with varying the probability of correct answer π_p of simulated workers for *BirthPlaces* and *Heritages* datasets in Figure 11(a) and Figure 11(b), respectively. As we can easily expect, the accuracies increase with growing π_p for most of the algorithms. For both datasets, TDH+EAI achieves the best accuracy with all values of π_p. In Heritages dataset, a source provided less than 10 claims on average and it makes difficult for truth discovery algorithms to estimate the reliabilities of sources. Therefore, the baseline VOTE+ME shows good performance on Heritages dataset. Meanwhile, the performance of the state-of-the-art DOCS is significantly degraded on the Heritages dataset.

Execution times: We plotted the average execution times of the tested algorithms over every round in Figure 12. VOTE, CRH+ME, DOCS+MB and TDH+EAI run in less than 2.0 seconds per round on average for both datasets. Other algorithms except for ACCU+ME, POPACCU+ME and LFC+ME also take less than 5 seconds, which is acceptable for crowdsourcing. Since LFC builds the confusion matrix whose size is the square of the number of candidate values, LFC is the slowest with *BirthPlaces* data. On the other hand, for *Heritages* dataset which is collected



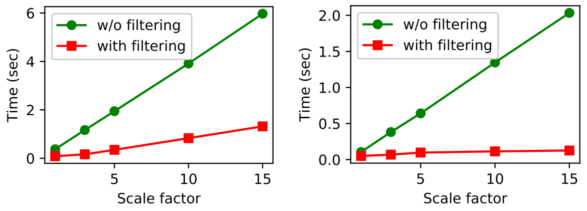
(a) *BirthPlaces* (b) *Heritages*

Figure 11: Varying π_p



(a) *BirthPlaces* (b) *Heritages*

Figure 12: Execution time per round



(a) *BirthPlaces* (b) *Heritages*

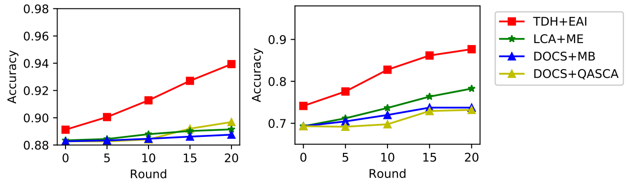
Figure 13: Execution time for task assignment per round

from much more sources than *BirthPlaces* dataset, ACCU and POPACCU take longer time for truth inference to calculate the dependencies between sources.

Effects of the filtering for task assignments: To test the scalability of our algorithm, we increase the size of both datasets by duplicating the data by upto 15 times. In Figure 13, with increasing data size, we plotted the execution times of our task assignment algorithm EAI with and without exploiting the upper bound proposed in Section 4.3. The filtering technique saved 78% and 94% of the computation time for the task assignment at the scale factor 15. The graphs show that the proposed upper bound enables us to scale for large data effectively. For the total execution time, including the truth inference, the filtering reduced 21% and 6% of the execution time on *BirthPlaces* and *Heritages* respectively at the scale factor 15.

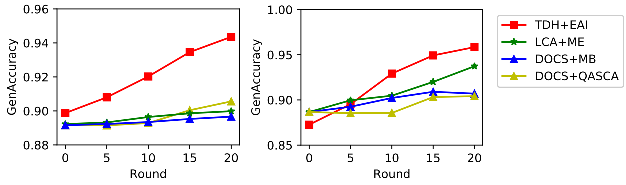
5.5 Crowdsourcing with Human Annotators

We evaluated the performance of the truth discovery algorithm by crowdsourcing real human annotations. For this experiment, we selected DOCS+QASCA, DOCS+MB and LCA+ME for comparison with the proposed algorithm TDH+EAI. This is because they are the best existing algorithms for each task assignment algorithm. We conducted this experiment with 10 human annotators for 20 rounds on our own crowdsourcing system. For each worker, we assigned 5 tasks in each round. Figure 14,15 and 16 show the performances of the algorithms against the rounds.



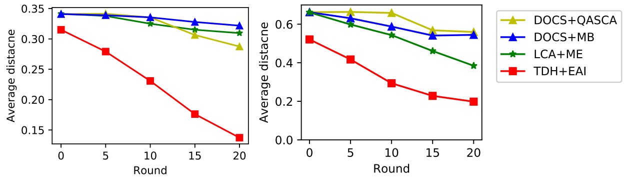
(a) *BirthPlaces* (b) *Heritages*

Figure 14: *Accuracy* with human annotations



(a) *BirthPlaces* (b) *Heritages*

Figure 15: *GenAccuracy* with human annotations



(a) *BirthPlaces* (b) *Heritages*

Figure 16: *AvgDistance* with human annotations

For both of the datasets, the results confirm that the proposed TDH+EAI algorithm outperforms the compared algorithms as in the previous simulations. Without crowdsourcing, the other algorithms show a higher *GenAccuracy* than TDH for *Heritages* dataset, because these algorithms tend to estimate the truths with more generalized form than TDH does. However, TDH+EAI shows the highest *GenAccuracy* after the 3rd round because it correctly estimates the reliabilities and the generalization levels of the sources by using the hierarchy. For *BirthPlaces* dataset, *Accuracies* of the algorithms increase a little bit faster than those in the experiment with simulated crowdsourcing. However, for *Heritages* dataset, *Accuracies* of the algorithms increase much slower than in the experiment with simulated crowdsourcing. It seems that finding the locations of a world heritages is a quite harder task than finding the birthplaces of celebrities because the birthplaces are often big cities (such as LA), which are familiar to workers, but World Cultural Heritages and World Natural Heritages are often located in unfamiliar regions.

5.6 Crowdsourcing with AMT

We evaluate the performances of TDH+EAI, DOCS+QASCA, DOCS+MB and LCA+ME based on the answers collected from Amazon Mechanical Turk (AMT). We collected answers for all objects in *Heritages* dataset from 20 workers in AMT. We made the collected answers available at http://kdd.snu.ac.kr/home/datasets/tdh.php. To evaluate the algorithms based on the collected answers, we assign 5 tasks for each worker in a round. We plotted the performance of the algorithms in Figure 17. Since we use more workers than we did in Section 5.5, the performances

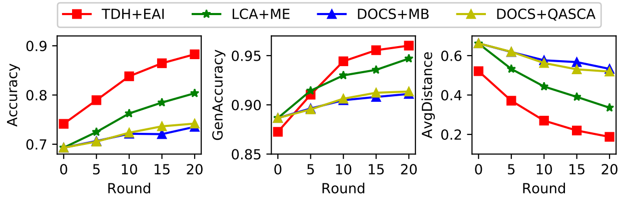


Figure 17: Crowdsourced truth discovery in *Heritages*

improve a little bit faster, but the trends are very similar to those with 10 human annotators in the previous section. We observe that our TDH+EAI outperforms all compared algorithms even with a commercial crowdsourcing platform.

5.7 Multi-truths Discovery Algorithms

Since there are multiple correct values including generalized values, we also implement multi-truth discovery algorithms such as DART[27], LFC[31] and LTM[38] to compare with our TDH algorithm. Since the multi-truths discovery algorithms independently generate the correct values, they may output the true values where there exist a pair of true values without ancestor-descendant relationship in the hierarchy. For example, from the given claimed values in Table 1, the multi-truth algorithms can answer that the 'Statue of Liberty' is located in LA and Liberty island. In this case, we cannot evaluate the result by our evaluation measures *Accuracy*, *GenAccuracy* and *AvgDistance*. Thus, to evaluate the performance of the tested algorithms, we utilize precision, recall and F1-score which are the evaluation measures typically used for multi-truths discovery. To use the multi-truths algorithms and the evaluation measures, we treat the ancestors of v and v itself as the multi-truths of v. LFC can work as either a single truth algorithm or a multi-truths algorithm. We refer to the multi-truth version of LFC as LFC-MT to avoid the confusion.

Table 5: Performance of truth discovery algorithms

		Dataset					
		BirthPlaces			*Heritages*		
Algorithm		Precision	Recall	F1	Precision	Recall	F1
Single truth	TDH	**0.899**	**0.921**	**0.910**	0.873	0.795	**0.832**
	VOTE	0.892	0.804	0.846	**0.899**	0.717	0.798
	LCA	0.892	0.913	0.903	0.878	0.711	0.786
	DOCS	0.892	0.913	0.902	0.887	0.722	0.796
	ASUMS	0.857	0.888	0.872	0.741	0.660	0.698
	POPACCU	0.847	0.858	0.852	0.859	0.694	0.768
	LFC	0.874	0.838	0.856	0.808	0.727	0.765
	MDC	0.844	0.853	0.848	0.807	0.792	0.800
	ACCU	0.830	0.842	0.836	0.766	0.631	0.692
	CRH	0.827	0.833	0.830	0.883	0.716	0.791
Multi -truths	LFC-MT	0.763	0.723	0.742	0.898	0.684	0.777
	DART	0.590	0.855	0.698	0.357	**0.994**	0.525
	LTM	0.780	0.472	0.588	0.871	0.672	0.759

Table 6: Performance evaluation for numerical data

	Change rate		Open price		EPS	
Algorithm	MAE	R/E	MAE	R/E	MAE	R/E
TDH	**0.0006**	**0.1011**	**0.0195**	**0.0354**	**0.0352**	**1.9513**
LCA	**0.0006**	**0.1011**	**0.0195**	**0.0354**	0.3831	16.2212
CRH	0.0020	1.6339	**0.0195**	**0.0354**	0.0610	1.9882
CATD	0.0104	2.3529	0.0211	0.0395	0.0803	3.2059
VOTE	**0.0006**	**0.1011**	**0.0195**	**0.0354**	0.0765	2.8402
MEAN	0.2837	30.8747	0.4047	0.5782	0.1762	7.3937

Table 5 shows the performance of the truth discovery algorithms in terms of precision, recall and F1-score. For both datasets, the TDH algorithm is the best in terms of F1-score. Recall that the VOTE algorithm tends to find a generalized value of the exact truth. Since a generalized truth generates a small number of multi-truths, the VOTE algorithm shows the highest precision in *Heritages* dataset. However, since its recall is much lower than that of our TDH algorithm, the F1-score of the VOTE algorithm is lower than that of the TDH algorithm. Similarly, although the DART algorithm has the highest recall in *Heritages* dataset, the precision of the DART algorithm is the smallest among the precisions of all compared algorithms.

5.8 Performance on a Numerical Dataset

To evaluate the extension to numerical data, we conducted an experiment on the stock datatset [23] which is trading data of 1000 stock symbols from 55 sources on every work day in July 2011. The detailed description of the data can be found in [23]. As we discussed at the end of Section 3.2, we can utilize our TDH algorithm for numeric dataset with implied hierarchy. We select three attributes 'change rate', 'open price' and 'EPS' of the dataset, and compared our TDH algorithm with the LCA, CRH, CATD[21], VOTE and MEAN algorithms. Among the second best performers DOCS and LCA in Table 4, we use only LCA for this experiment since DOCS requires the domain information while it is not available for this dataset. In addition to LCA, we implemented and tested the two algorithms CRH[22] and CATD[21] which are designed to find the truth in numerical data. Recall that VOTE is a baseline algorithm which selects the candidate value collected from majority sources. We also implemented a baseline algorithm, called MEAN, which estimates the correct value as the average of the claimed numeric values.

Table 6 shows the mean squared error (MAE) and the relative error (R/E) of the tested algorithms. The TDH algorithm performs the best for every attribute. The MEAN and CATD algorithms show worse performance than the other algorithms. Since they utilize an average or a weighted average of the claimed values, they are sensitive to outliers. The result confirms that our TDH algorithm is effective even for numerical data.

6 RELATED WORK

The problem of resolving conflicts from multiple sources (i.e., truth discovery) has been extensively studied [4, 5, 5, 7, 9, 16, 22, 24, 26, 27, 31, 33, 35–39, 42]. Truth discovery for categorical data has been addressed in [5, 7, 9, 22, 24, 31, 36, 39]. According to a recent survey [40], LFC[31] and CRH[22] perform the best in an extensive experiment with the truth discovery algorithms [4, 5, 16, 21, 35, 40, 42]. There exist other interesting algorithms [7, 9, 24, 39] which are not evaluated together in [40]. Accu[7] and PopAccu[9] combine the conflicting values extracted from different sources for the knowledge fusion [8]. They consider the dependencies between data sources to penalize the copiers' claims. DOCS[39] utilizes the domain information to consider the different levels of worker expertises on various domains. MDC[24] is a truth discovery algorithm devised for crowdsourcing-based medical diagnosis. The works in [26, 33, 37] studied how to resolve conflicts in numerical data from multiple sources.

The truth discovery algorithms in [30, 37–39] are based on probabilistic models. Resolving the conflicts in numerical data is addressed in [37] and discovering multiple truths for an object is studied in [38]. Probabilistic models for finding a single truth for each object is proposed in [30, 39]. However, none of those algorithms exploit the hierarchical relationships of claimed values for truth discovery. The work in [2] adopts an existing algorithm to consider hierarchical relationships.

Task assignment algorithms[3, 11, 14, 28, 39, 41] in crowdsourcing have been studied widely in recent years. The works in [3, 39, 41] can be applied to our crowdsourced truth discovery. For task assignment, AskIt[3] selects the most uncertain object for a worker. Meanwhile, the task assignment algorithm in [39] selects the object which is expected to decrease the entropy of the confidence the most. QASCA [41] chooses an object which is likely to most increase the accuracy. Since QASCA outperforms AskIt in the experiments presented in [39, 41], we do not consider AskIt in our experiments. In [14], task assignment for binary classification was investigated but it is not applicable to our problem to find the correct value among multiple conflicting values. Meanwhile, the task assignment algorithm is proposed in [28] for the case when the required skills for each task and the skill set of every worker is available. However, it is not applicable to our problem. A task assignment algorithm proposed in [11] assigns every object to a fixed number of workers. However, since we already have claimed values from sources, we do not have to assign all objects to workers.

7 CONCLUSION

In this paper, we first proposed a probabilistic model for truth inference to utilize the hierarchical structures in claimed values and an inference algorithm for the model. Furthermore, we proposed an efficient algorithm to assign the tasks in crowdsourcing platforms. The performance study with real-life datasets confirms the effectiveness of the proposed algorithms.

ACKNOWLEDGMENTS

We appreciate the reviewers for providing their insightful comments. This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT (NRF-2017M3C4A7063570). This research was also supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2016R1D1A1A02937186).

REFERENCES

[1] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *The semantic web*. Springer, 722–735.

[2] Valentina Beretta, Sébastien Harispe, Sylvie Ranwez, and Isabelle Mougenot. 2016. How Can Ontologies Give You Clue for Truth-Discovery? An Exploratory Study. In *WIMS*. 15.

[3] Rubi Boim, Ohad Greenshpan, Tova Milo, Slava Novgorodov, Neoklis Polyzotis, and Wang-Chiew Tan. 2012. Asking the right questions in crowd data sourcing. In *ICDE*. 1261–1264.

[4] Alexander Philip Dawid and Allan M Skene. 1979. Maximum likelihood estimation of observer error-rates using the EM algorithm. *Applied statistics* (1979), 20–28.

[5] Gianluca Demartini, Djellel Eddine Difallah, and Philippe Cudré-Mauroux. 2012. ZenCrowd: leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. In *WWW*. 469–478.

[6] Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. 2014. Knowledge Vault: A Web-scale Approach to Probabilistic Knowledge Fusion. In *SIGKDD*. 601–610.

[7] Xin Luna Dong, Laure Berti-Equille, and Divesh Srivastava. 2009. Integrating conflicting data: the role of source dependence. *PVLDB* 2, 1 (2009), 550–561.

[8] Xin Luna Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Kevin Murphy, Shaohua Sun, and Wei Zhang. 2014. From data fusion to knowledge fusion. *PVLDB* 7, 10 (2014), 881–892.

[9] Xin Luna Dong, Barna Saha, and Divesh Srivastava. 2012. Less is more: Selecting sources wisely for integration. In *PVLDB*, Vol. 6. 37–48.

[10] Xin Luna Dong and Divesh Srivastava. 2015. Knowledge curation and knowledge fusion: challenges, models and applications. In *SIGMOD*. 2063–2066.

[11] Ju Fan, Guoliang Li, Beng Chin Ooi, Kian-lee Tan, and Jianhua Feng. 2015. icrowd: An adaptive crowdsourcing framework. In *SIGMOD*. 1015–1030.

[12] Ju Fan, Meiyu Lu, Beng Chin Ooi, Wang-Chiew Tan, and Meihui Zhang. 2014. A hybrid machine-crowdsourcing system for matching web tables. In *ICDE*. 976–987.

[13] Daniel Haas, Jiannan Wang, Eugene Wu, and Michael J Franklin. 2015. Clamshell: Speeding up crowds for low-latency data labeling. *PVLDB* 9, 4 (2015), 372–383.

[14] Chien-Ju Ho, Shahin Jabbari, and Jennifer Wortman Vaughan. 2013. Adaptive task assignment for crowdsourced classification. In *ICML*. 534–542.

[15] David R Karger, Sewoong Oh, and Devavrat Shah. 2011. Iterative learning for reliable crowdsourcing systems. In *NIPS*. 1953–1961.

[16] Hyun-Chul Kim and Zoubin Ghahramani. 2012. Bayesian classifier combination. In *AISTATS*. 619–627.

[17] Younghoon Kim, Woohwan Jung, and Kyuseok Shim. 2017. Integration of graphs from different data sources using crowdsourcing. *Information Sciences* 385 (2017), 438–456.

[18] Younghoon Kim, Wooyeol Kim, and Kyuseok Shim. 2017. Latent ranking analysis using pairwise comparisons in crowdsourcing platforms. *Inf. Syst.* 65 (2017), 7–21.

[19] David D Lewis and William A Gale. 1994. A sequential algorithm for training text classifiers. In *SIGIR*. 3–12.

[20] Guoliang Li, Jiannan Wang, Yudian Zheng, and Michael J Franklin. 2016. Crowdsourced data management: A survey. *TKDE* 28, 9 (2016), 2296–2319.

[21] Qi Li, Yaliang Li, Jing Gao, Lu Su, Bo Zhao, Murat Demirbas, Wei Fan, and Jiawei Han. 2014. A confidence-aware approach for truth discovery on long-tail data. *PVLDB* 8, 4 (2014), 425–436.

[22] Qi Li, Yaliang Li, Jing Gao, Bo Zhao, Wei Fan, and Jiawei Han. 2014. Resolving conflicts in heterogeneous data by truth discovery and source reliability estimation. In *SIGMOD*. 1187–1198.

[23] Xian Li, Xin Luna Dong, Kenneth Lyons, Weiyi Meng, and Divesh Srivastava. 2012. Truth finding on the deep web: Is the problem solved?. In *PVLDB*, Vol. 6. 97–108.

[24] Yaliang Li, Nan Du, Chaochun Liu, Yusheng Xie, Wei Fan, Qi Li, Jing Gao, and Huan Sun. 2017. Reliable Medical Diagnosis from Crowdsourcing: Discover Trustworthy Answers from Non-Experts. In *WSDM*. 253–261.

[25] Yaliang Li, Jing Gao, Chuishi Meng, Qi Li, Lu Su, Bo Zhao, Wei Fan, and Jiawei Han. 2016. A Survey on Truth Discovery. *SIGKDD Explor. Newsl.* 17, 2 (Feb. 2016), 1–16.

[26] Yaliang Li, Qi Li, Jing Gao, Lu Su, Bo Zhao, Wei Fan, and Jiawei Han. 2015. On the discovery of evolving truth. In *SIGKDD*. 675–684.

[27] Xueling Lin and Lei Chen. 2018. Domain-aware multi-truth discovery from conflicting sources. *PVLDB* 11, 5 (2018), 635–647.

[28] Panagiotis Mavridis, David Gross-Amblard, and Zoltán Miklós. 2016. Using hierarchical skills for optimized task assignment in knowledge-intensive crowdsourcing. In *WWW*. 843–853.

[29] Jeff Pasternack and Dan Roth. 2010. Knowing what to believe (when you already know something). In *COLING*. 877–885.

[30] Jeff Pasternack and Dan Roth. 2013. Latent Credibility Analysis. In *WWW*. 1009–1020.

[31] Vikas C Raykar, Shipeng Yu, Linda H Zhao, Gerardo Hermosillo Valadez, Charles Florin, Luca Bogoni, and Linda Moy. 2010. Learning from crowds. *JMLR* 11, Apr (2010), 1297–1322.

[32] Princeton University. 2010. About WordNet. https://wordnet.princeton.edu/

[33] Mengting Wan, Xiangyu Chen, Lance Kaplan, Jiawei Han, Jing Gao, and Bo Zhao. 2016. From Truth Discovery to Trustworthy Opinion Discovery: An Uncertainty-Aware Quantitative Modeling Approach. In *SIGKDD*. 1885–1894.

[34] Jiannan Wang, Tim Kraska, Michael J Franklin, and Jianhua Feng. 2012. Crowder: Crowdsourcing entity resolution. *PVLDB* 5, 11 (2012), 1483–1494.

[35] Jacob Whitehill, Ting-fan Wu, Jacob Bergsma, Javier R Movellan, and Paul L Ruvolo. 2009. Whose vote should count more: Optimal integration of labels from labelers of unknown expertise. In *NIPS*. 2035–2043.

[36] Xiaoxin Yin, Jiawei Han, and S Yu Philip. 2008. Truth discovery with multiple conflicting information providers on the web. *TKDE* 20, 6 (2008), 796–808.

[37] Bo Zhao and Jiawei Han. 2012. A probabilistic model for estimating real-valued truth from conflicting sources. In *QDB*.

[38] Bo Zhao, Benjamin IP Rubinstein, Jim Gemmell, and Jiawei Han. 2012. A bayesian approach to discovering truth from conflicting sources for data integration. *PVLDB* 5, 6 (2012), 550–561.

[39] Yudian Zheng, Guoliang Li, and Reynold Cheng. 2016. DOCS: a domain-aware crowdsourcing system using knowledge bases. *PVLDB* 10, 4 (2016), 361–372.

[40] Yudian Zheng, Guoliang Li, Yuanbing Li, Caihua Shan, and Reynold Cheng. 2017. Truth inference in crowdsourcing: is the problem solved? *PVLDB* 10, 5 (2017), 541–552.

[41] Yudian Zheng, Jiannan Wang, Guoliang Li, Reynold Cheng, and Jianhua Feng. 2015. QASCA: A quality-aware task assignment system for crowdsourcing applications. In *SIGMOD*. 1031–1046.

[42] Denny Zhou, Sumit Basu, Yi Mao, and John C Platt. 2012. Learning from the wisdom of crowds by minimax entropy. In *NIPS*. 2195–2203.

A Utility-Preserving and Scalable Technique for Protecting Location Data with Geo-Indistinguishability

Ritesh Ahuja
University of Southern California
riteshah@usc.edu

Gabriel Ghinita
University of Massachusetts Boston
gabriel.ghinita@umb.edu

Cyrus Shahabi
University of Southern California
shahabi@usc.edu

ABSTRACT

Location-based apps provide users with personalized services tailored to their geographical position. This is highly-beneficial for mobile users, who are able to find points of interest close to their location, or connect with nearby friends. However, sharing location data with service providers also introduces privacy concerns. An adversary with access to fine-grained user locations can infer private details about individuals. *Geo-indistinguishability (GeoInd)* adapts the popular *differential privacy (DP)* model to make it suitable for protecting users' location information. However, existing techniques that implement GeoInd have major drawbacks. Some solutions, such as the planar Laplace mechanism, significantly lower data utility by adding excessive noise. Other approaches, such as the optimal mechanism, achieve good utility, but only work for small sets of candidate locations due to the use of computationally-expensive linear programming. In most cases, locations are used to answer online queries, so a quick response time is essential. In this paper, we propose a technique that achieves GeoInd and scales to large datasets while preserving data utility. Our central idea is to use the composability property of GeoInd to create a multiple-step algorithm that can be used in conjunction with a spatial index. We preserve utility by applying accurate GeoInd mechanisms and we achieve scalability by pruning the solution search space with the help of the index when seeking high-utility outcomes. Our extensive performance evaluation on top of real location datasets from social media apps shows that the proposed technique outperforms significantly the benchmark in terms of utility and/or computational overhead.

1 INTRODUCTION

The unprecedented growth in the area of mobile apps allows users to enjoy personalized services and receive information customized to their locations. In return for sharing their locations with the service provider, users can find restaurants and shopping malls nearby, can plan their travel itinerary with ease, or can connect with nearby friends. While the benefits of personalized location services are clear, there are also increasing risks associated with the sharing of fine-grained individual locations. A significant body of research [15–17, 21] shows that uncontrolled sharing of users' whereabouts can lead to a wide range of attacks, from stalking and assault, to various privacy breaches that may disclose sensitive personal details such as one's health status, political or religious orientation, etc.

The need for protecting users' locations has been the subject of intense study by the research community for more than a decade. Initial approaches considered cloaking of user locations to decrease the precision of coordinate reporting. However, it has been shown [15, 17] that this category of solutions does not provide sufficient protection, especially when dealing with sophisticated adversaries with access to background knowledge. Another category of techniques uses encryption [15]. While the privacy strength achieved with encryption is high, there are two drawbacks: first, only simple queries (e.g., nearest-neighbors) can be answered; second, the computational overhead of processing encrypted queries is high and requires extensive server-side changes.

The prominent model of *differential privacy (DP)* [13] has become the de-facto standard for data protection in the last few years. While there is work that shows how location can be protected with DP, these solutions (and in fact, the model itself) assume an aggregate release, where data are pooled together from a population of users over a period of time, and then aggregates are disclosed covering various regions of the dataspace. The objective of this publication model is to hide the presence of an individual in a released dataset, but cannot be used in operational mode (i.e., to protect the location attributes of a specific user asking a query). The more recent model of *Geo-indistinguishability (GeoInd)* [1, 2, 5] extends DP with a new distinguishability metric [6] which captures precisely the operational setting, and prevents the association of a user with an exact location. Specifically, GeoInd adds random noise to the actual user location in a way that prevents an adversary from inferring with high probability the user's whereabouts, *regardless* of the amount of background knowledge available to the attacker. Its powerful semantic protection guarantees derived from DP make GeoInd the only viable approach available at the present time for protecting locations in the operational setting[1].

Despite GeoInd being a promising model, existing techniques that implement it have some important drawbacks. We provide a simplified argument, without going into technical details, as the formal aspects of GeoInd will only be introduced in Section 2. In essence, GeoInd achieves protection by adding noise, but noise also decreases data utility. In addition, the protection achieved is independent of the adversary's background knowledge, further referred to as *prior*. However, an interesting result in [2] shows that, if one is aware of the adversary's prior, then it is possible to construct a GeoInd mechanism that improves utility considerably, while still providing protection for *any* prior. This result is important, because in practice locations where a user is expected to be are not random. For instance, datasets derived from social media apps show that users *check in* (i.e., report their location) at a set of well-defined *points of interests (POI)*. This discrete set of POI, combined with some other background knowledge factors (e.g., popularity of various POI) effectively functions as the adversary's prior. One can construct mechanisms for enforcing GeoInd in two ways: at one extreme, completely ignore the prior, and simply generate a reported location by adding (planar) Laplace noise to the actual location [1]. This approach is very efficient computationally, but can yield poor utility by generating large

[1] Another model exists which functions in the operational setting [24], but it only works against a well-defined set of adversarial prior knowledge. Constructing such a detailed prior is often not feasible in practice.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.20

noise. At the other extreme, one can take into account the prior and attempt to add the *optimal* amount of noise to any possible user location such that the expected distance between actual and reported locations is minimized (while still preserving GeoInd). The latter is called *optimal mechanism* [2] and is implemented using linear programming (LP).

Consider a regular grid on top of a set of locations of interest in a city area (grid partitioning is used for ease of presentation, the concept we convey remains valid for any set of discrete, or *logical* locations [12]). Each cell has a certain prior value, corresponding, for instance, to the weighted popularity of all POI in the respective cell. An optimal GeoInd protection mechanism will produce an output given by a linear program that considers all possible combinations of actual and reported cells, with specific constraints related to location protection. The computational complexity is cubic to the number of cells. Even for a relatively small search space, such as $12 \times 12 = 144$ cells, the execution time for LP solvers can be in the order of hours (we provide exact measurement results in Section 6). Such an approach is too slow.

Our proposed approach uses a multi-step algorithm that applies GeoInd recursively along a data indexing structure, according to the *composability* property of DP (see Section 2 for details). We illustrate this in Figure 1, on a three-level index. The actual location is the black dot. Level A consists of nine cells, and assume $A5$ is selected by the mechanism in the first step. In step two, we "zoom in" cell $A5$, and re-apply the mechanism on top of a four-cell grid this time. Finally, at the third level, $C3$ is chosen for release. The privacy budget is split across the three levels. Our approach can work with a relatively fine-grained grid at the leaf level; in this example, at the leaf level we obtain 144 cells, same as in the non-hierarchical case. However, in our case the size of the problem at each step consists of nine, four and four cells, respectively, so our approach will perform much faster. Also, since index node selection is performed at each step according to the output of the mechanism at the previous index level, GeoInd is preserved. Note that, the actual location may fall outside the selected cell at each level, but that is less likely to occur at the higher (i.e., coarser) index levels, so utility will not suffer significantly.

Although the underlying concept is simple, our approach raises a number of difficult challenges. Specifically, one must take carefully into account several factors that affect utility and performance, such as: how to determine the parameters of the index, such as height and fan-out; and how to decide how to split the privacy budget across distinct index levels. We provide an analytical model to synthesize important relationships between system parameters on one hand, and performance metrics such as utility and computational overhead on the other.

Our specific contributions include:

- We identify important drawbacks of existing techniques for geo-indistinguishability in terms of utility and/or computational overhead.
- We propose a multi-step algorithm that applies GeoInd mechanisms in conjunction with an index data structure in order to prune the search space when seeking optimal solutions.
- We develop an analytical cost model to characterize the utility-performance trade-off, and to select an appropriate set of parameter values for our approach.
- We perform an extensive experimental evaluation to measure utility and execution time on real datasets, and we show that the proposed approach significantly outperforms the benchmark.

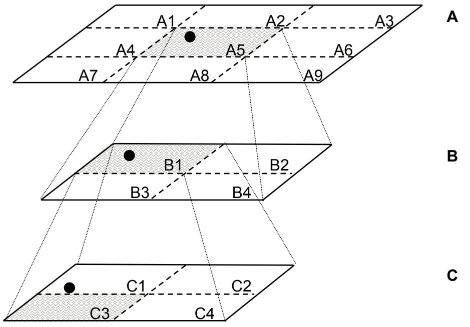


Figure 1: Multi-Step Approach Overview

The rest of the paper is organized as follows: Section 2 provides essential background information, followed by an overview of our approach in Section 3. We provide the details of our algorithm in Section 4. We derive an analytical model for performance characterization in Section 5 and we present a detailed experimental evaluation in Section 6. We survey related work in Section 7 and conclude with directions for future research in Section 8.

2 PRELIMINARIES

We introduce fundamental concepts used in the rest of the paper, such as the privacy model we adopt, and the mechanisms that implement it. We also discuss utility metrics, as well as the important property of *composability* on which our proposed multi-step algorithm for location protection relies.

2.1 GeoInd Definition

Geo-indistinguishability (GeoInd) was first introduced in [1], and extends the popular protection model of differential privacy (DP) [13]. DP is designed to prevent an adversary from learning with significant probability whether an individuals' data is present or not in a dataset. DP is achieved by adding random noise to the result of aggregate queries (e.g., count or sum). The fundamental concept behind DP is to bound the probability of distinguishing between the results of computations performed on *neighboring* datasets, defined as sets of records that differ in at most one entry. More formally, when the Hamming distance between two candidate datasets D_1 and D_2 is 1, then an adversary cannot determine with significant probability whether D_1 or D_2 was used to produce a certain result. The corresponding probabilities P_1 and P_2 are similar within a small multiplicating factor e^{ε}, where ε is called *privacy budget* (lower ε values correspond to tighter privacy settings).

However, DP is not suitable to protect the locations of mobile users in the *online* setting, for two reasons: first, DP works only for aggregate queries, and cannot be directly used for sanitizing individual records; second, DP can hide the presence of an individual record in a dataset, but it cannot protect the attribute values of individual data records, due to the *distinguishability metric* used (Hamming distance). The work in [4] extends DP by proposing a more flexible distinguishability metrics, resulting to the GeoInd definition.

Consider a set $\mathcal{X}$ of possible user locations, a set $\mathcal{Z}$ of possible reported locations (the two sets $\mathcal{X}$ and $\mathcal{Z}$ may coincide), and a distance metric $d_{\mathcal{X}}$. A probabilistic mechanism $K : \mathcal{X} \to \mathcal{P}(\mathcal{Z})$

takes as input a location in $\mathcal{X}$, and obfuscates it to produce some location $\mathcal{Z}$, which is reported to a location-based service (since K is a probabilistic function, its co-domain is the power-set of $\mathcal{Z}$, as it assigns each output location a probability of being reported). Mechanism K is said to achieve geo-indistinguishability with privacy level ε if for all $x, x' \in \mathcal{X}, z \in \mathcal{Z}$:

$$K(x)(z) \leq e^{\varepsilon d_{\mathcal{X}}(x,x')} \cdot K(x')(z) \tag{1}$$

Intuitively, Eq. (1) specifies that, given a reported value z, an adversary cannot distinguish whether the user location is x or x' by a factor larger than $e^{\varepsilon d_{\mathcal{X}}(x,x')}$. In practice, one appropriate choice for the $d_{\mathcal{X}}$ metric is the Euclidean distance, denoted in the rest of the paper as $d(\cdot, \cdot)$. Effectively, geo-indistinguishability enforces a constraint on the distributions $K(x)$, $K(x')$ produced by two different points x, x'. The authors in [5] also propose a practical interpretation of this definition: for all locations x' within a radius r from x, the user enjoys εr-indistinguishability. For small values of r, the adversary gains little information, i.e., s/he cannot pinpoint the user within a small region of a city (e.g., specific bar, restaurant, building). However, for large r values, the probability of locating the user within a circle of radius r increases, which preserves utility of location reporting. In other words, the mobile user is still able to retrieve information relevant to her current city or neighborhood, at a coarser granularity.

2.2 Utility and Composability

Utility. In order to enforce GeoInd, actual user locations must be perturbed through addition of random noise. Inherently, there is utility loss associated to this process: location-based services will return information relevant to the reported location, instead of the actual one. Ideally, a GeoInd mechanism should add the minimum amount of noise to achieve the GeoInd constraints, so utility is maximized and the deterioration in the quality of service received by the user is constrained. Inspired from previous work [1, 2, 5] we use the following practical measures of utility loss[2]:

- *Euclidean distance d:* this utility metric measures the Euclidean distance between the reported and actual user locations. It is a natural metric to describe the additional distance traveled by the user as a result of location obfuscation (assuming free-space movement). For instance, the nearest-neighbor Italian restaurant of the *reported* location could be several hundred meters away from the user location, even though there is another one situated only meters away from the user.
- *Squared Euclidean distance d^2:* the square of the Euclidean distance between reported and actual locations is also important, as it estimates the number of results that a user receives in a certain area. In practice, the user—knowing that the location will be obfuscated—may ask for more results. For instance, instead of asking for restaurants in a 200 meters radius, the user may increase the range to increase the chance that a nearby POI is actually returned. The size of the result set, and the effort required to filter the results, are estimated to increase linearly with the area of search, so the squared Euclidean distance may be a good predictor of utility loss.

Composability. An important property of GeoInd (inherited from its parent model DP) is *composability*. Specifically, if two mechanisms are applied in succession with budgets ε_1 and ε_2, the net amount of privacy obtained is equivalent to a privacy budget $(\varepsilon_1 + \varepsilon_2)$. Conversely, given a total privacy budget ε, we can split it into several terms and assign each term to a separate processing step, effectively obtaining a combined multiple-step algorithm that achieves the same amount of protection.

2.3 GeoInd Mechanisms

The GeoInd definition specifies the constraints that a protection technique must satisfy in order to obtain strong privacy guarantees. A GeoInd *mechanism* is a specific construction that achieves the GeoInd requirements. Next, we enumerate the most prominent mechanisms that implement GeoInd, introduced in [1, 5]. These mechanisms achieve various trade-offs in terms of performance and utility.

Planar Laplace Mechanism (PL). The simplest approach to preserving GeoInd is the Laplace mechanism [1]. The key idea is to perturb the exact user location by additive random noise drawn from a bi-variate Laplacian distribution with density defined as:

$$D_\varepsilon(x, z) = \frac{\varepsilon^2}{2\pi} e^{-\varepsilon d(x,z)} \tag{2}$$

Drawing an element from this distribution can be done by (i) switching to polar coordinates, (ii) selecting an angle θ uniformly from $[0, 2\pi)$, and (iii) selecting a radius r from the Gamma distribution $\Gamma_\varepsilon^{-1}(p)$, where $\Gamma_\varepsilon^{-1}(p) = 1 - (1 + \varepsilon p)^{(-\varepsilon p)}$, and p is uniformly chosen from (0,1). Finally, the reported location is $z = x + (r \cos\theta, r \sin\theta)$.

If only a restricted set of reported locations is allowed (e.g., discrete set $\mathcal{Z}$), the location produced by the mechanism can be mapped back to the closest element in the set. Although the Laplace mechanism provides an easy and practical way of achieving geo-indistinguishability, its utility may be low, as it introduces large noise.

Optimal Mechanism (OPT). While the PL mechanism is simple to implement and efficient to execute, it does not provide any optimality guarantees for utility. As a result, its resulting utility may be low in practice. To address this issue, the optimal mechanism of GeoInd (denoted further by *OPT*) has been introduced in [2]. We provide the details of OPT in Section 3.2. In a nutshell, OPT uses information about an adversary's prior knowledge, such as the likelihood of a user being present at a certain location. For instance, a user is more likely to be found in a city center area where there is a high concentration of POI, rather than in a secluded area near a suburb. OPT uses such prior information to produce a reported location z that minimizes utility loss. The disadvantage of OPT is that it has high computational overhead, as it requires solving a linear program with complexity cubic in the number of possible locations. In contrast, our multi-step algorithm allows us to reduce overhead by applying OPT recursively on an index structure.

A remarkable property of GeoInd related to the OPT mechanism is the following: even if the mechanism is tuned for a specific prior, it still preserves the privacy constraint for *any* prior. If the prior is not known by a GeoInd mechanism, then the utility cannot be improved significantly compared to PL. If the prior is known, then it can boost utility. Furthermore, GeoInd is satisfied even if the assumed prior and the adversary's background knowledge are different. In contrast, privacy models such as the one in [24] can

[2] We emphasize that a utility loss metric is a *different* concept than a distinguishability metric. Although the Euclidean distance can serve as both, the purpose of the two types of metrics is different: distinguishability metrics characterize the ability of an adversary to attack location privacy, whereas utility loss metrics measure the decrease in quality experienced by the user.

Table 1: Summary of Notations

Notation	Definition
x	*Actual* user location
z	*Reported* user location
$\mathcal{X}, \mathcal{Z}$	Set of actual and reported user locations
$\mathbb{G}, \mathbb{G}_i$	Set of grid cells (global, and at index level i)
$\hat{x}$	*Actual* location cell of user
$\hat{z}$	*Reported* location cell of user
g	Grid granularity
L	Side length of the square spatial region
h	Height of hierarchical grid index
ϵ_i	Privacy budget for grid level i
$\mathcal{B}$	Array storing budget at each index level
$d(x, z)$	Euclidean distance between locations x and z

only provide protection if the assumed prior and the adversary's knowledge coincide.

3 APPROACH OVERVIEW

In Section 3.1 we provide an overview of our system model; in Section 3.2 we introduce a baseline approach that applies the optimal mechanism of GeoInd on top of a regular grid. Table 1 summarizes the notations used in our presentation.

3.1 System Model

We consider an *online* scenario where mobile users are interested in retrieving points of interest relevant to their current location from an untrusted server. Based on the actual user location $x \in \mathcal{X}$, the sanitization algorithm computes a *reported* location $z \in \mathcal{Z}$. Both $\mathcal{X}$ and $\mathcal{Z}$ are assumed to be discrete and finite sets. Discrete locations, also called *logical* locations [12], are often used in the location privacy literature. In practice, this can be achieved by snapping continuous coordinates to an arbitrary granularity grid. We note that, in many existing geosocial network applications, the reported locations take the form of *check-ins* at discrete sets of two-dimensional coordinates corresponding to restaurants, coffee shops or other POIs. Our model is fully compatible with that setting.

In our model, the location sanitization is performed online at the user's mobile device. This approach is made possible by the properties of the GeoInd model, and it is an important advantage compared to other types of solutions that require a trusted centralized service, such as spatial k-anonymity (SKA) [16, 21]. Thus, the system model is much simpler, and it does not rely on unrealistic security assumptions, such as the presence of a trusted third party that performs anonymization, or the presence of a collaborating set of other mobile users who participate in the formation of cloaking regions. Furthermore, there are no changes required on the processing side at the server, which is an advantage compared to approaches that use encryption [15]. However, due to the fact that location protection is performed at the mobile device, the computational overhead incurred by sanitization is a very important concern. In our design, we focus on this constraint.

The mobile device will execute our sanitization technique before reporting its location (i.e., at runtime), and will also download in advance (*offline*) a set of objects that are required to support our technique, such as a set of maps annotated with additional pre-computed information (e.g., the properties of a certain city map and associated details required to construct a balanced index structure, as discussed in Section 5). This offline component is common for many software packages that support location-based apps, e.g., navigation services. Furthermore, in our approach, the amount of data that needs to be downloaded offline is small (in the order of tens of megabytes).

3.2 Baseline Approach

We provide a detailed description of the optimal GeoInd mechanism *OPT* [2] adapted to a regular grid. OPT is used as a building block in our multi-step approach, and also as a baseline in our evaluation. OPT produces the maximum utility achieveable under a given prior while preserving GeoInd. Specifically, given a privacy budget ε, a distinguishability metric $d_{\mathcal{X}}(\cdot,\cdot)$, a utility (or quality) metric $d_Q(\cdot,\cdot)$, and a prior Π defined over set $\mathcal{X}$, OPT determines mechanism K as the solution to the following linear programming problem:

Minimize:

$$\sum_{x \in \mathcal{X}, z \in \mathcal{Z}} \Pi_x \cdot K(x)(z) \cdot d_Q(x, z) \tag{3}$$

Subject to:

$$K(x)(z) \le e^{\epsilon d_{\mathcal{X}}(x,x')} \cdot K(x')(z) \qquad x, x', z \in \mathcal{X} \tag{4}$$

$$\sum_{z \in \mathcal{X}} K(x)(z) = 1 \qquad x \in \mathcal{X} \tag{5}$$

$$K(x)(z) \ge 0 \qquad x, z \in \mathcal{X} \tag{6}$$

This linear optimization problem can be solved by using standard techniques such as the Simplex or the Interior Point methods. However, the number of linear constraints in the program is $O(|\mathcal{X}|^2 \times |\mathcal{Z}|)$ (or cubic in the total number of locations, assuming that actual and reported locations belong to the same set). As a result, the method is unfeasible even when the set of locations has low cardinality (i.e., several hundred). In previous work [5] the cardinality of $\mathcal{X}$ is reduced by using a coarser grid, and locations in $\mathcal{X}$ are snapped to the centers of the grid cells. Let L denote the side length of the dataset (assumed to be a square, but any rectangular region can be scaled to suit the assumption). Grid granularity g is achieved by splitting the data domain into a regular grid with $g \times g$ cells, each with dimensions $L/g \times L/g$. While this approach reduces computational overhead, it also decreases utility, as all locations are snapped to the coarse grid.

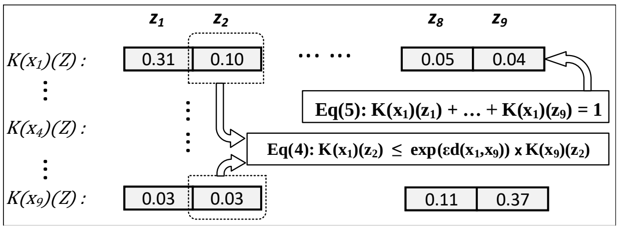

z1
z2
z8
z9
K(x1)(Z) :
0.31
0.10
0.05
0.04
Eq(5): K(x1)(z1) + … + K(x1)(z9) = 1
K(x4)(Z) :
Eq(4): K(x1)(z2) ≤ exp(εd(x1,x9)) x K(x9)(z2)
K(x9)(Z) :
0.03
0.03
0.11
0.37

Figure 2: An instance of $K(\mathcal{X})(\mathcal{Z})$ for a 3×3 grid.

Next, we focus on the application of OPT on top of a grid. Denote by $\mathbb{G}$ the set of grid cells, which represent the logical locations from which both actual and reported locations are selected, i.e., $\mathcal{X} = \mathcal{Z} = \mathbb{G}$. The prior is also defined with respect to $\mathbb{G}$. We can abstract an invocation of the OPT mechanism as a function call with the following parameters: $OPT(\epsilon, \mathbb{G}, \Pi, d_Q)$.

To provide an in-depth understanding of how OPT works, we consider the example of a regular grid of granularity $g = 3$, i.e., a total of nine cells. We illustrate in Figure 2 the layout of the stochastic matrix $K(\mathcal{X})(\mathcal{Z})$ from Eq.(4). We represent three of the nine rows of matrix K, corresponding to cells x_1, x_4 and x_9. The box at the top of the diagram illustrates the constraint in Eq. (5), representing the normalization condition, namely: the sum of probabilities in each matrix row must be 1 (each input cell must be mapped to some output cell). Likewise, the box in the middle of the diagram illustrates one of the $|\mathcal{X}|^3 = 81$ ε-geo-indistinguishability constraints corresponding to Eq. (4) in the linear program of the optimal mechanism.

The main drawback of OPT is its prohibitively high computational cost. To illustrate this drawback, we show in Figure 3 the trade-off between performance and utility when varying granularity, using a state-of-the-art commercial linear program solver on the Gowalla dataset (please see Section 6 for experimental setup details). As grid granularity increases, the utility improves but at the expense of a sharp rise in computation time. For the next higher granularity $g = 12$ (not shown in the graph), the optimization program was terminated after 24 hours without a solution.

The objective of our approach is to address these conflicting trends between utility and performance. Next, we present our *multi-step mechanism (MSM)* which operates on top of a hierarchical index structure and achieves a good compromise between utility and performance.

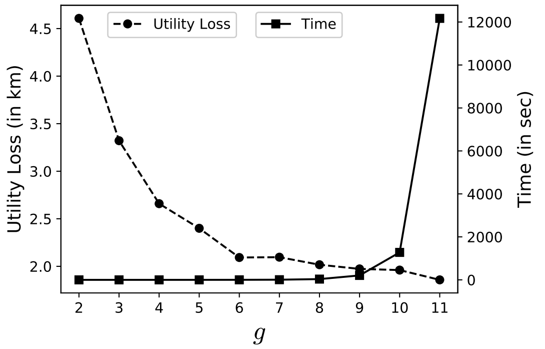


Figure 3: Effect of g on utility and running time of *OPT*

4 MULTI-STEP MECHANISM (MSM)

We introduce the multi-step mechanism (MSM) which operates over a *GeoInd-preserving Hierarchical Index (GIHI)* and transforms the user location according to OPT at each index level. The user inputs the total privacy budget ε to the algorithm, according to her privacy requirement. A separate component of our solution, the budget allocation strategy (discussed in detail in Section 5), determines how to distribute the privacy budget among the index levels. The output of the budget allocation procedure consists of the index height h, and the amount of privacy budget allocated to each level $B_i \in \mathcal{B}$, where $\mathcal{B}$ is the set of budget amounts per level, and $h = |\mathcal{B}|$. In the rest of the section, we focus on how the mechanism operates given the index height and budget splits.

Given granularity g and height parameter h, a GIHI is constructed over a data domain of size L^2 in a top-down fashion[3]. Each intermediate cell points to g^2 cells at the lower level that lie inside its spatial extent. Figure 4 illustrates the index structure[4] for $g = 2$ and $h = 3$. The cells in the hierarchical grid are labeled per level, and also within each level, as shown for level 1 in the diagram (to improve readability, we do not include the labels at lower levels). For instance, the leaf cell $C3_{3,0}$ is a child of $C2_{1,0}$, which in turn is a child of node $C1_{0,0}$. MSM iteratively visits each level of the hierarchical structure, starting with the virtual root node that covers the entire region. The root node is not a level in the tree, but it serves the purpose of simplifying the presentation, so that the algorithm can be expressed recursively starting from a single node. We denote as $\hat{x}_i$ the logical location of the user snapped to the center of the enclosing cell in the grid of granularity $g^i \times g^i$ at level i. Likewise, $\hat{z}_i$ denotes the logical location of the user output by the mechanism at level i. We define two simple procedures: *EnclosingCell*(x, i), which returns the cell at level i enclosing the input location x, and *centerOf*(C), which takes as input a cell C and returns the logical location at its center.

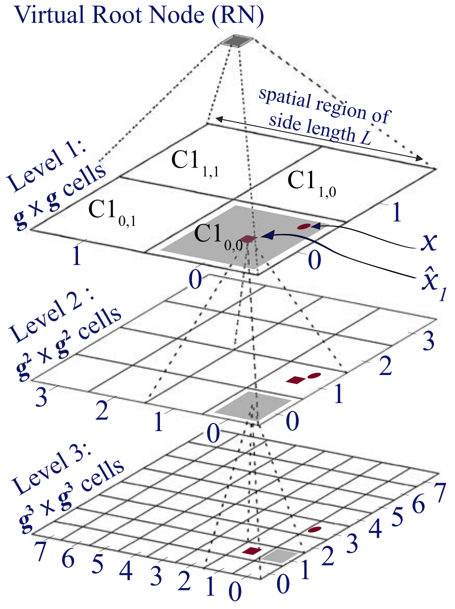


Figure 4: GeoInd preserving Hierarchical Index (GIHI)

Algorithm 1 presents the pseudo-code of our multi-step iterative process of location sanitization. In each iteration, the algorithm takes as input the location $\hat{z}_{i-1}$ selected in the previous iteration (in the first iteration it is set to the center of the virtual root node) and determines the cell that encloses $\hat{z}_{i-1}$, denoted by C (lines 4-6). Next, it translates the user's actual location x to a logical location $\hat{x}_i$ in the current level (lines 7-8) and constructs a partial grid $\mathbb{G}_i$ of g^2 cells within the spatial extent of C. Next (lines 11-13), MSM computes the matrix $K(\mathcal{X}_i)(\mathcal{Z}_i)$ as the solution to a linear optimization problem over the logical locations $\mathcal{X}_i, \mathcal{Z}_i \in \mathbb{G}_i$, consuming the privacy budget ε_i allocated to level i. MSM marks the end of the iteration at the current level by outputting $\hat{z}_i$ as a sample from the distribution $K(\hat{x}_i)(\mathcal{Z}_i)$. These iterations repeat for the entire height of the GIHI, consuming the entire privacy budget $\varepsilon = \varepsilon_1 + ... + \varepsilon_h$. After the final iteration (at the leaf level

[3] If the input dataset domain is not square, it can be scaled in advance of executing our algorithm to equalize the range in each dimension.

[4] For simplicity, we focus in this paper on an hierarchical grid, but the MSM concept applies to any hierarchical data structure without node overlap, e.g., R^+-trees or k-d-trees.

of the index), the output is the sanitized location that is reported to the service provider (line 14).

Note that, the actual location x may fall outside the selected cell at one or more levels of the index. This is an effect of the privacy requirement imposed by GeoInd, and thus a necessary aspect of protecting locations. However, it may also increase utility loss, as the distance between reported and actual cell may be large. To control the amount of utility loss, the objective of our budget allocation strategy (Section 5) is to split the budget in such a way that it is less likely for this event to occur at the higher (i.e., coarser) index levels. If this objective is achieved, utility will not decrease significantly.

Algorithm 1 Multi-Step Mechanism (MSM)

```
    Input: $\epsilon$, $g$, $\Pi$, $d_Q$
    Output: $\hat{z}$
 1: procedure PRIVATELYREPORTLOCATION($x$)
 2:     $\mathcal{B} \leftarrow$ getGridParameters($\epsilon$, $g$)
 3:     $h \leftarrow |\mathcal{B}|$
 4:     $\hat{z}_0 \leftarrow RootNode$          ▷ stores output cell of each iteration
 5:     for $i \leftarrow 1, h$ do
 6:         $C \leftarrow EnclosingCell(\hat{z}_{i-1}, i-1)$
 7:         $\mathbb{G}_i \leftarrow$ set of $g \times g$ cells in the spatial bounds of $C$
 8:         $\hat{x_i} = centerOf\,(EnclosingCell(x,i))$
 9:         if $\hat{x_i} \notin \mathcal{X}_i$ then                 ▷ $\mathcal{X}_i, \mathcal{Z}_i \in \mathbb{G}_i$
10:             $\hat{x_i} \leftarrow$ a random location in $\mathcal{X}_i$
11:         $\epsilon_i \leftarrow \mathcal{B}[i]$
12:         $K(\mathcal{X}_i)(\mathcal{Z}_i) \leftarrow OPT(\epsilon_i, \mathbb{G}_i, \Pi(\mathcal{X}_i), d_Q(.))$
13:         $\hat{z}_i \leftarrow$ sample from distribution $K(\hat{x}_i)(\mathcal{Z}_i)$
14:     return location $\hat{z}$ to the service provider
```

We illustrate the proposed mechanism with a running example over the GIHI structure from Figure 4. The exact location of the user x is depicted as a red-colored dot, while the logical locations (snapped to cell centers) $\hat{x}$ are shown as red colored squares. The shaded cells depict the cells enclosing the output location ($\hat{z}$) of each iteration. The algorithm first takes as input the entire dataspace domain (represented as the root node), and splits it into a grid $\mathbb{G}_1$ consisting of cells $\{C1_{i,j} : 0 \leq i, j \leq 1 \wedge C1_{i,j} \in RN\}$. Next, it maps the user's exact location x to $\hat{x}_1$ at the center of its enclosing cell at level 1, i.e., $C1_{0,0}$. Then, MSM computes $K(\mathcal{X}_1)(\mathcal{Z}_1)$, and outputs a location $\hat{z_1}$ sampled from the distribution $K(\hat{x}_1)(\mathcal{Z}_1)$.

Denote the selected cell as $\hat{z_1}$ = *centerOf*($C1_{0,0}$). In the next iteration, $\mathbb{G}_2$ is composed of the set of $g \times g$ cells within the spatial extent of $C1_{0,0}$, which is the enclosing cell of the output in the previous iteration. Location $\hat{z_2}$ is selected as output. Suppose $\hat{z_2}$ =*centerOf*($C2_{0,0}$), which implies that for the purposes of the last iteration, the actual location x falls outside the spatial extent of the cell enclosing $\hat{z_2}$. In this situation the user's location is assumed to be a random cell (e.g., $C3_{1,1}$) in $\{C3_{i,j} : 0 \leq i, j \leq 1\}$. The remainder part of the iteration continues by outputting $\hat{z_3}$ =*centerOf*($C3_{1,0}$). Finally, $\hat{z_3}$ is reported to the service provider, with a quality loss $d_Q(x, \hat{z_3})$.

A consequence of the grid based discretization scheme is that, the position of every user is always approximated by the center of the enclosing cell before being obfuscated by the mechanism. For example, consider that a user with his exact coordinates at a random location in a 1km^2 cell, requests the nearest bar to his position. In this case, he will receive an answer that is tailored, in the best case, to the center of his cell, which is on average 0.38km [14] away from the current location of the user. It is clear that the situation gets more problematic as the grid cells are larger, i.e., at coarser granularities. This is often the case when applying the conventional optimal mechanism [2] over a coarse grid (as discussed in Section 3.2, OPT can work only for very coarse grids, otherwise the execution time is extremely high). In contrast, our approach operates on a much finer-grained grid at the leaf level, thus gaining significantly in terms of utility. MSM also limits the size of the linear optimization problem, given that there are exactly g^2 logical locations at each step. This enables efficient computation of the linear program, making the overall execution time practical.

We end this section by informally discussing the fact that MSM preserves GeoInd. MSM is a textbook example of applying the *composability* property [19, 20] of differential privacy. The initial iteration of MSM takes as input the real user location, and the final iteration outputs the perturbed location. At each index level i, the OPT mechanism is applied with a fraction ε_i of the total privacy budget ε. The output of each MSM step (i.e., level) is pipelined into the input of the following step (at the next level). According to the composability property introduced in Section 2, MSM satisfies GeoInd with budget $\sum_{i=1}^{h} \varepsilon_i = \varepsilon$.

5 BUDGET ALLOCATION STRATEGY

The utility of MSM depends on important system parameters such as the height of the index and the budget allocation across levels. In this section, we provide an analytical cost model of utility that guides our decision on how to choose GIHI parameters. We assume we are given as input the size of the data domain specified as side length L, the grid granularity g (corresponding to a fanout of g^2 at each level), and the total budget ε. The objective is to determine index height h and the budget allocation ε_i for each level $i, 1 \leq i \leq h$.

A fundamental factor that guides our allocation strategy is the observation that if the actual location is mapped to a reported location in another cell, the utility loss is likely to be larger when the grid cell size is also large. Consequently, the utility loss is much larger when this event occurs near the root of the index, compared to the case when it occurs at the leaf level. Intuitively, the impact on utility of the event occurring at level i is g times higher than the same event occurring at level $i+1$. Consider a stochastic obfuscation mechanism that satisfies geo-indistinguishability, and denote its probability of mapping location (i.e., cell) x to z by $Pr[z|x]$. The probability density function of the output is indeed the distribution $K(x)(z)$, as discussed in Section 2. The proposed budget allocation strategy aims to precisely control $Pr[x|x]$, i.e. the probability of reporting cell x when the actual location is also within x. Based on the earlier observation, to reduce utility loss it is important to maintain $Pr[x|x]$ high at the upper levels of the index.

To calculate $Pr[x|x]$ precisely, we can solve the $|\mathcal{X}|^3$ constraints in the linear optimization program of Eq. (3), according to the input prior Π. However, estimating $Pr[x|x]$ in isolation poses a challenge, since it is not feasible to narrow down the effect on a single location of all other cells (as per Eq. (4)). Hence, we devise a method to estimate this value to an arbitrary precision. We denote the approximation of $Pr[x|x]$ as $\Phi(x)$, and compute it by estimating its complement, i.e., the probability of mapping x to another grid cell.

$$\Phi(x) = \frac{1}{\sum_n \exp\left(-\frac{\varepsilon L}{g}\sqrt{n}\right)} \qquad n = a^2 + b^2, (a,b) \in \mathbb{Z}^2 \tag{7}$$

where $\mathbb{Z}^2$ denotes the 2-dimensional infinite integer lattice whose points are tuples of integers. Switching to coordinates with origin at zero, and defining $\hat{0} \triangleq x$, the denominator can be understood to be the sum over the exponents of the *multiplicative distances* between $Pr[\hat{0}|\hat{0}]$ and $Pr[\hat{0}|\hat{n}]$. $\Phi(x)$ is constrained to the integer lattice to ensure that the probability mass of the entire cell is assigned to the center of the cell, giving us a close estimate of our desired probability.

While $\Phi(x)$ needs to be summed over the infinite lattice, the function $T(\varepsilon, g) \triangleq \sum_n \exp(-\varepsilon L/g\sqrt{n})$ converges quickly. In order to approximate this value within a factor of $\exp(-N)$ for any arbitrarily large N, we need only $O(N^2 f/\varepsilon L)$ terms. However, when the value of ε is small, which is often the case in differential privacy literature (with values of $\varepsilon \leq 0.5$ being the most common settings to achieve a reasonable privacy protection), straightforward computation of this value can be prohibitive. To this end, we can apply the two-dimensional Poisson summation to expand the series, followed by taking its Fourier transform, to obtain an efficient approximation when $0 \leq \varepsilon \leq \frac{2\pi g}{L}$:

$$T(\epsilon, g) = \frac{2\pi g^2}{\varepsilon^2 L^2} + \sum_{k=1}^{\infty} c_{2k-1}\left(\frac{\varepsilon L}{g}\right)^{2k-1} \tag{8}$$

where the coefficients $c_1, c_3, c_5, \ldots$ are given by

$$c_{2k-1} = 4\binom{-3/2}{k-1}(2\pi)^{-2k}\zeta\left(k+\frac{1}{2}\right)L\left(k+\frac{1}{2}, \chi_4\right). \tag{9}$$

where ζ is the Riemann zeta function [23] and $L(\cdot, \chi_4)$ is the Dirichlet L-series[25] of the non-principal Dirichlet character χ_4 (mod 4), which is known to converge absolutely as

$$L(s, \chi_4) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)^s} = 1 - \frac{1}{3^s} + \frac{1}{5^s} - \frac{1}{7^s} + - \cdots. \tag{10}$$

Since we can approximate $T(\varepsilon, g)$ efficiently, the same is true of $\Phi(x)$. In order to configure the budget at each level of the grid, we formulate our problem as follows:

PROBLEM 1. *Given a desired value of $Pr[x|x]$ as ρ, and the grid parameters: side length L, and the granularity g, estimate the minimum budget ε_i that ensures at least ρ percent chance of remaining within the boundaries of the current cell at level i. The budget ε_i can be calculated according to the following optimization problem:*

Minimize:

$$\text{Privacy Budget } \varepsilon \tag{11}$$

Subject to:

$$\begin{gathered} \left(\sum_n \exp\left(-\frac{\varepsilon L}{g}\sqrt{n}\right)\right)^{-1} - \rho \geq 0, \\ \varepsilon > 0, \\ 0 \leq \rho \leq 1 \end{gathered} \tag{12}$$

Since $T(\varepsilon, g)$ can not be written in a closed form, solving directly for ε is not achievable. Nevertheless, the expression in Eq. (12) is monotonic in ε, hence we can utilize a simple branch-and-bound technique [10] to efficiently get an answer with arbitrary precision.

Algorithm 2 summarizes the functionality of the proposed budget allocation strategy. Procedure *getGridParameters* calculates the minimum budget required for each level of the grid, such that with probability at least ρ a location enclosed within cell x is mapped to the same grid cell. For every iteration of the while loop, the procedure determines if there is any budget remaining for additional levels of the multi-step mechanism. Finally, the process halts when it expends all the budget, returning the height h and the budget allocated at each level of the grid.

Algorithm 2 Grid Configuration

```
Input: $\varepsilon, g$
Output: height $h$, budgets $\varepsilon_1, \varepsilon_2, \cdots, \varepsilon_h$
procedure GETGRIDPARAMETERS
    $v \leftarrow \varepsilon$                                    ▷ stores the remaining budget
    $i \leftarrow 1$                                    ▷ denotes the current grid level
    $\mathcal{B} \leftarrow \emptyset$                                    ▷ stores the budget for each level
    while true do
        $\mathcal{B}[i] = \varepsilon_i \leftarrow \max\{\text{solution to Problem 1}, v\}$
        $v = v - \varepsilon_i$
        $i = i + 1$
        if $v \leq 0$ then                                    ▷ budget expended
            return $\mathcal{B}$
```

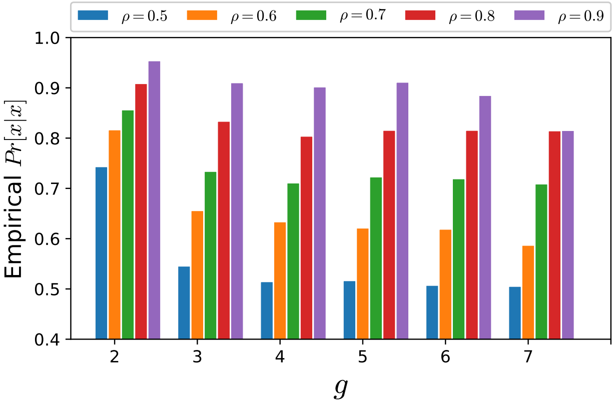


Figure 5: Accuracy of estimated Φ for varying g

We conclude this section by providing a numerical result to validate our analytical model. We illustrate the precision with which our budget allocation algorithm estimates an effective configuration of the grid for MSM (we defer the details of the experimental setup to Section 6.1). We plot the probability $Pr[x|x]$ for various levels of ρ and granularity g using the Gowalla dataset. Figure 5 shows the results, assuming a uniform global prior as input. Excluding the case when the granularity is 2, the predicted value of Φ is within ±5 percent of $Pr[x|x]$ reported in $K(\mathcal{X}, \mathcal{Z})$, thus validating the efficacy of the budget allocation scheme.

6 EXPERIMENTAL EVALUATION

In this section we evaluate the performance of our proposed Multi-Step Mechanism (MSM) in terms of utility and computational overhead. In Section 6.1 we present the details of the experimental setup. We provide the results of comparison with benchmarks in Section 6.2, followed by an in-depth analysis of MSM behavior when varying system parameters in Section 6.3.

6.1 Experimental Setup

Datasets. We use two real datasets collected from the operation of two prominent geo-social apps, namely *Gowalla* and *Yelp*. Each dataset consists of a set of user check-ins. Every check-in is described by a record consisting of user identifier, the latitude and longitude of the check-in location. The *Gowalla* dataset is a subset of the user check-ins collected by the SNAP project [9] from a location based social networking website. In our experiments, to simulate a realistic environment of a city and its suburbs, we focus on check-ins within a single urban area, namely Austin, Texas. In particular, we consider a large geographical region covering a $20 \times 20\text{km}^2$ area bounded to the South and North by latitudes 30.1927 and 30.3723, and to the West and East by longitudes -97.8698 and -97.6618. The selected data contains a total of $265,571$ check-ins from $12,155$ unique users during a time period between February 2009 and October 2010. The *Yelp* dataset was made available to the public by Yelp, Inc [3] as part of a dataset challenge, and contains user ratings for various points of interest. Again, to simulate a typical urban area, we utilize location data within the city of Las Vegas, NV and its surroundings. The filtered dataset contains $81,201$ check-ins from $7,581$ unique users within a $20 \times 20\text{km}^2$ area bounded between latitudes 36.0645, 36.2442 and longitudes -115.291, -115.069.

Implementation. All algorithms were implemented in C++ on a Ubuntu Linux 14.04 LTS operating system, and executed on an Intel Core 2 Duo 2.0 GHz CPU with 4GB RAM. All data and indices are stored in main memory. For the computation of the linear optimization problem, we utilize the C++ interface of the state-of-the-art commercial linear program solver in the Gurobi Optimization Suite [22]. We used the dual simplex method of solving linear programs throughout our evaluation since it consistently outperformed the primal simplex and interior-point methods in terms of numerical stability.

Prior Modeling. We compute a *global prior* for the Gowalla and Yelp dataset partitions with the help of a regular grid superimposed on top of the data domains. The grid granularity varies to match the grid structure used in each specific experiment. For a given granularity, we count the number of check-ins in every cell relative to total number of check-ins in the entire grid (a similar approach was taken in [2, 5]). The global prior Π describes the behavior of an average user (as a vector of probabilities for each grid cell), and is used in the computation of the optimal mechanism. We store a global prior on the finest effective granularity grid used in the experiments and aggregate this information to obtain priors on coarser grids. This procedure mimics the scenario where the aggregate information about past users of the service is made available from the service provider.

6.2 Comparison with Benchmarks

We evaluate our proposed MSM approach in comparison with two benchmarks: the basic optimal mechanism *OPT* introduced in [2] and described in detail in Section 3.2; and the planar Laplace (PL) mechanism, for which we also include a post-processing (or re-mapping) step by projecting its output to the grid (as discussed in [5]).

All evaluated mechanisms are constructed to satisfy GeoInd with privacy budget ε ranging from 0.1 to 0.9, which is a common range used in the majority of differential privacy work [11]. Recall from Section 2 that a smaller ε value corresponds to stronger privacy. Values higher than 1.0 are typically considered insufficient in terms of privacy (i.e., the attacker's gain in distinguishing among candidate locations becomes significant). We consider several granularities of the grid index structure, ranging from $g = 2$ up to $g = 6$ (recall that for our method, the effective fanout is $g \times g$ at each level). We consider a value range for the probability of hashing the user location in the same cell at a certain level (see Section 5 for details) between $\rho = 0.5$ and $\rho = 0.9$ (default value is set to $\rho = 0.8$). In all experiments, we measure the utility loss experienced by a user of a location-based service over a set of $3,000$ requests randomly selected from the set of check-ins corresponding to each dataset. We consider in turn both Euclidean ($d_Q = d$) and squared Euclidean ($d_Q = d^2$) utility metrics. The default value of ε is set to 0.5.

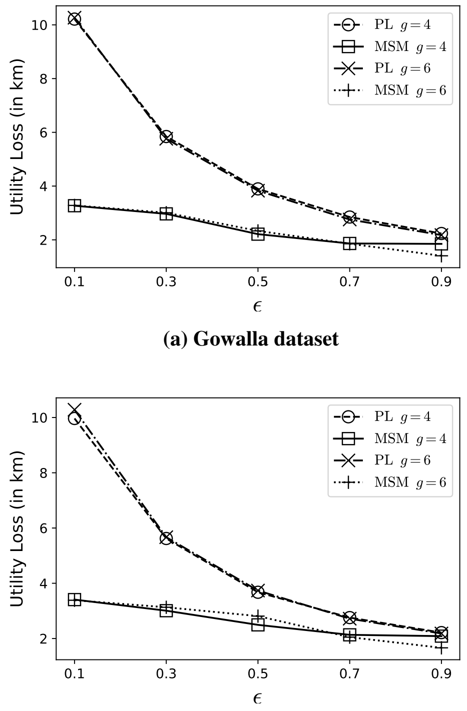
Table 2: MSM comparison against *OPT*, Gowalla dataset

Granularity		Utility Loss (in km)		Time (in sec)	
OPT	MSM	*OPT*	MSM	*OPT*	MSM
4	2	2.29	2.63	0.04	0.008
9	3	1.97	2.22	205.7	0.009
16	4	—	2.02	72hrs+	0.53

First, we compare the proposed MSM approach with the optimal mechanism OPT. Due to the high overhead of OPT, we are only able to perform the comparison in a restricted setting, for which the linear program completes within reasonable time. Table 2 presents the utility loss and execution time values of OPT and MSM for the same effective grid granularity (e.g., 9×9 granularity for *OPT* corresponds to a 3×3 granularity for our method, where the second level of MSM will have the same number of cells as the OPT grid). We focus on the Gowalla dataset for this experiment. We were not able evaluate the performance of OPT for the granularity of 16 (i.e., 256 locations) because the program did not complete within 72 hours. For 121 locations (i.e., 11×11 grid) OPT took 3.3 hours to complete (see Figure 3). Even for a granularity of 9, the optimal method would be completely unfeasible for online queries, given its running time of 205 seconds, whereas our approach achieves sub-second processing times in all runs. In terms of utility, for the same granularity at the leaf level OPT does outperform MSM. This result is not surprising, since our execution time gain is obtained by pruning the search space. However, the additional loss of MSM compared to OPT is not large. Furthermore, in practice, our approach is able to increase utility by using much finer-grained grids (as we show in the next section), whereas OPT can only function for coarse-grained grids. In the rest of our evaluation, we no longer consider OPT, since it is clearly not feasible in practical settings.

Next, we focus on the comparison between MSM and the planar Laplace mechanism (PL). Figures 6a and 6b plot the utility loss of MSM and PL for varying levels of privacy on both Gowalla and Yelp datasets, using d as utility metric. As expected, utility loss decreases for both mechanisms as ε grows: a larger budget corresponds to a weaker privacy requirement, hence less noise is added. The proposed MSM approach clearly outperforms PL in terms of utility, especially at low ε values, which are more important because they provide appropriate privacy. As ε approaches 1, the utility obtained by the two methods becomes similar, but as mentioned earlier, $\varepsilon = 1$ or higher does not provide sufficient protection. The gain of our approach over PL is more pronounced at low ε settings. For $\varepsilon = 0.1$, MSM utility loss is three times better than the one achieved by PL.

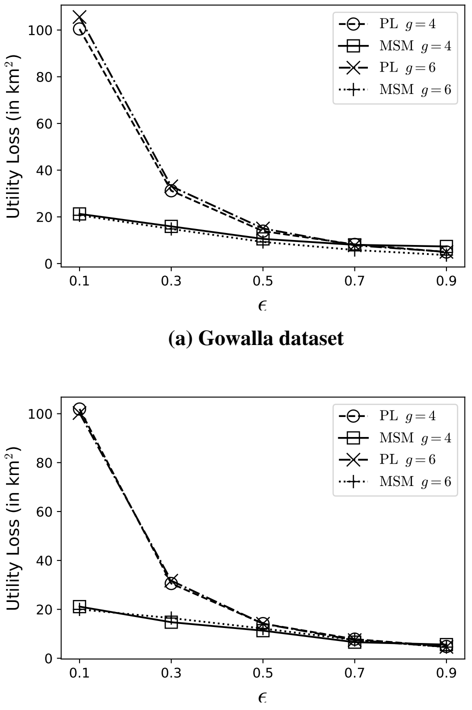
A similar trend can be observed when comparing MSM and PL using the squared Euclidean distance as utilty metric (Figures 7a



(a) Gowalla dataset

(b) Yelp dataset

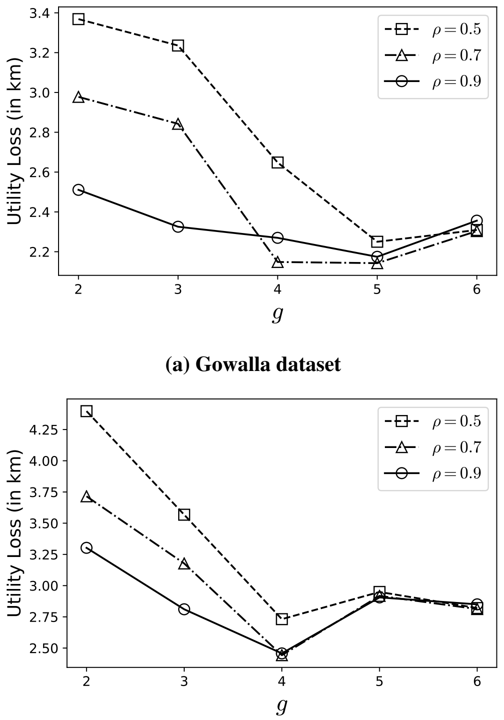
Figure 6: Effect of ϵ on utility loss (Euclidean utility metric).



(a) Gowalla dataset

(b) Yelp dataset

Figure 7: Effect of ϵ on utility loss (squared Euclidean utility metric).



(a) Gowalla dataset

(b) Yelp Dataset

Figure 8: Effect of varying granularity on the utility loss (Euclidean utility metric).

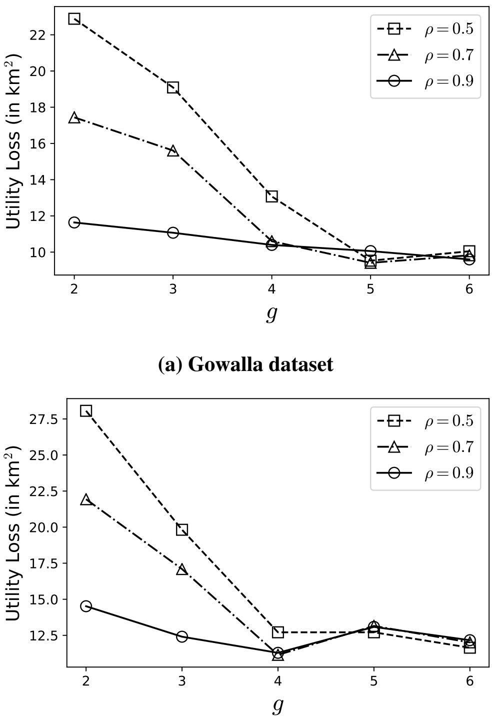
and 7b). This time, the gap between the two approaches is even larger, with MSM outperfoming PL by a factor of 5 at the low end of the privacy budget range. PL does however catch up with our method earlier than in the case of Euclidean utility metric (around the $\varepsilon = 0.5$ threshold). Still, MSM remains clearly superior in the range of tight privacy settings.

Our results show that MSM outperforms significantly the PL benchmark in terms of utility loss. As discussed in Section 2, PL is very fast in terms of execution time, and it takes on average 10 milliseconds to complete. In contrast, our method is more expensive. Still, the execution time is always below 1 second in the worst case (which occurred for the highest considered granularity case $g = 6$ and a large ε value). In most cases, the average runtime of our method is in the range of $100 - 200$ milliseconds. Even though this is higher than PL, $100 - 200$ milliseconds per query is a small price to pay in order to increase utility significantly. Furthermore, recall that this cost will be incurred at the client side, so there is no danger of the server being overloaded due to a spike in request. In terms of user experience, the additional time required for sanitizing locations is barely noticeable.

For the rest of the evaluation, we no longer consider PL, and we focus on analyzing the performance of the proposed MSM method when varying its system parameters values.

6.3 MSM System Parameter Analysis

Next, we evaluate the utility of MSM when varying grid granularity, while also considering several settings of ρ (shown as distinct lines in each graph). Recall that, for MSM the fanout at each level is $g \times g$, hence the first level (below the single virtual root node) has granularity $g \times g$, the second level granularity $g^2 \times g^2$, and



(a) Gowalla dataset

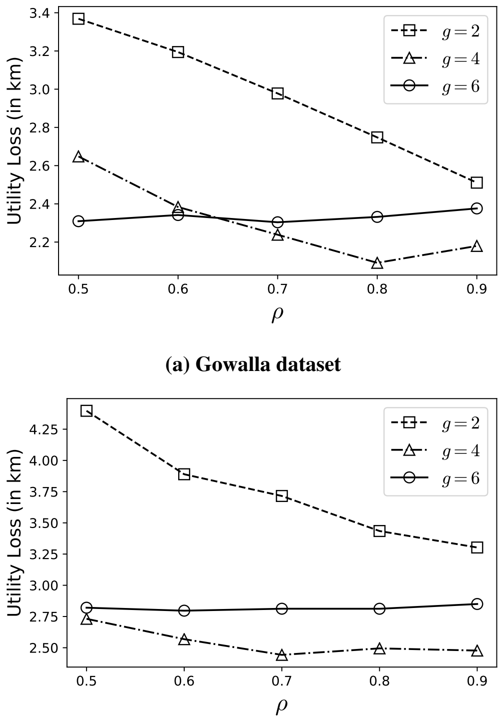
(b) Yelp Dataset

Figure 9: Effect of varying granularity on the utility loss (squared Euclidean utility metric).

so on. Figures 8a and 8b show the obtained results for the two datasets under Euclidean distance utility metric. The general trend is that of a “U”-shaped dependency: utility loss decreases initially as granularity increases, which intuitively is a result of a higher precision in reporting locations. However, after a certain point, the utility loss starts to increase, as a high granularity will determine more cases where the reported and actual locations are in different cells. As a side effect, as the area of each grid cell decreases, more budget is required at a certain level to maintain the same required probability ρ, which can starve lower index levels of privacy budget. We notice that the ideal granularity may also vary with the dataset, as data density typically affects accuracy. For the Gowalla dataset the best-performing granularity is at $g = 5$, whereas for the Yelp dataset it is $g = 4$. For a given granularity, we note that a higher value of ρ typically results in better utility, although there are some exceptions: for instance, at $g = 4$, the $\rho = 0.7$ case outperforms the $\rho = 0.9$ setting, for the reason explained above, namely the top level uses most of the budget, and the lower levels do not receive sufficient budget to accurately execute the linear program.

A similar trend is observed when using squared Euclidean distance as utility metric, as shown in Figures 9a and 9b. A higher ρ results in better utility in the majority of cases.

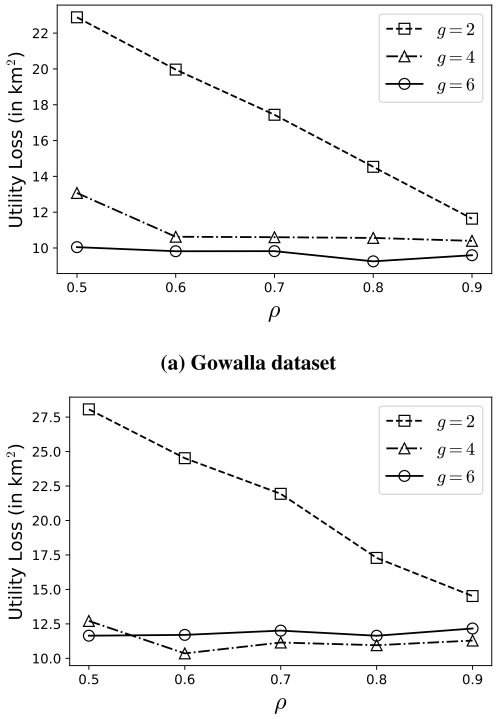
Finally, we measure the effect of varying parameter ρ, with several distinct settings of granularity (shown as different lines in each graph). Figures 10a and 10b summarize the results for Euclidean distance, whereas results for the squared Euclidean distance are shown in Figures 11a and 11b. For the lowest granularity $g = 2$, we note a clear decreasing trend in utility loss. As the grid



(a) Gowalla dataset

(b) Yelp Dataset

Figure 10: Effect of varying probability ρ on the utility loss (Euclidean utility metric).



(a) Gowalla dataset

(b) Yelp Dataset

Figure 11: Effect of varying probability ρ on the utility loss (squared Euclidean utility metric).

granularity grows gradually from one level to another, the algorithm for budget allocation is able to allocate budget in a smoother fashion, leading to steady progress as ρ grows (we emphasize that although the trend is more pronounced, the absolute value of utility is worse for the $g = 2$ setting compared to the rest). For the other settings of g, due to the fact that the transition from one level to the other is more abrupt, the net effect of ρ exhibits a not-so-well defined trend. The $g = 4$ case still shows a decreasing trend initially, but then utility loss starts to increase as ρ grows, as a result of budget starvation at lower index levels. In the case of $g = 6$, the budget starvation seems to manifest even at lower ρ values, so the trend is constant to slightly increasing. Note that, starvation is not necessarily a negative effect, in the sense that the utility can still be better than the other settings. Starving lower levels of budget may be a worthwhile choice, as long as the higher levels keep the reported cell close to the actual cell, and in effect the utility loss is low. We insist on the starvation concept mostly to describe the observed trends. We do, however, note that excessive starvation can sometime increase utility loss: in the case of the Yelp dataset for instance, we observe that the utility loss obtained at a finer granularity ($g = 6$) is higher than that obtained with $g = 4$.

7 RELATED WORK

In the past decade, a vast amount of research focused on preserving location privacy. The earliest approaches used *dummy generation* [18, 27] to protect locations of users who issue location-based queries. In [18], the user issues a number of redundant (fake) queries at random locations, thus decreasing the probability that an adversary guesses which is the real location. The work in [27] chooses an *anchor* around the real location, and focuses on how query processing can be done around the anchor in such a way that precise results are obtained with respect to the real location (e.g., exact nearest-neighbor queries). Both approaches are vulnerable to background-knowledge attacks: an adversary who knows the map features or the patterns of user movement can filter out the fake locations and reveal the real one.

The *spatial k-anonymity (SKA)* concept has been introduced in [16] to address the limitations of dummy generation. The main idea behind SKA is to construct a *cloaking region (CR)* that encloses at least *k real* users. This way, it is more difficult for the adversary to filter out locations and narrow down the query source. The work in [21] showed how SKA can be implemented on top of a spatial index, but may be subject to reverse engineering attacks because the CR generation algorithm is deterministic and takes as seed the location of the querying user. Later on, [17] introduced the *reciprocity* property, which shows that as long as the same CR is generated for all k users in an anonymity set, the adversary's probability of identifying the user issuing a query is bounded above by $1/k$. However, all SKA approaches no longer provide protection when users move, or when some users in the anonymity sets disconnect. In addition, they may reveal the exact user locations: for instance, if k users are situated inside a hospital, it is possible for the CR to be completely enclosed by the hospital area. Even if the attacker cannot identify which user issued the query, s/he learns that all users are in a hospital, which is a serious privacy breach. Some later work considered *spatial diversity* [12, 26], which attempted to enlarge cloaking regions such that association with sensitive features (e.g., hospitals, nightclubs) is reduced. However, the limitations of SKA remain in place for spatial diversity when users move or disconnect.

The introduction of the novel semantic model of *differential privacy (DP)* changed the landscape of location privacy approaches. DP is a statistical model designed to address the scenario of *aggregate queries*, where the presence of an individual in a dataset must be hidden. Therefore, it is not directly applicable in the context of online location reporting, which is our problem setting. However, DP can be used in the context of publishing historical datasets of locations or trajectories. For instance, the work in [11] shows how spatial indexes can be used to release geospatial datasets at different granularities. The authors consider quadtrees and *k-d*-trees, and propose a cost model to help allocate budget across different structure levels. While we also consider multiple index levels, our context is a completely different one (that of geo-indistinguishability). Furthermore, their model shows that it is best to allocate smaller budgets towards the root of the index, and preserve a larger proportion of the budget for the leaf level, whereas our findings for the GeoInd setting are exactly the opposite. Due to the fact that an error at the root of the structure has more impact on data utility, we found that it is important to allocate a higher proportion of the budget at the higher index structure levels. Some other work in the context of DP [7, 8] focuses on releasing trajectories using noisy counts of prefixes or n-grams in a trajectory, but similar to [11], the results apply to the offline setting only.

Geo-indistinguishability [1, 2, 5] is a novel and promising semantic model that inherits the powerful guarantees of DP, but adapts them to the setting of online location protection. Since its introduction in 2013, several research efforts focused on finding efficient and utility-preserving techniques for implementing GeoInd. Closest to our work is the research in [5], which proposed several mechanisms and post-processing algorithms to improve utility. However, as we show in our extensive performance evaluation, those techniques are either fast and inaccurate (planar Laplace in conjunction with remapping to grid), or they only work for very small sets of candidate locations (as is the case with the optimal mechanism). In contrast, our multiple-step approach is able to achieve high utility with very reasonable computational overhead.

8 CONCLUSION

We proposed a multiple-step algorithm for protecting location privacy according to the geo-indistinguishability model. By using a multi-level index structure, we are able to prune the solution search space of expensive GeoInd mechanisms, and achieve high utility with low performance overhead. We also derived cost models that show how to judiciously allocate the available privacy budget, and how to choose important index parameters to improve accuracy. The resulting approach outperforms significantly existing benchmarks. In future work, we plan to investigate more advanced cost models to better capture prior information, and thus further improve the accuracy of our approach. We will also investigate more complex index structures (e.g., *k-d*-trees and R^+ trees) which can adjust better to skewed distributions of priors.

Acknowledgements. The authors thank Noam D. Elkies for his guidance with the computation of the infinite series of exponents in Section 5. This work has been supported in part by NSF grants IIS-1320149 and CNS-1461963, the USC Integrated Media Systems Center, and unrestricted cash gifts from Oracle. The opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors such as NSF.

REFERENCES

[1] M. E. Andrés, N. E. Bordenabe, K. Chatzikokolakis, and C. Palamidessi. Geo-indistinguishability: Differential privacy for location-based systems. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 901–914, 2013.

[2] N. E. Bordenabe, K. Chatzikokolakis, and C. Palamidessi. Optimal geo-indistinguishable mechanisms for location privacy. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 251–262, 2014.

[3] Y. D. Challenge. Yelp dataset challenge, 2019.

[4] K. Chatzikokolakis, M. E. Andrés, N. E. Bordenabe, and C. Palamidessi. Broadening the scope of differential privacy using metrics. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 82–102. Springer, 2013.

[5] K. Chatzikokolakis, E. ElSalamouny, and C. Palamidessi. Efficient utility improvement for location privacy. In *Proceedings on Privacy Enhancing Technologies (PoPETS)*, pages 308–328, 2017.

[6] K. Chatzikokolakis, C. Palamidessi, and M. Stronati. Constructing elastic distinguishability metrics for location privacy. In *Proceedings on Privacy Enhancing Technologies (PoPETS)*, pages 156–170, 2015.

[7] R. Chen, G. Acs, and C. Castelluccia. Differentially private sequential data publication via variable-length n-grams. In *ACM CCS*, pages 638–649, 2012.

[8] R. Chen, B. C. Fung, B. C. Desai, and N. M. Sossou. Differentially private transit data publication: A case study on the montreal transportation system. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 213–221, 2012.

[9] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1082–1090, 2011.

[10] J. Clausen. Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen*, pages 1–30, 1999.

[11] G. Cormode, C. Procopiuc, E. Shen, D. Srivastava, and T. Yu. Differentially private spatial decompositions. In *ICDE*, pages 20–31, 2012.

[12] M. L. Damiani, E. Bertino, and C. Silvestri. The probe framework for the personalized cloaking of private locations. *Trans. Data Privacy*, 3(2):123–148, Aug. 2010.

[13] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. of Theory of Cryptography Conference (TCC)*, pages 265–284, 2006.

[14] S. R. Finch. *Mathematical constants*, volume 93. Cambridge university press, 2003.

[15] G. Ghinita, P. Kalnis, A. Khoshgozaran, C. Shahabi, and K. L. Tan. Private Queries in Location Based Services: Anonymizers are not Necessary. In *Proceedings of International Conference on Management of Data (ACM SIGMOD)*, 2008.

[16] M. Gruteser and D. Grunwald. Anonymous Usage of Location-Based Services Through Spatial and Temporal Cloaking. In *Proc. of USENIX MobiSys*, 2003.

[17] P. Kalnis, G. Ghinita, K. Mouratidis, and D. Papadias. Preserving Location-based Identity Inference in Anonymous Spatial Queries. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19(12), 2007.

[18] H. Kido, Y. Yanagisawa, and T. Satoh. An Anonymous Communication Technique using Dummies for Location-based Services. In *IEEE International Conference on Pervasive Services (ICPS)*, 2005.

[19] F. McSherry and I. Mironov. Differentially private recommender systems: Building privacy into the netflix prize contenders. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 627–636. ACM, 2009.

[20] F. D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 19–30, 2009.

[21] M. F. Mokbel, C.-Y. Chow, and W. G. Aref. The New Casper: Query Processing for Location Services without Compromising Privacy. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2006.

[22] G. Optimization. Inc., gurobi optimizer reference manual. *URL: http://www. gurobi. com*, 2017.

[23] B. Riemann. about the number of primes under a given size. *Ges. Math. Works and Scientific Nachla ss*, 2:145–155, 1859.

[24] R. Shokri, G. Theodorakopoulos, C. Troncoso, J.-P. Hubaux, and J.-Y. Le Boudec. Protecting location privacy: Optimal strategy against localization attacks. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 617–627, 2012.

[25] E. W. Weisstein. Dirichlet l-series. *Wolfram Research, Inc.*, 2003.

[26] M. Xue, P. Kalnis, and H. K. Pung. Location diversity: Enhanced privacy protection in location based services. In *Proceedings of the 4th International Symposium on Location and Context Awareness*, pages 70–87, 2009.

[27] M. L. Yiu, C. Jensen, X. Huang, and H. Lu. SpaceTwist: Managing the Trade-Offs Among Location Privacy, Query Performance, and Query Accuracy in Mobile Services. In *International Conference on Data Engineering (ICDE)*, pages 366–375, 2008.

Inves: Incremental Partitioning-Based Verification for Graph Similarity Search

Jongik Kim
Chonbuk National University
Jeonju, Republic of Korea
jongik@jbnu.ac.kr

Dong-Hoon Choi
KISTI
Daejeon, Republic of Korea
choid@kisti.re.kr

Chen Li
University of California
Irvine, CA
chenli@ics.uci.edu

ABSTRACT

We study the problem of graph similarity search with a graph edit distance (GED) constraint. Existing solutions adopt a filtering-and-verification framework, with a focus on the filtering phase where a feature-based index is used to reduce the number of candidate graphs to be verified. These solutions suffer from a computationally expensive verification phase. In this paper, we develop a novel technique called Inves that can significantly reduce the time of verifying a candidate graph. Its main idea is to judiciously and incrementally partition a candidate graph based on the query graph, and use the results to compute a lower bound of their distance. If a full GED computation is needed, Inves utilizes the collected information, and uses novel methods and an A* algorithm to search in the space of possible vertex mappings between the graphs to compute their GED efficiently. A main advantage of Inves is that it can be adopted by a plethora of graph similarity search algorithms. Our extensive experiments on both real and synthetic datasets show that Inves can significantly improve the performance of existing techniques by an order of magnitude.

1 INTRODUCTION

Graph data models are widely used in representing complex objects, such as chemical compounds, social networks, and biological structures. Graph search, which finds all occurrences of a query graph in a database of graphs, is a fundamental operation needed in many applications. To tolerate data inconsistency, natural noises, and different data representations in graph search, very often these applications require finding graphs similar to a given query graph. Various similarity measures have been proposed, such as maximum common subgraphs [2, 15], missing edges and features [23, 28], and graph alignment [17]. Among them, one of the commonly used metric is graph edit distance (GED) [5, 6, 22], which can capture the structural difference between graphs, and can be applied to many types of graphs [22, 24]. The GED between two graphs is the minimum number of graph edit operations to transform one to the other, where a graph edit operation is insertion, deletion, or substitution of a single vertex or edge.

The problem of graph similarity search is to find graphs in a database whose GED to a query graph is within a given threshold. This problem is challenging because GED computation between two graphs is NP-hard [22]. Generally, a scan-based approach that directly computes the GED between each data graph and the query graph is computationally prohibitive. Many existing solutions adopt a filtering-and-verification framework. An index structure is typically used to generate candidate graphs in the filtering phase, and each candidate is compared with the query graph to find if it is a true match in the verification phase. Existing studies mainly focus on developing a feature-based index to generate candidates in the filtering phase. For example, c-star [22] and k-AT [19] extract tree-structured features from data graphs and build an inverted index on the extracted features. GSimSearch [25, 26] builds an inverted index on path-based features of graphs. Pars [24] and MLIndex [12] utilize partitions of graphs as features to be indexed.

The performance of existing solutions can suffer from too many candidates and an expensive verification phase. Table 1 shows the performance of 100 queries using one of the index-based search algorithms, Pars [24], on an AIDS dataset containing 42,687 graphs (see Section 5 for details). In the table, we use the number of data graphs that have passed a primitive filter named the global label filter (refer to [25] and Section 3.5 for details of the filter). The number of candidates denotes those candidates that require full GED computations. For example, when the threshold $\tau = 5$, only 15.5% of data graphs are filtered from the index-based filtering phase. Experiments on other solutions show similar behaviors.

Table 1: Performance of Pars

GED threshold τ	1	2	3	4	5
# of data graphs	574	3,335	12,669	34,774	74,937
# of candidates	142	591	4,931	22,846	63,301
# of answers	105	135	161	221	278
Filtering ratio	75.3%	82.3%	61.1%	34.3%	15.5%

To solve this problem, in this paper we develop a novel verification technique, called Inves[1]. Given a set of candidate graphs generated from a filtering phase, the proposed technique can effectively reduce the time for verifying if the GED between each candidate graph and the query graph is within a given threshold. Its main idea is to judiciously and incrementally partition the candidate graph based on the query graph, and use the results to try to prune this pair. If a full GED computation is needed, Inves utilizes the collected information, and uses novel methods and an A* algorithm to search in the space of possible vertex mappings between the graphs to compute their GED efficiently. A main advantage of Inves is that it can be adopted by a plethora of graph similarity search algorithms.

The following are our contributions:

- We propose Inves as a novel incremental partitioning-based verification technique. Given a candidate graph and a query graph with a GED threshold, Inves incrementally isolates subgraphs of the candidate graph that cause mismatches with the query graph. If the number of isolated subgraphs is greater than

[1] It stands for Incremental partitioning-based verification technique for graph edit similarity search.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.21

the threshold, Inves filters out this pair since their GED cannot be within the threshold. In Section 3, we present the details of the incremental partitioning-based verification framework.

- If the pair of graphs cannot be pruned using the generated subgraphs, Inves employs efficient methods based on a well-known A* algorithm for GED computation [14]. It also takes advantage of the partitioning results by first considering those vertices that cause edit errors. In this way, it can significantly reduce the search space of the A* algorithm, thus improve the performance of GED computation (Section 4).
- We conduct extensive experiments to evaluate Inves on both real and synthetic data sets (Section 5). The results show the benefits of the various optimization methods in the technique. In addition, by adopting Inves in existing index-based algorithms, we can significantly reduce the total running time by an order of magnitude.

The rest of the paper is organized as follows: Section 2 provides preliminaries and reviews related work. Section 3 presents the proposed verification framework, and Section 4 provides our GED computation methods. Section 5 presents experimental results, and Section 6 concludes the paper.

2 PRELIMINARIES AND RELATED WORK

2.1 Graph Similarity Search Problem

We focus on undirected labeled simple graphs defined as follows. An undirected labeled simple graph g is a triple (V_g, E_g, L_g), where V_g is a set of vertices, $E_g \subseteq \{(u, v) \mid u \in V_g \wedge v \in V_g \wedge u \neq v\}$ is a set of edges, and L_g is a labeling function that maps vertices and edges to labels. $L_g(v)$ and $L_g(u, v)$ respectively denote the label of a vertex v and the label of an edge (u, v). If there is no edge between u and v, $L_g(u, v)$ returns a unique value λ distinguished from all other edge labels. There are no self-loops nor more than one edge between two vertices. For simplicity, in the rest of the paper, we use graph to denote undirected labeled simple graph.

The graph edit distance (or GED for short) between two graphs x and y, denoted by $ged(x, y)$, is the minimum number of graph edit operations that transform x to y. A graph edit operation is one of the following: (1) insertion of an isolated labeled vertex; (2) deletion of an isolated labeled vertex; (3) substitution of the label of a vertex; (4) insertion of a labeled edge; (5) deletion of a labeled edge; or (6) substitution of the label of an edge.

Example 1. *Figure 1 shows two graphs x and y, which include vertex labels representing atom symbols and edge labels (i.e., single and double lines) representing chemical bonds. Besides vertex labels, the graphs also include vertex identifiers. To transform x into y, we can do the following three graph edit operations on x: insertion of a single-bond edge between u_3 and u_5, and substitutions of labels of u_2 and u_8. Therefore, $ged(x, y)$ is 3.*

We formalize the problem of graph edit similarity search as follows.

Definition 1 (Graph Similarity Search Problem). *For a graph database $\mathcal{D} = \{x_1, \ldots, x_n\}$ and a query graph y with a GED threshold τ, the graph edit similarity search finds every data graph $x_i \in \mathcal{D}$ such that $ged(x_i, y) \leq \tau$.*

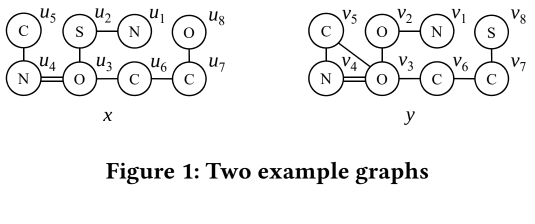


Figure 1: Two example graphs

2.2 A* algorithm for GED Computation

In this section, we review the most widely used algorithm for GED computation [14], which is based on A*. Given a pair of graphs x and y, the A* algorithm basically traverses all possible vertex mappings between x and y in a best-first fashion. It maintains a priority queue that contains states in its state-space tree, where each state in the tree represents a partial vertex mapping between the pair. The priority (or edit distance) of a state is determined by the sum of (1) the existing distance g: the edit operations detected from the initial state to the current state; and (2) an estimated distance h: a heuristic estimation of the edit operations from the current state to the goal. The A* algorithm guarantees that it finds an optimal mapping if h is not overestimated.

```
Algorithm 1: GED($x$, $y$, $\tau$)
  input  : $x$ and $y$ are graphs; $\tau$ is a GED threshold.
  output : if $ged(x,y) \leq \tau$, $ged(x,y)$; otherwise, $\tau$+1
1  $O \leftarrow$ order the vertices in $x$;
2  $Q \leftarrow \emptyset$; $Q$.push($\emptyset$);
3  while $Q \neq \emptyset$ do
4      $M \leftarrow Q$.pop();
5      if complete($M$) then return existingDistance($M$);
6      $u \leftarrow$ next unmapped vertex in $V_x \cup \{\varepsilon\}$ as per $O$;
7      foreach $v \in (V_y \cup \{\varepsilon\})$ s.t. $v \notin M$ do
8          $g \leftarrow$ existingDistance($M \cup \{u \rightarrow v\}$);
9          $h \leftarrow$ estimateDistance($M \cup \{u \rightarrow v\}$);
10         if $g + h \leq \tau$ then $Q$.pushQueue($M \cup \{u \rightarrow v\}$);
11 return $\tau + 1$;
```

The GED computation algorithm is outlined in Algorithm 1. It first determines the order of vertices in x, and pushes the initial state, i.e., an empty mapping, into the queue (Lines 1–2). In the main loop, it removes a mapping M from the queue that has a minimum edit distance (Line 4). If M contains all vertices of x and y, it returns the existing distance of M (Line 5). Otherwise, it expands its state-space tree by mapping the next unmapped vertex u in x (Line 6) to each unmapped vertex v in y (Line 7). It pushes each expanded state into the queue if the edit distance of the state is not greater than τ (Lines 8–10). In the algorithm, ε is used to denote an insertion or a deletion of a vertex. If it fails to find any mapping whose edit distance is not greater than τ, it returns $\tau + 1$ (Line 11).

2.3 Related Work

Previous work on the graph similarity search utilizes small overlapping substructures to establish a filtering condition between dissimilar graphs. Motivated by the gram idea used in string similarity searches, the k-AT algorithm [19] defines a q-gram as a tree rooted at a vertex v with all vertices reachable to v in q hops. A star structure, which is 1-gram defined by k-AT, has been proposed to set up a filtering condition through bipartite matching between star structures [22]. SEGOS [20] is a two-level index structure proposed to efficiently search star structures. The main focus of these approaches has been on the filtering phase to develop efficient index-based filtering methods using those substructures.

GSimSearch [25, 26] proposed a path-based q-gram and developed an index-based filtering technique based on the observation of the algorithm called ED-join [21] in string search. To further reduce the number of candidates, GSimSearch proposed local label filtering in its verification phase. However, this technique is based on small fixed-size substructures of graphs, thus edit errors are mainly captured from label differences, and structural differences are considered inside small substructures only.

There is recent work that makes use of large disjoint substructures of graphs to capture structural differences between graphs. Pars [24] partitions data graphs into disjoint subgraphs, and makes an index on the partitioned subgraphs. Using the index, it identifies data graphs having partitions contained in the query graph, and generates them as candidate graphs. It employs a random-graph-partitioning strategy and refines initial partitioning results based on a query workload. It also dynamically rearranges indexed partitions in a restricted way while searching its index structure. MLIndex [12] was proposed to reduce the number of candidates by indexing a few alternative partitioning results of data graphs. It defines a selectivity of a partition based on vertex and edge label frequencies, and divides a graph in a way to increase selectivities of partitions. Despite the efforts in the previous approaches, their filtering power of partitions is inherently limited because partitions of data graphs are determined offline, and one or a few rigid partitionings of a data graph cannot work well for all queries.

Other related work includes Mixed [27] and LBMatrix [3]. Mixed generates candidates by using small and large disjoint substructures of a query graph. LBMatrix has proposed a q-gram-based matrix index structure that can be stored in external memory to handle very large datasets.

3 INVES: VERIFICATION FRAMEWORK

In this section, we propose the Inves verification framework aiming to efficiently verify if the GED between a pair of graphs is within a given threshold. We first introduce the partition-based verification principle, then present the details of Inves.

3.1 Partition-based Verification Scheme

Due to the high cost of GED computation, it makes the graph similarity search impractical to directly compute the GED between a candidate and the query when there are many candidates generated from an index-based filtering phase. To efficiently verify a pair of graphs, in this paper we use a partition-based lower bound of the GED between the pair before computing the exact GED. We begin with the concept of an induced subgraph for defining graph partitions, then present the verification scheme.

Definition 2 (Induced Subgrpah Isomorphism). *A graph r is induced subgraph isomorphic to another graph s, denoted as $r \sqsubseteq s$, if there exists an injection $f\colon V_r \to V_s$ such that $\forall u \in V_r$, $f(u) \in V_s \wedge L_r(u) = L_s(f(u))$ and $\forall u \in V_r$, $\forall v \in V_r$, $L_r(u,v) = L_s(f(u), f(v))$. In this case, the graph r is called an induced subgraph of s.*

Recall that the edge labeling function $L_g(u,v)$ returns a unique value λ if there is no edge between u and v in a graph g. It enables us to check the inducedness of a subgraph in Definition 2.

Example 2. *Consider the graphs p_1, p_2, and y in Figure 2. $p_1 \sqsubseteq y$, but $p_2 \not\sqsubseteq y$ because $L_{p_2}(u_4, u_6) = \lambda \neq L_y(v_3, v_5) =$ single-bond.*

Given a graph g and a vertex set $V \subseteq V_g$, there is only one induced subgraph p of g such that $V_p = V$. That is, p is uniquely identified by V. Therefore, we use V interchangeably with the induced subgraph of g defined by V.

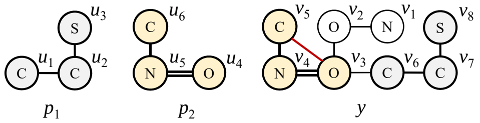


Figure 2: Induced subgraph isomorphism

Definition 3 (Graph Partitioning). *A partitioning of a graph g is $\mathcal{P}(g) = \{p_1, \ldots, p_k\}$ such that $\forall i\ p_i \sqsubseteq g$, $\forall i, j\ i \neq j \Rightarrow V_{p_i} \cap V_{p_j} = \emptyset$, and $V_g = \bigcup_{i=1}^{k} V_{p_i}$.*

Given a pair of graphs x and y, consider a partition $p \in \mathcal{P}(x)$. If $p \sqsubseteq y$, we say p is *matching* with y. Otherwise, we say p is *mismatching* with y. We also simply call p a matching (or mismatching) partition if y is clear from the context. An induced subgraph o of y such that $V_o \subseteq V_y$ is called an *occurrence* of p in y if and only if $p \sqsubseteq o$ and $o \sqsubseteq p$. In Figure 2, for example, $o = \{v_6, v_7, v_8\}$ is an occurrence of p_1 in y.

With the graph partitioning, a lower bound of the GED between a pair of graphs are calculated as follows.

Lemma 1. *Consider a pair of graphs x and y with a graph partitioning $\mathcal{P}(x)$. $lb(x,y) = |\{p \mid p \in \mathcal{P}(x) \wedge p \not\sqsubseteq y\}|$ is a lower bound of the GED between the pair.*

Proof. Since partitions of x share neither a vertex nor an edge, an edit operation on a partition does not affect to another partition. Therefore, each mismatching partition p requires at least one edit operation to transform x to y. □

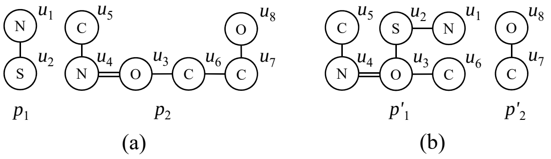
The following corollary states the partition-based verification scheme based on the lower bound in Lemma 1.

Corollary 1. *Given a GED threshold τ, consider a pair of graphs x and y with a graph partitioning $\mathcal{P}(x)$. If $lb(x,y) > \tau$, the pair can be pruned without the GED computation.*

Partition-based lower bounds and their variants have been extensively studied and discussed in the literature of string similarity search (e.g. [8, 11]) and approximate subsequence mapping (e.g. [1, 9, 10]). The same principle is well adopted in recent work for graph similarity search [12, 24, 27]. Our lower bound in Lemma 1 is a simple extension of existing partition-based approaches. While the focus of existing work is on building a partition-based inverted index for the filtering phase, our focus in this paper is on the verification phase to efficiently verify a candidate graph using the partition-based lower bound.

To obtain the lower bound in Lemma 1, we need $|\mathcal{P}(x)|$ induced subgraph isomorphism tests, which are generally NP-hard. However, former studies have empirically showed that subgraph isomorphism test is on average three orders of magnitude faster than GED computation [12, 24], and thus it can be practically used in deriving a partition-based lower bound.

Example 3. *Consider a pair of graphs x and y shown in Figure 1 with a GED threshold $\tau = 1$. If we partition x into $\{p_1, p_2\}$ as depicted in Figure 3(a), $lb(x,y) = 2$ because $p_1 \not\sqsubseteq y$ and $p_2 \not\sqsubseteq y$. Therefore, we can safely prune the pair without GED computation according to Corollary 1. If we partition x into $\{p_1', p_2'\}$ as illustrated in Figure 3(b), $lb(x,y) = 1$ since $p_1' \not\sqsubseteq y$ but $p_2' \sqsubseteq y$. Thus, we need a GED computation between the pair.*



(a) (b)

Figure 3: Two ways to partition x in Figure 1

As shown in the example above, the tightness of $lb(x, y)$ is highly dependent on the way to partition x. However, the graph partitioning problem is in general NP-hard [12, 24] and enumerating every possible partitioning to obtain an optimal partitioning is intractable. In the next section, we introduce a measure for a partitioning to develop a good partitioning technique.

3.2 A Qualitative Measure for a Partitioning

Consider a pair of graph x and y with a partitioning $\mathcal{P}(x)$. An inherent limitation of partition-based approaches is that the containment test of each partition is independent, and thus multiple partitions of x can be matching with y in overlapping areas of y. This limitation makes the partition-based bound loose. However, it is hard to tackle the problem because $lb(x, y)$ can exceed the GED if we use a non-overlapping alignment of partitions, where a mismatching partition p is allowed to be aligned to a subgraph of y whose size is less than the size of p. Finding a legal non-overlapping alignment of partitions (i.e., an alignment that results in a minimum lower bound) is computationally impractical.

Beside this fundamental limitation, the following are major problems that make the partition-based lower bound loose.

- P_1 In partition-based approaches, only one edit error is counted from a mismatching partition. A tighter bound can be calculated as $lb(x, y) = \sum_{p \in \mathcal{P}(x) \wedge p \not\sqsubseteq y} sed(p, y)$, where $sed(p, y)$ denotes the subgraph edit distance [22, 26] between p and y.
- P_2 A substructure of x that causes insertion or deletion errors can be divided into multiple partitions. In this case, those edit errors can be hidden between partitions to make the lower bound loose. They can be detected by enumerating every subgraph of x consisting of adjacent partitions, and investigating the subgraphs through subgraph edit distance computations.
- P_3 Edit errors can be buried in edges connecting different partitions and these errors also make the lower bound loose. To precisely find them, we need to solve the problem of placement of partitions into y.

Due to the complexities of subgraph edit distance and partition alignment, the problems above cannot be efficiently solved. The hardness of the limitation and problems also prevents us from accurately analyzing the tightness of $lb(x, y)$. In fact, there is no given proof on the tightness of existing partition-based bounds [6], and it is hard to measure the tightness of $lb(x, y)$ in a quantitative manner. To the best of our knowledge, the only theoretical analysis on the tightness $lb(x, y)$ is that increasing the number of partitions has more chance to get a tighter bound [12]. However, the analysis is based on an assumption that does not take the problem P_2 into consideration. In this paper, instead of a quantitative measure, we introduce a qualitative measure of goodness of a partitioning as stated in the following claim.

Claim 1. *Given two graphs x and y, a partitioning $\mathcal{P}(x)$ is a good partitioning if every mismatching partition $p \in \mathcal{P}(x)$ meets the following conditions.*

- C_1 *Edit errors in p are indivisible, or edit errors in p cannot be distributed over partitions (indivisibility). Ideally, p is minimal, that is, p loses its edit errors and become a matching partition if any vertex in p is removed (mininality).*
- C_2 *An edit error in an edge connecting p to another partition is captured by p, while preserving the condition C_1.*

The indivisibility constraint in C_1 alleviates the problem P_1 since each partition contains the least number of edit errors it can have. The minimality constraint in C_1 alleviates the problem P_2, because by removing unnecessary vertices that do not contribute to edit errors from a partition, those vertices can be combined with other vertices in another partition and cause edit errors. Claim 1 also has the condition C_2 to alleviate the problem P_3.

Although we develop a qualitative measure for a partitioning, it is hard to make a partitioning that exactly meets the measure because a graph partitioning problem even with a simple condition tends to be intractable [24]. Nonetheless, the measure can be a guideline for producing a partitioning to get a tighter bound. In the following sections, we develop a novel partitioning method based on this measure (Section 3.3 for C_1 and Section 3.4 for C_2).

3.3 Incremental Partitioning

In this section, we present a systematic way to produce mismatching partitions that approximately meet the condition C_1 in Claim 1. We begin with the definition of the incremental partitioning strategy.

Definition 4 (Incremental Partitioning). *Given two graphs x and y, an incremental partitioning of x is to extract mismatching partitions from x as follows. Let $V_x = \{u_1, \ldots, u_n\}$. We move the vertices in V_x one after another into a partition p, which is initially empty, while $p \sqsubseteq y$. Let the last vertex moved from x to p be u_l. We finally move u_{l+1} to p to make $p \not\sqsubseteq y$, and produce $\mathcal{P}(x) = \{p, x \backslash p\}$, where $x \backslash p$ denotes the induced subgraph s of x such that $V_s = V_x - V_p$. We repeat this partitioning strategy with $x \backslash p$ until either $x \backslash p \sqsubseteq y$ or $x \backslash p = \emptyset$.*

A graph partitioning produced by the incremental partitioning strategy in Definition 4 satisfies the following property.

Property 1. *Given a pair of graphs x and y, if x is partitioned into $\mathcal{P}(x) = \{p_1, \ldots, p_{k-1}, p_k\}$ using our incremental partitioning strategy, then $p_1, \ldots, p_{k-1}$ are mismatching with y and the last partition p_k, which can be empty, is matching with y. Therefore, $lb(x, y) = k - 1$.*

The following lemma states that the incremental partitioning strategy generates a partitioning that exactly meets the indivisibility constraint in the condition C_1 in Claim 1.

Lemma 2. *Given a partitioning $\mathcal{P}(x) = \{p_1, \ldots, p_{k-1}, p_k\}$ produced by the incremental partitioning strategy in Definition 4, it is not possible to divide any partition $p_i \in (\mathcal{P}(x) - \{p_k\})$ into two partitions p_{i1} and p_{i2} such that $p_{i1} \not\sqsubseteq y \ \wedge p_{i2} \not\sqsubseteq y$.*

Proof. For each $p_i = \{u_b, \ldots, u_e\}$ except p_k, our incremental partitioning scheme guarantees that $(p' = p_i - \{u_e\}) \sqsubseteq y$. Since u_e cannot be included in both p_{i1} and p_{i2}, either $p_{i1} \sqsubseteq p' \sqsubseteq y$ or $p_{i2} \sqsubseteq p' \sqsubseteq y$ should be satisfied. □

Example 4. *For a pair of graphs x and y in Figure 1, we incrementally partition x by comparing it with y as follows. Assume that vertices of x are investigated from u_1 to u_8. We first make $\mathcal{P}(x)$ by isolating $\{u_1, u_2\}$ from x into p'_1 as shown in Figure 4(a), because $(p'_1 - \{u_2\}) \sqsubseteq y$ but $p'_1 \not\sqsubseteq y$. Given two partitions of x, we further* *partition p_2 into p'_2 and p_3 by isolating a mismatching partition $\{u_3, u_4, u_5\}$ from p_2, as depicted in Figure 4(b). Since $p_3 \sqsubseteq y$, we cannot proceed the incremental partitioning. Hence, $lb(x, y) = 2$.*

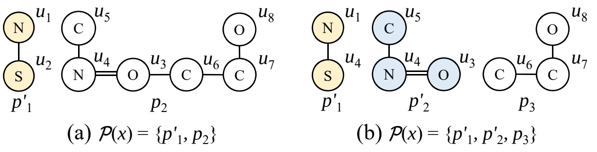

N
S
C
N
O
C
C
O
u_1
u_2
u_5
u_4
u_3
u_6
u_7
u_8
p'_1
p_2
(a) $\mathcal{P}(x) = \{p'_1, p_2\}$
p'_1
p'_2
p_3
(b) $\mathcal{P}(x) = \{p'_1, p'_2, p_3\}$

Figure 4: Incremental partitioning of x in Figure 1

When the incremental partitioning strategy produces a mismatching partition, an induced subgraph isomorphism test is an essential operation. The common principle on subgraph isomorphism test is to visit vertices based on connectivity of vertices and frequencies of vertices and edges [7, 16]. Following the existing solutions, we investigate vertices of x by considering infrequent vertices and edges early while preserving the connectivity.

Given a mismatching partition p in $\mathcal{P}(x)$ generated from the incremental partitioning strategy, we can find an induced subgraph of p that meets the minimality constraint in the condition C_1 as follows. Since the last vertex in p causes the mismatch, we enumerate every induced subgraph of p containing the last vertex and perform an induced subgraph isomorphism test against y to find a subgraph s such that $s \not\sqsubseteq y$ and $|V_s|$ is minimum. This process is obviously time consuming. Instead of finding a minimal one, we propose a method that refines a mismatching partition in $\mathcal{P}(x)$ to approximately meet the minimality constraint.

After we find a mismatching partition p, we rematch p against y using an alternative vertex ordering of p to remove unnecessary vertices from p that do not contribute to edit errors. Let the mismatching partition p be $\{u_1, \ldots, u_f\}$. Because $\{u_1, \ldots, u_{f-1}\}$ is matching with y by Definition 4, u_f causes the mismatching and edit errors are likely to be clustered in u_f and vertices adjacent to u_f. Therefore, by using the vertex u_f as the start vertex and reordering p in the same way (i.e., considering infrequent vertices and edges early while preserving the connectivity), we have a chance to reduce the size of the mismatching partition. The following example illustrates rematching of a mismatching partition to reduce the size of the mismatching partition.

Example 5. *Consider a pair of graphs x and y in Figure 5. Assume the vertices of x is ordered as $\{u_1, u_2, u_3, u_4, u_5, u_6\}$. Based on the order, we isolate $\{u_1, u_2, u_3, u_4\}$ into a separate partition p. In this case, $x \backslash p$ is matching with y and $lb(x, y) = 1$. We reorder vertices in the mismatching partition p into $\{u_4, u_3, u_2, u_1\}$ by using u_4 as the first vertex and preserving the connectivity of the vertices. By rematching p against y using the vertex ordering, we reduce the mismatching partition p to $\{u_4, u_3\}$. From $x \backslash p$, in this case, we can find one more mismatching partition $\{u_1, u_5\}$, which is refined from $\{u_1, u_2, u_5\}$ by the rematching method, and obtain a tighter bound $lb(x, y) = 2$.*

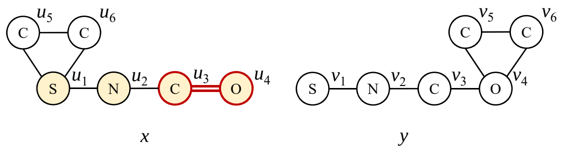


Figure 5: Rematching a mismatching partition

Table 2: Average number of rematching

GED threshold τ	1	2	3	4	5
AIDS	1.41	1.36	1.37	1.38	1.38
PROTEIN	1.50	1.81	1.76	1.69	1.59
PubChem	1.56	1.50	1.47	1.46	1.46

To further reduce the size of a mismatching partition, we repeat rematching while the partition size decreases. As the edit errors are likely to be clustered around the last vertex, we can expect that subgraph isomorphism tests are terminated early and the number of rematching is very small. Table 2 shows the average number of rematching for AIDS, PROTEIN, and PubChem datasets when extracting a mismatching partition (see Section 5 for details of the datasets and queries).

```
Algorithm 2: IncrementalPartitioning($x$, $y$)
   input  : $x$ and $y$ are graphs
   output : a partition-based GED lower bound $lb(x, y)$
1  DetermineVertexOrdering($x$);
2  $f \leftarrow$ InducedSI($x, y, \emptyset$);
3  if $f > |V_x|$ then return 0 ;
4  $p \leftarrow$ first $f$ vertices of $x$;
5  repeat
6      DetermineRematchingOrdering($p$);
7      $f \leftarrow$ InducedSI($p, y, \emptyset$);
8      $p \leftarrow$ first $f$ vertices of $p$;
9  until $|V_p|$ does not change;
10 $x' \leftarrow x \backslash p$;
11 foreach connected component $c \in x'$ do
12     if $|V_c| \le \alpha$ then $x' \leftarrow x' \backslash c$;
13 return 1+IncrementalPartitioning($x'$, $y$);
```

Algorithm 2 outlines the incremental partitioning algorithm. Given a pair of graphs x and y, the algorithm computes the lower bound $lb(x, y)$ by partitioning x based on the condition C_1 in Claim 1. It first determines the vertex ordering of x using DetermineVertexOrdering (omitted, Line 1) and then perform an induced subgraph isomorphism test of x against y based on the ordering (Line 2). InducedSI, which will be presented at the end of the next section, identifies and returns the least vertex position in x that makes the matching fail. If the position is greater than the number of vertices in x, then $x \sqsubseteq y$, and return $lb(x, y) = 0$ (Line 3). Otherwise, it extracts the vertices causing the mismatch into a partition p (Line 4).

The algorithm reduces the size of the mismatching partition p using the rematching method (Lines 5–9). DetermineRematchingOrdering (omitted, Line 6) is the same with DetermineVertexOrdering except that it uses the last vertex in p as the start vertex. After reordering vertices in p, the algorithm rematches p against y (Line 7). It repeats rematching while the size of the mismatching partition p shrinks (Line 9). The algorithm finally detaches p from x to make x' (Line 10).

After isolating a mismatching partition p from x, the remaining part of x, which is x', often forms a disconnected graph. We observed that a tiny connected component in a disconnected graph can cause a serious performance problem in subgraph isomorphism test. The existing subgraph isomorphism algorithms assume connected graphs, and thus they do not pay attention to this problem. To prevent this worst case in subgraph isomorphism test, the algorithm removes each tiny connected component c from x' such that $|V_c| \leq \alpha$, where α is a tunable parameter (Lines 11–12). Then, it recursively identifies the number of mismatching partitions in x' and returns $lb(x, y)$ (Line 13).

Correctness and Complexity of Algorithm 2: Whenever a mismatching partition is identified, the algorithm increments the lower bound by 1 (Line 13). Therefore, the algorithm correctly returns a lower bound by Lemma 1. Assuming the number of rematching is bound to a constant, the worst case complexity is $\sum_{p \in \mathcal{P}(x)} O((\gamma_p \cdot \gamma_p)^{|V_p|}) = O((\gamma_x \cdot \gamma_x)^{|V_x|})$, which is the same as traditional subgraph isomorphism, where γ_g denotes the maximum vertex degree in a graph g.

3.4 Exploiting Bridges

In this section, we propose a novel technique to detect and exploit edit errors buried in those edges connecting different partitions. With the proposed technique, we develop the *bridge constraint* to meet the condition C_2 in our qualitative measure. We first define bridge and then present formulas to count edit errors in bridges.

Definition 5 (Bridge). *Given a partition p, a bridge of a vertex $u \in p$ is an edge connecting u to a vertex $u' \notin p$.*

Lemma 3. *Given a partition p of a graph x and an occurrence o of p in another graph y, suppose a vertex $u \in p$ is mapped to a vertex $v \in o$.*

(1) The number of edit errors between bridges of u and v is

$$\mathcal{B}_e(u, v) = \Gamma(L_{br}(u), L_{br}(v)),$$

where $L_{br}(w)$ denotes the label multiset of the bridges of a vertex w, and $\Gamma(A, B)$ is $\max(|A - B|, |B - A|)$.

(2) The number of edit errors in bridges of p with respect to o is

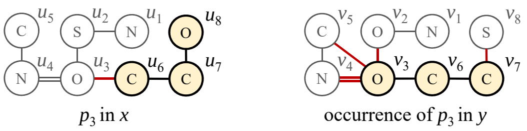
$$\mathcal{B}(p, o) = \mathcal{B}(M) = \sum_{u \to v \in M} \mathcal{B}_e(u, v),$$

where M denotes the vertex mapping between p and o, which are identified during induced subgraph isomorphism test of p.

Proof. (1) Let $D_1 = L_{br}(u) - L_{br}(v)$ and $D_2 = L_{br}(v) - L_{br}(u)$, and assume $|D_1| \geq |D_2|$. To transform $L_{br}(u)$ to $L_{br}(v)$, we need $|D_2|$ substitutions of labels in D_1 and $|D_1| - |D_2|$ deletions of labels in D_1. That is, we need $|D_2| + |D_1| - |D_2| = |D_1| = \Gamma(L_{br}(u), L_{br}(v))$ edit operations. (2) Since no bridge can shared by multiple vertices in p by Definition 5, the number of edit errors in p is the sum of the number of edit errors in the bridges of p. □

The following example illustrates the number of edit errors in bridges of a matching partition.

Example 6. *In Example 4, consider the matching partition $p_3 = \{u_8, u_7, u_6\}$ and its occurrence $o = \{v_3, v_6, v_7\}$ in y as shown in Figure 6. $\mathcal{B}_e(u_8, v_3) = 3$ because u_8 has no bridge while v_3 has 3 bridges. Likewise, $\mathcal{B}_e(u_7, v_6) = 0$ and $\mathcal{B}_e(u_6, v_7) = 0$. Therefore, $\mathcal{B}(p_3, o) = 3 + 0 + 0 = 3$.*



p_3 in x

occurrence of p_3 in y

Figure 6: Bridge errors of a matching partition in Figure 4

Given two partitions p and p' of a graph x, suppose that a bridge e connecting p and p' causes one edit error with respect to another graph y. When we count edit errors in bridges of x, the edit error in e is counted twice (i.e., once in p and once in p'). Hence, we can use half of the edit errors counted in bridges so that we do not over-count edit errors in x. Lemma 4 formally states this observation.

Lemma 4. *Given a pair of graphs x and y, consider a matching partition p in x and an occurrence o of p in y. The mapping between p and o causes at least $\lfloor \mathcal{B}(p, o)/2 \rfloor$ edit errors.*

Proof. In this proof, we consider deletion or substitution errors of bridges in x only. Insertion of bridges to x can be proved similarly. Consider we have a partitioning of x such that the i^{th} partition p_i has e_i bridges. Since each bridge is shared by two partitions, bridges should be distributed in a disjoint manner. We distribute bridges to each partition using the following procedure.

```
initially, all bridges are unassigned;
$p$ ← an arbitrary partition;
while there is an unassigned bridge in $x$ do
    if no unassigned bridge is connected to $p$ then
        $p$ ← an arbitrary partition to which at least one
          unassigned bridge connected;
    $e$ ← an unassigned bridge connected to $p$;
    assign $e$ to $p$;
    $p$ ← the partition connected to $p$ via $e$;
```

The procedure above guarantees that at least $\lfloor e_i/2 \rfloor$ bridges are assigned to p_i because if a partition loses a bridge, another bridge (if exists) is always assigned to the partition. If we consider bridges causing edit errors only (i.e., each of e_i bridges causes an edit error), p_i has at least $\lfloor e_i/2 \rfloor$ edit errors. Since there are $\mathcal{B}(p, o)$ edit errors in the bridges connected to p, we can always assign at least $\lfloor \mathcal{B}(p, o)/2 \rfloor$ edit errors to p using the procedure. □

By pushing edit errors in bridges into a matching partition, we can make a rigorous partition matching condition called the bridge constraint as follows.

Corollary 2. *Given a partition p of a graph x and another graph y, p is matching with y if and only if there exists an induced subgraph o of y such that $V_o \subseteq V_y$, $o \sqsubseteq p$, $p \sqsubseteq o$, and $\mathcal{B}(p, o) < 2$.*

Example 7. *In Example 6, since o is the only occurrence of p_3 in y and $\mathcal{B}(p_3, o) \geq 2$, p_3 is mismatching with y by Corollary 2. Therefore, in Example 4, the graph x is divided into four partitions (three mismatching partitions and one empty partition), and we obtain a tighter lower bound $lb(x, y) = 3$.*

Notice that our bridge constraint detects edit errors much more accurately than the half-edge subgraph isomorphism used in existing techniques [12, 24]. For example, in Example 6 and 7, existing techniques cannot detect any edit errors in p_3 (we omit the precise comparison in the interest of space; refer to Pars[24] for the details of the half-edge subgraph isomorphism).

By integrating the bridge constraint with the induced subgraph isomorphism test, we can detect a mismatching partition early to approximately preserve the indivisibility and minimality constraints in C_1 of Claim 1. Algorithm 3 encapsulates our induced subgraph isomorphism test with the bridge constraint.

Algorithm 3: InducedSI(x, y, M)

```
input  : $x$ and $y$ are graphs;
         $M$ is a mapping vector (initially $\emptyset$).
output : the least position in $x$ where the matching fails.
1  $iteration \leftarrow |M| + 1$;
2  if $iteration > |V_x|$ then return $iteration$ ;
3  $u \leftarrow$ the $iteration^{th}$ vertex in $V_x$;
4  $C \leftarrow \{v \mid v \in y \wedge v \notin M \wedge L_x(u) = L_y(v)\}$;
5  foreach $v \in C$ do
6      if $\forall u' \rightarrow v' \in M\ L_x(u, u') = L_y(v, v')$ and
       $\mathcal{B}(M \cup \{u \rightarrow v\}) < 2$ then
7          $depth \leftarrow$ InducedSI($x$, $y$, $M \cup \{u \rightarrow v\}$);
8          if $iteration < depth$ then $iteration \leftarrow depth$;
9          if $iteration > |V_x|$ then return $iteration$;
10 return $iteration$;
```

Like most existing subgraph isomorphism techniques, our algorithm also adopts the Ullmann's algorithm [18] with a difference that ours returns the least vertex position in a partition where the induced subgraph isomorphism test fails. Given a pair of graphs x and y, the algorithm maps the vertices in x one by one to find a mapping M between x and y. For the current vertex u of x (Line 3), it enumerates all unused vertices $v \in y$ whose label is equivalent to the label of u (Line 4), and test if the vertex mapping $u \rightarrow v$ is valid (Lines 6). Then, the bridge constraint in Corollary 2 is applied to the vertex mapping $M \cup \{u \rightarrow v\}$ (Line 6). If it is a valid mapping, the algorithm goes down to the next vertex of x (Line 7). It keeps track of the least position (or maximum iteration count) in x where the induced subgraph isomorphism will fail (Lines 1, 8), and returns the position if $x \not\sqsubseteq y$ (Line 10). If $x \sqsubseteq y$, the algorithm returns $|V_x| + 1$ (Lines 2, 9).

Example 8. *Given a pair of graph x and y depicted in Figure 7, consider we perform* InducedSI$(x, y, \emptyset)$. *Let us assume the vertex ordering of x is from u_1 to u_6. At the first iteration,* InducedSI *adds $u_1 \rightarrow v_7$ into M, and considers $u_2 \rightarrow v_2$ at the second iteration. Because $L_x(u_2, u_1) = L_y(v_2, v_7)$ and $\mathcal{B}(\{u_1 \rightarrow v_7\} \cup \{u_2 \rightarrow v_2\}) = 1$, it adds $u_2 \rightarrow v_2$ into M. At the third iteration, it maps the next vertex u_3 to v_3, and checks the inducedness: $L_x(u_3, u_1) = L_y(v_3, v_7) = \lambda$ and $L_x(u_3, u_2) = L_y(v_3, v_2)$. Then, it tests the bridge constraint and fails to find an occurrence because $\mathcal{B}(\{u_1 \rightarrow v_7, u_2 \rightarrow v_2\} \cup \{u_3 \rightarrow v_3\}) = 2$. Therefore, it returns its iteration count 3, which denotes $\{u_1, u_2, u_3\}$ is a mismatching with y.*

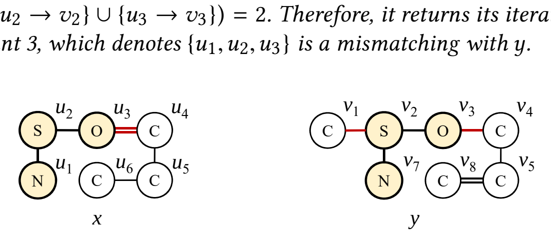


Figure 7: Example of InducedSI

Correctness of Algorithm 3: Given two vertices u and v in a graph g, $L_g(u, v)$ returns a unique value λ when there is no edge between u and v. Therefore, it correctly checks the inducedness of x in Line 6. Mismatching caused by bridge differences is also detected from Line 6, where the correctness is guaranteed by Corollary 2. Because the algorithm basically follows Ullmann's algorithm except the test of inducedness, it correctly computes the induced containment of x. It can be inductively verified the algorithm correctly returns the least position where the isomorphism test fails.

3.5 Verification Algorithm

In this section, we provide Inves verification algorithm. Given a pair of graph x and y and a GED threshold τ, Inves incrementally partitions x to obtain a GED lower bound, and prune the pair if the lower bound is greater than τ. Otherwise, Inves directly calculates the GED between x and y.

Algorithm 4: InvesVerifier(x, y, τ)

```
input  : $x$ and $y$ are graphs; $\tau$ is a GED threshold.
output : a boolean value of $ged(x, y) \leq \tau$
1  if $\Gamma(L_V(x), L_V(y)) + \Gamma(L_E(x), L_E(y)) > \tau$ then
2      return false;
3  $lb \leftarrow$ IncrementalPartitioning($x$, $y$);
4  if $lb > \tau$ then return false;
5  $p \leftarrow$ the last partition of $\mathcal{P}(x)$;
6  if $|V_p|/|V_x| > \beta$ then
7      $M \leftarrow$ vertex mapping between $V_p$ and $V_y$;
8      if GEDPartial($M, x, y, \tau$) $\leq \tau$ then return true;
9  return GED($x, y, \tau$) $\leq \tau$;
```

Algorithm 4 shows the details of Inves verification algorithm. Using the label differences of vertices and edges, it first computes a loose GED lower bound and prune the pair if the bound is greater than τ, where $L_V(g)$ and $L_E(g)$ denote the label multisets of vertices and edges in a graph g respectively (Lines 1–2). This technique is originated from the letter-count filter in the problem of DNA read mapping [1, 4] and exploited recently in graph similarity search as a name of the global label filter [25]. Because the global label filter is very simple and highly selective, it is essentially used in graph similarity search (e.g., [24, 25]). After applying the global label filter, Algorithm 4 uses IncrementalPartitioning presented in Algorithm 2 to obtain a partition-based lower bound (Line 3). If the lower bound is greater than τ, it prune the pair (Line 4). We remark that it is obviously optimized by pushing the threshold into IncrementalPartitioning and terminating the partitioning process as soon as $\tau + 1$ mismatching partitions are found.

If the algorithm fails to prune the pair, the last partition $p \in \mathcal{P}(x)$, which can be an empty partition, is matching with y according to Property 1 (Line 5), and a vertex mapping M between p and y is obtained from InducedSI (Line 7). The algorithm exploits this mapping to compute the GED by using it as the initial state of the A* algorithm (i.e., pushing the mapping into the queue instead of an empty mapping in Line 2 of Algorithm 1) (GEDPartial, Line 8). This procedure is called a *partial GED computation.* Notice that the distance calculated by the partial GED computation is an upper bound of the GED of the pair. If it finds the pair meets τ through the partial GED computation, therefore, it can save the time for traversing vertices in M. To prevent frequent invocations of partial GED computation for false positives, we use the partial GED computation only when the size of matching partition is big enough (the tunable parameter β in Line 6). If it fails to identify if $ged(x, y) \leq \tau$ from the partial GED computation, it finally performs a full GED computation between x and y (Line 9).

Correctness of Algorithm 4: It can be seen GEDPartial correctly returns a GED upper bound. Hence, the correctness of the algorithm is guaranteed by Lemma 1 and Corollary 1.

4 INVES: EFFICIENT GED COMPUTATION

In this section, we develop new methods on top of Algorithm 1 to improve the performance of GED computation. We first propose a method to accurately calculate an estimated distance of a vertex mapping. We then propose a vertex ordering technique that takes advantage of the partitioning results of InvesVerifier.

The performance of the A* algorithm in Algorithm 1 depends on the accuracy of an estimated distance of unmapped vertices and edges. Riesen et al. proposed a bipartite heuristic [14], which gives a lower bound of the distance between unmapped parts with bipartite matching. GSimSearch [25, 26] show that the lower bound of the bipartite heuristic is exactly the same as the label difference in unmapped parts in the unweighted case. This approach does not improve the accuracy of the estimation, but it is significantly faster than the bipartite heuristic because the bipartite heuristic uses the Hungarian algorithm [13] with a high complexity of $O(n^3)$.

To improve the accuracy of the estimated distance, in this paper we distinguish bridges of mapped vertices (i.e., edges connecting mapped vertices to unmapped vertices) from unmapped edges. For a vertex mapping M, each $u \to v \in M$ has $\mathcal{B}_e(u, v)$ edit errors in bridges. Since two different mapped vertices in a graph cannot share any bridges, the total edit errors in the bridges of M are $\mathcal{B}(M)$. Therefore, the estimated distance of M, denoted by $h(M)$, can be calculated by the sum of $\mathcal{B}(M)$ and the label difference in unmapped vertices and unmapped edges except bridges. Formally:

$$h(M) = \mathcal{B}(M) + \Gamma(L_V(x'), L_V(y')) + \Gamma(L_E(x'), L_E(y')),$$

where x' and y' respectively denote the unmapped part of x and y except bridges. The following example illustrates that our method accurately calculates an estimated distance.

Example 9. For the graphs x and y in Figure 8, consider a vertex mapping $\{u_1 \to v_1, u_2 \to v_2, u_3 \to v_3, u_4 \to v_4\}$ is given. The existing distance in the mapping is 1 (due to the label difference between u_2 and v_2). The estimated distance can be calculated in the following two different ways, where the first is the technique used in the previous work while the second is ours.

(1) If we use the label difference of unmapped parts of x and y, we calculate an estimated distance 1 because x has one more single bond (u_5, u_7) in its unmapped part.
(2) If we use the bridge method, the number of edit errors in the bridges is 2 because $\mathcal{B}_e(u_2, v_2) = 1$ and $\mathcal{B}_e(u_4, v_4) = 1$. By using the label difference of the unmapped vertices and the unmapped edges except the bridges, we get an additional edit error 1. By adding the two distances from the bridge method and label filtering, the estimated distance becomes 3.

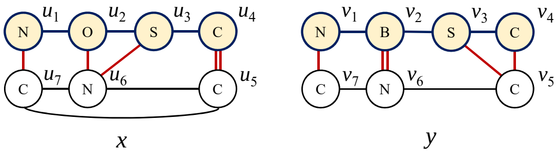


Figure 8: Estimating distance using bridges

The second method is to reorder vertices of the graph x (Line 1 in Algorithm 1). Similar to the problem of subgraph isomorphism, a proper vertex order is also crucial in the GED computation. Since most candidates generated from the filtering phase are false positives, we limit our discussion here to a false positive only (i.e., $ged(x, y) > \tau$). In general, as a vertex mapping M contains more edit errors, the search space of A* algorithm is reduced. For example, consider all edit errors in M and there is no edit error in the remaining vertices and edges. In this case, the A* algorithm can abandon M immediately. To make M contain as many edit errors as possible, therefore, we first consider vertices and edges causing edit errors by placing vertices in the mismatching partitions in the front positions. Since our partitioning method makes the size of a mismatching partition as small as possible, the A* algorithm accurately identifies many edit errors at higher levels of the state-space tree. It is worth noting that our first method is essential to detecting edit errors in those mismatching partitions isolated due to edit errors in bridges.

```
Algorithm 5: GEDVertexOrder($\mathcal{P}(x)$)
  input  : $\mathcal{P}(x)$ is the partitioning result of $x$.
  output : $O$ is an ordered set of vertices in $x$
1 $O \leftarrow \emptyset$;
2 foreach mismatch partition $p \in \mathcal{P}(x)$ do  $O \leftarrow O \cup V_p$;
3 DetermineVertexOrdering($O$);
4 $O \leftarrow O \cup V_x \backslash O$;
5 Return $O$;
```

Another consideration is the connectivity among vertices traversed by the A* algorithm. To reduce the search space, it is important to select the next vertex (Line 8 of Algorithm 1) that is connected to a vertex in M. Algorithm 5 is the vertex ordering algorithm. It first pushes vertices in mismatching partitions to O (Lines 1–2). Then, it orders the vertices in O using DetermineVertexOrdering, which traverses the vertices as described in Section 3 (Line 3). It finally places the remaining vertices (i.e., vertices in a matching partition) at the end (Line 4).

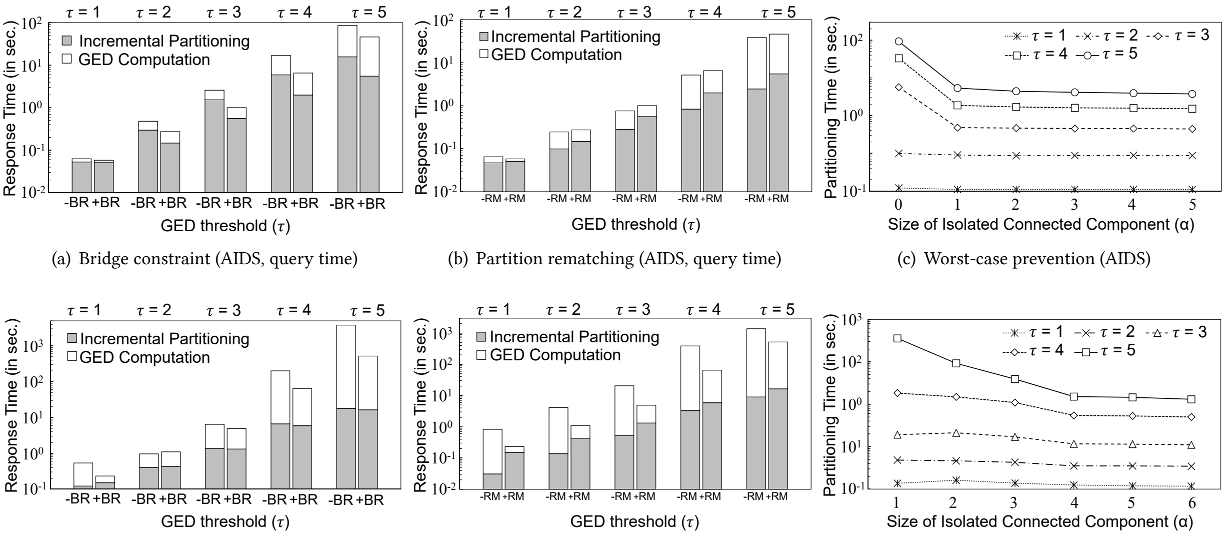
5 EXPERIMENTS

5.1 Experimental Setup

We used the following public real datasets.

- AIDS is an antiviral screen compound dataset containing 42, 687 chemical compounds, published by National Cancer Institute[2]. The dataset contains graphs with large size variation. It is a popular benchmark used in many graph search techniques.
- PROTEIN is a protein dataset from the Protein Data Bank[3], containing 600 protein structures. It contains denser and less label-informative graphs.
- PubChem is a chemical compound dataset from the PubChem Project[4]. We used a subset of PubChem consisting of 22, 794 chemical compounds. Graphs in the PubChem dataset contain repeating substructures and have less size variation compared with the AIDS and PROTEIN datasets.

[2] http://dtp.nci.nih.gov/docs/aids/aids_data.html
[3] http://www.iam.unibe.ch/fki/databases/iam-graph-database/download-the-iam-graph-database
[4] http://pubchem.ncbi.nlm.nih.gov, Compound_000975001_001000000.sdf



(a) Bridge constraint (AIDS, query time)

(b) Partition rematching (AIDS, query time)

(c) Worst-case prevention (AIDS)

(d) Bridge constraint (PubChem, query time)

(e) Partition rematching (PubChem, query time)

(f) Worst-case prevention (PubChem)

Figure 9: Evaluating optimization methods in Inves incremental partitioning

Table 3 summarizes the datasets, where N_{L_v} and N_{L_e} denote the number of distinct vertex and edge labels in the dataset, respectively.

Table 3: Statistics of datasets

Dataset	$\lvert\mathcal{D}\rvert$	$\lvert V\rvert_{\text{avg}}$	$\lvert E\rvert_{\text{avg}}$	$\lvert V\rvert_{\text{max}}$	$\lvert E\rvert_{\text{max}}$	N_{L_v}	N_{L_e}
AIDS	42,687	25.60	27.60	222	247	62	3
PROTEIN	600	32.63	62.14	126	149	3	5
PubChem	22,794	48.11	50.56	88	92	10	3

For the scalability test, we also used synthetic datasets (see Section 5.3). For each dataset, we conducted experiments using a workload of 100 queries graphs randomly sampled from the dataset[5]. Candidates requiring GED computation, query response time, incremental partitioning time, and GED computation time are measured and reported on the basis of these 100 queries.

All experiments were run on a machine with 32GB RAM, and an Intel core i7 at 3.4 GHz, running a 64-bit Ubuntu OS. We implemented Inves in C++, and compiled it using GCC 4.4.3 with the -O3 flag[6]. By scanning each dataset once, we pre-computed vertex and edge frequencies, label multisets of vertices and edges of each graph, and the label multiset of edges connected to each vertex. This pre-computation was performed offline and excluded from the query time.

5.2 Evaluating Optimization Methods in Inves

We first evaluated various optimization methods used in Inves. Since the technique is orthogonal to how candidates are generated, we scanned each dataset, and directly applied InvesVerifier on each graph to measure the query time. Notice that y-axis is log-scaled in all experiments.

5.2.1 Methods in Incremental Partitioning.

5.2.1 Methods in Incremental Partitioning. Figure 9(a) and (d) show the effect of the bridge constraint on the AIDS and PubChem datasets. -BR is the InducedSI without the bridge constraint in Corollary 2, and +BR denotes the case where the bridge constraint is added to InducedSI. +BR significantly improved the performance by utilizing differences of bridges connecting different partitions. For example, GED computation time was reduced by 2.4 times on the AIDS dataset and 3.3 times on the PubChem dataset when $\tau = 4$. The significant improvement can be explained by the number of candidates requiring GED computation. Table 4 shows the results on the AIDS and PubChem datasets. As shown in the table, +BR substantially reduced the size of the candidates requiring GED computation.

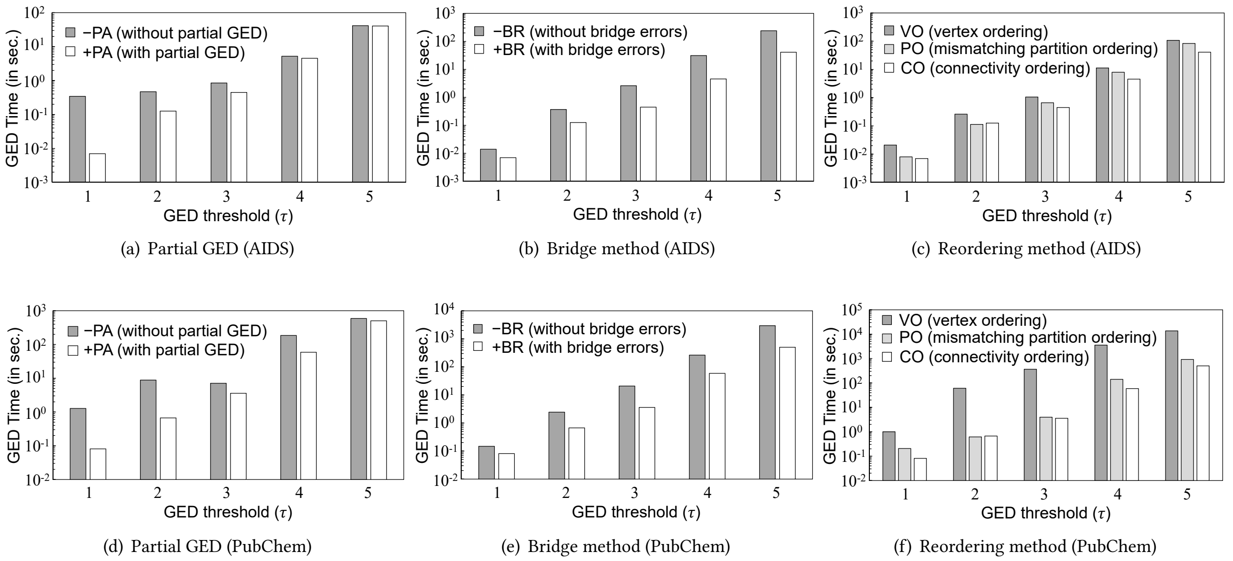
Table 4: Bridge constraint (number of candidates)

GED threshold (τ)		1	2	3	4	5
AIDS	−BR	135	517	4,686	21,780	60,443
	+BR	125	253	1,086	6,902	30,565
PubChem	−BR	226	397	1,138	5,827	25,354
	+BR	219	369	838	3,675	16,907

Figure 9(b) and (e) show the effect of the partition rematching method on the AIDS and PubChem datasets, where +RM and -RM denotes the results with and without the partition rematching method, respectively. On the PubChem dataset,+RM significantly reduced the GED computation time. For example, the GED time of +RM was about 10, 6, 5, 6, and 3 times faster than -RM for $\tau \in [1, 5]$, respectively. Table 5 shows the number of candidates requiring GED computation with and without the rematching method. Interestingly, the number of candidates reduced by +RM was smaller than that of +BR, while +RM achieved a higher performance gain on GED computation. This is because, although -BR does not use the bridge constraint, the bridge errors are considered in our GED algorithm and those false positives having bridge errors are quickly pruned when computing GED. It can also be explained by the size of mismatching partitions. Since the rematching method reduces the size of mismatching partitions,

[5] For the PubChem dataset, we manually replaced a few queries because existing techniques did not evaluate those queries in a reasonable amount of time.

[6] The source code of Inves is available at https://github.com/JongikKim/Inves



(a) Partial GED (AIDS)

(b) Bridge method (AIDS)

(c) Reordering method (AIDS)

(d) Partial GED (PubChem)

(e) Bridge method (PubChem)

(f) Reordering method (PubChem)

Figure 10: Evaluating optimization methods in Inves GED computation

A* algorithm can identify more edit errors at higher levels of the state-space tree and prune false positives early. Table 6 shows average sizes of mismatching partitions on the PubChem dataset.

Table 5: Partition rematching (number of candidates)

GED threshold (τ)		1	2	3	4	5
AIDS	−RM	129	279	1,451	8,807	36,265
	+RM	125	253	1,086	6,902	30,565
PubChem	−RM	223	384	951	4,184	18,322
	+RM	219	369	838	3,675	16,907

Table 6: Average mismatching partition size (PubChem)

GED threshold (τ)	1	2	3	4	5
−RM	4.49	17.93	27.34	33.79	37.73
+RM	1.78	9.13	18.28	26.5	31.69

On the AIDS dataset, however, +RM slightly degraded the overall performance because the GED computation on the AIDS dataset is much faster than that on the PubChem dataset and the rematching overhead of +RM is greater than the performance gain on GED computation. From the experiments, we observed that the rematching method should be used for steady performance even though it increases the partitioning time.

Figure 9(c) and (f) show the effect of the tunable parameter α described in Algorithm 2 in Section 3.3. As shown in the figure, tiny connected components greatly degraded the partitioning performance. Without the worst-case prevention method, the partitioning was extremely slow on the PubChem dataset and it did not finish in a reasonable amount of time. For this reason, we omit the result for $\alpha = 0$ in Figure 9(f). Based on the experiment, we used $\alpha = 1$ for the AIDS dataset and $\alpha = 4$ for the PubChem dataset. We performed similar experiments on the PROTEIN dataset, and chose $\alpha = 1$.

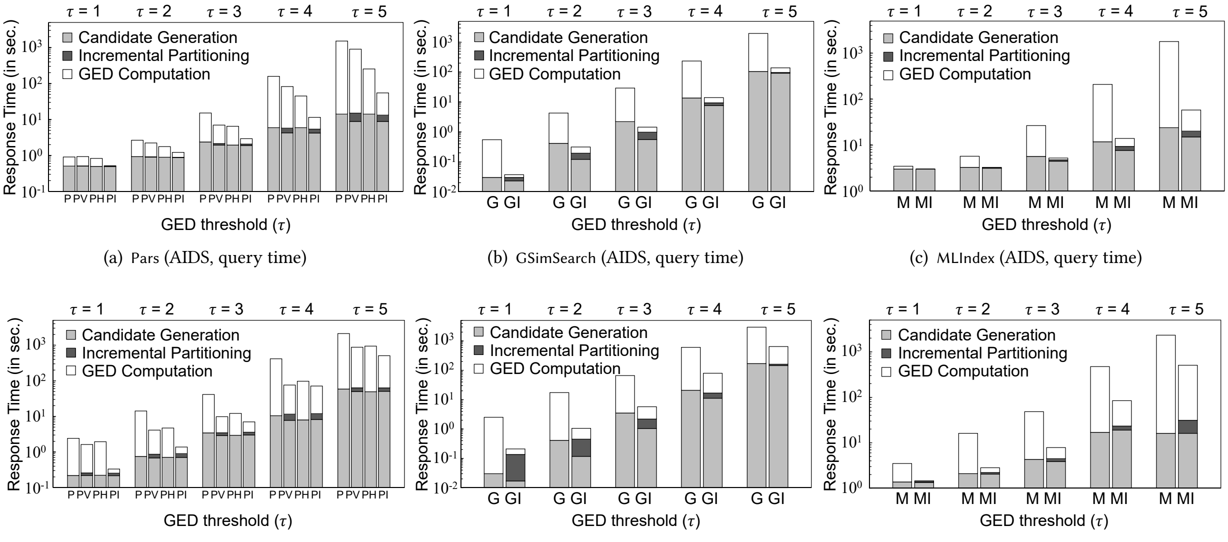
5.2.2 Methods in GED Computation. Evaluation results of the GED computation methods on the AIDS and PubChem datasets are shown in Figure 10. Figure 10(a) and (d) show the effect of the partial GED computation, where -PA and +PA denote InvesVerifier with and without this method, respectively. In our experiments, we observed that the tunable parameter β in InvesVerifier is relatively insensitive, and used $\beta = 0.7$ (we omit the results in the interest of space). +PA showed a good performance for low GED thresholds on both datasets. When a GED threshold was low, most candidates were answers, thus we had more chances to find an answer through a partial GED computation. As the threshold increased, however, most query time was spent on verifying false positives, and this method provided a marginal benefit only.

Figure 10(b) and (e) show the experimental results of our GED computation method with and without exploiting bridge errors, which are denoted by +BR and -BR, respectively. By precisely calculating edit errors in bridges, +BR reduced GED computation time up to 7 times on both datasets.

Figure 10(c) and (f) show the evaluation results of alternative vertex orderings. VO, PO, and CO denote the original vertex order, the vertex order considering vertices in mismatching partitions first, and the vertex order considering connectivity of vertices in Algorithm 5, respectively. PO significantly outperformed VO on both datasets. For example, PO is about 1.5 to 3 times faster than CO on the AIDS dataset. The performance on the PubChem dataset was extremely poor when VO was used. For example, VO was about 100 times slower than PO when $\tau = 3$. This is because the dataset contains graphs having repeating substructures, and thus the A* algorithm cannot efficiently prune the state-space tree without a proper vertex ordering. Considering the connectivity of vertices in different partitions, CO exhibited best performance for all the GED thresholds on both datasets.

5.3 Improving existing techniques

In this experiment, we evaluated how Inves can be adopted by existing techniques to improve their performance. We chose three representative techniques:



(a) Pars (AIDS, query time)

(b) GSimSearch (AIDS, query time)

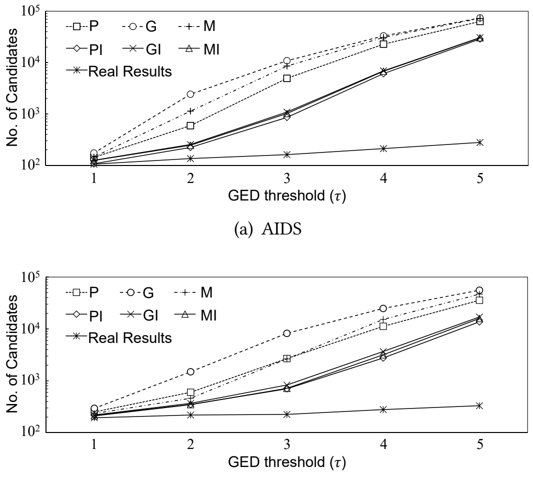
(c) MLIndex (AIDS, query time)

(d) Pars (PubChem, query time)

(e) GSimSearch (PubChem, query time)

(f) MLIndex (PubChem, query time)

Figure 11: Improvement of existing techniques



(a) AIDS

(b) PubChem

Figure 12: Number of candidates

(1) GSimSearch, labeled as G in figures, is a path-based q-gram approach proposed in [25, 26]. According to the results of [25], we used $q = 4$ for the AIDS dataset and $q = 3$ for the PROTEIN dataset. We conducted experiments on gram lengths for the PubChem dataset, and used $q = 4$ based on the results.

(2) Pars, labeled as P, is the state-of-the art partition-based approach [24]. For best performance, we improved Pars in the following ways. Since sequential scanning of indexed partitions was very slow, we implemented an index access method by modifying the SwiftIndex [16]. After we generated candidates using the index access method, we applied their recycling subgraph isomorphism test to each candidate. We also improved its verification process by using the local label filtering and the vertex ordering based on mismatching q-grams proposed in GSimSearch.

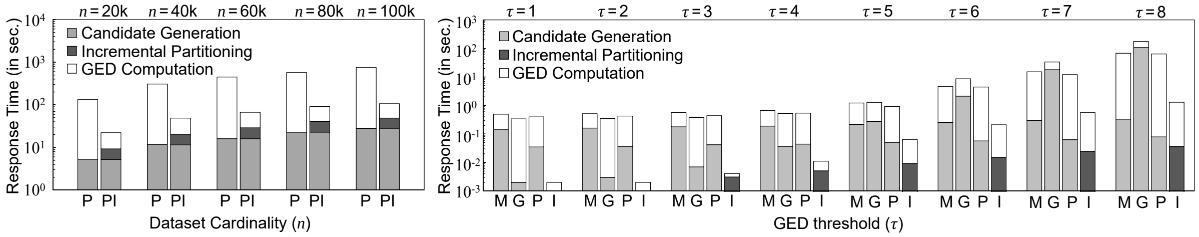
(3) MLIndex, labeled as M, is a multi-layered index technique proposed in [12]. We also improved MLIndex in the same way as Pars.

For each of the three implemented techniques, we adopted Inves in their verification phase. We labeled the corresponding improved techniques as PI, GI, and MI, respectively. We did not include other existing techniques such as c-star [22], SEGOS [20] and k-AT [19] since GSimSearch and Pars consistently outperformed these techniques [24–26]. We also excluded Mixed [27] because its performance results showed no significant differences with the performance of GSimSearch as reported in [3].

Figure 11 shows the results on the AIDS and PubChem datasets (see Figure 13(b) for the results of the PROTEIN dataset). In Figure 11(a) and (d), PV and PH denote Pars with the proposed verifier (without our GED computation methods), and Pars with the bridge method for GED (without the incremental partitioning), respectively. As shown in Figures 11, Inves improved the performance of all these three existing techniques by up to an order of magnitude. The significant improvement can be explained by the results of PV, PH, and PI in Figure 11(a) and (d). Both our partitioning and GED computation methods significantly reduced the total search time. When the incremental partitioning and GED computation methods are used together, the overall performance improvement was even more. In Figure 11(a), for example, when $\tau = 4$, PV was about 2 times faster than P, and PH was about 4 times faster than P. PI was about 25 times faster than P. Similar results were also observed on GSimSearch and MLIndex. Another important indicator of the improvement is the number of candidates requiring GED computation. As shown in Figures 12, Inves generated much smaller sets of candidates for the AIDS and PubChem datasets.

Scalability tests: We also evaluated the scalability of the proposed technique, on both the PROTEIN dataset and synthetic datasets generated by a graph generator[7]. The generator measured the graph size in terms of the number of edges ($|E|$), and

[7] GraphGen (http://www.cse.ust.hk/graphgen/) is a popular graph generator widely used in related work (e.g., [12, 24, 26]) for scalability tests.



(a) Dataset cardinality (Synthetic, $\tau = 3$)

(b) GED threshold (PROTEIN)

Figure 13: Evaluating scalability

the density of a graph defined as $d = 2|E|/|V|(|V| - 1)$. We used $|E| = 20$ and $d = 0.3$ for the experiments, which were default values of the generator. The cardinality of vertex and edge label domains were set to 2 and 1, respectively. Figure 13(a) shows the results. To evaluate the scalability, we generated five synthetic datasets consisting of 20k, 40k, 60k, 80k, and 100k graphs. In the experiments, 100 query graphs were randomly sampled from each dataset and the GED threshold was fixed as 3. The query time grew steadily and the growth ratios of query times were similar in P and PI.

Figure 13(b) shows the scalability results of the GED threshold. Because the PROTEIN dataset contains dense graphs, we chose the dataset to increase τ up to 8. Since the dataset contains 600 graphs only, we separately ran Inves, denoted by I in the figure, by scanning all the data graphs. As shown in the figure, Inves scaled best in terms of response time and outperformed existing techniques by up to about 65 times.

6 CONCLUSIONS

In this paper, we developed a novel technique called Inves for verifying if the graph edit distance (GED) between two graphs is within a threshold, an important and expensive step in graph similarity search. Its main idea is to judiciously and incrementally partition a candidate graph based on the query graph, and use the results to compute a lower bound of their distance. If a full GED computation is needed, Inves utilizes the collected information, and uses novel methods and an A* algorithm to search in the space of possible vertex mappings between the graphs to compute their GED efficiently. We presented a full specification of the technique, and conducted extensive experiments on both real and synthetic datasets. The results showed that the technique can significantly improve the performance of existing techniques [12, 24–26] by an order of magnitude.

ACKNOWLEDGMENTS

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2018-2015-0-00378) supervised by the IITP (Institute for Information & communications Technology Promotion). Dong-Hoon Choi was partially supported by ETRI R&D Program ("Development of Big Data Platform for Dual Mode Batch-Query Analytics, 14Z1400") funded by the Government of Korea. Chen Li has been partially supported by NSF award III 1745673.

REFERENCES

[1] A. Ahmadi, A. Behm, N. Honnalli, C. Li, and X. Xie. 2012. Hobbes: Optimized gram-based methods for efficient read alignment. *Nucleic Acids Res.* 40 (2012), e41.

[2] H. Bunke and K. Shearer. 1998. A graph distance metric based on the maximal common subgraph. *Pattern Recogn. Lett.* 19, 3-4 (1998), 255–259.

[3] X. Chen, H. Huo, J. Huan, and J. S. Vitter. 2017. Efficient graph similarity search in external memory. *IEEE Access* 5 (2017), 4551–4560.

[4] A. Döring, D. Weese, T. Rausch, and K. Reinert. 2008. SeqAn an efficient, generic c++ library for sequence analysis. BMC Bioinformatics. *BMC Bioinformatics* 9 (2008), 11.

[5] A. Fischer, C. Y. Suen, V. Frinken, K. Riesen, and H. Bunke. 2015. Approximation of graph edit distance based on Hausdorff matching. *Pattern Recognition* 48, 2 (2015), 331 – 343.

[6] K. Gouda and M. Arafa. 2015. An improved global lower bound for graph edit similarity search. *Pattern Recogn. Lett.* 58 (2015), 8–14.

[7] W.-S. Han, J. Lee, and J.-H. Lee. 2013. Turbo$_{\text{ISO}}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD Conference.* 337–348.

[8] J. Kim. 2015. An effective candidate generation method for improving performance of edit similarity query processing. *Information Systems* 41 (2015), 116–128. Issue 1.

[9] J. Kim, C. Li, and X. Xie. 2014. Improving read mapping using additional prefix grams. *BMC Bioinformatics* 15 (2014), 42.

[10] J. Kim, C. Li, and X. Xie. 2016. Hobbes3: Dynamic generation of variable-length signatures for efficient approximate subsequence mappings. In *ICDE.* 169–180.

[11] G. Li, D. Deng, J. Wang, and J. Feng. 2011. Pass-Join: A Partition based method for similarity joins. *PVLDB* 5, 3 (2011), 253–264.

[12] Y. Liang and P. Zhao. 2017. Similarity search in graph databases: a multi-layered indexing approach. In *ICDE.*

[13] J. Munkres. 1957. Algorithms for the assignment and transportation problems. *J. SIAM* 5 (1957), 32–38.

[14] K. Riesen, S. Fankhauser, and H. Bunke. 2007. Speeding up graph edit distance computation with a bipartite heuristic. In *MLG.*

[15] H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang. 2010. Connected substructure similarity search. In *SIGMOD Conference.* 903–914.

[16] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB* 1, 1 (2008), 364–375.

[17] Y. Tian, R. C. Mceachin, C. Santos, D. J. States, and J. M. Patel. 2007. SAGA: A subgraph matching tool for biological graphs. *Bioinformatics* 23, 2 (2007), 232–239.

[18] J. R. Ullmann. 1976. An algorithm for subgraph isomorphism. *J. ACM* 23, 1 (1976), 31–42.

[19] G. Wang, B. Wang, X. Yang, and G. Yu. 2012. Efficiently indexing large sparse graphs for similarity search. *IEEE Trans. on Knowl. and Data Eng.* 24, 3 (2012), 440–451.

[20] X. Wang, X. Ding, A. K. H. Tung, S. Ying, and H. Jin. 2012. An efficient graph indexing method. In *ICDE.* 210–221.

[21] C. Xiao, W. Wang, and X. Lin. 2008. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB* 1, 1 (2008), 933–944.

[22] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou. 2009. Comparing stars: On approximating graph edit distance. *PVLDB* 2, 1 (2009), 25–36.

[23] S. Zhang, J. Yang, and W. Jin. 2010. SAPPER: Subgraph indexing and approximate matching in large graphs. *PVLDB* 3, 1 (2010), 1185–1194.

[24] X. Zhao, C. Xiao, X. Lin, Q. Liu, and W. Zhang. 2013. A partition-based approach to structure similarity search. *PVLDB* 7, 3 (2013), 169–180.

[25] X. Zhao, C. Xiao, X. Lin, and W. Wang. 2012. Efficient graph similarity join with edit distance constraints. In *ICDE.* 834–845.

[26] X. Zhao, C. Xiao, X. Lin, W. Wang, and Y. Ishikawa. 2013. Efficient processing of graph similarity queries with edit distance constraints. *The VLDB Journal* 22, 6 (2013), 727–752.

[27] W. Zheng, L. Zou, X. Lian, D. Wang, and D. Zhao. 2015. Efficient graph similarity search over large graph databases. *IEEE Trans. on Knowl. and Data Eng.* 27, 4 (2015), 964–978.

[28] G. Zhu, X. Lin, K. Zhu, W. Zhang, and J. X. Yu. 2012. TreeSpan: Efficiently computing similarity all-matching. In *SIGMOD Conference.* 529–540.

Flow Motifs in Interaction Networks

Chrysanthi Kosyfaki
Dept. of Computer Science and Engeenering
University of Ioannina, Greece
xkosifaki@cs.uoi.gr

Nikos Mamoulis
Dept. of Computer Science and Engeenering
University of Ioannina, Greece
nikos@cs.uoi.gr

Evaggelia Pitoura
Dept. of Computer Science and Engeenering
University of Ioannina, Greece
pitoura@cs.uoi.gr

Panayiotis Tsaparas
Dept. of Computer Science and Engeenering
University of Ioannina, Greece
tsap@cs.uoi.gr

ABSTRACT

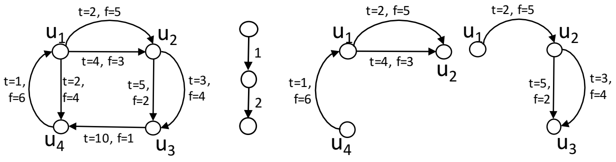
Many real-world phenomena are best represented as interaction networks with dynamic structures (e.g., transaction networks, social networks, traffic networks). Interaction networks capture flow of data which is transferred between their vertices along a timeline. Analyzing such networks is crucial toward comprehending processes in them. A typical analysis task is the finding of motifs, which are small subgraph patterns that repeat themselves in the network. In this paper, we introduce *network flow motifs*, a novel type of motifs that model significant flow transfer among a set of vertices within a constrained time window. We design an algorithm for identifying flow motif instances in a large graph. Our algorithm can be easily adapted to find the top-k instances of maximal flow. In addition, we design a dynamic programming module that finds the instance with the maximum flow. We evaluate the performance of the algorithm on three real datasets and identify flow motifs which are significant for these graphs. Our results show that our algorithm is scalable and that the real networks indeed include interesting motifs, which appear much more frequently than in randomly generated networks having similar characteristics.

1 INTRODUCTION

Interaction networks include a large number of highly connected components that dynamically exchange information. Examples of such graphs are neural networks, food webs, signal transfer pathways, the bitcoin network, social networks, and traffic networks. An interaction network captures *flow of data* (e.g., money, messages, passengers, etc.) which is transferred between its vertices along a timeline. In such a network, there could be multiple edges connecting the same pair of vertices, modeling data exchange between them at different times. Figure 1(a) shows a small example of an interaction network, where the vertices represent users who exchange money. The edges are annotated by timestamped interactions; e.g., edge u_1u_2 with label $t = 2, f = 5$ denotes that user u_1 sent 5 units of flow (money) to user u_2 at time 2.

Interaction networks are a powerful and versatile model, and as such they have been studied extensively in the literature [12, 23, 24]. In this paper, we consider the problem of finding small characteristic patterns in the networks, such as chains, triangles or cycles. These patterns are called *network motifs*. A motif is a subgraph that appears significantly more often in a real network than in a randomized network with similar characteristics [15]. Finding motifs is a method of identifying functional properties of a network. Previous work mainly focused on static motif patterns [15, 21]. Recently, there has been increasing interest in analyzing temporal networks [6, 8, 9, 17, 23, 24], where edges carry timestamps that signify the time of interaction between vertices. However, to the best of our knowledge, there is no previous work on motif search that considers the flow of data between connected nodes. Motivated by this, we define the concept of *flow motifs* in temporal interaction networks and study their identification.

Our definition of flow motifs extends a well-accepted definition of temporal motifs [17]. We define flow motifs as small graphs whose edges are ordered; the order defines how the data flows between the vertices. An instance of the motif is a subgraph of the interaction network, whose edges obey the total order specified by the edges of the motif. Moreover, the time difference between the temporally last and first edges should not exceed a pre-defined threshold δ which is a parameter of the motif. These requirements are the same as in the temporal motif definition of [17], which however disregards the data flow in interactions. The distinctive feature of our flow motifs is that, in a flow motif instance, multiple edges of the graph can instantiate a single edge of the motif, if they satisfy the order constraint with the edges that instantiate the motif's previous and next edges. The flow values in the edge-set that instantiates a motif edge are *aggregated* to a single value, which captures the *total flow* passing through the motif edge. The *minimum aggregated flow* at any motif edge defines the flow of the instance. In order for the instance to be valid, we require that its flow exceeds a threshold ϕ.



(a) multigraph (b) motif (c) instance 1 (d) instance 2

Figure 1: Example of graph, motif, and instances

Consider again the interaction network of Figure 1(a). Assuming that the motif of interest is a chain of three nodes (Figure 1(b)), where the labels in edges specify the flow order and that $\delta = 5$ and $\phi = 5$, the two subgraphs of Figures 1(c) and 1(d) are instances of the motif because the sets of edges mapped to each motif edge satisfy (i) the time order constraint of the motif and (ii) thresholds δ and ϕ. For example, in Figure 1(d), both edges that connect u_2 to u_3 are temporally after the edge that connects u_1 to u_2 and their aggregated flow is 6 ($\geq \phi$); in addition, the

Series ISSN: 2367-2005

10.5441/002/edbt.2019.22

time difference between the temporally first and last edges in the instance is $5 - 2 = 3$ ($\leq \delta$).

Overall, a valid flow motif instance should satisfy three requirements: (a) a *structural constraint*, defined by the graph structure of the motif; (b) a *temporal constraint* defined by the temporal window size δ; (c) a *flow constraint* defined by the minimum flow value ϕ.

Flow motifs correspond to frequently occurring sub-structures with high activity that appear in short time windows. Finding instances of flow motifs is of great importance in understanding interaction networks. For instance, in networks that model money transfers, flow motifs correspond to transaction patterns involving significant flow of money that appear more frequently than expected. Flow motif search is of particular interest to Financial intelligent units (FIUs); these are organizations which identify suspicious flow patterns that may suggest criminal behavior (e.g., money laundering). Belize FIU (fiubelize.org) and Hong Kong's JFIU (www.jfiu.gov.hk) indicate as suspicious patterns which include 'smurfing' (i.e., numerous small-volume transfers which aggregate to large amounts), cyclic transactions between parties, and chains of significant money transfers within limited time (e.g., payments out which are paid in on the same or on the previous day). In addition, bitcoin theft has been associated to flow patterns in [14]. In communication and social networks, flow motifs may reveal common patterns of influence [3, 11]. For example, the strength of the relationships between two social network users is correlated with the frequency of online interactions between them [22]. This implies that groups of users with frequent communication between them within a short period have high chance to influence each other.

Given a large interaction network, we propose an algorithm that takes as input a flow motif and efficiently finds its instances in the network. Our algorithm operates in two phases. First, the structural matches of the motif (disregarding temporal and flow information) are identified. Then, for each structural match, we find the motif instances which satisfy the temporal and flow constraints. This is achieved by sliding a time window of the same length as the duration constraint of the motif and systematically finding the combinations of edges that constitute motif instances. Compared to motif search algorithms from previous work, our algorithm is novel in that it considers the aggregated flow on multiple edges that connect the same pair of nodes in the network during the construction of the motif instances. Due to the large number of possible edge combinations, the problem is harder compared to finding instances of motifs, by disregarding flows and multiple edges. Our algorithm effectively uses the duration and flow constraints to prune the space. We also suggest a variant of the algorithm that identifies the top-k instances of an input flow motif with the highest flow. Finally, we propose a dynamic programming module for the algorithm, for the problem of finding the motif instance with the maximum flow.

We evaluate the performance of the algorithm on three real datasets of different nature (bitcoin user network, facebook network, and Passenger flow network). We compare the performance of our algorithm to a baseline method which builds up motif instances by joining their components and demonstrate the superiority of our approach against this alternative method. We also show that our tested flow motifs indeed appear more frequently in real networks than in randomized networks having the same characteristics as the real ones.

In summary, this paper makes the following contributions:

- We propose the novel concept of flow motif. To our knowledge, this is the first work that defines and studies the search of flow motifs in interaction networks.
- We propose an efficient algorithm for finding flow motif instances in large interaction networks and variants of it that identify the instances of a motif with the maximum flow.
- We evaluate our approach using three real datasets, and demonstrate that it scales well for large data.
- We investigate the significance of the tested motifs in the real networks.

The rest of the paper is organized as follows. Section 2 describes work related to network flow motifs, which are then formally defined in Section 3. Our motif search algorithm is presented in Section 4. Section 5 shows how to extend our algorithm to find the k instances of a given motif with the maximum flow. In Section 6, we experimentally evaluate our algorithm and the significance of the motifs by using a randomization approach. Finally, in Section 7, we conclude our paper and give directions for future work.

2 RELATED WORK

There has been a lot of research interest in motif search and mining in interaction networks [5, 20, 21]. In this section we summarize the most representative works in static and temporal networks.

Static Networks. Milo et al. [15] introduced the concept of motifs and studied their identification in large graphs. They defined a network motif as a *"pattern of interconnections occurring in complex networks at numbers that are significantly higher than those in randomized networks"*. They investigated motif discovery in directed networks, which do not carry temporal information (i.e., the motifs do not consider the time when the interactions took place).

FANMOD [21] is an efficient tool for finding network motifs in static networks, up to a size of eight vertices. Given a subgraph size, the tool either enumerates all subgraphs of that size or samples them uniformly. The identified subgraphs are grouped into classes based on their isomorphism. The significance of each class is finally measured by counting their frequencies in a number of random graphs (generated by swapping edges between vertices in the original network).

Temporal Networks. In *temporal networks*, the interactions between vertices are labeled by the time when they happen. Fundamental definitions, concepts, and problems on temporal networks are given in [6]. For instance, the concept of *time-respecting path* and its relation to network flows are defined and studied here.

Paranjape et al. [17] define motifs in temporal networks as small connected graphs, whose edges are temporally ordered. Instances of a motif are subgraphs that structurally match the motif and their edges obey the order. In addition, the time-difference between the temporally last and the first edges should not exceed a motif *duration* constraint δ. They propose a general algorithmic framework for computing the number of motif instances in a graph and fast algorithms that count certain classes of temporal motifs. Our network flow motifs are similar to the temporal motifs of [17], however, in our case (i) a motif edge can be instantiated by multiple edges of the graph and (ii) we introduce and consider a minimum flow requirement.

Another work that defines and studies the enumeration of temporal motifs is [8]. In the context of this work, the interactions between vertices are not instantaneous but they carry a duration interval. Motifs are again subgraphs whose edges are temporally ordered. As opposed to [17], there is no δ threshold between the last and the first edge in a motif instance. Instead, a maximum time-difference Δt between consecutive edges in a motif instance is allowed.

Rocha et al. [2] also define motifs that model the information spread in temporal networks. They study the impact of time ordering information by comparing the instances of the motifs by considering or not the temporal order. The flow motifs defined and used in [2] are different to ours, because in our case (i) we consider the flow on edges (ii) we define the flow in a motif differently and (iii) our input graph and the motif instances are multigraphs.

Communication motifs are suggested as a model for capturing the structure of human interaction in networks over time. Zhao et al. [23] studied the evolution of such behavioral patterns in social networks. For any two adjacent interactions, the term *maximum flow* is used to characterize those interactions that are the most probable to belong to the same information propagation path among any such adjacent interactions. On the other hand, in our context, flow refers to the data (e.g., money, messages, etc.) being transferred from one node along network paths. Another work that studies behavioral patterns in social networks by defining and mining communication motifs between people in social networks is [4]. A scalable mining technique (called COMMIT) for such communication motifs is proposed.

A recent work that studies the structure of social networks and the temporal relations between entities in them is [24]. Temporal pattern search is proposed as a tool in this direction. In order to facilitate the efficient retrieval of pattern instances, occurrences of small patterns are precomputed and indexed.

Flow can also be used to describe other concepts. In [9], the authors study the information propagation problem. They try to identify all time-respecting paths in temporal networks to model potential pathways for information spread. Our work is diffent in that (i) we are interested in specific motifs and (ii) we consider the flow on edges. The identification of time-respecting paths (as defined in [9]) that form cycles is studied in [10], where an efficient algorithm (2SCENT) is proposed.

Motif discovery in Heterogeneous Information Networks (HiNs) which carry temporal information was also recently studied [12]. In such graphs, some nodes are associated to events (which happened at a specific time). A motif is then defined by a graph and a maximum temporal difference between the events that instantiate its event nodes. As in the rest of previous work, any data flow on the edges of the network is disregarded in the definition and search of motifs.

3 DEFINITIONS

In this section, we formally define flow motifs and the graph wherein they are identified. Table 1 shows the notations used frequently in the paper.

The input to our problem is a directed multigraph $G(V, E)$, where each pair of nodes $u, v \in V$ can be connected by any number of edges in E. We denote by $E(u, v)$ the edge-set from $u \in V$ to $v \in V$. Each edge $e \in E$ is annotated by a unique *timestamp* $t(e)$ in a continuous time domain $\mathcal{T}$ and a positive real number $f(e)$, called *flow*.

Table 1: Table of notations

Notations	Description
$G_M(V_M, E_M)$	graph structure of motif M
δ	duration constraint of a motif
ϕ	flow constraint of a motif
$\ell(e)$	order of edge e in a motif M
SP_M	spanning path of motif M
e_i or $SP_M[i]$	i-th edge of motif M
$SP_M[i:j]$	subpath $e_i \dots e_j$ of SP_M
$G(V, E)$	input graph
$E(u, v)$	set of edges in G from u to v
$f(e)$	flow on edge e
$t(e)$	timestamp of edge e
$f(G_I)$	flow of motif instance G_I
$G_T(V, E_T)$	time-series graph equivalent to $G(V, E)$
(t, f)	flow interaction element on an edge of E_T
$R(u, v)$	time series on edge $(u, v) \in E_T$
$R(e_i)$	time series on edge of E_T mapped to e_i
S	set of structural matches of a motif
G_s	structural match of a motif

Figure 2 shows an example of an input graph G from a real application, where vertices correspond to users (addresses) of the bitcoin network and edges correspond to transactions between them. Each edge is annotated by the timestamp of the transaction followed by the transaction amount. For example, user u_1 at timestamps 13 and 15 sent 5 and 7 bitcoins, respectively, to u_2.

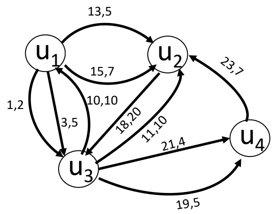


Figure 2: Example of an interaction graph (bitcoin user graph)

Definition 3.1 (Flow Motif). A network flow motif M is a triplet (G_M, δ, ϕ) consisting of (i) a directed graph $G_M(V_M, E_M)$ with $m = |E_M|$ edges, where each edge e is labeled by a unique number $\ell(e)$ in $[1, m]$; (ii) a value δ, which defines an upper-bound on the duration of the motif; and (iii) a value ϕ, which defines a lower bound on the flow of the motif.

The labels of the edges in the motif graph G_M define a total order of the edges that models the direction of the flow in G_M. For example, if G_M consists of two edges (u, v) and (v, w) and we have $\ell(u, v) = 1$ and $\ell(v, w) = 2$, this means that the flow in the graph originates from node u, it is first transferred to v, and then from v to w.

Figure 3 shows some examples of motifs (we only show the motif graphs G_M, but not the thresholds δ and ϕ). The numbers in the parentheses denote the number of nodes and edges in the motifs. For example, the motif labeled $M(3, 3)$ models a cyclic flow between three nodes.

We assume that the ordering of the edges according to their labels defines a *path* in the graph G_M. We refer to this path as the *spanning path* of the motif, and we denote it as SP_M. The spanning path is not necessarily a simple path, i.e., there may be repeated vertices in the path. We sometimes refer to a motif graph

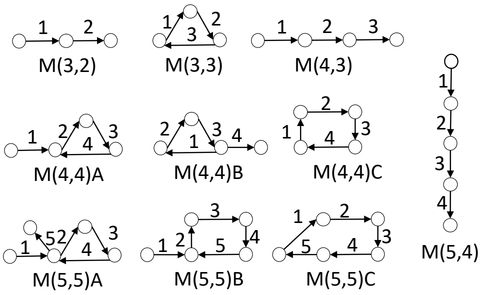
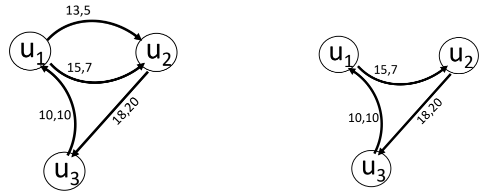


Figure 3: Examples of motifs.



(a) maximal instance

(b) non-maximal instance

Figure 4: Examples of motif instances

G_M by its spanning path $SP_M = e_1e_2 \dots e_m$, i.e., the total order of its edges, where e_i denotes the edge with label i. For example, we may refer to motif $M(3,3)$ in Figure 3 by the sequence $SP_{M(3,3)} = e_1e_2e_3$ of its three edges. In addition, we use e_i or $SP_M[i]$ to denote the i-th edge of the motif, and $SP_M[i:j]$ to denote the subsequence of edges $e_i \dots e_j$ along the path. We now define motif instances as follows.

Definition 3.2 (Flow Motif Instance). An instance of a motif $M = (G_M, \delta, \phi)$ in the graph $G(V, E)$ is a subgraph, $G_I(V_I, E_I)$, $V_I \subseteq V$, $E_I \subseteq E$ of G with the following properties:

- There is a bijection $\mu : V_M \rightarrow V_I$ from the vertex set of the motif graph V_M to the instance vertex set V_I.
- For every edge $(u,v) \in E_M$ there is a non-empty set of edges $E_I(\mu(u), \mu(v))$ in G_I, such that $E_I(\mu(u), \mu(v)) \subseteq E(\mu(u), \mu(v))$. In addition, $E_I = \bigcup_{(u,v) \in E_M} E_I(\mu(u), \mu(v))$.
- The edge-sets in G_I are *time-respecting*: For every pair of edges (u,v) and (v,w) in E_M, if $l(u,v) < l(v,w)$, then for every pair of edges $e_i \in E_I(\mu(u), \mu(v))$, $e_j \in E_I(\mu(v), \mu(w))$, $t(e_i) < t(e_j)$.
- The maximum time difference between any two edges in E_I is at most δ.
- The sum of flows of any edge-set in E_I is at least ϕ.

The first two conditions express a structural requirement on the matching subgraph, the third and fourth conditions temporal constraints, and the last condition a minimum flow constraint. Figure 4(a) shows an instance of $M(3,3)$ in the graph of Figure 2, assuming that $\delta = 10$ and $\phi = 7$. u_3, u_1, and u_2 are mapped to the first, second, and third node of $M(3,3)$ according to the order of its edges. u_1 and u_2 in the instance are linked by two edges which are both temporally after the edge(s) that link u_3 to u_1 and before the edge(s) that link u_2 to u_3. The maximum time difference between any two edges is 8 ($\leq \delta$) and the aggregate flows on $E_I(u_3, u_1)$, $E_I(u_1, u_2)$, and $E_I(u_2, u_3)$ are 10, 12, and 20, respectively (i.e., each of them is at least ϕ). If we denote $M(3,3)$ by its spanning path $SP_{M(3,3)} = e_1e_2e_3$, we can express the instance of Figure 4(a) by $[e_1 \leftarrow \{(10,10)\}, e_2 \leftarrow \{(13,5),(15,7)\}, e_1 \leftarrow \{(18,20)\}]$.

For the ease of exposition, we define the flow $f(G_I)$ of an instance G_I of motif M as the minimum total flow among all edge-sets $E_I(\mu(u), \mu(v))$ which instantiate the edges (u,v) of M. Formally:

$$f(G_I) = \min_{(u,v) \in E_M} \sum_{e \in E_I(\mu(u), \mu(v))} f(e) \qquad (1)$$

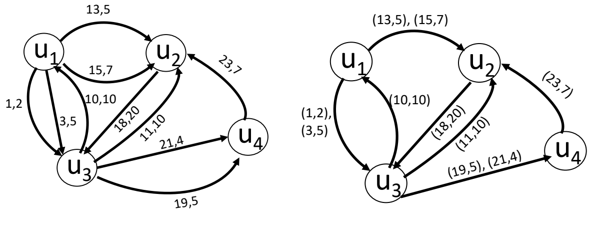
We now define the concept of motif instance maximality.

Definition 3.3 (Instance Maximality). An instance $G_I(V_I, E_I)$ of a motif $M = (G_M, \delta, \phi)$ is maximal iff, the addition of one more edge to any edge-set $E_I(\mu(u), \mu(v))$ of G_I from the corresponding edge-set $E(\mu(u), \mu(v))$ of G violates the duration or flow constraints of the motif.

For example, assuming that $\delta = 10$ and $\phi = 7$, Figure 4(b) shows an instance of $M(3,3)$ in the graph of Figure 2, which is not maximal. This is because the addition of edge (13,5) to $E_I(u_1, u_2)$ results in the valid instance of Figure 4(a). In this paper, we focus on finding maximal instances of motifs only, because non-maximal ones are redundant and considering them can mislead towards the importance of a motif. For example, if $\phi = 0$, all combinations of subsets of the edge-sets that form a valid motif instance are also valid (but not maximal) instances. Considering them would exponentially increase the total number of motif instances, potentially over-estimating its importance.

4 FINDING FLOW MOTIF INSTANCES

We now present an efficient algorithm for enumerating the maximal instances of a given motif $M(V_M, E_M)$ in an input graph $G(V, E)$. For the ease of presentation, we consider the input graph G not as a temporal multi-graph, but as a graph where all original edges from a vertex $u \in V$ to a vertex $v \in V$ are *merged* to a single edge. The single edge (u,v) is associated with an *interaction time-series* $R(u,v) = \{(t_1, f_1), (t_2, f_2), \dots\}$. Each pair (t_i, f_i) represents a *flow interaction* occurring at time t_i with flow transfer f_i from u to v. The interaction time series is ordered in time. Figure 5 shows an example of how the edges of a multigraph G are merged to time series. For example, the two edges from u_1 to u_2 are considered as a single edge; the two edges with timestamps 13 and 15 are now considered as a time series on a single edge (u_1, u_2). The conversion of the multigraph to a graph does not have to be explicitly performed; for each connected pair of vertices, it suffices to consider their multiple edges ordered by timestamp. We will use $G_T(V, E_T)$ to denote this graph and we will refer to it as the *time series graph.*



(a) multigraph

(b) time series graph

Figure 5: From a multigraph to a time series graph

Our algorithm takes as input the multigraph $G(V, E)$ and a motif $M = (G_M, \delta, \phi)$, and finds all instances of M in G. The algorithm operates on the time series graph G_T and works in two phases P1 and P2:

P1 Find the set S of all *structural matches* of graph G_M in graph G_T, disregarding the labels on the edges and constraints δ and ϕ.

P2 For each $G_s \in S$, using the time series of the edges in G_s, find all instances of M in G_s (which should satisfy the duration and flow constraints defined by δ and ϕ).

We now elaborate on the two phases.

Phase P1: To illustrate phase P1, as an example, consider the graph G_T of Figure 5(b) and the motif $M(3,3)$ shown in Figure 3. Figure 6 shows all six structural matches of $M(3,3)$ in G_T found in phase P1. The labels $\{e_1, e_2, e_3\}$ on the edges of the matches indicate the edges of the motif on which they are mapped. For example, edge (u_1, u_2) of the first match is mapped to the first edge e_1 of the motif.

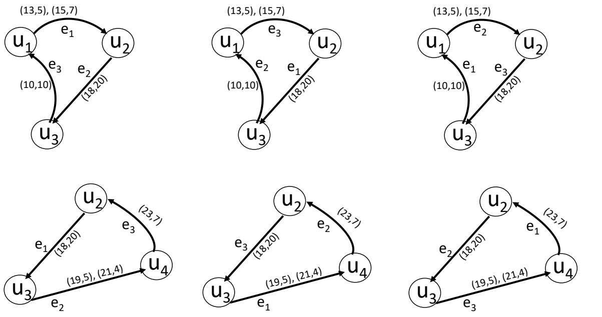


Figure 6: Structural matches of $M(3,3)$ (phase P1)

Algorithmically, for phase P1, any graph pattern matching algorithm for static graphs can be used (e.g., [21]). In our implementation, we exploit the fact that the ordering of the edges defines a path. Using a modified depth-first search algorithm on G_T, we can extract all paths of length $|E_M|$ that are structural matches of G_M in G_T. Specifically, in a loop, we map every node in G_T to the first node in G_M (i.e., the origin node of the first edge in G_M) and recursively find all paths that originate from that node and map them to the spanning path SP_M of G_M. For example, for motif $M(3,3)$, the depth-first search algorithm should make sure that the last vertex of the traversed path is the same as the first vertex of the path. Hence, the algorithm on the graph G of our running example would identify path $u_1u_2u_3u_1$ as a match of $M(3,3)$.

Phase P2: In phase P2, given the set of structural matches S, for each $G_s \in S$, we process the time series on the edges of G_s in order to find valid flow motif instances. In a nutshell, we slide a time window of length δ along the set of all (t_i, f_i) interactions on the edges of G_s; for all sets of interactions within δ time difference, we find all combinations thereof which constitute valid motif instances. Note that each structural match G_s from phase P1 may produce an arbitrary number of flow motif instances, as each time window position can generate different instances depending on the combinations of edge flows we use.

To illustrate, consider again $M(3,3)$ (for $\delta = 10$) and a possible structural match, shown in Figure 7. We will get different flow motif instances depending on whether we consider window $[10,20]$ or $[15,25]$. Furthermore, even for the specific time-window $[10,20]$, we can get different flow motif instances depending on how we combine the edges in this window. For example, one possible flow motif instance is $[e_1 \leftarrow \{(10,5)\}, e_2 \leftarrow \{(11,3),(16,3)\}, e_3 \leftarrow \{(19,6)\}]$, while another flow motif instance is $[e_1 \leftarrow \{(10,5)\}, e_2 \leftarrow \{(11,3)\}, e_3 \leftarrow \{(14,4),(19,6)\}]$. Note that the flow in the former case is 5, while in the latter is 3, meaning that the latter instance would be rejected for $\phi = 5$.

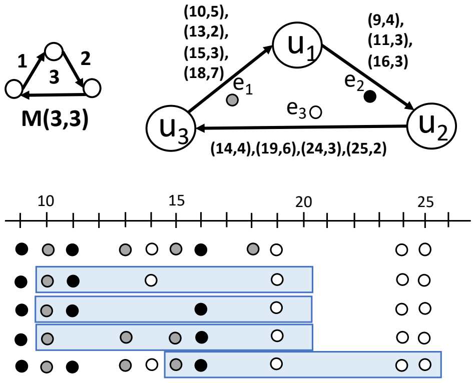


Figure 7: Example for Algorithm 1

Algorithm 1 is applied in phase P2 to find all instances of the motif M in a match G_s (found in phase P1). The algorithm slides a window T of length δ over the time domain, to find subsets of edges in G_s that satisfy the duration constraint δ and can generate maximal motif instances. Given a specific window T we run procedure FindInstances in order to generate all possible maximal flow-motif instances that satisfy the flow constraint ϕ. The procedure is recursive on the length m of the spanning path $SP_M = e_1e_2 \ldots e_m$ of the motif.

FindInstances takes as input the graph instance G_s, a spanning path SP, a time-window T and the threshold ϕ. Let $R(e_i)$ be the interaction time series on the edge of G_s which is mapped to edge e_i of the motif. If the spanning path consists of a single edge e_1, then the procedure finds the set $R_T(e_1) \subseteq R(e_1)$ of all elements in $R(e_1)$, which are within the time-window T, and aggregates their flow. If the total flow $f(R_T(e_i))$ of these elements satisfies the flow constraint ϕ, the edge-set of G corresponding to $R_T(e_i)$ becomes an instance of SP and it is returned. For longer spanning paths, the procedure considers again the first edge $e_1 = SP[1]$. For every prefix T_p of the window T that contains instances of the edge e_1, it computes the set $R_{T_p}(e_1) \subseteq R(e_1)$ of all (t, f) interaction elements in $R(e_1)$ for which $t \in T_p$. If $R_{T_p}(e_1)$ is non-empty and satisfies the flow constraint, then FindInstances is recursively called on the rest of the spanning path $SP_{next} = SP[2:m]$, with time window $T_{next} = T - T_p$. This recursive call will return the set of valid instances within time-window T_{next} for the sub-motif defined by SP_{next}. Each of these instances is *concatenated* to $R_{T_p}(e_1)$ to create a new valid instance for SP.

The condition at line 16 of the algorithm helps us to find invalid prefixes of the motif instances early. In other words, if a sub-series $R_{T_p}(e_1)$ which is candidate for instantiating a motif edge does not qualify ϕ, we do not consider the possible instances that include the elements of $R_{T_p}(e_1)$ as an instance of e_1. Hence, the search space is effectively pruned.

Figure 7 illustrates the functionality of Algorithm 1. On top, the figure shows motif $M(3,3)$ and a structural match G_s of it, where each edge is labeled by the time series of flows between the corresponding nodes (e.g., at time 10, u_2 sent to u_1 a flow of 5). The elements on the edges of G_s are illustrated (as sequences of dots ordered by time) at the bottom of the figure, colored by

Algorithm 1 Instance finding module

```
Require: $\delta$, $\phi$, time window $T$, structural match $G_s$
$\mathcal{I} \leftarrow \emptyset$                                  ▷ set of instances of $G_s$ in $T$
for each maximal time window $T$ that satisfies $\delta$ do
    $\mathcal{I} \leftarrow \mathcal{I} \cup$ FindInstances($G_s$, $SP_M$, $T$, $\phi$)
end for
return $\mathcal{I}$

procedure FindInstances($G_s$, $SP$, $T$, $\phi$)
    $\mathcal{I} \leftarrow \emptyset$                              ▷ set of instances of $G_s$ in $T$
    if $length(SP) = 1$ then
        $R_T(e_1) \leftarrow$ all $(t, f)$ elements of $R(e_1)$ in $T$
        if $f(R_T(e_1)) \geq \phi$ then                      ▷ $\phi$ condition check
            add $R_T(e_1)$ to $\mathcal{I}$
        end if
    else
        for each prefix $T_p$ of time window $T$ do
            $R_{T_p}(e_1) \leftarrow$ all $(t, f)$ elements of $R(e_1)$ in $T_p$
            if $f(R_{T_p}(e_1)) \geq \phi$ then              ▷ $\phi$ condition check
                $SP_{next} \leftarrow SP[2:m]$                  ▷ suffix of $SP$
                $T_{next} \leftarrow T - T_p$                    ▷ suffix of $T$
                $\mathcal{I}_{next} \leftarrow$ FindInstances($G_s$, $SP_{next}$, $T_{next}$, $\phi$)
                for each $I \in \mathcal{I}_{next}$ do
                    add $R_{T_p}(e_1) \circ I$ to $\mathcal{I}$
                end for
            end if
        end for
    end if
    return $\mathcal{I}$
end procedure
```

the edge they belong to (e.g., black for e_2). The first row of dots includes all (t, f) elements, i.e., the first black dot corresponds to element $(9, 4)$ on edge (u_1, u_2), which is mapped to the second edge e_2 of $M(3, 3)$. To find the motif instances that comprise of nodes and edges in G_s, we slide a window of length δ along the timeline. Assuming that $\delta = 10$, the first position of the sliding window is $[10, 20]$. The algorithm finds all prefixes of elements in $R(e_1)$ that fall in this window and for each such prefix, it generates recursively the combinations of elements from other edges that form valid instances (according to δ). For example, for the prefix $T_p = [10, 10]$, which includes just the first element $(10, 5)$ from e_1, the 2nd and the 3rd line of dots in the figure show the valid instances formed. Specifically, these instances are $[e_1 \leftarrow \{(10, 5)\}, e_2 \leftarrow \{(11, 3)\}, e_3 \leftarrow \{(14, 4), (19, 6)\}]$ and $[e_1 \leftarrow \{(10, 5)\}, e_2 \leftarrow \{(11, 3), (16, 3)\}, e_3 \leftarrow \{(19, 6)\}]$. Note that the ϕ constraint is applied at every prefix in order to prune the search space if it is violated (e.g., if $\phi = 5$, any instance $[e_1 \leftarrow \{(10, 5)\}, e_2 \leftarrow \{(11, 3)\}, \dots]$ would be rejected. Note also that there is no instance which contains just the first two elements of e_1 but not the third one, because there is no element from e_2 which is temporally between $(13, 2)$ and $(15, 3)$. Finally, note that the next position of the sliding window is $[15, 25]$ because the position $[13, 23]$ which starts from the 2nd element of e_1 does not include any new elements from e_3 compared to the previous window position $[10, 20]$; hence, considering window position $[13, 23]$ would result in redundant (i.e., non-maximal) instances and this position is skipped.

We have not explained yet how window positions are skipped in Algorithm 1. First, only window positions which start at elements of $R(e_1)$ are considered; in-between positions (e.g., window $[11, 21]$ in Figure 7) would result in redundant (non-maximal) instances because there will be a subsequent position for which $R(e_1)$ (and the other sets) can only expand (e.g., window $[13, 23]$ in Figure 7). Second, from those window positions that are considered, we skip those, where $R(e_m)$ (i.e., the interaction time series, which is mapped to the last edge e_m of the motif) is not expanded with new elements, compared to the previous valid window position. In our example, $[13, 23]$ is skipped because no element is added to $R(e_3)$, compared to position $[10, 20]$. If we used window position $[13, 23]$, we would generate instances that would not be maximal because we could add to each of them element $(10, 5)$ of e_1 without violating the δ constraint. In summary, in consecutive window positions where module FindInstances is applied, the first elements of $R(e_1)$ should be different and the last elements of $R(e_m)$ should also be different.

Algorithm 1 does not miss any maximal instances because it systematically explores the combinations of edge-sets which are time-respecting and maximal within a window. Moreover, the windows have maximal lengths and in each of them the produced instances essentially include the temporally first (t_i, f_i) element that maps to e_1 and the temporally last (t_i, f_i) element that maps to e_m. At least one of these pairs changes in the next window position; therefore, instances produced at different windows do not violate the maximality condition.

Complexity Analysis. In the worst case, for each G_s and each time window, we should consider all combinations of edges in G that instantiate the edges of the motif. For example, when $\phi = 0$, prefix-based pruning cannot be applied. In the worst case, $G_s = G$ and the edges in G ordered by timestamp are assigned to the sequence of motif edges in a round-robin fashion. That is, the temporally first edge of G is mapped to e_1, the second to e_2, etc. In this case, assuming the loosest possible constraints $\delta = \infty, \phi = 0$, the number of combinations of pairs to be considered (which all form valid motif instances) is $O(|E|/m)^m$, i.e., exponential to the number of edges m in the motif. In addition, the number of structural matches is also exponential to m. In practice, G_T is sparse (or V is small) and the constraints δ and ϕ help in pruning combinations of edges that do not form instances early, which renders the algorithm scalable, as we will show in the experimental evaluation.

5 TOP-K FLOW MOTIF SEARCH

Setting an appropriate value for the parameters δ and ϕ could be hard for non-experts of the domain. Parameter δ is intuitively easier to be set to a time constraint that makes sense to the application (for example, the analyst could be interested in patterns of bitcoin transactions which happen within an hour or day). On the other hand, ϕ is less intuitive, as too large values could result in too few or zero instances, whereas too small values could result in thousands of instances which may overwhelm the user. One solution to this problem is to replace the ϕ constraint by a ranking of the motif instances G_I with respect to their flow (see Equation 1). In other words, we may opt to search for the k instances G_I of the motif (with $\phi = 0$) that satisfy δ, which have the maximum flow $f(G_I)$.

To solve this top-k flow motif search problem, we can use our algorithm with a small number of modifications. Phase P1 is identical; we should still find the set S of all structural matches. Then, for each $G_s \in S$, we apply phase P2, by making the following changes to Algorithm 1. First, we keep track in a priority queue (heap) the top-k instances in terms of their minimum flow so far. Second, in place of ϕ, we use the flow $f(G_I^k)$ of the k-th instance G_I^k so far as a dynamic (floating) threshold.

5.1 Finding the top motif instance

For the special case, where k=1, the top-1 motif instance search problem can potentially be solved faster with the help of a dynamic programming (DP) algorithmic module. Recall that the objective of procedure FindInstances in Algorithm 1 is to find the motif instances in a structural match G_s, within a time window T, which qualify ϕ. We can replace this module by a dynamic programming algorithm that finds the instance of maximum flow within T. This DP module can be described by Algorithm 2.

Algorithm 2 DP module for top-1 instance search

```
Require: $\delta$, time window $T$, structural match $G_s$
maxflow ← 0        ▷ keeps track of max flow found at any instance
for each maximal time window $T$ that satisfies $\delta$ do
    for all timestamps $t_i$ in $T$ do
        compute $Flow([t_1, t_i], 1) = flow([t_1, t_i], 1)$
    end for
    for $\kappa$ = 2 to $n$ do
        for all timestamps $t_i$ in $T$ do
            compute $Flow([t_1, t_i], \kappa)$ by Eq. 2
        end for
    end for
    maxflow = max (maxflow, $Flow([t_1, t_\tau, m])$)
end for
return maxflow
```

Specifically, let $[t_1, t_2, \ldots, t_\tau]$ be the sequence of timestamps in T for which there is a (t, f) interaction element in G_s. Let M_κ be the prefix of M which includes its fist κ edges only and $Flow([t_1, t_i], \kappa)$ be the flow of the top-1 motif instance of M_κ in the time window $[t_1, t_i]$. Then, $Flow([t_1, t_i], \kappa)$ can be recursively computed as follows:

$$Flow([t_1, t_i], \kappa) = \max_{1<j\leq i}\{\min(Flow([t_1, t_{j-1}], \kappa-1), flow([t_j, t_i], \kappa))\}, \quad (2)$$

where $flow([t_j, t_i], \kappa)$ is the total flow of all (t, f) elements of the time series $R(e_\kappa)$ on the κ-th edge of G_s, whose timestamps are in the time interval $[t_j, t_i]$. The $Flow([t_1, t_i], 1)$ array is initialized by scanning the elements of the first edge of G_s in T. Then, for each $\kappa > 1$, $Flow([t_1, t_i], \kappa)$ is computed using array $Flow([t_1, t_i], \kappa-1)$. Finally, $Flow([t_1, t_\tau], m)$ corresponds to the top-1 flow of any motif instance in G_s within time window T. By applying this algorithm for every window T, we can find the top instance in G_s. Repeating this for each G_s gives us the top-1 instance of M in G.

Table 2 shows the steps of the DP module in the course of finding the top-1 instance in time window $[10, 20]$ (assuming that δ=10) for the structural match of $M(3, 3)$ shown in Figure 7. The first row shows the values of $Flow([t_1, t_i], 1)$ for the first edge of the motif and for all values of t_i (i.e., columns of the table). (Recall that the starting timestamp t_1 of the time window is 10.) The second row shows, for the first two edges of the motif, the value of $Flow([t_1, t_i], 2)$ for all values of t_i, as well as the value of t_j, which determines $Flow([t_1, t_i], 2)$. For all t_i, the value of t_j that maximizes the flow is 11 and for $t_i \geq 16$ the flow becomes $min(5, 3 + 3) = 5$. Finally, the last row shows the maximum flow for the best arrangement of (t, f) pairs to all three edges of the motif, for all prefixes of the time window. Note that the last value corresponds to the entire window and contains the flow of the best instance of the entire motif in $[10, 20]$, which is 5. The cells of the matrix in bold show how the top-1 instance, i.e., $[e_1 \leftarrow \{(10, 5)\}, e_2 \leftarrow \{(11, 3), (16, 3)\}, e_3 \leftarrow \{(19, 6)\}]$, can be identified.

Table 2: Example of the DP module

t_i	10	11	13	14	15	16	18	19
κ=1	**5**	5	7	7	7	7	10	10
κ=2		3 (t_j=11)	3 (t_j=11)	3 (t_j=11)	3 (t_j=11)	5 (t_j=11)	**5 (t_j=11)**	5 (t_j=11)
κ=3			0 (t_j=13)	4 (t_j=14)	4 (t_j=14)	4 (t_j=14)	4 (t_j=14)	**5 (t_j=19)**

Complexity Analysis. For each G_s and each time window, we should consider all binary splits of the window at each iteration (i.e., for each edge in M). Hence the time complexity is $O(\tau^2|E|)$, where τ is the number of timestamps in T for which there is an (t_i, f_i) element in G_s. The space complexity is $O(\tau \cdot |E|)$ because we only need all $Flow([t_1, t_i], \kappa-1)$ for $\kappa-1$ when we process the κ-th edge. The overall time complexity per structural match in S is $O(|S|\delta\tau^2|E|)$, since the number of windows to be considered is $O(\delta)$. The number of structural matches $|S|$ is exponential to m, as discussed in our previous analysis.

Extensibility. The DP module can be used to solve top-1 problems at a finer granularity. In particular, it can be used to find the top-1 instance for each structural match G_s. This may be desirable if we want to compare the sets of entities that constitute the structural instances (e.g., groups of bitcoin users) based on their max-flow interactions. In addition, we might be interested in finding the top-1 instance for each position of the sliding time window T. This can be part of analysis tasks that compare the volume of interactions (according to the motif structure) at different periods of time.

6 EXPERIMENTAL EVALUATION

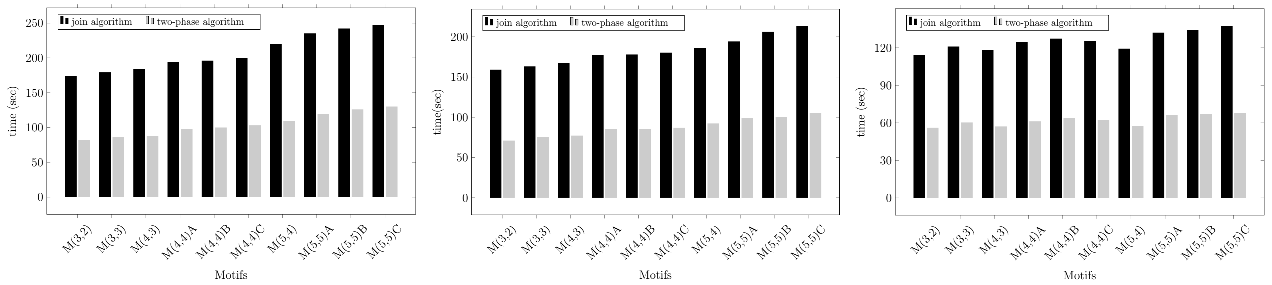
The goal of our experimental evaluation is twofold: test the performance and scalability of our algorithms and study the significance of flow motifs. We implemented the algorithm presented in Section 4 and its two variants proposed in Section 5 (top-k instance search, dynamic programming module for top-1 search). As a baseline, we also implemented an alternative motif instance finding method based on finding and joining instances of motif components in a hierarchical manner.

We evaluate the performance of all these methods on three real networks, to be described in Section 6.1. We measure the efficiency and scalability of the tested methods as a function of the problem parameters δ and ϕ on the motif structures shown in Figure 3. These graphs model representative flows of interaction that could be of interest to data analysts (e.g., $M(3, 3)$ corresponds to cyclic transactions in a money-exchange network, $M(4, 3)$ corresponds to paths of region-to-region movements in a passenger flow network). We also assess the statistical significance of the tested motifs in three real graphs. All algorithms were implemented in Python3 and we ran all the experiments on a machine with an Intel Xeon CPU E5-2620 prossesor running Ubuntu 18.04.1 LTS.

6.1 Dataset Description

We used three datasets extracted from real interaction networks: the **Bitcoin** network, the **Facebook** network and a **Passenger** flow network. Table 3 shows statistics of the datasets. The third column is the distinct number of node pairs $(u, v) \in V$, for which there is at least one edge (i.e., interaction) from u to v. This number equals to the number $|E_T|$ of edges in the corresponding time-series graph G_T. We now provide more details about them.

Bitcoin network. We downloaded all transactions in the bitcoin blockchain [16] in the period February 1st 2014 to November



(a) Bitcoin Network

(b) Facebook Network

(c) Passenger Network

Figure 8: Our two-phase algorithm vs. the join algorithm

Table 3: Statistics of Datasets

Dataset	#nodes	#connected node pairs	#edges	Avg. flow per edge
Bitcoin	24.6M	88.9M	123M	4.845
Facebook	45800	264000	856000	3.014
Passenger	289	77896	215175	1.933

30 2014 and converted them to a bitcoin *user graph*.[1] Nodes correspond to users and for each transaction of f bitcoins in the blockchain from user u to user v at time t, we added an edge from u to v with label (t, f). Since the same bitcoin user may control and use multiple addresses, we applied a well-known heuristic [1, 7] to *merge* addresses that are considered to belong to the same user to a single network node. Specifically, we merged addresses that appear together as input in the same transaction. We did not take into account insignificant transactions with amounts under 0.0001 BTC. Bitcoin is a relatively sparse graph and the cases of two nodes being connected by multiple edges is rare. Finding motif instances in the Bitcoin network can help towards understanding complex interactions between users and can possibly help toward identifying suspicious transactions like money laundering and bitcoin theft [14].

Facebook network: We consider Facebook as an interaction network between users. We divide the time into 30-second intervals $[t_s, t_e)$ and for each pair of users u and v we aggregate all interactions from u to v and add an edge from u to v with label (t_s, f), where f is the total number of interactions from u to v in this interval. We consider as interactions the posts of likes by u targeting v or the messages sent from u to v. We created the Facebook user network using data from April 2015 to October 2015; the same dataset is used in [19]. The Facebook network is relatively sparse and each pair of connected nodes have about four edges on average. Motif search on this graph can help in analyzing influence [3, 11] and finding important interactions among users [13].

Passenger flow network: We processed trips of yellow taxis in NYC in January 2018.[2]. Each record includes the pick-up and drop-off taxi zones (regions) the date/time of the pick-up and drop-off, and the number of passengers inside the taxi. Using these records, we created an interaction network where the nodes are the taxi zones; for each record, we generate an edge that links the corresponding nodes and carries the timestamp of the activity (i.e., the pickup time) and the corresponding flow (i.e., the number of passengers). This Passenger flow network is dense; in addition, each pair of connected nodes have about three edges on average. Motif instances found in this passenger flow graph can help in understanding the flow of movement between different regions.

6.2 Efficiency and Scalability

In this section, we evaluate the efficiency and scalability of our algorithm when applied to find the instances of the motifs depicted in Figure 3. The default values for the duration constraint δ are 600 sec., 600 sec., and 900 sec. on Bitcoin, Facebook, and Passenger, respectively. These value represent realistic time intervals for the corresponding applications. The corresponding default values for ϕ are 5, 3, and 2, respectively.

6.2.1 Comparison to a competitor. In our first set of experiments, we compare our algorithm with an alternative motif instance finding algorithm which is based on progressively finding and joining instances of motif subgraphs.

Specifically, this *join algorithm* starts by accessing each edge (u, v) of the time series graph G_T and finding all time-intervals of length at most δ and their aggregated flows. For each such interval $[t_s, t_e]$ a quintuple (u, v, t_s, t_e, f) is generated. These tuples are kept in two tables; C_1 sorts them by starting vertex u and C_2 sorts them by ending vertex v. In the next step, C_2 and C_1 are merge-joined to find all pairs (c_2, c_1) having $c_2.u = c_1.v$ and also satisfying $c_1.t_e - c_2.t_s \leq \delta$. The set P of all these tuple pairs constitute results of all sub-motifs of M which include two consecutive edges. In the next step, P is self-joined again to produce instances of sub-motifs of M with three consecutive edges. This is done by finding pairs $\{(c_2, c_1), (c_2', c_1')\}$ of couples in P for which $c_1 = c_2'$ and $c_1'.t_e - c_2.t_s \leq \delta$. The next steps are applied in a similar manner until the instances of the entire motif M are constructed. Note that for each motif or sub-motif that closes a cycle (e.g., $M(3, 3)$), we check the additional condition that the starting vertex of the first motif edge in the instance is the same as the target vertex of the last edge. At each step, we apply a merge join for the production of sub-motif instances, after having sorted the tuples produced in the previous step accordingly.

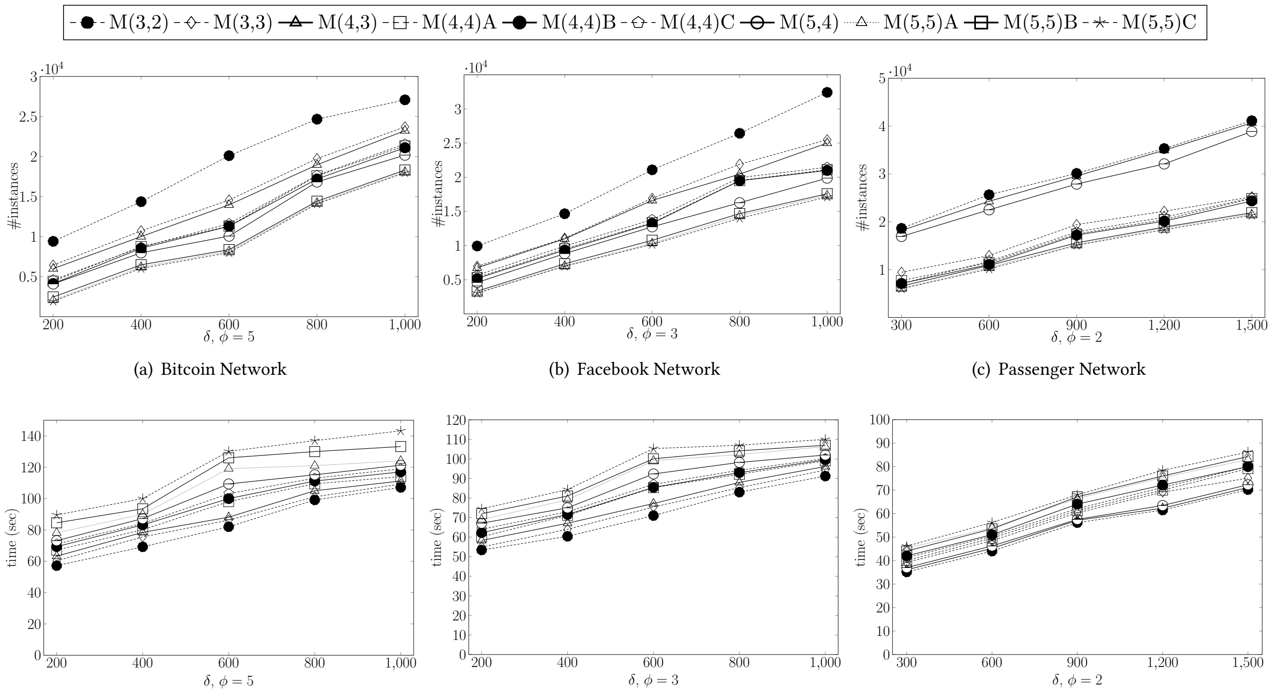
Figure 8 compares the runtime cost of the join algorithm with that of our two-phase algorithm presented in Section 4. For all motifs, we used the default values for δ and ϕ. Our two-phase algorithm is typically twice as fast as the join algorithm. This is due to the fact that the join algorithm produces a large number of intermediate results (i.e., sub-motif instances), which are avoided by our method. Many of these sub-motif instances do not end up as components of any instance of the complete motif, so their generation is redundant. In the rest of this section, we do not include additional comparisons with the join algorithm since it was always found to be slower than our approach.

[1] data obtained from http://www.vo.elte.hu/bitcoin

[2] obtained from http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml

Table 4: Number of structural matches and runtime in phase P1 of motif search

	Motif	M(3,2)	M(3,3)	M(4,3)	M(4,4)A	M(4,4)B	M(4,4)C	M(5,4)	M(5,5)A	M(5,5)B	M(5,5)C
Bitcoin	Instances	634K	485K	484K	210K	205K	213K	145K	122K	124K	121K
	Time (sec)	47.02	49.23	50.15	57.05	60	61.16	64.35	69.11	73.02	75.15
Facebook	Instances	415K	276K	272K	113K	113K	114K	97K	90K	91K	90K
	Time(sec)	40.02	43.43	44.21	48.45	49.32	49.01	52.33	50.12	52.07	54.31
Passenger	Instances	27893	16455	25778	14877	14569	14903	22134	12345	12567	12009
	Time(sec)	19.14	21.33	22.15	26.22	29.03	29.11	25.04	30.45	31.14	32



(a) Bitcoin Network

(b) Facebook Network

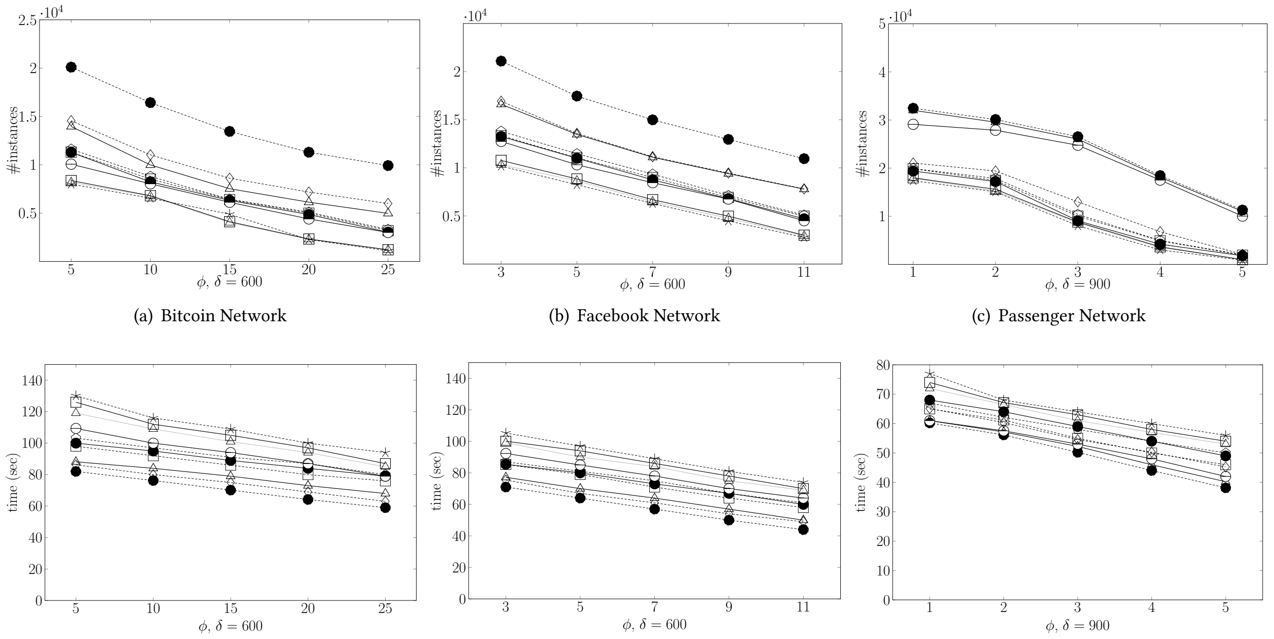
(c) Passenger Network

(d) Bitcoin Network

(e) Facebook Network

(f) Passenger Network

Figure 9: Number of instances and time for different values of δ



(a) Bitcoin Network

(b) Facebook Network

(c) Passenger Network

(d) Bitcoin Network

(e) Facebook Network

(f) Passenger Network

Figure 10: Number of instances and time for different values of ϕ

6.2.2 Sensitivity to δ and ϕ. The next set of experiments evaluate the performance of our algorithm on the different datasets and motifs, for various values of the constraints δ and ϕ. Table 4 shows the number of structural matches found and the time spent by the algorithm just for its first phase, which is independent of the δ and ϕ values (since these constraints are not used when searching for the structural matches). This cost constitutes a lower bound for our algorithm. Naturally, more complex motifs require more time but they also have fewer structural matches.

Figures 9 and 10 show the number of instances and total runtime of our algorithm for different values of δ (in seconds) and ϕ. When we vary δ, we set ϕ to its default value and vice versa. As expected, in all cases, when δ increases the number of instances and the runtime increases. The algorithm scales well as its cost increases at a lower pace compared to the results found.

When comparing the different motifs, note that the simpler ones (e.g., $M(3,2)$ and $M(3,3)$) naturally have more instances and are cheaper to search compared to the more complex ones (e.g., $M(5,5)A$). The relative order between the motifs is similar in the Bitcoin and Facebook networks. In both networks, cyclic flow is quite common; i.e., motifs containing cycles have a similar number of instances as motifs without cycles having the same number of edges. On the other hand, in the Passenger network, acyclic motifs dominate in terms of number of instances. This is expected, as it is relatively rare that passengers move between regions on a map forming cycles compared to moving along a chain of different regions.

The behavior is also consistent to our expectation when ϕ varies; the number of instances and the runtime drop when ϕ increases. The algorithm becomes faster because partial motif instances that do not qualify ϕ are pruned early.

6.2.3 Top-k flow motif instance search. We now evaluate the results and the performance of top-k motif search on the three datasets, when using the default values of δ. In the first experiment, we run the version of our algorithm which finds the top-k motif instances that have the maximum flow. For each run, we record the flow of the k-th instance in Figure 11. As expected, the flow of the k-th instance drops as k increases; the drop rate decreases when k becomes large (note that the x-axis is not linear). In the second experiment, we compare the runtime of the general top-k algorithm with its version that employs the dynamic programming module proposed in Section 5.1. The barcharts show that the second phase of the algorithm benefits from the use of dynamic programming (the runtime drops 20% to 40%). The improvement is better on the Passenger network.

6.2.4 Scalability to the dataset size. In the next experiment, we test the performance of our algorithm on samples of the original datasets having different sizes. For each of the three datasets, we take samples defined by prefixes of the total period covered by the timestamps of the edges included in the sample. Specifically, for the Bitcoin network we define 5 samples: B1, B2, B3, B4, B5. B1 includes all transactions happened in the first month of the 9-month period of the complete dataset. B2, B3, B4, and B5 cover the first 2, 4, 6, and 9 months, respectively. Similarly, F1, F2, F3, F4, and F5 cover the first 1, 2, 3, 4, and 6 months of the entire dataset, respectively. Lastly, T1, T2, T3, and T4 cover the first 8, 16, 24, and 31 days of January 2018, respectively. Figure 13 shows the growth in the number of instances and in the runtime of the algorithm for the different motifs. Observe that the algorithm scales well as its cost grows at a slower pace compared to the number of instances and the size of the input data.

6.3 Significance of Motifs

In the last experiment, we assess the significance of the different flow motifs in our networks. Following the standard practice [18], we generated randomized versions of our datasets, we computed the number of instances of each motif in each of these datasets, and we compared it against the same number for the real dataset. A large divergence between real and randomized numbers indicates a significant motif.

Specifically, from each dataset (e.g.. Bitcoin network) we generated random datasets by keeping the structure of the corresponding graph fixed, and permuting the flows on the edges. Recall that in the original input multigraph $G = (V, E)$ each edge e is associated with a timestamp $t(e)$ and a flow value $f(e)$. A pair of nodes (u, v) is connected by a set of edges $E(u, v)$. Given the entire set of flow values $\{f(e) : e \in E\}$, we compute a random permutation π of the flow values and reassign them to the graph edges in this order. This generates a randomized dataset $G_r(V, E)$ with the same set of nodes and the same set of edges; each edge e has the same timestamp $t(e)$, and flow value $\pi(f(e))$. Hence, G_r is derived from G by "shuffling" the flow values on the edges.

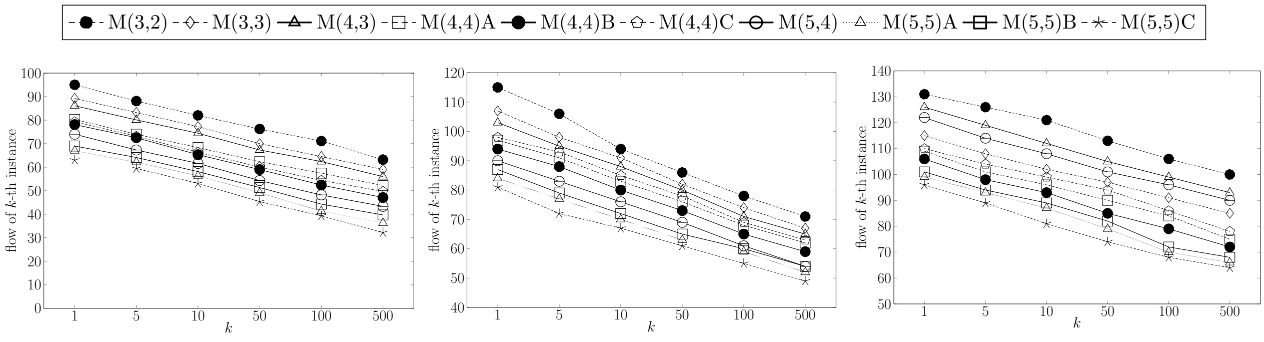
The random graph G_r has the same structure as G and the edges in the graph appear at the same timestamps. Therefore, all structural matches of the motifs in G will also appear in G_r. In addition, putting aside the flow constraint ϕ, the motif instances in the two graphs will be the same, when considering only δ. What changes is the flow value of each motif instance, which will result in a different number of flow motif instances in G_r compared to G, for non-zero values of ϕ. Our goal is to see whether the motif instances that satisfy the ϕ constraint in the real data are significantly more than those in the randomized data.

We generated 20 different random graphs for each real network according to the procedure we described above. We found the instances of each motif in all these random datasets. In addition, we computed the mean and standard deviation of the number of motif instances in all 20 random graphs per real dataset. To assess the significance of a motif in the real data, we compared the number of instances in the real data with those in the random data. Figure 14 shows, for each dataset and motif, the distribution of the numbers of instances for all random graphs in a box plot, and the corresponding number in the real graph (marked by a diamond). Each real value is also associated with the *z-score* (shown above the corresponding diamond), which is computed as follows. For some motif M, let r_M denote the number of instances of the motif in the real data, let μ_M denote the mean number of motif instances in the randomized data, and let σ_M denote the standard deviation. The z-score z_M of the motif is computed as

$$z_M = \frac{r_M - \mu_M}{\sigma_M}$$

The higher the z-score, the further the value r_M from μ_M.

The first observation is that the number of instances in all random graphs is much lower compared to that in the corresponding real network and these values do not deviate much from their mean. The empirical p-value (the fraction of random datasets with number of instances greater than that of the real data) is zero, indicating statistical significance of the motif occurrences in all cases. This is consistent with the intuition that the flow is not arbitrarily generated or consumed at the vertices of the network, but it is transferred from one node to another. To discriminate between the different motifs we look at the z-scores. We observe that for the Bitcoin network, two out of the three top z-scores are

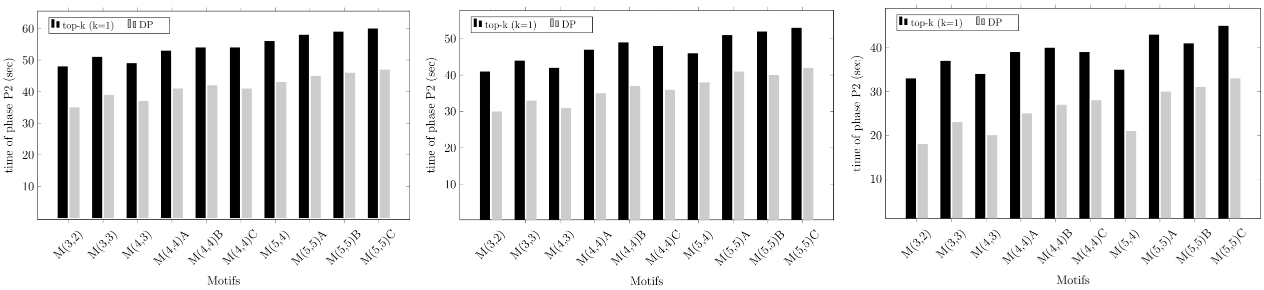


(a) Bitcoin Network

(b) Facebook Network

(c) Passenger Network

Figure 11: Flow of k-th instance

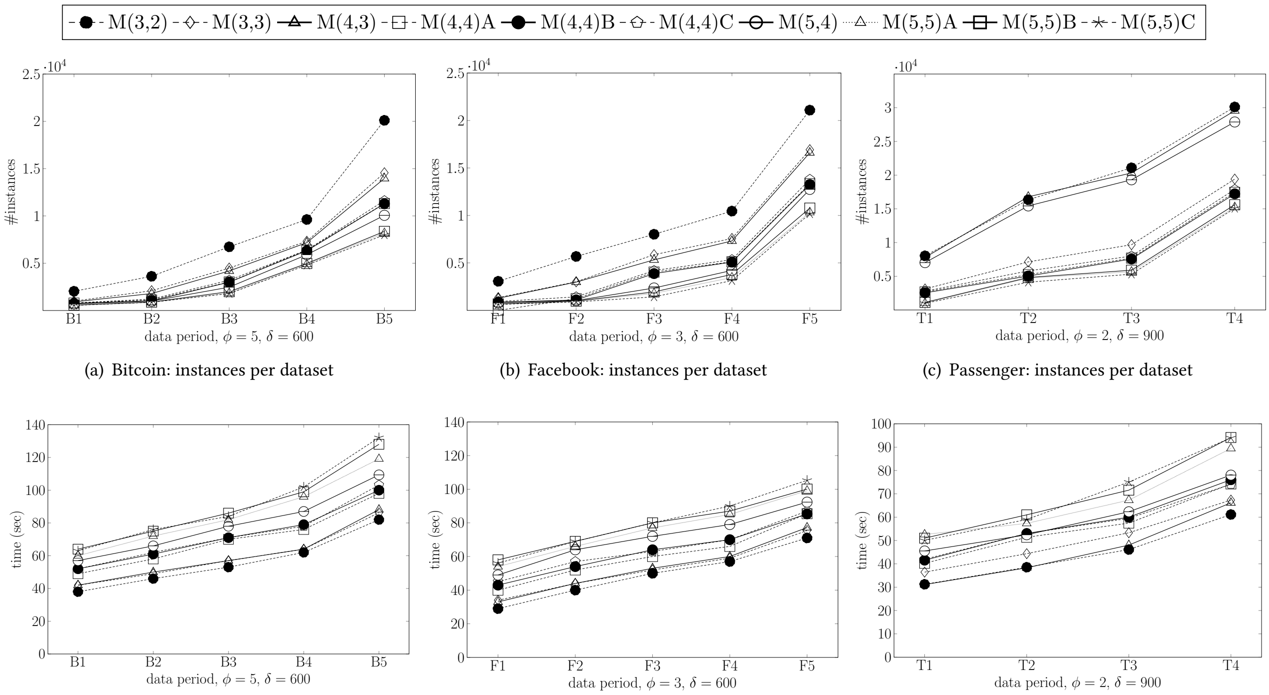


(a) Bitcoin Network

(b) Facebook Network

(c) Traffic Network

Figure 12: Efficiency of the dynamic programming module



(a) Bitcoin: instances per dataset

(b) Facebook: instances per dataset

(c) Passenger: instances per dataset

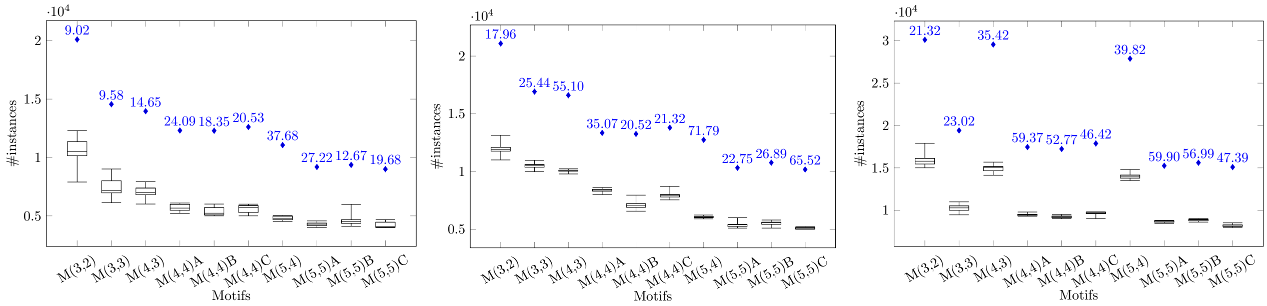
(d) Bitcoin: time per dataset

(e) Facebook: time per dataset

(f) Passenger: time per dataset

Figure 13: Scalability to input graph size

for motifs that contain cycles, indicating that large flow movements that close a cycle are statistically over-represented in the bitcoin network. A similar observation holds for the Passenger flow network, where three out of the top-three motifs contain a cycle. A different pattern emerges in the Facebook dataset, where two out of the three highest z-scores are for chains of nodes. We conjecture that this due to propagation trees of information in the Facebook network, which result in chains with significantly



(a) Bitcoin Network

(b) Facebook Network

(c) Passenger Network

Figure 14: Number of instances in random networks (box plots), in real networks (diamonds), and z-scores

high flow movement. It is interesting that the significance of the discovered motifs varies in the different types of interaction networks, indicating differences in the way flow is distributed.

7 CONCLUSION

In this paper, we introduced the novel concept of *network flow motifs*. To the best of our knowledge we are the first to define and study motifs in interaction networks, which consider both the temporal and flow information of the interactions. We proposed an efficient algorithm for enumerating flow motif instances in large graphs and variants of that find the top-k instances of maximal flow. We evaluated our algorithm on three real datasets and demonstrated its scalability. In addition, we compared it to a baseline motif instance finding method based on joining instances of motif components and showed its superiority. Finally, we studied the statistical significance of a wide range of representative motifs on the real graphs and showed that they indeed appear more frequently than in random networks with the same characteristics. This indicates that the flow is transferred from one node to another (as opposed to being arbitrarily consumed or generated) and that there are subgraphs in the network where significant flow is transferred at certain periods of time.

In the future, we plan to investigate in more detail the distribution of motif instances in the real networks. For example, we can group the motif instances per structural match, in order to identify the structural matches (i.e., sets of vertices in the graph G) with the largest activity and how this activity is spread along the timeline. Another direction is to improve the efficiency of the algorithm, by processing multiple structural instances together in phase P2. Since two or more structural matches may share the same prefix, we can compute the flow instances of their common prefix simultaneously before expanding these instances to complete ones for the different motifs. In addition, we will work towards a version of the algorithm which focuses on counting instances of (possibly multiple) motifs without constructing them (along the direction of previous work [17]). Finally, we will generalize the definition of flow motifs to capture other graph structures besides paths (e.g., directed acyclic graphs with forks and joins) and study their search in large networks.

ACKNOWLEDGMENT

This research has been co-financed by the European Regional Development Fund of the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH – CREATE – INNOVATE (project code: T1EDK-00108).

REFERENCES

[1] Rémy Cazabet, Rym Baccour, and Matthieu Latapy. 2017. Tracking Bitcoin Users Activity Using Community Detection on a Network of Weak Signals. In *COMPLEX NETWORKS*. 166–177.

[2] Luis Enrique Correa da Rocha and Vincent D. Blondel. 2013. Flow Motifs Reveal Limitations of the Static Framework to Represent Human interactions. *CoRR* abs/1303.3245 (2013).

[3] Manuel Gomez-Rodriguez, Jure Leskovec, and Andreas Krause. 2012. Inferring Networks of Diffusion and Influence. *TKDD* 5, 4 (2012), 21:1–21:37.

[4] Saket Gurukar, Sayan Ranu, and Balaraman Ravindran. 2015. COMMIT: A Scalable Approach to Mining Communication Motifs from Dynamic Networks. In *SIGMOD*. 475–489.

[5] Petter Holme. 2015. Modern temporal network theory: A colloquium. *CoRR* abs/1508.01303 (2015). arXiv:1508.01303 http://arxiv.org/abs/1508.01303

[6] David Kempe, Jon M. Kleinberg, and Amit Kumar. 2002. Connectivity and Inference Problems for Temporal Networks. *J. Comput. Syst. Sci.* 64, 4 (2002), 820–842.

[7] Dániel Kondor, Márton Pósfai, István Csabai, and Gábor Vattay. 2013. Do the rich get richer? An empirical analysis of the BitCoin transaction network. *PLoS ONE* 9, 2 (2013), e86197.

[8] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. 2011. Temporal motifs in time-dependent networks. *CoRR* abs/1107.5646 (2011). arXiv:1107.5646 http://arxiv.org/abs/1107.5646

[9] Rohit Kumar and Toon Calders. 2017. Information Propagation in Interaction Networks. In *EDBT*. 270–281.

[10] Rohit Kumar and Toon Calders. 2018. 2SCENT: An Efficient Algorithm to Enumerate All Simple Temporal Cycles. *PVLDB* 11, 11 (2018), 1441–1453.

[11] Jure Leskovec, Mary McGlohon, Christos Faloutsos, Natalie S. Glance, and Matthew Hurst. 2007. Patterns of Cascading Behavior in Large Blog Graphs. In *SDM*. 551–556.

[12] Yuchen Li, Zhengzhi Lou, Yu Shi, and Jiawei Han. 2018. Temporal Motifs in Heterogeneous Information Networks. In *MLG Workshop @ KDD*.

[13] Julian J. McAuley and Jure Leskovec. 2012. Learning to Discover Social Circles in Ego Networks. In *NIPS*. 548–556.

[14] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage. 2013. A fistful of bitcoins: characterizing payments among men with no names. In *IMC*. 127–140.

[15] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon1. 2004. Network Motifs: Simple Building Blocks of Complex Networks. *Science* 298, 5594 (2004), 824–827.

[16] Satoshi Nakamoto. 2007. Bitcoin: A peer-to-peer electronic cash system http://bitcoin.org/bitcoin.pdf.

[17] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. 2017. Motifs in Temporal Networks. In *WSDM*. 601–610.

[18] Sayan Ranu and Ambuj K. Singh. 2009. GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases. In *ICDE*. 844–855.

[19] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. *Computer Communication Review* 45, 5 (2015), 123–137.

[20] Konstantinos Semertzidis and Evaggelia Pitoura. 2016. Durable graph pattern queries on historical graphs. In *ICDE*. 541–552.

[21] Sebastian Wernicke and Florian Rasche. 2006. FANMOD: a tool for fast network motif detection. *Bioinformatics* 22, 9 (2006), 1152–1153.

[22] Rongjing Xiang, Jennifer Neville, and Monica Rogati. 2010. Modeling relationship strength in online social networks. In *WWW*. 981–990.

[23] Qiankun Zhao, Yuan Tian, Qi He, Nuria Oliver, Ruoming Jin, and Wang-Chien Lee. 2010. Communication motifs: a tool to characterize social communications. In *CIKM*. 1645–1648.

[24] Andreas Züfle, Matthias Renz, Tobias Emrich, and Maximilian Franzke. 2018. Pattern Search in Temporal Social Networks. In *EDBT*. 289–300.

DynFD: Functional Dependency Discovery in Dynamic Datasets

Philipp Schirmer
bakdata GmbH
Berlin, Germany
philipp.schirmer@bakdata.com

Thorsten Papenbrock
Hasso Plattner Institute
University of Potsdam, Germany
thorsten.papenbrock@hpi.de

Sebastian Kruse
Hasso Plattner Institute
University of Potsdam, Germany
sebastian.kruse@hpi.de

Dennis Hempfing
Hasso Plattner Institute
University of Potsdam, Germany
dennis.hempfing@hpi-alumni.de

Torben Meyer
Hasso Plattner Institute
University of Potsdam, Germany
torben.meyer@hpi-alumni.de

Daniel Neuschäfer-Rube
Hasso Plattner Institute
University of Potsdam, Germany
daniel.neuschaefer-rube@
hpi-alumni.de

Felix Naumann
Hasso Plattner Institute
University of Potsdam, Germany
felix.naumann@hpi.de

ABSTRACT

Functional dependencies (FDs) support various tasks for the management of relational data, such as schema normalization, data cleaning, and query optimization. However, while existing FD discovery algorithms regard only static datasets, many real-world datasets are constantly changing – and with them their FDs. Unfortunately, the computational hardness of FD discovery prohibits a continuous re-execution of those existing algorithms with every change of the data.

To this end, we propose DynFD, the first algorithm to discover and maintain functional dependencies in dynamic datasets. Whenever the inspected dataset changes, DynFD *evolves* its FDs rather than recalculating them. For this to work efficiently, we propose indexed data structures along with novel and efficient update operations. Our experiments compare DynFD's incremental mode of operation to the repeated re-execution of existing, static algorithms. They show that DynFD can maintain the FDs of dynamic datasets over an order of magnitude faster than its static counter-parts.

1 FUNCTIONAL DEPENDENCIES

Traditional data profiling algorithms solve the problem of *discovering* all metadata of type X in dataset Y. However, once those algorithms finish, the dataset usually keeps evolving, thereby rendering the discovered metadata outdated. An *incremental* data profiling algorithm, on the contrary, acknowledges the dynamic nature of data by *maintaining* all metadata of some type. It takes as input the data and its (statically profiled) metadata and, then, updates the metadata with every change, i. e., insert, update, and delete of the data. In this paper, we propose such an incremental algorithm for functional dependencies.

For an instance r of a relation R, a functional dependency (FD) $X \longrightarrow A$ holds iff all records with the same values for the set of attributes $X \subseteq R$ also share the same value for attribute $A \in R$ [5]. We say that X *functionally determines* A.

Definition 1.1. The *functional dependency* $X \longrightarrow A$ with $X \subseteq R$ and $A \in R$ is *valid* for instance r of R, iff $\forall t_i, t_j \in r\colon t_i[X] = t_j[X] \Rightarrow t_i[A] = t_j[A]$. We call X the left-hand side (LHS) and A the right-hand side (RHS), respectively.

FDs often arise from real-world relationships. Consider, for example, a relation with information about people including their *ZIP code* and *city* name. In such a dataset, the ZIP codes functionally determine the city names of the records, because ZIP codes are a more fine-grained localization. The presence or absence of certain dependencies, in general, helps to understand complex semantic relationships in data exploration scenarios. Being able to track the validity of certain dependencies over time provides even deeper insights. In a product database, for instance, the FD *num_sales* ⟶ *num_shipments* might hold only overnight, because shipments are delayed in daily business. Or the FD *product* ⟶ *price* in a pricing database was temporarily violated at the time of a system migration. Apart from data exploration, further applications for functional dependencies include schema normalization [4], query optimization [14], data integration [11], data cleansing [2], and data translation [3]. Given that the functional dependencies are not only known for a snapshot of the data but over a longer period of time, it is for any of these use cases easier to identify *robust* dependencies. Continuous *change patterns* for non-robust dependencies, on the other hand, can be of interest themselves; and *sudden changes* of thus far robust FDs might signal data quality issues, i. e., erroneous updates.

The most interesting FDs for all such applications are *minimal, non-trivial* FDs. An FD $X \longrightarrow A$ is non-trivial if $A \notin X$, otherwise it would hold on any instance r of R and, hence, would not characterize r. An FD is minimal if no *generalization*, i. e., no subset of the FD's LHS also describes a valid FD. More formally, an FD $X' \longrightarrow A$ is a generalization of another FD $X \longrightarrow A$ if $X' \subset X$. On the other hand, an FD $X'' \longrightarrow A$ is a *specialization* of an FD $X \longrightarrow A$ if $X \subset X''$, i. e., if its LHS is a superset of the other FD's LHS. Any valid, minimal FD implies that all of its specializations are valid as well, which makes them particularly interesting. As a result, given the *complete* set of minimal FDs, all other FDs can be inferred from it. For this reason, it suffices to discover and maintain only minimal, non-trivial FDs.

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.23

Despite the restriction to minimal FDs, the discovery of all such dependencies is still expensive: Liu et al. have shown that, when using nested loops for the dependency validations, the complexity of the discovery is in $O\left(n^2 \left(\frac{m}{2}\right)^2 2^m\right)$ for relations with m attributes and n records [10]. More sophisticated, index-based algorithms, such as [8] or [13], avoid the quadratic complexity of the candidate validations (n^2), but the algorithms' complexity w. r. t. the number of attributes stays exponential (2^m). This is inevitable due to the potentially exponential number of discovered FDs. The discovery problem becomes even harder when taking data changes into account. As mentioned above, all state-of-the-art FD discovery algorithms operate only on *static datasets* and one needs to re-execute them after every change of the data to maintain the FDs on *dynamic datasets*. Unfortunately, this is computationally far too expensive in practical situations: It is not unusual for those algorithms to take minutes or even hours to complete.

However, the following two observations suggest that this problem can be solved with a novel, *incremental* algorithm: First, most changes affect only a small subset of records and only these small deltas need to be investigated for causing metadata changes – the majority of records still support the same FDs as before. Second, most changes in the FDs are minor, meaning that a close specialization or generalization of a former minimal FD becomes a new minimal FD.

On the face of this opportunity, we propose DynFD, the first algorithm that maintains the complete and exact set of minimal, non-trivial FDs on dynamic data. The algorithm monitors data changes, i. e., inserts, updates, and deletes, and calculates their effect on the metadata. The changes are grouped into batches of configurable size so that a user can specify timeliness of the metadata (at the cost of performance). So rather than frequently re-computing all FDs, DynFD continuously deduces FD changes from the previous set of FDs and the batch of change operations. In detail, our contributions are the following:

(1) *FD maintenance algorithm.* We present DynFD, an algorithm that incorporates data changes, i. e., inserts, updates, and deletes, as batches into sets of minimal functional dependencies. While updates are simply handled as a combination of insert and delete, the algorithm offers specialized handling strategies for inserts and deletes (Section 2).

(2) *FD maintenance data structures.* To update the dependencies efficiently, our DynFD algorithm needs to maintain several data structures, such as position list indexes, dictionary-encoded records, and FD prefix trees, over time. We explain these data structures and how they need to change w. r. t. newly inserted or deleted tuples. We also propose a novel cover inversion algorithm to deduce an FD negative cover from an FD positive cover (Section 3).

(3) *FD maintenance pruning rules and techniques.* We devise novel pruning rules and techniques for insert and delete operations that allow DynFD to optimize or even skip certain validations. For validations that cannot be skipped, we propose efficient validation methods that exploit the incremental nature of the data changes (Section 4 and Section 5).

(4) *Evaluation.* We provide an exhaustive evaluation of DynFD w. r. t. scalability and speed-up. We investigate the effectiveness of our pruning and maintenance strategies and compare DynFD to HyFD, the state-of-the-art FD discovery algorithm for static data (Section 6).

Table 1: An example relation with four initial tuples. A batch of changes inserts and deletes tuples, as indicated by the "–" and "+" signs, respectively.

ID	firstname	lastname	zip	city
1	Max	Jones	14482	Potsdam
2	Max	Miller	14482	Potsdam
– 3	Max	Jones	10115	Berlin
4	Anna	Scott	13591	Berlin
+ 5	Marie	Scott	14467	Potsdam
+ 6	Marie	Gray	14469	Potsdam

2 OVERVIEW OF DYNFD

Before we discuss implementation details, we give an overview of our proposed algorithm DynFD. This algorithm operates on a single relation, such as the one exemplified in Table 1, which may or may not contain an initial set of tuples (here: tuples 1–4). The relation is then subject to a series of batches of changes. Each batch inserts and/or deletes tuples from the relation. For instance, Table 1 shows a batch that removes tuple 3 and inserts tuples 5 and 6. Note that tuple updates can be expressed by a delete and an insert operation. Also, note that the size of the batches is at the discretion of users and their use cases and allows for a trade-off between granularity and performance of the FD maintenance process.

As explained in Section 1, each batch *can* change the set of minimal FDs in a relation. For instance, while the FD $z \longrightarrow c$ continues to be a minimal FD in Table 1 before and after the batch has been applied, $f \longrightarrow c$ becomes a new minimal FD and $fc \longrightarrow z$ ceases to be a (minimal) FD.

To discover these changes, DynFD comprises three principal components as can be seen in Figure 1: (i) the *data structures*, which are position list indexes and dictionary-encoded records, concisely model all relevant features of the relation to determine the currently valid FDs; (ii) the *positive cover* indexes all minimal FDs and allows to reason on the effect of insert operations; and (iii) the *negative cover* with all maximal non-FDs allows to process delete operations analogously. If the profiled relation contains initial tuples, we employ the static algorithm HyFD [13] to bootstrap the data structures and the positive cover. The negative cover can then be derived from the positive cover via a *cover inversion* as we describe in Section 3.2.

Having initialized all necessary data structures, DynFD begins to monitor changes of the profiled relation. These changes arrive as a stream that is first transformed and then processed in batches, i. e., non-overlapping groups of insert, update, and delete operations. The batches can be, e. g., equally sized groups of change operations or, alternatively, all operations from within a tumbling time window. Each batch is processed according to the following observation: According to Definition 1.1, insert operations can introduce violations to existing FDs, but never remove them. As a result, those FDs become invalid. We can efficiently retrace these changes by operating on the positive cover, i. e., on the existing minimal FDs. Delete operations constitute the opposite case: Exiting violations may be removed, thereby introducing new FDs – or in other words, existing non-FDs may become valid. Here, the negative cover is more appropriate to efficiently reason on delete operations. For this reason, we handle insert and delete operations separately, yet with the same basic principles.

Figure 1: Processing one batch of insert and delete operations with DynFD.

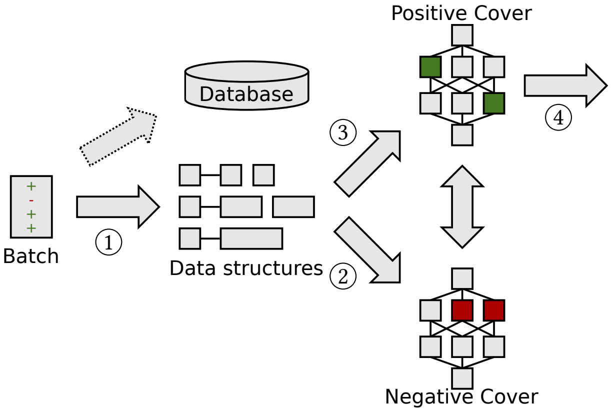


Figure 1 depicts the processing pipeline in more detail. In Step (1) of this process, DynFD efficiently updates its data structures according to the changes in the batch (see Section 3.1). In doing so, DynFD does not need to perform potentially expensive read operations on the database. Not accessing the database is particularly important, also because reads would lead to race conditions with the changes applied by the database itself, i. e., a change that is being processed by DynFD might not have been (fully) applied by the database, yet, or the database may have already applied a subsequent change that DynFD has not yet seen. In Step (2), DynFD processes all deletes in the batch by checking whether they resolve any maximal non-FD in the negative cover (see Section 5); if a non-FD becomes a valid FD, this change is also propagated to the positive cover. Then, in Step (3), the algorithm processes all inserts by checking whether they introduce a violation to any minimal FD in the positive cover (see Section 4); in case they do, the positive cover is updated and the changes are propagated to the negative cover. Hence, due to the change propagation, Steps (2) and (3) may both affect both covers. In Step (4), DynFD finally signals all changed FDs to the user and is then ready to process the next batch.

The attentive reader may have noticed that we choose to process deletes before inserts, although the other way around is also possible. The decision on which type of operation to process first is particularly important for the special but common case of tuple updates, which we split into an insert and a delete. By processing the delete first, we avoid operating on an intermediate relation that contains both the old and the new version of the updated tuple. Such an almost duplicate tuple would violate many dependencies, in particular key dependencies, for the time of its existence. Hence, many FDs would change only to change back when the (almost) duplicate is removed again. So in short, processing deletes first significantly reduces the number of temporarily changing FDs.

3 DATA STRUCTURES

Rather than recalculating the FDs of a relation after each batch of changes, DynFD creates and maintains several data structures from which to derive the FDs. For that matter, we discern two types of data structures, namely those that represent the relation in a compact format that is suitable to efficiently check the validity of FD candidates; and the positive and negative cover, which we use to evolve the set of minimal FDs and maximal non-FDs, respectively. In the following, we briefly describe those data structures and how to update them in incremental scenarios.

3.1 Representing relations compactly

When validating FD candidates against a relation, the actual values within that relation are irrelevant. Instead, we merely need to know, which tuple pairs have identical values for which attributes. Therefore, it is sufficient for DynFD to represent the relation as *compressed records* [13]. Consider the example in Table 2, which represents the initial state of Table 1, i. e., before the change. Essentially, the compressed records replace all values of the original relation with a number that uniquely identifies that value within its column. For instance, the city name "Potsdam" is replaced by the value 0 and the city name "Berlin" is replaced by the value 1. This replacement not only has a small memory footprint but also allows for more efficient equality comparisons. This scheme is further optimized by replacing unique values with the value "−1". If DynFD encounters a "−1", it can skip all comparisons, as by definition all other tuples must have distinct values for the affected column.

Table 2: The dictionary-encoded example of Table 1.

ID	f	l	z	c
1	0	0	0	0
2	0	−1	0	0
3	0	0	−1	1
4	−1	−1	−1	1

While compressed records are well-suited to compare individual tuple pairs to quickly detect FD violations and rule out some FD candidates, they are not suitable to validate FD candidates as a whole. Therefore, DynFD complements the compressed records with *position list indexes* (Plis) [13], also known as stripped partitions [8]. In few words, a Pli lists the *clusters* of tuple IDs that have the same values for a certain attribute. For our example data from Table 1, we therefore obtain the Plis $\pi_f = \{\{1, 2, 3\}, \{4\}\}$ for firstname, $\pi_l = \{\{1, 3\}, \{2\}, \{4\}\}$ for lastname, $\pi_z = \{\{1, 2\}, \{3\}, \{4\}\}$ for zip, and $\pi_c = \{\{1, 2\}, \{3, 4\}\}$ for city. The Pli π_X for a set of attributes X can be computed via Pli intersection, i. e., by intersecting all pairs of overlapping clusters in the Plis π_Y and π_Z with $Y \cup Z = X$. Hence, $\pi_X = \pi_Y \cap \pi_Z$ and, for example, $\pi_{fc} = \pi_f \cap \pi_c = \{\{1, 2\}, \{3\}, \{4\}\}$. A functional dependency $X \longrightarrow A$ holds iff $\pi_X \cap \pi_A = \pi_X$, i. e., π_A does not split any cluster in π_X so that all records with equal values in X have also equal values in A [8]. Calculating $\pi_X \cap \pi_A$ can be done efficiently by using the compressed records: The (well-known) FD validation algorithm uses the Pli for some attribute $A \in X$ as an index to sets of tuples in the compressed records that are, then, grouped by same X values and checked against their A values. For more details and optimizations of this validation algorithm, we refer the interested reader to [13].

Compressed records and Plis synergize well in validating FD candidates, which has already been shown for static FD discovery. In a static setup, however, both data structures identify each record by its row number, i. e., position in the relational table. These row numbers, change in the dynamic setting, because the table grows and shrinks. For this reason, we assign a continuous number as a surrogate key to each record to identify it. The main challenge for using compressed records and Plis in the dynamic case is that we need to update these data structures with every batch of changes. For this, we propose the following:

Insert. A newly inserted record adds an entry to both data structures: We first add its identifier to all Plis. For every attribute, we read the attribute's value from the new record and fetch the attribute's Pli from the list of Plis. We then need to find the cluster in the Pli that corresponds to the said value and add the record's identifier to it. To find that cluster, the algorithm needs to remember the value of each cluster. It does so by storing and maintaining an additional *inverted index* on top of the Plis, i. e., the inverted index points each value to the Pli cluster in which it occurs. Following this mapping, it is easy to find the clusters were DynFD needs to add an identifier; if some value has no cluster in a Pli yet, it creates a new cluster. Given the cluster numbers of all attributes for the new record, updating the dictionary-encoded records is done by simply appending the array of these cluster numbers to the list of dictionary-encoded records.

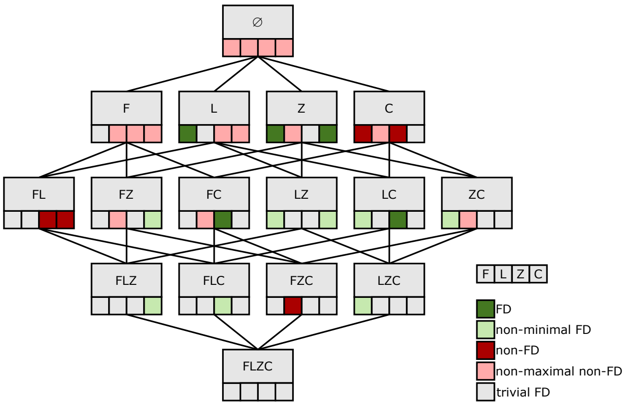
Delete. To delete a record from the data structures, we follow a similar, quick look-up strategy: For every attribute, DynFD first retrieves the Pli cluster that corresponds to the attribute's value in the deleted record. Then, it removes the identifier of the deleted record from these clusters; if a cluster becomes empty, the algorithm deletes the cluster from the Pli entirely. After the Pli update, DynFD also removes the record's compressed representation from the list of dictionary-encoded records. To find the record, we use another additional index, the *hash index*, that points the identifiers to their dictionary-encoded records. Alternatively, one could also find the record via binary search on the list of dictionary-encoded records, but the hash index has an infinitesimally small memory footprint in comparison to the other data structures and offers better performance than binary search. Once the compressed record is determined, DynFD deletes it from the list and the hash index.

3.2 Organizing functional dependencies

Let us now describe how DynFD organizes its discovered FDs and non-FDs. The FD search space is usually modeled as a *powerset lattice*, which is a graph representation of all possible attribute combinations. Due to the partial order of the power set, every two elements have a unique supremum and a unique infimum so that the graph can connect each node $X \subseteq R$ to its direct subsets $X \setminus A$ and direct supersets $X \cup \{B\}$ (with $A \in X$, $B \in R \backslash X$). Every node in the lattice represents one Lhs attribute combination for a set of FD candidates; each such FD candidate is defined by its Rhs attribute, which can bee seen as annotations at every node. In this way, all $2^m \cdot m$ possible FDs are covered by the lattice. During FD discovery, we classify each annotation and, hence, the respective Lhs $\longrightarrow$ Rhs candidate as either valid or invalid FD.

Figure 2 depicts the lattice of FD candidates for the initial state of our example relation from Table 1 (tuples 1 to 4). The five minimal FDs $l \longrightarrow f$, $z \longrightarrow f$, $z \longrightarrow c$, $fc \longrightarrow z$, and $lc \longrightarrow z$ have been discovered with a static profiling algorithm and we can infer all non-minimal and invalid FDs from them. To show how valid and invalid FDs are located in this search space visualization, all annotations have been color coded: Green cells represent valid FDs whereas red cells represent invalid FDs or short *non-FDs*. Stronger colors denote minimality for FDs and maximality for non-FDs. Note that a non-FD is maximal if no specialization of it is also a non-FD. Trivial FDs are shown in grey, because they are not of interest. So for example, we find the valid, minimal FD $fc \longrightarrow z$ as the dark green annotation for Rhs attribute Z in Lhs node *FC*.

Figure 2: FD lattice for the data shown in Table 1.



The complete set of minimal FDs is called the *positive cover*, because all non-minimal FDs can be derived from it. The complete set of maximal non-FDs, in contrast, is called *negative cover*, because it defines all existing non-FDs.

DynFD stores both the negative and the positive cover as *FD prefix trees*, which are technically prefix trees with annotations: Each node in the tree represents a Lhs attribute, any path starting from the root node represents a Lhs, and annotations on the nodes indicate valid Rhs attributes for the respective paths [6]. FD prefix trees are not only a compact data structure for storing FDs, they also offer efficient look-up functions for FD generalizations and specializations – functions that are called frequently by DynFD.

The positive cover, the Plis, and the dictionary compressed records represent the initial input for DynFD. By running the static FD discovery algorithm HyFD first, we can simply obtain all three data structures directly from that algorithm; otherwise, if only the set of minimal FDs is given, it is trivial to construct them in a preprocessing step. What is not trivial is the calculation of the negative cover, i. e., all maximal non-FDs, from the given FDs. The process of calculating the positive from the negative cover is known as *cover inversion* [6] or *dependency induction* [13], but its inverse, the calculation of the negative from the positive cover, has not been studied before. With Algorithm 1, we hence present the first inversion algorithm for this step.

We start with an empty FD prefix tree *nonFds* (line 1). For every attribute A of a relation R, the algorithm then adds the most specific non-FD, which states that all other attributes do not functionally determine A (lines 2-4). Initialized in this way, the negative cover invalidates all possible FDs and basically states that there are no FDs in the data. This initialization is probably not true, but serves as a starting point for successive refinement. Hence, the algorithm then checks for every valid FD whether it covers some non-FD by looking up all specializations of that FD in the negative cover (lines 5-6). This look-up is implemented as a simple depth-first search in the FD prefix tree. If a non-FD has been found to be a specialization of a valid FD, it must in fact be valid. For this reason, Algorithm 1 removes it from the negative cover (line 8). Then, we need to check all direct generalizations of the removed non-FD for being maximal non-FDs. The inversion algorithm creates each of these generalizations by removing one Lhs attribute from the current non-FD's Lhs (lines 9-11). If such a created non-FD is maximal, i. e., if it has no specialization in the cover, it is added to the negative cover (lines 12-13); otherwise, if

Algorithm 1: Cover inversion

```
Data: relation $R$, positive cover $fds$
Result: negative cover $nonFds$
$nonFds \leftarrow \emptyset$;
for $A \in R$ do
    $initialLhs \leftarrow R \setminus \{A\}$;
    $nonFds \leftarrow nonFds \cup \{initialLhs \rightarrow A\}$;
for $fd \in fds$ do
    $violated \leftarrow nonFds.getSpecializations(fd)$;
    for $nonFd \in violated$ do
        $nonFds \leftarrow nonFds \setminus \{nonFd\}$;
        for $l \in fd.getLhs()$ do
            $newLhs \leftarrow nonFd.getLhs() \setminus \{l\}$;
            $gen \leftarrow (newLhs \rightarrow nonFd.getRhs())$;
            if $\neg nonFds.containsSpecialization(gen)$ then
                $nonFds \leftarrow nonFds \cup \{gen\}$;
return $nonFds$
```

it is not maximal, it is simply discarded. Repeating this for every valid, minimal FD creates the negative cover.

Applied to the example data shown in Table 1, we start with the assumption that $flz \rightarrow c$, $flc \rightarrow z$, $fzc \rightarrow l$, and $lzc \rightarrow f$ are maximal non-FDs. Because the non-FD $lzc \rightarrow f$ is a specialization of the minimal FD $l \rightarrow f$, it needs to be removed as a non-FD. We then generalize it to $zc \rightarrow f$ as a new maximal non-FD candidate. This non-FD is a specialization of the minimal FD $z \rightarrow f$ (that we check next) so we generalize it once more, to $c \rightarrow f$. After also applying the remaining three minimal FDs to the negative cover, the inversion algorithms yields the final maximal non-FDs $fzc \rightarrow l$, $fl \rightarrow z$, $fl \rightarrow c$, $c \rightarrow f$, and $c \rightarrow z$.

4 HANDLING INSERTS

Having discussed DynFD's principal workflow along with its basic data structures, we now present a mechanism to maintain FDs in the light of insert operations (delete operations are discussed in the following section). Inserts can only render previously valid FDs invalid. Because the minimal FDs in the positive cover imply all valid FDs, they are the starting point for our validations. To process a batch of inserts, we validate all known minimal FDs and specialize them in case they became invalid. To ensure minimality for newly created specializations, we validate the minimal FDs from most general to most specific. In this way, we can check for each new specialization, if there is a valid generalization in the positive cover; the specialization would then not be minimal, because its generalization has already been determined to be true (minimality pruning). We discuss this main validation process, which is basically a *lattice traversal* type FD discovery approach, in Section 4.1.

To identify invalid FDs more quickly, we add an optimization to the validation process that, if the process becomes inefficient, progressively searches for FD violations. If a certain amount of FDs has been found invalid, the lattice traversal starts creating and checking many new candidates – most of which are usually invalid. For this situation, related work proposed *dependency induction* and sampling techniques that find most such violations faster than validating all candidates individually [13]. In Section 4.3, we propose an adapted version of these techniques for our dynamic setup.

Algorithm 2: Lattice-based FD validation

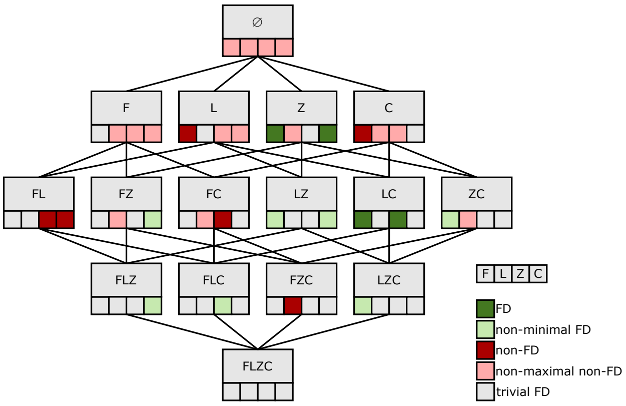
```
Data: relational schema $R$, positive cover $fds$, negative
      cover $nonFds$
Result: updated $fds$, $nonFds$
1  for $level \in 0...|R|$ do
2      $invalidFds \leftarrow \emptyset$;
3      for $fd \in fds.getLevel(level)$ do
4          if $\neg isValid(fd)$ then
5              $invalidFds \leftarrow invalidFds \cup \{fd\}$;
6      for $nonFd \in invalidFds$ do
7          $fds \leftarrow fds \setminus \{nonFd\}$;
8          $nonFds.removeGeneralizations(nonFd)$;
9          $nonFds \leftarrow nonFds \cup \{nonFd\}$;
10         $rhs \leftarrow nonFd.getRhs()$;
11         for $r \in R \setminus (nonFd.getLhs() \cup \{rhs\})$ do
12             $newLhs \leftarrow nonFd.getLhs() \cup \{r\}$;
13             $spec \leftarrow \langle newLhs \rightarrow rhs \rangle$;
14             if $\neg fds.containsGeneralization(spec)$ then
15                 $fds \leftarrow fds \cup \{spec\}$;
16     if $|invalidFds| \;/\; |fds.getLevel(level)| > 0.1$ then
17         $progressiveViolationSearch(fds, nonFds)$
18 return $fds$, $nonFds$
```

Note that whenever either of the two validation strategies, i. e., the lattice traversal or the dependency induction, discovers a non-FD, this non-FD must also be added to the negative cover. The process for updating the negative cover with a new non-FD covers two simple steps: First, remove all generalizations of the new non-FD from the cover (they are not maximal any more); then, add the new non-FD to the negative cover (without a check for specializations, because the non-FD used to be a valid FD before and is, therefore, inevitably maximal).

4.1 Incremental FD validation process

Algorithm 2 shows the lattice traversal-based FD validation algorithm that is executed for every batch in the dynamic setting. We start with the most general FDs in the positive cover and proceed to ever larger FDs (line 1). On each level of the lattice, the algorithm validates all minimal FDs and stores the invalid ones (lines 2-5). The validation function *isValid()* implements an optimized version of the Pli intersection technique that we touched on in Section 3.1; we discuss the optimization in Section 4.2. Iterating over all found non-FDs (line 6), the algorithm first removes each non-FD from the positive cover (line 7). As stated before, these non-FDs must also be maximal and are, therefore, added to the negative cover (lines 8-9). Afterwards, Algorithm 2 generates and adds all specializations of the current non-FD to the positive cover that are minimal w. r. t. the existing FDs (lines 10-15): To generate the specializations, we first add each attribute that is not already part of the Lhs or Rhs to the new Lhs (lines 11-13); each specialization is checked for generalizations in the positive cover before it is finally added to the cover (lines 14-15). On the next level, Algorithm 2 automatically validates these specializations. However, before moving to the next level, we check which fraction of validations in the current level has led to non-FDs (line 16). If this fraction is greater than 10% (see [13] for why this is a good threshold), then we consider the lattice traversal to be

Figure 3: Lattice after handling inserts in Table 1.



inefficient and start a progressive search for violations (line 17). When the search returns, it might yield updated versions of the positive and negative cover. Algorithm 2 then proceeds to the next lattice level until all minimal FDs have been checked.

Applying this to our example, we have the initial candidates $z \longrightarrow c$, $z \longrightarrow f$, $l \longrightarrow f$, $lc \longrightarrow z$, and $fc \longrightarrow z$. They are represented by the dark green cells in Figure 2. We start at the top of the lattice and work our way to the bottom. When validating the most general FDs, we find that $l \longrightarrow f$ is not valid anymore and, hence, a candidate for a maximal non-FD. The only new candidate is $lc \longrightarrow f$, since $lz \longrightarrow f$ is not minimal. Validating the now most general candidates shows that $fc \longrightarrow z$ is also invalid. There are no new candidates to be added. It also follows that $c \longrightarrow z$ is no maximal non-FD anymore and needs to be removed from the negative cover. Because no candidates are left, we are done and found the minimal FDs holding after inserting the new records. The corresponding lattice is shown in Figure 3.

4.2 Cluster pruning

To validate the minimal FDs in the positive cover, we build on the Pli-based validation algorithm presented in [13]: This algorithm uses the single-column Plis and dictionary-encoded records to dynamically calculate the Pli intersection of all Lhs attributes; at the same time, the algorithm checks the resulting clusters against the Rhs attribute clusters. If a check fails, i. e., if it reveals an FD violation, the algorithm terminates the validation process early. Furthermore, it performs this check for all FD candidates with the same Lhs simultaneously.

Our DynFD algorithm enhances this validation strategy: Instead of validating the FD against the entire dataset, we validate it against only the newly added and a few related records. Recall from Definition 1.1 that an FD is invalidated by a pair of records with equal values in the Lhs attributes but different values in the Rhs attribute. Because for inserts we validate only previously valid FDs, all pairs of *old* records still satisfy the FD. Only pairs of records containing at least one new record might introduce violations. Thus, the validation step for inserts needs to check such pairs only.

We integrate this optimization into our validation algorithm as follows: Given a fixed ordering of attributes by their respective Pli sizes, the validation starts by iterating the clusters of the Pli of the first Lhs attribute. For each cluster, the algorithm dynamically calculates the intersection with all other Lhs clusters to check the result against the Rhs clusters. At this point, which is, before calculating the intersections, DynFD first checks if the current cluster of the first Lhs attribute contains an identifier of a new record. This check is very simple, because the identifier numbers are assigned monotonically increasing (see Section 3.1) and the identifiers in each cluster are sorted. Hence, DynFD simply checks whether the last entry in the cluster is less than the identifier of the first insert-record in the current batch: If so, the cluster can be ignored; otherwise, the cluster contains at least one new record and needs to be checked via dynamically intersecting the Lhs attributes and probing the result against the Rhs attribute clusters.

As a result, DynFD's validation function *isValid()* (Algorithm 2 line 4) checks only the delta of the current batch for changing an FD and not the entire dataset. This optimization significantly improves the efficiency of the validation step, as many unnecessary comparisons are saved.

4.3 Violation search

If the lattice traversal becomes inefficient (Algorithm 2 line 16-17), DynFD switches to a strategy that we call *violation search*. This strategy is a best effort approach to find violations for formerly valid FDs via comparing records: Given two records r_i and r_j with their dictionary-encoded signature, we can easily compute the set of attributes X, in which r_i and r_j hold same values, and the set of attributes Y, in which r_i and r_j hold different values. It follows that $X \longrightarrow Y$ are all non-FDs.

Any newly inserted record can cause violations only with those partner records that have at least one value with the inserted record in common; records that do not share any value with the inserted record need not be considered. With DynFD's Plis, we can easily retrieve all those partner records by simply collecting all Pli clusters of the inserted record. Comparing the inserted record to all records in these clusters would, in fact, reveal all new violations, but the comparison costs are quadratic in the number of records, which is usually too expensive. For this reason, DynFD compares an inserted or changed record only to a small, promising subset of partner records.

Related work has shown that record pairs with possibly many overlapping values are promising candidates for finding new violations [13]. A sorting approach was demonstrated that moves pairs with high overlap closer together so that near neighborhoods become promising candidates. These neighborhoods are then progressively explored by moving ever larger windows over the sortings. If the violation search becomes inefficient, which is when less than 10% of the comparisons reveal new violations, the search ends. DynFD implements the exact same progressive search, but it compares only those record pairs that include at least one inserted (or updated) record.

For every discovered non-FD, DynFD needs to update both the positive and the negative cover. Algorithm 3 shows the necessary steps: It first updates the positive cover (lines 1-9) and then the negative cover (lines 10-13). To update the positive cover, the algorithm collects all invalidated FDs and removes them from the cover (lines 1-3). For each of these invalidated FDs, it also generates all direct specializations adding the minimal ones to the positive cover (lines 4-9). To update the negative cover, Algorithm 3 first checks if it contains a specialization (line 10); only if there is no specialization, the current non-FD is maximal and we add it to the negative cover (lines 11-12).

Algorithm 3: Dependency induction from a non-FD

```
Data: relational schema $R$, nonFd, positive cover fds,
      negative cover nonFds
Result: updated fds, nonFds
invalid ← fds.getGeneralizations(nonFd);
for fd ∈ invalid do
    fds ← fds \ {fd};
    rhs ← fd.getRhs();
    for $r \in R \setminus$ (nonFd.getLhs() ∪ {rhs}) do
        newLhs ← fd.getLhs() ∪ {r};
        spec ← (newLhs → rhs);
        if ¬fds.containsGeneralization(spec) then
            fds ← fds ∪ {spec};
if ¬nonFds.containsSpecialization(nonFd) then
    nonFds.removeGeneralizations(nonFd);
    nonFds ← nonFds ∪ {nonFd};
return fds, nonFds
```

5 HANDLING DELETES

To handle deletes, we propose a lattice traversal approach that validates the non-FDs in the negative cover level-wise starting with the most specific non-FDs and successively proceeding to more general ones. Because the maximal non-FDs imply all other non-FDs, the validation algorithm checks and, if necessary, generalizes only maximal non-FDs. Due to the level-wise iteration of the lattice, we can test any newly derived generalization for specializations in the negative cover to ensure that only maximal non-FDs are added back into the negative cover (maximality pruning). We describe this main validation process of the negative cover in more detail in Section 5.1 and its optimizations in Section 5.2 and Section 5.3.

5.1 Incremental non-FD validation process

Algorithm 4 shows the lattice traversal-based *non-FD* validation algorithm. This algorithm basically inverts the lattice traversal algorithm for inserts (see Algorithm 2): It operates on the negative cover instead of the positive cover (line 3), it traverses the cover from the most special non-FDs to the most general non-FDs instead of from most general to most special FDs (line 1), it transfers updates to the positive cover instead of to the negative cover (lines 8-9), and it generalizes de-facto-valid non-FDs instead specializing de-facto-invalid FDs (lines 10-14).

The validation function *isValid()* (line 4) is the same validation function that we already introduced for the validation of FDs in the insert scenario, but we now expect the outcomes to be mostly non-FDs. For non-FDs, DynFD adds an additional pruning technique to the validation that we explain in Section 5.2.

One major difference to the insert scenario, though, is that deleted records *resolve* violations and do not *introduce* them. For this reason, the progressive violation search, which compares promising record pairs in the quest for new non-FDs, makes no sense due to the lack of new non-FDs. Instead, we propose optimistic *depth-first searches* if the number of valid FDs exceeds 10% of all validated non-FDs in a current level (lines 15-16). We explain these optimistic depth-first searches in Section 5.3.

Applying the algorithm to our example, we start with the initial candidates for maximal non-FDs $fzc \rightarrow l$, $fl \rightarrow z$, $fl \rightarrow c$, $fc \rightarrow z$, $l \rightarrow f$, and $c \rightarrow f$. They are the dark red cells in Figure 3 and we traverse the lattice from bottom to top. Starting with the most specific candidate, nothing changes. In the next step, it turns out that both $fl \rightarrow z$ and $fl \rightarrow c$ become valid. Thus, they are both candidates for new minimal FDs. Furthermore, we need to add the generalizations $f \rightarrow c$, $l \rightarrow z$, and $l \rightarrow c$ to the negative cover. $f \rightarrow z$ is not maximal and therefore not a new candidate. Validating the remaining five candidates shows that also $f \rightarrow c$ is valid and thus a candidate for a new minimal FD. $fl \rightarrow c$ is not a minimal FD anymore. There are no new candidates for maximal non-FDs and we end up with six minimal FDs. The corresponding lattice is shown in Figure 4.

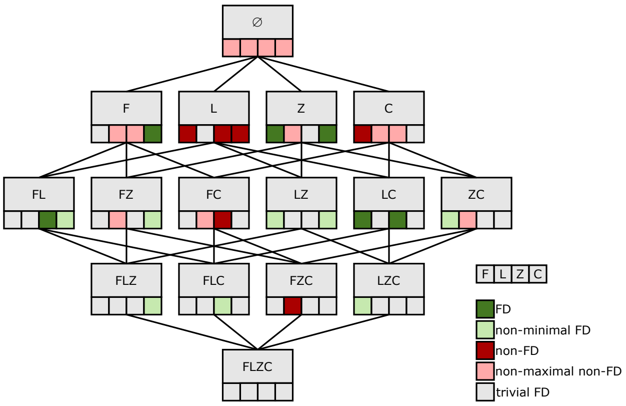
Algorithm 4: Lattice-based non-FD validation

```
Data: relational schema $R$, positive cover fds, negative
      cover nonFds
Result: updated fds, nonFds
1  for level ∈ |R|...0 do
2      validFds ← ∅;
3      for nonFd ∈ nonFds.getLevel(level) do
4          if needsValidation(nonFd) ∧ isValid(nonFd) then
5              validFds ← validFds ∪ {nonFd};
6      for fd ∈ validFds do
7          nonFds ← nonFds \ {fd};
8          fds.removeSpecializations(fd);
9          fds ← fds ∪ {fd};
10         for r ∈ fd.getLhs() do
11             newLhs ← fd.getLhs() \ {r};
12             gen ← (newLhs → fd.getRhs());
13             if ¬nonFds.containsSpecialization(gen) then
14                 nonFds ← nonFds ∪ {gen};
15     if |validFds| / |nonFds.getLevel(level)| > 0.1 then
16         depthFirstSearch(validFds, fds, nonFds)
17 return fds, nonFds
```

5.2 Validation pruning

Most FD candidates that we validate in the delete scenario are non-FDs and the purpose of validation is to confirm that there is still at least one violation to each candidate. Although the validation algorithm terminates as soon as it finds the first violation to a candidate, in many cases it still checks a lot of matching

Figure 4: Lattice after validating non-FDs



Algorithm 5: Depth first search for FDs

```
Data: fd, positive cover fds, negative cover nonFds
Result: updated fds, nonFds
1  for r ∈ fd.getLhs() do
2      newLhs ← fd.getLhs() \ {r};
3      newFd ← (newLhs → rhs);
4      if fds.containsGeneralization(newFD) ∨ isValid(newFd) then
5          depthFirst(newFd, fds, nonFds);
6  deduceNonFds(fd, fds, nonFds);
7  return fds, nonFds
```

value combinations, which is expensive. To avoid many of these checks, DynFD stores a violating record pair, which is a pair of two identifiers whose records contradict the FD, as a *surrogate violation* for every maximal non-FD in the negative cover. As long as these two records exist in the data, the algorithm does not need to check the corresponding non-FD.

So what DynFD does is the following: Whenever the algorithm creates a non-FD, it also attaches the record pair that made the FD invalid to the respective non-FD lattice node. When this lattice node needs to be validated (see Algorithm 4 line 4), the algorithm first calls the function *needsValidation()* to check whether one of its two attached records was deleted in the current batch. In case both records are still present, which is usually true, no validation is needed; otherwise, the algorithm has to run the validation to, depending on the result, either attach a new violating record pair or remove the non-FD.

In order to consistently attach violating record pairs to all non-FDs in the negative cover throughout the dynamic discovery process, we need to consider two procedures in DynFD that identify new non-FDs: the candidate validation (*isValid()* function) and the sampling (Section 4.3). Both procedures know a violating record pair whenever an FD candidate is invalidated so that they can simply attach this record pair to a new non-FD. Conversely, if records are deleted, they need to be consistently removed from the non-FDs. For this purpose, we index all non-FD annotations (*recordID* → *nonFD*) and, for every batch of deletes, remove from the negative cover all record identifiers (and their respective partner record identifiers) if a batch deletes them. The initial non-FD annotations in the negative cover are calculated on the fly with the first batch, because whenever a maximal non-FD lacks a violation annotation, Algorithm 4 validates it anyway.

5.3 Depth-first searches

If a prior non-FD becomes valid, we successively check all its generalizations. These checks can continue for many levels in the lattice and stretch out to an exponential number of candidates (exponential in the number of attributes); this makes the validation very expensive. The new non-FDs are, however, often covered by only a few maximal non-FDs. Hence, we propose optimistic depth-first searches that target maximal non-FDs of small Lhs-arity; these non-FDs prune many candidate non-FDs from the lattice.

If more than 10% of the non-FDs in one level of the negative cover became true FDs (Algorithm 4 line 15), we start the optimistic depth-first search. This subroutine takes the *validFds*, which are the former non-FDs that have been found valid, as input. For a sample of 10% of these *seed FDs*, DynFD aggressively searches their generalizations for new maximal non-FDs. We consider only a sample of the seed FDs, because the depth-first searches are an optimistic optimization attempt and should not change the search strategy entirely – most FDs still change only a bit making breath-first search in general more effective. The 10% efficiency threshold and the 10% seed sample are hard-coded parameters that have shown to be efficient settings for most datasets. The non-FDs discovered in the depth-first searches might be new overall maximal non-FDs and are, hence, used to update both the negative cover *nonFds* and the positive cover *fds*.

Algorithm 6: Dependency induction from a FD

```
Data: fd, positive cover fds, negative cover nonFds
Result: updated fds, nonFds
1  valid ← nonFds.getSpecializations(fd);
2  for nonFd ∈ valid do
3      nonFds ← nonFds \ {nonFd};
4      rhs ← nonFd.getRhs();
5      for r ∈ fd.getLhs() do
6          newLhs ← nonFd.getLhs() \ {r};
7          gen ← (newLhs → rhs);
8          if ¬nonFds.containsSpecialization(gen) then
9              nonFds ← nonFds ∪ {gen};
10 if ¬fds.containsGeneralization(fd) then
11     fds.removeSpecializations(fd);
12     fds ← fds ∪ {fd};
13 return fds, nonFds
```

The algorithm that performs an optimistic depth-first search for one seed FD is depicted in Algorithm 5. It implements a recursive depth-first traversal of the positive cover starting with the seed FD *fd*. Given a valid FD, Algorithm 5 first generates all its direct generalizations by gradually removing each attribute from the Lhs (lines 1-3). For each generalization, the algorithm then checks if it has an own generalization in the positive cover. In that case, the generalization is true and it must not be validated; otherwise, Algorithm 5 validates the generalization (line 4). For every valid generalization, we continue the depth-first search (line 5). After handling all generalizations, the current FD is used to deduce new non-FDs in the negative cover (line 6). This step is done at last, because the deduction is expensive and, if some more general FDs have already been used for deduction in some recursion, there is less to be deduced for the current FD.

The function *deduceNonFds()* updates both negative and positive cover with a new, true FD. It is basically an exact inversion of the deduction Algorithm 3 for non-FDs as we now specialize the negative cover and generalize the positive cover accordingly. Technically, we switch negative and positive cover, and generalization and specialization, which yields Algorithm 6. This algorithm starts by retrieving all non-FD specializations of the known FD from the negative cover (line 1). All these FDs are valid now. Hence, it then removes each such FD from the negative cover (lines 2-3). To generalize the new valid FDs into possibly true non-FDs, the algorithm removes each attribute of the valid FD's Lhs once (lines 5-7). If such a generalization is maximal, it is added to the negative cover (lines 8-9). After updating the negative cover, Algorithm 6 also updates the positive cover with the known FD by adding the FD to the positive cover and removing all its specializations (lines 10-12).

6 EVALUATION

We now evaluate the performance of our FD maintenance algorithm DynFD on multiple real-world datasets. The evaluation covers a detailed analysis of our proposed pruning rules and techniques and compares DynFD to the repeated execution of HyFD, the state-of-the-art FD discovery algorithm for static setups.

To evaluate DynFD, we need some datasets with a change history and a special experimental setup. That is what we discuss first. We then evaluate DynFD's batch processing times and its throughput. Afterwards, we analyze the algorithm's performance w. r. t. different batch sizes and, then, compare the batch times to HyFD's batch times. As a final set of experiments, we evaluate our four main pruning strategies: *cluster pruning*, *violation search*, *validation pruning*, and *depth-first searches.*

6.1 Datasets and experimental setup

Datasets. For our experiments, we use six real-world datasets and their change history: The *artist* relation from the MusicBrainz database [9], the *claims* relation from the airport baggage claims dataset published by Homeland Security [15], and the Wikipedia infobox relations *cpu*, *disease*, *actor*, and *single* put together by Google [7]. The six datasets and their characteristics, i. e., number of columns, rows, changes, and FDs, are listed in Table 3. The insert, delete, and update percentage columns refer to the number of changes that we have for these datasets. For each dataset, we highlight the characteristics that make the dataset interesting: The selection contains wide (*actor*) and long (*artist*) relations, ones with insert (*single*), and update (*cpu*) heavy changes, and a dataset with particularly many changes (*claims* and *disease*). Hence, it fairly represents real-world data in general.

Changes. The six datasets are given as a series of dataset dumps in different versions. Because DynFD requires the individual change operations that transformed one version into its successor version, we extracted all inserts, deletes, and updates from the change history of each dataset. The first version in each history serves as the initial dataset and the sequence of changes broken down into fixed-sized batches constitutes the dynamic input for the FD maintenance task. In practice, the size of the batches depends on the change rate, the size of the data, and use case specific currentness requirements; in our experiments, we test different batch sizes and their impact on the FD maintenance performance.

Experiments. Given the initial dataset and the batched sequence of changes, all experiments process the entire sequence of batches as fast as possible. This gives us an upper bound for the throughput performance: If the actual change rate is higher, the maintenance would apply back-pressure and throttle the database. The research question is, though, what throughput we can achieve.

Hardware. All experiments have been executed on a Dell PowerEdge R620 with two Intel Xeon E5-2650 2.00 GHz CPUs and 128 GB RAM. This server runs on CentOS 6.4 and uses OpenJDK 64-Bit 1.8.0_111 as Java environment.

6.2 Batch processing performance

In this first experiment, we evaluate the batch processing performance of DynFD on all datasets by fixing the batch size to 100 and measuring the processing time of each such batch. We run up to 100 batches of each change history, which corresponds to 10,000 changes per dataset; for *cpu* and *actor*, we process only 1,463 and 5,647 changes respectively, because this is their entire change history length. The results are shown in Table 4.

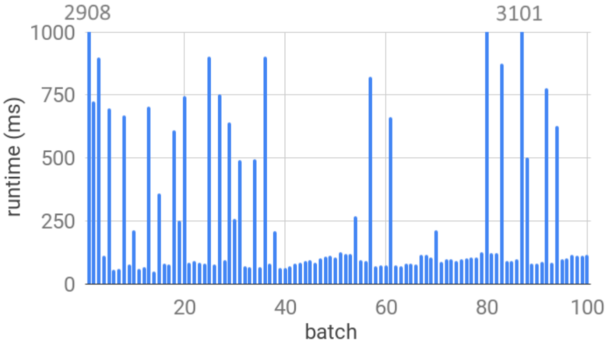


Figure 5: Runtime per batch (size 100) on *single*.

The measurements show that both the accumulated *runtime* and the *throughput* are affected by the width (#Column) and the length (#Rows) of the data: Although *single* has about three times more rows than *actor*, *actor*'s three times more columns result in almost half the throughput performance; if the number of rows is, however, significantly larger as for the *artist* dataset, the throughput also drops significantly, i. e., to only 17 changes per second. Considering the complexity of FD discovery, which is also the complexity of processing each batch, this observation is no surprise. Note that the comparably low throughput for *cpu* is due to the fact that the dataset is very short (62 rows) so that every batch (100 changes) basically rewrites the entire dataset.

The *average batch times* together with the *percentile* times show that the batch times have many outliers, i. e., runtime spikes that differ greatly from the average runtime of a batch. For most batches, the set of minimal FDs does not change much and our maintenance strategies do a good job skipping through these batches; for some batches, however, the FDs do change, sometimes even significantly. Then, DynFD needs to evolve FDs and invest some additional effort, which we tried to minimize with our pruning strategies.

Figure 5 supports this observation by plotting the individual batch times for the *single* dataset: The majority of batches is processed very quickly while some batches take orders of magnitude longer. This plot looks similar for the other datasets, namely a default batch processing time with occasional runtime spikes.

6.3 Batch size scalability

In the previous experiment, we fixed the batch size to 100 changes per batch. To see the impact of the batch size, we now scale it from 10 to 1,000 changes per batch. Figure 6 shows the average runtime of DynFD per batch w. r. t. the different batch sizes. The average in this experiment is calculated over the first 10,000 changes per dataset and both axes of the chart are in log-scale.

Most batch processing costs, such as the costs for updating the data structure and for checking whether the deletes resolved any violations, scale linearly with the size of the batch. The experimental results in Figure 6, however, show that 100 times more changes in a batch cause only about 10 times longer batch times on all datasets. This means that increasing the batch size also increases the throughput, as the processing costs per change decrease. This is because some batch activities, such as the expensive validation of the positive cover, constitute constant costs per batch regardless of its size (if no FD changes). The number of

Table 3: Characteristics of the datasets used in our evaluation.

Dataset	#Columns	#Rows (initial)	#Changes	#FDs (initial)	#FDs (final)	%Inserts	%Deletes	%Updates
cpu	15	62	1,463	209	327	4.3	0.2	**95.5**
disease	13	1,600	**361,828**	23	29	1.0	0.6	98.4
actor	**83**	3,655	5,647	347	326	64.9	0.5	34.6
single	26	12,451	12,614	193	248	**96.1**	0.0	3.9
artist	18	**1,122,887**	25,470	226	278	61.8	3.7	34.5
claims	8	1054	**202,913**	32	3	100.0	0.0	0.0

Table 4: Performance of DynFD on all datasets.

Dataset	runtime [sec]	throughput [changes/sec]	avg batch time [ms]	99th percentile [ms]	95th percentile [ms]	90th percentile [ms]
cpu	1.1	1,318.0	74.0	201.4	158.8	116.8
disease	1.5	6,844.6	14.6	62.2	26.4	19.0
actor	25.9	218.0	454.5	1,375.9	1,132.2	931.0
single	26.8	373.0	268.0	2,715.8	834.1	717.1
artist	577.1	17.3	5,771.0	15,033.1	13,979.2	12,367.9
claims	1.8	5,602.2	17.9	89.6	56.2	44.1

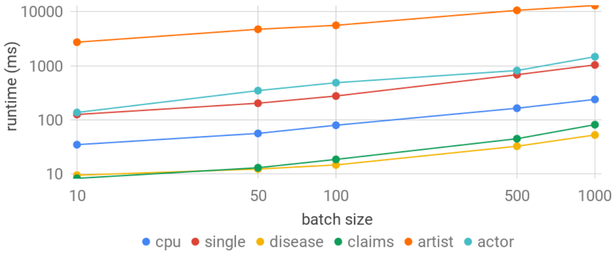


Figure 6: Average runtime for different batch sizes.

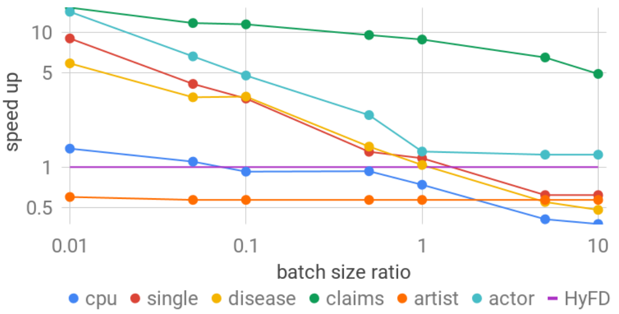


Figure 7: Speedup of DynFD relative to the runtime of HyFD.

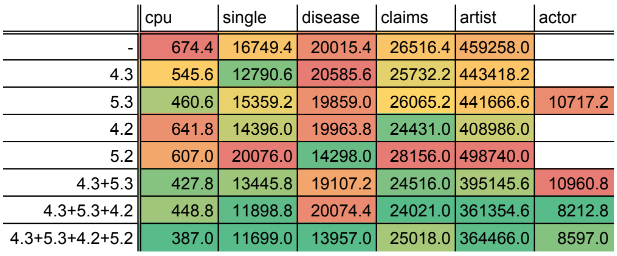
changing FDs per batch does also not increase linearly with the batch size: Intuitively, if we look at the data less often, we probably overlook some changes, e. g., if $A \longrightarrow B$ changes to $AC \longrightarrow B$ and then to $ACD \longrightarrow B$, we might *observe* only the change of $A \longrightarrow B$ to $ACD \longrightarrow B$. Hence, we observe that the average batch processing time increases sub-linearly when increasing the batch size in DynFD. It is, furthermore, remarkable that the runtime increase is similar for all datasets although they differ greatly in size and change patterns.

6.4 Competitive evaluation

Our next experiment compares the batch processing performance of DynFD to repeated executions of the static profiling algorithm HyFD. For this comparison, it is especially interesting to see for which batch sizes our dynamic algorithm exhibits superior performance compared to a static profiling approach. For this purpose, we now scale it up from small to excessively large. We also now define the batch size relative to the initial dataset size, to make the measurements comparable across differently sized datasets: We start with a batch size containing as many changes as 1% of the initial dataset length, i. e., 1% of #Rows, and increase the batch size to 1000% of initial dataset size. Then, we plot the runtime of DynFD relative to the runtime of HyFD for each dataset, e. g., a speedup of 5 indicates that DynFD was 5 times faster than HyFD with a particular batch size ratio on a particular dataset; 1 is equally fast and speedups smaller than 1 are slower than HyFD.

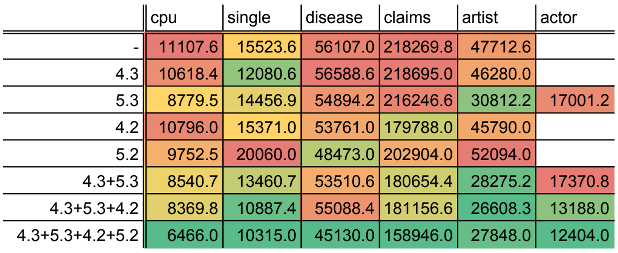
The results depicted in Figure 7 show that DynFD is more than an order of magnitude faster than HyFD for small and medium batch sizes. This superiority decreases as we increase the batch size – until HyFD becomes faster for most datasets. DynFD's poor performance on *artist* is due to the fact that a batch size of 1% initial dataset size already covers 11,228 changes – about half of the entire change history. The first batch in each experiment is also more expensive than later batches, because this is when DynFD collects the initial violation annotations for the negative cover. Because 10% batch size ratio already covers the entire change history for *artist*, the performance does not change after that measurement. For *cpu*, we observe the opposite effect: The dataset is so tiny that simply re-profiling it with every batch is the best option anyway.

According to the measurements for *disease*, *single*, and *actor*, the inflection point at which static profiling (with HyFD) tends to overtake dynamic profiling (with DynFD) is at about 100% batch size ratio, i. e., when a batch basically re-writes the entire dataset.



	cpu	single	disease	claims	artist	actor
-	674.4	16749.4	20015.4	26516.4	459258.0	
4.3	545.6	12790.6	20585.6	25732.2	443418.2	
5.3	460.6	15359.2	19859.0	26065.2	441666.6	10717.2
4.2	641.8	14396.0	19963.8	24431.0	408986.0	
5.2	607.0	20076.0	14298.0	28156.0	498740.0	
4.3+5.3	427.8	13445.8	19107.2	24516.0	395145.6	10960.8
4.3+5.3+4.2	448.8	11898.8	20074.4	24021.0	361354.6	8212.8
4.3+5.3+4.2+5.2	387.0	11699.0	13957.0	25018.0	364466.0	8597.0

Figure 8: Runtime with different sets of pruning strategies and a fixed batch size of 1,000.



	cpu	single	disease	claims	artist	actor
-	11107.6	15523.6	56107.0	218269.8	47712.6	
4.3	10618.4	12080.6	56588.6	218695.0	46280.0	
5.3	8779.5	14456.9	54894.2	216246.6	30812.2	17001.2
4.2	10796.0	15371.0	53761.0	179788.0	45790.0	
5.2	9752.5	20060.0	48473.0	202904.0	52094.0	
4.3+5.3	8540.7	13460.7	53510.6	180654.4	28275.2	17370.8
4.3+5.3+4.2	8369.8	10887.4	55088.4	181156.6	26608.3	13188.0
4.3+5.3+4.2+5.2	6466.0	10315.0	45130.0	158946.0	27848.0	12404.0

Figure 9: Runtime with different sets of pruning strategies and a relative batch size of 10% initial dataset size.

6.5 In-depth performance analysis

DynFD proposes an FD maintenance algorithm with four major pruning strategies. The basic algorithm performs the incremental data structure updates and the level-wise valuations of the positive and negative cover; it basically enables the dynamic evolution of FDs. The four pruning strategies, which are *cluster pruning* (see Section 4.2), *violation search* (see Section 4.3), *validation pruning* (see Section 5.2), and *depth-first searches* (see Section 5.3), aim to reduce the maintenance effort as much as possible. Let us now evaluate how effective each individual strategy is and what the performance implications are.

In this final set of experiments, we execute DynFD with different sets of pruning strategies on all datasets. Because the *violation search* is so important for the algorithm, i. e., the performance drops significantly without any form of this strategy, we let the baseline algorithm run a naive sampling that compares changed records only to their direct neighbors w. r. t. some sorting. For each combination of strategies and dataset, we measure the processing time for all batches of fixed size 1,000 (Figure 8) and relative size 10% (Figure 9), respectively.

The measurements for both batch sizes show that the composition of all pruning strategies performs best in general. It is not the optimal composition of pruning strategies for all datasets, but the performance is reliably good throughout all our datasets. With a few exceptions, every strategy tends to improve the performance a bit. One exception is the *violation search* (4.3) on the *disease* dataset, because a naive version of this strategy is already in use by the baseline; and the optimized version, which is the version that runs progressively increasing windows, does not improve upon this naive strategy. The second exception is the *validation pruning* (5.2) that performs poorly on *claims*, *single*, and *artist*: On *claims*, the batches simply do not contain any deletes so that the strategy introduces the overhead of labeling non-FDs with violations without ever needing them. On *single* and *artist*, the annotations did not effectively prevent violations, because many of the annotated violations did vanish, i. e., the violations in these datasets are simply not stable enough to prevent enough violations that could balance the overhead of maintaining the violations.

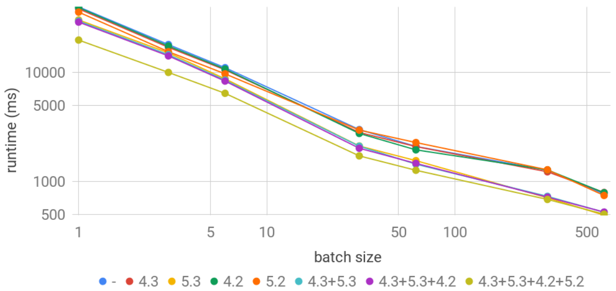


Figure 10: Runtime on *cpu* with different sets of pruning strategies and different batch sizes.

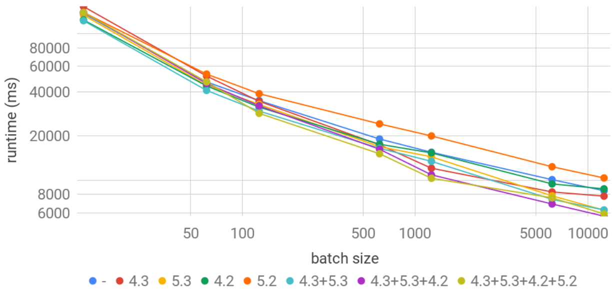


Figure 11: Runtime on *single* with different sets of pruning strategies and different batch sizes.

The measurements in Figure 10 and Figure 11 present the runtimes of the different strategy compositions for different batch sizes. The lines show that the composition of *all* pruning strategies also performs reliably well, i. e., best or close to best, regardless of the batch size.

7 RELATED WORK

DynFD is the first algorithm to maintain FDs under inserts, updates, and deletes in dynamic data. Nonetheless, we identify two categories of related work, namely (i) the discovery of FDs in static data and (ii) the maintenance of metadata in dynamic data in general.

7.1 Discovering FDs in static data

Research has brought up many FD discovery algorithms for static data, which can be classified into *column-based*, *row-based*, and *hybrid* algorithms [12]. In the following, we briefly describe one popular representative of each class.

One of the earliest FD discovery algorithms is the column-based algorithm Tane that models the search space, i. e., the set of all candidate FDs, as a powerset lattice of attribute combinations [8]. It traverses this lattice in a level-wise bottom-up fashion until it arrives at the non-trivial, minimal FDs. Although, DynFD adopts the lattice-shaped search space, it traverses it both bottom-up and top-down in response to data changes. For the candidate validation, Tane proposed *stripped partitions* (also: *position list indexes*, Plis) that we also use in DynFD.

The row-based FDEP algorithm proposes a fundamentally different strategy [6]: It compares all pairs of records in the input relation to deduce the complete *negative cover*, i. e., all candidate FDs that are violated by some tuple pair. The non-trivial, minimal FDs are derived from the negative cover via cover inversion. DYNFD also maintains a negative cover, but for the purpose of processing tuple deletions and not to infer the positive cover.

The hybrid algorithm HYFD combines column- and row-based techniques to avoid possibly many ineffective candidate validations and tuple comparisons [13]. By interleaving the two discovery principles and by having them exchange intermediate results, HYFD significantly outperforms all non-hybrid competitors. In DYNFD, we tailor the hybrid discovery approach to work on dynamic data. Our experiments show that the proposed extensions and adjustments have a great, positive impact on the performance of the approach.

7.2 Maintaining metadata in dynamic data

Arguably, maintaining metadata in dynamic data has received much less attention than static data profiling. Therefore, this section widens its scope to maintaining *any* kind of metadata, not only FDs.

To the best of our knowledge, the only existing FD maintenance algorithm was proposed by Wang et al. in [17]. The algorithm deals only with tuple deletions and neither inserts nor updates. Similar to DYNFD, the algorithm uses bottom-up and top-down approaches as well as indexes to evolve the FDs. Unlike DYNFD, however, it does not maintain a negative cover of non-FDs that significantly improves and distinguishes the handling of deletes from the static case.

The SWAN algorithm by Abedjan et al. is an incremental discovery algorithm for unique column combinations (UCCs), i. e., key candidates in dynamic datasets [1]. Starting from a pre-calculated set of UCCs, SWAN actively applies all insertions and deletions to the current metadata set. The algorithm groups change operations into batches and uses various indexes to optimize the change calculations – two principles that are also used by DYNFD. In contrast to SWAN, however, DYNFD operates on both a positive and a negative cover representation of the metadata, which enable additional pruning strategies.

Lastly, Shaabani and Meinel proposed an incremental algorithm to maintain inclusion dependencies (INDs) in dynamic data [16]. The algorithm uses a concept called *attribute clustering* that can be used in an incremental setup to evolve INDs w. r. t. sets of data changes. Besides the fact that this algorithm also handles both inserts and updates in batches, it has little in common with DYNFD, because IND discovery is very different from FD discovery.

8 CONCLUSION

In this paper, we introduced DYNFD, a novel algorithm that maintains the functional dependencies of dynamic datasets. To evolve the set of FDs with every batch of inserts, updates, and deletes, the algorithm continuously adapts its validation structures as well as a negative and a positive cover of FDs. With DYNFD, we proposed a new cover inversion algorithm and four pruning strategies that stabilize the maintenance performance. Our evaluation shows that, if the batch size is small, the dynamic algorithm is more than an order of magnitude faster than repeated executions of a state-of-the-art, static profiling algorithm.

Because profiling static data is already a challenging task, profiling dynamic data is a research direction that has not been studied much thus far. With DYNFD, we made a first approach to solve this task for functional dependencies. Our experiments show that this algorithm greatly improves upon using static data profiling algorithms in dynamic scenarios, but it leaves several interesting research questions open:

(1) *Devising a measure for interestingness* could serve to track only interesting and, hence, fewer dependencies; this would greatly improve the maintenance performance.

(2) *Incorporating knowledge about existing database constraints* into the maintenance process could help to prune further validations; FDs with a key constraint on their LHS, for instance, cannot invalidate.

(3) *Exploiting the specifics of update operations*, such as the fact that most updates do not alter all attribute values but only a few, could help to devise further pruning rules.

REFERENCES

[1] Z. Abedjan, J. Quiané-Ruiz, and F. Naumann. Detecting unique column combinations on dynamic data. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1036–1047, 2014.

[2] P. Bohannon, W. Fan, and F. Geerts. Conditional functional dependencies for data cleaning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 746–755, 2007.

[3] C. R. Carlson, A. K. Arora, and M. M. Carlson. The application of functional dependency theory to relational databases. *The Computer Journal*, 25(1):68–73, 1982.

[4] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[5] E. F. Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971.

[6] P. A. Flach and I. Savnik. Database dependency discovery: a machine learning approach. *AI Communications*, 12(3):139–160, 1999.

[7] Google. Distributing the Edit History of Wikipedia Infoboxes. https://research.googleblog.com/2013/05/distributing-edit-history-of-wikipedia.html, 2013. [Online; accessed 8-October-2018].

[8] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[9] Internet Archive. MusicBrainz Data Dumps. https://archive.org/details/musicbrainzdata, 2018. [Online; accessed 8-October-2018].

[10] J. Liu, J. Li, C. Liu, and Y. Chen. Discover dependencies from data - a review. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(2):251–264, 2012.

[11] R. J. Miller, M. A. Hernández, L. M. Haas, L. Yan, C. T. H. Ho, R. Fagin, and L. Popa. The Clio project: Managing heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

[12] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proceedings of the VLDB Endowment (PVLDB)*, 8(10):1082–1093, 2015.

[13] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 821–833, 2016.

[14] G. N. Paulley. *Exploiting Functional Dependence in Query Optimization.* PhD thesis, University of Waterloo, 2000.

[15] H. Security. TSA Claims Data. https://www.dhs.gov/tsa-claims-data, 2018. [Online; accessed 8-October-2018].

[16] N. Shaabani and C. Meinel. Incremental discovery of inclusion dependencies. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 2:1–2:12, 2017.

[17] S.-L. Wang, W.-C. Tsou, J.-H. Lin, and T.-P. Hong. *Maintenance of Discovered Functional Dependencies: Incremental Deletion*, pages 579–588. Springer, Heidelberg, 2003.

CLX: Towards verifiable PBE data transformation

Zhongjun Jin[1] Michael Cafarella[1] H. V. Jagadish[1] Sean Kandel[2]
Michael Minar[2] Joseph M. Hellerstein[2,3]
[1]University of Michigan, Ann Arbor [2]Trifacta Inc. [3]UC Berkeley
{markjin,michjc,jag}@umich.edu,{skandel,mminar}@trifacta.com,hellerstein@berkeley.edu

ABSTRACT

Effective data analytics on data collected from the real world usually begins with a notoriously expensive pre-processing step of data transformation and wrangling. Programming By Example (PBE) systems have been proposed to automatically infer transformations using simple examples that users provide as hints. However, an important usability issue—**verification**—limits the effective use of such PBE data transformation systems, since the verification process is often effort-consuming and unreliable.

We propose a data transformation paradigm design CLX (pronounced "clicks") with a focus on facilitating verification for end users in a PBE-like data transformation. CLX performs pattern clustering in both input and output data, which allows the user to verify at the pattern level, rather than the data instance level, without having to write any regular expressions, thereby significantly reducing user verification effort. Thereafter, CLX automatically generates transformation programs as regular-expression replace operations that are easy for average users to verify.

We experimentally compared the CLX prototype with both FlashFill, a state-of-the-art PBE data transformation tool, and Trifacta, an influential system supporting interactive data transformation. The results show improvements over the state of the art tools in saving user verification effort, without loss of efficiency or expressive power. In a user study on data sets of various sizes, when the data size grew by a factor of 30, the user verification time required by the CLX prototype grew by 1.3× whereas that required by FlashFill grew by 11.4×. In another user study assessing the users' understanding of the transformation logic — a key ingredient in effective verification — CLX users achieved a success rate about twice that of FlashFill users.

1 INTRODUCTION

Data transformation, or data wrangling, is a critical pre-processing step essential to effective data analytics on real-world data and is widely known to be human-intensive as it usually requires professionals to write ad-hoc scripts that are difficult to understand and maintain. A *human-in-the-loop* Programming By Example (PBE) approach has been shown to reduce the burden for the end user: in projects such as FlashFill [6], BlinkFill [24], and Foofah [11], the system synthesizes data transformation programs using simple examples the user provides.

Problems — Most of existing research in PBE data transformation tools has focused on the "system" part — improving the efficiency and expressivity of the program synthesis techniques. Although these systems have demonstrated some success in efficiently generating high-quality data transformation programs for real-world data sets, **verification**, as an indispensable interaction procedure in PBE, remains a major bottleneck within existing PBE data transformation system designs. The high labor cost, may deter the user from confidently using these tools.

Any reasonable user who needs to perform data transformation should certainly care about the "correctness" of the inferred transformation logic. In fact, a user will typically go through rounds of "verify-and-specify" cycles when using a PBE system. In each interaction, a user has to verify the correctness of the current inferred transformation logic by validating the transformed data instance by instance until she identifies a data instance mistakenly transformed; then she has to provide a new example for correction. **Given a potentially large and varied input data set, such a verification process is like "finding a needle in a haystack" which can be extremely time-consuming and tedious.**

A naïve way to simplify the cumbersome verification process is to add explanations to the transformed data so that the user does not have to read them in their raw form. For example, if we can somehow know the desired data pattern, we can write a checking function to automatically check if the post-transformed data satisfies the desired pattern, and highlight data entries that are not correctly transformed.

However, a *data explanation* procedure alone can not solve the entire verification issue; the undisclosed transformation logic remains untrustworthy to the end user. Users can at best verify that existing data are converted into the right form, but **the logic is not guaranteed to be correct and may function unexpectedly on new input** (see Section 2 for an example). Without good insight into the transformation logic, PBE system users cannot tell if the inferred transformation logic is correct, or when there are errors in the logic, they may not be able to debug it. **If the user of a traditional PBE system lacks good understanding of the synthesize program's logic, she can only verify it by spending large amounts of time testing the synthesized program on ever-larger datasets**.

Naïvely, previous PBE systems can support *program explanation* by presenting the inferred programs to end users. However, these data transformation systems usually design their own Domain Specific Languages (DSLs), which are usually sophisticated. The steep learning curve makes it unrealistic for most users to quickly understand the actual logic behind the inferred programs. Thus, besides more explainable data, a desirable PBE system should be able to present the transformation logic in a way that most people are already familiar with.

Insight — Regular expressions (regexp) have been known to most programmers of various expertise and regexp replace operations have been commonly applied in data transformations. The influential data transformation system, Wrangler (later as Trifacta), proposes simplified natural-language-like regular expressions which can be understood and used even by non-technical data analysts. This makes regexp replace operations a good choice for an *explainable transformation language*. The challenge then is how to automatically synthesize regexp replace operations as the desired transformation logic in a PBE system.

A regexp replace operation takes in two parameters: an *input pattern* and a *replacement function*. Suppose an input data set is given, and the desired data pattern can be known, the challenge is to determine a suitable input pattern and the replacement function to convert all input data into the desired pattern. Moreover, if the input data set is heterogeneous with many formats, we need to find out an unknown set of such input-pattern-and-replace-function pairs.

Pattern profiling can be used to discover clusters of data patterns within a data set that are are useful to generate regular replace operations. Moreover, it can also serve as a data explanation approach helping the user quickly understand the pre- and post-transformation data which reduces the verification challenge users face in PBE systems.

Proposed Solution — In this project, we propose a new data transformation paradigm, CLX, to address the two specific problems within our claimed verification issue. The CLX paradigm has three components: two algorithmic components—*clustering* and *transformation*—with an intervening component of *labeling*. In this paper, we present an instantiation of the CLX paradigm. We present (1) an efficient pattern clustering algorithm that groups data with similar structures into small clusters, (2) a DSL for data transformation, that can be interpreted as a set of regular expression replace operations, (3) a program synthesis algorithm to infer desirable transformation logic in the proposed DSL.

Through the above means, we are able to greatly ameliorate the usability issue in verification within PBE data transformation systems. Our experimental results show improvements over the state of the art in saving user verification effort, along with increasing users' comprehension of the inferred transformations. Increasing comprehension is highly relevant to reducing the verification effort. In one user study on a large data set, when the data size grew by a factor of 30, the CLX prototype cost 1.3× more verification time whereas FlashFill cost 11.4× more verification time. In a separate user study accessing the users' understanding of the transformation logic, CLX users achieved a success rate about twice that of FlashFill users. Other experiments also suggest that the expressive power of the CLX prototype and its efficiency on small data are comparable to those of FlashFill.

Organization — After motivating our problem with an example in Section 2, we discuss the following contributions:

- We define the data transformation problem and present the PBE-like CLX framework solving this problem. (Section 3)
- We present a data pattern profiling algorithm to hierarchically cluster the raw data based on patterns. (Section 4)
- We present a new DSL for data pattern transformation in the CLX paradigm. (Section 5)
- We develop algorithms synthesizing data transformation programs, which can transform any given input pattern to the desired standard pattern. (Section 6)
- We experimentally evaluate the CLX prototype and other baseline systems through user studies and simulations. (Section 7)

We explore the related work in Section 8 and finish with a discussion of future work in Section 9.

2 MOTIVATING EXAMPLE

Bob is a technical support employee at the customer service department. He wanted to have a set of 10,000 phone numbers in various formats (as in Figure 1) in a unified format of "(xxx) xxx-xxxx". Given the volume and the heterogeneity of the data, neither manually fixing them or hard-coding a transformation script was convenient for Bob. He decided to see if there was an automated solution to this problem.

(734) 645-8397
(734)586-7252
734-422-8073
734.236.3466
...

Figure 1: Phone numbers with diverse formats

\({digit}3\)\ {digit}3\-{digit}4 **(734) 645-8397** ... *(10000 rows)*

Figure 2: Patterns after transformation

\({digit}3\){digit}3\-{digit}4 **(734)586-7252** ... *(2572 rows)*
{digit}3\-{digit}3\-{digit}4 **734-422-8073** ... *(3749 rows)*
\({digit}3\)\ {digit}3\-{digit}4 **(734) 645-8397** ... *(1436 rows)*
{digit}3\.{digit}3\.{digit}4 **734.236.3466** ... *(631 rows)*
...

Figure 3: Pattern clusters of raw data

```
Replace '/^\(({digit}{3})\)({digit}{3})\-({digit}{4})$/'
    in column1 with '($1) $2-$3'
Replace '/^({digit}{3})\-({digit}{3})\-({digit}{4})$/' in
    column1 with '($1) $2-$3'
...
```

Figure 4: Suggested data transformation operations

Bob found that Excel 2013 had a new feature named FlashFill that could transform data patterns. He loaded the data set into Excel and performed FlashFill on them.

Example 2.1. Initially, Bob thought using FlashFill would be straightforward: he would simply need to provide an example of the transformed form of each ill-formatted data entry in the input and copy the exact value of each data entry already in the correct format. However, in practice, it turned out not to be so easy. First, Bob needed to carefully check each phone number entry deciding whether it is ill-formatted or not. After obtaining a new input-output example pair, FlashFill would update the transformation results for the entire input data, and Bob had to carefully examine again if any of the transformation results were incorrect. This was tedious given the large volume of heterogeneous data **(verification at string level is challenging)**. After rounds of repairing and verifying, Bob was finally sure that FlashFill successfully transformed all existing phone numbers in the data set, and he thought the transformation inferred by FlashFill was impeccable. Yet, when he used it to transform another data set, a phone number "+1 724-285-5210" was mistakenly transformed as "(1) 724-285", which suggested that the transformation logic may fail anytime **(unexplainable transformation logic functions unexpectedly)**. Customer phone numbers were critical information for Bob's company and it was important not to damage them during the transformation. With little insight from FlashFill regarding the transformation program generated, Bob was not sure if the transformation was reliable and had to do more testing **(lack of understanding increases verification effort)**.

Bob heard about CLX and decided to give it a try.

Example 2.2. He loaded his data into CLX and it immediately presented a list of distinct string patterns for phone numbers in the input data (Figure 3), which helped Bob quickly tell which part of the data were ill-formatted. After Bob selected the desired pattern, CLX immediately transformed all the data and showed a new list of string patterns as Figure 2. **So far, verifying the transformation result was straightforward.** The inferred program is presented as a set of Replace operations on raw patterns in Figure 3, each with a picture visualizing the transformation effect. Bob was not a regular expressions guru, but

Notation	Description
$\mathcal{S} = \{s_1, s_2, \ldots\}$	A set of ad hoc strings $s_1, s_2, \ldots$ to be transformed.
$\mathcal{P} = \{p_1, p_2, \ldots\}$	A set of string patterns derived from $\mathcal{S}$.
$p_i = \{t_1, t_2, \ldots\}$	Pattern made from a sequence of tokens t_i
$\mathcal{T}$	The desired target pattern that all strings in $\mathcal{S}$ needed to be transformed into.
$\mathcal{L} = \{(p_1, f_1), (p_2, f_2), \ldots\}$	Program synthesized in CLX transforming data the patterns of $\mathcal{P}$ into $\mathcal{T}$.
$\mathcal{E}$	The expression $\mathcal{E}$ in $\mathcal{L}$, which is a concatenation of Extract and/or ConstStr operations. It is a transformation plan for a source pattern. We also refer to it as an *Atomic Transformation Plan* in the paper.
$Q(\widetilde{\mathrm{t}}, p)$	Frequency of token $\widetilde{\mathrm{t}}$ in pattern p
$\mathcal{G}$	Potential expressions represented in Directed Acyclic Graph.

Table 1: Frequently used notations

Token Class	Regular Expression	Example	Notation
digit	`[0-9]`	"12"	$\langle D \rangle$
lower	`[a-z]`	"car"	$\langle L \rangle$
upper	`[A-Z]`	"IBM"	$\langle U \rangle$
alpha	`[a-zA-Z]`	"Excel"	$\langle A \rangle$
alpha-numeric	`[a-zA-Z0-9_-]`	"Excel2013"	$\langle AN \rangle$

Table 2: Token classes and their descriptions

these operations seemed simple to understand and verify. Like many users in our User Study (Section 7.3), **Bob had a deeper understanding of the inferred transformation logic with** CLX **than with** FlashFill**, and hence, he knew well when and how the program may fail, which saved him from the effort of more blind testing.**

3 OVERVIEW

3.1 Patterns and Data Transformation Problem

A data pattern, or string pattern, is a "high-level" description of the attribute value's string. A natural way to describe a pattern could be a regular expression over the characters that constitute the string. In data transformation, we find that groups of contiguous characters are often transformed together as a group. Further, these groups of characters are meaningful in themselves. For example, in a date string "11/02/2017", it is useful to cluster "2017" into a single group, because these four digits are likely to be manipulated together. We call such meaningful groups of characters as *tokens*.

Table 2 presents all *token classes* we currently support in our instantiation of CLX, including their class names, regular expressions, and notation. In addition, we also support tokens of constant values (e.g., ",", ":"). In the rest of the paper, we represent and handle these tokens of constant values differently from the 5 token classes defined in Table 2. For convenience of presentation, we denote such tokens with constant values as ***literal tokens*** and tokens of 5 token classes defined in Table 2 as ***base tokens***.

A pattern is written as a sequence of tokens, each followed by a quantifier indicating the number of occurrences of the preceding token. A quantifier is either a single natural number or "+", indicating that the token appears at least once. In the rest of the paper, to be succinct, a token will be denoted as "$\langle\widetilde{\mathrm{t}}\rangle\mathfrak{q}$" if $\mathfrak{q}$ is a number (e.g., $\langle D\rangle 3$) or "$\langle\widetilde{\mathrm{t}}\rangle+$" otherwise (e.g., $\langle D\rangle+$). If $\widetilde{\mathrm{t}}$ is a literal token, it will be surrounded by a single quotation mark, like ':'. When a pattern is shown to the end user, it is presented as a *natural-language-like regular expression* proposed by Wrangler [13] (see regexps in Fig 4).

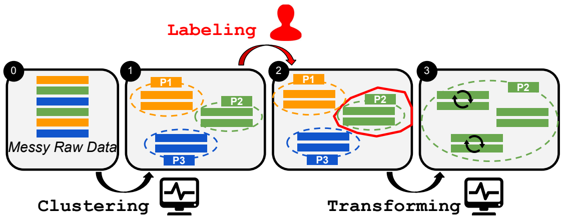


Figure 5: "CLX" Model: Cluster–Label–Transform

With the above definition of data patterns, we hereby formally define the problem we tackle using the CLX framework—data transformation. Data transformation or wrangling is a broad concept. Our focus in this paper is to apply the CLX paradigm to transform a data set of heterogeneous patterns into a desired pattern. A formal definition of the problem is as follows:

Definition 3.1 (Data (Pattern) Transformation). *Given a set of strings $\mathcal{S} = \{s_1, \ldots, s_n\}$, generate a program $\mathcal{L}$ that transforms each string in $\mathcal{S}$ to an equivalent string matching the user-specified desired target pattern $\mathcal{T}$.*

$\mathcal{L} = \{(p_1, f_1), (p_2, f_2), \ldots\}$ is the program we synthesize in the transforming phase of CLX. It is represented as a set regexp replace operations, Replace(p, f)[1], that many people are familiar with (e.g., Fig 4).

With above definitions of patterns and data transformations, we present the CLX framework for data transformation.

3.2 CLX Data Transformation Paradigm

We propose a data transformation paradigm called Cluster-Label-Transform (CLX, pronounced "clicks"). Figure 5 visualizes the interaction model in this framework.

Clustering — The clustering component groups the raw input data into clusters based on their data patterns/formats. Compared to raw strings, data patterns is a more abstract representation. The number of patterns is fewer than raw strings, and hence, it can make the user understand the data and verify the transformation more quickly. Patterns discovered during clustering is also useful information for the downstream program synthesis algorithm to determine the number of regexp replace operations, as well as the desirable input patterns and transformation functions.

Labeling — Labeling is to specify the desired data pattern that every data instance is supposed to be transformed into. Presumably, labeling can be achieved by having the user choose among the set of patterns we derive in the clustering process assuming some of the raw data already exist in the desired format. If no input data matches the target pattern, the user could alternatively choose to manually specify the target data form.

Transforming — After the desired data pattern is labeled, the system automatically synthesizes data transformation logic that transforms all undesired data into the desired form and also proactively helps the user understand the transformation logic.

In this paper, we present an instantiation of the CLX paradigm for *data pattern transformation.* Details about the clustering component and the transformation component are discussed in Section 4 and 6. In Section 5, we show the domain-specific-language (DSL) we use to represent the program $\mathcal{L}$ as the outcome of program synthesis, which can be then presented as the regexp replace operations. The paradigm has been designed to allow new

[1] p is the regular expression, and f is the replacement string indicating the operation on the string matching the pattern p.

algorithms and DSLs for transformation problems other than data pattern transformation; we will pursue other instantiations in future work.

4 CLUSTERING DATA ON PATTERNS

In CLX, we first cluster data into meaningful groups based on their structure and obtain the pattern information, which helps the user quickly understand the data. To minimize user effort, this clustering process should ideally not require user intervention.

LearnPADS [4] is an influential project that also targets string pattern discovery. However, LearnPADS is orthogonal to our effort in that their goal is mainly to find a comprehensive and unified description for the entire data set whereas we seek to partition the data into clusters, each cluster with a single data pattern. Also, the PADS language [3] itself is known to be hard for a non-expert to read [29]. Our interest is to derive simple patterns that are comprehensible. Besides the explainability, efficiency is another important aspect of the clustering algorithm we must consider, because input data can be huge and the real-time clustering must be interactive.

To that end, we propose an automated means to hierarchically cluster data based on data patterns given a set of strings. The data is clustered through a two-phase profiling: (1) tokenization: tokenize the given set of strings of ad hoc data and cluster based on these initial patterns, (2) agglomerative refinement: recursively merge pattern clusters to formulate a ***pattern cluster hierarchy*** that allows the end user to view/understand the pattern structure information in a simpler and more systematic way, and also helps CLX generate a simple transformation program.

4.1 Initial Clustering Through Tokenization

Tokenization is a common process in string processing when string data needs to be manipulated in chunks larger than single characters. A simple parser can do the job.

Below are the rules we follow in the tokenization phase.

- Non-alphanumeric characters carry important hints about the string structure. Each such character is identified as an individual literal token.
- We always choose the most precise base type to describe a token. For example, a token with string content “cat” can be categorized as “lower”, “alphabet” or “alphanumeric” tokens. We choose “lower” as the token type for this token.
- The quantifiers are always natural numbers.

Here is an example of the token description of a string data record discovered in tokenization phase.

Example 4.1. Suppose the string “Bob123@gmail.com” is to be tokenized. The result of tokenization becomes [$\langle U\rangle$, $\langle L\rangle$2, $\langle D\rangle$3, ‘@’, $\langle L\rangle$5, ‘.’, $\langle L\rangle$3].

After tokenization, each string corresponds to a data pattern composed of tokens. We create the initial set of pattern clusters by clustering the strings sharing the same patterns. Each cluster uses its pattern as a label which will later be used for refinement, transformation, and user understanding.

Find Constant Tokens — Some of the tokens in the discovered patterns have constant values. Discovering such constant values and representing them using the actual values rather than base tokens helps improve the quality of the program synthesized. For example, if most entities in a faculty name list contain “Dr.”, it is better to represent a pattern as [‘Dr.’,‘\ ’, ‘$\langle U\rangle$’, ‘$\langle L\rangle$+’] than [‘$\langle U\rangle$’, ‘$\langle L\rangle$’, ‘.’, ‘\ ’, ‘$\langle U\rangle$’, ‘$\langle L\rangle$+’]. Similar to [4], we find tokens with constant values using the statistics over tokenized strings in the data set.

Algorithm 1: Refine Pattern Representations

```
Data: Pattern set $\mathcal{P}$, generalization strategy $\tilde{g}$
Result: Set of more generic patterns $\mathcal{P}_{final}$
1  $\mathcal{P}_{final}, \mathcal{P}_{raw} \leftarrow \emptyset$;
2  $C_{raw} \leftarrow \{\}$;
3  for $p_i \in \mathcal{P}$ do
4      $p_{parent} \leftarrow getParent(p_i, \tilde{g})$;
5      add $p_{parent}$ to $\mathcal{P}_{raw}$;
6      $C_{raw}[p_{parent}] = C_{raw}[p_{parent}] + 1$;
7  for $p_{parent} \in \mathcal{P}_{raw}$ ranked by $C_{raw}$ from high to low do
8      $p_{parent}.child \leftarrow \{p_j | \forall p_j \in \mathcal{P}, p_j.isChild(p_{parent})\}$;
9      add $p_{parent}$ to $\mathcal{P}_{final}$;
10     remove $p_{parent}.child$ from $\mathcal{P}$;
11 Return $\mathcal{P}_{final}$;
```

4.2 Agglomerative Pattern Cluster Refinement

In the initial clustering step, we distinguish different patterns by token classes, token positions, and quantifiers, the actual number of pattern clusters discovered in the ad hoc data in tokenization phase could be huge. User comprehension is inversely related to the number of patterns. It is not very helpful to present too many very specific pattern clusters all at once to the user. Plus, it can be unacceptably expensive to develop data pattern transformation programs separately for each pattern.

To mitigate the problem, we build *pattern cluster hierarchy*, i.e., a hierarchical pattern cluster representation with the leaf nodes being the patterns discovered through tokenization, and every internal node being a *parent pattern*. With this hierarchical pattern description, the user can understand the pattern information at a high level without being overwhelmed by many details, and the system can generate simpler programs. Plus, we do not lose any pattern discovered previously.

From bottom-up, we recursively cluster the patterns at each level to obtain *parent patterns*, i.e., more generic patterns, formulating the new layer in the hierarchy. To build a new layer, Algorithm 1 takes in different generalization strategy $\tilde{g}$ and the child pattern set $\mathcal{P}$ from the last layer. Line 3-5 clusters the current set of pattern clusters to get parent pattern clusters using the generalization strategy $\tilde{g}$. The generated set of parent patterns may be identical to others or might have overlapping expressive power. Keeping all these parent patterns in the same layer of the cluster hierarchy is unnecessary and increases the complexity of the hierarchy generated. Therefore, we only keep a small subset of the parent patterns initially discovered and make sure they together can cover any child pattern in $\mathcal{P}$. To do so, we use a counter C_{raw} counting the frequencies of the obtained parent patterns (line 6). Then, we iteratively add the parent pattern that covers the most patterns in $\mathcal{P}$ into the set of more generic patterns to be returned (line 7-10). The returned set covers all patterns in $\mathcal{P}$ (line 11). Overall, the complexity is $O(n \log n)$, where n is the number of patterns in $\mathcal{P}$, and hence, the algorithm itself can quickly converge.

In this paper, we perform three rounds of refinement to construct the new layer in the hierarchy, each with a particular generalization strategy:

(1) natural number quantifier to ‘+’

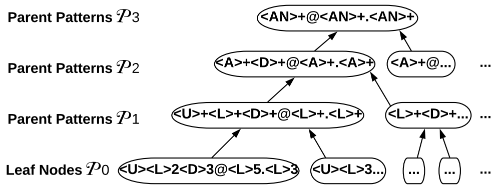


Figure 6: Hierarchical clusters of data patterns

(2) $\langle L \rangle$, $\langle U \rangle$ tokens to $\langle A \rangle$
(3) $\langle A \rangle$, $\langle N \rangle$, ’-’, ‘_’ tokens to $\langle AN \rangle$

Example 4.2. Given the pattern we obtained in Example 4.1, we successively apply Algorithm 1 with Strategy 1, 2 and 3 to generalize parent patterns $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ and construct the pattern cluster hierarchy as in Figure 6.

4.3 Limitations

The pattern hierarchy constructed can succinctly profile the pattern information for many data. However, the technique itself may be weak in two situations. First, as the scope of this paper is limited to addressing the syntactic transformation problem (Section 5), the pattern discovery process we propose only considers syntactic features, but no semantic features. This may introduce the issue of “misclustering”. For example, a date of format “MM/DD/YYYY” and a date of format “DD/MM/YYYY” may be grouped into the same cluster of “$\langle N \rangle 2/\langle N \rangle 2/\langle N \rangle 4$”, and hence, transforming from the former format into the latter format is impossible in our case. Addressing this problem requires the support for semantic information discovery and transformation, which will be in our future work. Another possible weakness of “fail to cluster” is also mainly affected by the semantics issue: we may fail to cluster semantically-same but very messy data. E.g., we may not cluster the local-part (everything before ‘@’) of a very weird email address “Mike'John.Smith@gmail.com” (token $\langle AN \rangle$ cannot capture ‘'’ or ‘.’). Yet, this issue can be easily resolved by adding additional regexp-based token classes (e.g., emails). Adding more token classes is beyond the interest of our work.

5 DATA PATTERN TRANSFORMATION PROGRAM

As motivated in Section 1 and Section 3, our proposed data transformation framework is to synthesize a set of regexp replace operations that people are familiar with as the desired transformation logic. However, representing the logic as regexp strings will make the program synthesis difficult. Instead, to simplify the program synthesis, we propose a new language, UniFi, as a representation of the transformation logic internal to CLX. The grammar of UniFi is shown in Figure 7. We then discuss how to explain a inferred UniFi program as regexp replace operations.

The top-level of any UniFi program is a Switch statement that conditionally maps strings to a transformation. Match checks whether a string s is an *exact match* of a certain pattern p we discover previously. Once a string matches this pattern, it will be processed by an *atomic transformation plan* (expression $\mathcal{E}$ in UniFi) defined below.

Definition 5.1 (Atomic Transformation Plan). *Atomic transformation plan is a sequence of parameterized string operators that converts a given source pattern into the target pattern.*

The available string operators include ConstStr and Extract. $\mathsf{ConstStr}(\widetilde{s})$ denotes a constant string $\widetilde{s}$. $\mathsf{Extract}(\widetilde{\mathfrak{t}_i},\widetilde{\mathfrak{t}_j})$ extracts from the i^{th} token to the j^{th} token in a pattern. In the rest of the paper, we express an Extract operation as Extract(i,j), or Extract(i) if $i = j$. A token $\mathfrak{t}$ is represented as $(\widetilde{\mathfrak{t}}, \mathfrak{r}, \mathfrak{q}, i)$: $\widetilde{\mathfrak{t}}$ is the token class in Table 2; $\mathfrak{r}$ represents the corresponding regular expression of this token; $\mathfrak{q}$ is the quantifier of the token expression; i denotes the index (one-based) of this token in the source pattern.

$$\begin{aligned}
\text{Program } \mathcal{L} &:= \mathsf{Switch}((b_1, \mathcal{E}_1), \ldots, (b_n, \mathcal{E}_n)) \\
\text{Predicate } b &:= \mathsf{Match}(s, p) \\
\text{Expression } \mathcal{E} &:= \mathsf{Concat}(f_1, \ldots, f_n) \\
\text{String Expression } f &:= \mathsf{ConstStr}(\widetilde{s}) \mid \mathsf{Extract}(\widetilde{\mathfrak{t}_i}, \widetilde{\mathfrak{t}_j}) \\
\text{Token Expression } \mathfrak{t}_i &:= (\widetilde{\mathfrak{t}}, \mathfrak{r}, \mathfrak{q}, i)
\end{aligned}$$

Figure 7: UniFi Language Definition

As with FlashFill [6] and BlinkFill [24], we only focus on syntactic transformation, where strings are manipulated as a sequence of characters and no external knowledge is accessible, in this instantiation design. *Semantic transformation* (e.g., converting “March” to “03”) is a subject for future work. Further–again like BlinkFill [24]–our proposed data pattern transformation language UniFi does not support loops. Without the support for loops, UniFi may not be able to describe transformations on an unknown number of occurrences of a given pattern structure.

We use the following two examples used by FlashFill [6] and BlinkFill [24] to briefly demonstrate the expressive power of UniFi, and the more detailed expressive power of UniFi would be examined in the experiments in Section 7.4. For simplicity, Match(s, p) is shortened as Match(p) as the input string s is fixed for a given task.

Example 5.1. This problem is modified from test case “**Example 3**” in BlinkFill. The goal is to transform all messy values in the medical billing codes into the correct form “[CPT-XXXX]” as in Table 3.

Raw data	Transformed data
CPT-00350	[CPT-00350]
[CPT-00340	[CPT-00340]
[CPT-11536]	[CPT-11536]
CPT115	[CPT-115]

Table 3: Normalizing messy medical billing codes

The UniFi program for this standardization task is

```
Switch((Match("\[<U>+\-<D>+"),
     (Concat(Extract(1,4),ConstStr(']')))),
   (Match("<U>+\-<D>+"),
     (Concat(ConstStr('['),Extract(1,3),
       ConstStr(']'))))
   (Match("<U>+<D>+"),
     (Concat(ConstStr('['),Extract(1),
     ConstStr('-'),Extract(2),ConstStr(']')))))
```

Example 5.2. This problem is borrowed from “**Example 9**” in FlashFill. The goal is to transform all names into a unified format as in Table 4.

Raw data	Transformed data
Dr. Eran Yahav	Yahav, E.
Fisher, K.	Fisher, K.
Bill Gates, Sr.	Gates, B.
Oege de Moor	Moor, O.

Table 4: Normalizing messy employee names

A UniFi program for this task is

```
Switch((Match("<U><L>+\.\ <U><L>+\ <U><L>+"),
    Concat(Extract(8,9),ConstStr(','),
        ConstStr(' '),Extract(5))),
    (Match("<U><L>+\ <U><L>+\,\ <U><L>+\."),
     Concat(Extract(4,5),ConstStr(','),
        ConstStr(' '),Extract(1))),
    (Match("<U><L>+\ <U>+\ <U><L>+"),
     Concat(Extract(6,7),ConstStr(','),
        ConstStr(' '),Extract(1))))
```

Program Explanation — Given a UniFi program L, we want to present it as a set of regexp replace operations, Replace, parameterized by *natural-language-like regexps* used by Wrangler [13] (e.g., Figure 4), which are straightforward to even non-expert users. Each component of $(b, \mathcal{E})$, within the Switch statement of L, will be explained as a Replace operation. The replacement string f in the Replace operation is created from p and the transformation plan $\mathcal{E}$ for the condition b. In f, a ConstStr($\tilde{s}$) operation will remain as $\tilde{s}$, whereas a Extract($\tilde{t_i}, \tilde{t_j}$) operation will be interpreted as $\$\tilde{t_i} \dots \$\tilde{t_j}$. The pattern p in the predicate $b = \mathsf{Match}(s, p)$ in UniFi naturally becomes the regular expression p in Replace with each tokens to be extracted surrounded by a pair of parentheses indicating that it can be extracted. Note that if multiple consecutive tokens are extracted in p, we merge them as one component to be extracted in p and change the f accordingly for convenience of presentation. Figure 4 is an example of the transformation logic finally shown to the user.

In fact, these Replace operations can be further explained using visualization techniques. For example, we could add a *Preview Table* (e.g., Figure 8) to visualize the transformation effect in our prototype in a sample of the input data. The user study in Section 7.3 demonstrates that our effort of outputting an explainable transformation program helps the user understand the transformation logic generated by the system.

6 PROGRAM SYNTHESIS

We now discuss how to find the desired transformation logic as a UniFi program using the pattern cluster hierarchy obtained. Algorithm 2 shows our synthesis framework.

Given a pattern hierarchy, we do not need to create an *atomic transformation plan* (Definition 5.1) for every pattern cluster in the hierarchy. We traverse the pattern cluster hierarchy top-down to find valid *candidate source patterns* (line 6, see Section 6.1). Once a source candidate is identified, we discover all *token matches* between this source pattern in Q_{solved} and the target pattern (line 7, see Section 6.2). With the generated token match information, we synthesize the data pattern normalization program including an atomic transformation plan for every source pattern (line 11, see Section 6.3).

6.1 Identify Source Candidates

Before synthesizing a transformation for a source pattern, we want to quickly check whether it can be a *candidate source pattern* (or source candidate), i.e., it is possible to find a transformation from this pattern into the target pattern, through `validate`. **If we can immediately disqualify some patterns, we do not need to go through more expensive data transformation synthesis process for them.** There are a few reasons why some pattern in the hierarchy may not be qualified as a candidate source pattern:

(1) The input data set may be ad hoc and a pattern in this data set can be a description of noise values. For example, a data set of phone numbers may contain "N/A" as a data record because the customer refused to reveal this information. In this case, it is meaningless to generate transformations.

Algorithm 2: Synthesize UniFi Program

```
   Data: Pattern cluster hierarchy root $\mathcal{P}_R$, target pattern $\mathcal{T}$
   Result: Synthesized program $\mathcal{L}$
1  $Q_{unsolved}, Q_{solved} \leftarrow [\,]$;
2  $\mathcal{L} \leftarrow \emptyset$;
3  push $\mathcal{P}_R$ to $Q_{unsolved}$;
4  while $Q_{unsolved} \neq \emptyset$ do
5      $p \leftarrow$ pop $Q_{unsolved}$;
6      if validate($p, \mathcal{T}$) = $\top$ then
7          $\mathcal{G} \leftarrow$ findTokenAlignment($p, \mathcal{T}$);
8          push $\{p, \mathcal{G}\}$ to $Q_{solved}$;
9      else
10         push $p$.children to $Q_{unsolved}$;
11 $\mathcal{L} \leftarrow$ createProgs($Q_{solved}$);
12 Return $\mathcal{L}$
```

(2) We may be fundamentally not able to support some transformations (e.g., semantic transformations are not supported as in our case). Hence, we should filter out certain patterns which we think semantic transformation is unavoidable, because it is impossible to transform them into the desired pattern without the help from the user.

(3) Some patterns are too general; it can be hard to determine how to transform these patterns into the target pattern. We can ignore them and create transformation plans for their children. For instance, if a pattern is "$\langle AN\rangle$+,$\langle AN\rangle$+", it is hard to tell if or how it could be transformed into the desired pattern of "$\langle U\rangle\langle L\rangle$+ : $\langle D\rangle$+". By comparison, its child pattern "$\langle U\rangle\langle L\rangle$+, $\langle D\rangle$+" seems to be a better fit as the candidate source.

Any input data matching no candidate source pattern is left unchanged and flagged for additional review, which could involve replacing values with NULL or default values or manually overriding values.

Since the goal here is simply to quickly prune those patterns that are not good source patterns, the checking process should be able to find unqualified source patterns with *high precision* but not necessarily *high recall*. Here, we use a simple heuristic of *frequency count* that can effectively reject unqualified source patterns with high confidence: examining if there are sufficient base tokens of each class in the source pattern matching the base tokens in the target tokens. The intuition is that any source pattern with fewer base tokens than the target is unlikely to be transformable into the target pattern without external knowledge; base tokens usually carry semantic meanings and hence are likely to be hard to invent *de novo*.

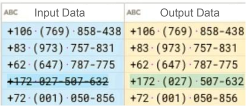
To apply frequency count on the source pattern p_1 and the target pattern p_2, `validate` (denoted as $\mathcal{V}$) compares the *token frequency* for every class of base tokens in p_1 and p_2. The token frequency Q of a token class $\widetilde{\langle t\rangle}$ in p is defined as

$$Q(\widetilde{\langle t\rangle}, p) = \sum_{i=1}^{n} \{t_i.\mathrm{q} \mid t.name = \widetilde{\langle t\rangle}\}, p = \{t_1, \dots, t_n\} \qquad (1)$$

If a quantifier is not a natural number but "+", we treat it as 1 in computing Q.

Suppose $\mathfrak{T}$ is the set of all token classes (in our case, $\mathfrak{T} = [\langle D\rangle, \langle L\rangle, \langle U\rangle, \langle A\rangle, \langle AN\rangle]$), $\mathcal{V}$ is then defined as

$$\mathcal{V}(p_1, p_2) = \begin{cases} \text{true} & \text{if } Q(\widetilde{\langle t\rangle}, p_1) \geq Q(\widetilde{\langle t\rangle}, p_2), \forall \widetilde{\langle t\rangle} \in \mathfrak{T} \\ \text{false} & \text{otherwise} \end{cases} \qquad (2)$$



Input Data	Output Data
+106·(769)·858-438	+106·(769)·858-438
+83·(973)·757-831	+83·(973)·757-831
+62·(647)·787-775	+62·(647)·787-775
~~+172·027-507-632~~	+172·(027)·507-632
+72·(001)·050-856	+72·(001)·050-856

Figure 8: Preview Tab

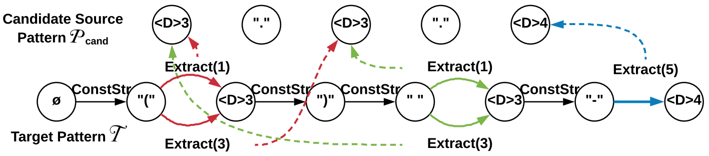


Figure 9: Token alignment for the target pattern $\mathcal{T}$

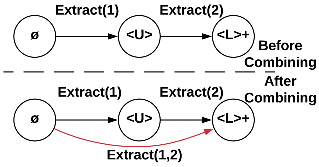


Figure 10: Combine Extracts

```
Algorithm 3: Token Alignment Algorithm
Data: Target pattern $\mathcal{T} = \{t_1, \dots, t_m\}$, candidate source
      pattern $\mathcal{P}_{cand} = \{t'_1, \dots, t'_n\}$, where $t_i$ and $t'_i$ denote
      base tokens
Result: Directed acyclic graph $\mathcal{G}$
1  $\widetilde{\eta} \leftarrow \{0, \dots, n\}$; $\eta^s \leftarrow 0$; $\eta^t \leftarrow n$; $\xi \leftarrow \{\}$;
2  for $t_i \in \mathcal{T}$ do
3      for $t'_j \in \mathcal{P}_{cand}$ do
4          if SyntacticallySimilar($t_i, t'_j$) = ⊤ then
5              $e \leftarrow$ Extract($t'_j$);
6              add $e$ to $\xi_{(i-1,i)}$;
7      if $t_i.type$ = 'literal' then
8          $e \leftarrow$ ConstStr($t_i.name$);
9          add $e$ to $\xi_{(i-1,i)}$;
10 for $i \in \{1, \dots, n-1\}$ do
11     $\xi_{in} \leftarrow \{\forall e_p \in \xi_{(i-1,i)}, e_p$ is an Extract operation$\}$;
12     $\xi_{out} \leftarrow \{\forall e_q \in \xi_{(i,i+1)}, e_q$ is an Extract operation$\}$;
13     for $e_p \in \xi_{in}$ do
14         for $e_q \in \xi_{out}$ do
15             if $e_p.srcIdx + 1 = e_q.srcIdx$ then
16                 $e \leftarrow$ Extract($e_p.t_i, e_q.t_j$);
17                 add $e$ to $\xi_{(i-1,i+1)}$;
18 $\mathcal{G} \leftarrow Dag(\widetilde{\eta}, \eta^s, \eta^t, \xi)$;
19 Return $\mathcal{G}$
```

Example 6.1. Suppose the target pattern $\mathcal{T}$ in Example 5.1 is ['[', $\langle U\rangle$+, '-', $\langle D\rangle$+, ']'], we know

$$Q(\langle D\rangle, \mathcal{T}) = Q(\langle U\rangle, \mathcal{T}) = 1$$

A pattern ['[', $\langle U\rangle$3, '-', $\langle D\rangle$5] derived from data record "[CPT-00350" will be identified as a source candidate by `validate`, because

$$Q(\langle D\rangle, p) = 5 > Q(\langle D\rangle, \mathcal{T}) \wedge$$
$$Q(\langle U\rangle, p) = 3 > Q(\langle U\rangle, \mathcal{T})$$

Another pattern ['[', $\langle U\rangle$3, '-'] derived from data record "[CPT-" will be rejected because

$$Q(\langle D\rangle, p) = 0 < Q(\langle D\rangle, \mathcal{T})$$

6.2 Token Alignment

Once a source pattern is identified as a source candidate in Section 6.1, we need to synthesize an atomic transformation plan between this source pattern and the target pattern, which explains how to obtain the target pattern using the source pattern. To do this, we need to find the token matches for each token in the target pattern: discover all possible operations that yield a token. This process is called *token alignment.*

For each token in the target pattern, there might be multiple different token matches. Inspired by [6], we store the results of the token alignment in Directed Acyclic Graph (DAG) represented as a $DAG(\widetilde{\eta}, \eta^s, \eta^t, \xi)$. $\widetilde{\eta}$ denotes all the nodes in DAG with η^s as the source node and η^t as the target node. Each node corresponds to a position in the pattern. ξ are the edges between the nodes in $\widetilde{\eta}$ storing the source information, which yield the token(s) between the starting node and the ending node of the edge. Our proposed solution to token alignment in a DAG is presented in Algorithm 3.

Align Individual Tokens to Sources — To discover sources, given the target pattern $\mathcal{T}$ and the candidate source pattern $\mathcal{P}_{cand}$, we iterate through each token t_i in $\mathcal{T}$ and compare t_i with all the tokens in $\mathcal{P}_{cand}$.

For any source token t'_j in $\mathcal{P}_{cand}$ that is *syntactically similar* (defined in Definition 6.1) to the target token t_i in $\mathcal{T}$, we create a token match between t'_j and t_i with an Extract operation on an edge from t_{i-1} to t_i (line 2-9).

Definition 6.1 (Syntactically Similar). *Two tokens t_i and t_j are syntactically similar if: 1) they have the same class, 2) their quantifiers are identical natural numbers or one of them is '+' and the other is a natural number.*

When t_i is a literal token, it is either a symbolic character or a constant value. To build such a token, we can simply use a ConstStr operation (line 7-9), instead of extracting it from the source pattern. This does not violate our previous assumption of not introducing any external knowledge during the transformation.

Example 6.2. Let the candidate source pattern be [$\langle D\rangle$3, '.', $\langle D\rangle$3, '.', $\langle D\rangle$4] and the target pattern be ['(', $\langle D\rangle$3, ')', ' ', $\langle D\rangle$3, '-', $\langle D\rangle$4]. Token alignment result for the source pattern $\mathcal{P}_{cand}$ and the target pattern $\mathcal{T}$, generated by Algorithm 3 is shown in Figure 9. In Figure 9, a dashed line is a token match, indicating the token(s) in the source pattern that can formulate a token in the target pattern. A solid line embeds the actual operation in UniFi rendering this token match.

Combine Sequential Extracts — The Extract operator in our proposed language UniFi is designed to extract one or more tokens sequentially from the source pattern. Line 4-9 only discovers sources composed of an Extract operation generating an individual token. *Sequential extracts* (Extract operations extracting multiple consecutive tokens from the source) are not discovered, and this token alignment solution is not complete. We need to find the *sequential extracts.*

Fortunately, discovering sequential extracts is not independent of the previous token alignment process; sequential extracts are combinations of individual extracts. With the alignment results ξ generated previously, we iterate each state and combine every pair of Extracts on an incoming edge and an outgoing edge that extract two consecutive tokens in the source pattern (line 10-17). The Extracts are then added back to ξ. Figure 10 visualizes combining two sequential Extracts. The first half of the figure (titled "Before Combining") shows a transformation plan that generates a target pattern pattern $\langle U\rangle\langle D\rangle$+ with two operations—Extract(1) and Extract(2). The second half of the figure (titled "After Combining") showcases merging the incoming edge and the outgoing edge (representing the previous two operations) and formulate a new operation (red arrow), Extract(1,2), as a combined operation of the two.

A benefit of discovering sequential extracts is it helps yield a "simple" program, as described in Section 6.3.

Correctness — Algorithm 3 is *sound* and *complete*, which is proved in Appendix A in the technical report [12].

6.3 Program Synthesis using Token Alignment Result

As we represent all token matches for a source pattern as a DAG (Algorithm 3), finding a transformation plan is to find a path from the initial state 0 to the final state l, where l is the length of the target pattern $\mathcal{T}$.

The Breadth First Traversal algorithm can find all possible atomic transformation plans for this DAG. However, not all of these plans are equally likely to be correct and desired by the end user. The hope is to prioritize the correct plan. The Occam's razor principle suggests that the simplest explanation is usually correct. Here, we apply **Minimum Description Length** (MDL) [23], a formalization of Occam's razor principle, to gauge the *simplicity* of each possible program.

Suppose $\mathcal{M}$ is the set of models. In this case, it is the set of atomic transformation plans found given the source pattern $\mathcal{P}_{cand}$ and the target pattern $\mathcal{T}$. $\mathcal{E} = f_1 f_2 \ldots f_n \in \mathcal{M}$ is an atomic transformation plan, where f is a string expression. Inspired by [21], we define *Description length* (DL) as follows:

$$L(\mathcal{E}, \mathcal{T}) = L(\mathcal{E}) + L(\mathcal{T}|\mathcal{E}) \tag{3}$$

$L(\mathcal{E})$ is the *model description length*, which is the length required to encode the model, and in this case, $\mathcal{E}$. Hence,

$$L(\mathcal{E}) = |\mathcal{E}| \log m \tag{4}$$

where m is the number of distinct types of operations.

$L(\mathcal{T}|\mathcal{E})$ is the *data description length*, which is the sum of the length required to encode $\mathcal{T}$ using the atomic transformation plan $\mathcal{E}$. Thus,

$$L(\mathcal{T}|\mathcal{E}) = \sum_{f_i \in \mathcal{E}} \log L(f_i) \tag{5}$$

where $L(f_i)$ the length to encode the parameters for a single expression. For a Extract(i) or Extract(i,j) operation, $L(f) = \log |\mathcal{P}_{cand}|^2$ (recall Extract(i) is short for Extract(i,i)). For a ConstStr($\widetilde{s}$), $L(f) = \log c^{|\widetilde{s}|}$, where c is the size of printable character set ($c = 95$).

With the concept of description length described, we define the minimum description length as

$$L_{min}(\mathcal{T}, \mathcal{M}) = \min_{\mathcal{E} \in \mathcal{M}} \Big[L(\mathcal{E}) + L(\mathcal{T}|\mathcal{E}) \Big] \tag{6}$$

In the end, we present the atomic transformation plan $\mathcal{E}$ with the minimum description length as the default transformation plan for the source pattern. Also, we list the other k transformation plans with lowest description lengths.

Example 6.3. Suppose the source pattern is "$\langle D \rangle 2 / \langle D \rangle 2 / \langle D \rangle 4$ ", the target pattern $\mathcal{T}$ is "$\langle D \rangle 2 / \langle D \rangle 2$". The description length of a transformation plan $\mathcal{E}_1$ = Concat(Extract(1,3)) is $L(\mathcal{E}_1, \mathcal{T}) = 1 \log 1 + 2 \log 3$. In comparison, the description length of another transformation plan $\mathcal{E}_2$ = Concat(Extract(1), ConstStr('/'),Extract(3)) is $L(\mathcal{E}_2, \mathcal{T}) = 3 \log 2 + \log 3^2 + \log 95 + \log 3^2 > L(\mathcal{E}_1, \mathcal{T})$. Hence, we prefer $\mathcal{E}_1$, a clearly simpler and better plan than $\mathcal{E}_2$.

6.4 Limitations and Program Repair

The target pattern $\mathcal{T}$ as the sole user input so far is more ambiguous compared to input-output example pairs used in most other PBE systems. Also, we currently do not support "semantic transformation". We may face the issue of "semantic ambiguity"—mismatching syntactically similar tokens with different semantic meanings. For example, if the goals is to transform a date of pattern "DD/MM/YYYY" into the pattern "MM-DD-YYYY" (our clustering algorithm works in this case). Our token alignment algorithm may create a match from "DD" in the first pattern to "MM" in the second pattern because they have the same pattern of $\langle D \rangle 2$. The atomic transformation plan we *initially* select for each source pattern can be a transformation that mistakenly converts "DD/MM/YYYY" into "DD-MM-YYYY". Although our algorithm described in Section 6.3 often makes good guesses about the right matches, the system still infers an imperfect transformation about 50% of the time (Appendix E in the technical report [12]).

Fortunately, as our token alignment algorithm is complete and the program synthesis algorithm can discover all possible transformations and rank them in a smart way, the user can quickly find the correct transformation through *program repair*: replace the initial atomic transformation plan with another atomic transformation plans among the ones Section 6.3 suggests for a given source pattern.

To make the repair even simpler for the user, we deduplicate equivalent atomic transformation plans defined below before the repair phase.

Definition 6.2 (Equivalent Plans). *Two Transformation Plans are* equivalent *if, given the same source pattern, they always yield the same transformation result for any matching string.*

For instance, suppose the source pattern is [$\langle D \rangle 2$, '/', $\langle D \rangle 2$]. Two transformation plans $\mathcal{E}_1$ = [Extract(3), Const('/'), Extract(1)] and $\mathcal{E}_2$ = [Extract(3), Extract(2), Extract(1)] will yield exactly the same output because the first and third operations are identical and the second operation will always generate a '/' in both plans. If two plans are *equivalent*, presenting both rather than one of them will only increase the user effort. Hence, we only pick the simplest plan in the same equivalence class and prune the rest. The methodology detecting the equivalent plans is elaborated in Appendix B in the technical report [12].

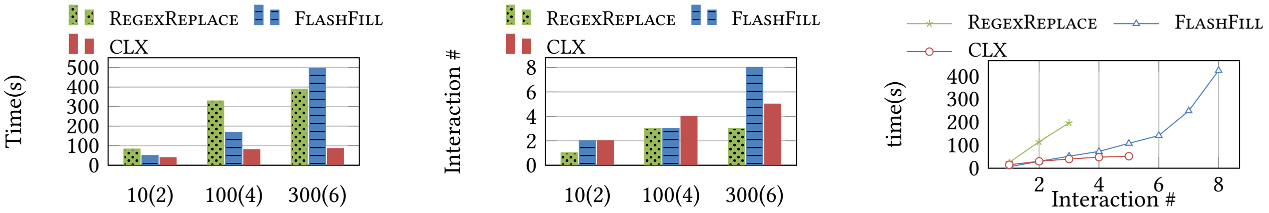
Overall, the repair process does not significantly increase the user effort. In those cases where the initial program is imperfect, 75% of the time the user made just a single repair (Appendix E in the technical report [12]).

7 EXPERIMENTS

We make three broad sets of experimental claims. First, we show that as the input data becomes larger and messier, CLX tends to be less work to use than FlashFill because verification is less challenging (Section 7.2). Second, we show that CLX programs are easier for users to understand than FlashFill programs (Section 7.3). Third, we show that CLX's expressive power is similar to that of baseline systems, as is the required effort for non-verification portions of the PBE process (Section 7.4).

7.1 Experimental Setup

We implemented a prototype of CLX and compared it against the state-of-the-art PBE system FlashFill. For ease of explanation, in this section, we refer this prototype as "CLX". Additionally, to make the experimental study more complete, we



(a) Overall completion time

(b) Rounds of interactions

(c) Interaction timestamps for 300(6)

Figure 11: Scalability of the system usability as data volume and heterogeneity increases (shorter bars are better)

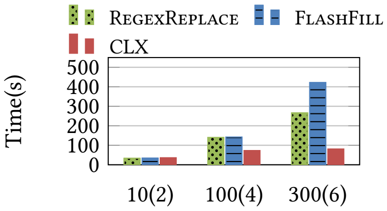


Figure 12: Verification time (shorter bars are better)

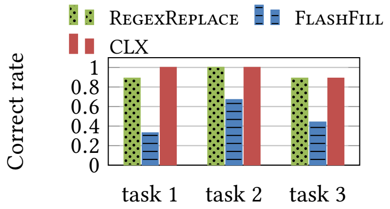


Figure 13: User comprehension test (taller bars are better)

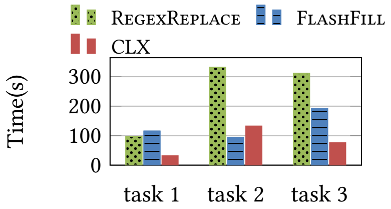


Figure 14: Completion time (shorter bars are better)

had a third baseline approach, a non-PBE feature offered by TrifactaWrangler[2] allowing the user to perform string transformation through manually creating Replace operations with simple natural-language-like regexps (referred as RegexReplace). All experiments were performed on a 4-core Intel Core i7 2.8G CPU with 16GB RAM. Other related PBE systems, Foofah [11] and TDE [9], target different workloads and also share the same verification problem we claim for PBE systems, and hence, are not considered as baselines.

7.2 User Study on Verification Effort

In this section, we conduct a user study on a real-world data set to show that (1) verification is a laborious and time-consuming step for users when using the classic PBE data transformation tool (e.g., FlashFill) particularly on a large messy data set, (2) asking end users to hand-write regexp-based data transformation programs is challenging and inefficient, and (3) the CLX model we propose effectively saves the user effort in verification during data transformation and hence its interaction time does not grow fast as the size and the heterogeneity of the data increase.

Test Data Set — Finding public data sets with messy formats suitable for our experiments is very challenging. The first experiment uses a column of 331 messy phone numbers from the "Times Square Food & Beverage Locations" data set [19].

Overview — The task was to transform all phone numbers into the form "$\langle D\rangle$3-$\langle D\rangle$3-$\langle D\rangle$4". We created three test cases by randomly sampling the data set with the following data sizes and heterogeneity: "10(2)" has 10 data records and 2 patterns; "100(4)" has 100 data records and 4 patterns; "300(6)" has 300 data records and 6 patterns.

We invited 9 students in Computer Science with a basic understanding of regular expressions and not involved in our project. Before the study, we educated all participants on how to use the system. Then, each participant was asked to work on one test case on a system and we recorded their performance.

We looked into the user performances on three systems from various perspectives: *overall completion time*, *number of interactions*, and *verification time*. The *overall completion time* gave us a quick idea of how much the cost of user effort was affected when the input data was increasingly large and heterogeneous in this data transformation task. The other two metrics allowed us to check the user effort in verification. While measuring completion time is straightforward, the other two metrics need to be clarified.

Number of interactions. For FlashFill, the number of interactions is essentially the number of examples the user provides. For CLX we define the number of interactions as the number of times the user verifies (and repairs, if necessary) the inferred atomic transformation plans. We also add one for the initial labeling interaction. For RegexReplace, the number of interactions is the number of Replace operations the user creates.

Verification Time. All three systems follow different interaction paradigms. However, we can divide the interaction process of into two parts, *verification* and *specification*: the user is either busy inputting (typing keyboards, selecting, etc.) or paused to verify the correctness of the transformed data or synthesized/hand-written regular expressions.

Measuring verification time is meaningful because we hypothesize that PBE data transformation systems become harder to use when data is large and messy not because the user has to provide a lot more input, but it becomes harder to verify the transformed data at the instance level.

Results — As shown in Figure 11a, "100(4)" cost 1.1× more time than "10(2)" on CLX, and "300(6)" cost 1.2× more time than "10(2)" on CLX. As for FlashFill, "100(4)" cost 2.4× more time than "10(2)", and "300(6)" cost 9.1× more time than "10(2)". Thus, in this user study, the user effort required by CLX grew slower than that of FlashFill. Also, RegexReplace cost significantly more user effort than CLX but its cost grew not as quickly as FlashFill. This shows good evidence that (1) manually writing data transformation script is cumbersome, (2) the user interaction time grows very fast in FlashFill when data size and heterogeneity increase, and (3) the user interaction time in CLX also grows, but not as fast.

Now, we dive deeper into understanding the causes for observation (2) and (3). Figure 11b shows the number of interactions in all test cases on all systems. We see that all three systems required a similar number of interactions in the first two test cases. Although FlashFill required 3 more interactions than

[2] TrifactaWrangler is a commercial product of Wrangler launched by Trifacta Inc. The version we used is 3.2.1

Task ID	Size	AvgLen	MaxLen	DataType
Task1	10	11.8	14	Human name
Task2	10	20.3	38	Address
Task3	100	16.6	18	Phone number

Table 5: Explainability test cases details

CLX in case “300(6)”, this could hardly be the main reason why FlashFill cost almost 5*x* more time than CLX.

We take a close look at the three systems’ interactions in the case of “300(6)” and plot the timestamps of each interaction in Figure 11c. The result shows that, in FlashFill, as the user was getting close to achieving a perfect transformation, it took the user an increasingly longer amount of time to make an interaction with the system, whereas the interaction time intervals were relatively stable in CLX and RegexReplace. Obviously, the user spent a longer time in each interaction NOT because an example became harder to type in (phone numbers have relatively similar lengths). We observed that, without any help from FlashFill, the user had to eyeball the entire data set to identify the data records that were still not correctly transformed, and it became harder and harder to do so simply because there were fewer of them. Figure 12 presents the average verification time on all systems in each test case. “100(4)” cost 1.0× more verification time than “10(2)” on CLX, and “300(6)” cost 1.3× more verification time than “10(2)” on CLX. As for FlashFill, “100(4)” cost 3.4× more verification time than “10(2)”, and “300(6)” cost 11.4× more verification time than “10(2)”. The fact that the verification time on FlashFill also grew significantly as the data became larger and messier supports our analysis and claim.

To summarize, this user study presents evidence that FlashFill becomes much harder to use as the data becomes larger and messier mainly because verification is more challenging. In contrast, CLX users generally are not affected by this issue.

7.3 User Study on Explainability

Through a new user study with the same 9 participants on three tasks, we demonstrate that (1) FlashFill users lack understanding about the inferred transformation logic, and hence, have inadequate insights on how the logic will work, and show that (2) the simple program generated by CLX improves the user’s understanding of the inferred transformation logic.

Additionally, we also compared the overall completion time of three systems.

Test Set — Since it was impractical to give a user too many data pattern transformation tasks to solve, we had to limit this user study to just a few tasks. To make a fair user study, we chose tasks with various data types that cost relatively same user effort on all three systems. From the benchmark test set we will introduce in Section 7.4, we randomly chose 3 test cases that each is supposed to require same user effort on both CLX and FlashFill: Example 11 from FlashFill (task 1), Example 3 from PredProg (task 2) and “phone-10-long” from SyGus (task 3). Statistics (number of rows, average/max/min string length of the raw data) about the three data sets are shown in Table 5.

Overview — We designed 3 multiple choice questions for every task examining how well the user understood the transformation regardless of the system he/she interacted with. All the questions were formulated as “Given the input string as *x*, what is the expected output”. All questions are shown in Appendix C in the technical report [12].

Sources	# tests	AvgSize	AvgLen	MaxLen	DataType
SyGus [26]	27	63.3	11.8	63	car model ids, human name, phone number, university name and address
FlashFill [6]	10	10.3	15.8	57	log entry, phone number, human name, date, name and position, file directory, url, product name
BlinkFill [24]	4	10.8	14.9	37	city name and country, human name, product id, address
PredProg [25]	3	10.0	12.7	38	human name, address
Prose [22]	3	39.3	10.2	44	country and number, email, human name and affiliation
Overall	47	43.6	13.0	63	

Table 6: Benchmark test cases details

During the user study, we asked every participant to participate all three tasks, each on a different system (completion time was measured). Upon completion, each participant was asked to answer all questions based on the transformation results or the synthetic programs generated by the system.

Explainability Results — The correct rates for all 3 tasks using all systems are presented in Figure 13. The result shows that the participants were able to answer these questions almost perfectly using CLX, but struggled to get even half correct using FlashFill. RegexReplace also achieved a success rate similar to CLX, but required higher user effort and expertise.

The result suggests that FlashFill users have insufficient understanding about the inferred transformation logic and CLX improves the users’ understanding in all tasks, which provides evidence that verification in CLX can be easier.

Overall Completion Time — The average completion time for each task using all three systems is presented in Figure 14. Compared to FlashFill, the participants using CLX spent 30% less time on average: ∼ 70% less time on task 1 and ∼ 60% less time on task 3, but ∼ 40% more time on task 2. Task 1 and task 3 have similar heterogeneity but task 3 (100 records) is bigger than task 1 (10 records). The participants using FlashFill typically spent much more time on understanding the data formats at the beginning and verifying the transformation result in solving task 3. This provides more evidence that CLX saves the verification effort. Task 2 is small (10 data records) but heterogeneous. Both FlashFill and CLX made imperfect transformation logic synthesis, and the participants had to make several corrections or repairs. We believe CLX lost in this case simply because the data set is too small, and as a result, CLX was not able to exploit its advantage in saving user effort on large-scale data set. The study also gives evidence that CLX is sometimes effective in saving user verification effort in small-scale data transformation tasks.

7.4 Expressivity and Efficiency Tests

In a simulation test using a large benchmark test set, we demonstrate that (1) the expressive power of CLX is comparable to the other two baseline systems FlashFill and RegexReplace, and (2) CLX is also pretty efficient in costing user interaction effort.

Test Set — We created a benchmark of 47 data pattern transformation test cases using a mixture of public string transformation test sets and example tasks from related research publications (will be released upon the acceptance of the paper). The information about the number of test cases from each source, average raw input data size (number of rows), average/max data instance length, and data types of these test cases are shown in Table 6.

Baselines	CLX Wins	Tie	CLX Loses
vs. FlashFill	17 (36%)	17 (36%)	13 (28%)
vs. RegexReplace	33 (70%)	12 (26%)	2 (4%)

Table 7: User effort simulation comparison.

A detailed description of the benchmark test set is shown in Appendix D in the technical report [12].

Overview — We evaluated CLX against 47 benchmark tests. As conducting an actual user study on all 47 benchmarks is not feasible, we simulated a user following the "lazy approach" used by Gulwani et al. [8]: a simulated user selected a target pattern or multiple target patterns and then repaired the atomic transformation plan for each source pattern if the system proposed answer was imperfect.

Also, we tested the other two systems against the same benchmark test suite. As with CLX, we simulated a user on FlashFill; this user provided the first positive example on the first data record in a non-standard pattern, and then iteratively provided positive examples for the data record on which the synthetic string transformation program failed. On RegexReplace, the simulated user specified a Replace operation with two regular expressions indicating the matching string pattern and the transformed pattern, and iteratively specified new parameterized Replace operations for the next ill-formatted data record until all data were in the correct format.

Evaluation Metrics — In experiments, we measured how much user effort all three systems required. Because systems follow different interaction models, a direct comparison of the user effort is impossible. We quantify the user effort by *Step*, which is defined differently as follows

- For CLX, the total Steps is the sum of the number of correct patterns the user chooses (Selection) and the number of repairs for the source patterns whose default atomic transformation plans are incorrect (Repair). In the end, we also check if the system has synthesized a "perfect" program: a program that successfully transforms all data.
- For FlashFill, the total Steps is the sum of the number of input examples to provide and the number of data records that the system fails to transform.
- For RegexReplace, each specified Replace operation is counted as 2 Steps as the user needs to type two regular expressions for each Replace, which is about twice the effort of giving an example in FlashFill.

In each test, for any system, if not all data records were correctly transformed, we added the number of data records that the system fails to transform correctly to its total *Step* value as a punishment. In this way, we had a coarse estimation of the user effort in all three systems on the 47 benchmarks.

Expressivity Results — CLX could synthesize right transformations for 42/47 (~ 90%) test cases, whereas FlashFill reached 45/47 (~ 96%). This suggests that the expressive power of CLX is comparable to that of FlashFill.

There were five test cases where CLX failed to yield a perfect transformation. Only one of the failures was due to the expressiveness of the language itself, the others could be fixed if there were more representative examples in the raw data. "Example 13" in FlashFill requires the inference of advanced conditionals (Contains keyword "picture") that UniFi cannot currently express, but adding support for these conditionals in UniFi is straightforward. The failures in the remaining four test cases were mainly caused by the lack of the target pattern examples in the data set. For example, one of the test cases we failed is a name transformation task, where there is a last name "McMillan" to extract. However, all data in the target pattern contained last names comprising one uppercase letter followed by multiple lowercase letters and hence our system did not realize "McMillan" needed to be extracted. We think if the input data is large and representative enough, we should be able to successfully capture all desired data patterns.

RegexReplace allows the user to specify any regular expression replace operations, hence it was able to correctly transform all the input data existed in the test set, because the user could directly write operations replacing the exact string of an individual data record into its desired form. However, similar to UniFi, RegexReplace is also limited by the expressive power of regular expressions and cannot support advanced conditionals. As such, it covered 46/47 (~ 98%) test cases.

User Effort Results — As the *Step* metric is a potentially noisy measure of user effort, it is more reasonable to check whether CLX costs more or less effort than other baselines, rather than to compare absolute *Step* numbers. The aggregated result is shown in Table 7. It suggests CLX often requires less or at least equal user effort than both PBE systems. Compared to RegexReplace, CLX almost always costs less or equal user effort. A detailed discussion about the user effort on CLX and comparison with other systems is in Appendix E in the technical report [12].

8 RELATED WORK

Data Transformation — FlashFill (now a feature in Excel) is an influential work for syntactic transformation by Gulwani [6]. It designed an expressive string transformation language and proposed the algorithm based on version space algebra to discover a program in the designed language. It was recently integrated to PROSE SDK released by Microsoft. A more recent PBE project, TDE [9], also targets string transformation. Similar to FlashFill, TDE requires the user to verify at the instance level and the generated program is unexplainable to the user. Other related PBE data cleaning projects include [11, 24].

Another thread of seminal research including [21], Wrangler [13] and Trifacta created by Hellerstein et al. follow a different interaction paradigm called "predictive interaction". They proposed an inference-enhanced visual platform supporting many different data wrangling and profiling tasks. Based on the user selection of columns, rows or text, the system intelligently suggests possible data transformation operations, such as Split, Fold, or pattern-based extraction operations.

Pattern Profiling — In our project, we focus on clustering ad hoc string data based on structures and derive the structure information. The LearnPADS [4] project is somewhat related. It presents a learning algorithm using statistics over symbols and tokenized data chunks to discover pattern structure. LearnPADS assumes that all data entries follow a repeating high-level pattern structure. However, this assumption may not hold for some of the workload elements. In contrast, we create a bottom-up pattern discovery algorithm that does not make this assumption. Plus, the output of LearnPADS (i.e., PADS program [3]) is hard for a human to read, whereas our pattern cluster hierarchy is simpler to understand. Most recently, Datamaran[5] has proposed methodologies for discovering structure information in a data set whose record boundaries are unknown, but for the same reasons as LearnPADS, Datamaran is not suitable for our problem.

Program Synthesis — Program synthesis has garnered wide interest in domains where the end users might not have good programming skills or programs are hard to maintain or reuse including data science and database systems. Researchers have built various program synthesis applications to generate SQL queries [16, 20, 27], regular expressions [1, 17], data cleaning programs [6, 28], and more.

Researchers have proposed various techniques for program synthesis. [7, 10] proposed a constraint-based program synthesis technique using logic solvers. However, constraint-based techniques are mainly applicable in the context where finding a satisfying solution is challenging, but we prefer a high-quality program rather than a satisfying program. Version space algebra is another important technique that is applied by [6, 14, 15, 18]. [2] recently focuses on using deep learning for program synthesis. Most of these projects rely on user inputs to reduce the search space until a quality program can be discovered; they share the hope that there is one simple solution matching most, if not all, user-provided example pairs. In our case, transformation plans for different heterogeneous patterns can be quite distinct. Thus, applying the version space algebra technique is difficult.

9 CONCLUSION AND FUTURE WORK

Data transformation is a difficult human-intensive task. PBE is a leading approach of using computational inference to reduce human burden in data transformation. However, we observe that standard PBE for data transformation is still difficult to use due to its laborious and unreliable verification process.

We proposed a new data transformation paradigm CLX to alleviate the above issue. In CLX, we build data patterns to help the user quickly identify both well-formatted and ill-formatted data which immediately saves the verification time. CLX also infers regexp replace operations as the desired transformation, which many users are familiar with and boosts their confidence in verification.

We presented an instantiation of CLX with a focus on data pattern transformation including (1) a pattern profiling algorithm that hierarchically clusters both the raw input data and the transformed data based on data patterns, (2) a DSL, UniFi, that can express many data pattern transformation tasks and can be interpreted as a set of simple regular expression replace operations, (3) algorithms inferring a correct UniFi program.

We presented two user studies. In a user study on data sets of various sizes, when the data size grew by a factor of 30, the user verification time required by CLX grew by 1.3× whereas that required by FlashFill grew by 11.4×. The comprehensibility user study shows the CLX users achieved a success rate about twice that of the FlashFill users. The results provide good evidence that CLX greatly alleviates the verification issue.

Although building a highly-expressive data pattern transformation tool is not the central goal of this paper, we are happy to see that the expressive power and user effort efficiency of our initial design of CLX is comparable to those of FlashFill in a simulation study on a large test set in another test.

CLX is a data transformation paradigm that can be used not only for data pattern transformation but other data transformation or transformation tasks too. For example, given a set of heterogeneous spreadsheet tables storing the same information from different organizations, CLX can be used to synthesize programs converting all tables into the same standard format. Building such an instantiation of CLX will be our future work.

10 ACKNOWLEDGMENTS

This project is supported in part by by Trifacta, NSF grants IIS-1250880, IIS-1054913, NSF IGERT grant 0903629, a Sloan Research Fellowship, a CSE Dept. Fellowship, and a University of Michigan MIDAS grant. The authors are grateful to Athena Jiang, Tejas Dharamsi, Joe McKenney, Karthik Sethuraman, Sachin Chawla, and the anonymous reviewers for their helpful feedback.

REFERENCES

[1] A Blackwell. 2001. SWYN: A visual representation for regular expressions. *Your Wish Is My Command: Programming by Example* (2001), 245–270.

[2] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdelrahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy I/O. *arXiv preprint arXiv:1703.07469* (2017).

[3] Kathleen Fisher and Robert Gruber. 2005. PADS: A Domain-specific Language for Processing Ad Hoc Data. In *PLDI*.

[4] Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. 2008. From Dirt to Shovels: Fully Automatic Tool Generation from Ad Hoc Data. In *POPL*.

[5] Yihan Gao, Silu Huang, and Aditya G. Parameswaran. 2018. Navigating the Data Lake with Datamaran: Automatically Extracting Structure from Log Datasets. In *SIGMOD*.

[6] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *POPL*.

[7] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. *PLDI*.

[8] William R Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. In *PLDI*.

[9] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narasayya, and Surajit Chaudhuri. 2018. Transform-Data-by-Example (TDE): An Extensible Search Engine for Data Transformations. In *PVLDB*.

[10] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *ICSE*.

[11] Zhongjun Jin, Michael R. Anderson, Michael Cafarella, and H. V. Jagadish. 2017. Foofah: Transforming Data By Example. In *SIGMOD*.

[12] Zhongjun Jin, Michael Cafarella, H. V. Jagadish, Sean Kandel, Michael Minar, and Joseph M. Hellerstein. 2018. CLX: Towards verifiable PBE data transformation. *arXiv preprint arXiv:1803.00701 [cs.DB]* (2018).

[13] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *CHI*.

[14] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53, 1 (2003), 111–156.

[15] Tessa A Lau, Pedro M Domingos, and Daniel S Weld. 2000. Version Space Algebra and its Application to Programming by Demonstration. In *ICML*.

[16] Fei Li and HV Jagadish. 2014. Constructing an interactive natural language interface for relational databases. In *PVLDB*.

[17] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and HV Jagadish. 2008. Regular expression learning for information extraction. In *EMNLP*.

[18] Tom M Mitchell. 1982. Generalization as search. *Artificial intelligence* 18, 2 (1982), 203–226.

[19] NYC OpenData. 2017. Times Square Food & Beverage Locations" data set. https://opendata.cityofnewyork.us/. (2017).

[20] Li Qian, Michael J Cafarella, and HV Jagadish. 2012. Sample-driven schema mapping. In *SIGMOD*.

[21] Vijayshankar Raman and Joseph M Hellerstein. 2001. Potter's wheel: An interactive data cleaning system. In *VLDB*, Vol. 1. 381–390.

[22] Microsoft Research. 2017. Microsoft Program Synthesis using Examples SDK. https://microsoft.github.io/prose/. (2017).

[23] Jorma Rissanen. 1978. Modeling by shortest data description. *Automatica* 14, 5 (1978), 465–471.

[24] Rishabh Singh. 2016. BlinkFill: Semi-supervised Programming by Example for Syntactic String Transformations. In *PVLDB*.

[25] Rishabh Singh and Sumit Gulwani. 2015. Predicting a correct program in programming by example. In *CAV*.

[26] Syntax-Guided Synthesis. 2017. SyGuS-COMP 2017: The 4th Syntax Guided Synthesis Competition took place as a satellite event of CAV and SYNT 2017. http://www.sygus.org/SyGuS-COMP2017.html. (2017).

[27] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *PLDI*.

[28] Bo Wu and Craig A Knoblock. 2015. An Iterative Approach to Synthesize Data Transformation Programs.. In *IJCAI*.

[29] Kenny Zhu, Kathleen Fisher, and David Walker. 2012. Learnpads++: Incremental inference of ad hoc data formats. *Practical Aspects of Declarative Languages* (2012), 168–182.

Reverse-Engineering Conjunctive Queries from Provenance Examples

Daniel Deutch
Tel Aviv University
danielde@post.tau.ac.il

Amir Gilad
Tel Aviv University
amirgilad@mail.tau.ac.il

ABSTRACT

The *provenance* of a query result details relevant parts of the input data as well as the computation leading to each output tuple. Multiple lines of work have studied the tracking and presentation of provenance, showing its effectiveness in explaining and justifying query results. The willingness of application owners to share provenance information for these purposes may however be hindered by the resulting exposure of the underlying query logic, which may be proprietary and confidential. We therefore formalize and study the following problem: when a (small) subset of the query results along with their provenance is given, what information is revealed about the underlying query? Our model is based on the provenance semiring framework and applies to many previously proposed provenance models. We analyze two flavors of the problem: (1) how many queries may be consistent with a given provenance example? and (2) what is the complexity of inferring a consistent query, or one that is a "best fit"? Our theoretical analysis shows that there may be many (for some models, even infinitely many in presence of self-joins) consistent queries, yet we provide practically efficient algorithms to find (best-fit) such queries. We experimentally show that the algorithms are generally successful in correctly reverse engineering queries, even when given only a few output examples and their provenance.

1 INTRODUCTION

Provenance information is a form of meta-data that is associated with query results, to describe the origin of each piece of data and the computational process leading to its generation. In particular, provenance was shown to be useful as means of providing explanations for query results and allowing for their validation [6, 8, 12, 15, 16, 21, 23, 29, 30, 36]. On the other hand, query owners who publish this information for their users may wish to keep the original query private due to its proprietary logic or due to criteria they wish to keep obfuscated. Thus, owners may wonder if their query can be reverse engineered from a few leaked outputs and their provenance w.r.t this query.

Example 1.1. Consider a bank whose clients must meet certain criteria to have their loan requests approved: they must have a positive credit score and one guarantor associated with the private banking division to co-sign the loan. If a client is already associated with the private banking division, she does not need a guarantor.

The bank's database contains the tables shown in Figures 1a, 1b, 1c, and 1d. The tables correspond to clients and their balance (table *B*), pairs of possible guarantors and borrowers (table *G*), clients with positive credit score (table *PCS*) and clients associated with the private banking division (table *PB*), respectively (ignore the *prov.* column for now).

The following Conjunctive Query (CQ), Q_{real}, captures the bank's loan conditions as specified above:

$$Q_{real}(id_1, b_2) :- B(id_1, b_1), B(id_2, b_2), G(id_2, id_1), PCS(id_1), PB(id_2)$$

It takes the balance of two clients, specifies that the second client is the guarantor of the first, and makes sure that the borrowing client has a positive credit score and that the guarantor is associated with the private banking division. The output is a table of ids and balances where for each borrower id, we show how much security the bank has for the loan, represented as the balance of the guarantor.

The bank may wish to explain to each applicant the reason underlying the answer to its own request, but to avoid exposure of the general criteria, i.e. of the query Q_{real}, since these criteria are part of the bank's confidential business strategy. If other clients obtain some of these explanations, they may understand the general criteria. The question then arises: given examples of the output and explanations provided for multiple clients, can one infer the underlying query?

In the above example, the answers and explanations can be formally defined as output and provenance examples. This leads us to define and study for first time on the modeling and algorithmic challenges stemming from the problem of reverse engineering queries from output and provenance examples.

Modeling. We propose a novel formal model for the problem, as follows. We are given output tuples, and their provenance, represented as instances of the previously proposed semiring model [30]. We have chosen the semiring model due to its generality: as shown in [29], many previously proposed provenance models may be captured via corresponding choices of semirings. Intuitively, different semirings represent different granularities of explanations given to users (see Example 1.2 in this section). We then formally define what it means for a CQ to be consistent with output examples and their provenance, thus defining the *query-by-provenance* problem. Intuitively, we look for a CQ that, when evaluated with respect to the input database *does not only yield the specified example tuples, but also its derivation of these tuples (i.e. their provenance) is "consistent" with the prescription made by the provenance.* We do not expect the full provenance to be provided (i.e. all explanations for a loan being approved); instead, we define "consistency" of provenance in a less restrictive manner, by leveraging the inclusion property of provenance from [29]. Last, we define the notion of inclusion-minimality to capture the idea of the query not only matching the provenance, but rather being a "best-fit" for it.

Theoretical Analysis. We then study in depth the query-by-provenance problem, for five different choices of semirings, namely, provenance polynomials [29, 30], trio provenance [36], positive

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.25

prov.	ID_1	Amount
a	1	1000
b	2	1000
c	3	300

(a) Balance B

prov.	ID_1	ID_2
d	1	2
e	2	2
f	3	1

(b) Guarantor G

prov.	ID_1
h	1
i	2

(c) Positive Credit Score PCS

prov.	ID
j	1
k	2
l	3

(d) Private Banking PB

A	B	$\mathbb{N}[X]$	$\mathbb{B}[X]$	$Trio(X)$	$Why(X)$	$PosBool(X)$
2	1000	$2b^2eik + badij$	$b^2eik + badij$	$2beik + badij$	$beik + badij$	$beik + badij$
1	300	$acfhl$	$acfhl$	$acfhl$	$acfhl$	$acfhl$

(e) Example Data and Provenance Via Different Semirings

Figure 1: Database Tables and Provenance Example

boolean expressions [32], and why-provenance [12]. Each semiring corresponds to an explanation with a different granularity. We next demonstrate two such provenance semirings, namely provenance polynomials ($\mathbb{N}[X]$) and why-provenance (Why(X)), in the context of our running example.

Example 1.2. Reconsider the query Q_{real} and the database in Example 1.1. Tuples in the database are associated with annotations (a to l), intuitively serving as the tuple identifiers (note that the annotations are not part of the relations). Figure 1e presents two output tuples of the query when evaluated on the database. Consider for instance the first row describing the provenance of the tuple $(2, 1000)$ (the id of the borrower with the balance of the guarantor). The $\mathbb{N}[X]$ column shows the polynomial $2b^2eik + badij$, which consists of three monomials (a monomial with coefficient x is considered as x monomials), each corresponding to a different assignment of input tuples to the atoms of Q_{real}, or different explanations for approving the loan request of the client with id 2. For instance, $2b^2eik$ stems from two assignments (implied by the coefficient 2) that use the tuple annotated by b twice and the tuples annotated by e, i, k once. This explanation immediately exposes the number of query atoms (5, as is the number of annotations in the monomial) and the number atoms from each relation. In the context of loan conditions, this explanation exposes that a borrower can be her own guarantor (from the double use of the tuple b). In contrast, in the $Why(X)$ model, we represent each explanation as the **set** of tuples that contributed to the formation of the output tuple, thus "dropping" both coefficients and exponents: $beik + badij$. Considering this kind of explanation, we no longer know the number of query atoms and the number of atoms from each relation. In particular, we no longer expose that a client associated with the private banking division does not need a guarantor.

We study two factors that determine to what extent does a set of output examples and their provenance reveal a hidden underlying query, as follows. The first factor is the number of possible consistent queries, and the second is the complexity of reverse-engineering a consistent query. In more detail, when the provenance is given in $\mathbb{N}[X]$ or $\mathbb{B}[X]$, we provide a bound on the number of consistent queries which is exponential in the sum of arities of all relations whose tuples participate in a single provenance monomial, and provide an example where there are indeed exponentially many consistent queries. We then show that finding a consistent query w.r.t $\mathbb{N}[X]\backslash\mathbb{B}[X]$-examples is NP-complete in the number of attributes of the output tuple, and we design a practically efficient algorithm.

Reconstructing queries from examples of why, trio or PosBool provenance is more cumbersome: there may be infinitely many consistent and inclusion-minimal queries that lead to the same provenance (with different exponents, due to self-joins, that are abstracted away in these models). To this end, we prove a small world property, namely that if a consistent query exists, then there exists such query of size bounded by some parameters of the input. The bound by itself does not suffice for an efficient algorithm (trying all detailed expressions of sizes up to the bound would be inefficient), but we leverage it in devising an efficient algorithm for these provenance semirings. The algorithm is similar to the $\mathbb{N}[X]$ and $\mathbb{B}[X]$ case but also includes an option to expand the monomials, gradually, generating self joins when necessary.

Experimental Analysis. We have also conducted an experimental study of our algorithms, assessing their ability to reverse engineer the correct TPC-H [40] queries as well as highly complex join queries presented as a baseline in [43]. We have executed the queries while tracking provenance, and then showed our algorithms randomly sampled portions of the output and provenance. We report how many examples were required for the algorithms to correctly identify the underlying query, and what were the differences between the actual and inferred query when incorrect. In the vast majority of the cases, our algorithms converged to the underlying query after having viewed only a small number of examples; when this was not the case, the inferred query was typically similar to the actual one, e.g. containing additional constants due to the same value recurring in the viewed examples. Last, further experiments indicate the computational efficiency of our algorithms. Our experiments show that, although theoretically provenance examples may correspond to a large or even infinite number of queries, in practice, publishing provenance information, even for a small number of output tuples, reveals the full logic of the query. Hence, in this respect, there is no advantage in publishing a less detailed form of provenance.

2 RELATED WORK

Reverse Engineering Queries from Partial/Full Output. There is a large body of literature on learning queries from examples, in different variants. A first axis of these variants concerns learning a query whose output *precisely* matches the example (e.g. [33, 41, 43]), versus one whose output contains the example tuples and possibly more (e.g. [33–35, 38] and the somewhat different problem in [44]). The first is mostly useful e.g. in a use-case where an actual query was run and its result, but not the query itself, is available. This may be the case if e.g. the result was exported and sent. The second, that we adopt here, is geared towards examples provided manually by a user, who may not be expected to provide a full account of the output. Another distinguishing factor between works in this area is the domain of queries that are inferred; due to the complexity of the problem, it is typical (e.g. [10, 35, 43]) to restrict the attention to join queries, and many works also impose further restrictions on the join graph [17, 41]. We do not impose such restrictions and are able to infer complex CQs. Last, there is a prominent line of work on query-by-example in the context of *data exploration* [3, 9, 10, 24, 37]. Here users typically provide an initial set of examples, leading to the generation of a consistent query (or multiple such queries); the queries and/or their results are presented to users, who may in turn provide feedback used to refine the queries, and so on. In our settings, the number of examples required for convergence to the actual intended query was typically small. In cases where more examples are needed, an interactive approach is expected to be useful in our setting as well. Works along the lines of [2, 7, 42] explored the problem of reverse engineering queries from positive and negative examples and the general complexity of different variations of the problem.

The fundamental difference between our work and previous work in this field in this area is the assumed input - output examples and provenance information (in particular no foreign keys are known; in fact, we do not even need to know the entire input database, but rather just tuples used in explanations). *While our approach requires more input, it greatly reduces the search space for the query. The advantage becomes more significant as the database size increases and the schema becomes more complex.* Furthermore, we are able to leverage the provenance information and reconstruct the original query, or a very similar one (1) in a highly complex setting where the underlying queries includes multiple joins and self-joins, (2) with only very few examples (up to 5 were typically sufficient to obtain over 90% recall, and less than 20 in all but one case were sufficient to converge to the actual underlying query), and (3) in split-seconds for a small number of examples, and in 1.3 seconds even with 500 examples. No previous work, to our knowledge, has exhibited the combination of these characteristics.

Data Provenance. Data Provenance has been extensively studied, for different formalisms including relational algebra, XML query languages, Nested Relational Calculus, and functional programs (see e.g. [11, 15, 25, 26, 28, 30, 36, 39]). In contrast to these lines of work that focus on provenance tracking and usages, we have focused on learning queries from output examples and their partial provenance, based on the semiring framework. This includes quite a few of the models proposed in the literature, but by no means all of them. Investigating query-by-provenance for other provenance models is an intriguing direction for future work.

The high-level question of what can be learned from provenance has been extensively studied in the context of workflow privacy, e.g. [13, 18–20, 27]. In contrast, we do not focus on black-box modules, but rather on detailed fine-grained provenance obtained from queries. This makes the technical results of these works inapplicable to our setting. [14] describes a general framework for provenance security, but, for the relational database case, focuses on what parts of the *data* are disclosed, while the underlying query is assumed to be known.

Last, we note that in [4] we have leveraged provenance information for designing a user-interactive SPARQL interface, including a component of inferring SPARQL queries from examples and provenance. There are many differences in the model, and consequently none of our results here follow from [4]. In particular, our focus in [4] was on single output nodes which if translated to the relational settings means $k = 1$ (recall that k is the output arity); in that setting it is consequently PTIME to find a consistent query (compare to our NP-hardness result). SPARQL is also bounded to 2 attributes per relation, implying that $r = 2n$, as opposed to the relational setting where the number of attributes is not necessarily 2. Indeed, a prominent challenge in our experimental study stemmed from the number of variables, in particular for the TPC-H queries, where $n << r$.

3 PRELIMINARIES

3.1 Conjunctive Queries

We will focus in this paper on CQs (see e.g. [31]). Fix a database schema $\mathcal{S}$ with relation names $\{\mathcal{R}_1, ..., \mathcal{R}_n\}$ over a domain C of constants. Further fix a domain $\mathcal{V}$ of variables. A *CQ* Q over $\mathcal{S}$ is an expression of the form $T(\vec{u}) : -\mathcal{R}_1(\vec{v_1}), \ldots, \mathcal{R}_l(\vec{v_l})$ where T is a relation name not in $\mathcal{S}$. For all $1 \leq i \leq n$, $\vec{v_i}$ is a vector of the form $(x_1, \ldots, x_k)$ where $\forall 1 \leq j \leq k.\ x_j \in \mathcal{V} \cup C$. $T(\vec{u})$ is the query head, denoted $head(Q)$, and $\mathcal{R}_1(\vec{v_1}), \ldots, \mathcal{R}_l(\vec{v_l})$ is the query body and is denoted $body(Q)$. The variables appearing in $\vec{u}$ are called the *head variables* of Q, and each of them must also appear in the body. We use CQ to denote the class of all CQs, omitting details of the schema when clear from context.

We next define the notion of *derivations* for CQs. A derivation α for a query $Q \in CQ$ with respect to a database instance D is a mapping of the relational atoms of Q to tuples in D that respects relation names and induces a mapping over arguments, i.e. if a relational atom $R(x_1, ..., x_n)$ is mapped to a tuple $R(a_1, ..., a_n)$ then we say that x_i is mapped to a_i (denoted $\alpha(x_i) = a_i$). We require that a variable x_i will not be mapped to multiple distinct values, and a constant x_i will be mapped to itself. We define $\alpha(head(Q))$ as the tuple obtained from $head(Q)$ by replacing each occurrence of a variable x_i by $\alpha(x_i)$.

Example 3.1. Reconsider our example query Q_{real} presented in Example 1.2 (the database is depicted in Figures 1a, 1b, 1c, and 1d). Now, consider the result tuple (1, 300). It is obtained through the derivation that maps the atoms to six distinct tuples from the database to the three atoms (in order of the atoms). These are the tuples annotated by **a, c, f, h, l**. The tuple (2, 1000) is obtained through three derivations: the first (second and third) maps the tuple annotated **b** (**b**) to the first atom, the tuple annotated by **a** (**b**) to the second atom, the tuple annotated by **d** (**e**) to the third atom, the tuple annotated by **i** (**i**) to the fourth, and **j** (**k**) to the remaining atom.

3.2 Provenance

The tracking of *provenance* to explain query results has been extensively studied in multiple lines of work, and [29] has shown that different such models may be captured using the *semiring approach* (originally proposed in [30]). We next overview several aspects of the approach that we will use in our formal framework.

Commutative Semirings. A *commutative monoid* is an algebraic structure $(M, +_M, 0_M)$ where $+_M$ is an associative and commutative binary operation and 0_M is an identity for $+_M$. A *commutative semiring* is then a structure $(K, +_K, \cdot_K, 0_K, 1_K)$ where $(K, +_K, 0_K)$ and $(K, \cdot_K, 1_K)$ are commutative monoids, $\cdot_K$ is distributive over $+_K$, and $a \cdot_K 0_K = 0 \cdot_K a = 0_K$.

Annotated Databases. We will capture provenance through the notion of databases whose tuples are associated ("annotated") with elements of a commutative semiring. For a schema $\mathcal{S}$ with relation names $\{\mathcal{R}_1, ..., \mathcal{R}_n\}$, denote by $Tup(\mathcal{R}_i)$ the set of all (possibly infinitely many) possible tuples of $\mathcal{R}_i$.

Definition 3.2 (adapted from [30]). A *K-relation* for a relation name $\mathcal{R}_i$ and a commutative semiring K is a function R :

$Tup(\mathcal{R}_i) \mapsto K$ such that its *support* defined by $\text{supp}(R) \equiv \{t \mid R(t) \neq 0\}$ is finite. We say that $R(t)$ is the annotation of t in R. A K-database D over a schema $\{\mathcal{R}_1, ..., \mathcal{R}_n\}$ is then a collection of K-relations, over each $\mathcal{R}_i$.

Intuitively a K-relation maps each tuple to its annotation. We will sometimes use $D(t)$ to denote the annotation of t in its relation in a database D. We furthermore say that a K-relation is *abstractly tagged* if each tuple is annotated by a distinct element of K (intuitively, its identifier).

Provenance-Aware Query Results. We then define CQs as mappings from K-databases to K-relations. Intuitively we define the annotation (provenance) of an output tuple as a combination of annotations of input tuples. This combination is based on the query derivations, via the intuitive association of alternative derivations with the semiring "+" operation, and of joint use of tuples in a derivation with the "·" operation.

Definition 3.3 (adapted from [30]). Let D be a K-database and let $Q \in CQ$, with T being the relation name in $head(Q)$. For every tuple $t \in T$, let α_t be the set of derivations of Q w.r.t. D that yield t. $Q(D)$ is defined to be a K-relation T s.t. for every t, $T(t) = \sum_{\alpha \in \alpha_t} \prod_{t' \in Im(\alpha)} D(t')$. $Im(\alpha)$ is the image of α.

Summation and multiplication in the above definition are done in an arbitrary semiring K. Different semirings give different interpretations to the operations [29].

Provenance Polynomials ($\mathbb{N}[X]$). The most general form of provenance for positive relational algebra (see [30]) is the *semiring of polynomials with natural numbers as coefficients*, namely $(\mathbb{N}[X], +, \cdot, 0, 1)$. The idea is that given a set of basic annotations X (elements of which may be assigned to input tuples), the output of a query is represented by a sum of products as in Def. 3.3, with only the basic equivalence laws of commutative semirings in place. Coefficients serve in a sense as "shorthand" for multiple derivations using the same tuples, and exponents as "shorthand" for multiple uses of a tuple in a derivation. In Example 3.1, we see that a tuple may have several derivations, serving as the explanations for it. The two monomials are separated by a + sign, as opposed to a single monomial. We address this as part of our solution presented in Section 5.

Many additional forms of provenance have been proposed in the literature, varying in their level of abstraction and the details they reveal on the derivations. [29] showed that these can be captured by congruence relations. Formally, consider the function $f_K : \mathbb{N}[X] \mapsto \mathbb{N}[X]$ as a congruence relation defined by $P_1 \equiv P_2$ if $f_K(P_1) = f_K(P_2)$, where f_K varies based on the semiring K. We exemplify these notions using our running example of the third row in Figure 1e.

Boolean Provenance Polynomials ($\mathbb{B}[X]$). The boolean provenance semiring, $(\mathbb{B}[X], +, \cdot, 0, 1)$, is the semiring of polynomials over variables X, where coefficients are either 1 or 0 (intuitively corresponding to boolean coefficients). We can think of the $\mathbb{B}[X]$ semiring as the $\mathbb{N}[X]$ semiring after applying the homomorphism $f_{\mathbb{B}}[X] : \mathbb{N}[X] \mapsto \mathbb{N}[X]$ which maps all non-zero coefficients to 1. In the third row of our example, the polynomial $2b^2eik + badij$ becomes $b^2eik + badij$ after mapping the two coefficients 2, 1 to 1.

In the context of our loan example, if we were to obtain the provenance as $\mathbb{N}[X]$ or $\mathbb{B}[X]$, we could infer that a client can get a loan without needing a guarantor.

Trio. This semiring can again be modeled as the image of the surjective homomorphism $f_{trio} : \mathbb{N}[X] \mapsto \mathbb{N}[X]$ dropping all exponents in the provenance polynomial. Consider again the third row of our example. For instance, the polynomial $2b^2eik + badij$ becomes $2beik + badij$ after the function f_{trio} maps all exponents to 1.

Why. A natural approach to provenance tracking, referred to as *why*-provenance [12], capturing each derivation as a *set* of the annotations of tuples used in the derivation. The overall why-provenance is thus a *set of such sets*. As shown (in a slightly different way) in [29], this corresponds to using provenance polynomials but without "caring" about exponents and coefficients. Formally, consider the function $f_{why} : \mathbb{N}[X] \mapsto \mathbb{N}[X]$ that *drops all coefficients and exponents* of its input polynomial. Using our example again, the polynomial $2b^2eik + badij$ is converted to $beik + badij$ under f_{why}.

Positive Boolean Expressions. This is a semiring of positive boolean expressions over variables X, i.e., the expressions are composed of disjunction, conjunction, and constants which are true or false. Formally, we define $f_{PosBool} : \mathbb{N}[X] \mapsto \mathbb{N}[X]$ that dropping all exponents and coefficients in the provenance polynomial, and considers + as $\vee$ and · as $\wedge$, i.e., allowing for the absorption of monomials. For the purpose of the demonstration, consider the abstract polynomial $2b^2eik + baeik$. Under $f_{PosBool}$ it is converted to $beik$ since all exponents and coefficients, and the monomial $baeik$ are absorbed into $beik$ $((b \wedge b \wedge e \wedge i \wedge k) \vee (b \wedge a \wedge e \wedge i \wedge k) \equiv b \wedge e \wedge i \wedge k)$.

In our running example, trio, why and PosBool provenance reveals less information about the loan conditions, e.g., who is the guarantor of the client with id 2 in Figure 1e.

As we demonstrated, each provenance model represents the provenance in a specific level of detail, and all models other than $\mathbb{N}[X]$ incur some loss of information in the provenance description.

4 QUERY-BY-PROVENANCE

We define in this section the problem of learning queries from examples and their provenance. We first introduce the notion of such examples, using provenance.

Definition 4.1 (Examples with provenance). Given a semiring K, a *K-example* is a pair (I, O) where I is an abstractly-tagged K-database called the *input* and O is a K-relation called the *output*.

Intuitively, annotations in the input only serve as identifiers, and those in the output serve as explanations – combinations of annotations of input tuples contributing to the output.

We next define the notion of a query being consistent with a K-example. In the context of query-by-example, a query is consistent if its evaluation result includes all example tuples (but maybe others as well). We resort to [29] for the appropriate generalization to the provenance-aware settings:

Definition 4.2. Let $(K, +_K, \cdot_K, 0, 1)$ be a semiring and define $a \leq_K b$ iff $\exists c.\ a +_K c = b$. If $\leq_K$ is a (partial) order relation then we say that K is naturally ordered.

Given two K-relations R_1, R_2 we say that $R_1 \subseteq_K R_2$ iff $\forall t. R_1(t) \leq_K R_2(t)$.

Note that if $R_1 \subseteq_K R_2$ then in particular $supp(R_1) \subseteq supp(R_2)$, so the notion of containment w.r.t. a semiring is indeed a faithful extension of "standard" relation containment. In terms of provenance, we note that for $\mathbb{N}[X]$ and $Why(X)$, the natural order corresponds to inclusion of monomials: $p_1 \leq p_2$ if every monomial in p_1 appears in p_2. The order relation has different interpretations in other semirings.

We are now ready to define the notion of consistency with respect to a K-example, and introduce our problem statement. Intuitively, we look for a query whose output is contained in the example output, and for each example tuple, the provenance is "reflected" in the computation of the tuple by the query.

Definition 4.3 (Problem Statement). Given a K-example (I,O) and a CQ Q we say that Q is consistent with respect to the example if $O \subseteq_K Q(I)$. QUERY-BY-PROVENANCE is the problem of finding a consistent query for a given K-example.

The above definition allows multiple CQs to be consistent with a given K-example. This is in line with the conventional wisdom in query-by-example. Throughout the paper we will always assume that the provenance is non-empty, since if it is empty, we return to the setting of the classic query-by-example problem.

A consistent query can be very general, as we demonstrate in the following example. A natural desideratum (employed in the context of "query-by-example"), is that the query is "inclusion-minimal". This notion extends naturally to K-databases.

Definition 4.4. A consistent query Q (w.r.t. a given K-example Ex) is inclusion-minimal if for every query Q' such that $Q' \subsetneq_K Q$ (i.e. for every K-database D it holds that $Q'(D) \subseteq_K Q(D)$, but not vice-versa), Q' is not consistent w.r.t. Ex.

We next demonstrate the notion of consistent and inclusion-minimal queries with respect to a given K-example.

Example 4.5. We now treat Figure 1e as an $\mathbb{N}[X]$-example. Consistent queries must derive the example tuples in the ways specified in the polynomials (and possibly in additional ways). The query Q_{real} from Example 1.2 is of course a consistent query with respect to it, since it generates the example tuples and the provenance of each of them according to Q_{real} is the same as that provided in the example. Q_{real} is not the only consistent query, since the following query, $Q_{general}$ (which simply performs a Cartesian product), is also consistent:

$$Q_{general}(id_1, b_2) : -B(id_1, b_1), B(id_2, b_2), G(id_3, id_4), \\ PCS(id_5), PB(id_6)$$

However, Q_{real} is also an inclusion-minimal query, as opposed to $Q_{general}$ since $Q_{real} \subsetneq Q_{general}$.

In the following section we study the complexity of the above computational problems for the different models of provenance. We will analyze the complexity with respect to the different facets of the input, notations for which are provided in Table 1.

Table 1: Notations

Ex	K-example
I	Input database
O	Output relation and its provenance
m	Total number of monomials
k	Number of attributes of the output relation
n	(Maximal) Number of elements in a monomial
r	Sum of arities in the atoms of a query body
d	Number of distinct relation names in the provenance

We list our results for the QUERY-BY-PROVENANCE problem in Table 2. The columns of the table list: the bound for the number of possible consistent queries for each of the semirings, the maximal length of a consistent query if one exists, and the lower and upper bounds of finding a consistent query. Each result has a reference to the relevant proposition. Some of the proofs are omitted for brevity and can be found in the full version [22].

5 LEARNING FROM $\mathbb{N}[X]$, $\mathbb{B}[X]$-EXAMPLES

We start our study with the case where the given provenance consists of $\mathbb{N}[X]$ expressions. This is the most informative form of provenance under the semiring framework, and thus the most informative explanation. In particular, we note that given the $\mathbb{N}[X]$ provenance, the number of query atoms (and the relations occurring in them) are trivially identifiable. What remains is the choice of variables to appear in the query atoms (body and head), and as a consequence, decide how to order the tuples in each monomial.

We first bound the number of possible consistent queries:

Proposition 5.1. *For every choice of $r, k \in \mathbb{N}$, every $\mathbb{N}[X]$-example has at most $O(B_r r^k 2^r)$ consistent queries, where B_r is the Bell number of r. Furthermore, there exists an $\mathbb{N}[X]$-example whose output tuples have arity of k, and monomials have arity of r, for which there exists exponentially many (in both r and k) consistent queries.*

Proof. We start by analyzing the case for boolean queries, i.e., $k = 0$. Consider the set of indexes $\{1, \ldots, r\}$ in the query body. Every partition of this set of indexes defines a different consistent query, since the partition determines exactly which indexes will have the same variable, assuming we impose an order on the query atoms (e.g., the query $Q() : -R(x, x, x), R(x, x, x), T(x, y)$ is determined by the partition $\{1, 2, 3, 4, 5, 6, 7\}, \{8\}$). We can thus say that the maximum number of consistent boolean queries w.r.t an $\mathbb{N}[X]$-example is the number of partitions of the set $\{1, \ldots, r\}$ into disjoint non-empty subsets. This number is the Bell number B_r. Each subset of the partition can then either be instantiated with a constant or with a distinct variable. As there can be at most r subsets in the partition, the number of options is bounded by 2^r. In the general case, when queries are not boolean but have k output attributes, we further have to choose which variables will be projected to the head. Since the maximum number of unique variables in the query body is r, there are at most r^k options to do so. Therefore the number of consistent queries can be bounded by $B_r r^k 2^r$. For the second part of the claim, consider an $\mathbb{N}[X]$-example for which a query with a single constant at every index, both in the head of the query and its body, is consistent (e.g. $Q(1, 1) : -R(1, 1, 1), R(1, 1, 1), T(1, 1)$). In this case, replacing every subset of 1s with different variables, will also result in a consistent query. As there are exponentially many subsets of attributes to the query, there are exponentially many consistent queries w.r.t the $\mathbb{N}[X]$-example. □

We have shown a bound for the number of different consistent queries, but it is also important to note that there can be multiple inclusion-minimal queries. It is easy to show an example where there are may be exponentially many inclusion-minimal queries. We next show an upper bound

Table 2: Results for the QUERY-BY-PROVENANCE Problem

Semiring	Number of consistent queries	Small world query length	Lower bound	Upper bound
$\mathbb{N}[X]$	$O(B_r r^k 2^r)$ (Prop. 5.1)	n (Trivial)	NP-complete in k (Prop. 5.4)	$O(n^{2k}m + krn^k)$ (Prop. 5.2)
$\mathbb{B}[X]$				
$Trio(X)$	∞ (Prop. 6.1)	$k + d(n-1)$ (Prop. 6.5)	Find minimal sized query is NP-complete in k (Prop. 6.4)	$O(n^{O(k)}mkr)$ (Prop. 6.7)
$PosBool(X)$				
$Why(X)$				

5.1 An Efficient Algorithm for Finding a Query

Although the number of query atoms is known from the given example, finding a consistent query efficiently is non-trivial. An important observation in this respect is that we can focus on finding atoms that "cover" the attributes of the output relation (i.e. that include the right values of the output tuples, in the right order), and the number of required such atoms is at most k (the arity of the output relation). We may need further atoms so that the query realizes all provenance tokens (eventually, these atoms will also be useful in imposing e.g. join constraints), and this is where care is needed to avoid an exponential blow-up with respect to the provenance size. To this end, we observe that we may generate a "most general" part of the query simply by generating atoms with fresh variables, and without considering all permutations of parts that do not contribute to the head. This will suffice to guarantee a consistent query, but may lead to the generation of a too general query; this issue will be addressed in Section 5.2.

```
Algorithm 1: FindConsistentQuery (N[X])
      input  : An $\mathbb{N}[X]$ example $Ex = (I, O)$
      output : A consistent query $Q$ or an answer that none
               exists
  1 Let $(t_1, M_1), ..., (t_m, M_m)$ be the tuples and
      corresponding provenance monomials of $O$ ;
  2 $(V, E) \leftarrow BuildLabeledGraph((t_1, M_1), (t_2, M_2))$ ;
  3 foreach consistent matchings $E' \subseteq E$ s.t. $|E'| \leq k$ do
  4     if $\cup_{e \in E'} label(e) = \{1, \ldots, k\}$ then
  5         $Q \leftarrow BuildQueryFromMatch(E', Ex)$ ;
  6         foreach $1 < j < m$ do
  7             if not consistent$(Q, t_j, M_j)$ then
  8                 Go to next matching ;
  9         return $Q$ ;
 10 Output "No consistent query exists";
```

We next detail the construction, shown in Algorithm 1. We separate (line 1) monomials so that each (t_i, M_i) is a tuple along with a single monomial of its provenance expression. We then start by picking two tuples and monomials (see below a heuristic for making such a pick) and denote the tuples by t_1 and t_2 and their provenance by $M_1 = a_1 \cdot \ldots \cdot a_n$ and $M_2 = b_1 \cdot \ldots \cdot b_n$ respectively. Our goal is to find all "matches" of *parts* of the monomials so that all output attributes are covered. To this end, we define (line 2) a full bipartite graph $G = (V_1 \cup V_2, E)$ where each of V_1 and V_2 is a set of n nodes labeled by $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ respectively. We also define labels on each edge, with the label of (a_i, b_j) being the set of all attributes that are *covered* by a_i, b_j, in the following sense: an output attribute A is covered if there is an input attribute A' whose value in the tuple corresponding to a_i (b_j), matches the value of the attribute A in t_1 (respectively t_2).

We then (lines 3-4) find all matchings, *of size k or less*, that cover all output attributes; namely, that the union of sets appearing as labels of the matching's edges equals $\{1, ..., k\}$. As part of the matching, we also specify which subset of the edge label attributes is chosen to be covered by each edge (the number of such options is exponential in k). It is easy to observe that if such a cover (of any size) exists, then there exists a cover of size k or less. We further require that the matching is consistent in the sense that the permutation that it implies is consistent.

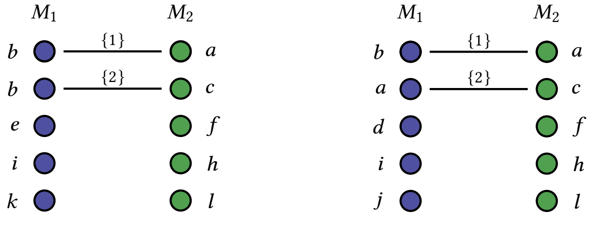
For each such matching we generate (line 5) a "most general query" Q corresponding to it, as follows. We first consider the matched pairs a_i, b_j one by one, and generate a query atom for each pair. This is done by assigning the same variable to the head attribute A covered by the edge and to the attribute covering it in the new atom A'. Note that the query generation is done here based only on k pairs of provenance atoms, rather than all n atoms, since we only examine the k edges of the matching.

To this end, we further generate for each provenance token a_i that was not included in the matching a new query atom with the relation name of the tuple corresponding to a_i, and fresh variables. Intuitively, we impose minimal additional constraints, while covering all head attributes and achieving the required query size of n.

Each such query Q is considered as a "candidate", and its consistency needs to be verified with respect to the other tuples of the example (line 7). One way of doing so is simply by evaluating the query with respect to the input, checking that the output tuples are generated, and their provenance includes those appearing in the example. As a more efficient solution, we test for consistency of Q with respect to each example tuple by first assigning the output tuple values to the head variables, as well as to the occurrences of these variables in the body of Q (by our construction, they can occur in at most k query atoms). For query atoms corresponding to provenance annotations that have not participated in the cover, we only need to check that for each relation name, there is the same number of query atoms and of provenance annotations relating to it. A subtlety here is in handling coefficients; for the part of provenance that has participated in the cover, we can count the number of assignments. This number is multiplied by the number of ways to order the other atoms (which is a simple combinatorial computation), and the result should exceed the provided coefficient.

Proposition 5.2. *Given a $\mathbb{N}[X]$-example, Algorithm 1 finds a consistent query if one exists.*

Choosing the two tuples. For correctness and worst case complexity guarantees, any choice of tuples as a starting point for the algorithm (line 1) would suffice. Naturally, this choice still affects the practical performance, and we aim at minimizing the number of candidate matchings. A simple but effective heuristic is to choose two tuples and monomials for which the number of distinct values (both in the output tuple and in input tuples participating in the derivations) is maximal.



(a) Matching A **(b) Matching B**

Figure 2: Matchings for Example 5.3

Example 5.3. Reconsider our running example depicted in Figure 1e. Assume that the monomials b^2eik and $acfhl$ were picked for graph generation. Each monomial forms a side in the bipartite graph. The algorithm now traverses all partial matchings of size ≤ 2, and checks whether aligning two tuples that are connected by an edge, one below the other, forms a vector of constants that also appears in an attribute of the two corresponding output tuples. Figure 2a depicts a matching of size 2 which consists of the edges $(b, a), (b, c)$ where the first attribute of input tuples **b** and **a** covers the first output attribute (aligning the tuple b below the tuple a creates the constants vector $\binom{2}{1}$ that appear in the first attribute of the output tuple), and the second attribute of input tuples **b** and **c** covers the second output attribute (aligning the tuple b below the tuple c creates the constants vector $\binom{1000}{300}$ that appears in the second attribute of the output tuple).

Generating a query based on this matching results in the query $Q_{general}$ from Example 4.5, since the first and second projected variables have been set to the first and third attributes of the most general atoms $B(id_1, b_1)$ and $B(id_2, b_2)$, respectively and the other atoms remain most general with no joins and projected attributes. The algorithm now verifies the consistency of $Q_{general}$ with respect to the other monomials by assigning the output tuple to the head, e.g. assigning id_1 and b_2 to 2 and 1000 (for the output tuple (2, 1000)), and returns $Q_{general}$. There are other valid partial matchings of these two tuples that will yield a somewhat different query to $Q_{general}$. If we were to choose the monomials $badij$ and $acfhl$, we would have the different matching shown in Figure 2b (where the edges (b, a) and (a, c) cover the head attributes). Note that, in general, a single edge can cover several attributes of the output.

$Q_{general}$ is a very general query, and is probably not the original one. We will adapt our solution so that it finds a more "precise" query, i.e., an inclusion-minimal query (Definition 4.4 in Section 4) in the next subsection.

Complexity. The algorithm's complexity is $O(n^{2k}m + krn^k)$: at most n^k matchings are considered; for each matching, a single query is generated, and consistency is validated in $O(n^k)$ for each of the m example tuples. Furthermore, for each of the matchings, we scan the attributes of the tuples in the matching in $O(kr)$. The exponential factor only involves k, which is much smaller than n and m in practice. Can we do even better? We can show that if $P \neq NP$, there is no algorithm running in time polynomial in k.

Proposition 5.4. *Deciding the existence of a consistent query with respect to a given $\mathbb{N}[X]$-example is NP-complete in k.*

5.2 Achieving a tight fit

To find inclusion-minimal queries, we next refine Algorithm 1 as follows. We do not halt when finding a single consistent query, but instead find all of those queries obtained for some matching. For each consistent query Q, we examine queries obtained from Q by (i) equating variables and (ii) replacing variables by constants where possible (i.e. via an exact containment mapping [1]). We refer to both as *variable equating*. To explore the possible combinations of variable equatings, we use an algorithm inspired by data mining techniques (e.g., [5]): in each iteration, the algorithm starts from a minimal set of variable equatings that was not yet determined to be (in)consistent with the example. E.g., in the first iteration it starts by equating a particular pair of variables. The algorithm then tries, one-by-one, to add variable equatings to the set, while applying transitive closure to ensure the set reflects an equivalence relation. If an additional equating leads to an inconsistent query, it is discarded. Each equatings set obtained in this manner corresponds to a homomorphism h over the variables of the query Q, and we use $h(Q)$ to denote the query resulting from replacing each variable x by $h(x)$.

Importantly, by equating variables or replacing variables by constants we only impose further constraints and obtain no new derivations. In particular, the following result holds, as a simple consequence of Theorem 7.11 in [29] (note that we must keep the number of atoms intact to be consistent with the provenance):

Proposition 5.5. *Let Q be a CQ over a set of variables $\mathcal{V}$. Let $h : \mathcal{V} \mapsto \mathcal{V} \cup C$ be a homomorphism. For every $\mathbb{N}[X]$-example Ex, if Q is not consistent with Ex, then neither is $h(Q)$.*

We can model the homomorphism process between queries, achieved by variable equating, as a lattice whose leaves represent a single variable equating in the query and its top element is the query with a single variable in all atoms. Every variable equating implies a move from the current lattice node to a parent of that node (see next an illustration and example). By this model, the proposition above actually determines that if a query represented by a node in the lattice is consistent, then all of the queries represented by the descendants of this node are also consistent and furthermore, all queries represented by the frontier of the lattice are consistent. We next use these observations to establish an algorithm for finding an inclusion-minimal query.

Consequently, the algorithm finds a *maximal set of variable equatings* that is consistent with the query, by attempting to add at most $O(r^2)$ different equatings, since there are $\binom{r}{2}$ pairs of variables (r is the sum of arities of the atoms in the body of Q, see Table 1). We record every query that was found to be (in)consistent – in particular, every subset of a consistent set of equatings is also consistent – and use it in the following iterations (which again find maximal sets of equatings).

Checking for consistency. This check may be done very efficiently for query atoms that contribute to the head, since we only need to check that equality holds for the provenance annotations assigned to them. For other atoms we no longer have their consistency as a given and in the worst case we would need to examine all matchings of these query atoms to provenance annotations.

Example 5.6. Reconsider our running example query $Q_{general}$ in Example 4.5. A part of the lattice is depicted in Figure 3. The algorithm starts by considering individually each pair of variables as well as pairs of variables and constants co-appearing in the two output tuples or in the tuples used in their provenance. In our example, when considering the lattice element $\{id_1 = id_4\}$, the

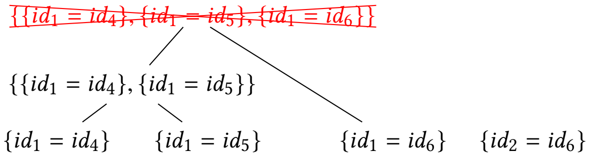

~~{{id1 = id4}, {id1 = id5}, {id1 = id6}}~~
{{id1 = id4}, {id1 = id5}}
{id1 = id4}
{id1 = id5}
{id1 = id6}
{id2 = id6}

Figure 3: Part of the lattice in Example 5.6

algorithm will find that the query $Q_{id_1=id_4}$ (i.e. $Q_{general}$ after equating id_1 and id_4), is still consistent. Next, the algorithm will find that equating id_1, id_5 in $Q_{id_1=id_5}$ also yields a consistent query so it will proceed with $Q_{id_1=id_4,id_1=id_5}$. However, if we try to add the equality $id_1 = id_6$, the consistency check will discover that $Q_{id_1=id_4,id_1=id_5,id_1=id_6}$ is not consistent anymore, so we will not continue on this path of the lattice (marked with a red "x" in Figure 3). Of course, multiple steps may yield the same equivalence classes in which case we perform the computation only once. The resulting query is Q_{real} shown in Example 1.2. Any further step with respect to Q_{real} leads to an inconsistent query, and so it is returned as output.

Proposition 5.7. *Given a consistent query w.r.t an $\mathbb{N}[X]$-example, the procedure finds an inclusion-minimal query.*

For each consistent query found by Algorithm 1, there may be multiple inclusion-minimal queries obtained in such a manner (though the number of such queries observed in practice was not very large, see Section 7). If we wish to provide a single query as output, when multiple inclusion-minimal queries are obtained, a natural heuristic that we employ is to prefer a query with the least number of unique variables (this was implemented in our experimental study).

Adapting the Solution for $\mathbb{B}[X]$. In the $\mathbb{B}[X]$ semiring, exponents are kept but coefficients are not, so we can adapt the algorithm of $\mathbb{N}[X]$ (Algorithm 1), omitting the treatment of coefficients. In $\mathbb{N}[X]$, for the part of the provenance that has participated in the cover, we count the number of assignments and multiply by the number of ways to order the other atoms (which is a simple combinatorial computation), and verify that the result exceeds the provided coefficient. In $\mathbb{B}[X]$, we remove this step of the algorithm.

6 LEARNING FROM WHY, TRIO, AND POSBOOL EXAMPLES

We next study the problem of learning queries from $Why(X)$-examples. This semiring is less detailed than $\mathbb{N}[X]$ and thus often easier to store and present, but is in turn more challenging for query inference. We will adapt the solution to the $Trio(X)$ and $PosBool(X)$ semirings.

A natural approach is to reduce the problem of learning from a $Why(X)$-example to that of learning from an $\mathbb{N}[X]$-example (note that a $Why(X)$-example without self-joins is equivalent to an $\mathbb{N}[X]$-example). Recall that the differences are the lack of coefficients and the lack of exponents. The former is trivial to address (see solution for $\mathbb{B}[X]$ in the previous section), but the latter means that we do not know the number of query atoms. In particular, for $Why(X)$-examples we have:

Proposition 6.1. *There exists a $Why(X)$-example with an infinite number of non-equivalent consistent queries.*

A first plausible idea is to consider the $Why(X)$-example as an $\mathbb{N}[X]$-example, i.e., assume the number of query atoms is equal to the number of tuples in the largest monomial of the example. Surprisingly, we cannot be sure that this suffices:

Proposition 6.2. *There exists a $Why(X)$ example for which there is no consistent CQ with n atoms (recall that n is the length of the largest monomial, see Table 1), but there exists a consistent CQ with more atoms.*

prov.	A	B	C
a	1	2	3
b	3	4	5
c	6	7	8
d	7	6	8

(a) Relation R

A	B	C	prov.
1	1	5	$a \cdot b$
6	7	8	$c \cdot d$

(b) $Why(X)$ Example

Figure 4: Source and $Why(X)$ Example for Prop. 6.2

Proof. Let *Ex* denote the example depicted in Figure 4. A consistent query with two atoms can impose two possible orderings on the tuples in each monomial. We show that both orderings do not form a consistent query. Consider the first ordering of the tuples. A consistent query needs to be satisfied by the assignments:

$$Q(1,1,5) : -R(1,2,3), R(3,4,5)$$
$$Q(6,7,8) : -R(6,7,8), R(7,6,8)$$

But no query can be satisfied by these two assignments because the first assignment requires that the second variable in the head will be bound by the first variable in the first atom in the body. But, the second assignment requires that the second variable in the head will be different from the variable in the mentioned position. Thus, there is no query that is satisfied by both assignments.

Consider the second ordering. A consistent query needs to be satisfied by the assignments:

$$Q(1,1,5) : -R(3,4,5), R(1,2,3)$$
$$Q(6,7,8) : -R(6,7,8), R(7,6,8)$$

Again, observe that in the first assignment, the first variable in the head must be bound by the first variable in the second atom in the body. But, in the second assignment, the first variable in the head is different from the variable in the mentioned position. Thus, again, there is no query that is satisfied by both assignments, establishing that there is no consistent query of length 2 w.r.t *Ex*.

Now, consider the monomials are $a \cdot a \cdot b$ and $c \cdot d \cdot d$. A consistent query w.r.t these monomials needs to satisfy the assignments:

$$Q(1,1,5) : -R(1,2,3), R(1,2,3), R(3,4,5)$$
$$Q(6,7,8) : -R(6,7,8), R(7,6,8), R(7,6,8)$$

Hence, the following query is consistent:

$$Q(x,y,z) : -R(x,w,u), R(y,v,u), R(k,m,z)$$

□

However, finding one consistent query is not difficult, using a method that takes all possible tuple alignments into account:

Proposition 6.3. *If there exists a consistent query w.r.t a given $Why(X)$-example, there is a consistent query of length n^m.*

Proof. Denote the example as *Ex* and the consistent query as Q'. Consider the query Q which has at most n^m atoms and is built in the following manner: have each tuple in *Ex* participate in all possible permutations of tuples in the rows above and below it with the same relation, imposing an order on the tuples in a monomial and then treat the resulting monomials as an $\mathbb{N}[X]$-example, and create a query by replacing each unique constant column with a variable (see previous example). We first show that Q is safe (note that Q' is consistent, so it is in particular safe). Suppose the atoms that have attributes projected to the head in Q' are $a'_1, \ldots, a'_i$. Take the tuples to which they are mapped to in each row j, $t^j_1, \ldots, t^j_i$, then Q also contains atoms $a_1, \ldots, a_i$ mapped to $t^j_1, \ldots, t^j_i$, respectively in each row j (this is because in part of the ordering imposed on the monomials, $t^1_1, \ldots, t^m_1$ appear one below another and the same goes for $t^1_x, \ldots, t^m_x$ for every $2 \leq x \leq i$). Therefore, the same attributes of $a'_1, \ldots, a'_i$ exist in $a_1, \ldots, a_i$ and can also be projected to the head in Q. Furthermore, Q is consistent by the way it was built. Every atom was created for a specific combination of tuples, so every atom of Q can be mapped to a specific tuple appearing in every row. In addition, for every row j, the atoms of Q cover all tuples in this row since an atom was created for each permutation of tuples, and in particular, there is at least one atom that was created to match each tuple in row j. □

Based on this result, we can find a consistent query in a straightforward way. However, a query that has n^m atoms is very long and probably overfits the example. In general, in the $\mathbb{N}[X]$ case, the number of query atoms is set by the provenance and the only degree of freedom is variable equatings. On the other hand, for the $Why(X)$ case, the number of query atoms is also flexible. This calls for a different criterion of "tight fit" for $Why(X)$ provenance, which is minimizing the number of atoms in a consistent query. In general, we can show the following result regarding this criterion:

Proposition 6.4. *Given a $Why(X)$-example, deciding the existence of a consistent query with $\leq n$ atoms is NP-complete in k.*

We can however show a "small world" property, that will guide our solution.

Proposition 6.5. *For any $Why(X)$-example, if there exists a consistent query then there exists a consistent query with $k+d\cdot(n-1)$ atoms or less, where d is the number of distinct relation names occurring in the provenance monomials (see Table 1).*

Intuitively, there are at most k atoms contributing to the head. The worst case is when only one "duplicated" annotation contributes to the head, and then in each provenance monomial there are at most $n-1$ remaining annotations. If the query includes a single relation name ($d = 1$), then a query with at most $n-1$ more atoms would be consistent. Otherwise, as many atoms may be needed for each relation name.

Together with our algorithm for $\mathbb{N}[X]$, Proposition 6.5 dictates a simple algorithm that exhaustively goes through all $\mathbb{N}[X]$ expressions that are compatible with the $Why(X)$ expressions appearing in the example, and whose sizes are up to $k + d\cdot(n-1)$. This, however, would be highly inefficient.

Instead of the inefficient exhaustive algorithm, we next present a much more efficient algorithm for finding a consistent query of minimal length. The algorithm will operate greedily by duplicating atoms in the query body when a self-join is implied by one of the provenance monomials.

The pseudo-code for an efficient algorithm for finding CQs consistent with a given $Why(X)$-example is given in Algorithm 2. The idea is to traverse the examples one by one, trying to "expand" (by adding atoms) candidate queries computed thus far to be consistent with the current example. We start (line 1), as in the $\mathbb{N}[X]$ case, by "splitting" monomials if needed so that each tuple is associated with a single monomial. We maintain a map $\boldsymbol{Q}$ whose values are candidate queries, and keys are the parts of the query that contribute to the head, in a canonical form (e.g. atoms are lexicographically ordered). This will allow us to maintain only a single representative for each such "contributing part", where the representative is consistent with all the examples observed so far. For the first step (line 2) we initialize $\boldsymbol{Q}$ so that it includes only (t_1, M_1) (just for the first iteration, we store an example rather than a query). We then traverse the remaining examples one by one (line 3). In each iteration i, we consider all queries in $\boldsymbol{Q}$; for each such query Q, we build a bipartite graph (line 6) whose one side is the annotations appearing in M_i, and the other side is the *atoms of* Q. The label on each edge is the set of head attributes covered jointly by the two sides: in the first iteration this is exactly as in the $\mathbb{N}[X]$ algorithm, and in subsequent iterations we keep track of covered attributes by each query atom. Then, instead of looking for *matchings* in the bipartite graph, we find (line 7) all *sub-graphs* whose edges cover all head attributes (again specifying a choice of attributes subset for each edge). Intuitively, having e edges adjacent to the same provenance annotation corresponds to the same annotation appearing with exponent e, so we "duplicate" it e times (lines 8-10). On the other hand, if multiple edges are adjacent to a single query atom, we also need to "split" (Lines 11-13) each such atom, i.e. to replace it by multiple atoms (as many as the number of edges connected to it). Intuitively each copy will contribute to the generation of a single annotation in the monomial. Now (line 14), we construct a query Q' based on the matching and the previous query "version" Q: the head is built as in Algorithm 1, and if there were x atoms not contributing to the head with relation name R in Q, then the number of such atoms in Q' is the maximum of x and the number of annotations in M_i of tuples in R that were not matched. Now, we "combine" Q' with Q'' which is the currently stored version of a query with the same contributing atoms (lines 15- 16). Combining means setting number of atoms for each relation name not contributing to the head to be the maximum of this number in Q' and Q''.

Algorithm 2 stores queries according to the atoms that have the same variables as the ones in the head. Each matching that does not stem from the existing matchings in $\boldsymbol{Q}$ (i.e., splitting some of the atoms in an existing cover) will not result in a consistent query since it will not be consistent with the previous rows. The number of combinations of size k can there be out of n different relations is n^k. $\boldsymbol{Q}$ does not store sets of contributing atoms that are subsets of other existing keys.

Complexity. The number of keys in $\boldsymbol{Q}$ is exponential only in k; the loops thus iterate at most $m \cdot n^k \cdot n^k \cdot (n + n^2)$ times, so the overall complexity is $O(n^{O(k)} \cdot mkr)$.

Achieving a tight fit. Algorithm 2 produces a set of candidate queries, which may not be syntactically minimal. To discard atoms that are "clearly" redundant, we first try removing atoms not contributing to the head, and test for consistency. We then perform the process of variable equating as in Section 5.2.

Example 6.6. Reconsider our running example, but now with the why-provenance given in Figure 1e. If we start from the first monomials of the tuples (2, 1000) and (1, 300) then we generate a bipartite graph with $V_1 = \{b, e, i, k\}$ and $V_2 = \{a, c, f, h, l\}$, and obtain the cover E' (seen in Figure 5a) where the edge (b, a)

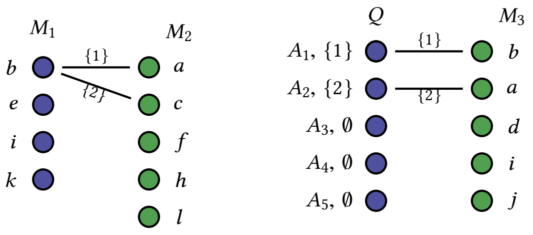
Algorithm 2: FindConsistentQuery (Why(X))

```
input : A $Why(X)$ example Ex
output: A set of consistent queries (possibly empty, if none exists)
Let $(t_1, M_1), ..., (t_m, M_m)$ be the tuples and corresponding provenance monomials of $Ex$ ;
$\mathcal{Q} \leftarrow \{NULL : (t_1, M_1)\}$ ;
foreach $2 \le i \le m$ do
    foreach $Q \in$ values($\mathcal{Q}$) do
        $\mathcal{Q} \leftarrow \mathcal{Q} - \{Q\}$ ;
        $(V_1 \cup V_2, E) \leftarrow BuildGraph(Q, (t_i, M_i))$ ;
        foreach sub-graph $E' \subseteq E$ s.t. $|E'| \le k$ and $\cup_{e \in E'}$ label$(e) = \{1, \dots, k\}$ do
            foreach provenance annotation $a$ in $M_i$ do
                if $a$ is an endpoint of more than one edge in $E'$ then
                    $E' \leftarrow split(E', a)$ ;
            foreach atom $C \in Q$ do
                if $C$ is an endpoint of more than one edge in $E'$ then
                    $E' \leftarrow split(E', C)$ ;
            $Q' \leftarrow BuildQuery(E', Q)$;
            $Q'' \leftarrow \mathcal{Q}.get(contribs(Q'))$ ;
            $\mathcal{Q}.put(contribs(Q'), combine(Q', Q''))$ ;
return $values(\mathcal{Q})$ ;
```

covers the first head attribute and (b, c) covers the second. The fact that the tuple **b** is connected to two edges will lead to the split of this tuple to the atoms $B(id_1, b_1)$ and $B(id_2, b_2)$. When we continue with E', no duplication is performed, and we get a query Q with the two atoms $B(id_1, b_1)$, $B(id_2, b_2)$ contributing to the head, and three most general atoms. Then, we match Q to the monomial $badij$, resulting in a sub-graph matching both $B(id_1, b_1)$ and $B(id_2, b_2)$ to **b** and **a**, respectively. After variables equating (see Subsection 5.2), we obtain the query Q_{real} shown in Example 1.2 (other covers are possible, but they will also result in Q_{real} after variable equating). If instead we start with the monomials $badij$ and $acfhl$, the partial matching chosen will be the edge (b, a) which covers the first head attribute and the edge (a, c) which covers the second and forming the atoms $B(id_1, b_1)$, $B(id_2, b_2)$ again. Continuing to the monomial $beik$, we would split the tuple b into two tuples since the matching will include the edges $(B(id_1, b_1), b)$ that covers the first head attribute and $(B(id_2, b_2), b)$ which covers the second.

Proposition 6.7. *Given a Why(X)-example, Algorithm 2 finds a consistent query if one exists.*

Adapting the Solution for $Trio(X)$ and $PosBool(X)$. In the $Trio(X)$ semiring coefficients are kept but exponents are not. To handle this case we employ Algorithm 2, with a simple modification: upon checking consistency of a candidate query with a tuple, then we further check that there are as many derivations that use the tuples of the monomial as dictated by the coefficient (as done in Section 5). In the semiring of positive boolean expressions ($PosBool(X)$) + and · are interpreted as disjunction and conjunction, respectively. If the expressions are given in DNF, Algorithm 2 may be used here as well. The only difference is the possible absorption of monomials ($a + a \cdot b \equiv a$), but we already assume that only a subset of the monomials are given. If the expressions are given in an arbitrary form there is an additional (exponential time) pre-processing step of transforming them into DNF.



(a) 1st iteration in 6.6 **(b) 2nd iteration in 6.6**

Figure 5: Subgraphs for Example 6.6

7 IMPLEMENTATION AND EXPERIMENTS

Our experimental study is composed of experiments based on the actual output and provenance of the benchmark queries in [40, 43] measuring both accuracy and scalability. All experiments were performed on Windows 8, 64-bit, with 8GB of RAM and Intel Core Duo i7 2.59 GHz processor. We have implemented our algorithms in JAVA with MS SQL server as the underlying database management system.

As we have shown in Sections 5 and 6, the algorithms for the $\mathbb{N}[X]$ and $Why(X)$ semirings also capture the other semirings we have studied, with slight modifications. Namely, the algorithms for $\mathbb{N}[X]$ also apply for the $\mathbb{B}[X]$ semiring, ignoring the treatment of coefficients, and the algorithms presented for $Why(X)$ also capture the $PosBool(X)$ semiring (since there is only an absorption of monomials), and the $Trio(X)$ semiring with the handling of coefficients as done for $\mathbb{N}[X]$ in Algorithm 1. Hence, our experiments focus on the $\mathbb{N}[X]$ and $Why(X)$ models. In particular, Algorithm 1 will behave in the same manner for $\mathbb{B}[X]$-examples as for $\mathbb{N}[X]$-examples, and Algorithm 2 will behave in the same manner for $PosBool(X) \backslash Trio(X)$-examples as for $Why(X)$-examples.

To examine our approach, we have used multiple queries with varying complexity. Namely, Q1–Q6 from [43] as well as (modified, to drop aggregation and arithmetics) the TPC-H queries TQ2–TQ5, TQ8 and TQ10. The queries have 2–8 atoms, 18–60 variables, and multiple instances of self-joins (we show Q6 for illustration in Figure 6; the reader is referred to [40, 43] for the other queries).

7.1 Accuracy

We have used the system to "reverse engineer" the queries mentioned above. This part of the experiments had two objectives: (1) understanding the number of examples needed to infer the exact query (2) measure the precision and recall of the query inferred by the system with comparison to the original. We have evaluated each query using a provenance-aware query engine, and have then sampled random fragments (of a given size that we vary) of the output database and its provenance (we have tried both $\mathbb{N}[X]$ and Why(X)), feeding it to our system. In each experiment we have gradually added random examples until our algorithm has retrieved the original query. This was repeated 3 times. We report (1) the *worst-case* (as observed in the 3 executions) number of examples needed until the original query is inferred, and (2) for fewer examples (i.e. before convergence to the actual query),

Table 3: Results for the TPC-H query set and the queries from [43] with $\mathbb{N}[X]$ provenance

Query	Worst-case number of examples to learn the original query	Difference between original and inferred queries for fewer examples
Q1 (TQ3)	14	The inferred Query includes an extra join on a “status" attribute of two relations. Only 2–3 values are possible for this attribute, and equality often holds.
Q2	2	
Q3	5	For 2 examples, the inferred query contained an extra constant. For 3 and 4 examples, it included an extra join.
Q4	19	For 2 examples, the inferred query included an extra constant. For 3–18, it included an extra join on a highly skewed “status" attribute.
Q5	11	The inferred query included an extra join on a “name" attribute.
Q6	3	The inferred query included an extra constant.
TQ4	234	The inferred query included an extra join on “orderstatus" and “linestatus" attributes of two relations (they have two possible values). One of the original join conditions has led to the occurrence of the same value in these attributes in the vast majority of joined tuples.
TQ10	4	The inferred query contained an extra constant.
TQ2	3	The inferred query contained an extra constant.
TQ5	3	The inferred query contained an extra constant.
TQ8	18	For 2 examples, the inferred query contained an extra constant. For 4-17 exam– –ples, the query had an extra join between a “status" attribute of two relations.

$ans(a, b) : -supplier(c, a, add, k, p, d, c_1),$
$partsupp(h, c, v, j, c_2), nation(w, na_2, r, c_8),$
$part(h, i, z, q, t, s, e, rp_2, c_3), region(r, u, c_7),$
$partsupp(h, o, x, n, c_4), nation(k, na1, r, c_6),$
$supplier(o, b, y, w, p_2, d_2, c_5)$

Figure 6: $Q6$

the differences between the inferred queries and the actual one in the worst-case run of the experiment.

The results are reported in Table 3. For some queries the convergence is immediate, and achieved when viewing only 2–5 examples. For other queries, more examples are needed, but with one exception (TQ4), we converge to the original query after viewing at most 19 tuples for the different queries. For TQ4 only a very small fraction of the output tuples reveal that an extra join should not have appeared, and so we need one of these tuples to appear in the sample. Furthermore, even for smaller sets of examples, the inferred query was not “far" from the actual query. The most commonly observed difference involved extra constants occurring in the inferred query (this typically happened for a small number of examples, where a constant co-occurred by chance). Another type of error was an extra join in the inferred query; this happened often when two relations involved in the query had a binary or trinary attribute (such as the “status" attribute occurring in multiple variants in TPC-H relations), which is furthermore skewed (for instance, when other join conditions almost always imply equality of the relevant attributes). We have also measured the precision and recall of the output of the inferred query w.r.t. that of the original one. Obviously, when the original query was obtained, the precision and recall were 100%. Even when presented with fewer examples, in almost all cases already with 5 examples, the precision was 100% and the recall was above 90%. The only exception was Q5 with 75% recall for 5 examples.

The results for $Why(X)$ are shown in Table 4. For queries with no self-join, the observed results were naturally the same as in the $\mathbb{N}[X]$ case; we thus report the results only for queries with self-joins (some queries included multiple self joins). When presented with a very small number of examples, our algorithm was not always able to detect the self-joins (see comments in Table 7); but the overall number of examples required for convergence has only marginally increased with respect to the $\mathbb{N}[X]$ case.

7.2 Execution Times

As we have established in the accuracy experiments, a small number of examples will usually suffice to infer the original query. To also account for the execution time of our algorithms, we have increased the number of examples up to 500, which is about twice as many examples needed to infer each of the queries. The results for $\mathbb{N}[X]$ provenance and Q1–Q6 exhibit good scalability: the computation time for 500 examples was 0.4 seconds for Q1 (TQ3), 0.1 seconds for Q2, 0.7 seconds for Q3, 1 second for Q4 and 0.4 and 1.1 seconds for Q5 and Q6 respectively. The performance for the TPC-H queries was similarly scalable: for 500 examples, the computation time of TQ2 and TQ10 (which are the queries with the maximum number of head attributes: 8 and 7, resp.) was 0.2 and 0.4 seconds respectively. The runtimes for TQ4 and TQ8 were 0.1, 0.9 seconds resp. The number of example for TQ5 was limited due to the query output size. For 15 examples, the runtime for this query was 0.2 seconds. We have repeated the experiment using $Why(X)$ provenance. The computation time was generally fast, and only slightly slower than the $\mathbb{N}[X]$ case, with a max runtime of 1.3 seconds for Q4 and TQ8. This is consistent with our complexity analysis.

Effect of Tuples Choice. Recall that Algorithm 1 starts by finding consistent queries w.r.t two example tuples and explanations. In Section 5, we have described a heuristic that chooses the two tuples with the least number of shared values. We have measured the effect of this optimization on Q6 and found that using the optimization leads to a single matching in the graph, as oppose to a random choice of tuples that has led to 4 matchings. Making such a random choice, instead of using our optimization led to a runtime which was more than 1.3 times slower on average.

8 DISCUSSION AND CONCLUSIONS

We have formalized and studied the problem of “query-by-provenance", where queries are inferred from example output tuples and their

Table 4: Results for the TPC-H query set and the queries from [43] containing self-joins with $Why(X)$ provenance

Query	Worst-case number of examples to learn the original query	Difference between original and inferred queries for fewer examples
Q2	2	
Q3	5	The inferred query for 2–4 examples did not include self-joins.
Q4	19	For 2–3 examples, the inferred query did not include self-joins. For 4–18 examples, the query had an extra join on a "status" attribute.
Q5	13	The inferred query for 2–12 examples did not include self-joins.
Q6	3	The inferred query included an extra constant.
TQ8	18	For 2–3 examples, the inferred query contained an extra constant. For 4-17 the query had an extra join between a "status" attribute of two relations.

provenance. We have theoretically analyzed and experimentally demonstrated the effectiveness of the approach in inferring highly complex CQs, including ones with multiple self-joins, based on a small number of output and provenance examples and their provenance. Note that for UCQs, a stronger definition is needed.

A notable provenance model that we have not discussed so far is that of lineage [16], which entails a set representation of the provenance, i.e., the contributing input tuples' annotations for a given output tuple are represented as a set. The semiring model can support lineage, and the interpretation of Definition 4.2 in this case is simply that each output example is associated with a subset of its contributing tuples. In general, this means that without looking at the full input database or at least its schema, we gain very little information (e.g. we are not guaranteed that the given annotations come from a relation that is projected to the head). In particular it is straightforward to show a reduction from the classic reverse engineering query problem, which is intractable by [42].

Acknowledgments. This research has been funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 804302). The contribution of Amir Gilad is part of a Ph.D. thesis research conducted at Tel Aviv University.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

[2] Azza Abouzied, Dana Angluin, Christos H. Papadimitriou, Joseph M. Hellerstein, and Avi Silberschatz. 2013. Learning and verifying quantified boolean queries by example. In *PODS.* 49–60.

[3] Azza Abouzied, Joseph M. Hellerstein, and Avi Silberschatz. 2012. Playful Query Specification with DataPlay. *PVLDB* 5, 12 (2012), 1938–1941.

[4] Efrat Abramovitz, Daniel Deutch, and Amir Gilad. 2018. Interactive Inference of SPARQL Queries Using Provenance. In *ICDE.* 581–592.

[5] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. 1998. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. In *SIGMOD.* 94–105.

[6] Tarun Arora et al. 1993. Explaining Program Execution in Deductive Systems. In *DOOD.* 101–119.

[7] Pablo Barceló and Miguel Romero. 2017. The Complexity of Reverse Engineering Problems for Conjunctive Queries. In *ICDT.* 7:1–7:17.

[8] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. 2014. Query-Based Why-Not Provenance with NedExplain. In *EDBT.* 145–156.

[9] Angela Bonifati, Radu Ciucanu, Aurélien Lemay, and Slawek Staworko. 2014. A Paradigm for Learning Queries on Big Data. In *Data4U@VLDB.*

[10] Angela Bonifati, Radu Ciucanu, and Slawek Staworko. 2014. Interactive Join Query Inference with JIM. *PVLDB* 7, 13 (2014), 1541–1544.

[11] P. Buneman, J. Cheney, and S. Vansummeren. 2008. On the expressiveness of implicit provenance in query and update languages. *ACM Trans. Database Syst.* (2008), 28:1–28:47.

[12] P. Buneman, S. Khanna, and W.C. Tan. 2001. Why and Where: A Characterization of Data Provenance. In *ICDT.* 316–330.

[13] Artem Chebotko, Seunghan Chang, Shiyong Lu, Farshad Fotouhi, and Ping Yang. 2008. Scientific Workflow Provenance Querying with Security Views. In *WAIM.* 349–356.

[14] James Cheney. 2011. A Formal Framework for Provenance Security. In *CSF.* 281–293.

[15] J. Cheney, L. Chiticariu, and W. C. Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* (2009), 379–474.

[16] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Trans. Database Syst.* (2000), 179–227.

[17] Anish Das Sarma, Aditya Parameswaran, Hector Garcia-Molina, and Jennifer Widom. 2010. Synthesizing View Definitions from Data *(ICDT).* 89–103.

[18] Susan B. Davidson, Sanjeev Khanna, Tova Milo, Debmalya Panigrahi, and Sudeepa Roy. 2011. Provenance views for module privacy. In *PODS.* 175–186.

[19] Susan B. Davidson, Sanjeev Khanna, Sudeepa Roy, Julia Stoyanovich, Val Tannen, and Yi Chen. 2011. On provenance and privacy. In *ICDT.* 3–10.

[20] Susan B. Davidson, Sanjeev Khanna, Val Tannen, Sudeepa Roy, Yi Chen, Tova Milo, and Julia Stoyanovich. 2011. Enabling Privacy in Provenance-Aware Workflow Systems. In *CIDR.* 215–218.

[21] Daniel Deutch, Nave Frost, and Amir Gilad. 2017. Provenance for Natural Language Queries. *PVLDB* 10, 5 (2017), 577–588.

[22] D. Deutch and A. Gilad. 2018. Full Version. http://www.cs.tau.ac.il/~amirgilad/full/RevEngFull.pdf. (2018).

[23] Daniel Deutch, Amir Gilad, and Yuval Moskovitch. 2015. Selective Provenance for Datalog Programs Using Top-K Queries. *PVLDB* 8, 12 (2015), 1394–1405.

[24] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2014. Explore-by-example: An Automatic Query Steering Framework for Interactive Data Exploration. In *SIGMOD.* 517–528.

[25] R. Fink, L. Han, and D. Olteanu. 2012. Aggregation in Probabilistic Databases via Knowledge Compilation. *PVLDB* 5, 5 (2012), 490–501.

[26] F. Geerts and A. Poggi. 2010. On database query languages for K-relations. *J. Applied Logic* (2010), 173–185.

[27] Yolanda Gil and Christian Fritz. 2010. Reasoning about the Appropriate Use of Private Data through Computational Workflows. In *AAAI.*

[28] B. Glavic, J. Siddique, P. Andritsos, and R. J. Miller. 2013. Provenance for Data Mining. In *TaPP.*

[29] T. J. Green. 2009. Containment of conjunctive queries on annotated relations. In *ICDT.* 296–309.

[30] T. J. Green, G. Karvounarakis, and V. Tannen. 2007. Provenance semirings. In *PODS.* 31–40.

[31] Garcia-Molina Hector, Jeffrey D Ullman, and Jennifer Widom. 2002. *Database systems: The complete book.* Prentice-Hall.

[32] Tomasz Imieliński and Witold Lipski, Jr. 1984. Incomplete Information in Relational Databases. *J. ACM* (1984), 761–791.

[33] Dmitri V. Kalashnikov, Laks V.S. Lakshmanan, and Divesh Srivastava. 2018. FastQRE: Fast Query Reverse Engineering. In *SIGMOD.* 337–350.

[34] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. 2014. Exemplar Queries: Give Me an Example of What You Need. *PVLDB* 7, 5, 365–376.

[35] Fotis Psallidas, Bolin Ding, Kaushik Chakrabarti, and Surajit Chaudhuri. 2015. S4: Top-k Spreadsheet-Style Search for Query Discovery *(SIGMOD).* 2001–2016.

[36] Anish Das Sarma, Martin Theobald, and Jennifer Widom. 2008. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. In *ICDE.* 1023–1032.

[37] Thibault Sellam and Martin L. Kersten. 2013. Meet Charles, big data query advisor *(CIDR).*

[38] Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. 2014. Discovering Queries Based on Example Tuples. In *SIGMOD.* 493–504.

[39] Wang Chiew Tan. 2003. Containment of Relational Queries with Annotation Propagation. In *DBPL.* 37–53.

[40] TPC. [n. d.]. TPC benchmarks. ([n. d.]). http://www.tpc.org/

[41] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2014. Query Reverse Engineering. *The VLDB Journal* 23, 5 (2014), 721–746.

[42] Yaacov Y. Weiss and Sara Cohen. 2017. Reverse Engineering SPJ-Queries from Examples. In *PODS.* 151–166.

[43] Meihui Zhang, Hazem Elmeleegy, Cecilia M. Procopiuc, and Divesh Srivastava. 2014. Reverse Engineering Complex Join Queries. In *SIGMOD.* 809–820.

[44] Moshé M. Zloof. 1975. Query by Example. In *AFIPS NCC.* 431–438.

A Six-dimensional Analysis of In-memory Aggregation

Puya Memarzia [1], Suprio Ray [2], and Virendra C. Bhavsar [3]

Faculty of Computer Science, University of New Brunswick, Fredericton, Canada {[1] pmemarzi, [2] sray, [3] bhavsar}@unb.ca

ABSTRACT

Aggregation is widely used to extract useful information from large volumes of data. In-memory databases are rising in popularity due to the demands of big data analytics applications. Many different algorithms and data structures can be used for in-memory aggregation, but their relative performance characteristics are inadequately studied. Prior studies in aggregation primarily focused on small selections of query workloads and data structures, or I/O performance. We present a comprehensive analysis of in-memory aggregation that covers baseline and state-of-the-art algorithms and data structures. We describe 6 analysis dimensions which represent the independent variables in our experiments: (1) algorithm and data structure, (2) query and aggregate function, (3) key distribution and skew, (4) group by cardinality, (5) dataset size and memory usage, and (6) concurrency and multithread scaling. We conduct extensive evaluations with the goal of identifying the trade-offs of each algorithm and offering insights to practitioners. We also provide a glimpse on how the CPU cache and TLB are affected by these dimensions. We show that some persisting notions about aggregation, such as the relative performance of hashing and sorting, do not necessarily apply to modern platforms and state-of-the-art implementations. Our results show that the ideal approach in a given situation depends on the input and the workload. For instance, sorting algorithms are faster in holistic aggregate queries, whereas hash tables perform better in distributive queries.

1 INTRODUCTION

Despite recent advances in processing power, storage capacity, and transfer speeds, our need for greater query efficiency continues to grow at a rapid pace. Aggregation is an essential and ubiquitous data operation used in many database and query processing systems. It is considered to be the most expensive operation after joins, and is an essential component in analytical queries. All 21 queries in the popular TPC-H benchmark include aggregate functions [12]. A common aggregation workload involves grouping the dataset tuples by their key and then applying an aggregate function to each group.

Many prior studies on in-memory aggregation limited the scope of their research to a narrow set of algorithms, datasets, and queries. For example, many studies do not evaluate holistic aggregate functions [11, 32, 49]. The datasets used in most studies are based on a methodology proposed by Grey et al. [18]. These datasets do not evaluate the impact of data shuffling, or enforce deterministic group-by cardinality where possible. Some studies only evaluate a proposed algorithm against a naive implementation, rather than comparing it with other state-of-the-art implementations [20, 45]. Other studies have focused on secondary aspects, such as optimizing query planning for aggregations [48], distributed and parallel algorithms [11, 20, 40, 49, 50], and iceberg queries [45]. Additionally, some data structures have been proposed for in-memory query processing or as drop-in replacements for other popular data structures, but have not been extensively studied in the context of aggregation workloads [4, 26, 29]. Real-world applications cover a much more diverse set of scenarios, and understanding them requires a broader and more fundamental view. Due to these limitations, it is difficult to gauge the usefulness of these studies in other scenarios.

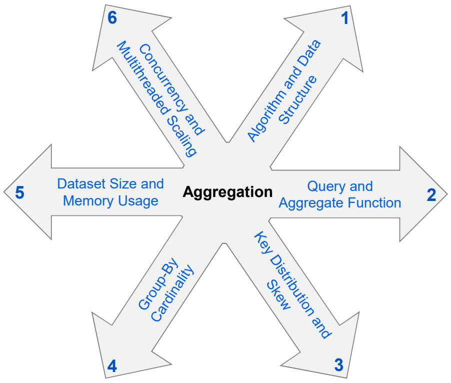

6
Concurrency and Multithreaded Scaling
1
Algorithm and Data Structure
5
Dataset Size and Memory Usage
Aggregation
Query and Aggregate Function
2
4
Group-By Cardinality
Key Distribution and Skew
3

Figure 1: An overview of the analysis dimensions

Different combinations of methodologies and evaluation parameters can produce very different conclusions. Applying the results from an isolated study to a general case may result in poor performance. For example, methods and optimizations for distributive aggregation are not necessarily ideal for holistic workloads. Our goal is to conduct a comprehensive study on the fundamental algorithms and data structures used for aggregation. This paper examines six fundamental dimensions that affect main memory aggregation. These dimensions represent well-known parameters which can be used as independent variables in our experiments. Figure 1 depicts an overview of the analysis dimensions. Figure 12 depicts a decision flow chart that summarizes our observations.

Dimension 1: Algorithm and Data Structure. In recent years, there have been many studies on main-memory data structures, such as tree-based indexes and hash tables. Many of these data structures can be used for in-memory aggregation. Aggregation algorithms can be categorized by the data structure used to store the data. Based on this we divide the algorithms into three main categories: sort-based, hash-based, and tree-based algorithms. We propose a framework that aims to cover many of the scenarios that could be encountered in real workloads. Over the course of this paper, we evaluate and discuss implementations from all three categories.

Dimension 2: Query and Aggregate Function. An aggregation query is primarily defined by its aggregate function. These functions are typically organized into three categories: distributive, algebraic, and holistic [17]. Distributive aggregate functions, such as *Count*, can be independently computed in a distributed manner. Algebraic aggregates are constructed by combining several distributive aggregates. For example, the *Average* function is a combination of *Sum* and *Count*. Holistic aggregate functions, such as *Median*, cannot be distributed in the same way as the two

Series ISSN: 2367-2005

10.5441/002/edbt.2019.26

previous categories because they are sensitive to the sort order of the data. Aggregation queries are also categorized based on whether their output is a single value (scalar), or a series of rows (vector). We evaluate a set of queries that cover both distributive and holistic, and vector and scalar categories.

Dimension 3: Key Distribution and Skew. The skew and distribution of the data can have a major impact on algorithm performance. Popular relational database benchmarks, such as TPC-H [12], generally focus on querying data that is non-skewed and uniformly distributed. However, it has been shown that these cases are not necessarily representative of real-world applications [13, 22]. Recently, researchers have proposed a skewed variant of the TPC-H benchmark [10]. The sizes of cities and the length and frequency of words can be modeled with Zipfian distributions, and measurement errors often follow Gaussian distributions [18]. Furthermore, skewed keys can be encountered as a result of joins [6] and composite queries. Our datasets are based on the specifications defined by [18] with a few additions. We cover the impact of both skew and ordering.

Dimension 4: Group-by Cardinality. Group-by cardinality is related to skew in the sense that both dimensions affect the number of duplicate keys. However, the group-by cardinality of a dataset directly determines the size (number of groups) of the aggregation result set. Prior studies have indicated that group-by cardinality has a major impact on the relative performance of different aggregation methods [2, 20, 24, 32]. These studies claim that hashing performs faster than sorting when the group-by cardinality is low relative to the dataset size, and that this performance advantage is reversed when the cardinality is high. We find that the accuracy of this claim depends on the implementation. We evaluate the performance impact of group-by cardinality, as well as its relationship with CPU cache and TLB misses.

Dimension 5: Dataset Size and Memory Usage. Recent advances in computer hardware have encouraged the use of main-memory database systems. These systems often focus on analytical queries, where aggregation is a key operation. Although memory has become cheaper and denser, this is offset by the increasing demands of the industry. Our goal is to shed some light on the trade-off between memory efficiency and performance.

Dimension 6: Concurrency and Multithreaded Scaling. Nowadays, query processing systems are expected to support intra-query parallelism in addition to interquery parallelism. Concurrency imposes additional challenges, such as reducing synchronization overhead, eliminating race conditions, and multithreaded scaling. We explore the viability and scalability of several multi-threaded implementations.

The key contributions of this paper are:

- Evaluation of aggregation queries using sort-based, hash-based, and tree-based implementations
- Methodology to generate synthetic datasets that expands on prior work
- Extensive experiments that include comparison of distributive and holistic aggregate functions, vector and scalar aggregates, range searches, evaluation of memory efficiency and TLB and cache misses, and multithreaded scaling
- Insights on performance trends and suggestions for practitioners

The remainder of this paper is organized as follows: We describe the queries in Section 2. We elaborate on the algorithms and data structures in Section 3. In Section 4 we specify the characteristics of our synthetic datasets. We present and discuss our experimental setup and evaluation results in Section 5, and summarize our findings in Section 6. In Section 7 we categorize and explore the related work. Finally, we conclude the paper in Section 8.

2 QUERIES

In this section, we describe the queries used for our experiments. In Table 1 we describe each query along with a simple example. Our goal is to evaluate and compare different aggregation variants. There are three main categories of aggregate functions: distributive, algebraic, and holistic. Distributive functions, such as *Count* and *Sum*, can be processed in a distributed manner. This means that the input can be split and processed in multiple partitions, and then the intermediate results can be combined to produce the final result. Algebraic functions consist of two or more Distributive functions. For instance, *Average* can be broken down into two distributive functions: *Count* and *Sum*. Holistic aggregate functions cannot be decomposed into multiple distributive functions, and require all of the input data to be processed together. For example, if an input is split into two partitions and the *Mode* is calculated for each partition, it is impossible to accurately determine the Mode for the total dataset. Other examples of Holistic functions include *Rank*, *Median*, and *Quantile*.

The output of an aggregation can be either in *Vector* or *Scalar* format. In Vector aggregates, a row is returned in the output for each unique key in the designated column(s). These columns are commonly specified using the *group-by* or *having* keywords. The output value is returned as a new column next to the group-by column. Scalar aggregates process all the input rows and produce a single scalar value as the result.

Sometimes it is desirable to filter the aggregation output based on user defined thresholds or ranges. We study an example of a range search combined with a vector aggregate function in *Q7*. In a real-world environment, it may be possible to push the range conditions to an earlier point in the query plan, but if several different range scans are desired, early filtering may not be possible. The main purpose of this query is to evaluate each data structure's efficiency at performing a range search in addition to the aggregation.

3 DATA STRUCTURES AND ALGORITHMS

In this section, we introduce the data structures and algorithms that we use to implement aggregate queries. We divide these algorithms into three categories: sort-based, hash-based, and tree-based. In order to facilitate reproducibility, we have selected open-source data structures and sort algorithms where possible. We also consider several state-of-the-art data structures, such as ART[26], HOT[8], and Libcuckoo[29]. Since the performance of algorithms can shift with hardware architectures, we also consider some of the more fundamental algorithms and data structures, such as a B+Tree [7]. Throughout this section we will state theoretical time complexities using n as the number of elements and k as the number of bits per key.

The implementation of an aggregate operator can be broken down into two main phases: the *build* phase and the *iterate* phase. Consider this example using a hash table and a vector aggregate function (refer to Q1 in Table 1). During the build phase, each key (created from the group-by attribute or attributes) is looked up in the hash table. If it does not exist, it is inserted with a starting value of one. Otherwise, the value for the existing key is incremented. Once the build phase is complete the iterate phase reads the key-value pairs from the hash table and writes

Table 1: Aggregation Queries

Query	SQL Representation (example)	Aggregate Function	Category	Output Format

Q1	`SELECT product_id, COUNT(*)`
	`FROM sales GROUP BY product_id`
		Count	Distributive	Vector

Q2	`SELECT student_id, AVG(grade)`
	`FROM grades GROUP BY student_id`
		Average	Algebraic	Vector

Q3	`SELECT product_id, MEDIAN(amount)`
	`FROM products GROUP BY product_id`
		Median	Holistic	Vector

Q4	`SELECT COUNT(sale_id)`
	`FROM sales`
		Count	Distributive	Scalar

Q5	`SELECT AVG(grade)`
	`FROM grades`
		Average	Algebraic	Scalar

Q6	`SELECT MEDIAN(part_id)`
	`FROM parts`
		Median	Holistic	Scalar

Q7	`SELECT product_id, COUNT(*)`
	`FROM sales WHERE product_id`
	`BETWEEN 500 AND 1000`
	`GROUP BY product_id`
		Count with Range Condition	Distributive	Vector

the resulting items to the output. A similar procedure is used for tree data structures. The calculation of the aggregate value during the build phase (early aggregation) is only possible when the aggregate function is distributive or algebraic. As a result, holistic aggregate values cannot be calculated until all records have been inserted. Sort-based approaches "build" a sorted array using the group-by attributes. As a result, all the values for each group are placed in consecutive locations. The aggregate values are calculated by iterating through the groups.

3.1 Sort-based Aggregation Algorithms

Sorting algorithms are a crucial building block in any query processing system. Many popular database systems, such as Microsoft SQL and Oracle, employ both sort-based and hash-based algorithms. We examine several algorithms designed for sorting arrays of fixed length integers, although some of the approaches could be adapted to variable length strings.

3.1.1 ***Quicksort.*** Quicksort is a sorting method based on the concept of divide and conquer that was invented by Tony Hoare [19] and remains very popular to this day. The average time complexity of Quicksort is $O(n\log(n))$. The worst case time complexity is considerably worse at $O(n^2)$, but this is rare, and is mitigated on modern implementations [21, 35] .

3.1.2 ***Introsort.*** Introspective sort (Introsort) is a hybrid sorting algorithm that was proposed by David Musser [33]. Introsort can be regarded as an algorithm that builds on Quicksort and improves its worst case performance. This sorting algorithm starts by sorting the dataset with Quicksort. When the recursion depth passes a certain threshold, the algorithm switches to Heapsort. This threshold is defined as the logarithm of the number of elements being sorted. This algorithm guarantees a worst case time complexity of $O(n\log(n))$.

The GCC variant of Introsort [21] differs from the original algorithm in two ways. Firstly, the recursion depth is set to $2 * \log(n)$. Secondly, the algorithm switches to Insertion sort, which is fast on small data chunk, but has a time complexity of $O(n^2)$.

3.1.3 ***Radix Sort (MSB and LSB).*** Radix sorting works by sorting the data one bit (binary digit) at a time. There are two variants of Radix Sort, based on the order in which the bits are processed: Most Significant Bit (MSB) Radix Sort, and Least Significant Bit (LSB) Radix Sort. As the names suggest, MSB sorts the data starting from the top (leftmost) bits, and works its way down. In comparison, LSB starts from the bottom bits. The time complexity of Radix sort is $O(k * n)$ where k is the key width (number of bits in the key), and n is the number of elements.

3.1.4 ***Spreadsort.*** Spreadsort is a hybrid sorting algorithm that combines the best traits of Radix and comparison-based sorting. This algorithm was invented by Steven J. Ross in 2002 [37]. Spreadsort uses MSB Radix partitioning until the size of the partitions reaches a predefined threshold, at which point it switches to comparison-based sorting using Introsort. Comparison-based sorting is more efficient on small sequences of data compared to Radix partitioning. The time complexity of the MSB Radix phase is $O(n\log(k/s+s))$ where k is the key width, and s is the maximum number of splits (default is 11 for 32 bit integers). As mentioned, time complexity of Introsort is $O(n\log(n))$.

3.1.5 ***Sorting Microbenchmarks.*** In order to obtain a basic understanding of the performance of these algorithms and how they compare, we evaluate five integer sorting algorithms on a variety of datasets. The tested algorithms are: Quicksort, Introsort, MSB Radix Sort, LSB Radix Sort, and Spreadsort. We test each algorithm on five data distributions: random integers between one and five, random integers between one and one million, random integers between one thousand and one million, presorted sequential integers, and reverse sorted sequential integers. We measure the time to sort ten million integers from each distribution. The results, depicted in Figure 2, show that Introsort and Spreadsort generally outperform the other sorting algorithms.

3.2 Hash-based Aggregation Algorithms

Hash tables are particularly efficient in workloads that require fast random lookups, which they perform in constant time. A hash function transforms a key into an address within the table. However, hash tables do not generally guarantee any ordering of the keys (lexicographical or chronological). It is possible to pre-sort the data and construct a hash function that guarantees

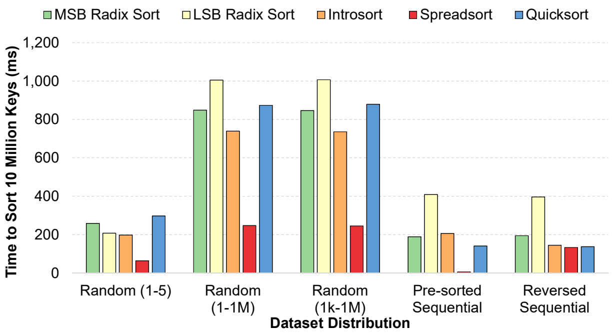


Figure 2: Sort Algorithm Microbenchmark

ordered keys (minimal perfect hashing [15, 30]). However, the impact on query execution time would be quite severe.

Hash tables are not well-suited to gradual dynamic growth, as growing the table may entail rehashing all existing elements as well. In principle, a hash table's size could be tuned to anticipate the dataset group-by cardinality. However, in practice it is difficult to estimate the cardinality, particularly when there are several group-by columns. Cardinality estimation errors result in excessive memory usage if too high, and costly rehash operations if too low. In our experiments we assume that only the size of the dataset is known, hence we set the initial size of the hash tables accordingly.

Hash tables can be categorized based on their collision resolution scheme. Collision resolution defines how a hash table resolves conflicts caused by multiple keys hashing to the same location. We now describe four collision resolution schemes and the implementations that use them: linear probing, quadratic probing, separate chaining, and cuckoo hashing.

3.2.1 ***Linear probing.*** Linear probing is part of the family of collision resolution techniques called open addressing. Open addressing hash tables typically store all the items in one contiguous array. They do not use pointers to link data items. Linear probing specifies the method used to search the hash table. An insertion begins from the hash index and probes forward in increments of one until the first empty bucket is found. Linear probing hash tables do not need to allocate extra memory to store new items as long as the table has empty slots. However, they may encounter an issue called primary clustering, where colliding records form long sequences of occupied slots. These sequences displace incoming keys, and they grow each time they do so, resulting in the high number of displacement of records.

We implement a custom linear probing hash table using several industry best practices, such as maintaining a power of two table size. If the desired size is not a power of two then the nearest greater power of two is chosen. This is a popular optimization that allows the table modulo operation to be replaced with a much faster bitwise AND. The downside to this policy is that is easier to overshoot the available memory. In order to resolve this, our implementation falls back to the modulo operation and the table size is set to the nearest prime number if possible, and the exact size parameter is used as the final fallback.

3.2.2 ***Quadratic probing.*** Quadratic probing is an open addressing scheme that is very similar to linear probing. Like linear probing it calculates a hash index and searches the table until a match is found. Rather than probing in increments of one, a quadratic function is used to determine each successive probe index. For example, with an arbitrary hash function $h(x)$ and quadratic function $f(x) = x^2$, the algorithm probes $h(x)$, $h(x)+1$, $h(x)+4$, $h(x)+9$ instead of a linear probe sequence of $h(x)$, $h(x)+1$, $h(x)+2$, $h(x)+3$. This approach greatly reduces the likelihood of clustering, but it does so at the cost of reducing data locality.

Google **Sparse Hash** and **Dense Hash** [41] are based on open addressing with quadratic probing. Sparse Hash favors memory efficiency over speed, whereas Dense Hash targets faster speed at the expense of higher memory usage.

3.2.3 ***Separate chaining.*** Separate chaining is a way of resolving collisions by chaining key-value pairs to each other with pointers. Buckets with colliding items resemble a single linked list. The main advantages of separate chaining include fast insert performance, and relatively versatile growth. The use of pointer-linked buckets reduces data locality, which is important for lookups and updates. However, unlike linear probing, separate chaining hash tables do not suffer from primary clustering.

Separate chaining hash tables remain popular in recent works [2, 3, 9, 39]. Templated separate chaining hash tables are included as part of the Boost and standard C++ libraries. Additionally, the Intel TBB library provides versatile hash tables that support concurrent insertion and iteration.

3.2.4 ***Cuckoo Hashing.*** Cuckoo hashing was originally proposed by Pagh et al. [34]. Its core concept is to store items in one of two tables, each with a corresponding hash function (this can be extended to additional tables). If a bucket is occupied by another item, the existing item is displaced and reinserted into the other table. This process continues until all items stabilize, or the number of displacements exceeds an arbitrary threshold. Cuckoo hashing provides a guarantee that reads take no more than two lookups. Its main drawback is relatively slower and less predictable insert operations, and the possibility of failed insertions.

In [29], researchers from Intel labs presented a concurrent cuckoo hashing technique **Libcuckoo**. Libcuckoo introduces improvements to the insertion algorithm by leveraging hardware transactional memory (HTM). This hardware feature allows concurrent modifications of shared data structures to be atomic. Their experimental results indicate that Libcuckoo outperforms MemC3 [14], and Intel TBB [35].

3.3 Tree-based Aggregation Algorithms

Hash-based and sort-based aggregation approaches are very popular, mainly due to a heavy focus of past studies on "write once read once" (WORO) aggregation workloads, as opposed to "write once read many" (WORM). We consider several tree data structures, and assess their viability for aggregation.

Trees are commonly used to evaluate range conditions. However, aggregation benchmarks, such as TPC-H, do not include range queries. Tree data structures are well suited to incremental dynamic growth. The trade-off is higher time complexities for both insert and lookup operations, compared to hash tables.

We divide the tree data structures into *comparison trees* and *radix trees.* Comparison trees have traditionally served as indexing structures, but Radix trees are being increasing adopted in recent main memory databases, such as HyPer [23] and Silo [44]. The *Btree* family and *Ttree* are comparison trees, and *ART* and *Judy* are Radix trees.

3.3.1 ***Btree.*** The *B-tree* is a popular tree data structure that was initially invented in 1971 by Bayer et al. [5], and forms the basis for many modern variants [7, 28]. A B-tree is a balanced *m-way* tree where *m* is the maximum number of children per

Table 2: Data Structure Time Complexity

Data Structure	Average Case Insert	Worst Case Insert	Average Case Search	Worst Case Search
ART	$O(k)$	$O(k)$	$O(k)$	$O(k)$
Judy	$O(k)$	$O(k)$	$O(k)$	$O(k)$
Btree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
Ttree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
Separate Chaining	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Linear Probing	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Quadratic Probing	$O(1)$	$O(\log(n))$	$O(1)$	$O(\log(n))$
Cuckoo Hashing	$O(1)$ (amortized)	$O(n)$ (rehash)	$O(1)$	$O(1)$

node. The B-tree is perhaps best recognized as a popular disk-based data structure used for database indexing, although they are also used for in-memory indexes. The defining characteristics of B-trees are that they are shallow and wide due to using a high fanout. This reduces the number of node lookups as each node contains multiple data items. B-trees may also include pointers between leaf nodes to facilitate more efficient range scans. The time complexity for inserting n items into a B-tree is $O(n\log(n))$. We use a cache-optimized implementation based on the *STX B+tree* [7] which we will henceforth refer to as *Btree*.

3.3.2 ***Ttree****.* *Ttree* (spelled "T-tree" in the literature) was originally proposed in 1986 by Lehman et al. [25]. Its intended purpose was to provide an index that could outperform and replace the disk-oriented B-tree for in-memory operations. Although the Ttree showed a lot of promise when it was first introduced, we show in section 3.4 that advancements in hardware design have rendered it obsolete on modern processors.

3.3.3 ***ART****.* ART (Adaptive Radix Tree) [26] is a Radix tree variant with a variable fan-out. Its inventors present it as a data structure that is as fast as a hash table, with the added bonus of sorted output, range queries, and prefix queries. ART uses SIMD instructions to concurrently compare multiple keys in parallel. ART saves on memory consumption by using dynamic node sizes and merging inner nodes when possible. Radix trees have several key advantages compared to comparison trees. The height of a radix tree depends on the length of the keys, rather than the number of keys. Additionally, in contrast with comparison trees, they do not need to perform re-balancing operations.

We also considered *HOT* [8], which builds on the same principles as ART. However, we found its performance with integer keys to be noticeably worse, as its main focus is string keys.

3.3.4 ***Judy Arrays****.* Judy Arrays (henceforth referred to as Judy) were invented by Doug Baskins [4], and defined as a type of sparse dynamic array designed for sorting, counting, and searching. They are intended to replace common data structures such as hash tables and trees. Judy is implemented as a 256-way Radix tree that uses variable fan out and a total of 20 compression techniques to reduce memory consumption and improve cache efficiency [1]. Judy is fine-tuned to minimize cache misses on 64 byte cache-lines. Like many other tree data structures, the size of a Judy array dynamically grows with the data and does not need to be pre-allocated.

3.4 Data Structure Microbenchmarks

We use a microbenchmark to evaluate each data structure's efficiency in a store and lookup workload. We separately measure the time it takes to build the data structure (build phase), and the time to read all the items in the data structure (iterate phase). All hash tables are sized to the number of elements. The results are depicted in Figure 3 using the abbreviations outlined in Table 3. With the exception of Hash_LC, the build phase is faster on all the hash tables due $O(1)$ insert complexity. Hash_LC performs poorly in the build phase because it is designed as a concurrent data structure. We evaluate its concurrent scalability in Section 5.8. Hash_LP and Hash_Dense provide the fastest overall times. Btree is noticeably faster in the iterate phase, but it takes a relatively long time to build due to the cost of balancing the tree. Due to the relatively poor performance exhibited by Ttree in both phases, we opt to omit it from subsequent experiments.

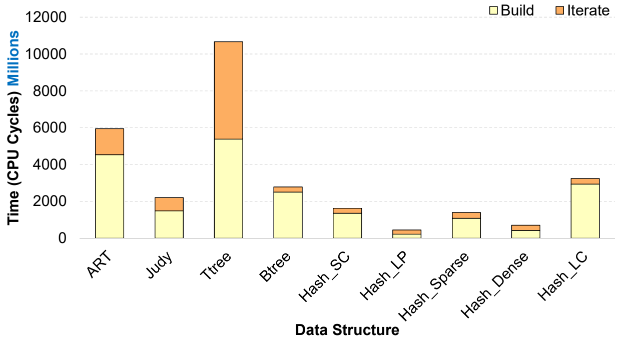


Figure 3: Data Structure Microbenchmark

3.5 Time Complexity

It is a well known fact that time complexities for algorithms are not always the best predictors of real-world performance. This is due to a number of factors, including hidden constants and overheads arising from the implementation, hardware characteristics such as CPU architecture cache and TLB, compiler optimizations, and operating systems. On modern systems, cache misses are particularly expensive. Nevertheless, time complexity is widely used as a mean to understand and compare the relative performance of different algorithms.

Table 2 provides an overview of the known time complexities for each of the data structures that we evaluate. Here n denotes the number of elements, and k the number of bits in the key.

4 DATASETS

In order to effectively evaluate the algorithms, we generate a set of synthetic datasets that vary in terms of input size, group-by cardinality, key distribution, and key range. Our datasets are based on the highly popular input distributions described in prior works [11, 18, 20]. We employ several modifications to these datasets, with the goal of expanding the data characteristics

Table 3: Algorithms and Data Structures

Label	Type	Description
ART	Tree	Adaptive Radix Tree [26]
Judy	Tree	Judy Array [4]
Btree	Tree	STX B+Tree [7]
Hash_SC	Hash	std::unordered_map [21] (Separate Chaining)
Hash_LP	Hash	Linear Probing (Custom)
Hash_Sparse	Hash	Google Sparse Hash [41]
Hash_Dense	Hash	Google Dense Hash [41]
Hash_LC	Hash	Intel libcuckoo [29]
Introsort	Sort	std::sort (Introsort) [21]
Spreadsort	Sort	Boost Spreadsort [42]

Table 4: Dataset Distributions

Abbreviation	Description	Cardinality
Rseq	Repeating Sequential	Deterministic
Rseq-Shf	Rseq Uniform Shuffled	Deterministic
Hhit	Heavy Hitter	Deterministic
Hhit-Shf	Hhit Uniformly Shuffled	Deterministic
Zipf	Zipfian	Probabilistic
MovC	Moving Cluster	Probabilistic

that we evaluate. Some datasets, such as the sequential dataset, produce very predictable patterns. For such datasets, we generate an additional variant with uniform random shuffling. In [11] it is mentioned that the group-by cardinality is often probabilistic. We enforce deterministic group-by cardinality in cases where the target distribution of the dataset would not be affected.

Throughout this paper we use *random* to refer to a uniform random function with a fixed seed, and *shuffling* refers to the use of the aforementioned function to shuffle all the records in a dataset. The number of records in the dataset is denoted as n records and the group-by cardinality is c.

In the repeating sequential dataset (*RSeq*), we generate a series of segments that contain multiple number sequences. The number of segments is equal to the group-by cardinality, and the number of records in each segment is equal to the dataset size divided by the cardinality. A shuffled variant of the repeating sequential dataset (*RSeq-Shf*) is also generated. This dataset mimics transactional data where the key incrementally increases.

In the heavy hitter dataset (*HHit*), a random key from the key range accounts for 50% of the total keys. The remaining keys are produced at least once to satisfy the group-by cardinality, and then chosen on a random basis. In a variant of this dataset, the resulting records are shuffled so that the heavy hitters are not concentrated in the first half of the dataset. Real-world examples of heavy hitters include top selling products, and network nodes with the highest traffic.

In the Zipfian dataset (*Zipf*), the distribution of the keys is skewed using Zipf's law [36]. According to Zipf's law, the frequency of each key is inversely proportional to its rank. We first generate a Zipfian sequence with the desired cardinality c and Zipf exponent $e = 0.5$. Then we take n random samples from this sequence to build n records. The final group-by cardinality is non-deterministic and may drift away from the target cardinality as c approaches n. The Zipf distribution is used to model many big data phenomena, such as word frequency, website traffic, and city population.

Table 5: Experiment Parameters

Dataset	Repeating Sequential, Heavy Hitter, Moving Cluster, Zipfian
Dataset Size	100M, 10M, 1M, 100k
Group-by Cardinality	100, 1000, 10000, 100000, 1000000, 10000000
Algorithm	Hash_LP, Hash_SC, Hash_LC, Hash_Sparse, Hash_Dense, ART, Judy, Btree, Introsort, Spreadsort, Hash_TBBSC, Sort_BI, Sort_QSLB
Thread Count	1, 2, 3, 4, 5, 6, 7, 8 (logical core count)
Query	Q1 (Vector Distributive), Q3 (Vector Holistic), Q6 (Scalar Distributive), Q7 (Vector Distributive with Range)

In the moving cluster dataset (MovC), the keys are chosen from a window that gradually slides. The i^{th} key is randomly selected from the range $\lfloor (c - W)i/n \rfloor$ to $\lfloor (c - W)i/n + W \rfloor$, where the window size $W = 64$ and the cardinality c is greater than the window size ($c >= W$). The moving cluster dataset provides a gradual shift in data locality and is similar to workloads encountered in streaming or spatial applications.

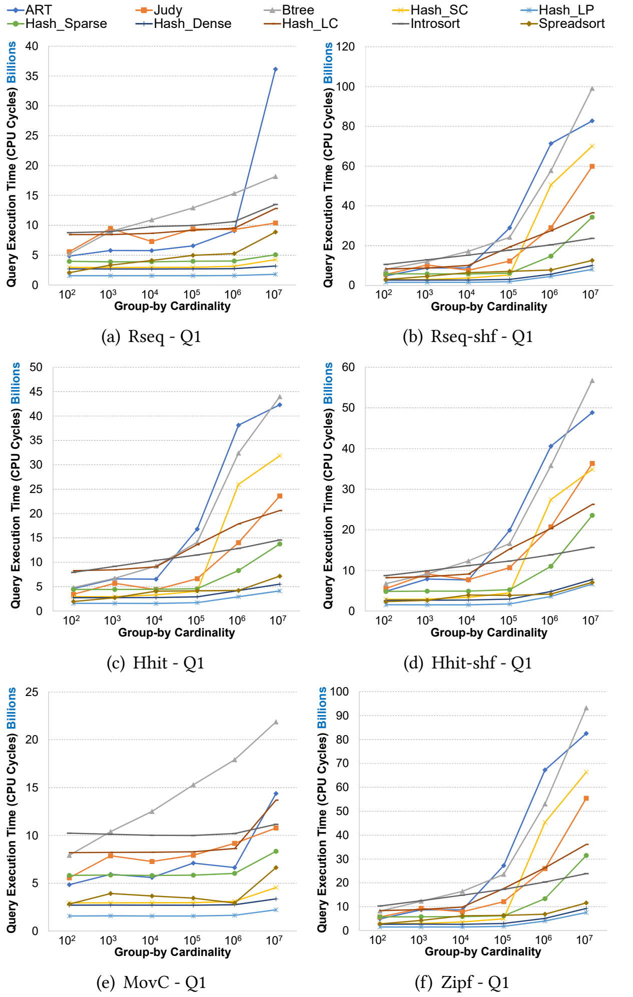
5 RESULTS AND ANALYSIS

In this section we evaluate the efficiency of the aggregation algorithms. We examine and compare the performance impact of dataset size, group-by cardinality, key skew and distribution, data structure algorithm, and the query and aggregation functions. We also evaluate peak memory usage as a measure of each algorithm's memory efficiency. The experimental parameters are outlined in Table 5.

For each experiment the input dataset is preloaded into main memory. We do not measure the time to read the input from disk. Throughout this paper we aim to understand how each of these dimensions can affect main memory aggregation workloads. Due to space constraints we only show the results for Q1, Q3, Q6 and Q7. We start the experiments with two common vector aggregation queries (Q1 and Q3) in Section 5.2. Due to the popularity of these queries, we further analyze these queries by evaluating cache and TLB misses in Section 5.3, memory usage for different dataset sizes in Section 5.4, and data distributions in Section 5.5. Additionally, we evaluate range searches (Q7) in Section 5.6 and scalar aggregation queries (Q6) in Section 5.7. We examine multithreaded scaling in Section 5.8. Finally, we summarize our findings in Section 6. We now outline our experimental setup.

5.1 Platform Specifications

The experiments are evaluated on a machine with an Intel Core i7 6700HQ processor at 3.5GHz, 16GB of DDR4 RAM at 2133MHz, and a 512GB SSD. The CPU is a quad core based on the Skylake microarchitecture, with hyper-threading (8 logical cores), 256KB of L1 cache, 1MB of L2 cache, and 6MB of L3 cache. The TLB can hold 64 entries in the L1 Data TLB, and 1536 entries in the L2 Shared TLB (4KB pages). The code is compiled and run on Ubuntu Linux 16.04 LTS, using the GCC 7.2.0 compiler, and the -O3 and -march=native optimization flags. We now present and discuss the experimental results.



(a) Rseq - Q1

(b) Rseq-shf - Q1

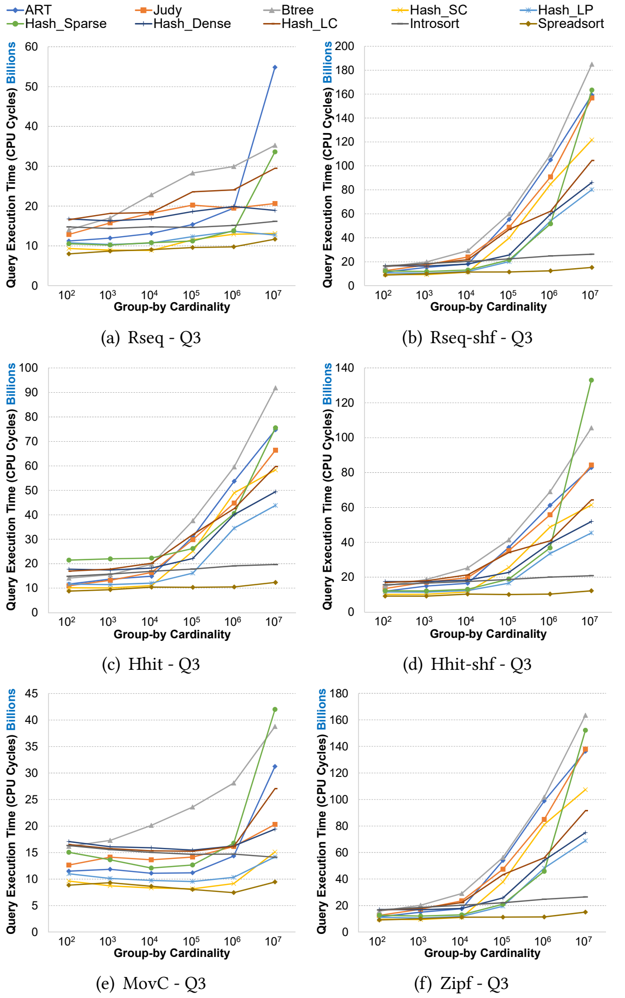
(c) Hhit - Q1

(d) Hhit-shf - Q1

(e) MovC - Q1

(f) Zipf - Q1

Figure 4: Vector Aggregation Q1 - 100M Records



(a) Rseq - Q3

(b) Rseq-shf - Q3

(c) Hhit - Q3

(d) Hhit-shf - Q3

(e) MovC - Q3

(f) Zipf - Q3

Figure 5: Vector Aggregation Q3 - 100M records

5.2 Results - Vector Aggregation

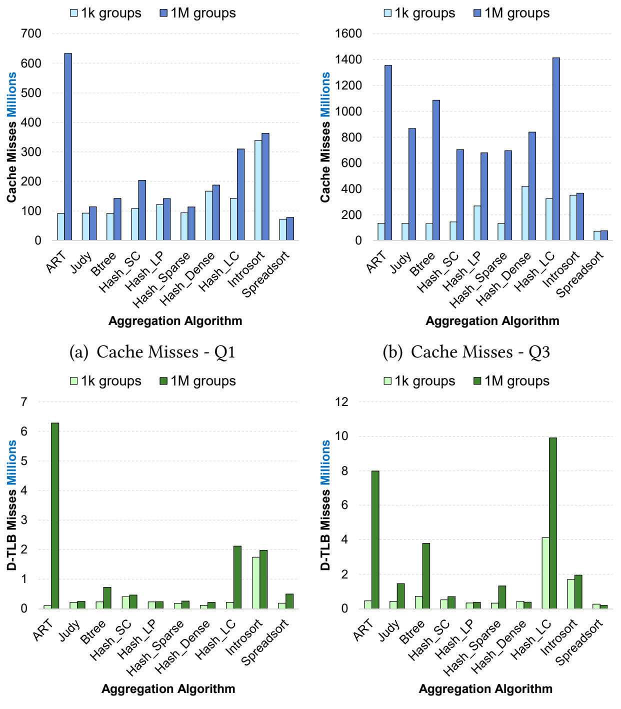
We begin our experiments by evaluating Q1 and Q3 (see Table 1), which are based on commonly used aggregate functions. Due to space constraints and the similarity between Algebraic and Distributive functions, we do not show results for Q2. In these experiments, we keep the dataset size at a constant 100M and vary the group-by cardinality from 10^2 to 10^7. In each chart we measure the query execution time for a given query and dataset distribution, and the group-by cardinality increases from left to right. The results for Q1 (Vector COUNT) and Q3 (Vector MEDIAN) are shown in Figures 4 and 5 respectively. A larger group-by cardinality means more unique keys, and fewer duplicates. In tree-based algorithms the data structure dynamically grows to accommodate the group-by cardinality. This is reflected in the gradual increase in query execution time. The insert performance of ART and Judy depends on the length of the keys, which increases with cardinality. Additionally, the compression employed by ART and Judy are more heavily used at high cardinality.

The results for Q3 show that Spreadsort is the fastest algorithm across the board. The overall trend shows that hash-based algorithms, such as Hash_SC and Hash_LP, are competitive with Spreadsort until the group-by cardinality exceeds 10^4. The execution times for both Spreadsort and Introsort show considerably less variance, whereas the worsening of data locality results in sharp declines in performance for the hash-based and tree-based implementations. The performance of Hash_Sparse dramatically worsens at 10^7 groups, suggesting that the combination of Hash_Sparse's gradual growth policy and the extra space needed by this query result in a much steeper decline in performance compared to Q1.

In order to understand why Hash_LP outperforms all the other algorithms in Q1, we need to consider several factors. Firstly, the average insert time complexity (as shown in Table 2) is unaffected by group-by cardinality. Secondly, the cache-friendly layout of Hash_LP takes great advantage of data locality compared to the other hash tables. Lastly, compared to Q3, Q1 does not require additional memory to store the values associated with each key. This reduces the pressure on the cache and TLB, and allows Hash_LP to compete with memory efficient approaches such as Spreadsort. We further explore cache and TLB behavior in Section 5.3 and memory consumption in Section 5.4.

5.3 Results - Cache and TLB misses

Cache and TLB behavior are metrics of algorithm efficiency. Together with runtime and memory efficiency, they paint a picture of how different algorithms compare with each other. Processing large volumes of data in main memory often leads to many cache and TLB misses, which can hinder performance. A cache miss can sometimes be satisfied by a TLB hit, but a TLB miss incurs a page table lookup, which is considerably more expensive. Using



(a) Cache Misses - Q1

(b) Cache Misses - Q3

(c) TLB misses - Q1

(d) TLB misses - Q3

Figure 6: Cache and TLB misses - Rseq 100M Dataset

the *perf* tool, we measure the CPU cache misses and data-TLB (D-TLB) misses of Q1 and Q3 with low cardinality (10^3 groups) and high cardinality (10^6 groups) datasets. The results are depicted in Figure 6.

It is interesting to compare the results in Figure 6(c) with the performance discrepancy between Hash_LP and Spreadsort in Q1. At low cardinality, the number of TLB misses is relatively close between the two algorithms. However, at high cardinality, Spreadsort exhibits considerably higher TLB misses. Similarly, in Figure 4(a) we see the runtime gap between the two algorithms widen in Hash_LP's favor as the cardinality increases to 10^7.

Although this metric is not a guaranteed way to predict the relative performance of the algorithms, it is a fairly reliable measure of scalability and overall efficiency. The cache behavior of Spreadsort is consistently good. Other algorithms, such as ART, exhibit large jumps in both cache and TLB misses. This correlates with similar gaps in the query runtimes, and it is noted in [8] that ART's memory efficiency and performance may degrade if it creates many small nodes due to the dataset distribution.

5.4 Results - Memory Efficiency

Memory efficiency is a performance metric that is arguably as important as runtime speed. We now measure the peak memory consumption at various dataset sizes.

To do so we lock the group-by cardinality at 10^3 and vary the dataset size from 10^5 up to 10^8. These measurements are taken by using the Linux */usr/bin/time -v* tool to acquire the *maximum resident set size* for each configuration. The results for Q1 are depicted in Table 6 and the memory usage of Q3 is shown in Table 7. The results show that the hash tables consume the most memory, followed by the tree data structures. The sort algorithms are the most memory efficient because they sort the data in-place. In order to maintain good insert performance, most hash tables consume more memory than they need to store the items, and some will only resize to powers of two. Hash_Dense's memory usage is particularly high because it uses 6× the size of the entries in the hash table when performing a resize. After the resize is completed, the memory usage shrinks down to 4× the previous size. Comparing Tables 6 and 7, we see a jump in memory usage from Q1 to Q3. This is due to the fact that Q3 requires the data structures to store the keys and all associated values in main memory, whereas Q1 only requires the keys and a count value. Consequently, holistic queries like Q3 will generally consume more memory.

Table 6: Peak Memory Usage (MB) - Q1 on Rseq 10^3 Groups

Dataset Size Algorithm	10^5	10^6	10^7	10^8
ART	4.45	11.61	131.61	1027.44
Judy	4.31	11.53	131.49	1027.45
Btree	4.54	11.79	131.66	1027.60
Hash_SC	5.45	19.41	159.07	1540.95
Hash_LP	5.23	18.67	156.29	1529.44
Hash_Sparse	4.61	11.94	131.68	1027.58
Hash_Dense	6.42	27.33	336.02	2814.70
Hash_LC	26.95	44.44	263.14	2069.90
Introsort	4.50	11.74	131.66	1027.59
Spreadsort	4.53	11.55	131.65	1027.44

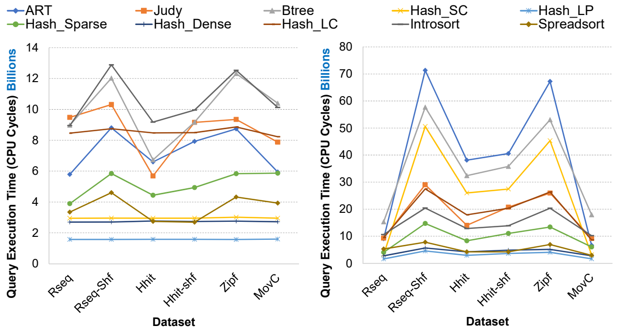
Table 7: Peak Memory Usage (MB) - Q3 on Rseq 10^3 Groups

Dataset Size Algorithm	10^5	10^6	10^7	10^8
ART	5.07	15.26	132.57	1212.46
Judy	4.87	15.37	132.68	1212.82
Btree	5.14	15.33	132.78	1212.64
Hash_SC	5.88	23.28	211.39	1986.78
Hash_LP	7.80	45.55	437.76	4264.07
Hash_Sparse	5.11	15.92	137.78	1255.51
Hash_Dense	12.74	79.16	1156.55	9404.48
Hash_LC	30.01	68.44	686.95	5575.09
Introsort	4.45	11.61	131.57	1027.50
Spreadsort	4.31	11.66	131.59	1027.48

5.5 Results - Dataset Distribution

These experiments show the performance impact of the data key distribution. The results are presented in Figures 7(a) and 7(b). In each figure, we vary the key distribution while keeping the dataset size at a constant 100 million records. To get a better understanding of how this factor ties in with cardinality, we show results for both low and high cardinality (10^3 and 10^6 groups).

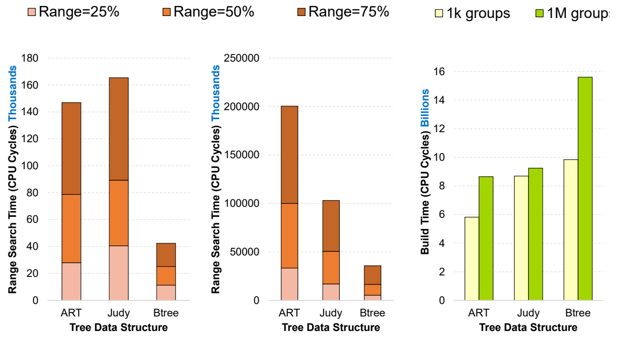
The results point out that Zipf and Rseq-Shf are generally the most performance-sensitive datasets. The shuffled variants of Rseq and HHit take longer to run due to a loss of memory locality. By comparing the two figures we can see that this effect is amplified by group-by cardinality, as it increases the range of keys that must be looked up in the cache. In the low cardinality dataset, the number of unique keys is small compared to the dataset size. Introsort is the overall slowest algorithm at low cardinality and its performance is around the middle of the pack at high cardinality. This is in line with prior works suggesting that sort-based aggregation is faster when the group-by cardinality is high [32]. However, as we can see in the results produced by Spreadsort, the algorithm also performs well at low cardinality which contradicts earlier claims. Due to this, it may be worth revisiting hybrid sort-hash aggregation algorithms in the future.



(a) 10^3 groups (low cardinality)

(b) 10^6 groups (high cardinality)

Figure 7: Vector Q1 - Variable Key Distributions - 100M records

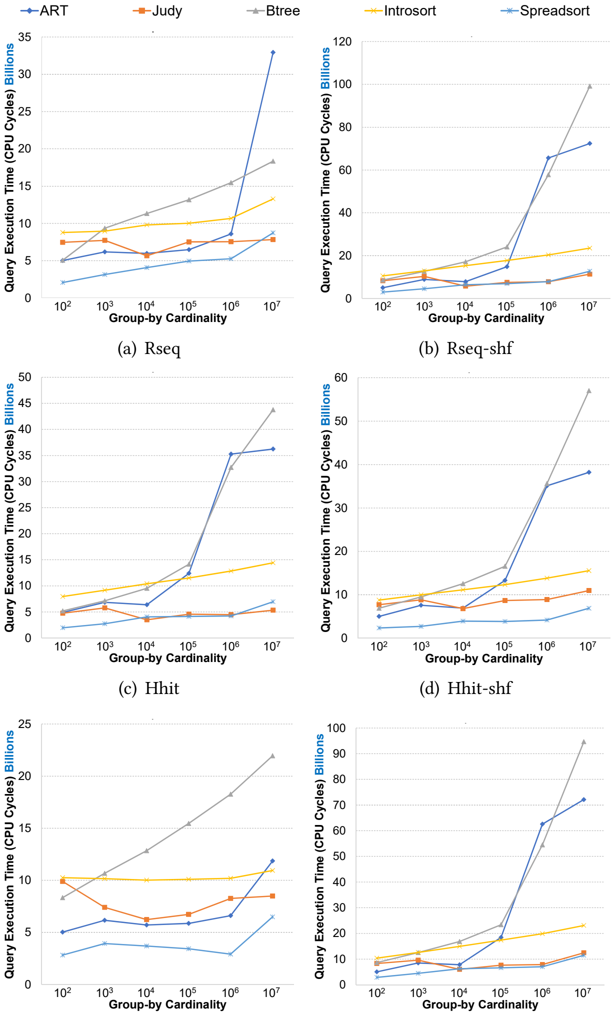


(a) Range Search Time - 10^3 groups (low cardinality)

(b) Range Search Time - 10^6 groups (high cardinality)

(c) Build time by dataset cardinality

Figure 8: Range Search Aggregation Q7 - 100M records



(a) Rseq

(b) Rseq-shf

(c) Hhit

(d) Hhit-shf

(e) MovC

(f) Zipf

Figure 9: Scalar Aggregation Q6 - 100M records

The results also highlight an interesting trend when it comes to shuffled/unordered data. Observe that ART's performance in Figure 7(b) significantly worsens when going from Rseq to Rseq-Shf or indeed any unordered distribution. The combination of high cardinality and unordered data increase pressure on the cache and TLB. If we consider how well Spreadsort performs in these situations, then the results indicate that presorting the data before invoking the ART-based aggregate operator could significantly improve performance. However, careful consideration of the algorithm and dataset is required to avoid increasing the runtime.

5.6 Results - Range Search

The goal of this experiment is to evaluate algorithms that provide a native range search feature, and combine this with a typical aggregation query. Although it is possible to implement an integer range search on a hash table, this would not work for strings and other keys with non-discrete domains. Consequently, we focus on the tree-based aggregation algorithms. Q7 calculates the vector count aggregates for a range of keys. The tuples that do not satisfy the range condition could be filtered out before building the index (if the range is known in advance). We assume that (a) the data has already been loaded into the data structure, and (b) this is a a Write Once Read Many (WORM) workload, and multiple range searches will be satisfied by the same index.

We evaluate the time it takes to perform a range search on each of the tree-based data structures for ranges that cover 25%, 50% and 75% of the group-by cardinality (the smaller ranges are run first). The results are shown in Figure 8. In Figure 8(c), we see that the time to build the tree dominates the runtime. The search times shown in Figures 8(a) and 8(b) indicate that Btree significantly outperforms the other algorithms if the tree is prebuilt. This is likely due to the pointers that link each leaf node, resulting in one $O(\log(n))$ lookup operation to find the lower bound of the search, and a series of pointer lookups to complete the search. At low cardinality (10^3 groups), the range search time for ART is 12% lower than Judy, but it is 94% higher at high cardinality (10^6 groups). If we factor in the build time and consider the workload WORO, then ART is the fastest algorithm.

5.7 Results - Scalar Aggregation

Unlike Vector aggregate functions, which output a row for each unique key, Scalar aggregate functions return a single row. We evaluate Q6 on the tree-based and sort-based aggregation algorithms. Hash tables are unsuitable for this query because the keys need to be in lexicographical order to calculate the median. Figure 9 shows the query execution time for Q6 with different datasets. The overall winner of this workload is the Spreadsort algorithm. In the case of a dynamic or WORM workload, a tree-based algorithm would have two advantages of faster lookups and requiring considerably less computation for new inserts. A good candidate for tree-based scalar aggregation is Judy, as it outperforms Introsort on all the datasets, and comes close to the

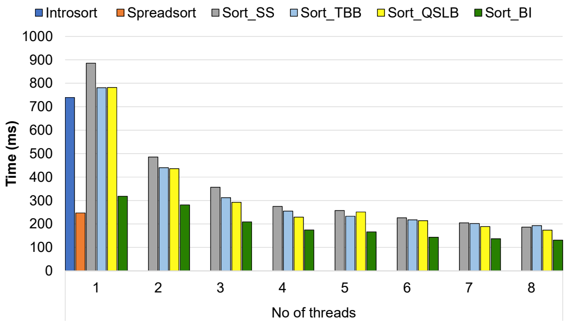
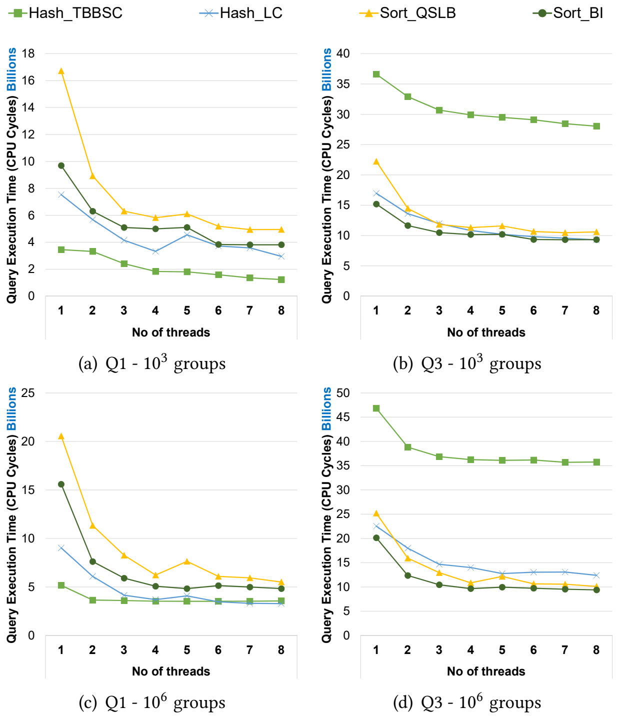


Figure 10: Parallel Sort Algorithm Microbenchmark



(a) Q1 - 10^3 groups

(b) Q3 - 10^3 groups

(c) Q1 - 10^6 groups

(d) Q3 - 10^6 groups

Figure 11: Multithreaded Scaling - Rseq 100M

performance of Spreadsort in three out of the six datasets. Although ART wins over Judy in some cases, it is inconsistent and its worse case performance is significantly worse, rendering it a poor candidate for this workload. The conclusion is in line with our expectation. To calculate the scalar median of a set of keys, Spreadsort is the fastest algorithm. If an index has already been built then Judy is usually the quickest in producing the answer.

5.8 Multithreaded Scalability

A concurrent algorithm's ability to provide a performance advantage over a serial implementation depends on two main factors: the problem size, and the algorithmic efficiency. Considerations pertaining to algorithmic efficiency include various overheads associated with concurrency, such as contention and synchronization. In order to implement concurrent aggregate operators, a suitable data structure must fulfill three requirements. Firstly, they must be designed for data-level parallelism that can scale with an increasing number of threads. Secondly, they must support thread-safe insert and update operations. It is not uncommon to encounter data structures that support concurrent *put* and *get* operations, but provide no way to safely modify existing values. Lastly, they must provide a means to iterate through their content, preferably without requiring prior knowledge of the range of values. In this section, we evaluate the performance and scalability of concurrent data structures and algorithms which fulfill all three criteria. We considered and ultimately rejected several candidate tree data structures. HOT [8] does not support concurrent incrementing of values (needed by Q1) or multiple values per key (needed by Q3). BwTree [28, 46] is a concurrent B+Tree originally proposed by Microsoft. However, our preliminary experiments found its performance to be very poor, as limitations in its API prevent efficient update operations. These characteristics have been discovered by other researchers as well [47]. The concurrent variant of ART [27] currently lacks any form of iterator, which is essential to our workloads.

Table 8: Concurrent Algorithms and Data Structures

Label	Type	Description
Hash_TBBSC	Hash	TBB Separate Chaining (Concurrent Unordered Map [35])
Hash_LC	Hash	Intel Libcuckoo [29]
Sort_BI	Sort	Block Indirect Sort [42]
Sort_QSLB	Sort	Quicksort with Load Balancing (GCC Parallel Sort [43])

We use a microbenchmark to select two parallel sorting algorithms from among four candidates. We vary the number of threads from one to eight, and include the two fastest single-threaded sorting algorithms for comparison. The workload consists of sorting random integers between 1-1M, similar to the microbenchmark presented in Section 3. The results are shown in Figure 10. Sort_BI is a novel sorting algorithm, based on the concept of dividing the data into many parts, sorting them in parallel, and then merging them [42]. Sort_TBB is a Quicksort variant that uses TBB task groups to create worker threads as needed (up to number of threads specified). Sort_SS (Samplesort) [42] is a generalization of Quicksort that splits the data into multiple buckets, instead of dividing the data into two partitions using a pivot. Lastly, Sort_QSLB [43] is a parallel Quicksort with load balancing. Considering the performance and scalability at 8 threads, we select the Sort_BI and Sort_QSLB algorithms to implement sort-based aggregate operators.

We selected four concurrent algorithms and algorithms, listed in Table 8, all of which are actively maintained open-source projects. We introduced Hash_LC in Section 3, and Hash_TBBSC is a concurrent separate chaining hash table that is similar to Hash_SC. We evaluate the multithreaded scaling for Q1 and Q3, on both low and high dataset cardinality. The results are depicted in Figure 11. We observe that both hash tables are faster in Q1, and Hash_TBBSC outperforms Hash_LC regardless of the cardinality. Sort-based approaches take the lead in Q3. The gap between sorting and hashing increases at higher cardinalities. This echoes our previous single-threaded results. The performance of Hash_TBBSC degrades significantly in Q3, because storing the values requires the use of a concurrent data structure (in this case a concurrent vector) as the value type. This is a limitation of the hash table implementation, which results in additional overhead due to synchronization and fragmentation [35]. We also considered implementing Q3 using TBB's concurrent multimap, but the performance was significantly worse. Hash_LC does not suffer from these issues, as it provides an interface for user-defined *upsert* functions. We observe similar trends with other data distributions, which we omit here due to space constraints.

6 SUMMARY AND DISCUSSION

Based on the insights we gained from our experiments, we present a decision flow chart in Figure 12 that summarizes our main observations with regards to the algorithms and data structures. We acknowledge that our experiments do not cover all possible situations and configurations, and our conclusions are based on these computational results and observations.

We start by picking a branch depending on the output format of the aggregation query. If the query is scalar, the workload determines the best algorithm. If query workload is "Write Once Read Once" (WORO) then the Spreadsort algorithm provides the fastest overall runtimes. If we require a reusable data structure that can satisfy multiple queries of this category, then Judy is a more suitable option. Going back to the start node, if the aggregation query is vector, our decision is determined by the aggregate function category. Holistic aggregates (such as Q3) are considerably faster, and more memory efficient with the sorting algorithms, particularly Spreadsort (single-threaded) and Sort_BI (multithreaded). This advantage is more noticeable at high group-by cardinality. If the query is distributive (such as Q1) then our experiments show that Hash_LP (single-threaded) and Hash_TBBSC (multithreaded) are the fastest algorithms. For aggregate queries that include a range condition, we found that Btree greatly outperformed the other algorithms in terms of search times. This advantage is only relevant if we assume that the tree has been prebuilt. Otherwise, ART is the best performer in this category, due to its advantage in build times.

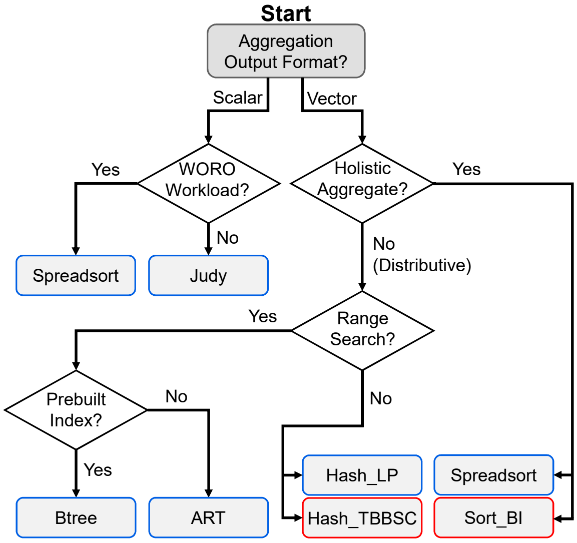


Figure 12: Decision Flow Chart

7 RELATED WORK

There have been a broad range of studies on the topic of aggregation. With the growing popularity of in-memory analytics in recent years, memory-based algorithms have gained a lot of attention. We explore some of the work that is thematically close to our research.

Some studies have proposed novel index structures for database operations. Notably, recent studies have looked into replacing comparison trees with radix trees. In [26] Leis et al. proposed an adaptive radix tree (ART) designed for in-memory query processing. The authors evaluated their data structure with the TPC-C benchmark, which does not focus on analytical queries or aggregation. Based on a similar concept Binna et al. propose HOT [8] (Height Optimized Trie). The core concepts behind this approach are reducing the height of the tree on sparsely distributed keys, and the use of AVX2 SIMD instructions for intra-cycle parallelism. The authors demonstrate that HOT significantly outperforms other indexes, such as ART [26], STX B+tree [7], and Masstree [31], on insert/read workloads with string keys. However, for integer keys, ART maintains a notable performance advantage in insert performance, and is competitive in read performance.

The duality of hashing and sorting for database operations is a topic that continues to generate interest. The preferred approach has changed many times as hardware, algorithms, and data have evolved over the years. Early database systems relied heavily on sort-based algorithms. As memory capacity increased, hash-based algorithms started to gain traction [16]. In 2009 Kim et al. [24] compared cache-optimized sorting and hashing algorithms and concluded that hashing is still superior. In [38] Satish et al. compared several sorting algorithms on CPUs and GPUs. Their experiments found that Radix Sort is faster on small keys, and Merge Sort with SIMD optimizations is faster on large keys. They predicted that Merge Sort would become the preferred sorting method in future database systems. It can be argued that this prediction has yet to materialize.

Müller et al. proposed an approach to hashing and sorting with an algorithm that can switch between them in real time [32]. The authors modeled their algorithm based on their observation that hashing performs better on datasets with low cardinality, but sorting is faster when there is high data cardinality. This holds true with some basic hashing or sorting algorithms, but there are algorithms for which this model does not apply. Their approach adjusts to the cardinality and skew of the dataset by setting a threshold on the number of groups found in each cache-sized chunk of the data. This approach cannot be used for holistic aggregation queries, as the data is divided into chunks.

Balkesen et al. [2] compared the performance of highly optimized sort-merge and radix-hashing algorithms for joins. Their implementations leveraged the extra SIMD width and processing cores found in modern processors. They found that hashing outperformed sorting, although the gap got much smaller for very large datasets. The authors predicted that sorting may eventually outperform hashing in the future, if the SIMD registers and data key sizes continue to expand.

Parallel aggregation algorithms focus on determining efficient concurrent designs for shared data structures. A key question in parallel aggregation is whether threads should be allowed to work independently, or to work on a shared data structure. Cieslewicz et al. [11] present a framework to select a parallel strategy based on a sample from the dataset. Surprisingly, the authors claim that sort-based aggregation can only be faster than hash-based aggregation if the input is presorted. We found that in the context of single threaded algorithms, sort-based aggregation is quite competitive with hash-based.

In [49], the authors examined several previously proposed parallel algorithms, and propose a new algorithm called PLAT based on the concept of partitioning, and a combination of local and global hash tables. Most of these algorithms do not support holistic aggregation, because they split the data into multiple hash tables in order to reduce contention. Furthermore, none of the algorithms are ideal for scalar aggregation as they do not guarantee lexicographical ordering of the keys.

8 CONCLUSION

Aggregation is an integral aspect of big data analytics. With rising RAM capacities, in-memory aggregation is growing in importance. There are many different factors that can affect the performance of an aggregation workload. Knowing and understanding these factors is essential for making better design and implementation decisions.

We presented a six dimensional analysis of in-memory aggregation. We used microbenchmarks to assess the viability of 20 different algorithms, and implemented aggregation operators using 14 of those algorithms. Our extensive experimental framework covered a wide range of data structures and algorithms, including serial and concurrent implementations. We also varied the query workloads, datasets, and the number of threads. We gained a lot of useful insights from these experiments. Our results show that some persisting notions about aggregation do not necessarily apply to modern hardware and algorithms, and that there are certain combinations that work surprisingly better than conventional wisdom would suggest (see Figure 12).

To our knowledge, this is the first performance evaluation that conducted such a comprehensive study of aggregation. We demonstrated with extensive experimental evaluation that the ideal approach in a given situation depends on the input and the workload. For instance, sorting-based approaches are faster in holistic aggregation queries, whereas hash-based approaches perform better in distributive aggregation queries.

REFERENCES

[1] Victor Alvarez, Stefan Richter, Xiao Chen, and Jens Dittrich. 2015. A comparison of adaptive radix trees and hash tables. In *ICDE*. IEEE, 1227–1238.

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *VLDBJ* 7, 1 (2013), 85–96.

[3] Çağrı Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2015. Main-memory hash joins on modern processor architectures. *TKDE* 27, 7 (2015), 1754–1766.

[4] Doug Baskins. 2002. General purpose dynamic array - Judy. http://judy.sourceforge.net/index.html.

[5] R Bayer and E Mccreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1 (1972), 173–189.

[6] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in Parallel Query Processing. In *PODS*. ACM, 212–223.

[7] Timo Bingmann. 2013. STX B+ Tree C++ Template Classes. https://github.com/bingmann/stx-btree.

[8] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD*. ACM, 521–534.

[9] Spyros Blanas, Yinan Li, and Jignesh M Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*. ACM, 37–48.

[10] Peter Boncz, Angelos-Christos Anatiotis, and Steffen Kläbe. 2017. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 103–119.

[11] John Cieslewicz and Kenneth A Ross. 2007. Adaptive aggregation on chip multiprocessors. In *VLDBJ*. 339–350.

[12] Transaction Processing Performance Council. 2017. TPC-H benchmark specification 2.17.3. http://www.tpc.org/tpch.

[13] Alain Crolotte and Ahmad Ghazal. 2012. Introducing Skew into the TPC-H Benchmark. In *TPCTC*. 137–145.

[14] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing.. In *NSDI*, Vol. 13. 371–384.

[15] Edward A Fox, Qi Fan Chen, Amjad M Daoud, and Lenwood S Heath. 1991. Order-preserving minimal perfect hash functions and information retrieval. *TOIS* 9, 3 (1991), 281–308.

[16] Goetz Graefe, Ann Linville, and Leonard D. Shapiro. 1994. Sort vs. hash revisited. *TKDE* 6, 6 (1994), 934–944.

[17] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery* 1, 1 (1997), 29–53.

[18] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J Weinberger. 1994. Quickly generating billion-record synthetic databases. In *SIGMOD*. ACM, 243–252.

[19] C. A. R. Hoare. 1961. Algorithm 64: Quicksort. *CACM* 4, 7 (1961), 321.

[20] Peng Jiang and Gagan Agrawal. 2017. Efficient SIMD and MIMD parallelization of hash-based aggregation by conflict mitigation. In *ICS*. ACM, 24.

[21] Nicolai M Josuttis. 2012. *The C++ standard library: a tutorial and reference*. Addison-Wesley.

[22] Thomas Kejser. 2014 (accessed June 16, 2017). TPC-H Schema and Indexes. http://kejser.org/tpc-h-schema-and-indexes/.

[23] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE, 195–206.

[24] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *VLDBJ* 2, 2 (2009), 1378–1389.

[25] Tobin J Lehman and Michael J Carey. 1986. A study of index structures for main memory database management systems. In *VLDB*, Vol. 1. 294–303.

[26] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. IEEE, 38–49.

[27] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*. ACM, 1–8.

[28] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. IEEE, 302–313.

[29] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *EuroSys*. ACM, 27:1–27:14.

[30] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. 2017. Fast and scalable minimal perfect hashing for massive key sets. *CoRR* (2017). arXiv:1702.03154 Retrieved from http://arxiv.org/abs/1702.03154.

[31] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Eurosys*. ACM, 183–196.

[32] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-efficient aggregation: Hashing is sorting. In *SIGMOD*. 1123–1136.

[33] David R Musser. 1997. Introspective sorting and selection algorithms. *Software: practice and experience* 27, 8 (1997), 983–993.

[34] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo hashing. In *ESA*. Springer, 121–133.

[35] Chuck Pheatt. 2008. Intel® threading building blocks. *Journal of Computing Sciences in Colleges* 23, 4 (2008), 298–298.

[36] David MW Powers. 1998. Applications and explanations of Zipf's law. In *NeMLaP3/CoNLL98*. Association for Computational Linguistics, 151–160.

[37] Steven J Ross. 2002. The Spreadsort High-performance General-case Sorting Algorithm.. In *PDPTA*. 1100–1106.

[38] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D Nguyen, Victor W Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*. ACM, 351–362.

[39] Anil Shanbhag, Holger Pirk, and Sam Madden. 2016. Locality-Adaptive Parallel Hash Joins using Hardware Transactional Memory. In *Data Management on New Hardware*. Springer, 118–133.

[40] Ambuj Shatdal and Jeffrey F Naughton. 1995. Adaptive parallel aggregation algorithms. In *SIGMOD*. ACM, 104–114.

[41] Craig Silverstein. 2005. Implementation of Google sparse_hash_map and dense_hash_map. https://github.com/sparsehash/sparsehash.

[42] Steven Ross, Francisco Tapia, and Orson Peters. 2018. Boost C++ Library 1.67. www.boost.org.

[43] GCC Team et al. 2018. Gcc, the gnu compiler collection. http://gcc.gnu.org.

[44] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. ACM, 18–32.

[45] Brett Walenz, Sudeepa Roy, and Jun Yang. 2017. Optimizing Iceberg Queries with Complex Joins. In *SIGMOD*. ACM, 1243–1258.

[46] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. 2018. Building a Bw-tree takes more than just buzz words. In *SIGMOD*. ACM, 473–488.

[47] Zhongle Xie, Qingchao Cai, Gang Chen, Rui Mao, and Meihui Zhang. 2018. A Comprehensive Performance Evaluation of Modern in-Memory Indices. In *ICDE*. 641–652.

[48] Weipeng P Yan and Per-Ake Larson. 1995. Eager aggregation and lazy aggregation. In *VLDB*, Vol. 95. 345–357.

[49] Yang Ye, Kenneth A Ross, and Norases Vesdapunt. 2011. Scalable aggregation on multicore processors. In *DMSN*. 1–9.

[50] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. 2009. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*. ACM, 247–260.

BionicDB: Fast and Power-Efficient OLTP on FPGA

Kangnyeon Kim
University of Toronto
knkim@cs.toronto.edu

Ryan Johnson
Amazon Web Services
frj@amazon.com

Ippokratis Pandis
Amazon Web Services
ippo@amazon.com

Abstract

Hardware specialization has been considered as a promising way to overcome the power wall, ushering in heterogeneous computing paradigm. Meanwhile, several trends, such as cloud computing and advanced FPGA technology, are converging to eliminate the barriers to custom hardware deployment, allowing it to be both technologically and economically feasible. In this paper, we conduct a pioneering study of hardware specialization for OLTP databases and present a fast and power-efficient transaction processing system built on FPGA, called BionicDB. With an order of magnitude higher power-efficiency inherently offered by FPGA, BionicDB performs faster or comparable to state-of-the-art software OLTP system by accelerating indexing and inter-worker communication.

1 INTRODUCTION

For the past decade, multicore has been a dominant scaling path. However, multicore hardware is getting more and more stagnant over time due to the growing scalability pressure and continuing power wall[16, 21]. To tackle the situation, hardware specialization has garnered a great deal of attention. It is widely regarded as a way to provide substantial power saving, while providing application-specific acceleration at the expense of generality [22, 30, 32, 34, 38, 44, 45].

In the meantime, field-programmable gate array (FPGA) technology has been matured, making custom hardware deployment more viable and economical. Its reconfigurability can return economic gains in accommodating quickly changing application demands by lifting manufacturing burden; we can reconfigure hardware to update or patch on-the-fly, as we do so for software. More importantly, inherent power efficiency of FPGA provides a great opportunity to overcome the power wall. Although running at low clock frequency, fine-grained, massive parallelism of FPGA could potentially compensate the low clock frequency with suitable custom hardware design on top.

For these reasons, datacenters increasingly integrate FPGA as a primary platform to run various custom accelerators for some data-intensive applications, such as search engine, deep learning and OLAP databases [4, 38]. Characteristically, those are compute-bound, dataflow applications. For such workloads, FPGA's massive fine-grained parallelism holds great promise for compute acceleration while CPU's limited and stagnant parallelism becomes a major computation bottleneck in dealing with huge data volume.

However, hardware specialization for transaction processing (OLTP) has been rarely explored. It has even been regarded as questionable because of OLTP's very different characteristics from previous cases: OLTP is generally bound by memory stalls and communication, rather than computation. In this situation, computation acceleration does not promise meaningful performance gain. Meanwhile, multicore CPU is returning diminishing performance gain with growing power consumption, putting scalability efforts at risk.

Inspired by the findings and technology trends, we conclude that it is imperative to explore an OLTP-oriented hardware that is power-efficient to operate under restrictive power budget and fast enough to meet performance requirement at the same time. FPGA can easily satisfy the power goal, but the question is how fast it can be while preserving the power efficiency.

In this paper, we report the design and implementation of BionicDB on FPGA as a case of hardware specialization for OLTP. BionicDB is an OLTP-oriented hardware optimized for in-memory, partitioned databases. Preserving the inherent power efficiency of FPGA, it achieves high performance thanks to various OLTP-oriented custom hardware that accelerate 1) index and 2) inter-worker communication. Also, it takes a hybrid processor-accelerator (software-hardware) architecture to complement each other. Our experimental results show that BionicDB can achieve an order of magnitude power saving while providing competitive performance compared to state-of-the-art software system; with the same number of worker threads, BionicDB can be faster by up to 4.5x when fully utilized and maintains comparable performance in TPC-C transactions where BionicDB is substantially underutilized.

The main contributions of this paper are as follows.

- We establish a holistic strategy for fast and power-efficient OLTP through hardware specialization.
- We propose index pipelining and transaction interleaving for index acceleartion.
- We suggest on-chip message-passing for faster inter-worker communication in partitioned databases.
- We show that hybrid processor-accelerator (software-hardware) approach is required for OLTP.

The remainder of this paper is organized as follows. Section 2 covers background on FPGA. Section 3 discusses design decisions, laying the foundations of BionicDB. Section 4 describes how BionicDB works in detail. Section 5 evaluates BionicDB, comparing to a software OLTP system. Section 6 summarizes related work. Section 7 contains our conclusion and suggests possible future research directions.

2 FIELD PROGRAMMABLE GATE ARRAYS

FPGA is a reconfigurable hardware platform and becoming an enabling technology for hardware specialization for a wide variety of application. Its main advantages are low power consumption and massive fine-grained parallelism that can be leveraged for acceleration. While CPU still remains as a central processing resource, certain CPU-unfriendly work can be offloaded to custom accelerator built on FPGAs for higher efficiency.

The main building blocks of FPGA that enable reconfigurability are lookup tables (LUT), flip-flops (FF) and programmable routing fabric. The first two components are used to implement logic functions, and the routing fabric interconnects them programmably. In addition to the programmable elements, modern FPGA chips commonly include hardwired blocks for higher efficiency, such as

Series ISSN: 2367-2005 10.5441/002/edbt.2019.27

Challenge	Solution
Memory stalls	Index pipelining
Memory stalls	Transaction interleaving
Inter-worker communication	On-chip message-passing over DORA
Heavy control-flow	Hybrid processor-accelerator design

Table 1: Design summary

block RAMs (BRAMs), digital signal processors (DSPs) and even embedded processors. Hardware can be designed with hardware description language (HDL) and CAD tools provided by a vendor compile HDL code into a bitstream that physically implements the hardware design on a target chip.

FPGA trades efficiency for reconfigurability, largely because of the area/speed/power overhead from programmable fabric, residing in a middle ground between ASIC and CPU in terms of efficiency and flexibility; ASIC provides the highest efficiency with lowest flexibility, while CPU provides the opposite. But FPGA can be an excellent platform for applications where the volume of production does not justify ASIC manufacturing. In datacenters where servers are dynamically re-purposed for a changing set of applications, a FPGA-augmented server can reprogram both software and hardware at runtime, allowing more efficient resource provisioning [38]. It also provides a chance to free up a number of CPUs from cycle-consuming jobs and spare them for more suitable (SW-friendly) tasks, returning higher datacenter efficiency.

3 DESIGN

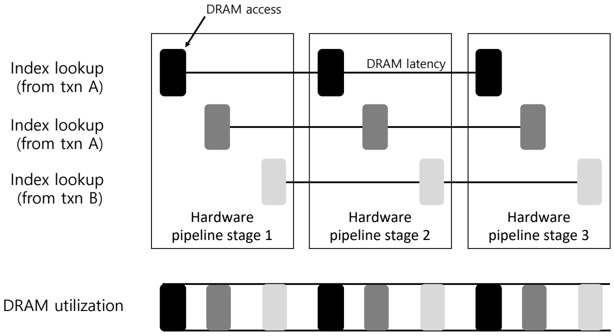
BionicDB aims to provide 1) power saving 2) and high performance at the same time for OLTP. Software-only systems on low-power processors usually end up trading one for another; [40] reports that an ARM processor sacrifices performance by 3x with merely 25% energy efficiency gain, compared to a Xeon processor when running a high-performance in-memory OLTP. Therefore, we leverage hardware specialization with FPGA to achieve both goals. Table 1 summarizes the specific challenges and solutions to address them.

3.1 Acceleration Strategy

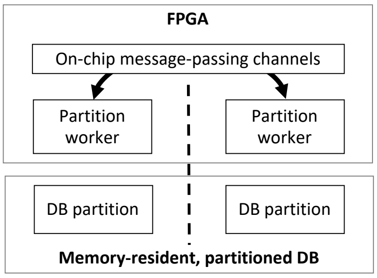
We start from understanding bottlenecks in modern OLTP to identify “what to accelerate" and establish a strategy on “how to accelerate". Typically, OLTP systems serve massively concurrent transactions that make small random updates to databases [1, 20]. Such workload characteristic naturally gives high pressure on indexing and thread coordination.

Index. Although OLTP databases heavily rely on index, the CPU is frequently bound by memory stalls from dependent pointer chasing within an index probe, wasting huge amounts of cycles. For example, a recent study reported roughly 40% overhead from index in a state-of-the-art software OLTP system [24]. But existing software solutions are limited to overcome the memory wall [19, 25]: 1) cache optimizations and bigger cache are easily undermined by OLTP's access randomness; 2) prefetching often fails to hide memory latency due to the lack of computation to overlap with; 3) software pipelining is inefficient with irregularity; 4) and group/dynamic prefetching is bound by the limited size of the instruction window of a CPU.

BionicDB alleviates the memory stalls through index pipelining. The key concept of index pipelining is to overlap multiple index accesses through hardware pipelining. It can provide higher memory-level parallelism, thereby improving single-worker performance. Also, we aim to exploit index parallelism not only within a transaction, but also across transactions. For that, transaction interleaving captures inter-transaction index parallelism, improving the utilization of index pipelining. Figure 1a illustrates how we can achieve higher memory-level parallelism with these techniques.


DRAM access
Index lookup (from txn A)
Index lookup (from txn A)
Index lookup (from txn B)
DRAM latency
Hardware pipeline stage 1
Hardware pipeline stage 2
Hardware pipeline stage 3
DRAM utilization

(a) Index acceleration by overlapping index operations.


FPGA
On-chip message-passing channels
Partition worker
Partition worker
DB partition
DB partition
Memory-resident, partitioned DB

(b) Fast on-chip message-passing for partitioned DB.

Figure 1: OLTP acceleration strategy

Communication. Partitioned databases can simplify the hardware complexity; theoretically, database resources are core-private eliminating the burden of complex thread coordination, such as concurrent index implementation. For that reason, we adopt DORA [35, 36] which is a scalable partitioned database.

In regard to performance, it has very low overhead for a single-partition transaction because inter-worker communication does not take place at all. However, it requires inter-worker communication overhead when a transaction spans across partitions (multi-site transaction). Recent studies [7, 8, 12] have revisited message-passing semantic for partitioned databases, but software message-passing can involve 1) memory latency when a message is evicted from cache and 2) thread synchronization at concurrent message queues that can be a scalability bottleneck. Despite the problems, there is no alternative or a bypass for software, because the shared-memory is the only available communication semantic in most CPUs. In other words, the shared-memory bottlenecks are inevitable with software message-passing even when application-level communication semantic is purely message-passing.

For faster communication in partitioned database, As illustrated in Figure 1b, BionicDB provides an on-chip message-passing method that can completely eliminate the overhead of software message-passing. As a result, multisite transactions, a remaining concern in partitioned databases, can be accelerated.

3.2 Hybrid Hardware-Software Approach

While software-only is inefficient for the challenges of memory stalls and communication, the heavy control-flow in transaction logic, such as conditional branches and dynamic loops, makes

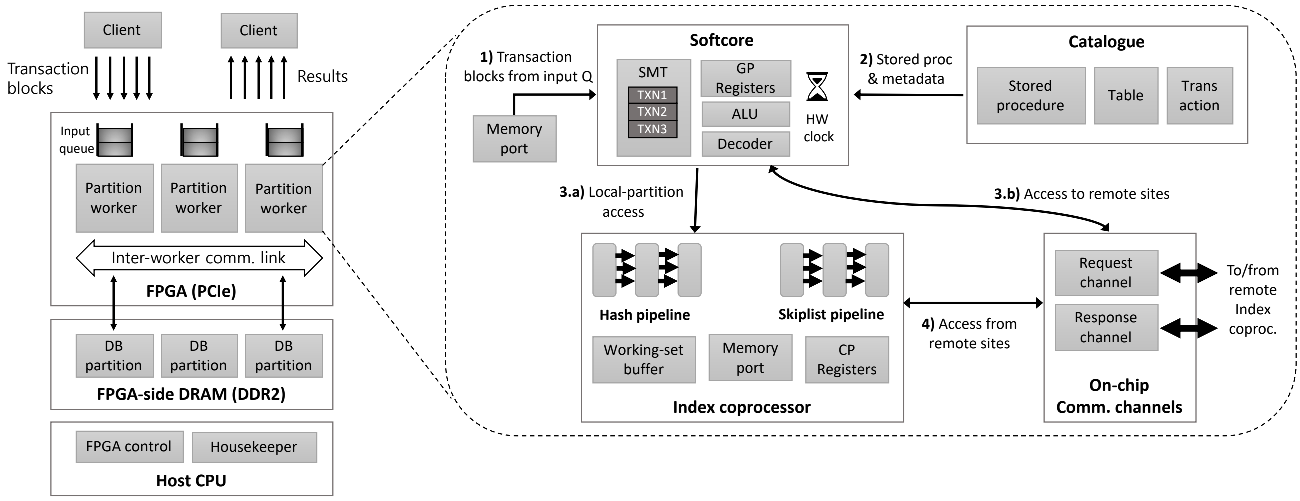


Figure 2: Architecture of BionicDB and the processing flow within a partition worker: 1) ingesting an input transaction block from memory, 2) fetching stored procedure code and metadata from the catalogue, 3.a) dispatching an index operation to the local index coprocessor 3.b) or to a remote site through the on-chip communication channels and 4) processing an index operation from remote workers.

general-purpose processor technology indispensable. This naturally leads to hybrid processor-accelerator approach that can deal with the inefficiency of both hardware-only (dynamic control-flow) and software-only (memory stalls, inter-worker communication) solutions.

The next question is how to integrate them. In our hardware environment where FPGA is connected through PCIe, the excessive PCIe latency (1us) between FPGA and host CPU can absorb most acceleration benefits. Therefore, we implement a custom softcore (microprocessor built around reconfigurable fabric) and acceleration fabric altogether on a PCIe-attached FPGA chip for tight integration, moving the majority of the OLTP components. The mission of the softcore is to execute pre-compiled stored procedures, while interacting with the index/communication acceleration fabric. As a result, hardware and software can closely collaborate, complementing each other. We design a custom core from the scratch, rather than using a vendor-provided softcore, to easily implement custom features and to integrate with the acceleration fabric more efficiently.

Although unexplored in this work, upcoming in-socket FPGA hardware [33] where FPGA is placed closer to host CPU is promising for hybrid software-hardware approach in that it allows to re-use existing software ecosystem, while enabling fast FPGA roundtrips thanks to low-latency interconnects, such as QPI.

4 ARCHITECTURE

4.1 Hardware Description

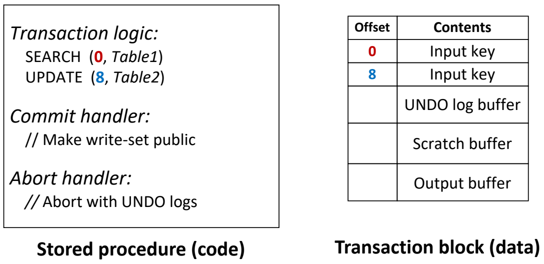
We build BionicDB on Micron HC-2 machine (formerly, Convey HC-2)[1] which contains four Xilinx Virtex-5 LX330 FPGA chips on a PCIe card and two-socket Intel Xeon X5670 processors. The memory subsystem of the FPGA card includes 64GB of DDR2 RAM, 8 memory controllers and 16 scatter-gather DIMMs. It can provide up to 80GB/s memory bandwidth for random 64-bit accesses when its 16 scatter-gather DIMMs are fully utilized with all FPGA chips enabled. However, current implementation of BionicDB uses only a single FPGA chip out of 4 and 8 DIMMs out of 16, bound by the maximum bandwidth of 10GB/s. It is also worth noting that in-memory OLTP is typically bound by memory latency, rather than bandwidth, due to small random accesses.

4.2 Overview

Figure 2 illustrates the architecture of BionicDB. BionicDB implements OLTP functionalities (stored procedure execution, indexing, concurrency control and transaction management) on a PCIe-attached FPGA chip, leaving a few background housekeeping jobs (signaling FPGA to start/stop, memory management, interactions with clients) to the host CPU. The database is partitioned, entirely residing in FPGA-side on-board DRAM. Each partition is accessed by a single partition worker in FPGA, and we fit multiple partition workers on a FPGA chip as many as its logic capacity allows for thread-level parallelism. The main components of a partition worker are 1) stored procedure execution engine and 2) acceleration fabric. The former includes the softcore and the catalogue that stores stored procedures and metadata, and the latter is composed of a local index coprocessor and communication channels.

We briefly describe the processing flow during transaction execution. The client should upload a pre-compiled stored procedure along with all metadata to the catalogue in advance. After that, the client can submit a transaction block to BionicDB for executing the transaction. Then, a partition worker fetches the transaction block from its input queue (step 1 in Figure 2). Once a transaction block arrives, the softcore fetches the corresponding stored procedure code from the catalogue and executes it (step 2 in Figure 2). During the execution of a stored procedure, it makes a database access by asynchronously passing an index request to the local index coprocessor or a remote site through the communication channels (step 3 in Figure 2). Meanwhile, the index operations issued by the local softcore (foreground requests) can be overlapped in the index coprocessor. The index operations from remote sites (background requests) also can be overlapped in the index coprocessor (step 4 in Figure 2). After a batch of transactions is entirely executed, the transactions are committed in serial order.

[1] https://www.micron.com/about/about-convey-computer-accelerator-products/hc-series



```
Transaction logic:
  SEARCH (0, Table1)
  UPDATE (8, Table2)

Commit handler:
  // Make write-set public

Abort handler:
  // Abort with UNDO logs
```

Offset	Contents
0	Input key
8	Input key
	UNDO log buffer
	Scratch buffer
	Output buffer

Figure 3: Stored procedure interface.

4.3 Softcore

In BionicDB, a transaction can be executed by invoking an associated stored procedure that is pre-compiled with BionicDB instructions (will be described in this section). Then, the softcore executes the instructions while interacting with the index accelerator where most performance gains come from.

Stored procedure execution. A stored procedure is composed of three parts: 1) transaction logic code, 2) commit handler and 3) abort handler. The first part represents the original transaction logic and the rest implements a commit protocol (currently, commit/abort handlers should be defined by users). Depending on the execution results during transaction logic, BionicDB moves on to either the commit or the abort handler.

Once a stored procedure is registered, a client can submit a transaction block to BionicDB to invoke the transaction at runtime. A transaction block contains a transaction ID, input data and buffers for result sets, intermediate data and transaction logs. When it receives a transaction block, the softcore executes the matching stored procedure code with the transaction ID, using the input data and buffers in the transaction block. In the example shown in Figure 3, the stored procedure (left) tries to search a tuple with a key at offset 0 and update a tuple with a key at offset 8 in the transaction block (right).

When generating a stored procedure, a compiler should translate SQL statements into machine code, as Hekaton does with T-SQL [14] (we used manually-written stored procedures, as the compiler is beyond the scope of this paper). A client can register a new transaction or change an existing one by uploading the stored procedure code to BionicDB along with metadata to work with, such as transaction information and table schema. It does not require FPGA reconfiguration that involves hours-long synthesis, so BionicDB can accommodate workload changes quickly.

Instruction set architecture. The instruction set of BionicDB is composed of a subset of typical CPU instructions and DB instructions that encapsulate index operations (see Table 2). Figure 4 briefly illustrates how the softcore processes each instruction type during stored procedure execution. CPU instructions are directly executed by the softcore in five steps as simple RISC CPUs do: Instruction Fetch, Decode, Execute, Memory and Writeback. We ruled out instruction pipelining and out-of-order execution as previous studies showed that such features do not translate into meaningful improvement in OLTP [6, 17].

Instruction	Type
INSERT	DB
SEARCH	DB
SCAN	DB
UPDATE	DB
REMOVE	DB
ADD/SUB/MUL/DIV/MOV	CPU
CMP	CPU
LOAD/STORE	CPU
JMP/BE/BLE/BLT/BGT/BGE	CPU
RET	CPU
COMMIT/ABORT	CPU

Table 2: BionicDB instructions. DB instructions invoke index operations provided by the index coprocessor.

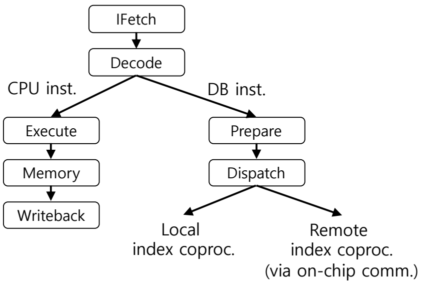


Figure 4: Instruction execution steps of the softcore.

DB instructions are added for invoking indexing services provided by the index coprocessor. For DB instructions, the softcore collects metadata in Prepare stage, such as index type (hash or skiplist) and transaction begin timestamp. In the next stage (Dispatch), the softcore passes the DB instruction with the metadata to the local index coprocessor or a remote one via on-chip communication channels, depending on the destination partition. DB instructions are asynchronously forwarded for overlapping multiple index operations.

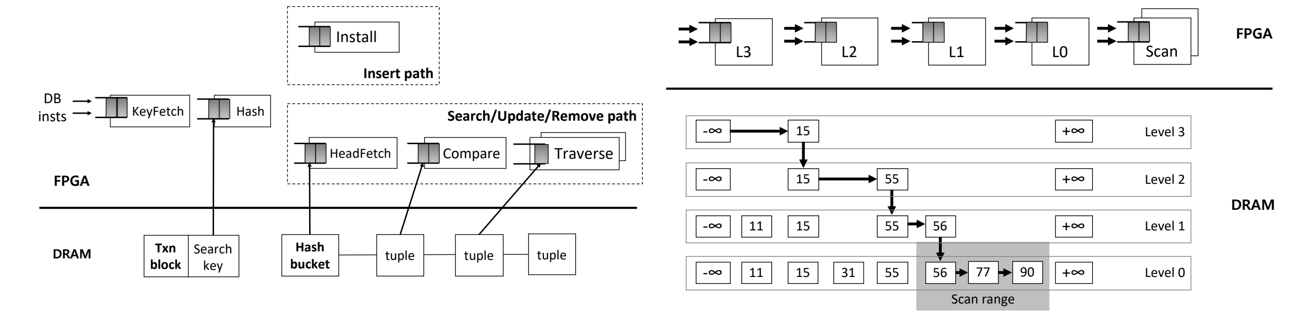
General-purpose registers (GP registers) are available to store data or pointers, and special-purpose registers, including a program counter (PC) and a status register that contains carry/zero/sign/overflow flags, are also available as typical CPUs. Additionally, coprocessor registers (CP registers) are provided; the result of a DB instruction is returned asynchronously to a CP register specified in the DB instruction. Then, the softcore can copy the result from the CP register to a GP register by executing a RET instruction. Therefore, a DB instruction must be paired with a RET instruction on the same CP register. The GP/CP register files are implemented on BRAMs, instead of flip-flops, for resource efficiency, and 256 GP/CP registers are provided to a single softcore. General-purpose cache memory was not implemented.

The addressing mode is base-offset. At the beginning of a transaction, a BionicDB worker sets a base address register with the start address of a transaction block and reaches memory locations within the block by adding the base address and an offset value. The offset value could be either the content of a GP register or an immediate value which is inlined to an instruction directly.

4.4 Index Coprocessor

The index coprocessor processes DB instructions from the local softcore or a remote one. The key technique for acceleration is hardware pipelining. We decompose an index algorithm into sub-functions that can work in parallel and implement each sub-function as a hardware pipeline stage. Each pipeline stage is a finite-state machine that is awakened on source data arrival from off-chip DRAM, performs a certain task, issues memory requests designating the next stage as a destination and moves on to the next incoming instruction. There could be multiple outstanding DB instructions between neighboring stages.

Index pipelining can overlap multiple index accesses, achieving higher memory-level parallelism. But pipeline hazards can happen when different stages access the same memory location. We describe details of each index and discuss how to prevent pipeline hazards in the following sections. Figure 5 shows the



(a) Hash index. Sub-functions of hash index operations are mapped to hardware pipeline stages.

(b) Skiplist. Sub-ranges of skiplist are mapped to hardware pipeline stages. Scan is done by dedicated modules.

Figure 5: Index pipelining to overlap multiple index accesses. Hash index deals with point access and skiplist provides range scan.

index pipelining with hash and skiplist indexes. For point access, BionicDB implements hash index that processes INSERT, SEARCH, UPDATE and REMOVE instructions. Skiplist index can handle SCAN instructions for range query, INSERT and REMOVE instructions. Both indexes support variable-length key.

4.4.1 Hash index for point access We illustrate how the HW hash index works in Figure 5a. KeyFetch stage takes a DB instruction and issues memory requests to fetch a search key from a transaction block, designating Hash stage as a destination. Then, the memory response containing the search key is queued at Hash stage. Hash stage takes the search key from its input queue, computes a hash value and loads a hash table entry with the hash value. (we use Sdbm hash function[2] for its minimal use of hardware resources; it requires neither a huge lookup table nor an expensive operation like modulo).

At this point, Hash stage forwards an INSERT instruction to Install stage and the other instructions to HeadFetch stage. For INSERT instruction, Install stage takes a hash table entry and appends a new tuple to the entry. For SEARCH/UPDATE/REMOVE instructions, HeadFetch stage checks the content of hash table entry. If the value is NULL indicating that there is no tuple installed, it returns a “NotFound" message to the destination CP register specified in the DB instruction. Otherwise, it issues memory requests to read the first item of a hash bucket. KeyComp stage compares the search key against the first item’s key. If they match, it examines visibility check (will be explained in Section 4.7) for concurrency control, otherwise, it passes the instruction to Traverse stage to follow a hash conflict chain.

Traverse stage follows a hash conflict chain until it finds a matching tuple or reaches the end of the chain. Unlike other stages that contain only computation without memory stalls, this stage could involve multiple memory stalls. Thus, Traverse could take much longer time with poorly distributed hashing. We decouple this stage from Compare stage so that an instruction that follows a long hash conflict chain does not block succeeding instructions that terminate at Compare stage. If hash conflict is frequent, multiple Traverse stages could be populated for balanced dataflow over the pipeline. Also, a hash function with good distribution and sufficiently large hash table could minimize the activation of Traverse stage by reducing hash conflicts.

Pipeline hazards and prevention. There are two hazards in hash index: insert-after-insert and search-after-insert (see Figure 6). In the former case, lost update can happen between in-flight inserts accessing the same hash table entry, if the following request reads a hash table entry before the preceding request updates it (see Figure 6a). Although not illustrated in the figure, a SEARCH instruction could read an inconsistent hash table entry at Hash stage before Install stage finishes updating it. In both cases, the reason for the hazards is that Hash and Install stages access the same hash table entry without coordination.

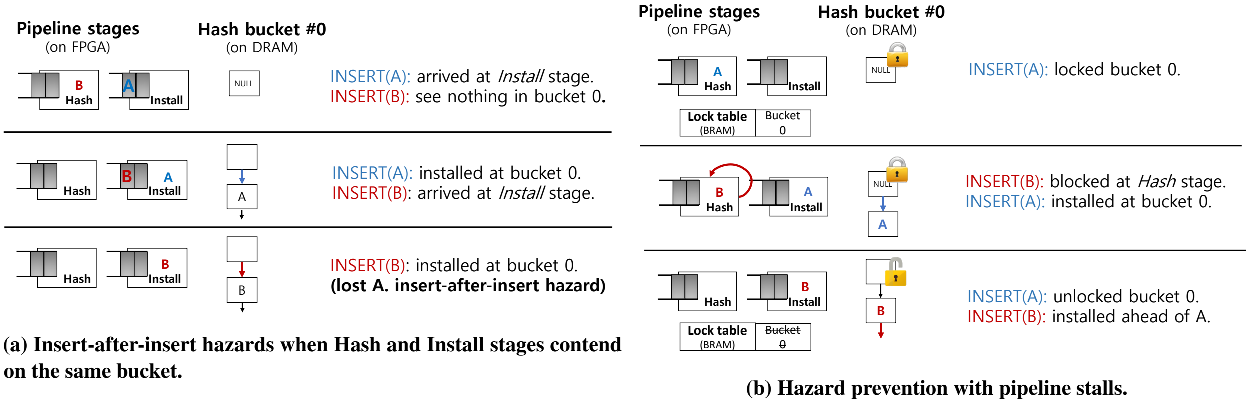
To prevent the hazards, we use a pipeline-stall based coordination scheme. BionicDB tracks the hash values of in-flight instructions that passed Hash stage in a lock table on BRAM (content-addressable memory could be used for faster check), and Hash stage checks the lock table before accessing a hash bucket. If a duplicate hash value is detected, it blocks until the lock entry disappears (see Figure 6b). The lock table entry is deleted by terminal stages when an instruction completes.

4.4.2 Skiplist for range scan For range scan, we choose skiplist because it is efficient with index pipelining for range scan (will be discussed later in this section). A skiplist index is a collection of linked lists at multiple levels [37]. A skiplist node (tower) includes a tuple and an array of pointers to the next towers at different levels. The bottom link is a list of all towers, while upper ones are short-cuts and contain towers with probabilistic distribution; the number of towers decreases exponentially from the bottom through the top, offering logarithmic time complexity on average for traversal by preferring to traverse taller towers first.

Traversal pipelining. BionicDB maps skiplist indexing on deeply pipelined datapath. Each pipeline stage covers an exclusive range of levels. We illustrate an example of index pipelining for skiplist in Figure 5b. In the example, four levels are mapped on four skiplist pipeline stages. The level 3 stage starts pointer chasing horizontally at the top level, stops at the tower having key 15 because its next tower contains an upper bound and drills down to the lower level. As it goes out-of-range, it passes the instruction to the next pipeline stage (level 2) and immediately moves on to the next incoming instruction. Meanwhile, level 2 stage takes over the instruction and performs same tasks within its coverage. The level 0 stage that exclusively owns the bottom level finally receives the request and locates the right tower for the given key, which is 56. For balanced pipelining, it is important to customize range binding properly. If skiplist towers are substantially sparser at upper levels than lower ones, upper pipeline stages could be assigned larger ranges.

Insert. During the traversal for INSERT instruction, the insert path, the pointers to the predecessor and successor towers at each

[2] http://www.cse.yorku.ca/~oz/hash.html



INSERT(A): arrived at *Install* stage.
INSERT(B): see nothing in bucket 0.

INSERT(A): installed at bucket 0.
INSERT(B): arrived at *Install* stage.

INSERT(B): installed at bucket 0.
(lost A. insert-after-insert hazard)

(a) Insert-after-insert hazards when Hash and Install stages contend on the same bucket.

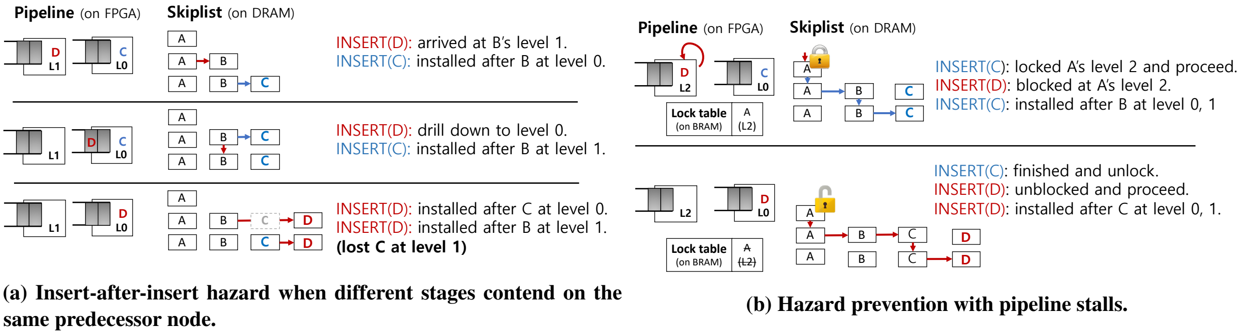
INSERT(A): locked bucket 0.

INSERT(B): blocked at *Hash* stage.
INSERT(A): installed at bucket 0.

INSERT(A): unlocked bucket 0.
INSERT(B): installed ahead of A.

(b) Hazard prevention with pipeline stalls.

Figure 6: Hash index hazards and prevention.



INSERT(D): arrived at B's level 1.
INSERT(C): installed after B at level 0.

INSERT(D): drill down to level 0.
INSERT(C): installed after B at level 1.

INSERT(D): installed after C at level 0.
INSERT(D): installed after B at level 1.
(lost C at level 1)

(a) Insert-after-insert hazard when different stages contend on the same predecessor node.

INSERT(C): locked A's level 2 and proceed.
INSERT(D): blocked at A's level 2.
INSERT(C): installed after B at level 0, 1

INSERT(C): finished and unlock.
INSERT(D): unblocked and proceed.
INSERT(D): installed after C at level 0, 1.

(b) Hazard prevention with pipeline stalls.

Figure 7: Skiplist hazard and prevention.

level below the new tower's height, is recorded in BRAMs. From the bottom level, the bottom-level stage installs a new tower on the insert path.

Scan. Scan does not require to record the path because it only needs to reach the bottom level. We assign a dedicated scanner module after the bottom-level stage. The bottom-level stage passes a scan request with the first tower in the scan range to the scanner. Then, a scanner collects committed and visible tuples within the scan range. The result scan set is stored in a designated buffer space in a transaction block (on DRAM), and the size of scan set is returned through a CP register. Like Traverse stage in hash index, decoupled scanner stages prevent long-running scan requests from blocking following requests that complete at the bottom-level stage. When necessary, redundant scanners could distribute heavy scan loads, ameliorating unbalanced dataflow.

Insert-insert hazard and prevention. In skiplist, only insert-insert hazard can happen. When consecutive inserts traverse down through common path, the traversal path of the following request could be invalidated when the preceding one has overwritten it with a new tower's address. Figure 7a shows an example of the insert-insert hazard. In the example, INSERT(D) request drills down a wrong tower (which is B) because the preceding insert C has not updated the tower at level 1. In turn, INSERT(D) installs a new tower on its inconsistent path, consequently, C is lost at level 1.

To prevent insert-insert hazard in skiplist, we apply a similar blocking policy to the hash index to prevent the skiplist hazard (see Figure 7b). For all in-flight INSERTs, we record the entry points of insert paths in a lock table. All skiplist pipelines should check the lock table before switching to the next tower or a lower level and block when encountering a locked traversal path. The lock is eliminated by the terminal stage (the bottom-level stage) when an INSERT operation completes.

Stall-free range scan. Unlike insert, skiplist scan can be stall-free, allowing efficient pipelining. A scan request might traverse down an inconsistent image of the skiplist, when missing towers installed by previous in-flight inserts at short-cut levels. However, all towers inserted previously are visible at the bottom link because a single dedicated pipeline stage serializes requests in order. After the bottom-level stage passes a scan request, a scanner starts pointer chasing from the first tower in the scan range. Therefore, scan does not miss any previous inserts in the end. During scan, the scanner could see towers inserted after scan started, but they are ignored by timestamp-based visibility check. In terms of performance, a scan request could choose a slower traversal path by missing recent short-cuts, but higher concurrency is more beneficial for overall performance.

4.5 Transaction Interleaving

The index coprocessor could be heavily underutilized with insufficient intra-transaction index parallelism when transactions are serially executed. As an extreme example, a single record transaction in key-value workloads that issues only a single index request can entirely eliminate the chance to overlap index accesses. To prevent the underutilization, the softcore exploits inter-transaction index parallelism by interleaving multiple transactions. We describe the details of transaction interleaving and discuss what factors could affect its efficiency.

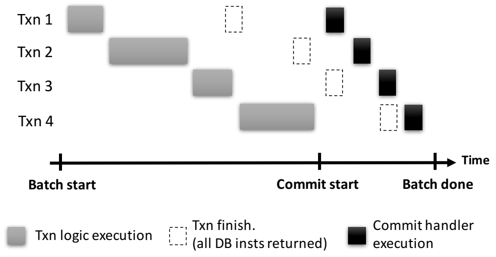


Figure 8: Transaction interleaving and 2 phase execution.

Transaction grouping. Whenever a transaction block enters the softcore, the softcore tries to add the transaction to the current batch as follows. It first checks the metadata in the catalogue to figure out how many GP/CP registers are required for the transaction. If there are enough registers remaining, the transaction joins the current batch with an exclusive range of GP/CP registers allocated. For the purpose, the softcore maintains the base GP/CP register addresses and renames the registers in the stored procedure instructions by adding the base register addresses. After that, the base register addresses are updated to indicate next available range, and the stored procedure is executed immediately. If the allocation fails, the current batch is closed and the new transaction is scheduled after the current batch commits.

Two-phase execution and interleaving. Figure 8 shows how interleaving is done. BionicDB executes transactions in two phases: 1) transaction logic and 2) commit/abort. Starting from the first transaction in a batch, the softcore executes a stored procedure code. When it reaches the end of a stored procedure, it saves the transaction context in a BRAM buffer, including program counter register, the base address of transaction block and register address range, and switches to the next transaction without waiting for outstanding DB instructions to complete.

When the first phase of a batch is finished by batch closure, it returns back to the first transaction and restores the context saved in the BRAM buffer. At this point, the program counter (PC) indicates the address of the first instruction of the commit handler. The commit handler then waits for all outstanding DB instruction to return. Depending on the results from them, the softcore continues to commit or jumps to the abort handler. Any exception, such as a DB instruction failure by CC or a voluntary abort, caught will trigger the abort handler. Then, the commit/abort handler performs a CC protocol to finish the transaction batch. After the second phase of a batch is finished, the softcore opens a new batch and executes the pending input transactions in the batch.

Discussion. With transaction interleaving, DB instructions across transactions can be overlapped. It is particularly beneficial for small transactions with insufficient intra-transaction index parallelism. Also, the overhead of transaction interleaving is marginal: transaction contexts are stored entirely in a context table on BRAMs, and a single switch takes 10 cycles to save current context and restore the next one from the context table.

The benefit of interleaving could be absorbed by data dependency within a transaction; data dependency is formed when a DB instruction requires an output from the previous DB instruction in the same transaction. Data dependency becomes a barrier that forces to wait for outstanding DB instructions to complete within a transaction, getting rid of the chance to overlap index requests in the next transactions. However, we found that current interleaving is still promising for certain workloads where transactions are small and data dependency-free (for example, YCSB transactions). To deal with heavy data dependency, it might be helpful to switch between transactions dynamically whenever desired, but current implementation does not support such dynamic scheduling.

4.6 On-chip Message-passing Channels

Partitioned databases, such as H-store and DORA, can minimize synchronization overhead across concurrent threads with single-threading policy; a partition is not thread-safe, but it is guaranteed to be accessed by a dedicated worker exclusively. Therefore, a partition worker cannot directly access a remote partition, instead, it should send a request message to a remote site. Then, a delegate worker on the remote partition processes the request on behalf of the initiator and returns a response message back. Hence, inter-worker message-passing is required when a transaction spans over multiple partitions.

To reduce the communication overhead and, thereby, accelerate cross-partition transactions, BionicDB provides on-chip message-passing channels. Unlike software message-passing that can include memory latency and thread synchronization during communication, request/response messages are directly exchanged between workers at on-chip speed without memory round-trips and thread synchronization. Despite the low clock frequency of FPGA, the communication latency is still low due to low-overhead message-passing protocol. The latency in exchanging a single pair of request and response messages takes 6 cycles in total (the latency could vary slightly depending on congestion). We believe that message-passing is a suitable communication semantic for single-threaded databases where data is strongly isolated between partition workers, while most CPUs have to conform the shared-memory semantic for backward compatibility.

Multisite transaction handling. Let us explain how a multisite transaction is processed using with the on-chip communication channels in detail. Each worker is assigned a communication link that consists of request and response channels. When the softcore decodes a DB instruction and finds out that the target partition is remote, it creates a request packet with the instruction and sends it through the request channel asynchronously. A request packet is piggybacked with a transaction timestamp for concurrency control and source/destination worker IDs for routing. At a remote site, a background unit monitors the request channel and catches an inbound request packet having a matching worker ID. In turn, the DB instruction is dispatched immediately to the remote index coprocessor as a background request. At this point, local (foreground) and remote (background) requests can be overlapped in the index coprocessor. But, it does not cause inconsistency because the index coprocessor is capable of dealing with pipeline anomalies and concurrency control. After completion of a background request, its result is sent back to the initiator through the response channel. The response message returned to the initiator is written back to the destination CP register asynchronously, and the initiator's softcore takes the result later when executing a RET instruction with the CP register.

Scaling on-chip message-passing. We discuss several scaling issues, leaving them to future work. First, the current topology of the on-chip communication is crossbar which does not scale. When scaling up BionicDB on datacenter-grade FPGAs that can fit tens or hundreds of BionicDB workers in a single chip (resource consumption will be provided in Section 5), a scalable on-chip communication topology, such as ring or tree, will be required. Also, BionicDB is currently a single-chip, single-node system.

Given that typical FPGA offerings have merely around tens of GBytes of on-board DRAM, it is vital to scale BionicDB across multiple FPGA nodes in a shared-nothing cluster like H-store [23]. Fortunately, some cloud FPGA services, such as AWS F1, support multi-chip, multi-node deployment[4]. Therefore, the message-passing channels should be diversified with additional connectivities for inter-node communication.

4.7 Concurrency Control

Currently, BionicDB employs a variant of the basic single-version timestamp CC [11]. Its strengths and weaknesses are well-known [47], but we use it for the sake of simplicity as CC is not the focus of this paper. A transaction is assigned a hardware timestamp value at the beginning. During transaction lifecycle, DB instructions issued by the transaction are packed with the transaction timestamp and passed to the index coprocessor. Then, visibility check is performed as follows by the index coprocessor against a matching tuple with the search key.

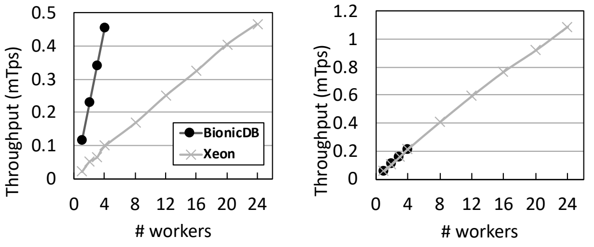
Visibility check. Each tuple is associated with latest read and write timestamps, and they are compared against the transaction timestamp when accessed. The read permission is granted on a tuple having a lower write time, and the write permission is granted on a tuple having a lower read time. If the transaction is the latest reader, read time of the tuple is updated with the transaction timestamp immediately. There are minor deviations from the original algorithm. First, any access to an uncommitted (dirty) tuple during runtime is blindly rejected without the care of the serial order and triggers transaction abort immediately. Also, read set is not buffered. If the second access to a previously visited tuple is denied by concurrent updates, the transaction should abort to ensure repeatable read.

If a DB instruction passes the visibility check, the address of the matching tuple with a “success" return code is written back to the CP register specified in the DB instruction. Otherwise, an error code is written. An UPDATE instruction only marks the dirty bit and returns without actual modification. With the tuple address returned, the softcore later performs in-place update after backing up the original tuple in UNDO log buffer in a transaction block. An REMOVE instruction returns after marking both dirty and tombstone bits.

Commit protocol. When a commit handler takes over the softcore after transaction logic execution, it collects the results from CP registers by executing RET instruction. If an error code is detected, the softcore jumps to the abort handler. After making sure successful execution, the softcore iterates write-set to cleanup dirty marks and overwrite their write time with the transaction's begin timestamp. If aborted, the abort handler restores the write-set from the UNDO logs in a transaction block and removes the dirty marks.

4.8 Logging and Recovery

Although logging and recovery are currently missing, we discuss a possible way to guarantee durability. BionicDB can adopt the VoltDB's command logging approach for recovery[29]. After executed by BionicDB, each transaction block contains the commit state and the commit timestamp of the transaction, preserving the input arguments. BionicDB can recover the database simply by re-executing the committed transaction blocks in the commit timestamp order. To this end, the host CPU should store the input transaction blocks that were processed in a durable storage before returning them to clients. After system failure, the host CPU should load the last checkpoint image and the persisted transaction blocks (command logs) from the disk. Then, the host CPU can replay the committed transaction blocks, ignoring the uncommitted ones. It must replay them in the commit timestamp order to ensure correct recovery. After recovery, the hardware clocks of BionicDB should be re-initialized to the latest commit timestamp, and the host CPU can signal BionicDB to resume transaction processing.



(a) YCSB-C (read-only).

(b) TPC-C NewOrder and Payment mix.

Figure 9: Comparison of overall performance of BionicDB to Silo on Xeon (4 chips).

5 EVALUATION

This section evaluates BionicDB for the following purposes.

- We show that BionicDB can provide high performance and substantial power saving in OLTP workloads, comparing to state-of-the-art SW system.
- We figure out how much speedup comes from each acceleration feature and which factors can limit the performance.

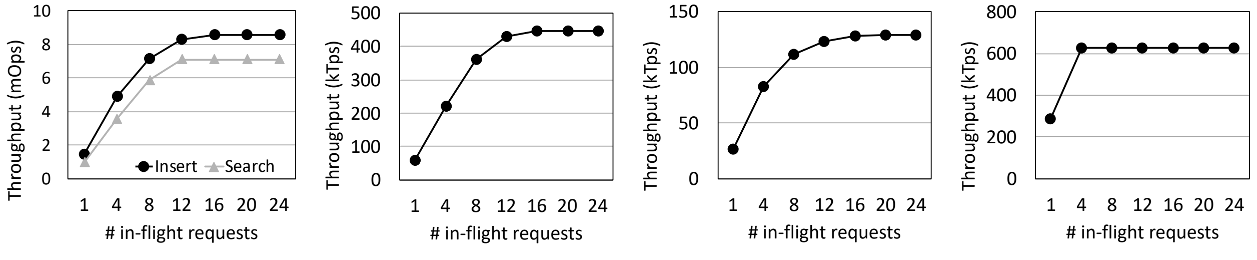
5.1 Experimental Setup

In all experiments, we fit the entire databases in DRAM. For BionicDB, we populated input transaction blocks in advance by host CPUs (ideally, remote clients should submit transaction blocks through network cards, and BionicDB should process them without the intervention of the host CPU). Unless stated otherwise, we report the aggregate throughput of four BionicDB workers.

5.2 Hardware

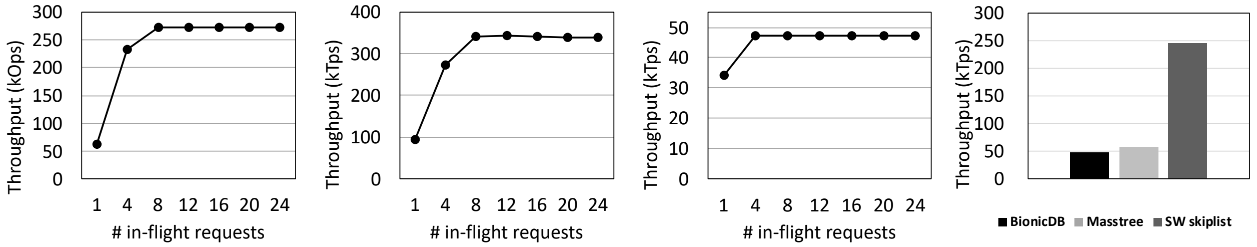
Xilinx Virtex5 LX330. We built BionicDB on a single Virtex5 LX330 FPGA chip which was manufactured with 65nm technology and released a decade ago. It contains merely 200K logic cells, allowing to fit only four BionicDB workers within a single chip. However, recent datacenter-grade FPGA chips containing millions of logic cells, such as Xilinx Virtex Ultrascale+ used in AWS F1 instances or Intel Arria 10, could accommodate tens or hundreds of BionicDB workers with a single chip [2, 5], providing ample thread-level parallelism. The clock frequency of BionicDB was set to 125MHz.

Intel Xeon E7 4807. To compare BionicDB to a software engine, we ran Silo[43] on 4 hexa-core Xeon chips (24 physical cores in total). Its clock frequency is 1.87GHz, and each core is assigned private 32KB L1I/D cache and 256KB L2 cache. L3 cache is 18MB and shared by all cores on a chip. In choosing a CPU for comparison, our focus was making sure the CPU is roughly in the same generation with Virtex5 for fairness. Xeon E7 4807 was released later (in 2011) than Virtex5 and manufactured with more advanced process technology (32nm).



(a) KeyValue **(b) YCSB-C (read-only)** **(c) TPC-C NewOrder** **(d) TPC-C Payment**

Figure 10: Throughput of hash index with varying index parallelism.



(a) Insert **(b) Point query** **(c) Scan (50 tuples)** **(d) vs. SW (scan)**

Figure 11: Throughput of skiplist with varying index parallelism and comparison of scan performance to SW.

5.3 Benchmarks

TPC-C. We ran a mix of the TPC-C NewOrder and Payment (50:50). Most transactions are local, but 1% of the NewOrder and 15% of the Payment transactions are cross-partition by default. We partitioned the database by Warehouse and replicated Item table which is read-only across partitions. The number of partitions was fixed to the number of workers. We modified the Payment transaction by enforcing to pick up a customer with *customer id* for both BionicDB and Silo.

YCSB. The YCSB transaction includes 16 independent DB accesses with no data dependency. The YCSB table schema consists of a 8B integer key and 1KB payload. We populated 300K records per partition, and the size of a partition is 300MB (the number of partitions is equal to the the number of workers). We ran the YCSB-C (read-only) and YCSB-E transactions. The YCSB-B (read-intensive) was ommitted due to similar results to YCSB-C. We modified the YCSB-E transaction to make it scan-only and fixed the scan range to be 50 records which is the average scan length with the default workload setting.

5.4 Overall Performance

We compared the overall performance of BionicDB and Silo in Figure 9. As explained in Section 5.1, we only presented BionicDB from 1 to 4 workers because of limited logic capacity of Virtex5 FPGA. In Figure 9a, we ran YCSB-C transactions. With the same number of workers, BionicDB outperformed Silo by 4.5x. Silo matched the throughput of 4 BionicDB workers with 24 cores enabled over 4 CPU chips. YCSB-C transaction exploited both intra-, inter-transaction parallelism provided by index pipelining and transaction interleaving thanks to sufficient index parallelism and the absence of data dependency that permits aggressive transaction interleaving.

We also ran a mix of the TPC-C NewOrder and Payment (50:50) and plotted in Figure 9b. Unlike the YCSB result, BionicDB achieved only comparable performance to Silo with the same number of workers because of insufficient index parallelism of Payment transaction (only 4 index lookups) and heavy data dependency that nearly eliminated the chance for transaction interleaving (in fact, the TPC-C transactions were executed almost in serial).

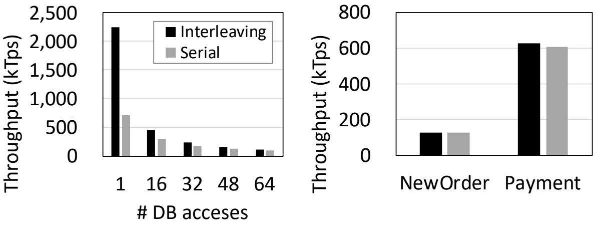
These results reveal both the promise and challenge of BionicDB at the same time. BionicDB can outperform state-of-the-art SW OLTP system when fully utilized, but it can be underutilized by insufficient index parallelism and heavy data dependency. However, the results confirm that BionicDB can provide still competitive performance even with much lower frequency (125MHz) and a limited microprocessor (no instruction-level parallelism and general-purpose cache memory), by taking a suitable acceleration approach for OLTP. From the next section, we evaluate each acceleration component to understand their impacts quantitatively.

5.5 Impact of Index Pipelining

We evaluated index pipelining in Figure 10 and Figure 11, We controlled the degree of coprocessor parallelism by changing the maximum number of in-flight DB requests over the index coprocessor. To focus on the index coprocessor, all experiments in this section run local transactions only.

Hash. In Figure 10a, we ran a non-transactional key-value workload to see the peak performance of the hash index. A single transaction repeated issuing 60 insert/search instructions in bulk for 20,000 times (in total 1.2M inserts and searches). The peak performance for insert and search reached 8.5Mops and 7Mops, respectively, and saturated between 12 and 16 index parallelism. This implies that there were 3 or 4 in-flight requests between pipeline stages in average.

We plotted the throughput of the YCSB-C (read-only) and the TPC-C NewOrder transactions in Figure 10b and Figure 10c. Both figures show the similar trend with the previous result in the



(a) YCSB-C (b) TPC-C

Figure 12: Transaction interleaving vs. serial execution.

KV workload. It proves that the transactions have sufficient intra-transaction parallelism. Figure 10d shows the performance of the TPC-C Payment transactions. Due to the limited index parallelism, the performance did not improve after 4 requests.

Skiplist. We instantiated 8-stage skiplist pipelines with an extra scanner module. The maximum height of a skiplist tower was set to 20. Figure 11a shows the performance of sequential loading. The pipeline was saturated at 8 in-flight requests. It increased sharply from 1 to 4 and modestly from 4 to 8 requests. The reason for lower pipeline parallelism than hash index is that skiplist pipeline stages contain multiple memory stalls during horizontal pointer chasing, leaving nearly no in-flight requests between stages, unlike the hash pipeline. Hence, the index parallelism was bound by the depth of pipeline. The modest curve from 4 to 8 in-flight requests is related to unbalanced dataflow over skiplist pipeline. Pipeline stages covering upper levels were less busy than lower levels as towers were sparser. Figure 11b shows the performance of point query. It shows similar trend with the previous result, but the throughput is higher because tower installation was skipped.

To measure scan performance, we ran modified YCSB-E transactions (Section 5.1) and presented the performance in Figure 11c. We can find that the pipelining efficiency was deteriorated. This is because a single scanner became a bottleneck in the pipeline. Thus, heavy scan loads should be distributed over multiple scanners for balanced pipelining. In Figure 11d, we compared the scan performance to Masstree and SW skiplist on the Xeon CPU introduced in Section 5.1. The number of workers was four across all indexes. Since its pipelining efficiency was largely eliminated, HW skiplist was slower than Masstree by 20% and SW skiplist by 5x. With extra FPGA resources, HW skiplist could be improved with deeper pipelining and multiple scanners (in this experiment, pipelining depth and the number of scanners were bound by current FPGA resource). To catch up with SW skiplist, at least 5 scanners would be required.

5.6 Impact of Transaction Interleaving

Figure 12 shows the performance comparison between transaction interleaving and serial execution. In this experiment, all transactions were local. In Figure 12a, we changed the the size of transaction footprint, or intra-transaction parallelism, by varying the number of DB instructions in a YCSB-C transaction. With a single record transaction, transaction interleaving was 3x faster than serial execution. In serial execution, the coprocessor was underutilized by small transactions. Whereas, transaction interleaving was able to overlap index requests across transactions, resulting in much higher utilization. As we increase the number of requests, the performance gap shrank. This is because coprocessor was utilized better in serial execution with more intra-transaction parallelism.

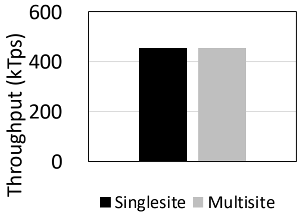


Figure 13: Single-site (100% local access) vs. multi-site transaction (75% remote, 25% local access).

Figure 12b illustrates the throughput of the TPC-C NewOrder and Payment transactions. In both transactions, there was no noticeable difference because heavy data dependency hindered transaction interleaving. The NewOrder transaction had enough intra-transaction index parallelism, but failed to exploit inter-transaction parallelism due to data dependency, leaving the coprocessor idle during commit phase. The Payment transaction's case was worse. It was not able to exploit both intra-, inter-transaction parallelism because of limited parallelism and data dependency combined, leading to significant underutilization. It is also evidenced by the modest difference in overall performance from the YCSB-C transaction that includes four times more index roundtrips.

5.7 Impact of On-chip Message-passing

Latency analysis. Table 3 compares the communication latency of on-chip message-passing against software message-passing. We assumed 20ns and 80ns for the latency of shared-cache and DDR3, respectively. Based on the estimates, we calculated the total communication latency for exchanging a single request/response pair that takes two iterations of message-passing, assuming that cache communication takes two cache reads on a modified-state cacheline, and DRAM communication takes two rounds of memory read and write. Despite the slow frequency (125MHz), the latency of on-chip message-passing is 48ns which is comparable to cache communication and much faster than DDR3 communication. If cache miss happens, the latency of software communication can rapidly increase. Also, we did not take synchronization cost into account, giving favor to software message-passing. In practice, software message-passing could suffer much higher communication latency in the presence of cache misses and thread contention.

Primitive		Latency (ns)	Total comm. delay (ns)
On-chip MP		24	48
Software MP	L3 cache	20	40
	DDR3	80	320

Table 3: Latencies of message-passing methods.

Throughput of cross-partition transactions. It is widely known that frequent multisite transactions in non-partitionable workloads can be a serious bottleneck in partitioning-based systems. We now evaluate the performance of multi-site transactions to see if the on-chip message-passing can accelerate them. We ran the cross-partition YCSB-C transactions with uniform random keys and plotted the throughput in Figure 13. In the cross-partition transaction, 75% of DB accesses are remote, and the rests are local accesses. As an ideal case, we also plotted the performance of local transactions that do not involve inter-worker communication at all. The result shows that on-chip, message-passing communication imposed negligible overhead, achieving almost same performance with the ideal case. In all other workloads, we observed the same result. This confirms that the message-passing, on-chip communication of BionicDB eliminated the overhead of communication and accelerated cross-partition transactions effectively.

5.8 Power Consumption and Resource Utilization

We estimated the power consumption of BionicDB on Virtex5 LX330 with Xilinx Power Estimator (XPE). The total power consumption was approximately 11.5W. The thermal design power (TDP) of a single Xeon E7 4807 is 95W, and the aggregate TDP of four chips is 380W.

Module	Flip-flops	Look-up tables	Block RAMs
Hash	12,932	14,504	24
Skiplist	27,300	35,968	36
Softcore	7,080	8,796	12
Catalogue	1,484	1,964	8
Communication	2,482	3,191	8
Memory arbiters	1,192	5,800	0
HC-2 modules	98,507	76,639	103
Virtex5 LX330 Total	207,360	207,360	288
Utilization	72%	70%	70%

Table 4: Resource utilization of BionicDB with 4 workers

Table 4 reports the resource utilization of BionicDB with four workers on a Virtex5 LX-330 chip. The entire hardware design consumed around 70% of FFs, LUTs and BRAMs. Almost half of the total logic cells were taken by HC-2's infrastructures, such as host interface, crossbar memory interconnects and a custom processor which were not used by BionicDB at all. Four BionicDB workers consumed approximately 70k LUTs and 53k FFs in total. Out of BionicDB resources, the skiplist index consumed almost 50%, and hash index consumed around 20%. The stored procedure execution modules, the softcore and catalogue, took only 15% thanks to the resource-efficient design choices.

6 RELATED WORK

Many existing database accelerators have focused on offloading SQL computation. They commonly exploit massive fined-grained parallelism for compute acceleration, avoiding instruction decoding overhead and memory wall. (application-specific functions are wired on datapath directly, and data stream flow over them) [18, 27].

Oracle SPARC M7 processor integrates in-memory database acceleration fabric (DAX), offloading filtering and de/compression [3]. Ibex[44] is a FPGA-based MySQL storage engine that offloads filtering and aggregation functions to FPGA. It provides the notion of near-data processing by placing the SQL accelerator between CPU and SSD. Bharat et al., suggested FPGA coprocessor for OLAP acceleration in in-memory HTAP system [41]. They offloaded filtering and data decompression to FPGA coprocessors on a PCIe board while the host CPU focus on transaction processing. The FPGA coprocessor directly accesses compressed memory-resident data, decompresses them, and performs filtering. To saturate PCIe bandwith, data scan tiles are replicated. Q100[45] is a dataflow hardware for SQL. ASIC tiles that perform SQL operators are provided, along with a custom instruction set to control data stream over the tiles. DoppioDB offloaded regular expression evaluation and analytic operators on Intel's Xeon-FPGA machine [33, 39]. Do et al. explored in-storage SQL processing inside flash memory SSD [15], but the system used embedded processors for acceleration without custom hardware. Kocberber et al. suggested Widx which is an on-chip hash indexing accelerator for OLAP workloads [26]. They identified hash index lookup as the main bottleneck due to poor memory-level parallelism in OLAP workloads and offloaded the it to on-chip acceleration fabric.

For stream acceleration, Glacier[30] implemented SQL circuits on an FPGA fabric, installed on datapath between network card and host CPU. Handshake join [42] is a highly parallel stream join algorithm with massive parallel processing resources such as FPGA or GPGPU. The key idea is fine-grained parallel join processing between two streams flowing from the opposite directions. A large number of tuple pairs are evaluated at once, exploiting massive computation parallelism. FQP[31] is a flexible stream query processor that can support changing query logic without FPGA reconfiguration. It implemented filtering and stream join operators.

For transaction processing, there have been a few studies with custom hardware approach. Cipherbase uses FPGA as a coprocessor for security, offloading expression evaluation, some index operations and en/decryption [9]. Also, there have been hardware key-value store systems with hash index [10, 13, 22, 46] and transactional graph processing on FPGA [28]. However, standalone transaction processing hardware is still missing in the landscape. Many existing SQL accelerators do not solve OLTP's main problems: memory stalls and communication. Low-end processor, such as ARM, could be a power efficient option, but it often sacrifices performance [40].

7 CONCLUSION

In this paper, we explored hardware specialization for OLTP and presented the design and implementation of BionicDB built on FPGA. We discussed index pipelining and transaction interleaving for index acceleration, and on-chip message-passing for faster communication in partitioned databases. Also, we argued that OLTP requires tightly integrated SW-HW architecture. The experimental results confirmed that transaction processing can be done at substantially lower power cost while providing competitive performance. Possible future directions include scaling up BionicDB on a modern FPGA chip and scaling out over multiple chips and nodes.

References

[1] 1992. TPC Transcation Processing Performance Council. http://www.tpc.org/default.asp

[2] 2014. Xilinx Ultrascale datasheet. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

[3] 2015. Oracle SPARC M7 processor. http://www.oracle.com/us/products/servers-storage/sparc-m7-processor-ds-2687041.pdf

[4] 2017. AWS EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/

[5] 2017. Intel Stratix 10 FPGA. https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html

[6] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a Modern Processor: Where Does Time Go?. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 266–277. http://dl.acm.org/citation.cfm?id=645925.671662

[7] Raja Appuswamy, Angelos C. Anadiotis, Danica Porobic, Mustafa K. Iman, and Anastasia Ailamaki. 2017. Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-contention Workloads. *Proc. VLDB Endow.* 11, 2 (Oct. 2017), 121–134. https://doi.org/10.14778/3149193.3149194

[8] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The Case For Heterogeneous HTAP. (2017).

[9] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security With Cipherbase, In 6th Biennial Conference on Innovative Data Systems Research (CIDR'13). https://www.microsoft.com/en-us/research/publication/orthogonal-security-with-cipherbase/

[10] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A High Concurrency Key-Value Store with Integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 251–266. https://doi.org/10.1145/3035918.3064030

[11] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (June 1981), 185–221. https://doi.org/10.1145/356842.356846

[12] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. 2013. Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores. In *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304 (OPODIS 2013)*. Springer-Verlag New York, Inc., New York, NY, USA, 83–97. https://doi.org/10.1007/978-3-319-03850-6_7

[13] Sai Rahul Chalamalasetti, Kevin Lim, Mitch Wright, Alvin AuYoung, Parthasarathy Ranganathan, and Martin Margala. 2013. An FPGA Memcached Appliance. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '13)*. ACM, New York, NY, USA, 245–254. https://doi.org/10.1145/2435264.2435306

[14] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[15] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1221–1230. https://doi.org/10.1145/2463676.2465295

[16] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *ISCA*.

[17] Brian Gold, Anastassia Ailamaki, Larry Huston, and Babak Falsafi. 2005. Accelerating Database Operators Using a Network Processor. In *Proceedings of the 1st International Workshop on Data Management on New Hardware (DaMoN '05)*. ACM, New York, NY, USA, Article 1. https://doi.org/10.1145/1114252.1114260

[18] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding Sources of Inefficiency in General-purpose Chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 37–47. https://doi.org/10.1145/1815961.1815968

[19] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastasia Ailamaki, and Babak Falsafi. 2007. Database Servers on Chip Multiprocessors: Limitations and Opportunities. (01 2007).

[20] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP Through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 981–992. https://doi.org/10.1145/1376616.1376713

[21] Mark D. Hill and Michael R. Marty. 2008. Amdahl's Law in the Multicore Era. *Computer* 41 (2008), 33–38. Issue 7.

[22] Z. Istvan, G. Alonso, M. Blott, and K. Vissers. 2013. A flexible hash table design for 10GBPS key-value stores on FPGAS. In *2013 23rd International Conference on Field programmable Logic and Applications*. 1–8.

[23] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 603–614. https://doi.org/10.1145/1807167.1807233

[24] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1675–1687.

[25] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous Memory Access Chaining. *Proc. VLDB Endow.* 9, 4 (Dec. 2015), 252–263. https://doi.org/10.14778/2856318.2856321

[26] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 468–479. https://doi.org/10.1145/2540708.2540748

[27] H. T. Kung. 1982. Why systolic architectures? *Computer* 15, 1 (Jan 1982), 37–46. https://doi.org/10.1109/MC.1982.1653825

[28] Xiaoyu Ma, Dan Zhang, and Derek Chiou. 2017. FPGA-Accelerated Transactional Execution of Graph Workloads. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 227–236. https://doi.org/10.1145/3020078.3021743

[29] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. 2014. Rethinking main memory OLTP recovery. In *2014 IEEE 30th International Conference on Data Engineering*. 604–615. https://doi.org/10.1109/ICDE.2014.6816685

[30] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Streams on Wires: A Query Compiler for FPGAs. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 229–240. https://doi.org/10.14778/1687627.1687654

[31] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2013. Flexible Query Processor on FPGAs. *Proc. VLDB Endow.* 6, 12 (Aug. 2013), 1310–1313. https://doi.org/10.14778/2536274.2536303

[32] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. 2017. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/3020078.3021740

[33] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta. 2011. A Reconfigurable Computing System Based on a Cache-Coherent Fabric. In *2011 International Conference on Reconfigurable Computing and FPGAs*. 80–85. https://doi.org/10.1109/ReConFig.2011.4

[34] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric Chung. 2015. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware. https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/

[35] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-oriented Transaction Execution. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 928–939. https://doi.org/10.14778/1920841.1920959

[36] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. 2011. PLP: Page Latch-free Shared-everything OLTP. *Proc. VLDB Endow.* 4, 10 (July 2011), 610–621. https://doi.org/10.14778/2021017.2021019

[37] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (June 1990), 668–676. https://doi.org/10.1145/78973.78977

[38] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2015. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *IEEE Micro* 35 (May 2015), 10–22.

[39] D. Sidler, M. Owaida, Z. IstvÃąn, K. Kara, and G. Alonso. 2017. doppioDB: A hardware accelerated database. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–1. https://doi.org/10.23919/FPL.2017.8056864

[40] Utku Sirin, Raja Appuswamy, and Anastasia Ailamaki. 2016. OLTP on a Server-grade ARM: Power, Throughput and Latency Comparison. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN '16)*. ACM, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/2933349.2933359

[41] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. 2012. Database Analytics Acceleration Using FPGAs. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, New York, NY, USA, 411–420. https://doi.org/10.1145/2370816.2370874

[42] Jens Teubner and Rene Mueller. 2011. How Soccer Players Would Do Stream Joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. ACM, New York, NY, USA, 625–636. https://doi.org/10.1145/1989323.1989389

[43] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[44] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading. *Proc. VLDB Endow.* 7, 11 (July 2014), 963–974. https://doi.org/10.14778/2732967.2732972

[45] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: The Architecture and Design of a Database Processing Unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/2541940.2541961

[46] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. 2016. Bluecache: A Scalable Distributed Flash-based Key-value Store. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 301–312. https://doi.org/10.14778/3025111.3025113

[47] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220. https://doi.org/10.14778/2735508.2735511

Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management

Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck,
Matthias Uflacker, Hasso Plattner
Hasso Plattner Institute
Potsdam, Germany
firstname.lastname@hpi.de

ABSTRACT

Research in data management profits when the performance evaluation is based not only on individual components in isolation, but uses an actual DBMS end-to-end. Facilitating the integration and benchmarking of new concepts within a DBMS requires a simple setup process, well-documented code, and the possibility to execute both standard and custom benchmarks without tedious preparation. Fulfilling these requirements also makes it easy to reproduce the results later on.

The relational open-source database Hyrise (VLDB, 2010) was presented to make the case for hybrid row- and column-format data storage. Since then, it has evolved from being a single-purpose research DBMS towards becoming a platform for various projects, including research in the areas of indexing, data partitioning, and non-volatile memory. With a growing diversity of topics, we have found that the original code base grew to a point where new experimentation became unnecessarily difficult. Over the last two years, we have re-written Hyrise from scratch and built an extensible multi-purpose research DBMS that can serve as an easy-to-extend platform for a variety of experiments and prototyping in database research.

In this paper, we discuss how our learnings from the previous version of Hyrise have influenced our re-write. We describe the new architecture of Hyrise and highlight the main components. Afterwards, we show how our extensible plugin architecture facilitates research on diverse DBMS-related aspects without compromising the architectural tidiness of the code. In a first performance evaluation, we show that the execution time of most TPC-H queries is competitive to that of other research databases.

1 INTRODUCTION

Hyrise was first presented in 2010 [19] to introduce the concept of hybrid row- and column-based data layouts for in-memory databases. Since then, several other research efforts have used Hyrise as a basis for orthogonal research topics. This includes work on data tiering [7], secondary indexes [16], multi-version concurrency control [42], different replication schemes [43], and non-volatile memories for instant database recovery [44].

Over the years, the uncontrolled growth of code and functionality has become an impediment for future experiments. We have identified four major factors leading to this situation:

- Data layout abstractions were resolved at runtime and incurred costs that sometimes had a disproportional overhead.
- Prototypical components have been implemented to work in isolation, but did not interact well with other components.
- The lack of SQL support required query plans to be written by hand and made executing standard benchmarks tedious.
- Accumulated technical debt made it difficult to understand the code base and to integrate new features.

For these reasons, we have completely re-written Hyrise and incorporated the lessons learned. We redesigned the architecture to provide a stable and easy to use basis for holistic evaluations of new data management concepts. Hyrise now allows researchers to embed new concepts in a proper DBMS and evaluate performance end to end, instead of implementing and benchmarking them in isolation. At the same time, we allow most components to be selectively enabled or disabled. This way, researchers can exclude unrelated components and perform isolated measurements. For example, when developing a new join implementation, they can bypass the network layer or disable concurrency control.

In this paper, we describe the new architecture of Hyrise and how our prior learnings have led to a maintainable and comprehensible database for researching concepts in relational in-memory data management (Section 2). Furthermore, we present a plugin concept that allows testing different optimizations without having to modify the core DBMS (Section 3). We compare Hyrise to other database engines, show which approaches are similar, and highlight key differences (Section 4). Finally, we evaluate the new version and show that its performance is competitive (Section 5).

1.1 Motivation and Lessons Learned

The redesign of Hyrise reflects our past experiences in developing, maintaining, and using a DBMS for research purposes. We motivate three important design decisions.

Decoupling of Operators and Storage Layouts. The previous version of Hyrise was designed with a high level of flexibility in the storage layout model: each table could consist of an arbitrary number of containers, which could either hold data (in uncompressed or compressed, mutable or immutable forms) or other containers with varying horizontal and vertical spans. In consequence, each operator had to be implemented in a way where it could deal with all possible combinations of storage containers. This made the process of adding new operators cumbersome and led to a system where some operators made undocumented assumptions about the data layout (e.g., that all partitions used the same encoding type). Instead of relying on operators to properly process data structures with varying memory layouts, Hyrise now follows an iterator-based approach. By accessing data through iterators, the implementation of new operators is decoupled from the implementation of new data storage concepts without compromising the flexibility. Operators can implement custom specializations for specific iterators, but execution falls back to the default iterator if no implementation exists. The iterator abstraction is explained in Section 2.3.

10.5441/002/edbt.2019.28

Benchmarking. Just as modern software development processes require that fundamental development steps, such as setting up the environment, building the code, and running tests should require only one step each, we believe that the same is true for benchmarks. This might seem obvious, but in our experience, also with other databases, both productive and research, this is almost never the case. In Hyrise, benchmarks are now single binaries that generate their data, run the queries, and print the results. As of writing, the TPC-H benchmark is included in the code base; the alternative data generator JCC-H [8], as well as the TPC-C and Join-Order [30] benchmarks are work-in-progress. Custom benchmarks can be easily added by creating table and query files, which are automatically executed by a generic benchmark runner (cf. Section 2.10). To facilitate reproducibility, all benchmark results contain the parameters relevant to their execution, including the Git commit hash, information about the utilized scheduler, thread count, and more. Benchmarks are executed by the CI process for each commit on the master branch to aid in the identification of performance regressions.

Memory Management & Metaprogramming. When the development of Hyrise started 10 years ago, C++11 was not yet finalized. This meant that memory management had to be done manually using *new/malloc* and *delete/free*, resulting in spurious segmentation faults and considerable debugging efforts. In the new version, Hyrise almost exclusively uses shared or unique smart pointers, which guarantee that objects are only deleted right after there is no remaining pointer to them. The overhead of reference counting for shared pointers seems to be well justified by the noticeable reduction in time spent on debugging memory management issues. Only when these pointers become an actual bottleneck, we look into whether the memory management in that particular component can be made more explicit.

Hyrise employs template metaprogramming to decouple the supported data types, storage layers, encoding types, and operators while at the same time preserving a high degree of compile-time type safety. We use `Boost.Hana` to automatically generate code for the different supported data types and for the resolution of our iterators. Mostly because of this, but also because of other C++17 features that are used all over the code base, Hyrise can only be compiled with current versions of GCC and Clang.

1.2 Building a Database with Students

We believe that the best way to learn about database systems is to program them yourself. Therefore, in addition to our PhD research projects, we use Hyrise in a graduate-level database class in which students develop, integrate, and test new features, such as additional join implementations, optimizer rules, or index structures. Not only does this hands-on experience help their understanding of databases and improve their programming skills; it also raises their interest to research database topics in greater depth, for example as part of their Master's theses. As a result, a considerable fraction of the Hyrise code has been developed together with students as part of their class assignments or thesis work.

Developing a database with students also entails one of the main challenges in the process, which is a relatively short developer turnover time. Even those Master's students who decide to specialize in the field rarely spend more than two years on the project. We address this challenge with strict code reviews, a high degree of test coverage (currently at >85%), and by encouraging new students to highlight existing components that are difficult to understand. On the technical side, we use automatic formatting (clang-format), multiple linting tools (cpplint and clang-tidy), sanitizers (ASan, UBSan, TSan, and Memcheck), and enforce most compiler warnings (`-Wall -Wextra -pedantic -Werror`).

The main criterion for building a maintainable research DBMS is that students without prior database knowledge can be brought up to speed and contribute code in six weeks. The fact that we were able to prove this repeatedly in our database seminars shows us that we are on the right track.

2 SYSTEM ARCHITECTURE

In this section, we give an overview of the new architecture of Hyrise. In places where it improves clarity, we refer to the original Hyrise architecture as *Hyrise1* and to that of the rewritten DBMS *Hyrise2*. After a general description of the high-level architecture, which follows the visualization in Figure 1 and focuses on query execution, we describe major components in their respective subsections, which are also given in the figure.

We decided to make as much functionality and as many components optional as reasonably possible. This decision is based on a learning from Hyrise1, where we found it difficult to isolate the root of performance issues because of the number of involved components. In Hyrise2, even core concepts, such as optimization, concurrency control, or scheduling, can be disabled. Without an optimizer, queries get executed close to how they are defined in SQL; for example, joins are only identified if `JOIN ... ON ...` is used. If MVCC is turned off, all tables are read-only, do not store information with regards to transactions or concurrency, and validation operators are not inserted into the query plan. If the scheduler is turned off, tasks are immediately executed in the same thread (while still guaranteeing progress). Similarly, Hyrise2 can be benchmarked with and without the just-in-time compiler, the network interface, or logging.

2.1 General Overview

Figure 1 shows the core components of Hyrise. We discuss them by following the process of answering a user-provided SQL query. Users have three options to submit queries to Hyrise. As a first option, we provide a command line interface, which can not only be used to submit queries, but also offers convenience functions for generating TPC-C or TPC-H benchmark tables, visualizing query plans, and toggling optional Hyrise components. As a second option of interacting with the database, Hyrise has an integrated TCP/IP server implementing the wire protocol of PostgreSQL. Users can send queries using PostgreSQL's interactive terminal *psql* or existing drivers for PostgreSQL. More details on how this interface is designed and implemented can be found in Section 2.5. Lastly, the third option is the SQL-C++ interface, which is used by our benchmark binaries (cf. Section 2.10) and also enables hand-written optimization of query plans. Currently, the benchmark binaries use the SQL-C++ interface. We are working on a benchmark implementation that utilizes the network interface.

All of the three entry points hand the user's SQL string to Hyrise's SQL Pipeline (cf. Section 2.6), which consists of multiple steps that transform the provided SQL string to an efficient query plan. The *SQL Parser* translates the SQL string to an abstract syntax tree expressed as C++ data structures. Next, the *SQL to LQP Translator* changes the abstraction level by creating a logical query plan (*LQP*), which is a directed acyclic graph (DAG) whose nodes loosely resemble the operations of the relational algebra. Finally, the *Optimizer* applies a series of rules to the LQP, which

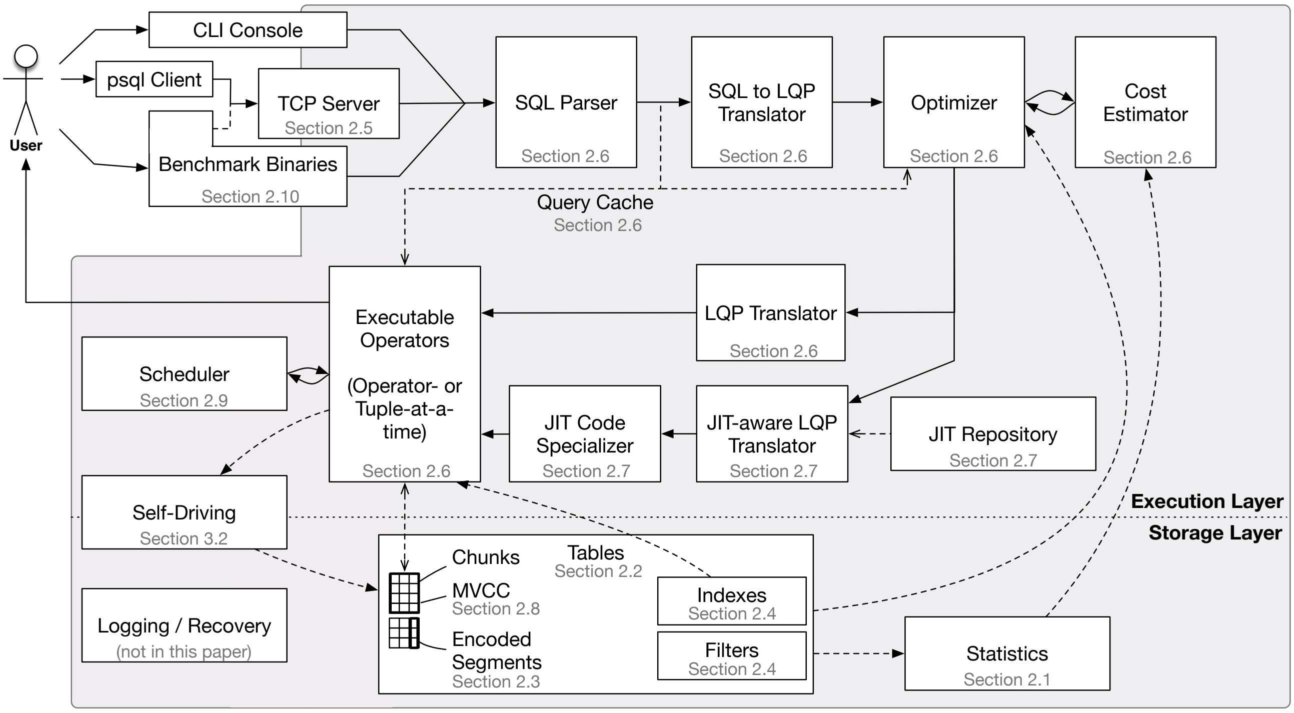


Figure 1: Hyrise Architecture Overview.

transform it into a more efficient, semantically equivalent version of the query. Some rules work by simply applying logical properties of the query plan (e.g., single-table predicates should almost always be pushed below joins), while others utilize information about the referenced tables that is only known at runtime. This information is collected from auxiliary data structures, such as general *statistics*, *indexes*, and *filters* (cf. Section 2.4). At the time of writing, statistics rely on histograms (equal height, equal width, equal distinct count) and other simpler metrics, e.g., the number of distinct values. Filters are probabilistic data structures that support approximate membership queries and allow the pruning of chunks (i.e., logically ruling out the necessity to access a chunk for a given predicate), whereas indexes return qualifying positions for a certain predicate directly without scanning through the data.

Hyrise implements a query plan cache, which can store both logical and physical query plans. Thereby, translation and optimization can be skipped to avoid doing these steps repeatedly for the same queries. While the cache does not yet auto-parameterize incoming queries, users can use prepared statements for queries with varying parameters. Both implicitly cached queries and prepared statements use the same caching data structures and shortcuts shown in the figure.

After optimization, the LQP is passed to the *LQP Translator*, which translates logical operators (such as predicates, joins, and sorts) to physical operators. Physical operators are concrete implementations of the logical concepts and more than one implementation might exist for a logical operator. For example, we implement joins as either sort-merge joins (cf. [2]), hash joins (cf. [4]), or nested-loop joins. Based on the optimizer's hints, one of these implementations is chosen for each logical operator. The result of the LQP Translator is another DAG, which we call Physical Query Plan (*PQP*). If the just-in-time compiler (cf. Section 2.7) is enabled, the LQP is not passed to the LQP translator, but to a *JIT-aware LQP Translator*, which creates JIT operators whose code is specialized using runtime information.

In both cases, the resulting PQP is then handed to the scheduler, which takes care of executing the translated operators. Once all operators have been executed, the resulting table is returned to the user.

Having looked at the entities that make executing a query possible, we now go over the data structures that are accessed for this. During execution, the primary table data as well as secondary data structures such as *indexes* are accessed by the operators. Tables are horizontally partitioned into *chunks*. Within a chunk, we have vertical partitions called *segments* where the segments across all chunks constitute the columns. Data within a segment might be encoded (cf. Section 2.3), for example using dictionary or run-length encoding. Additionally, chunks hold the data needed for multi-version concurrency control (MVCC, cf. Section 2.8). For more information on our storage layout refer to Section 2.2.

There are two more components shown in Figure 1: Self-Driving and Logging / Recovery. We envision future database systems to be self-driving [26], meaning that they autonomously adjust their configurations. Therefore, a *self-driving* component that assesses the database's current workload and tunes the configuration accordingly is part of Hyrise as well. This component is explained in more detail in Section 3.2. *Logging and recovery* are currently work-in-progress and are not described in this paper.

2.2 Storage Layout

Hyrise1 offers hybrid layouts, providing a maximum of flexibility with regards to storage layouts. This flexibility causes increased system complexity as well as runtime overhead by introduced abstractions. While the underlying system architecture of Hyrise2 supports hybrid layouts, we focus on columnar-oriented data for now. Figure 2 depicts the storage layout for an example table.

Storage layouts for HTAP database must support efficient read and write operations. This is often achieved by separating the data into read- and write-optimized partitions. Data is always added to write-optimized partitions. Update and delete operations invalidate entries in read-optimized partitions. From time to time, data has to be moved from write- to read-optimized partitions.

This transformation may happen by merging data of write-optimized partitions into read-optimized partitions [15, 27]. Merging may require re-encoding of already compressed data. Furthermore, the merge algorithm introduces implementation-specific complexity. For example, modifications to currently merged data have to be handled. In addition, the same data is encoded repeatedly during consecutive merge processes. Last, merge costs increase with the size of involved partitions.

To avoid a merge process in Hyrise2, tables are implicitly divided into *Chunks*, horizontal partitions of a certain size. Optimal chunk sizes are both data- and workload-dependent. A self-driving database system would decide on these autonomously. So far, our experiments showed suitable sizes to be between roughly fifty thousand and a few million records. The optimal chunk size is largely independent of the width of the table, both in terms of data sizes and number of columns in our default setup, where a column-based layout and dictionary encoding are used.

There are two types of chunks, mutable and immutable chunks. Initially, chunks are mutable and append-only containers. Data is added in a plain, unencoded fashion. When a chunk's capacity is reached it becomes immutable. Once this happens, encodings (cf. Section 2.3) can be asynchronously applied. Chunks encapsulate fractions of all of the table's columns, so-called segments.

There are a couple of advantages of the chunk-based approach. First, by implicitly partitioning the data, both multiprocessing (one core processes one chunk) and data placement, e.g., in NUMA environments, are simplified. Chunks can easily be distributed over multiple NUMA nodes, thereby leveraging multiple memory busses and CPUs for simultaneous processing.

Furthermore, auxiliary data structures like indexes and filters can be created on a per-chunk basis. Thus, these data structures are only created for those chunks where they yield a certain benefit. It also offers the flexibility to create different structures, for example, different index types for different chunks. The same can be applied to encodings: Some segments of a column might stay unencoded, others dictionary-encoded, and further segments run length-encoded.

Chunks are implicitly prunable entities. Thereby, in some cases, they can be excluded early from query processing without having to process the contained data. This can be achieved by using approximate membership query properties of filters (cf. Section 2.4) or characteristics of certain encoding types.

Partitioning the data into chunks has some drawbacks that need to be mitigated. First, it introduces the memory cost of storing per-chunk metadata. If, however, chunk sizes are chosen to be hundreds of thousands or millions of rows, this is not an issue as we show in Section 5.2. Second, for some encoding types, chunks may introduce redundant storage of information. For example, chunks encoded using the dictionary encoding store values that occur in all chunks over and over again in every chunk-local dictionary. On the other hand, if the overlap of values across chunks is low, the size of the dictionary can be kept low, and the number of needed bits for the attribute vector is reduced. Thus, choosing the optimal chunk size is a tradeoff between memory overhead and flexibility.

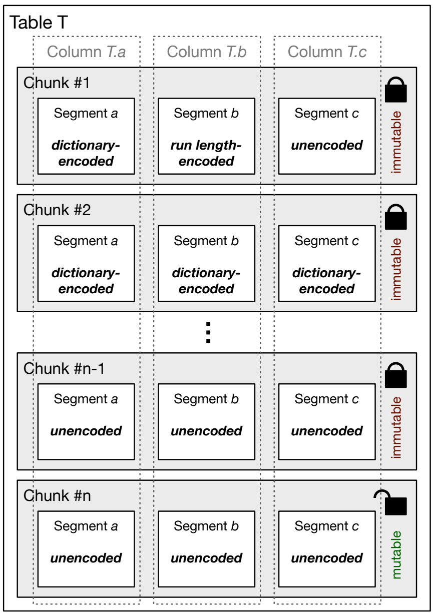


Figure 2: Depiction of Hyrise's storage layout for an exemplary table T with *n* chunks and three attributes.

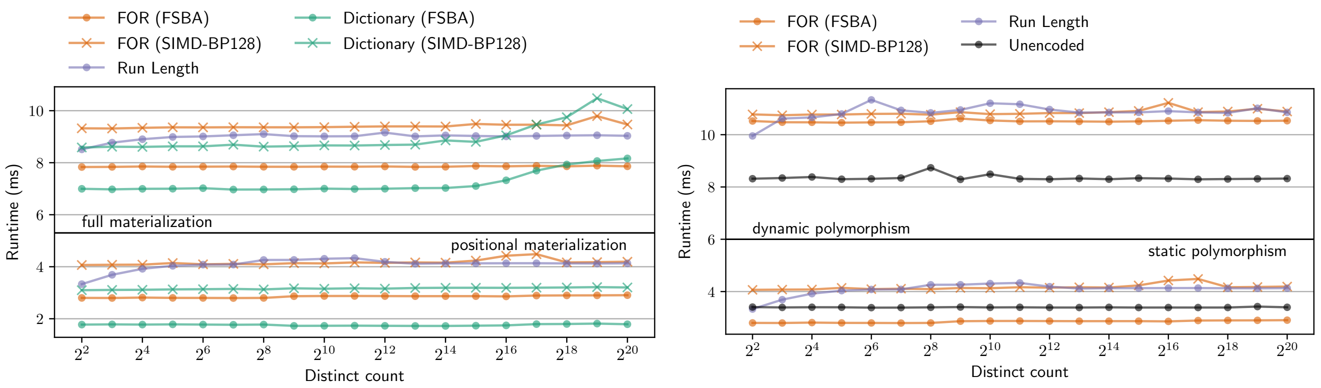
2.3 Segment Encoding

To our best knowledge, all modern columnar and memory-resident databases employ some form of column encoding. This is to (i) compress data and reduce memory consumption, (ii) better utilize the available memory bandwidth by increasing the entropy, and (iii) increase performance since operations on integer-encoded columns can be vectorized and processed by modern CPUs more efficiently. This effect is even stronger when relational operators can operate on encoded data without prior decoding.

Hyrise supports both logical (i.e., mapping input data to an integer representation) and physical (i.e., further compressing integer codes) encoding schemes (cf. [13]). The implemented logical schemes include frame of reference, run length, and order-preserving dictionary encoding. The physical ones include fixed-size byte alignment and SIMD-BP128 for null suppression. Logical and physical encoding schemes can be arbitrarily combined so that existing logical schemes can profit from a new physical encoding without modification.

Hyrise1 includes several encoding and compression schemes, but as mentioned above, no abstraction layer separated the data layout and the execution engine. This caused maintainability and performance issues and led us to formulate the following requirements for an encoding framework in Hyrise2:

- The encoding framework should be an abstraction layer where operators do not need to be implemented for each added encoding type.
- Still, implementing specialized access methods for certain encodings should be possible. For example, scans on dictionary-encoded columns should search for the integer value id, without having to decompress the data.
- Performance should be on par with manually optimized encoding schemes. This means that the compiler should be able to statically resolve the abstractions without having to resort to virtual method calls in hot loops.



(a) Full vs. positional accessing.

(b) Effect of polymorphism.

Figure 3: Runtime comparison for an aggregation accessing 25% of 1M integer values. Left: impact of decoding the full vector upfront vs. positional decoding. Right: impact of static polymorphism via C++ templating vs. dynamic polymorphism as done in the previous version of Hyrise.

- Abstractions should be analyzable by the compiler to the point where auto-vectorization (i.e., the automatic generation of SIMD code) is possible.
- The framework should seamlessly cooperate with our long-term plans on just-in-time query plan compilation.
- Instead of defining encodings per column, it should be possible to have different encodings in different segments.

We fulfill these requirements by heavily relying on static polymorphism, i.e., C++ templates. For each implementation of an encoding type, an *iterable* is implemented. These iterables inherit from a common base class using the curiously recurring template pattern (CRTP). The base class provides the `with_iterators` interface to operators, but instead of implementing this method using virtual inheritance, the CRTP is used to statically call a protected `_on_with_iterators` method. Operators pass a functor (i.e., a lambda or closure) to `with_iterator` as shown in Listing 1. Optionally, `with_iterators` takes a position list, which is used to selectively iterate over the values that, for example, are the result of a previous scan operation.

```
segment_iterable.with_iterators(
  [&](auto left_it, auto left_end) {
    for (; left_it != left_end; ++left_it) {
      const auto left = *left_it;

      if (!left.is_null() && predicate(left.value())) {
        matches.emplace_back(left.chunk_offset());
      }}});
```

Listing 1: Implementation of a vectorizable[1] scan operator using the iterators provided by the encoding framework

Not only the iterators, but also the functors (in the example, the predicate) are resolved at compile time. This allows us to define an adaptable and flexible encoding framework that avoids virtual method calls. Using iterators allows us to hide implementation details of encoded columns, which eases the maintainability of operators. In case query compilation is used, the iterators also provide efficient accesses for tuple-at-a-time processing without incurring virtual method calls. A downside of this approach is that the number of template instantiations grows exponentially. For a scan on a single column, we instantiate $|DataTypes| * |EncodingSchemes| * |Comparators|$ templates, resulting in a compile time of up to five minutes for the most complex operators. To prevent that cost from slowing down our development, the static resolution only takes place for *Release* builds; *Debug* builds use conventional virtual method calls.

The same iterator model can also be used to implement hybrid data layouts. A row-oriented segment type can provide iterators for each included attribute. Because the iterators are resolved at compile-time, accesses to attributes within one tuple would result in contiguous memory accesses.

Figure 3 shows two micro-benchmarks evaluating the performance of our encoding schemes and the encoding framework. As Hyrise is optimized for HTAP workloads, which include frequent positional accesses, random access iterators play an important role. Figure 3a shows the overhead of decoding the encoded vector beforehand (cf. [25]) compared to using positional random access iterators. For most encodings, positional accesses are 2−3× faster, even when decoding large position lists. For typical OLTP queries with short position lists, the advantage is even more pronounced.

The performance advantage of static polymorphism over dynamic polymorphism, i.e., virtual method calls, is shown in Figure 3b. In this benchmark, we aggregated a set of randomly chosen positions (25% of 1M integer values). No matter the encoding type, the cost of static polymorphism is significantly lower, with the biggest improvement being a factor of 3×.

2.4 Indexes and Filters

During execution, two types of secondary data structures are used to reduce the amount of data accessed: (i) secondary indexes and (ii) filters, which are lightweight probabilistic data structures for chunk pruning. Moreover, Hyrise uses these data structures during query optimization for cardinality estimation.

Secondary Indexes. Indexes in Hyrise yield qualifying positions for one or more predicate(s). Three secondary index structures are implemented: (i) adaptive radix trees (ART, cf. [31]), (ii) B-trees[2], and (iii) the group-key index [16]. The group-key index has been developed particularly for Hyrise. It builds on compressed position lists and exploits order-preserving dictionaries.

[1] To enable the compiler's auto-vectorization, it is helpful to first iterate over a constant number of rows at a time. This is not shown in the example for the purpose of brevity.

[2] Google C++ B-tree: https://code.google.com/archive/p/cpp-btree/

Filters. Filters are space-efficient (i.e., significantly smaller than a secondary index) auxiliary data structures, which allow the pruning of chunks (similar to partition elimination) for a given predicate. Hyrise supports min-max filters, counting quotient filters [37], and pruning-optimized histograms, which are comparable to adaptive range filters [3]. The latter two are not only capable of pruning but also support selectivity estimation.

Both indexes and filters are created on a per-chunk basis on immutable chunks and not globally for the whole table so that no maintenance cost is caused by inserts, updates, or deletes. This simplifies the code base and avoids computational overhead in forms of logging or heavy updates of large data structures. However, it is conceptually possible to add, e.g., a B-tree index to mutable chunks when required in OLTP scenarios. As filters and indexes are chunk-local, they can be selectively created for chunks where the estimated performance improvement outweighs the necessary memory space.

An important difference to previous work on filters is that, in Hyrise, they are integrated with the query optimizer, instead of being only used in the execution phase. Doing so enables optimizations that can only be used at query planning time: First, chunk pruning can be propagated through conjunctive predicate chains down to the plan node that initially represents the input table (cf. [6]). That plan node is configured to skip chunks that would later be excluded by one of the predicates. As a result, the number of accessed rows is reduced from the start and not only at the location of the respective predicate. Second, in case chunk pruning has a significant impact on the selectivity of a predicate, this knowledge can be exploited for operator-reordering, which would not be possible when pruning is done later in the execution phase. Similarly, we plan to use indexes not only in the execution phase but also for estimating cardinalities (cf. [32]).

2.5 Networking

Hyrise implements the wire protocol of PostgreSQL [40]. Reusing existing wire protocols is common [41] for new systems for several reasons. First, existing clients and drivers can be reused. This is advantageous for database products as well as for research platforms as it offers the possibility to access Hyrise from many programming languages and makes it accessible to many users. Second, tools such as *Wireshark* can be used to investigate how the sent and received PostgreSQL messages are encapsulated in network packets. As Hyrise is a research platform, we only implement the features needed for receiving SQL queries and returning results, but do not implement features such as user authentication or SSL. This keeps our implementation lean. The wire protocol uses TCP/IP, and our server is implemented using the asynchronous network features of *Boost.Asio*.

2.6 SQL

Figure 4 depicts the different steps in our SQL Pipeline. A dedicated *SQLPipeline* class is the main entry point to everything related to query execution. It takes an SQL string as a parameter and returns one or more tables. Optionally, all intermediary artifacts can be inspected by the developer in their text or graph forms. This was designed based on our experience with other databases where, in some cases, it is difficult to even understand the path that an SQL query takes through the system and which steps are involved. In the following paragraphs, we describe the steps of the SQL pipeline.

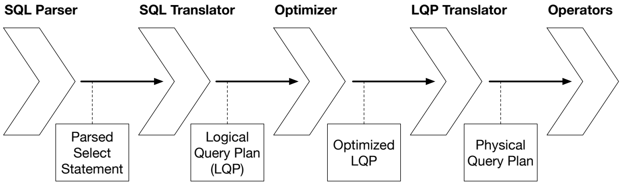


Figure 4: The different steps of the SQL Pipeline, leading from an SQL string to executable operators

Parsing. The SQL parser transforms the SQL query string into data structures that can be accessed and modified programmatically. When we started to add SQL support to Hyrise, we found no easy-to-use component that would transform an SQL string into an Abstract Syntax Tree (AST) that uses C(++) structs. While open-source databases come with their own parser (e.g., PostgreSQL), we found these to be too interwoven with the rest of the database for our purposes. Thus, we built a standalone C++ SQL parser based on Flex and Bison and released it as open source software[3].

SQL-to-LQP Translation. Having parsed the SQL query string into an Abstract Syntax Tree, which still resembles the structure of the SQL Query, we now translate the AST into a *Logical Query Plan (LQP)*, which is based on the relational algebra. Each edge in the LQP represents a table, either a user-defined table that is stored in the central *Storage Manager* or an intermediary table generated by the previous operator. It could also be a user-defined SQL view, which we have stored as its LQP and can embed into the query plan at this point. Operators do not need to perform expensive materializations of intermediary results, but can also pass positional references to the next operator. Having these references avoids expensive materializations. These operators are represented as nodes, which hold information about their input relations and attributes and parameters for their execution. LQP nodes are, however, not executable operators but only form the basis for logical query optimization. They get translated into physical operators only after all optimization steps have been performed. An example of a Logical Query Plan is shown in Figure 5.

Most LQP nodes, such as Predicates, Joins, or Aggregations, are one-to-one representations of their equivalent in the relational algebra. One node type that stands out is the *Projection*, which is our workhorse for most non-trivial column operations. This includes more complex logical operators (nested AND/ORs), string manipulation, but also the execution of subselects: In the initial LQP, all subselects are expressed as sub-LQPs attached to a Projection node close to the point of their first use. As such, subselects are executed as if they were stand-alone queries. For non-correlated subselects, this is done only once. For correlated subselects, the query plan contains placeholders that are replaced with the correlated attributes during the execution. Obviously, this is quite inefficient, which is why the optimizer later rewrites the LQP into a more efficient, join-based version.

Optimization. All optimizations are achieved by rules that are executed on the Logical Query Plan (LQP). Rules are maintained by the optimizer and are part of an optimization pipeline that contains single-pass and multiple-pass rules. While some rules need

[3] Hyrise SQL Parser: https://github.com/hyrise/sql-parser

to be executed only once (e.g., the substitution of constant expressions), others can be re-executed if the LQP has been changed by a different rule or if they themselves improve the resulting LQP in multiple passes. A rule takes an LQP as a modifiable input and returns whether it has modified that LQP. The information on whether a rule modified the LQP is used by the optimizer to decide whether iterative rules should be executed again. At the end of every rule stands a valid LQP. Thus, the optimization process can be skipped or stopped after a certain time, e.g., if the expected runtime of the query is determined to be too low to warrant further optimization efforts.

Out of the eight currently implemented rules, we use three examples to highlight how these rules operate on the LQP: the *Predicate Pushdown Rule*, the *Join-Ordering Rule*, and the *Chunk Pruning Rule*. The Predicate Pushdown Rule is a rule that is almost always applicable: For every LQP, it makes sense to execute cheap filtering predicates as early as possible, that is, before more expensive joins or aggregations. Currently, it is applied to all trivial predicates. Correctly estimating the cost of more complex predicates (such as nested predicates or LIKE expressions) is work in progress. The Join-Ordering Rule is an example, which relies on the statistics component to gather the estimated selectivities of the join predicates and on the cost estimator to estimate the cost of the different joins. These joins are then ordered using DpCcp [34] in what is considered to be the most effective order. Finally, our third example rule, the Chunk Pruning Rule, uses the LQP and the filter information discussed in Section 2.4 to identify chunks that will not contribute to the final result. While the LQP is not structurally changed by this rule, the table nodes at the bottom are augmented with the information of which chunks do not need to be passed to the first operator (cf. Section 2.4).

LQP-to-PQP Translation. Finally, the LQP has to be translated into a PQP. As described above, there are potentially multiple physical implementations of a logical operator. The optimizer has already left hints in the LQP nodes. An example of such a hint is when a logical predicate node contains the information that a secondary index can and should be used. Starting from the bottom, each LQP node is now translated into one of the available physical operators. Because all decisions have already been made by the optimizer, nothing of great interest happens here. This is different for the JIT-Aware LQP Translator, which we describe in Section 2.7.

Prepared Statements and Query Plan Caching. One goal in the design of the previous steps was to keep the SQL Pipeline lean, which is why the cost of query planning is comparably low. Still, commonly reoccurring queries can profit from previously generated query plans. We treat SQL Prepared Statements and Query Plan Caching similarly. In both cases, we store a mapping from an SQL query string to a Physical Query Plan. The only difference is that for Prepared Statements, entries in this mapping are manually maintained, while the query plan cache is limited and automatic eviction takes place. For Prepared Statements, we store placeholders instead of actual values. Before the execution of such a statement, these placeholders are replaced with actual values.

Regular SQL queries are currently cached with all parameters left in place. Automatically replacing these with placeholders would increase the number of cache hits but at the same time introduce further complexity. For example, for skewed workloads, where the reuse of a previously generated query plans may not be safe [5], caching PQPs might not always result in the best

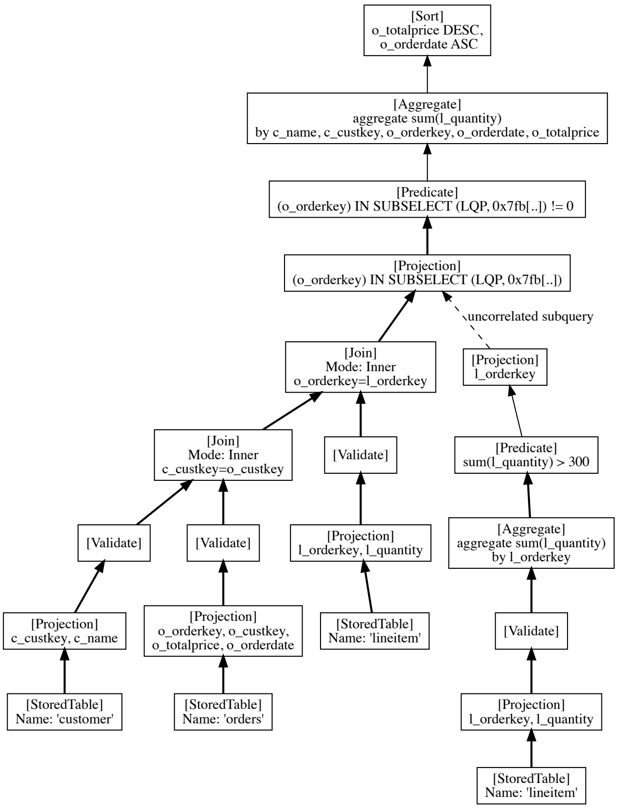


Figure 5: A sample visualization of a Logical Query Plan, here of TPC-H query 18

execution strategy. For these cases, we can easily switch the cache to only cache optimized LQPs, which are then re-optimized once the actual parameters are bound to the placeholders. Because the optimizer has already operated on the cached plan, we can save on optimization time compared to starting the optimization from scratch. Deciding when a PQP is safe to cache and reuse without re-optimization is subject to future work.

2.7 Just-in-Time Compilation

Hyrise's JIT engine is based on three components: The *JIT Repository*, which holds JITtable operators; the *JIT-Aware LQP Translator*, which translates Logical Query Plans into a chain of JITtable operators; and the JIT Code Specializer, which optimizes the operator code and fuses multiple operators into a single loop.

It is based on a code specialization approach, where, during development, generalized C++ code is written for each operator. This code still contains the virtual method calls for calling the next operator, the switches for different column and encoding types, or checks for null values. At runtime, when this information is available, the JIT compiler replaces these abstractions with their concrete values. This is done using LLVM's ORC (On Request Compilation) interface. For the examples, this means that virtual method calls to the next operator can be inlined, type switches can be removed and replaced with the known type, and checks for null values can be removed if the column is known to be non-nullable. The result of this optimization step is a single binary that represents all logical operators between two pipeline breakers. Because of this, we can optimize across operator boundaries and profit from keeping values in the CPU caches (or even registers) as well as from (auto-)vectorization of the code. We believe that this specialization approach is a better fit for our goals as it does not require developers to write code-generating code. The downside of this approach is a high complexity of the specializer component.

We maintain two execution engines, one with and one without support for just-in-time compilation, for three reasons. First, having two engines allows us to compare their performance and identify bottlenecks. Second, despite all effort to make the development of JIT operators as simple as possible, we consider it to still be more complicated as developing traditional operators. Not only do developers have to understand the JIT model, but they also need to familiarize themselves with our specialization engine in order to be able to debug or profile the resulting binary code. Especially for students who work on the project for just a semester, this becomes a limiting factor. Third, most of the research projects evaluated on Hyrise are orthogonal to the optimizations achieved with JIT. The filter-based access pruning methods presented in Section 2.4, for example, can prune the same number of chunks no matter if the remaining chunks are evaluated using traditional or JITted operators. As such, the relative performance impact is comparable, even if the absolute performance of the JITted queries is better.

In some cases, we can achieve a 22x performance improvement over the traditional, operator-based approach, for example when complex expressions have to be calculated. At the current stage, not all operators are implemented as JIT operators (most notably the different joins) and the JIT-aware LQP Translator automatically falls back to non-JITtable implementations. Also, the encoding-specific optimizations have not made it into the JIT component yet, so table scans on dictionary-encoded segments have to decompress all values. Because of this, the JIT component has to be explicitly enabled.

2.8 Concurrency

As described in Section 2.2, chunks become immutable when they reach their maximum capacity. This makes it easier to efficiently encode them without having to consider future modifications. Updates to these chunks are thus implemented in an insert-only fashion as invalidations and reinsertions. While a table's last chunk is mutable and would theoretically support in-place updates, we keep the architecture simple by following the insert-only approach for that chunk as well. For each chunk, we store three vectors: the (1) *begin* and (2) *end commit ids* of the transactions that inserted or invalidated this row, as well as the *transaction id* of a transaction that currently has this row in its set of modified rows.

When a transaction starts, it is assigned a unique (not necessarily contiguous) transaction id by the transaction manager. Also, it stores the commit id of the last transaction that was committed successfully. We previously called this "last commit id of a transaction" [42] or $lcid_T$, but have since renamed it to snapshot commit id, which we believe communicates its purpose better.

A transaction can identify rows that are visible by comparing the begin commit id with its snapshot id: If it is higher, the row has been inserted after the transaction has started and should not be visible. Similarly, a lower end commit id means that the row has already been deleted by the time the transaction started. Invalidations use the transaction id field for two purposes: First, when checking the visibility of a row, seeing its own transaction id in a row signals to the transaction that it has already modified (i.e., inserted or invalidated) the row. This keeps us from having to maintain a separate list of updated rows. Second, as modifying the transaction id is an atomic compare-and-swap operation, it identifies concurrency conflicts. If two transactions concurrently try to set the transaction id of a single row, only one can succeed and the other has to abort. For more detail on our implementation of MVCC, please refer to our description of transaction processing in Hyrise1 [42].

2.9 Scheduling

Most databases do not leave the task of scheduling their work items to the operating system's scheduler [18]. We, too, have instead implemented a cooperative task-based scheduler that tries to keep the OS scheduling out of the equation. Our unit of work is a *task*, which can be an operator, a subroutine within an operator, a maintenance job, or any other subroutine. The easiest type of task has been modeled after `std::thread` to take a function object or a lambda. This keeps the entry threshold for new developers low. All tasks are stored in a task queue, which is using a lock-free `tbb::concurrent_queue`. Tasks can have dependencies on other tasks and only tasks with fulfilled dependencies get emplaced in the queue by the scheduler. Once a task finishes, it iterates over its list of successors and asks them to check if they are now ready to be scheduled.

Hyrise spawns one active worker thread per CPU core. It polls the queue for new tasks to be executed. Once it has begun executing a task, it continues to do so uninterruptedly until it finishes. A task can also spawn subtasks, which are then emplaced in the scheduling queue and executed in parallel.

On NUMA systems, we use one queue per node. Tasks can specify a preferred node, for example when they should be scheduled close to the data that they are processing. When the queue on one node runs dry, workers on that node perform work stealing and attempt to help other nodes with finishing their queue. To prevent high contention on the queues, workers back off for a fixed time interval (currently 10 milliseconds) if the work stealing was unsuccessful.

In line with our goal to keep the system modular, the scheduler can be entirely disabled so that tasks are executed in the main thread. When `schedule` is called on a task, it is either directly executed or, if it has predecessors, their predecessors are executed first. As a result, when measuring the multi-threaded scalability of our system, there are differences between the measurements for one core with and without scheduler. This allows us to inspect the cost of the scheduler.

2.10 Benchmark Runner

Running standard benchmarks on different databases is often a tedious task. Benchmark tables have to be generated and, in many cases, the generated CSV files need some modifications before they can be loaded. For many research databases, some features of SQL are not supported and queries need to be reformulated accordingly. Finally, many databases have special configuration parameters that users need to be aware of. In our case, this includes the default chunk size and encoding options. Our goal is to provide both developers and new users with a binary that is a one-stop solution for executing such benchmarks.

Currently, Hyrise supports the TPC-H benchmark with the binary `hyriseBenchmarkTPCH`. Support for the alternative data generator JCC-H, as well as the benchmarks TPC-C and Join Order Benchmark are in development. These binaries return a JSON file with the total number of executed queries per second, as well as the individual run times of each query and additional information about the setup. Configuration parameters like default chunk size, encoding, or the number of used threads can optionally be set either via the command line or in a JSON file. Our idea is that by providing a git commit id and optionally a JSON file, results can easily be communicated in a reproducible way. Finally, users can provide their own table and queries in .csv and .sql files, which are then automatically executed. We use this to experiment with enterprise-specific workloads and real-world data that, unfortunately, cannot be published. A tool with a similar goal is OLTP-Bench [14], which we plan to support in the future.

3 PLUGINS

For Hyrise to be a multi-purpose research vehicle, it is important not to clutter the code base with limited-purpose extensions. Often, research concepts and their implementations tackle very distinctive challenges that are not necessary for normal database operation. We believe that these specialized implementations should not necessarily become part of the database core to avoid complicating the process of understanding Hyrise and its source code. Not merging them into the main code base also avoids behavior that is unexpected by other researchers, for example when self-tuning components automatically create new indexes. We see plugins as a solution that allows us to extend the system's functionality beyond that of typical database systems. In this section, we present the plugin interface offered by Hyrise and our plans for its use by a self-driving database.

3.1 Interface

Plugins are dynamic libraries, which can be loaded and unloaded by the user during database runtime. They can access all of Hyrise's components using the respective public interface. This means that, except for the boilerplate code needed to initialize them, the development of plugins is almost indistinguishable from that of the database core. Also, most code can be moved from the core into plugins without modification. The main limitation is that while plugins can call public methods, they cannot modify the code of existing components. For example, no new encoding type can be added via a plugin, as all encoding types have to be known during the compilation of Hyrise.

Plugins are implemented as Singletons to ensure that there is only a single instance in the system. The *Plugin Manager* is responsible for administrative work, such as loading and unloading of plugins via `libdl`. We provide a blueprint for plugins[4] that can be used as a starting point for developing new plugins.

3.2 Self-Driving Database

One prime use case for plugins in Hyrise is the area of self-driving database systems. In contrast to traditional databases, these systems do not rely on human database administrators (DBAs) anymore, but adjust their configuration and tune their physical database design autonomously. Such behavior can be achieved by employing workload-driven optimization and machine learning models. To achieve this efficiently, self-driving components need to access and manipulate database-internal data structures and processes. At the same time, it should still be possible to run Hyrise independently of such a plugin and its dependencies and it should not be necessary for other developers to consider such external components. In addition, components of such a self-driving system should be easily exchangeable to enable experiments with different strategies.

The basic architecture and conceptual ideas for a *Generalized Self-Driving Framework* [26] are currently under development in Hyrise. A couple of typical database configuration and physical design aspects should be adjusted autonomously, e.g., the selection of indexes, data placement in NUMA and in replicated systems, and an automatic selection of efficient encoding and compression schemes per chunk. These are either already part of Hyrise and are going to be transferred to the plugin-based self-driving system or are under development.

4 RELATED WORK

The field of database systems for analytical applications has seen vast progress over the last decade [1]. Amongst the first academic systems were MonetDB [9] and C-Store [46], which are disk- and OLAP-optimized systems. While the first column stores focused solely on analytical workloads, the rise of main memory-optimized databases also led to the use of column stores for mixed workload processing (OLxP or HTAP). Amongst these HTAP-optimized column stores are SAP HANA [15] and Hyper [23]. Other comparable database systems include academic systems like Quickstep [38] and Peloton [39]. In the remainder of this section, we discuss major design decisions of Hyrise and how other systems approached them.

The storage layouts of early column-oriented systems like MonetDB and C-Store closely resemble the *decomposition storage model* (DSM [12]), storing attributes in one large consecutive allocation. In addition to the base attributes, both systems further include additional sort orders (cf. C-Store's projections [46] and MonetDB's cracking [22]). A similar layout is used in SAP HANA with the difference that it tries to minimize the memory footprint and avoids redundant data storage. More recent systems use a different approach that splits columns into several smaller units. With the rising importance of NUMA systems, such a form of an automatic horizontal partitioning in fixed-size blocks eases the equal distribution of data over several nodes [10]. This storage paradigm is used, e.g., in Hyrise, HyPer, and Quickstep. Peloton's *tiles* are a hybrid storage layout and resemble the variable-width containers of the first version of Hyrise.

One of the main challenges for columnar databases is handling a steady stream of modifications in HTAP environments. C-Store introduced the separation into read- and write-optimized stores. A similar concept is used in SAP HANA and has been used in Hyrise1, namely separating each table in a read-optimized main partition and a write-optimized delta partition [27, 29]. In contrast, HyPer and Quickstep have chosen a model that is closer to the chunk concept in Hyrise. While Quickstep is read-optimized, HyPer uses uncompressed small blocks at the beginning and shifts to larger and more compressed blocks over time [17].

Storing aggregated and space-efficient data structures for early pruning of data is done by many database systems. The simplest form is to store the min/max values of each column (e.g., Netezza's zone maps or SAP HANA's synopses [36]) or small materialized aggregates [33]. HyPer uses so-called positional small materialized aggregates, which include scan ranges for multiple value ranges [28]. More sophisticated approaches comparable to Hyrise's filters are adaptive range filters [3] or SuRF [47].

In the area of modern in-memory databases, Hyper has been amongst the first systems that generated data-centric code using

[4] Exemplary plugin for Hyrise: https://github.com/Bensk1/example-plugin

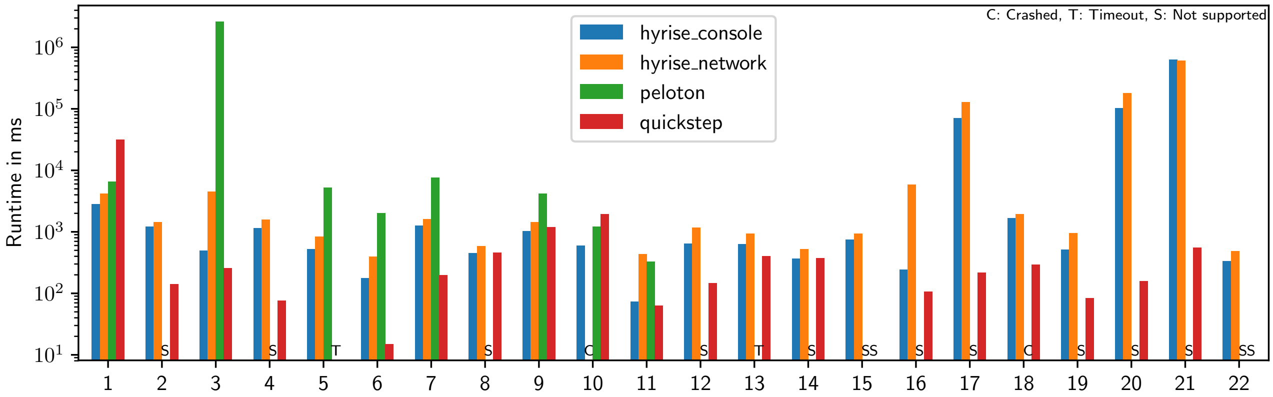


Figure 6: Runtime when executing the TPC-H queries on fresh database instances of Peloton, Quickstep, and Hyrise (SF=1)

LLVM IR [35]. Since then, other database systems, both research and productive, have started to compile queries at runtime, most generating either LLVM IR or C code that is then compiled. Hyrise differs from these approaches in that it does not generate new code but specializes existing code that is stored in LLVM IR and fuses it across operator boundaries. This approach has previously been used by other projects, such as DexterDB [21] or Sharygin et al.'s modifications to PostgreSQL [45]. Hyrise supports both compiled and vectorized queries, allowing for similar evaluations as done by Kersten et al. [24].

The idea of database systems that administrate themselves is almost as old as that of databases itself. Early work [20] dates back to the 1970s where certain aspects of the physical database design were tuned. In later years, vendors of commercial database systems integrated advisors into their products to support human DBAs [11]. Recently, self-driving databases [39] received fresh attention. New systems have a more holistic approach where systems do not rely on human interaction anymore and aim for administrating all aspects of databases instead of certain parts only. With its plugin architecture, Hyrise enables the integration and interplay of self-driving components with reusable components and clearly specified interfaces.

5 EVALUATION

Only recently has the SQL and expression subsystem reached the point where the syntax of all TPC-H queries was supported. Since then, we have constantly been working on improving the optimizer so that it generates better query plans. Although Hyrise still misses some LQP optimizations, which currently limit the performance of a few SQL queries, we can show that Hyrise is already in the same ballpark as other research databases.

5.1 Single Query Comparison

In this section, we compare Hyrise's performance with two other open-source research databases, namely Quickstep [38] and Peloton [39]. All databases were built from source in their release mode[5] using gcc 8.2. Our benchmark system has four Intel Xeon E7-4880 v2 CPUs, a total of 60 cores with 2.5 GHz (up to 3.1 in Turbo Boost Mode), and 2 TB of DRAM. We start the databases using either their network interface and `psql` (Peloton and Hyrise) or by sending queries to the command-line interface using `expect` (Quickstep and Hyrise).

The TPC-H CREATE statements have been slightly modified to reflect the level of data type support in the databases. DECIMAL has been replaced by FLOAT and DATE has been replaced by CHAR(10). While Quickstep seems to offer a date type, using it in comparisons gives us an error. Within the queries, slight modifications have been made to compensate for the lack of date functions. No indexes were created. This evaluation aims at comparing the database performance that researchers can expect when looking at a system for the first time. As such, no additional settings were made. Most importantly, the databases are running with their default number of threads: 1 for Hyrise[6], 120 for Quickstep, and 4 for Peloton. We expect that there are probably better settings for these databases or more advanced code branches that would lead to better results, but limit the evaluation to what is publicly available and what would be a reasonable attempt at generating first numbers.

Results can be found in Figure 6. Considering the mentioned limitations, we can see that for most queries, Hyrise's performance is within an order of magnitude of the other databases. It also shows that there is still optimization potential both within the optimizer and the query execution, but also especially within our network component.

5.2 Chunk Size Evaluation

We discussed Hyrise's storage layout, including the chunk concept, in Section 2.2. In this subsection, we evaluate the performance impact of chunks. The test setup (hardware, compiler, scale factor) remains unchanged from the previous section. Figure 7 depicts the throughput for selected TPC-H queries for chunk capacities varying from one thousand to ten million records per chunk. The throughput is shown relative to a setup without any form of chunking. In the figure, TPC-H queries for which the performance is only marginally impacted by the chunk capacity are combined into "Avg. of other queries". This experiment demonstrates that the chosen chunk capacity influences the throughput to a large extent. Compared to a chunk capacity of 10 000 000 rows, which effectively results in a single chunk for the given scale factor, a chunk capacity of 100 000 improves the throughput by a factor of 26 for TPC-H query 21. At the same time, chunks containing only 1 000 records diminish the throughput by 97% for TPC-H query 22.

[5] Git-Hashes for Hyrise: 9a60098b, Peloton: 3bc6d461, Quickstep: 5cbaa7ef

[6] In the default configuration, the scheduler is currently disabled as we are investigating a performance regression.

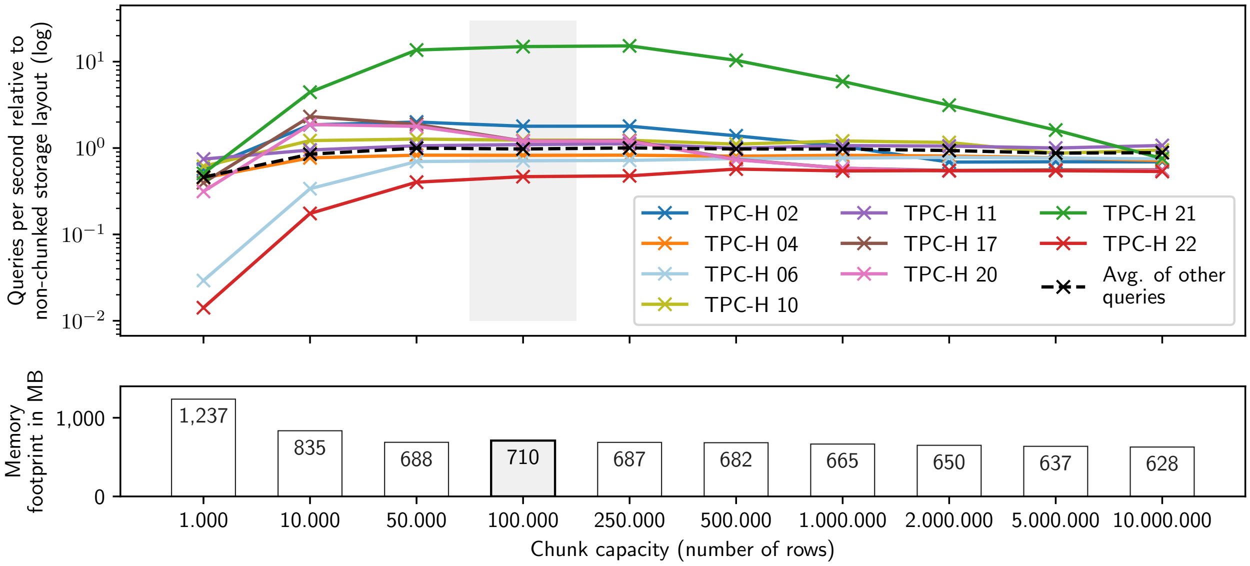


Figure 7: Performance impact and memory consumption of varying chunk capacities (the highlighted chunk capacity of 100 000 is Hyrise's default setting).

As stated earlier, chunks can influence the performance in different ways. First, chunks provide inherent partitions that can be used to distribute the workload. In the results above, however, multithreading was disabled and we see a second effect: Because chunks enable pruning, they can sometimes help in avoiding the access to large parts of the data. Whether pruning is possible depends on the underlying data. If pruning cannot be applied, queries might see drawbacks when too small chunks are used. Queries 6 and 22 with a chunk capacity of 1 000 are an example for this: pruning cannot be applied and many small chunks introduce a significant overhead. Overall, the best throughput is achieved with chunk sizes around 100 000. Compared to a non-chunked layout in Hyrise, the performance increases by 146%.

In addition, the chunk size affects a table's memory footprint as stated in Section 2.2. The lower half of Figure 7 shows the memory footprint for different chunk sizes of all TPC-H tables for a scale factor of 1 when applying dictionary encoding. The configuration that is best for throughput consumes roughly 10% more memory than the most space-efficient configuration. Fine-tuning this parameter on a per-table basis instead of setting it globally is subject to future work. Depending on the encoding scheme, a smaller chunk size can also reduce the footprint in edge cases, for example when a lower number of dictionary entries enables the use of fewer bits for their encoded representation.

6 SUMMARY

In this paper, we have presented how our research database Hyrise was re-engineered and rewritten to be a platform for future research projects. Our new architecture was built around the goals of being easy to understand and extend, enabling end-to-end benchmarking, and delivering high performance even in the presence of multiple abstraction layers. We have described which prior experiences with Hyrise1 have influenced the development, both from an architectural, and from a processual perspective.

Most components that are relevant to our query execution have been explained, including storage and encoding, auxiliary data structures, networking, the SQL pipeline, JIT compilation, multi-threading, and benchmarking. Furthermore, we have described our plugin interface and the self-driving plugin as an example for its use. Finally, we have evaluated chunks as the main storage concept of Hyrise, have shown how pruning can reduce the number of accessed rows during execution, and have compared Hyrise to other research databases.

Future work has been discussed for the separate components. From a high-level perspective, we will focus on implementing the missing LQP optimizations that are current bottlenecks to our performance, will implement more benchmarks, and improve the performance of the system by making better use of modern hardware components.

We invite the reader to experiment with Hyrise by following our first steps guide[7], which not only contains instructions on how to setup Hyrise and run first benchmarks within minutes, but also examples on how to run queries, and visualize them.

ACKNOWLEDGEMENTS

Hyrise is partly developed during Master's seminars and theses. The presented improvements would not have been possible without the contributions and ideas of over 60 students in the last two years. We would like to thank all of them for their dedication. The work of Moritz Eyssen has not only brought Hyrise its current degree of SQL support but, through his immensely helpful feedback on pull requests, also greatly contributed to the overall quality of the code base.

REFERENCES

[1] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2012. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases* 5, 3 (2012), 197–280.

[2] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *PVLDB* 5, 10 (2012), 1064–1075.

[7] Hyrise – First steps: https://github.com/hyrise/hyrise/wiki/FirstSteps

[3] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB* 6, 14 (2013), 1714–1725.

[4] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proc. ICDE*. 362–373.

[5] Harold R Berenson, Peter A Carlin, Nigel R Ellis, Cesar A Galindo-Legaria, Goetz Graefe, Ajay Kalhan, Craig C Peeper, and Samuel H Smith. 2002. Auto-parameterization of database queries. US Patent 6,356,887.

[6] Martin Boissier. 2018. Reducing the Footprint of Main Memory HTAP Systems: Removing, Compressing, Tiering, and Ignoring Data. In *Proc. VLDB PhD Workshop*.

[7] Martin Boissier, Rainer Schlosser, and Matthias Uflacker. 2018. Hybrid Data Layouts for Tiered HTAP Databases with Pareto-Optimal Data Placements. In *Proc. ICDE*. 209–220.

[8] Peter A. Boncz, Angelos-Christos Anatiotis, and Steffen Kläbe. 2017. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *Proc. TPCTC at VLDB*. 103–119.

[9] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*. 225–237.

[10] Craig Chasseur and Jignesh M. Patel. 2013. Design and Evaluation of Storage Organizations for Read-Optimized Main Memory Databases. *PVLDB* 6, 13 (2013), 1474–1485.

[11] Surajit Chaudhuri and Vivek R. Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *Proc. VLDB*. 3–14.

[12] George P. Copeland and Setrag Khoshafian. 1985. A Decomposition Storage Model. In *Proc. SIGMOD*. 268–279.

[13] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *Proc. EDBT*. 72–83.

[14] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288.

[15] Franz Färber, Norman May, Wolfgang Lehner, et al. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[16] Martin Faust, David Schwalb, Jens Krüger, and Hasso Plattner. 2012. Fast Lookups for In-Memory Column Stores: Group-Key Indices, Lookup and Maintenance. In *Proc. ADMS at VLDB*. 13–22.

[17] Florian Funke, Alfons Kemper, and Thomas Neumann. 2012. Compacting Transactional Data in Hybrid OLTP & OLAP Databases. *PVLDB* 5, 11 (2012), 1424–1435.

[18] Jana Giceva, Gerd Zellweger, Gustavo Alonso, and Timothy Roscoe. 2016. Customized OS support for data-processing. In *DaMoN*. 2:1–2:6.

[19] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. 2010. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB* 4, 2 (2010), 105–116.

[20] Michael Hammer. 1977. Self-adaptive automatic data base design. In *Proc. AFIPS*. 123–129.

[21] Carl-Philip Hänsch, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. 2015. Plan Operator Specialization using Reflective Compiler Techniques. In *Proc. BTW*. 363–382.

[22] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB* 4, 9 (2011), 585–597.

[23] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. ICDE*. 195–206.

[24] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *PVLDB* 11, 13 (2018), 2209–2222.

[25] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Proc. CIDR*.

[26] Jan Kossmann. 2018. Self-Driving: From General Purpose to Specialized DBMSs. In *Proc. VLDB PhD Workshop*.

[27] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. 2011. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB* 5, 1 (2011), 61–72.

[28] Harald Lang, Tobias Mühlbauer, Florian Funke, et al. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proc. SIGMOD*. 311–326.

[29] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krüger, and Martin Grund. 2013. High-Performance Transaction Processing in SAP HANA. *IEEE Data Eng. Bull.* 36, 2 (2013), 28–33.

[30] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

[31] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013*. 38–49.

[32] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *Proc. CIDR*.

[33] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*. 476–487.

[34] Guido Moerkotte and Thomas Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *Proc. VLDB*. 930–941.

[35] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.

[36] Anisoara Nica, Reza Sherkat, Mihnea Andrei, Xun Chen, Martin Heidel, Christian Bensberg, and Heiko Gerwens. 2017. Statisticum: Data Statistics Management in SAP HANA. *PVLDB* 10, 12 (2017), 1658–1669.

[37] Prashant Pandey, Michael A. Bender, Rob Johnson, and Robert Patro. 2017. A General-Purpose Counting Filter: Making Every Bit Count. In *Proc. SIGMOD*. 775–787.

[38] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-Up Approach. *PVLDB* 11, 6 (2018), 663–676.

[39] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *Proc. CIDR*.

[40] PostgreSQL 10 Documentation. 2018. Frontend/Backend Protocol. https://www.postgresql.org/docs/10/static/protocol.html

[41] Mark Raasveldt and Hannes Mühleisen. 2017. Don't Hold My Data Hostage - A Case For Client Protocol Redesign. *PVLDB* 10, 10 (2017), 1022–1033.

[42] David Schwalb, Martin Faust, Johannes Wust, Martin Grund, and Hasso Plattner. 2014. Efficient Transaction Processing for Hyrise in Mixed Workload Environments. In *IMDM at VLDB*. 112–125.

[43] David Schwalb, Jan Kossmann, Martin Faust, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. 2015. Hyrise-R: Scale-out and Hot-Standby through Lazy Master Replication for Enterprise Applications. In *Proc. IMDM at VLDB*. 7:1–7:7.

[44] David Schwalb, Girish Kumar, Markus Dreseler, Anusha S., Martin Faust, Adolf Hohl, Tim Berning, Gaurav Makkar, Hasso Plattner, and Parag Deshmukh. 2016. Hyrise-NV: Instant Recovery for In-Memory Databases Using Non-Volatile Memory. In *Proc. DASFAA, Part II*. 267–282.

[45] E. Yu. Sharygin, Ruben Buchatskiy, Roman Zhuykov, and A. R. Sher. 2017. Query compilation in PostgreSQL by specialization of the DBMS source code. *Programming and Computer Software* 43, 6 (2017), 353–365.

[46] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *Proc. VLDB*. 553–564.

[47] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proc. SIGMOD*. 323–336.

Efficient Computation of Probabilistic Core Decomposition at Web-Scale

Fatemeh Esfahani, Venkatesh Srinivasan, Alex Thomo, and Kui Wu
Computer Science Dept., University of Victoria, B.C., Canada
{esfahani,srinivas,thomo,wkui}@uvic.ca

ABSTRACT

Core decomposition is a popular tool for analyzing the structure of network graphs. For probabilistic graphs the computation comes with several challenges and the state-of-the-art approach is not scalable to large graphs. One of the challenges is to compute tail probabilities of vertex degrees in probabilistic graphs. To address this we employ a special version of the Central Limit Theorem (CLT) to obtain the tail probabilities efficiently. Based on our CLT methodology we propose a peeling algorithm to compute the core decomposition of a probabilistic graph that scales to very large graphs and is orders of magnitude faster than the state-of-the-art. Next, we propose a second algorithm that can handle graphs not fitting in memory by processing them sequentially one vertex at a time. This algorithm has the desirable property that it can produce close approximations to true core numbers of vertices in only a fraction of iterations needed for full completion. The graphs in our study are orders of magnitude larger than those considered in the literature. Our extensive experimental results confirm the scalability and efficiency of our algorithms; the largest graph we can process has more than 40 million nodes and 1.5 billion edges and we are able to compute its core decomposition on a commodity machine in about two and half hours.

1 INTRODUCTION

Probabilistic graphs are graphs in which each edge has a probability of existence (cf. [6–8, 19, 21, 43]). Mining probabilistic graphs has become the focus of interest in analyzing many real-world datasets, such as social, trust, communication, and biological networks due to the intrinsic uncertainty present in them. For instance, influence between users (cf. [7, 17, 21]) in a social network can be modeled as a probabilistic graph with probabilities on the edges representing the likelihood that some action of one user will be adopted by another. In terms of trust inference, probabilistic models with trust values as edge probabilities can be used to compute trust associated with a social relationship [27, 28]. In protein-protein networks (cf. [9]) interactions between proteins are obtained through laboratory experiments that are prone to measurement errors resulting in edges labeled with confidence levels that can also be interpreted as probabilities [14, 15, 41].

Discovering dense components is of great importance in analyzing network graphs [29]. A popular way to find such components is core decomposition which has been shown to have a wide variety of applications (cf. [1, 24, 30, 44, 46]). For instance, it can be used in measuring structural diversity is social contagion [44], describing biological functions of proteins in protein-protein interaction networks [30], analyzing network structure's properties to explore collaboration in software teams [46], and also as a metric for sentence selection in text summarization [1]. Probabilistic core decomposition naturally extends all the applications of deterministic core decomposition to probabilistic graphs. Other applications, showcased in [6], are facilitating influence maximization and task-driven team formation in probabilistic graphs.

In k-core computation the goal is to find the maximal subgraph in which each vertex has at least k neighbors within that subgraph. The set of all k-cores of a graph, for various k, forms the core decomposition of that graph [40]. Core decomposition in deterministic graphs has been thoroughly studied in literature [3, 11, 24, 35], and can be computed in $O(m)$ time, where m is the number of the edges in the input graph. However, in the probabilistic context, computing core decomposition is much more challenging.

Here we use the probabilistic (k, η)-core notion introduced by Bonchi, Gullo, Kaltenbrunner, and Volkovich in [6]. In (k, η)-core computation the goal is to find the maximal subgraph in which each vertex has at least k neighbors within that subgraph with probability no less than $\eta \in [0, 1]$. Threshold η is given by the user and defines the desired level of certainty of the output cores. A fundamental notion needed to compute the (k, η)-core is the η-degree of a vertex v. It is the maximum degree such that the probability for v to have that degree is no less than η.

Challenges and contributions. A significant initial challenge is computing η-degrees of graph vertices. In [6], the η-degree of each vertex v is computed using dynamic programming (DP) which has a complexity of $O(d_v^2)$, where d_v is the number of edges incident to v. Unfortunately, in many real social and web networks, d_v can be in the millions and a quadratic algorithm such as DP is impractical.

Our first contribution is the design of an efficient method for computing η-degrees. Our method is based on Lyapunov's special version of the Central Limit Theorem [25, 34] and we show its output to be virtually indistinguishable from exact computation for vertices with a high number of incident edges.

While solving the challenge of computing η-degrees is an important step forward, we still need efficient algorithms to compute core decomposition for large probabilistic graphs. We propose two efficient algorithms to solve this problem.

The first one, which we call the "peeling algorithm" (PA), recursively deletes (peels-off) the vertex of the smallest degree. Our contribution here is in designing efficient arrays for storing important bookkeeping information. Handling these arrays becomes challenging because, differently from the deterministic case, the process of keeping the vertices sorted based on their changing η-degrees is more complicated and we need to shuffle information carefully in order to keep the arrays up to date. Notably, our PA algorithm scales to datasets two orders of magnitude bigger than those that the state-of-the-art algorithm [6] can handle.

For the case when the input graph does not fit in memory, we propose a sequential algorithm (SA) based on the vertex-centric model of computation. The main idea of the SA algorithm is to

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

10.5441/002/edbt.2019.29

maintain an upper-bound, called vertex value, on the core number of each vertex. This upper-bound is initialized to be the η-degree of each vertex, and after each iteration of the algorithm it is tightened further until it reaches the exact core value. While being moderately slower than PA, there are two notable advantages associated with this algorithm. First, the SA algorithm has a memory footprint of $O(n)$ as opposed to $O(m)$ for PA, where n and m are the number of vertices and edges, respectively. This amounts to SA requiring 30 to 40 times lower memory for social and web network graphs in practice. Second, as shown in Section 6, after only a fraction of iterations of the algorithm, we can obtain an approximation very close to the true core numbers of vertices.

In summary, our contributions are as follows.

- We introduce an efficient approach to compute η-degrees using Lyapunov's central limit theorem which gives very accurate approximations on the probability that a vertex can have a certain degree. We prove the accuracy of the approach and show that this method of computing probabilistic degree is numerically stable.
- We propose a peeling algorithm (PA) based on recursive vertex deletions which, by using carefully engineered array structures, is able to scale to graphs two orders of magnitude larger than what the state-of-the-art algorithm can handle.
- For the case when the input graph does not fit into memory, we propose a sequential algorithm (SA) to produce the core decomposition in probabilistic graphs with a low memory footprint. This algorithm can also produce accurate approximations after only a fraction of total iterations, a useful feature to have in applications when exact core numbers are not necessary.

2 BACKGROUND

Cores of deterministic graphs. Let $G = (V, E)$ be an undirected graph, where V is a set of n vertices, and E is a set of m edges. For vertex $v \in V$, let $N_G(v)$ be the set of v's neighbors: $N_G(v) = \{u : (u, v) \in E\}$. The (deterministic) degree of v in G, is equal to $|N_G(v)|$.

Given $V' \subseteq V$, and $E_{V'} = \{(u, v) \in E : u, v \in V'\}$, graph $H = (V', E_{V'})$ is called the subgraph of G induced by V'. Let $k \in [0, d_{\max}(G)]$, where $d_{\max}(G)$ is the maximum vertex degree in G. The k-core of G is defined as the maximal induced subgraph $C_k(G) = (V', E_{V'})$ in which each vertex $v \in V'$ has degree of at least k. The set of all k-cores forms the core decomposition of G. Core decomposition of G is unique and it satisfies the following relation [40]: $G = C_0(G) \supseteq \cdots \supseteq C_{d_{\max}(G)-1} \supseteq C_{d_{\max}(G)}$. The coreness (or core number) of a vertex is defined as the maximum value of k such that the corresponding $C_k(G)$ contains v.

Probabilistic graphs. A probabilistic graph is a triple $\mathcal{G} = (V, E, p)$, where V and E are as before and $p : E \to (0, 1]$ is a function that maps each edge $e \in E$ to its existence probability p_e. For each vertex $v \in V$, the set of edges incident to v is denoted by $\mathcal{N}_v$. $d_v = |\mathcal{N}_v|$ is the number of all edges incident to v which is equal to the deterministic degree of v. In the most common probabilistic graph model (cf. [6, 19, 21]), the existence probability of each edge is assumed to be independent of other edges.

In order to analyze probabilistic graphs, we use the concept of *possible worlds* that are deterministic graph instances of $\mathcal{G}$ in which only a subset of edges appears. The probability of a possible world $G \sqsubseteq \mathcal{G}$ is obtained as follows: $\Pr(G) = \prod_{e \in E_G} p_e \prod_{e \in E \setminus E_G} (1 - p_e)$.

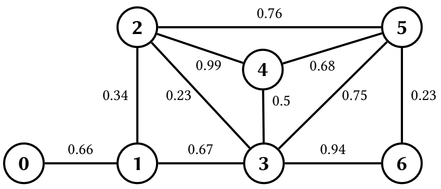


Figure 1: A probabilistic graph.

In a probabilistic graph $\mathcal{G}$, the notion of *η-degree*, where $\eta \in [0, 1]$, denoted by *η-deg*(v), of a vertex v is defined in [6] as the maximum k for which $\Pr_{G \sqsubseteq \mathcal{G}}[d_G(v) \geq k] \geq \eta$, where $k = 0, \ldots, d_v$, and the probability is taken over all the possible worlds $G \sqsubseteq \mathcal{G}$.

For instance, consider Figure 1. When $\eta = 0.5$, vertex 6 has degree at least 2 with probability $0.94 \cdot 0.23 = 0.2162$ (product of probabilities that each of the two edges is in a possible world), and it has degree at least 1 with the probability $1 - ((1 - 0.94) \cdot (1 - 0.23)) = 0.9538$ (complementary probability that none of the two edges is in a possible world) which is greater than the threshold. Thus, the η-degree of vertex 6 is 1.

In the rest of the paper, we use $\Pr[d(v) \geq k]$ to denote $\Pr_{G \sqsubseteq \mathcal{G}}[d_G(v) \geq k]$. The value of $\Pr[d(v) \geq k]$ decreases with the increase of k. Note that d_v is different from $d(v)$; the former is constant, whereas the latter is a random variable.

Cores of probabilistic graphs. In order to extend k-core decomposition to probabilistic graphs, the notion of *(k, η)-core* is defined in [6]:

Definition 1. *Given a probabilistic graph $\mathcal{G} = (V, E, p)$, and a threshold $\eta \in [0, 1]$, (k, η)-core is the maximal induced subgraph $C_{(k,\eta)}(\mathcal{G}) = (V', E_{V'}, p)$ in which the η-degree of each vertex $v \in V'$ is at least k. The set of all (k, η)-cores forms the core decomposition of $\mathcal{G}$.*

The core decomposition in probabilistic graphs is unique, and the (k, η)-cores are nested into each other similar to the deterministic case. The highest value of k for which v belongs to a (k, η)-core is called *η-core* number or probabilistic coreness of v.

Computing η-degrees using Dynamic Programming. We have that $\Pr[d(v) \geq k] = 1 - \sum_{i=0}^{k-1} \Pr[d(v) = i]$. One way of computing $\Pr[d(v) = i]$ is to use dynamic programming as proposed in [6]. However, this method of computing the η-degree has complexity of $O(d_v^2)$ for a vertex v of deterministic degree d_v. This is not practical when d_v is big, say over 20 thousand, which occurs often in all our datasets. In fact, DP cannot finish in reasonable time even for one such vertex. In addition, web-scale graphs normally have millions of nodes with moderate-high degree (e.g., a thousand or more), and if DP is applied to every such node, the total processing time increases considerably. In the next section we introduce an alternative way for fast computation of the η-degree of a vertex v using Lyapunov central limit theorem.

3 COMPUTING η-DEGREES USING CENTRAL LIMIT THEOREM

In this section, we first show how a special version of Central Limit Theorem (CLT) can be applied to accurately estimate $\Pr\big[d(v) \geq k\big]$. Then, we show theoretical bounds on the accuracy of this approximation. Specifically, we show that CLT can produce a very accurate approximation to tail probabilities of the vertex degree.

CLT, one of the most important theorems in statistics, states that given a set of random variables, their properly scaled sum converges to a normal distribution under certain conditions. There are different versions of CLT, with the most common one focusing on independent identically distributed (i.i.d.) random variables. In this paper, we consider a variant called Lyapunov CLT [33, 34] that can be applied when random variables are independent, but not necessarily identically distributed. The Lyapunov's condition imposes a limit on the growth of the third absolute central moment of each random variable in the given sequence ensuring the convergence of the normalized sum of that sequence to standard normal distribution. Formally, we have

Theorem 3.1. ***Lyapunov CLT [25].*** *Let $\xi_1, \xi_2, \cdots, \xi_n$ be a sequence of independent, but non-identically distributed random variables, each with finite expected value μ_k and variance σ_k. Let*

$$s_n^2 = \sum_{k=1}^{n} \sigma_k^2, \tag{1}$$

Lyapunov CLT states that if

$$\lim_{n\to\infty} \frac{1}{s_n^{2+\delta}} \sum_{k=1}^{n} \mathrm{E}[|\xi_k - \mu_k|^{2+\delta}] = 0, \tag{2}$$

for some $\delta > 0$, then $\frac{1}{s_n}\sum_{k=1}^{n}(\xi_k - \mu_k)$ converges in distribution to a standard normal random variable.

Equation (2) is called Lyapunov's condition which in practice is usually tested for the special case $\delta = 1$. The proof for this theorem can be found in [4, 13].

Computing η-degrees using Lyapunov CLT. In what follows, we show how Lyapunov CLT can help compute $\Pr[d(v) \geq k]$, for each vertex v of the input probabilistic graph $\mathcal{G}$.

Recall that for each vertex v we have a set of edges incident to v denoted by $\mathcal{N}_v$. Each e_i in $\mathcal{N}_v$ has existence probability p_i, which is independent of the other edge probabilities in $\mathcal{G}$. Corresponding to each edge e_i in $\mathcal{N}_v$, we define a Bernoulli random variable X_i which takes on 1 with probability p_i, and 0 with probability $(1 - p_i)$. Formally,

$$X_i = \begin{cases} 1 & \text{if edge } e_i \text{ incident to } v \text{ exists in the graph} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Since X_i is a Bernoulli random variable, we know that $\mathrm{E}[X_i] = \mu_i = p_i$ and $\mathrm{Var}[X_i] = p_i(1 - p_i)$. Using the fact that $d(v) = \sum_{i=1}^{d_v} X_i$, we have:

$$\Pr[d(v) \geq k] = \Pr\left[\sum_{i=1}^{d_v} X_i \geq k\right]. \tag{4}$$

According to Equation (4), finding the probability that a vertex v is of the degree at least k is equivalent to computing the probability that the sum of X_i's for v is at least k. In addition, Bernoulli random variables X_i's are independent, but not identically distributed. Thus, if condition (2) is satisfied and if d_v is large enough, we can conclude that $\frac{1}{s_{d_v}}\sum_{i=1}^{d_v}(X_i - \mu_i)$ has standard normal distribution, where $s_{d_v} = \sqrt{\sum_{i=1}^{d_v} p_i(1 - p_i)}$. To compute $\Pr[\sum_{i=1}^{d_v} X_i \geq k]$, we can subtract $\sum_{i=1}^{d_v}\mu_i$ from both sides of the inequality, and then divide by s_{d_v} which results in:

$$\Pr\Big[\sum_{i=1}^{d_v} X_i \geq k\Big] = \Pr\Big[\frac{1}{s_{d_v}}\sum_{i=1}^{d_v}(X_i - \mu_i) \geq \frac{1}{s_{d_v}}(k - \sum_{i=1}^{d_v}\mu_i)\Big]. \tag{5}$$

Using Lyapunov CLT, and setting

$$Z = \frac{1}{s_{d_v}}\sum_{i=1}^{d_v}(X_i - \mu_i), \tag{6}$$

we can say that Z has standard normal distribution. Thus

$$\Pr[d(v) \geq k] \cong \Pr[Z \geq z], \tag{7}$$

where $z = \frac{1}{s_{d_v}}(k - \sum_{i=1}^{d_v}\mu_i)$. In fact, since Z in Equation (6) has standard normal distribution, using the complementary cumulative distribution function [56], we can efficiently evaluate the right hand side of Equation (7). To find the η-degree we start with $k = 1$, and approximate $\Pr[d(v) \geq k]$ using Lyapunov CLT, finding the maximum k for which the probability is above threshold η.

We can apply Theorem 3.1 provided that Lyapunov's condition is satisfied. By setting $\delta = 1$ in Equation (2) we show that this condition holds for a sequence of non-identically distributed Bernoulli random variables.

Theorem 3.2. *Given a sequence of random variables $X_i \sim$ Bernoulli(p_i), for $i \in [1, n]$, the Lyapunov's condition (2) for $\delta = 1$ is satisfied whenever $s_n^2 = \sum_{k=1}^{n} p_k(1 - p_k) \to \infty$.*

Proof. For each Bernoulli random variable X_i we know that $\sigma_i^2 = p_i(1 - p_i)$, and $\mu_i = p_i$. Therefore, $s_n^2 = \sum_{k=1}^{n} p_k(1 - p_k)$ according to the equation (1). On the other hand, when $\delta = 1$, $\mathrm{E}[|X_k - \mu_k|^3]$ is computed as follows:

$$\begin{aligned} \mathrm{E}[|X_k - \mu_k|^3] &= p_k(1 - p_k)^3 + (1 - p_k)p_k^3, \\ &= p_k(1 - p_k)[(1 - p_k)^2 + p_k^2] \leq \sigma_k^2, \end{aligned} \tag{8}$$

where in inequality (8) we have used the fact that $(1-p_k)^2 + p_k^2 \leq 1$. Thus, $\sum_{k=1}^{n}\mathrm{E}[|X_k - \mu_k|^3] \leq s_n^2$. Substituting this in the Lyapunov's condition (2) for $\delta = 1$, we conclude that the condition is satisfied whenever $s_n^2/s_n^3 \to 0$ as $n \to \infty$ which means that s_n should go to infinity as n increases. This is equivalent to $s_n^2 = \sum_{k=1}^{n} p_k(1 - p_k) \to \infty$, and as a result the theorem follows. In other words, since we have

$$s_n^2 = \sum_{k=1}^{n} p_k(1 - p_k) = \sum_{k=1}^{n}\sigma_k^2 \geq n\sigma_{\min}^2, \tag{9}$$

for large n and as long as $\sigma_{\min}^2 = \min_k\{\sigma_k^2\}$ is away from zero, $n\sigma_{\min}^2 \to \infty$ which results in $s_n^2 \to \infty$ as n approaches infinity. □

Accuracy of the Approximation. In order to show the accuracy of the approximation, we refer to the Berry–Esseen theorem [57]: For a given sequence $Y_1, Y_2, \ldots$ of non identically distributed and independent random variables with $E(Y_i) = 0$, $E(Y_i^2) = \sigma_i^2$, and $E(|Y_i^3|) = \rho_i < \infty$, there exists a constant C_0 such that the following inequality is satisfied for all n:

$$\sup_{x\in\mathbb{R}} |F_n(x) - \Phi(x)| \leq C_0 \cdot \psi_0, \tag{10}$$

where F_n is the cumulative distribution of $S_n = \frac{Y_1+Y_2+\cdots+Y_n}{\sqrt{\sigma_1^2+\sigma_2^2+\cdots+\sigma_n^2}}$, which is the sum of Y_i's standardized by the variances, and Φ is the cumulative distribution of the standard normal distribution. In the above inequality ψ_0 is a function given by

$$\psi_0 = \psi_0(\vec{\sigma}, \vec{\rho}) = \left(\sum_{i=1}^{n}\sigma_i^2\right)^{-3/2} \cdot \sum_{i=1}^{n}\rho_i. \tag{11}$$

where $\vec{\sigma} = (\sigma_1, \cdots, \sigma_n)$, and $\vec{\rho} = (\rho_1, \cdots, \rho_n)$ are the vectors of σ_i's and ρ_i's respectively. It should be noted that the best upper-bound obtained so far for C_0 is 0.56 [42].

Based on the Berry–Esseen theorem, the more the number of edges incident to a vertex, the better the accuracy of the results. In the following corollary, we show how to obtain an upper-bound on the maximal error while approximating the true distribution of the sum of X_i's with the normal distribution.

Corollary 1. *For each vertex v in the probabilistic graph $\mathcal{G}$ with X_i's being Bernoulli random variables as defined in (3), where $i = 1, \dots, d_v$, the error bound on the approximation of the right-hand side of Equation (7) to the standard normal distribution is given as follows:*

$$\sup_{x\in\mathbb{R}} \left|F_{d_v}(x) - \Phi(x)\right| \le \frac{0.56}{\sqrt{p_1(1-p_1) + \cdots + p_{d_v}(1-p_{d_v})}}$$

Proof. Setting $Y_i = X_i - p_i$ in equation (6), we can apply the Berry–Essen theorem for random variables $Y_1, Y_2, \cdots, Y_{d_v}$, since for each Y_i, $\mathrm{E}[Y_i] = 0$. In addition,

$$\begin{aligned}\mathrm{E}[Y_i^2] &= \mathrm{E}[(X_i - p_i)^2] = \mathrm{Var}[X_i] = \sigma_i^2 = p_i(1-p_i),\\ \mathrm{E}[|Y_i^3|] &= \mathrm{E}[|X_i - p_i|^3] = \rho_i = (1-p_i)^3 p_i + p_i^3(1-p_i)\\ &= p_i(1-p_i)[(1-p_i)^2 + p_i^2] < \infty. \end{aligned} \tag{12}$$

It should be noted that the random variable

$$S_{d_v} = \frac{(X_1 - p_1) + (X_2 - p_2) + \cdots + (X_{d_v} - p_{d_v})}{\sqrt{\sigma_1^2 + \sigma_2^2 + \cdots + \sigma_{d_v}^2}} \tag{13}$$

in the Berry–Essen theorem is the same as the random variable Z in Equation (6).

Substituting σ_i^2 and ρ_i in Equation (11), we obtain:

$$\begin{aligned}\psi_0 &= \psi_0(\overrightarrow{\sigma}, \overrightarrow{\rho})\\ &= \left(\sum_{i=1}^{d_v} p_i(1-p_i)\right)^{-3/2} \cdot \left(\sum_{i=1}^{d_v} p_i(1-p_i)[(1-p_i)^2 + p_i^2]\right),\end{aligned} \tag{14}$$

Using the fact that $1 = (1-p_i+p_i)^2 = (1-p_i)^2 + p_i^2 + 2p_i(1-p_i) \ge (1-p_i)^2 + p_i^2$, we can simplify (14) to have:

$$\begin{aligned}\psi_0 &\le \left(\sum_{i=1}^{d_v} p_i(1-p_i)\right)^{-3/2} \cdot \left(\sum_{i=1}^{d_v} p_i(1-p_i)\right)\\ &= \left(\sum_{i=1}^{d_v} p_i(1-p_i)\right)^{-1/2} = \frac{1}{\sqrt{p_1(1-p_1) + \cdots + p_{d_v}(1-p_{d_v})}},\end{aligned} \tag{15}$$

Substituting (15) in (10) the stated claim follows. □

4 PEELING ALGORITHM (PA)

In this section we propose a graph peeling algorithm (PA). Graph peeling, that is, (1) recursively deleting the vertex v of smallest degree (2) setting v's coreness to be equal to its degree at the time of deletion, and (3) updating the degrees of v's neighbors while keeping them sorted, is a general idea that has been used broadly in core decomposition of deterministic graphs (cf. [3, 24]). However, it requires substantial algorithmic engineering in order to achieve high scalability when applied to probabilistic graphs. This is because when a vertex v is deleted in the peeling process, updating the η-degrees of v's neighbors and maintaining the data structures up to date are non-trivial. In order to tackle these challenges, we use efficient *array structures* and *lazy updates*, which delay updating the η-degrees of v's neighbors for as long as possible.

Computing and updating η-degrees. An expensive step in the peel-off process is computing initial η-degrees and updating them for those vertices that lose neighbors in a peel-off step. We will depart from the feature of the deterministic case of having vertices sorted at all times based on their current degrees and allow instead the vertices to not be on their precise order as long as at the time of their removal we can fix things up using the key functions and data structures that we define.

More specifically, the computation and update of η-degrees is delayed as much as possible by using easy to compute lower-bounds on η-degrees instead of exact values. It is only when a vertex is candidate for removal that we compute its exact η-degree. At that time the remaining graph is typically much smaller and the computation becomes significantly faster.

Based on Corollary 1, we know that Lyapunov CLT gives a very good approximation on tail probabilities of vertex degrees. We use the values produced by CLT (decremented by a small epsilon) in order to obtain lower-bounds on η-degrees. For simplicity of exposition, we refer to the lower-bound values simply as *values*.

One important difference between the core decomposition of probabilistic and deterministic graphs is that the η-degree of a vertex can decrease by *at most one* when a neighbor is removed (according to Lemma 2 in [6]) as opposed to *exactly one* in deterministic graphs. An array **A** stores, for each value in the input graph, the set of vertices with that value. In the PA algorithm at each iteration, we decrease the value of a vertex v by one if a v's neighbor is removed. When v's turn comes to be removed and processed, we compute its η-degree and if it is the same as its current value we remove v. Otherwise, we repeatedly swap v to the proper place in **A**. There could be several neighbor removals that did not change the η-degree of v, thus the proper place of v in **A** could be far from the next block of vertices, and therefore we might need to perform several swaps.

PA algorithm description. The PA algorithm is given in Algorithm 1. There we have arrays **d**, **b**, **p**, and **A**. For an example see Figure 1 and Table 1. Vertices in the PA algorithm are assumed to be labeled by numbers 0 to $n-1$. Array **d** initially stores for each vertex the lower-bound on the η-degree of that vertex. For instance, vertex 1 has a lower-bound of 1 in Table 1, so **d**[1] = 1. Initially, vertices are stored in **A** in ascending order of their lower-bounds. We have colored **A** in shades of green in Table 1. The first block in **A** contains all the vertices with a lower-bound equal to 0; the second block contains vertices with lower-bound equal to 1, and so on. In order to determine the index boundaries of such blocks in **A**, we define array **b** which stores the index boundaries of the vertex blocks in **A**. In Table 1 we have for instance **b**[1] = 2 and **b**[2] = 6. In order to handle swapping efficiently we define an array **p** which stores the position of each vertex in **A**. For instance, vertex 6 is at position 2 in **A**, therefore **p**[6] = 2.

There are two additional arrays as well; **gone** and **valid**. Since we do not remove vertices physically from the graph, we use array **gone** to keep track of the removed vertices at each step of the algorithm. Array **valid** tells for each vertex v if the η-degree of v is the same as the value **d**[v]. The array **gone** is initially set to false for all the vertices, because none of the vertices has been removed yet. Array **valid** is set to all-false vector because all the vertices are on their lower-bound at the beginning of the algorithm. For instance, in Table 1, we have **valid**[4] = false

which indicates that vertex 4 is on its lower-bound and its η-degree should eventually be computed.

We illustrate a few steps of the PA algorithm on the graph in Figure 1. We set $\eta = 0.5$. The PA algorithm starts with the first vertex v in array **A**, checking whether v is on its lower-bound or its η-degree is available. As can be seen in Table 1, vertex 0 is the first vertex in **A**. Since **valid**$[0] = false$, the vertex 0 is on its lower-bound, and its η-degree might be higher. Therefore, it might not be its turn to be removed. As such, the compute_swap function is called (line 19, Algorithm 1) to compute the η-degree, and then do the required number of swaps to the right, using Algorithm 3, to place the vertex in the proper block of **A**. *Thus, unlike deterministic case we do need to perform several swaps to the right.* The compute_swap function (Algorithm 4) computes the η-degree of vertex 0, which turns out to be 1. Thus, the vertex only needs one swap to the right in **A** to be placed in the block containing all the vertices with degree 1. In order to do each swap in constant time (using Algorithm 3), we swap vertex 0 with the last vertex in the same block which is 6. As a result, the positions of vertices 0 and 6 should be swapped as well: $\mathbf{p}[0] = 2, \mathbf{p}[6] = 1$. Vertex 0 becomes the last element of its current block (the block of vertices with degree 0). In order to make vertex 0 become an element of the next block, the index of the block of vertices with degree 1, **b**[1], is decremented by one to include vertex 0 as its first element. We have $\mathbf{b}[1] = 1$, and vertex 0 becomes the first element of the block of vertices with degree 1.

Table 2 shows the current status of arrays after swapping vertex 0 with 6. The updated values are shown in red color. As can be seen, **valid**$[0] = true$ because now the η-degree of vertex 0 is available. Now, vertex 6 is the first vertex in **A** and the algorithm processes it. The summary of the results is shown in Table 3. Since **valid**$[6] = true$ we can safely remove vertex 6. The corresponding index in array **gone** is set to true, and the coreness of vertex 6 is set to **d**[6], which is 1.

When a vertex v is removed, we process those neighbors u of v with a higher degree (lower-bound or exact) than v's (see lines 14-16), and decrement their degrees by one. Therefore, these neighbors should be moved one block to the left in **A**. This is done in constant time using the swap_left function (line 15) which is shown in Algorithm 2. For instance, vertex 3 is a neighbor of vertex 6 with degree 2, which is decremented by one, from 2 to 1, when vertex 6 is removed. Therefore, vertex 3 should be swapped to the block in the left in **A**, which contains vertices of degree 1. The process is similar to swapping to the right. However, when swapping to the left, vertex u is swapped with the first vertex, w in the same block in **A**. In addition, the positions of u and w are swapped in **p**. Then, the block index in **b** is incremented by one (line 7, Algorithm 2), making u the last element of the previous block.

Correctness of the algorithm. For every $v \in V$, and $C \subseteq V$, a vertex property function [3] is a function $\phi(v, C) : V \times 2^V \to \mathbb{R}$, and it is monotonic if $\forall C_1, C_2 \subseteq V : C_1 \subseteq C_2$ implies that $\phi(v, C_1) \leq \phi(v, C_2)$. According to the result by Batagelj and Zaversnik [3], for a monotonic vertex property function $\phi(v, C)$, the algorithm that repeatedly removes the vertex with smallest ϕ value gives the core decomposition. Since η-degree of a vertex is a monotonic vertex property function [6], then our peeling algorithm, which removes the vertex with smallest η-degree at each iteration of the algorithm, computes the desired core decomposition. Also, it should be noted that, the algorithm scans the next vertex in array **A** only when the η-degree of the previous vertex has been set. In addition, the lower-bounds never surpass the value of η-degrees. Thus, we conclude that the PA algorithm computes the desired core decomposition in probabilistic graphs.

Running time of the PA algorithm. The compute_swap function is dominated by two parts: **(1)** computing η-degrees takes $O(\eta\text{-deg}(u))$ for each vertex u **(2)** swapping a vertex to the proper block; each swap is done in constant time, and the maximum number of swaps required for a vertex u is $O(\eta\text{-deg}(u))$. However, as reported above, initially the difference between the lower-bounds obtained by Lyapunov CLT and the η-degrees is no more than one. Hence, either one or no swap is required initially. Thus, the compute_swap function takes in total

$$O\left(\sum_{v\in V}\;\sum_{u:(u,v)\in\mathcal{N}_v} \eta\text{-deg}(u)\right) = O\left(\sum_{v\in V} d_v\delta'\right) = O(m\delta'),$$

where m is the number of edges, and δ' is the maximum η-degree over all vertices at the time of their removal. The swap_left function swaps each vertex one block to the left in **A** in constant time. Therefore, the main cycle (Algorithm 1, line 6-19) takes $O(m\delta')$. In conclusion, the running time of the PA algorithm is $O(m\delta')$.

Index	0	1	2	3	4	5	6
d	0	1	1	2	1	1	0
A	0	6	1	2	4	5	3
p	1	3	4	7	5	6	2
b	0	2	6				
valid	false	false	false	false	false	false	false
gone	false	false	false	false	false	false	false

Table 1: Arrays d, b, A, p, valid, and gone in the PA algorithm for the graph in Figure 1.

Index	0	1	2	3	4	5	6
d	1	1	1	2	1	1	0
A	6	0	1	2	4	5	3
p	2	3	4	7	5	6	1
b	0	1	6				
valid	true	false	false	false	false	false	false
gone	false	false	false	false	false	false	false

Table 2: First step of the PA algorithm (after swapping vertex 0) for the graph in Figure 1.

Index	0	1	2	3	4	5	6
d	1	1	1	2	1	1	1
A	6	0	1	2	4	5	3
p	2	3	4	7	5	6	1
b	0	0	6				
valid	true	false	false	false	false	false	true
gone	false	false	false	false	false	false	false

Table 3: Second step of the PA algorithm executed on the graph in Figure 1.

5 SEQUENTIAL ALGORITHM (SA)

In this section we present a sequential algorithm (SA) which processes the vertices one-by-one, and as such, does not require the graph to be fully loaded into memory but rather one vertex at a time. Furthermore, as shown in Section 6, after only a fraction of iterations, SA is able to produce high quality results.

SA maintains bookkeeping information for each vertex and has a memory footprint of $O(n)$ as opposed to $O(m)$ for PA. More

Algorithm 1 PA k-core computation function

```
function K_CoreCompute(Graph $\mathcal{G}$, $\eta$)
    initialize($\mathbf{d}, \mathbf{b}, \mathbf{p}, \mathbf{A}, \mathcal{G}$)
    gone ← False                                    ▷ all-false vector
    valid ← False                                   ▷ all-false vector
    $i \leftarrow 1$
    while $i < n$ do
        $v \leftarrow \mathbf{A}[i]$
        if valid[$v$] = true then
            gone[$v$] ← true
            for all $u : (u, v) \in \mathcal{N}_v$ do
                if $\mathbf{d}[u] = \mathbf{d}[v]$ then
                    if valid[$u$] = false then
                        compute_swap($\mathbf{d}, \mathbf{p}, \mathbf{b}, \mathbf{A}, u$, valid)
                if $\mathbf{d}[u] > \mathbf{d}[v]$ then
                    swap_left($\mathbf{d}, \mathbf{p}, \mathbf{b}, \mathbf{A}, u$)
                    valid[$u$] ← false
            $i++$
        else
            compute_swap($\mathbf{d}, \mathbf{p}, \mathbf{b}, \mathbf{A}, v$, valid)
    return d
```

Algorithm 2 PA swap to left function

```
function SWAP_LEFT($\mathbf{d}, \mathbf{p}, \mathbf{b}, \mathbf{A}, u$)
    $du \leftarrow \mathbf{d}[u], pu \leftarrow \mathbf{p}[u]$
    $pw \leftarrow \mathbf{b}[du]$ , $w \leftarrow \mathbf{A}[pw]$
    if $u \neq w$ then
        $\mathbf{A}[pu] \leftarrow w, \mathbf{A}[pw] \leftarrow u$
        $\mathbf{p}[u] \leftarrow pw, \mathbf{p}[w] \leftarrow pu$
    $\mathbf{b}[du]++, \mathbf{d}[u]--$
```

Algorithm 3 PA swap to right function

```
function SWAP_RIGHT($\mathbf{d}, \mathbf{p}, \mathbf{b}, \mathbf{A}, u$)
    $du \leftarrow \mathbf{d}[u], pu \leftarrow \mathbf{p}[u]$
    $pw \leftarrow \mathbf{b}[du + 1] - 1$ , $w \leftarrow \mathbf{A}[pw]$
    if $u \neq w$ then
        $\mathbf{A}[pu] \leftarrow w, \mathbf{A}[pw] \leftarrow u$
        $\mathbf{p}[u] \leftarrow pw, \mathbf{p}[w] \leftarrow pu$
    $\mathbf{b}[du + 1]--, \mathbf{d}[u]++$
```

Algorithm 4 η-degree computation and swap function

```
function COMPUTE_SWAP($\mathbf{d}, \mathbf{p}, \mathbf{b}, \mathbf{A}, u$, valid)
    $\eta\_deg$ ← compute $\eta$-deg($u$)
    valid[$u$] ← true
    $diff \leftarrow \eta\_deg - \mathbf{d}[u]$
    for $j \leftarrow 1$ to $diff$ do
        swap_right($\mathbf{d}, \mathbf{p}, \mathbf{b}, \mathbf{A}, u$)
```

specifically, SA adopts the "semi-external" model of computation, which assumes that for each vertex we can fit a small constant amount of information in main memory while the edges of the graph are stored on disk. As other works have shown, this model is practical for a large number of real-world, web-scale graphs, and widely adopted to handle other graph problems [32, 45, 50, 51].

Algorithms that are sequential and semi-external do exist for deterministic core-decomposition (see [24, 35, 45]). They are based on the idea of maintaining an upper-bound on each vertex's coreness. This upper-bound is initialized to the degree of each vertex, and after each iteration of the algorithm it is tightened further using a simple *locality property* until it reaches the exact core value. The locality property is as follows. The coreness of a vertex v can at most be the largest value k such that v has at least k neighbors with a value greater or equal to k. Locality-based tightening (LBT) lowers the bound of a vertex to be the number k described above.

Unfortunately, this idea does not work for probabilistic core decomposition. LBT does not necessarily converge to true core values of vertices. For an example, consider Figure 1, $\eta = 0.5$, and Table 4. We initialize the upper-bounds to the η-degree of each vertex. Then, we execute a round of LBT. The bound for vertex 3 is tightened to 2. This is because the largest k for which vertex 3 has at least k neighbors with a value at least k is 2. These neighbors are 1, 2, 4, and 5 with a bound equal to 2. Running one more iteration of LBT does not produce any further tightening. However, the true core number of vertex 3 is 1 not 2 (see Table 4, last column). In the following, we tackle the problem with a new procedure we call *probabilistic bound tightening* (PBT). PBT takes into consideration the edge probability values and uses an optimized version of dynamic programming to gradually tighten the upper bounds of vertices to the true coreness. While PBT by itself can be used to compute coreness, we combine PBT with LBT in order to speed up the convergence, since LBT is faster than PBT.

First we show that the obtained values at the end of LBT are always an upper-bound to the true coreness values.

Proposition 1. *Let $\mathcal{G} = (V, E, p)$ be a probabilistic graph. Also, for each vertex $v \in V$, let k_v be the true coreness of v, and $\bar{k}_v$ be the value assigned to v at the end of the LBT phase. Then, $\bar{k}_v \geq k_v$.*

Proof. Once a LBT phase terminates, and the coreness of v is fixed to $\bar{k}_v$; then there should be a subset $V_{\bar{k}_v}$ of neighbors of v with the size at least $\bar{k}_v$, and $\forall u \in V_{\bar{k}_v} : \bar{k}(u) \geq \bar{k}_v$, where $\bar{k}(u)$ is the coreness assigned to u by LBT. However, considering the existence probability of edges incident to v, the value for $\Pr[d_{\mathcal{G}(V_{\bar{k}_v})}(v) \geq \bar{k}_v]$ can be less than η, where $\mathcal{G}(V_{\bar{k}_v})$ is the induced subgraph by $V_{\bar{k}_v}$. On the other hand, since $\Pr[d(v) \geq k]$ is monotonically non-increasing with the value of k, the true coreness of v should be in the interval $[0, \bar{k}_v]$ such that $\Pr[d_{\mathcal{G}(V_{\bar{k}_v})}(v) \geq k] \geq \eta > \Pr[d_{\mathcal{G}(V_{\bar{k}_v})}(v) \geq \bar{k}_v]$. Thus, $\bar{k}_v \geq k_v$. □

Once a LBT phase terminates, PBT starts to check whether the obtained value for each vertex is the true coreness of the vertex or not. If not, the gap is tightened further. We run LBT and PBT repeatedly, one after the other, until the core value for each vertex reaches a fixed point.

Before formally giving the algorithm, we present an example to illustrate how the SA algorithm works. We consider the probabilistic graph in Figure 1 and $\eta = 0.5$. LBT for this example was discussed earlier and the vertex values at the end of it are given in Table 4, third column. As can be seen, there are some vertices whose bounds are different from their true coreness (e.g. vertices 1 and 3). Therefore, the SA algorithm starts PBT to close this gap by checking for exactness of each bound to the true coreness of the vertex. For instance, consider vertex 1. We should check whether the coreness of it can be equal to 2 or not. In probabilistic core decomposition, a vertex v has coreness k if

$\Pr[d_{\mathcal{H}}(v) \geq k] \geq \eta$, where $\mathcal{H}$ is a maximal subgraph in which each vertex has degree of at least k. Checking $\Pr[d_{\mathcal{H}}(1) \geq 2]$, we see that this probability does not pass threshold $\eta = 0.5$, and therefore vertex 1 cannot have coreness of 2 in subgraph $\mathcal{H}$ which contains the neighbors 2 and 3. We perform the check of $\Pr[d_{\mathcal{H}}(v) \geq k]$ using an optimized version of dynamic programming explained later. Vertex 0 cannot belong to the subgraph because its bound is less than 2. For vertex 1, the maximum value, for which the probability passes the threshold, is equal to 1. As a result, the bound on the coreness of vertex 1 is updated to 1. The same process is done for vertex 3. The values obtained at the end of PBT are shown in the fourth column of Table 4, which contains exactly the true core value of each vertex.

v	η-degree	LBT-Round 1	PBT-Round 1	Coreness
0	1	1	1	1
1	2	2	1	1
2	2	2	2	2
3	3	2	1	1
4	2	2	2	2
5	2	2	2	2
6	1	1	1	1

Table 4: Upper-bounds obtained at LBT and PBT phase of SA. $\eta = 0.5$ for the example. LBT and PBT correspond to locality-based and probability bound tightening, respectively.

SA algorithm description. The main cycle of SA is given in Algorithm 5. We define two Boolean variables, *PBT_phase* and *etadegree_change*. Variable *PBT_phase* is used to determine whether a PBT phase has been started or not. The variable *etadegree_change* is used to record whether there is some vertex (in PBT phase) with its η-degree changed or not. Initially both of them are set to *false*. Then, we invoke LBT. Once this process terminates (line 4), the vertex value (upper-bound) of all the vertices should be checked for the possibility of gap between the current vertex value and its true coreness. In fact, for each vertex v it should be investigated whether the degree of v can be greater than or equal to its current vertex value with probability no less than η or not. If so, the current vertex value should be the same as its η-deg(v). Checking whether a vertex should be processed (in PBT) or not is done using the Boolean array *check* which contains a flag for each vertex. If the flag is set to *true*, its vertex value needs to be checked, using the *PBT* function (see line 10). Then, the algorithm enters a PBT phase (lines 8-10). We make a clone, *checkNow*, of *check*; then we reinitialize *check* to the all-false vector (lines 8-9). The clone is needed to be used in the *for* loop in the *PBT* function (line 3, Algorithm 6). If after a PBT phase terminates there is some vertex with its η-degree not equal to its current vertex value, a new LBT phase is started again (line 15) and this process continues until a fixed point is obtained for all the vertices (lines 11-12).

PBT is given in Algorithm 6. The implementation iterates over each vertex v, and checks if there is a gap between v's value and its true coreness; if so, that vertex should be processed. In this process, only the neighbors of v whose current values are greater or equal to v's value are considered because it is only them that can contribute to v's coreness. We compute $\Pr[d(v) \geq C[v]]$ and check whether it passes threshold η or not. If $\Pr[d(v) \geq C[v]] < \eta$, then, since we know that $\Pr[d(v) \geq k]$ is non-increasing with k, the maximum k for which the probability passes the threshold is returned and stored in variable *localValue* (line 5). In this case, all the values from $(C[v] - 1)$ down to *localValue* should be checked as candidate values for the coreness of v and the maximum value (variable *max* in line 16), is chosen as the new value of v. This way we make sure that the algorithm does not go below the real coreness value at each PBT step (lines 8-15). For each value j in the *for* loop (line 8) we check whether $\Pr[d(v) \geq j] \geq \eta$ (line 9). If so, *max* is set to j (line 10) and the vertex value is updated to *max* (line 16). Otherwise, the value for which the probability passes the threshold is stored in variable *value* (line 13), and if it is greater than *max*, the latter is updated to the former (lines 14-15).

It should be noted that in the *for* loop, similar to what we do in lines 4-5 of Algorithm 6, we should compute $\Pr[d(v) \geq j]$ and check if it passes the threshold or not. For this, we use an optimized dynamic programming process as follows. Variable j starts from $C[v] - 1$ and goes down to *localValue*. In order to avoid computing these probabilities from scratch, once we compute $\Pr[d_{\mathcal{H}}(v) \geq C[v]]$, we cache the following probabilities $\Pr[d_{\mathcal{H}}(v) = 0], \cdots, \Pr[d_{\mathcal{H}}(v) = C[v]]$, where $\mathcal{H}$ contains all the neighbors of v whose upper-bound is at least $C[v]$. Then, since for $j = C[v] - 1$, $\mathcal{H}'$ contains all the neighbors whose upper-bound is exactly equal to j (we denote this set by V_j) and higher, we can have $\mathcal{H}' = V_j \cup \mathcal{H}$. Therefore, to compute $\Pr[d_{\mathcal{H}'}(v) \geq j]$ we use the computed probabilities in $\mathcal{H}$ and only consider vertices in V_j. This way we optimize DP to compute the probabilities very fast since V_j is typically small (e.g., about 100 in our later evaluated large-scale graphs). For the next iteration, we store all the probabilities computed in the previous iteration and use them to compute new probabilities. In the following we describe our DP process in more detail.

Let assume that we have computed $\Pr[d_{\mathcal{H}}(v) = k]$, where $k = 0, 1, ..., j$, and $\mathcal{H}$ is a subgraph of the input probabilistic graph in which each vertex has core value at least j. Also, let V_{j-1} contain all the vertices (including the neighbours of v) whose core value is exactly $j-1$. Thus, $\mathcal{H}' = \mathcal{H} \cup V_{j-1}$ is the subgraph whose vertices have core value at least $j - 1$. We denote $\{e_1, ..., e_x\}$ to be the set of edges incident to v such that for each $e_i = (u_i, v)$, $u_i \in V_{j-1}$, where $i = 1, ..., x$. In order to evaluate $\Pr[d_{\mathcal{H}'}(v) = k']$, where $0 \leq k' \leq k$, and avoid from scratch computation, we denote the degree of v in the subgraph $\mathcal{H}'$ by $d(v \mid (\mathcal{H} \cup \{e_1, ..., e_x\}))$. Then, it holds that:

$$\begin{aligned} &\Pr[d(v \mid (\mathcal{H} \cup \{e_1, ..., e_x\})) = k'] = \\ &= p_{e_x} \Pr[d(v \mid (\mathcal{H} \cup \{e_1, ..., e_{x-1}\})) = k' - 1] + \\ &+ (1 - p_{e_x}) \Pr[d(v \mid (\mathcal{H} \cup \{e_1, ..., e_{x-1}\})) = k']. \end{aligned} \tag{16}$$

Letting $T(x, k') = \Pr[d(v \mid (\mathcal{H} \cup \{e_1, ..., e_x\})) = k']$, we have the following recursive formula:

$$T(x, k') = p_{e_x} T(x - 1, k' - 1) + (1 - p_{e_x}) T(x - 1, k') \tag{17}$$

with the following base cases:

$$\begin{cases} T(0, k') = \Pr[d_{\mathcal{H}}(v) = k'], & 0 \leq k' \leq k \\ T(x, -1) = 0, \end{cases} \tag{18}$$

As can be seen, in the base case of the recursive formula, we are using the previously computed probabilities to compute the probabilistic degree of vertex v in the new subgraph, which results in saving time significantly.

Since the vertex value of v is updated, all its neighbors with value at least the vertex value of v should be checked for validity of their vertex value. We show in the following that the PBT estimate of the coreness eventually equals the true coreness for each vertex.

Algorithm 5 SA k-core computation function

```
function SA_CORE_COMPUTATION(Graph $\mathcal{G}$)
    $PBT\_phase \leftarrow false$
    $C \leftarrow \mathbf{0}$                          ▷ array of core values
    LBT()
    $check \leftarrow$ True                            ▷ all-true vector
    $etadegree\_change \leftarrow false$
    while true do
        $checkNow \leftarrow check.clone()$
        $check \leftarrow$ False                       ▷ all-false vector
        PBT($\mathcal{G}, checkNow$)
        if $etadegree\_change = false$ then
            break
        else
            $PBT\_phase = true$
            LBT()
            $etadegree\_change \leftarrow false$
    return $cores$
```

Algorithm 6 SA probabilistic bound tightening function

```
1:  function PBT($\mathcal{G}, checkNow$)
2:      for all $v \in$ V do
3:          if $checkNow[v] = true$ then
4:              if $\Pr[d_{\mathcal{H}}(v) \geq C[v]] < \eta$ then
5:                  $localValue \leftarrow$ compute $\eta$-deg$_{\mathcal{H}}(v)$
6:                  $max \leftarrow localValue$
7:                  $m \leftarrow C[v] - 1$
8:                  for all $j \leftarrow m$ down to $localValue$ do
9:                      if $\Pr[d_{\mathcal{H}'}(v) \geq j] \geq \eta$ then
10:                         $max \leftarrow j$
11:                         break
12:                     else
13:                         $value \leftarrow$ compute $\eta$-deg$_{\mathcal{H}'}(v)$
14:                         if $value \geq max$ then
15:                             $max \leftarrow value$
16:                 $C[v] \leftarrow max$
17:                 $etadegree\_change \leftarrow true$
18:                 for all $u : (u, v) \in \mathcal{N}_v$ do
19:                     if $C[u] \geq C[v]$ then
20:                         $check[u] \leftarrow true$
```

Correctness of the SA algorithm. We prove this by contradiction. Suppose that after the last PBT phase, there is vertex u_1 such that $k(u_1) = k_1$ (real coreness) and $core[u_1] = k' > k_1$, where $core$ is the coreness assigned to u_1 after PBT phase. Since $k(u_1) = k_1$, k_1 is the maximum value such that $\Pr[d_{\mathcal{H}}(u_1) \geq k_1] \geq \eta$, and $\Pr[d_{\mathcal{H}'}(u_1) \geq i] < \eta$ for all $i > k_1$, where $\mathcal{H}$ and $\mathcal{H}'$ are the maximal induced subgraphs in which each vertex has degree at least k_1 and i, respectively, with probability no less than η. Based on the properties of cores of a graph, we know that $\mathcal{H}' \subseteq \mathcal{H}$. If all the neighbors of u_1 in $\mathcal{H}$ have coreness k_1, then u_1 will not have any neighbors with coreness greater than k_1 in $\mathcal{H}'$, so u_1 eventually sets $core[u_1]$ equal to k_1, which is a contradiction. The similar argument holds if all the neighbors of u_1 in $\mathcal{H}$ have coreness greater than k_1. Since $k(u_1) = k_1$ and $core[u_1] = k' > k_1$, there should exist a neighbor u_2 with $k(u_2) = k_1$ and $core[u_2] > k_1$ such that $\Pr[deg_{\mathcal{H}' \cup \{u_2\}}(u_1) \geq k'] \geq \eta$, because otherwise $\Pr[d_{\mathcal{H}'}(u_1) \geq k'] < \eta$. In fact, the existence of this neighbor will contribute to the assigned coreness of u_1 to be greater than k_1.

Now by reasoning similar to [35], we can build a sequence of vertices $S = \{u_i, u_{i+1}, u_{i+2}, ..., u_j = u_i\}$ connected to each other with $k(u_i) = k_1$ and $core[u_i] > k_1$. For each vertex u_i in S, let V_i be the set of all neighbors of u_i in $\mathcal{H}'$. Now, we can define a set $U = S \cup \bigcup_{u_i \in S} V_i$. The corresponding induced subgraph $\mathcal{G}(U)$ is a k'-core, because all the vertices in V_i have coreness at least k' with probability greater than or equal to η. Also, since for each vertex u_i in S, $\Pr[d_{V_i \cup \{u_{i+1}\}}(u_i) \geq k'] \geq \eta$, we have that $\mathcal{G}(U)$ is a k'-core where $k' > k_1$. Hence, we find a subgraph whose vertices have coreness $k' > k_1$ which is a contradiction because we assumed that each vertex in S has coreness k_1. Therefore, k_1 is not a maximal (i.e. true) coreness.

Running time of the SA algorithm. The time complexity of the algorithm is dominated by time complexity of the PBT because LBT has a time complexity of $O(N - K + 1)$ [35], where K is the number of vertices with minimal degree (in probabilistic graphs it refers to η-degree). To analyze the time complexity of a PBT step (Algorithm 6), let Δ be the maximum upper-bound on the η-degree over all the vertices in all the probabilistic bound tightening steps. Lines 4 and 5 (η-degree computation) are done simultaneously, and take time $O(d_v C[v])$ for each vertex v, where $C[v]$ is the upper-bound on the η-degree of v. In practice this computation is fast because the η-degree computation is performed on the subgraph $\mathcal{H}$ which contains only those neighbors of v whose upper-bound is at least $C[v]$ (not all the d_v vertices). In the worst case the inner loop is repeated $C[v]$ times, and each time the η-degree computation and the checking of the probability threshold (lines 9 and 13) are performed similarly to what explained above. Therefore, the time complexity of this part is: $\sum_{j=1}^{C[v]} O(jd_v)$. It should be noted that at each iteration in the for loop, we use the previously computed probabilities to avoid computing η-degrees from scratch. However, here we consider the worst case analysis of the algorithm. The time complexity of lines 18-20 is $O(d_v)$. As a result, each PBT iteration takes $\sum_{v \in V} \left(O(C[v]d_v) + \sum_{j=1}^{C[v]} O(jd_v) + O(d_v) \right) = \sum_{v \in V} \left(O(\Delta d_v) + O(\theta d_v) + O(d_v) \right)$, where $\theta = \Delta^2$. Therefore, the time complexity of each PBT round is $O(m\theta)$, where m is the total number of edges. In the worst case, we assume that at each PBT round, the difference between the actual coreness of a vertex and its initial estimate (the initial η-degree) decreases by one unit. Thus, in the worst case $\Gamma = \sum_{v \in V} \eta$-deg$(v)$ PBT steps are required. Therefore, the total running time can be expressed as: $O(\Lambda m)$, where $\Lambda = \Gamma\theta$.

As in [35], the complexity upper bounds for such iterative algorithms are not representative of practical performance. In practice, LBT and PBT are fast, and the number of iterations is only a handful, thus SA is an efficient algorithm for large datasets while requiring only $O(n)$ memory footprint.

6 EXPERIMENTS

In this section, we present our experimental results. Our implementations are in Java and the experiments are conducted on a commodity machine with Intel i7, 2.2Ghz CPU, and 12Gb RAM, running Ubuntu 14.03. The hard disk is Seagate Barracuda ST31000524AS 2TB 7200 RPM.

The statistics for all of the datasets we consider are shown in Table 5. We obtained flicker, dblp, and biomine from the authors of [6], and the rest of the datasets from Laboratory of Web Algorithmics.[1] The datasets are divided by horizontal lines according to their size, small (S), medium (M), large (L), and extra large (XL).

[1] http://law.di.unimi.it/datasets.php

We use the Webgraph framework [5] to store these datasets. The flickr, dblp, and biomine datasets already contained probability values. For the other datasets we generated probability values uniformly distributed in [0, 1].

We evaluate our algorithms in three important aspects. First is numerical stability. As [6] points out, using probability values may lead to numerical instability. We discuss this in Subsection 6.1. Second is the quality of our η-degree lower-bounds using Lyapunov CLT. We evaluate and discuss this in Subsection 6.2. Third is the efficiency of our proposed algorithms. We show our performance results in Subsection 6.3.

6.1 Numerical Stability

For evaluating numerical stability we refer to the results of η-degree computations shown in Table 6. The table contains results for $\eta = 0$, $\eta = 10^{-9}$ and $d_v = 100$, $d_v = 1000$. For each combination of η and d_v we compute η-degrees for 1000 randomly selected vertices. We show results for twitter-2010, but any of the other datasets above can be used for this experiment to obtain similar results.

Using the BigDecimal class in Java we adjust the numerical precision to different levels when using DP for computing η-degrees. We compare numerical results of DP using different precision levels in Java, where DPU (DP with unlimited precision) is the highest level of precision and thus the gold standard. The DP128 and DP256 columns in the table correspond to computing η-degrees using DP by setting precision to 128 bit and 256 bit, respectively. We observe that the more we increase the level of precision, the longer it takes to perform the computation. For example, when executing the DP algorithm for computing η-degrees for vertices with $d_v = 1000$, it takes more than 7000 times longer to perform the computation using unlimited precision than using the default (plain) number computation in Java.

Following [6], we start by setting $\eta = 0$ (top two parts). For this η the (k, η)-core decomposition of a probabilistic graph should coincide with the core decomposition of the deterministic graph derived by ignoring probabilities. The accuracy can thus be measured by comparing, for each vertex, the $(k, 0)$-core number with the core number obtained on the deterministic version of the graph.

We can see that for $\eta = 0$, we need a lot of precision (bits) to achieve error free computation. For example, when $d_v = 100$, we need at least 256-bit precision, whereas when the degree is 1000, we need DP with unlimited precision.

The DPLog2 column of the table shows the results if we operate in logarithmic space[2]. We can see that DP operating in logarithmic space is very stable and never produces any error at all, while being much faster than the DP variants operating with specified precision.

We test $\eta = 0$ to compare our results to previous work [6]. However, the situation changes significantly if η is greater than zero even by a very small amount. If we set $\eta = 10^{-9}$ (one billionth) or higher, we never get an error even for plain DP which is much faster than any other DP variant including the one in log space (see the two bottom parts). We tested with η in $[10^{-9}, 0.5]$ and obtained similar results.

Name	$\|V\|$	$\|E\|$
flickr	24,125	300,836
dblp	684,911	2,284,991
cnr-2000	325,557	2,738,969
biomine	1,008,201	6,722,503
ljournal-2008	5,363,260	49,514,271
arabic-2005	22,744,080	553,903,073
uk-2005	39,459,925	783,027,125
twitter-2010	41,652,230	1,202,513,046

Table 5: Dataset Statistics

	DP	DP128	DP256	DPU	DPlog2	CLT
NE	100%	99%	0%	0%	0%	0%
AE	8%	5%	0%	0%	0%	0%
AT	0.09	11.44	12.57	27.71	1.56	0.05
NE	100%	100%	100%	0%	0%	0%
AE	40%	35%	28%	0%	0%	0%
AT	3.3	1238	1499	26355	222	0.1
NE	0%	0%	0%	0%	0%	43%
AE	0%	0%	0%	0%	0%	1%
AT	0.19	14.41	15.34	43.86	2.14	0.27
NE	0%	0%	0%	0%	0%	0%
AE	0%	0%	0%	0%	0%	0%
AT	3.7	1244	1382	23934	171	0.3

Table 6: Error statistics and average running time for different precision levels for $\eta = 0$ and $\eta = 10^{-9}$. NE stands for Number of Errors, AE for Average Relative Error, and AT for Average Time (ms). Specifically, AE=$\|error\|/true\_value$.
1st part*: $\eta = 0, d_v = 100$;* **2nd part***: $\eta = 0, d_v = 1000$;* **3rd part***: $\eta = 10^{-9}$, $d_v = 100$;* **4th part***: $\eta = 10^{-9}$, $d_v = 1000$*
DP*: DP using plain numbers in Java;* **DP128, DP256, DPU***: DP using BigDecimal in Java and setting the precision to 128 bits, 256 bits, and unlimited, respectively;* **DPlog2***: DP doing computations in log space.*

The last column of Table 6 shows the accuracy of η-degree computation when using Lyapunov CLT instead of DP. For $\eta = 0$, computing η-degrees using CLT is error-free. More interesting are the bottom two parts for η greater than zero. For vertices with $d_v = 100$, CLT makes errors in computation (on 43% of the vertices), however, those errors are small; only 1% on average of the η-degree value.

When considering vertices with $d_v = 1000$, the computation using CLT is error-free and furthermore it is orders of magnitude faster than the variants of DP. Again, we also tested with a variety of η levels and obtained similar results. Guided by the above, in our algorithms, we set the threshold on d_v to start using Lyapunov CLT for computing η-degrees at 1500. Recall that the η-degree computation is needed in PA when a vertex becomes candidate for removal, while in SA it is needed when initializing vertex values.

In summary, regarding numerical stability, our contribution is to show that DP is sensitive to the setting of η value and becomes error-free once η is greater than zero even by a small amount. This was not investigated thoroughly before. On the other hand, CLT is resilient to different η values (even $\eta = 0$). With respect to efficiency in computing η-degrees, when $d_v \geq 1000$, CLT can be used to produce error-free computations orders of magnitude faster than all the DP variants.

6.2 Accuracy of CLT as a lower-bound

Here we investigate the quality of the lower-bounds on η-degrees obtained using CLT. Namely, we compare CLT with another method for deriving lower-bounds used in [6], which utilizes a

[2] https://en.wikipedia.org/wiki/Logarithm

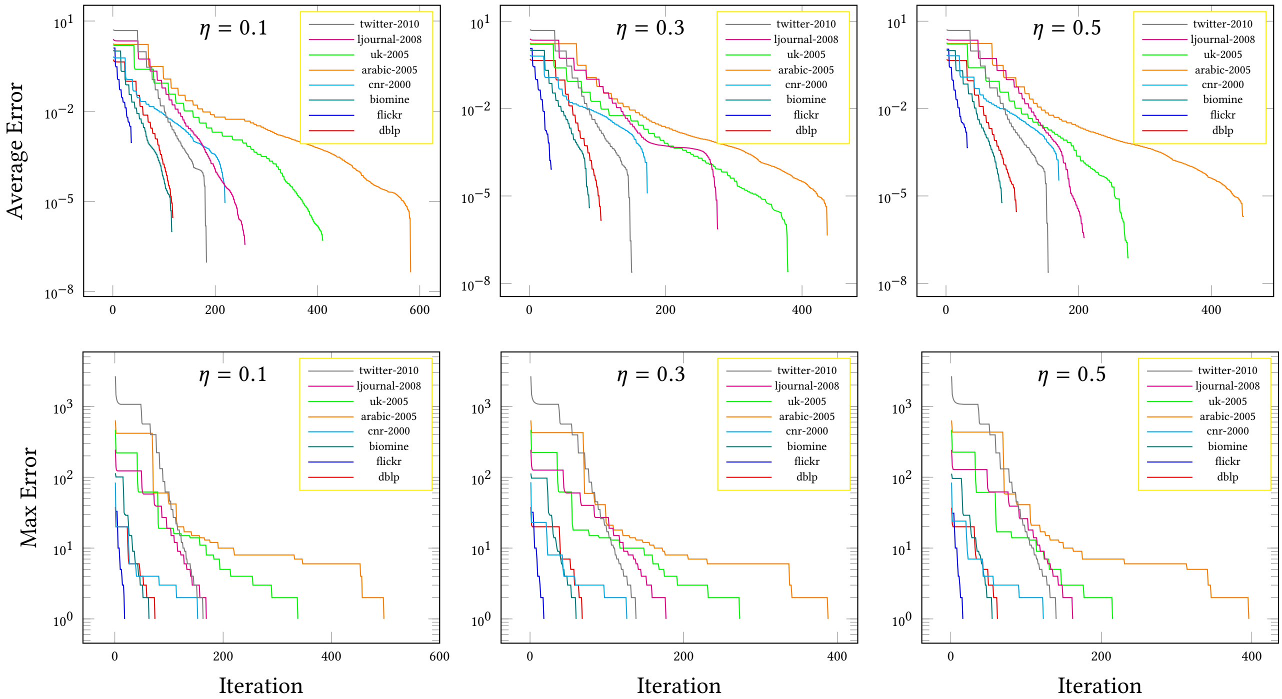


Figure 2: Average error (average of difference from true core coreness) and max error (maximum difference from true coreness) versus iterations for different values of η.

Dataset	Max Error	
	CLT	Beta Function
biomine	1	1,663
cnr-2000	1	9,056
ljournal-2008	1	9,453

Table 7: Maximum error of CLT and regularized beta function for selected datasets.

formula based on the regularized beta function.[3] We computed initial η-degrees for biomine, cnr-2000, and ljournal-2008 with $\eta = 0.1$ using DP, Lyapunov CLT, and regularized beta function. Then, we computed the maximum error for CLT and regularized beta function. We report the results in Table 7. As can be seen, the maximum error for CLT for all the datasets is one which means that the difference between the values obtained by CLT and the true values is either zero or at most one. On the other hand the max error for the regularized beta function is big, in the order of thousands.

The small value of error for CLT is of great importance in the main cycle of the PA algorithm where each vertex is processed based on its lower-bound before its true η-degree is computed. To summarize, since the difference between each vertex's lower-bound (computed by CLT) and its true η-degree is small, we only need to do a small number of swaps to place a vertex in the proper place in the array **A** resulting in significant savings in running time.

6.3 Efficiency of the Proposed Algorithms

Table 8 shows the running times of the PA (left) and SA (right) algorithms on the selected datasets. For twitter-2010 we report the results for different values of η ranging from 0.1 to 0.5. For the other datasets, we only show results for $\eta = 0.1$ and omit results for $\eta = 0.2, \ldots, 0.5$, since they are similar to those for $\eta = 0.1$ and their nature is the similar to what we see for twitter-2010.

For the small and medium datasets, PA produced the results in about 2 sec for the small and 4 to 6 sec for the medium datasets. PA also performed well on the large datasets; biomine and ljournal-2008, computing the core decomposition of these two graphs on average in only 40 sec and 2 min, respectively. Notably, for biomine for instance, our algorithm is 32 times faster than the algorithm of [6] (see also Table 10 which we discuss later).

The running time of PA is good on the very large datasets as well. On uk-2005 and arabic-2005, PA completed in about 28 min and 42 min, respectively; less than one hour; in contrast the state-of-art [6] was not able to complete for these datasets in our machine after one day. For twitter-2010, which is much larger than uk-2005 and arabic-2005, with a maximum η-degree of $1,500,282$, our PA algorithm completed in around two hours, which is impressive for processing such a big dataset on our consumer-grade machine.

The running times of SA are shown in Table 8 (right part). For the small and medium datasets the total running times of SA were similar to those of PA. For the large datasets, SA needed more time, 2-3 times more, than PA. However, we recall that SA has a much smaller memory footprint than PA, namely $O(n)$ as opposed to $O(m)$ for PA. As such we are trading time for space when using SA over PA, a beneficial feature to have when using low-cost, cloud-server machines.

Another benefit of using SA is that it can produce good approximate results after only a small fraction of total iterations. We discuss this more later in this section.

Effect of η values. We observe that the total running time does not change much as the value of η increases. This is because when η increases, the η-degree of a vertex might not change or slightly decrease and as such this does not have a significant effect on the total running time of the algorithms. This is also evident in

[3] http://mathworld.wolfram.com/RegularizedBetaFunction.html

Table 9 where the average coreness (kavg) and the maximum coreness (kmax) decrease only slightly as η increases. As before, we report the average coreness, the maximum coreness, and the maximum η-degree for twitter-2010 for $\eta = 0.1, \ldots, 0.5$, whereas for the other datasets we only show the values for $\eta = 0.1$.

Comparing with the algorithm of [6]. We also compared the running times of our algorithms, PA and SA, to that of the algorithm of [6] (which we call BGKV[4]), for $\eta = 0.1$. We considered the (k, η)-core implementation made available to us by the authors of [6] with numbers represented by using BigDecimal in Java. From this, we also created a second version, which we call BGKV-2, where we replaced the computations using BigDecimal with plain computations in Java. This is because as shown in Subsection 6.1, for η greater than 0, the use of BigDecimal for DP is not warranted.

Table 10 summarizes the running times on three datasets, flickr, dblp and biomine. For biomine, which is a large dataset, we were only able to use up to 64 precision bits. We can see that both PA and SA are significantly faster than BGKV. For example, for biomine our algorithms are more than 32 times faster than BGKV.

BGKV-2 is faster than BGKV. For the small datasets, flickr and dblp, it is even faster than PA and SA. This can be attributed to the fact that since the memory footprint is small, the benefit of using our algorithms is outweighed by the overhead of using the Webgraph compression structures in PA and SA.

The situation changes significantly as the size of the datasets grows. For biomine, BGKV-2 is about twice slower than PA and SA. For the rest of the datasets that are bigger than biomine, BGKV-2 cannot run to completion in our machine after one day. For those datasets, our algorithms, PA and SA, are the only algorithms that can produce results in a matter of minutes or few hours (for twitter-2010), Table 8.

Convergence speed of SA. To further investigate the execution of SA as it unfolds with time, we look at average error and maximum difference (max error) from the true core value over the sequence of iterations (see Figure 2). As shown by the plots in the top part of the figure, the average error sharply decreases for all the datasets which we consider, except for uk-2005 and arabic-2005 whose average errors decrease more gradually.

Similar to the average error, the maximum difference from the true core value (shown in the plots in the bottom part) drops quickly, becoming 1 in only a fraction of the total number of iterations. Furthermore, as the value of η increases, the total number of iterations required decreases.

These results show that SA produces approximate results of good quality in only a fraction of iterations needed for completion. For instance, for arabic-2005 with $\eta = 0.1$, the average error drops below 0.01 at iteration 200, only one third of the total number of required iterations (about 600, see the end of the curve). Depending on the application domain this can be a desirable property during data analysis.

Dataset	η	PA-Running Time	SA-Running Time
flickr	0.1	2.05	1.88
dblp	0.1	5.81	9.57
cnr-2000	0.1	3.99	8.49
biomine	0.1	40	40
ljournal-2008	0.1	120	342
arabic-2005	0.1	2,539	2,513
uk-2005	0.1	1,709	1,961
twitter-2010	0.1	8,096	21,662
	0.2	8,547	20,268
	0.3	8,358	19,670
	0.4	8,364	20,544
	0.5	8,929	20,111

Table 8: Running times (sec) of PA and SA. Numbers less than 10 have been rounded to 2 decimal places, and those above 10 have been rounded to the nearest integer.

Dataset	**η-degmax**	**kmax**	**kavg**	**η**
flickr	78	46	3.70439	0.1
dblp	162	26	1.99825	0.1
cnr-2000	9,255	38	5.77113	0.1
biomine	6,269	79	3.08697	0.1
ljournal-2008	9,664	156	5.4735	0.1
arabic-2005	287,949	1,088	15.2903	0.1
uk-2005	888,658	274	13.0279	0.1
twitter-2010	1,500,282	986	15.7391	0.1
	1,499,970	976	15.0873	0.2
	1,499,744	970	14.6407	0.3
	1,499,552	964	14.2919	0.4
	1,499,372	959	13.9927	0.5

Table 9: Maximum η-degree, maximum probabilistic coreness, average probabilistic coreness, value of the threshold η.

Algorithm		flickr	dblp	biomine
BGKV	pr=64	18	90	2,493
	pr=128	27	133	N.P.
	pr=256	35	148	N.P.
BGKV-2		0.49	4.26	85
PA		2.05	5.81	40
SA		1.88	9.57	40

Table 10: Running time (sec) of the algorithm in [6] with BigDecimal (BGKV), and without (BGKV-2) versus PA and SA. "pr" is the precision (bits) used. BGKV cannot run for biomine to completion after one day (we use N.P. for "Not Possible"). BGKV-2 is faster for the small datasets, flickr and dblp, but twice slower for biomine than PA and SA. For the rest of the datasets that are bigger than biomine, both BGKV and BGKV-2 cannot run to completion in our machine after one day. Our algorithms PA and SA can produce results for every dataset, see Table 8.

7 RELATED WORK

Among different notions of cohesive subgraphs, k-core is one of the most popular (cf. [1, 24, 30, 44, 46]. Other definitions of dense subgraphs such as maximal cliques can also be computed using k-core decomposition [16]. In deterministic graphs, the computation of k-core has been well studied. Batagelj and Zaversnik [3] give an efficient peeling algorithm for deterministic core decomposition. Montresor et al. [35] give a distributed algorithm for deterministic core decomposition and introduce the concept of locality-based bound tightening. Wen et al. [45] propose I/O efficient core decomposition algorithms which only allow node information to be loaded in memory. Khaouid et al. [24] consider deterministic core decomposition of large networks on a single PC. Sariyuce et al. [39] propose incremental k-core decomposition algorithms for dynamic graph data, in which edges are added/deleted on a regular basis. In a similar setting, a distributed k-core decomposition and maintenance algorithms are proposed in [2]. Core decomposition in large temporal graphs is addressed in [47].

[4] Abbreviation using the first letters of the authors' names.

For probabilistic graphs, the generalization of k-core is the notion of (k, η)-core introduced by Bonchi et al. [6], which we discussed in detail throughout the paper. Other notions of cohesive subgraphs are studied in probabilistic setting [18, 36, 55]. [36, 53] focus on the problem of finding k vertex sets with the largest maximal-clique probabilities. Truss decomposition as another notion of a cohesive subgraph has been studied in [18, 55].

Significant research has been done in mining and querying probabilistic graphs. Reachability is addressed in [12, 20, 22, 23]. Shortest paths are studied in [38, 48] and frequent subgraph mining in [10, 37, 52, 54]. Clustering analysis is investigated in [26, 31] and subgraph similarity in [49].

8 CONCLUSIONS

We presented two efficient algorithms, PA and SA, for computing the core decomposition of probabilistic graphs at web scale. An important contribution of this work is the use of Lyapunov Central Limit Theorem in these algorithms to compute tail probabilities for η-degrees. We evaluated our algorithms, and showed that they are efficient and numerically stable. Our algorithms were considerably faster than the-state-of-the-art for large datasets. For datasets larger than biomine, our algorithms PA and SA, were the only algorithms able to run to completion on a consumer grade machine. In particular, PA was able to compute probabilistic core decomposition for uk-2005, arabic-2005, and twitter-2010 in 28 min, 42 min and 2.2 hours, respectively, which is impressive for such large datasets. SA has smaller memory footprint and can produce approximate results of high quality in only a fraction of iterations needed for full completion.

REFERENCES

[1] L. Antiqueira, O. N. Oliveira Jr, L. da Fontoura Costa, and M. D. G. V. Nunes. 2009. A complex network approach to text summarization. *Inf. Sci.* 179, 5 (2009), 584–599.

[2] S. Aridhi, M. Brugnara, A. Montresor, and Y. Velegrakis. 2016. Distributed k-core decomposition and maintenance in large dynamic graphs. In *Proc. DEBS.* ACM, 161–168.

[3] V. Batagelj and M. Zaversnik. 2003. An O (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).

[4] P. Billingsley. 1995. *Mathematical Methods of Statistics* (3 ed.). Wiley.

[5] P. Boldi and S. Vigna. 2004. The webgraph framework I: compression techniques. In *Proc. WWW'04.* ACM, 595–602.

[6] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich. 2014. Core decomposition of uncertain graphs. In *Proc. SIGKDD.* ACM, 1316–1325.

[7] Ch. Borgs, M. Brautbar, J. Chayes, and B. Lucier. 2012. Influence Maximization in Social Networks: Towards an Optimal Algorithmic Solution. *CoRR* abs/1212.0884 (2012).

[8] C. Budak, D. Agrawal, and A. El Abbadi. 2011. Limiting the spread of misinformation in social networks. In *Proc. WWW'11.*

[9] G. Cavallaro. 2010. Genome-wide analysis of eukaryotic twin CX 9 C proteins. *Mol. Biosyst.* 6, 12 (2010), 2459–2470.

[10] Y. Chen, X. Zhao, X. Lin, and Y. Wang. 2015. Towards frequent subgraph mining on single large uncertain graphs. In *Proc. ICDM.* IEEE, 41–50.

[11] J. Cheng, Y. Ke, Sh. Chu, and MT. Özsu. 2011. Efficient core decomposition in massive networks. In *Proc. ICDE.* IEEE, 51–62.

[12] Y. Cheng, Y. Yuan, L. Chen, and G. Wang. 2015. The reachability query over distributed uncertain graphs. In *Proc. ICDCS.* IEEE, 786–787.

[13] H. Cramér. 1946. *Mathematical Methods of Statistics.* PUP.

[14] M. T. Dittrich, G. W. Klau, A. Rosenwald, T. Dandekar, and T. Müller. 2008. Identifying functional modules in protein–protein interaction networks: an integrated exact approach. *BOINFP* 24, 13 (2008), i223–i231.

[15] J. Dong and S. Horvath. 2007. Understanding network concepts in modules. *BMC system biology* 1, 1 (2007), 24.

[16] D. Eppstein, M. Löffler, and D. Strash. 2010. Listing all maximal cliques in sparse graphs in near-optimal time. In *ISAAC.* Springer, 403–414.

[17] A. Goyal, F. Bonchi, and L. V. Lakshmanan. 2010. Learning influence probabilities in social networks. In *Proc. WSDM.* ACM, 241–250.

[18] X. Huang, W. Lu, and L. V. Lakshmanan. 2016. Truss decomposition of probabilistic graphs: Semantics and algorithms. In *Proc. SIGMOD.* ACM, 77–90.

[19] R. Jin, L. Liu, and Ch. Aggarwal. 2011. Discovering highly reliable subgraphs in uncertain graphs. In *Proc. SIGKDD.* ACM, 992–1000.

[20] R. Jin, L. Liu, B. Ding, and H. Wang. 2011. Distance-constraint reachability computation in uncertain graphs. *PVLDB* 4, 9 (2011), 551–562.

[21] D. Kempe, J. Kleinberg, and É. Tardos. 2003. Maximizing the spread of influence through a social network. In *KDD'03.* https://doi.org/10.1145/956750.956769

[22] A. Khan, F. Bonchi, A. Gionis, and F. Gullo. 2014. Fast Reliability Search in Uncertain Graphs.. In *Proc. EDBT.* 535–546.

[23] A. Khan, F. Bonchi, F. Gullo, and A. Nufer. 2018. Conditional Reliability in Uncertain Graphs. *IEEE Trans. Knowl. Data Eng.* (2018).

[24] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. 2015. K-core decomposition of large networks on a single PC. In *Proc. VLDB.* ACM, 13–23.

[25] H. Kobayashi, B.L. Mark, and W. Turin. 2011. *Probability, Random Processes, and Statistical Analysis.* Cambridge University Press.

[26] G. Kollios, M. Potamias, and E. Terzi. 2013. Clustering large probabilistic graphs. *TKDE* (2013).

[27] N. Korovaiko and A. Thomo. 2013. Trust prediction from user-item ratings. *SNAM* 3, 3 (2013), 749–759.

[28] U. Kuter and J. Golbeck. 2010. Using probabilistic confidence models for trust inference in web-based social networks. *TOIT* 10, 2 (2010), 8.

[29] V. Lee, N. Ruan, R. Jin, and Ch. Aggarwal. 2010. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data.* Springer, 303–336.

[30] X. Li, M. Wu, Ch. Kwoh, and S. Ng. 2010. Computational approaches for detecting protein complexes from protein interaction networks: a survey. *BMC genomics* 11, 1 (2010), S3.

[31] L. Liu, R. Jin, Ch. Aggarwal, and Y. Shen. 2012. Reliable clustering on uncertain graphs. In *Proc. ICDM.* IEEE, 459–468.

[32] Y. Liu, J. Lu, H. Yang, X. Xiao, and Zh. Wei. 2015. Towards maximum independent sets on massive graphs. *PVLDB* 8, 13 (2015), 2122–2133.

[33] A. Lyapunov. 1900. Sur une proposition de la théorie des probabilités. *Bulletin de l'Academie Impériale des Sci. de St. Petersbourg* 13 (1900), 359-386.

[34] A. Lyapunov. 1901. Nouvelle forme de la théoreme dur la limite de probabilité. *Mémoires de lâĂŹAcademie Impériale des Sci. de St. Petersbourg* 12 (1901), 1-24.

[35] A. Montresor, F. De Pellegrini, and D. Miorandi. 2013. Distributed k-core decomposition. *IEEE Trans. Parallel Distrib. Syst.* 24, 2 (2013), 288-300.

[36] A. P. Mukherjee, P. Xu, and S. Tirthapura. 2015. Mining maximal cliques from an uncertain graph. In *Proc. ICDE.* IEEE, 243–254.

[37] O. Papapetrou, E. Ioannou, and D. Skoutas. 2011. Efficient discovery of frequent subgraph patterns in uncertain graph databases. In *Proc. EDT.* ACM, 355–366.

[38] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios. 2010. K-nearest neighbors in uncertain graphs. *PVLDB* 3, 1-2 (2010), 997–1008.

[39] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K. Wu, and Ü. V. Çatalyürek. 2016. Incremental k-core decomposition: algorithms and evaluation. *VLDB* 25, 3 (2016), 425–447.

[40] S. B. Seidman. 1983. Network structure and minimum degree. *Social networks* 5, 3 (1983), 269–287.

[41] R. Sharan, I. Ulitsky, and R. Shamir. 2007. Network-based prediction of protein function. *Mol. Syst. Biol.* 3, 1 (2007), 88.

[42] I. G. Shevtsova. 2010. An improvement of convergence rate estimates in the Lyapunov theorem. *Doklady Mathematics* 82, 3 (2010), 862-864.

[43] Y. Tang, X. Xiao, and Y. Shi. 2014. Influence Maximization: Near-Optimal Time Complexity Meets Practical Efficiency. *CoRR* abs/1404.0900 (2014).

[44] J. Ugander, L. Backstrom, C. Marlow, and J. Kleinberg. 2012. Structural diversity in social contagion. *Proc. Natl. Acad. Sci.* 109, 16 (2012), 5962–5966.

[45] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. Yu. 2016. I/O efficient core graph decomposition at web scale. In *Proc. ICDE.* IEEE, 133–144.

[46] T. Wolf, A. Schröter, D. Damian, L. D. Panjer, and T. H. Nguyen. 2009. Mining task-based social networks to explore collaboration in software teams. *IEEE Soft.* 26, 1 (2009), 58–66.

[47] H. Wu, J. Cheng, Y. Lu, Y. Ke, Y. Huang, D. Yan, and H. Wu. 2015. Core decomposition in large temporal graphs. In *Proc. Big Data.* IEEE, 649–658.

[48] Y. Yuan, L. Chen, and G. Wang. 2010. Efficiently answering probability threshold-based shortest path queries over uncertain graphs. In *Proc. DSFAA.* Springer, 155–170.

[49] Y. Yuan, G. Wang, L. Chen, and H. Wang. 2012. Efficient subgraph similarity search on large probabilistic graph databases. *PVLDB* 5, 9 (2012), 800–811.

[50] Zh. Zhang, J. Yu, L. Qin, L. Chang, and X. Lin. 2015. I/O efficient: computing SCCs in massive graphs. *VLDB* 24, 2 (2015), 245–270.

[51] Zh. Zhang, J. Yu, L. Qin, and Z. Shang. 2015. Divide & conquer: I/O efficient depth-first search. In *Proc. SIGMOD.* ACM, 445–458.

[52] Zh. Zou, H. Gao, and J. Li. 2010. Discovering frequent subgraphs over uncertain graph databases under probabilistic semantics. In *Proc. SIGKDD.* ACM, 633–642.

[53] Zh. Zou, J. Li, H. Gao, and Sh. Zhang. 2010. Finding top-k maximal cliques in an uncertain graph. In *Proc. ICDE.* IEEE, 649–652.

[54] Z. Zou, J. Li, H. Gao, and S. Zhang. 2010. Mining frequent subgraph patterns from uncertain graph data. *IEEE Trans. Knowl. Data Eng.* 22, 9 (2010), 1203–1218.

[55] Zh. Zou and R. Zhu. 2017. Truss decomposition of uncertain graphs. *KAIS* 50, 1 (2017), 197–230.

[56] D. Zwillinger and S. Kokoska. 2000. *CRC Standard probability and statistics tables and formulae.* Chapman and Hall/CRC, USA.

[57] D. Zwillinger and S. Kokoska. 2013. *Probability Theory.* Springer-Verlag, London.

Efficient Network Reliability Computation in Uncertain Graphs

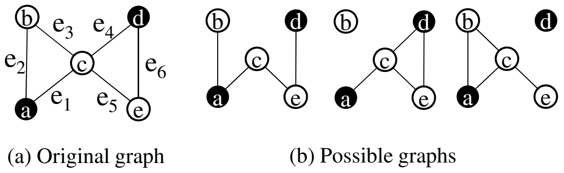
Yuya Sasaki†, Yasuhiro Fujiwara§†, Makoto Onizuka†
†Graduate School of Information Science and Technology, Osaka University, Osaka, Japan
§NTT Software Innovation Center, Tokyo, Japan
sasaki@ist.osaka-u.ac.jp,fujiwara.yasuhiro@lab.ntt.co.jp,onizuka@ist.osaka-u.ac.jp

ABSTRACT

Network reliability is an important metric to evaluate the connectivity among given vertices in uncertain graphs. Since the network reliability problem is known as #P-complete, existing studies have used approximation techniques. In this paper, we propose a new sampling-based approach that efficiently and accurately approximates network reliability. Our approach improves efficiency by reducing the number of samples based on the *stratified sampling*. We theoretically guarantee that our approach improves the accuracy of approximation by using lower and upper bounds of network reliability, even though it reduces the number of samples. To efficiently compute the bounds, we develop an extended BDD, called *S^2BDD*. During constructing the S^2BDD, our approach employs dynamic programming for efficiently sampling possible graphs. Our experiment with real datasets demonstrates that our approach is up to 51.2 times faster than existing sampling-based approach with a higher accuracy.

1 INTRODUCTION

To understand and design our world, we need to model and analyze relationships between objects. Objects and relationships can be modeled by a graph, whose vertices and edges represent the objects and the relationships, respectively. Graph analysis is widely used in many domains, and the *reachability* [8, 34, 37] and *network reliability* [5, 10, 33] are the fundamental research topics in graph analysis. Reachability techniques compute whether there are paths between two *terminals* (i.e., given vertices). On the other hand, network reliability techniques compute a probability that all pairs of terminals are connected in *uncertain graphs*. In an uncertain graph, each edge is associated with an *edge existence probability* to quantify the likelihood that the edge exists in the graph. Network reliability is more generalized than reachability in terms of two aspects (1) a probabilistic value (the reachability is binary) and (2) the number of terminals. Thus, network reliability techniques have two benefits over reachability techniques. First, we can handle the inherent *uncertainty* of relationships in the real-world by modeling the uncertainty as the edge existence probability [1, 23]. Second, we can flexibly specify arbitrary numbers of terminals. From the above two benefits, the network reliability can be widely used for the uncertain graph analysis [6, 36] and many practical applications [20]. For example, protein-protein interaction networks can be modeled by uncertain graphs since protein interactions are not always established due to the sensitivity to conditions [4, 17]. In such protein-protein interaction networks, analysts evaluate the network reliability among several proteins as the strengths of the relationships to elucidate the functions of proteins. The network reliability is also used in many domains such as communication networks [5, 29] and urban planning [13].



(a) Original graph (b) Possible graphs

Figure 1: Uncertain graph

Unfortunately, the computation cost of the network reliability is significantly large because it is #P-complete problem [33]. The high complexity of #P-complete is caused by the fact that the computation of the network reliability inherently requires to enumerate all *possible graphs* which have the same set of vertices and an arbitrary subset of the edges without their probabilities. Each possible graph has its probability computed from the existence probabilities of its edges. A set of possible graphs is logically equivalent with its original uncertain graph. To compute the network reliability, we sum up the probabilities of all possible graphs in which all the terminals are connected.

We explain an example of computation of the network reliability by using Figure 1. This figure shows an original uncertain graph and three examples of its possible graphs. The black vertices represent terminals. Let us assume that each edge has 0.7 as its existence probability. Since these possible graphs have four existent and two non-existent edges, their probabilities are 0.0216 (i.e., $0.7^4 \cdot (1 - 0.7)^2$). All these terminals are connected only in the left and middle possible graphs. Thus, their probabilities are added to the network reliability.

Problem Definition and Technical Overview

We approximate the network reliability since the computation cost of the network reliability is significantly large due to #P-complete problem. In this paper, we consider the problem of computing the approximate network reliability by sampling. We formally define the problem as follows.

Problem definition: **(Approximate network reliability)**. Given an uncertain graph $\mathcal{G}$, a set of terminals $\mathbb{T}$, and the number of samples s, we efficiently compute the approximate network reliability $\hat{R}[\mathcal{G}, \mathbb{T}]$.

The computation cost of sampling becomes considerable as the number of samples increases. To efficiently approximate the network reliability, we reduce the number of samples with keeping a high accuracy. Our challenges are (1) how to reduce the number of samples with a theoretical guarantee of the accuracy and (2) how to practically achieve the theoretical results from the first challenge. As for the first challenge, we extend the *stratified sampling* [32], which increases the accuracy of an estimated value by using the lower and upper bounds of the value. We first prove a theorem that we reduce the number of samples without sacrificing the accuracy of approximation.

We can reduce the number of samples in accordance with the theoretical results. The theoretical results have two requirements; (1) to efficiently compute the approximate network reliability, we need to efficiently obtain the tight lower and upper bounds of the network reliability and (2) to guarantee the approximation of accuracy, we need to sample possible graphs from the set of possible graphs that are not used to compute the bounds. There are no trivial techniques to effectively achieve them. Therefore, we develop an extended *binary decision diagram*, which we call *scalable and sampling BDD* (*S^2BDD* for short). The S^2BDD enables preferentially searching for possible graphs in which terminals are connected/disconnected. The connected and disconnected possible graphs are used for computing the lower and upper bounds. Our approach employs dynamic programming during constructing the S^2BDD for efficiently sampling the possible graphs. It enables avoiding sampling possible graphs from the set of possible graphs that are used to compute the bounds.

Furthermore, our approach becomes more efficient by reducing the size of graphs. Thus, we propose an extension technique of our approach which uses 2-edge connected components [7]. The extension technique prunes vertices and edges that do not affect the network reliability, decomposes the graph to several subgraphs, and transforms the subgraphs into a smaller graphs. It efficiently reduces the vertices and edges involved in the computation while preserving the network reliability.

Contributions and Organization

To the best of our knowledge, our approach is the first solution to achieve both high efficiency and accuracy to compute the network reliability. Our approach has the following attractive characteristic.

- Our approach improves the efficiency to compute an approximate network reliability by reducing the number of samples. The extension technique effectively reduces the size of graphs while preserving the network reliability.
- Our approach outputs more accurate network reliability than the existing approaches. We theoretically guarantee that our approach improves the accuracy of approximation, even though it reduces the number of samples.
- Our approach computes the exact answer for small-scale graphs due to the S^2BDD though the existing sampling-based approach cannot compute the exact answer.
- Our approach can be used to improve the performances on uncertain graph analyses [6, 18, 22] in terms of both accuracy and efficiency because many algorithms compute the network reliability by sampling techniques.

The remainder of this paper is organized as follows. Section 2 introduces related work. Section 3 then describes the preliminaries. Sections 4 and 5 present our approach and an extension technique for our approach, respectively. Section 6 describes algorithms of our approach with the extension. Section 7 shows the results obtained from the experiments, and Section 8 concludes the paper.

2 RELATED WORK

Querying and mining uncertain graphs have recently attracted much attention in the database and data mining research communities. We review some relevant works related to the network reliability problem.

Network reliability: For computing the network reliability, several approaches have been proposed such as cut-based approach and BDD-based approach. The cut-based [3, 15, 25] approach enumerates all cuts which are divides the terminals and then computes the network reliability by using the set of cuts. Harris and Srinivasan [15] proposed theoretical result to obtain the lower bound of network relaibility based on cuts. However, they do not mention how to efficiently obtain the cuts. The BDD-based approach is more efficient than the cut-based approach. The BDD-based approach [14, 26, 35] effectively avoids enumerating all possible graphs without sacrificing the exactness of the network reliability. However, it cannot be applicable to large graphs due to the large memory usage. The BDD-based approach first constructs a BDD, and then obtains the possible graphs in which terminals are connected by traversing the BDD. Recent work has shown that the BDD-based approach can be applied only to graphs with 100–200 edges because of limitations of memory space [14, 26]. The state-of-the-art library TdZDD[1] also can only be applied to very small-scale graphs. Herrmann and Soh [16] proposed a memory-efficient BDD that computes the network reliability by constructing a BDD and deleting unnecessary parts of it during the process. We partially adopt their idea to reduce the memory usage. There are several preprocessing and indexing techniques to efficiently compute the network reliability (and similar problems) [12, 24]. These techniques remove redundant parts of graphs, which have similar idea of our extension technique. However, these techniques cannot directly apply to k-terminal reliability. To the best of our knowledge, there has been no prior work on approximating the network reliability with BDD.

Reachability query in uncertain graphs: The reachability in uncertain graphs is a special type of network reliability (called s-t network reliability) [2]. Jin et al. [19] proposed a distance-constraint reachability query in uncertain graphs, which answers the probability that the distance from one vertex to another is less than or equal to a threshold. They proposed approximate algorithms as solutions to this problem. The approximate algorithms use unequal sampling techniques [31], and achieves higher accuracy than Monte Carlo sampling. Cheng et al. [9] proposed an algorithm to compute the reachability in distributed environments. The algorithm reduces the size of graphs without sacrificing the exactness of the result before computing the reachability. It divides the graph into several subgraphs and computes probabilities of the subgraphs in distributed environments. The algorithm is only applicable to directed acyclic graphs. While these algorithms [9, 19] deal with uncertain graphs, their objective is to compute reachability and their algorihms cannot be applied to computing the network reliability.

Other problems with uncertain graphs: Many existing works in uncertain graphs use the network reliability as the metric to evaluate the connectivity among vertices. The efficiency and accuracy of their algorithms depend on those of the sampling techniques. Although they use the sampling technique to compute the network reliability, they have not proposed efficient sampling techniques. Jin et al. [18] proposed an algorithm for finding reliable subgraphs in which the vertices are connected with a higher probability than a given threshold. Ceccarello et al. [6] proposed clustering techniques for uncertain graphs. The technique uses the network reliabilities between vertices as distances between them. Khan et al. [22] proposed a reliability search

[1] https://github.com/kunisura/TdZdd

Table 1: Notations

Symbol	Meaning
$\mathcal{G}$	Uncertain graph
$\mathbb{V}$	Set of vertices
$\mathbb{E}$	Set of edges $e = (v, v')$
$p(e)$	Edge existence probability of e
G_p	Possible graph
$\mathbb{E}_p$	Set of edges in G_p
$Pr[G_p]$	Existence probability of G_p
$\mathcal{G}_{\mathbb{E}}$	Intermediate graph
$\mathbb{E}_\exists$	Set of existent edges in $\mathcal{G}_{\mathbb{E}}$
$\mathbb{E}_\neg$	Set of non-existent edges in $\mathcal{G}_{\mathbb{E}}$
$Pr[\mathcal{G}_{\mathbb{E}}]$	Existence probability of $\mathcal{G}_{\mathbb{E}}$
$\mathbb{T}$	Set of terminals
$R[\mathcal{G}, \mathbb{T}]$	Network reliability of $\mathcal{G}$ for $\mathbb{T}$
$\hat{R}[\mathcal{G}, \mathbb{T}]$	Approximate network reliability of $\mathcal{G}$ for $\mathbb{T}$
k	The number of terminals
w	Maximum size of BDD
$\mathbb{F}_l$	Set of frontiers at layer l
$\lvert \cdot \rvert$	The number of elements in a set

that returns a set of vertices that are connected from given vertices with a higher probability than the threshold. These studies have different purposes, but they use the Monte Carlo sampling to compute the network reliability. Our approach can be used to improve their performances in terms of both accuracy and efficiency instead of using the Monte Carlo sampling.

3 PRELIMINARIES

As preliminaries of our approach, we explain uncertain graph and network reliability. Table 1 summarizes the notations.

3.1 Uncertain graph

Let $\mathcal{G} = (\mathbb{V}, \mathbb{E}, p)$ be a connected and undirected uncertain graph, where $\mathbb{V}$ is a set of vertices, $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ is a set of *uncertain* edges, and $p : \mathbb{E} \longrightarrow (0, 1]$ is a function that determines the edge existence probability $p(e)$ of uncertain edge $e \in \mathbb{E}$ in the graph. We denote edge $e \in \mathbb{E}$ between v and v' as $e = (v, v')$. A state of uncertain edge e is *existent* with a probability $p(e)$ or *non-existent* with a probability $(1-p(e))$. We assume that edge existence probabilities of different edges are independent of one another [6, 19].

A *possible graph* $G_p = (\mathbb{V}, \mathbb{E}_p)$ is a graph that contains a set of vertices and a subset of edges of $\mathcal{G}$ without their edge existence probabilities. Edges in $\mathbb{E} \backslash \mathbb{E}_p$ are non-existent in the possible graph. Although edges in possible graphs have no probabilities, the possible graphs themselves have existent probabilities. The existent probability $Pr[G_p]$ of possible graph G_p is as follows:

$$Pr[G_p] = \prod_{e \in \mathbb{E}_p} p(e) \cdot \prod_{e \in \mathbb{E} \backslash \mathbb{E}_p} (1 - p(e)).$$

The total number of the possible graphs of $\mathcal{G}$ is $2^{|\mathbb{E}|}$ because each edge is either existent or non-existent. We define $\mathbb{W}^{\mathcal{G}}$ as all possible graphs obtained from $\mathcal{G}$.

We define an *intermediate graph* $\mathcal{G}_{\mathbb{E}}(\mathbb{E}_\exists, \mathbb{E}_\neg)$, which is an uncertain graph with the set of existent edges $\mathbb{E}_\exists$, the set of non-existent edges $\mathbb{E}_\neg$, and the set of uncertain edges $\mathbb{E} \backslash (\mathbb{E}_\exists \cup \mathbb{E}_\neg)$. The existent probability $Pr[\mathcal{G}_{\mathbb{E}}(\mathbb{E}_\exists, \mathbb{E}_\neg)]$ of the intermediate graph $\mathcal{G}_{\mathbb{E}}(\mathbb{E}_\exists, \mathbb{E}_\neg)$ is as follows:

$$Pr[\mathcal{G}_{\mathbb{E}}(\mathbb{E}_\exists, \mathbb{E}_\neg)] = \prod_{e \in \mathbb{E}_\exists} p(e) \cdot \prod_{e \in \mathbb{E}_\neg} (1 - p(e)).$$

We simply use $Pr[\mathcal{G}_{\mathbb{E}}]$ as $Pr[\mathcal{G}_{\mathbb{E}}(\mathbb{E}_\exists, \mathbb{E}_\neg)]$. We define $\mathbb{W}^{\mathcal{G}_{\mathbb{E}}}$ as all possible graphs obtained from $\mathcal{G}_{\mathbb{E}}$. The total number of the possible graphs of $\mathcal{G}_{\mathbb{E}}(\mathbb{E}_\exists, \mathbb{E}_\neg)$ is $2^{|\mathbb{E} \backslash (\mathbb{E}_\exists \cup \mathbb{E}_\neg)|}$. We define that vertices are connected in intermediate graphs if there are paths among the vertices by existent edges, and vertices are disconnected if there are no paths among the vertices by existent and uncertain edges. Note that it is unsure to be connected or disconnected even if there are paths among the vertices by uncertain edges.

3.2 Network reliability

The network reliability is computed by summing up the probabilities of all possible graphs in which all terminals (a subset of vertices) are connected. The definition is as follows:

Definition 1 (Network reliability). *Given a set of k terminals $\mathbb{T}$ and an uncertain graph $\mathcal{G}$, the network reliability $R[\mathcal{G}, \mathbb{T}]$ is*

$$R[\mathcal{G}, \mathbb{T}] = \textstyle\sum_{G_p \in \mathbb{W}^{\mathcal{G}}} I(G_p, \mathbb{T}) \cdot Pr[G_p], \qquad (1)$$

where G_p denotes a possible graph, and $I(G_p, \mathbb{T})$ is an indicator function that returns one if all terminals in $\mathbb{T}$ are connected in G_p, and returns zero, otherwise.

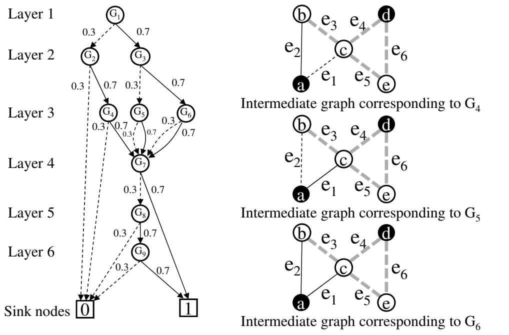
We denote by $\hat{R}[\mathcal{G}, \mathbb{T}]$ the approximate network reliability. We simply use R and $\hat{R}$ as $R[\mathcal{G}, \mathbb{T}]$ and $\hat{R}[\mathcal{G}, \mathbb{T}]$ for the given uncertain graph and terminals, respectively.

The network reliability with k terminals is called the *k-terminal reliability*, and it is known as the most generalized network reliability [14]. The network reliability problem is #P-complete [33]. Planar graphs can be more efficiently solved than general graphs, but it is also #P-complete [30]. Therefore, it has no polynomial time algorithm unless $P = NP$.

BDD [14] and sampling [19] are main techniques to compute the network reliability. BDD-based approach can compute the exact answer in small-scale graphs, while sampling-based appraoch can compute approximate answers in large-scale graphs.

3.2.1 Binary decision diagram. A BDD $\mathcal{D} = (\mathbb{N}, \mathbb{A})$ is a directed acyclic graph with sets of nodes $\mathbb{N}$ and arcs $\mathbb{A}$[2]. Figure 2(a) shows the BDD to compute the network reliability of the original graph in Figure 1. Nodes in the BDD correspond to intermediate graphs, and arcs in the BDD correspond to existent/non-existent edges. The BDD has a single node that has no incoming arcs, called the *root node* (node G_1 in Figure 2(a)). Each node has two outgoing arcs, called the *0-arc* and *1-arc* (represented by dashed and solid arrows in Figure 2(a), respectively). 0-arcs and 1-arcs indicate that edges are non-existent and existent in the uncertain graph, respectively. Each arc is associated with a *weight* that represents the existent or non-existent probability of the edge. We define *layer* l (≥ 1) as the depth from the root node. The nodes at layer l of the BDD correspond to the intermediate graphs whose edges $e_1, \ldots, e_{l-1}$ are existent/non-existent and the other edges $e_l, \ldots, e_{|\mathbb{E}|}$ are uncertain. The BDD has special nodes that have no outgoing arcs, called *sink nodes*. The sink nodes are of two types, called *1-sink* and *0-sink* (represented by rectangles with labels 1 and 0 in Figure 2(a), respectively). If the terminals in the intermediate graph are connected and disconnected, the arcs point at the 1-sink and 0-sink, respectively. We can obtain intermediate graphs in which terminals are connected by traversing the BDD from the root node to the 1-sink.

[2] To avoid confusion, we use the terms "vertex" and "edge" to refer to a vertex and an edge in an uncertain graph, respectively, and "node" and "arc" to refer to a vertex and an edge in a BDD, respectively.



Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
Layer 6
Sink nodes
G1
G2
G3
G4
G5
G6
G7
G8
G9
0.3
0.7
0
1
b
d
c
a
e
e1
e2
e3
e4
e5
e6
Intermediate graph corresponding to G4
Intermediate graph corresponding to G5
Intermediate graph corresponding to G6

(a) BDD (b) Intermediate graphs at layer 3

Figure 2: BDD for the original graph on Figure 1(a).

To construct the BDD, the *frontier-based method* is a common procedure [21, 26]. This method first orders edges $(e_1, \ldots, e_{|\mathbb{E}|})$. It generates the nodes on layer $l + 1$ by setting the states of e_l when a BDD is already constructed until layer l. In the frontier-based method, a vertex that has both existent/non-existent and uncertain edges are called a *frontier* f, and we denote by $\mathbb{F}_l$ the set of frontiers at layer l. Figure 2(b) shows intermediate graphs after processing e_1 and e_2, where solid black, dashed black, and dashed gray lines denote existent, non-existent, and uncertain edges, respectively. These intermediate graphs correspond to G_4, G_5, and G_6 in the BDD from the top, respectively. Vertices b and c are frontiers because they have both existent/non-existent and uncertain edges. Note that nodes at the same layer l have the same set of frontiers $\mathbb{F}_l$. The frontier-based method maintains several attributes on only the frontiers (e.g., the number of uncertain edges and the number of terminals connected to the frontiers). It merges the nodes if the attributes are the same. Thus, the frontier-based method can effectively reduce the number of nodes.

The size of the BDD is defined by the number of nodes in the BDD [14]. Generally, it exponentially increases as the number of edges in the uncertain graphs increases. As the size of the BDD increases, both of the computation cost and the memory usage increase. Thus, it is hard to compute the exact network reliability.

3.2.2 Sampling. Sampling is a basic approach for computing the approximate network reliability [9, 18, 19]. Given the number of samples s, the sampling-based approach repeats the following procedures s times: (1) picking a possible graph of $\mathcal{G}$ as a sample, G_{p_i} $(1 \leq i \leq s)$ according to the probabilities $Pr[G_{p_i}]$ from $\mathbb{W}^{\mathcal{G}}$ and then (2) computing whether all the terminals are connected or not in G_{p_i}. The time complexity of the sampling-based approach is $O(s \cdot (|\mathbb{V}| + |\mathbb{E}|))$. This is because it requires $O(|\mathbb{E}|)$ time to determine the states of all edges and $O(|\mathbb{V}| + |\mathbb{E}|)$ time to compute the connectivity by a depth first search for each sample.

The accuracy of the sampling-based approach is evaluated by its variance. Since the sampling-based approach is a randomized algorithm [28], the average network reliability is most likely to be closest to the exact network reliability. A small variance indicates a small rate of error (i.e., high accuracy). Note that *unbiased sampling* is necessary that samples possible graphs according to their probabilities for guaranteeing the theoretical variance. As the number of samples increases, the variance decreases but the computation cost increases. Therefore, there is a trade-off between the accuracy and the computation cost.

The *stratified sampling* is known as a successful method in the field of statistics [32]. The stratified sampling divides the population into subgroups and individually picks samples from each subgroup. The variance of the estimated value for the whole population are the sum of the variances of the estimated values for individual subgroups. Let L be the number of subgroups and R_i be the estimated total probabilities of possible graphs for subgroup i. The estimated network reliability is computed by summing up the total probabilities for the subgroups as follows:

$$\hat{R} = \textstyle\sum_{i=1}^{L} \hat{R}_i.$$

The variance is the sum of the individual variances for the subgroups as follows:

$$Var[\hat{R}] = \textstyle\sum_{i=1}^{L} Var[\hat{R}_i].$$

When we compute the exact values for the subgroups, the variances of the estimated network reliability for the subgroup become zero. Thus, when we compute the exact values for the subgroups, the variance of the estimated network reliability for the whole population decreases.

4 OUR APPROACH

In this paper, we solve the problem of the approximate network reliability. Section 4.1 provides an overview of our approach. Section 4.2 explains how to reduce the number of samples. Section 4.3 presents our extended BDD S^{2}BDD.

4.1 Overview

Our approach efficiently and accurately computes the approximate network reliability. We achieve high efficiency and accuracy with the following ideas:

- **Reduction of the number of samples**: Our approach significantly reduces the number of samples with keeping a high accuracy of approximation by using the lower and upper bounds of the network reliability.
- **Efficient computation of the bounds of network reliability**: We develop the S^{2}BDD to efficiently compute the bounds of the network reliability.
- **Dynamic programming**: During constructing S^{2}BDD, we employ dynamic programming for efficiently sampling possible graphs.

Our approach reduces the number of samples in accordance with the stratified sampling. We theoretically guarantee that the number of samples becomes small as the lower and upper bounds become tight without sacrificing the accuracy of approximation. We prove it in two representative estimators; Monte Carlo and Horvitz-Thompson estimators [32].

For achieving the theoretical result, we compute the lower and upper bounds by constructing the S^{2}BDD. We specify the maximum size w of S^{2}BDD for avoiding a large cost to construct the S^{2}BDD. Our approach deletes nodes on the S^{2}BDD when its size exceeds w. To effectively delete nodes, we define a heuristic function for preferentially keeping *high-priority* nodes in the S^{2}BDD; the priorities are computed from the possibilities of improving the bounds. The S^{2}BDD enables efficiently computing the bounds because nodes preferentially point at sink nodes.

For efficiently sampling possible graphs, our approach employs dynamic programming during constructing the S^{2}BDD. We can straightforwardly employ dynamic programming for sampling because sampling possible graphs from intermediate graphs is a sub problem of sampling possible graphs from the original uncertain graph. We also use the *stratified random sampling* for determining the number of samples for each sub problem. The stratified random sampling divides the set of possible graphs into subgroups and samples possible graphs from each subgroup.

4.2 Reducing the number of samples

In this section, we theoretically prove that our approach reduces the number of samples while keeping a high accuracy in accordance with the stratified sampling [11, 27]. As we mentioned in Section 4.3.3, the accuracy of sampling is evaluated by the variance of the estimated network reliability. Since the stratified sampling reduces the variance of the estimated network reliability, we can reduce the number of samples without sacrificing the accuracy of approximation.

To apply the stratified sampling, we divide the set $\mathbb{W}^{\mathcal{G}}$ of possible graphs into three subgroups $\mathbb{W}_c^{\mathcal{G}}$, $\mathbb{W}_d^{\mathcal{G}}$, and $\mathbb{W}_u^{\mathcal{G}}$. $\mathbb{W}_c^{\mathcal{G}}$ and $\mathbb{W}_d^{\mathcal{G}}$ include the sets of only possible graphs in which terminals are connected and disconnected, respectively. $\mathbb{W}_u^{\mathcal{G}}$ includes the set of possible graphs that are not included in $\mathbb{W}_c^{\mathcal{G}}$ and $\mathbb{W}_d^{\mathcal{G}}$. Let p_c and p_d be the sum of the probabilities of possible graphs in $\mathbb{W}_c^{\mathcal{G}}$ and $\mathbb{W}_d^{\mathcal{G}}$, respectively. Hence, from Definition 1, the upper and lower bounds are given as follows:

$$\begin{aligned} R &= \textstyle\sum_{G_p \in \mathbb{W}_c^{\mathcal{G}}} Pr[G_p] + \sum_{G_p \in \mathbb{W}_u^{\mathcal{G}}} I(G_p, \mathbb{T}) Pr[G_p] \\ &= p_c + \textstyle\sum_{G_p \in \mathbb{W}_u^{\mathcal{G}}} I(G_p, \mathbb{T}) Pr[G_p] \\ &\geq p_c. \\ R &= 1 - \textstyle\sum_{G_p \in \mathbb{W}_d^{\mathcal{G}}} Pr[G_p] - \sum_{G_p \in \mathbb{W}_u^{\mathcal{G}}} (Pr[G_p] - I(G_p, \mathbb{T}) Pr[G_p]) \\ &= 1 - p_d - \textstyle\sum_{G_p \in \mathbb{W}_u^{\mathcal{G}}} (Pr[G_p] - I(G_p, \mathbb{T}) Pr[G_p]) \\ &\leq 1 - p_d. \end{aligned}$$

Consequently, we have $p_c \leq R \leq 1 - p_d$. We reduce the number of sample by using the lower bound p_c and upper bound $1 - p_d$.

The variance also depends on estimators. In our approach, we exploit two representative estimators; Monte Carlo estimator and Horvitz-Thompson estimator. The Monte Carlo estimator is a basic technique for computing the average values of the samples. On the other hand, the Horvitz-Thompson estimator is unequal probability estimator, which provides smaller variance than the Monte Carlo estimator under sampling without replacement. We explain how to reduce the number of samples in the two estimators with keeping a high accuracy.

Monte Carlo estimator: The Monte Carlo estimator for R is:

$$\hat{R} = \frac{\sum_{i=1}^{s} I(G_{p_i}, \mathbb{T})}{s}.$$

The variance is computed by the following equation [11]:

$$Var[\hat{R}] = \frac{R(1-R)}{s}.$$

Because the random sampling is unbiased, i.e., $E(\hat{R}) = R$, the variance can be simply written as follows [27]:

$$Var[\hat{R}] = \frac{R(1-R)}{s} \approx \frac{\hat{R}(1-\hat{R})}{s}. \tag{2}$$

Let $Var[\hat{R}]'$ be the variance using the upper and lower bounds. $Var[\hat{R}]'$ is computed in accordance with the stratified sampling as follows [11, 27]:

$$Var[\hat{R}]' = \frac{(\hat{R}-p_c)(1-p_d-\hat{R})}{s}. \tag{3}$$

From Equations (2) and (3), we obtain the following equation:

$$\frac{\hat{R}(1-\hat{R})}{s} \geq \frac{(\hat{R}-p_c)(1-p_d-\hat{R})}{s}. \tag{4}$$

Therefore, we have $Var[\hat{R}] \geq Var[\hat{R}]'$. From Equation (4), we obtain the following theorem:

Theorem 1. *Given the number of samples s, the lower bound p_c, and the upper bound $1 - p_d$, the variance of network reliability by using Monte Carlo estimator with s' ($\leq s$) samples is less than and equal to that with s samples if s' is computed by the following equations:*

$$s' = \begin{cases} \lfloor s(1-p_d) \rfloor. & (p_c = 0) \\ \lfloor s(1-p_c) \rfloor. & (p_d = 0) \\ \lfloor s(1 - 4 \cdot p_c(1-p_c)) \rfloor. & (p_c = p_d) \\ \lfloor s(1 - 4 \cdot p_c(1-p_d)) \rfloor. & (p_c < p_d) \\ \lfloor s(1 - \min(4p_c(1-p_c), & \\ \quad 4(p_c(1-p_d) + (p_d - p_c))) \rfloor. & (p_c > p_d) \end{cases}$$

Proof: From Equation (4), we have the following equation such that the variance with s samples is equal to that with s' samples by using the lower and upper bounds:

$$\frac{(p_c-\hat{R})(1-p_d-\hat{R})}{s'} = \frac{\hat{R}(1-\hat{R})}{s}$$

Then, s' is computed as follows:

$$\begin{aligned} s' &= s \cdot \frac{(\hat{R}-p_c)(1-p_d-\hat{R})}{\hat{R}(1-\hat{R})} \\ &= s \cdot \left(1 - \frac{p_c(1-\hat{R}) + p_d(\hat{R}-p_c)}{\hat{R}(1-\hat{R})}\right) \end{aligned} \tag{5}$$

However, we cannot compute $\hat{R}$ before sampling s possible graphs. Therefore, we remove $\hat{R}$ from Equation (5) by dividing the patterns of p_c and p_d. First, if $p_c = 0$, s' is computed as follows:

$$s\left(1 - \frac{p_d \hat{R}}{\hat{R}(1-\hat{R})}\right) \leq s(1-p_d).$$
$$s' = \lfloor s(1-p_d) \rfloor.$$

Second, if $p_d = 0$, s' is computed as follows:

$$s\left(1 - \frac{p_c(1-\hat{R})}{\hat{R}(1-\hat{R})}\right) \leq s(1-p_c).$$
$$s' = \lfloor s(1-p_c) \rfloor.$$

Third, if $p_c = p_d$, s' is computed as follows:

$$s\left(1 - \frac{p_c(1-\hat{R}) + p_c(\hat{R}-p_c)}{\hat{R}(1-\hat{R})}\right) \leq s(1 - 4p_c(1-p_c)). \tag{6}$$
$$s' = \lfloor s(1 - 4p_c(1-p_c)) \rfloor.$$

In Equation (6), the maximum value of $\hat{R}(1-\hat{R})$ is 0.25. Thus, we substitute 0.25 for $\hat{R}(1-\hat{R})$ in the denominator. Fourth, if $p_c < p_d$, s' is computed as follows:

$$s\left(1 - \frac{p_c(1-\hat{R}) + p_d(\hat{R}-p_c)}{\hat{R}(1-\hat{R})}\right) \leq s(1 - 4p_c(1-p_d)).$$
$$s' = \lfloor s(1 - 4p_c(1-p_d)) \rfloor.$$

Finally, if $p_c > p_d$, s' is computed as follows:

$$s\left(1 - \frac{p_c(1-\hat{R}) + p_d(\hat{R}-p_c)}{\hat{R}(1-\hat{R})}\right) \leq s(1 - 4p_c(1-p_c)).$$
$$s\left(1 - \frac{p_c(1-\hat{R}) + p_d(\hat{R}-p_c)}{\hat{R}(1-\hat{R})}\right) \leq s(1 - 4(p_c(1-p_c) + (p_d - p_c)).$$
$$s' = \lfloor s(1 - \min(4p_c(1-p_c), 4(p_c(1-p_d) + (p_d - p_c)))) \rfloor. \tag{7}$$

In Equation (7), the minimum s' depends on the values of p_c and p_d. Consequently, we have that $s' \leq s$ for all patterns of p_c and p_d. □

Horvitz-Thompson estimator: The Horvitz-Thompson estimator for R is:

$$\hat{R} = \frac{\sum_{i=1}^{s} Pr[G_{p_i}] \cdot I(G_{p_i}, \mathbb{T})}{\pi_i},$$

where $\pi_i = 1 - (1 - Pr[G_{p_i}])^s$. The variance is:

$$Var[\hat{R}] = \sum_{i=1}^{s} \left(\frac{1-\pi_i}{\pi_i}\right) I(G_{p_i}, \mathbb{T}) Pr[G_{p_i}]^2$$
$$+ \sum_{i}^{s} \sum_{j, i \neq j}^{s} \left(\frac{\pi_{ij} - \pi_i \pi_j}{\pi_i \pi_j}\right) I(G_{p_i}, \mathbb{T}) I(G_{p_j}, \mathbb{T}) Pr[G_{p_i}] Pr[G_{p_j}],$$

where $\pi_{ij} = 1 - (1 - Pr[G_{p_i}])^s - (1 - Pr[G_{p_j}])^2 + (1 - Pr[G_{p_i}] - Pr[G_{p_j}])^s$. The variance is simplified as follows [19]:

$$Var[\hat{R}] = \frac{R(1-R)}{s} - \frac{\sum_{i=1}^{s}(s-1)I(G_{p_i}, \mathbb{T})Pr[G_{p_i}]^2}{2s}. \tag{8}$$

The variance using the lower and upper bounds is computed in accordance with the stratified sampling as follows:

$$Var[\hat{R}]' = \frac{(\hat{R}-p_c)(1-p_d-\hat{R})}{s} - \frac{\sum_{i=1}^{s}(s-1)I(G_{p_i}, \mathbb{T})Pr[G_{p_i}]^2}{2s}. \tag{9}$$

Theorem 2. *Given the number of samples s, the lower bound p_c, and the upper bound $1 - p_d$, the variance of network reliability by using Horvits-Thompson estimator with s' ($\leq s$) samples is less than and equal to that with s samples where s' is equal to the number of samples in Monte Carlo estimator in 1.*

Proof: From Equations (8) and (9), we have the following equation:

$$\frac{(\hat{R}-p_c)(1-p_d-\hat{R})}{s'} - \frac{\sum_{i=1}^{s}(s'-1)I(G_{p_i}, \mathbb{T})Pr[G_{p_i}]^2}{2s'}$$
$$= \frac{\hat{R}(1-\hat{R})}{s} - \frac{\sum_{i=1}^{s}(s-1)I(G_{p_i}, \mathbb{T})Pr[G_{p_i}]^2}{2s}.$$

The values of the right are the same because the estimator is unbiased. The proof for this follows Theorem 1. □

Our approach reduces the number of samples in accordance with Theorems 1 and 2. As a result, our approach is more efficient than the existing sampling-based approach.

4.3 Scalable and Sampling BDD: S^{2}BDD

We can reduce the number of samples by using the lower and upper bounds of network reliability. To efficiently obtain the bounds, we develop the S^{2}BDD. We efficiently search for the possible graphs in which terminal are connected and disconnected with high probabilities by constructing the S^{2}BDD. Furthermore, during constructing the S^{2}BDD, we sample possible graphs that are not used to compute the bounds, which is the requirement of stratified sampling. Our approach uses S^{2}BDD for both computing the bounds of network reliability and sampling possible graphs.

We design the S^{2}BDD to effectively reduce its size. The S^{2}BDD keeps a single layer and sink nodes while ordinary BDD contains all layers. This idea is based on the observation that the layer $l - 1$ is unnecessary after constructing the next layer l to both construct the layer $l + 1$ and obtain the bounds. We first define the S^{2}BDD and then explain how to construct it.

Definition 2. *Let $\mathbb{N}_l$ be a set of nodes at layer l. S^{2}BDD consists $\mathbb{N}_l$, the 1-sink, and the 0-sink. The S^{2}BDD maintains the following attributes on node $n \in \mathbb{N}$:*

- *p_n: the probability of the intermediate graph corresponding to node n.*
- *$\{c_{n,f}\}$ for all $f \in \mathbb{F}_l$: an identifier of connected component. If frontiers f and $f' \in \mathbb{F}_l$ are connected by existent edges, $c_{n,f}$ and $c_{n,f'}$ share the same identifier.*
- *$\{d_{n,f}\}$ for all $f \in \mathbb{F}_l$: the sum of the numbers of uncertain edges connected to the frontiers such that $\{f' \in \mathbb{F}_l | c_{n,f} = c_{n,f'}\}$.*
- *$\{t_{n,f}\}$ for all $f \in \mathbb{F}_l$: the number of the terminals that are connected to f by existent edges.*

The 1-sink and 0-sink maintain the probabilities p_c and p_d that terminals are connected and disconnected, respectively.

For example, in Figure 2, S^{2}BDD contains third and sink layers but does not contains first and second layers.

To construct an S^{2}BDD, we process edge e_l and generate the set of nodes $\mathbb{N}_{next}$ at layer $l + 1$. The construction method comprises four procedures; *generating*, *merging*, *deleting*, and *sampling*. The following sections explain these procedures in details.

4.3.1 Generating and Merging Procedures. The BDD-based approach uses the generating and merging procedures to construct the BDD. We extend these procedures to effectively compute the bounds without sacrificing the exactness of the network reliability. For extending the generating and merging procedures, we capture the feature of computing the network reliability such that we can skip the computation of nodes when we obtain the probabilities p_c and p_d exactly.

We first explain the generating procedure. The generating procedure sets the state of edge e_l (recall that arcs at layer l in the BDD corresponding to e_l) and then generates the set of new nodes $\mathbb{N}_{next}$ at layer $l + 1$. As the same as the traditional procedure, we generate two new nodes at layer $l + 1$ from every node at layer l according to the state of e_l. We set the attributes on the new nodes (i.e., p_n, $\{c_{n,f}\}$, $\{d_{n,f}\}$, and $\{t_{n,f}\}$). More specifically, p_n is set as $p_n \cdot p(e_l)$ when e_l is existent and set as $p_n \cdot (1 - p(e_l))$ when e_l is non-existent. $\{c_{n,f}\}$, $\{d_{n,f}\}$, and $\{t_{n,f}\}$ are computed from attributes of frontiers on nodes at layer l by merging attributes of frontiers and creating new frontiers. If all the terminals in the intermediate graph are connected, we add its probability to p_c, and if they are disconnected, we add its probability to p_d.

If we determine whether or not terminals are connected/disconnected with processing a smaller number of edges, we can obtain the tight bounds of the network reliability earlier. Let n, n', $\mathbb{F}$, and $\mathbb{F}'$ be the new node at layer $l + 1$, the node before setting e_l of n at layer l, the sets of frontier at layers $l + 1$ and l, respectively. We determine whether or not terminals are connected/disconnected based on following lemmas:

Lemma 4.1. *All the terminals $t \in \mathbb{T}$ are connected if the attributes of the frontiers satisfy one of the following conditions:*
Condition 1: *edge $e_l = (v, v')$ is existent, for $t_{n,f} = k$, $\exists f \in \mathbb{F}$.*
Condition 2: *edge $e_l = (v, v')$ is existent, for (1) $v \in \mathbb{F}'$, (2) $v' \notin \mathbb{F}' \cup \mathbb{F}$, (3) $t_{n',v} = k - 1$, and (4) $v \in \mathbb{T}$ (similarly, replacing v with v' and vice versa).*
Condition 3: *edge $e_l = (v, v')$ is existent, for (1) $v, v' \in \mathbb{F}$, (2) $c_{n',v} \neq c_{n',v'}$, and (3) $t_{n',v} + t_{n',v'} = k$.*

Proof: This is an immediate consequence of the definitions because all the terminals are connected. □

Lemma 4.2. *The terminals are disconnected if the attributes of the frontiers satisfy one of the following conditions:*
Condition 1: *edge $e_l = (v, v')$ is non-existent, for (1) $v \notin \mathbb{F}' \cup \mathbb{F}$, and (2) $v \in \mathbb{T}$ (similarly, for v').*
Condition 2: *edge $e_l = (v, v')$ is non-existent, for (1) $v \in \mathbb{F}'$, (2) $t_{n',v} > 0$, and (3) $d_{n',v} = 1$ (similarly, for v').*
Condition 3: *edge $e_l = (v, v')$ is existent or non-existent, for (1) $v, v' \in \mathbb{F}' \backslash \mathbb{F}$ and (2) ($t_{n',v} > 0$ or $t_{n',v'} > 0$).*

Proof: This is an immediate consequence of the definitions because the terminals are disconnected. □

Note that the state-of-the-art construction of the BDD uses only the condition 1 on Lemmas 1 and 2. As a result, the S^{2}BDD can more effectively tighten the bounds of network reliability.

We next explain the merging procedure. Since each intermediate graph on S^{2}BDD has different existent and non-existent edges, the attributes on each frontier are different (in general). The merging procedure merges the nodes that make a transition to the same sink nodes based on the following lemma:

Lemma 4.3. *Given nodes n_1 and n_2 at layer l, if we have for $\forall f \in \mathbb{F}_l$ (1) $c_{n_1,f} = c_{n_2,f}$ and (2) ($t_{n_1,f} = 0$ and $t_{n_2,f} = 0$) or ($t_{n_1,f} > 0$ and $t_{n_2,f} > 0$), then nodes derived from n_1 and n_2 with the same states of edges $e_{l+1}, \dots, e_{|\mathbb{E}|}$ make a transition to the same sink nodes.*

Proof: If n_1 and n_2 have (1) $\{c_{n_1,f}\} = \{c_{n_2,f}\}$ for all f in $\mathbb{F}_l$, the connected frontiers are the same in the intermediate graphs corresponding to n_1 and n_2. New nodes n'_1 and n'_2 derived from n_1 and n_2 are the same $\{c_{n'_1,f}\} = \{c_{n'_2,f}\}$ if they have the same states of edges $e_{l+1}, \dots, e_{|\mathbb{E}|}$. Thus, $\{c_{n_1,f}\}$ and $\{c_{n_2,f}\}$ for all f in $\mathbb{F}_l$ are the same until they make a transition to the sink nodes. Since the same $\{c_{n_1,f}\}$ and $\{c_{n_2,f}\}$ share the same connected components, each frontier has the same $\{d_{n_1,f}\}$ and $\{d_{n_2,f}\}$. In addition, frontiers f and f' must be connected if they connect to at least one terminals (i.e., $t_{n_1,f} > 0$ and $t_{n_2,f} > 0$). If (1) $\{c_{n_1,f}\} = \{c_{n_2,f}\}$ and (2) ($t_{n_1,f} = 0$ and $t_{n_2,f} = 0$) or ($t_{n_1,f} > 0$ and $t_{n_2,f} > 0$) for all f in $\mathbb{F}_l$, nodes derived from n_1 and n_2 with the same states of edges $e_{l+1}, \dots, e_{|\mathbb{E}|}$ have the same attributes on the frontiers, and thus they make a transition to the same sink nodes. □

The probabilities of the merged nodes are aggregated to one node. The probabilities p_c and p_d are consistent, regardless of whether or not the nodes are merged. These procedures do not sacrifice the exactness of the network reliability.

4.3.2 Deleting Procedure. The size of the S^{2}BDD increases exponentially as the size of the graph increases. If the size of S^{2}BDD increases, the computation cost increases to obtain the lower and upper bounds of the network reliability because it takes a large time to construct the S^{2}BDD. Hence, we control the size of S^{2}BDD by specifying the maximum size w. The deleting procedure deletes the nodes so that the size of an S^{2}BDD is not larger than w. One of major difficulties in designing this procedure pertains to which nodes should be kept in the S^{2}BDD for achieving higher efficiency and accuracy. According to Theorems 1 and 2, the number of samples effectively decreases as the probabilities p_c and p_d increase. We identify intermediate graphs in which terminals are highly likely connected or disconnected after processing a small number of edges. We make the following key observations in terms of the connectivity of terminals:

Observation 1 The terminals in the intermediate graph corresponding to node n are highly likely connected if $t_{n,f}$ is large for $\exists f \in \mathbb{F}_l$.

Observation 2 The terminals in the intermediate graph corresponding to node n are highly likely disconnected if $d_{n,f}$ is small and $t_{n,f} > 0$ for $\exists f \in \mathbb{F}_l$.

Furthermore, if the probability of node p_n is high and node n makes a transition to sink nodes, p_c and p_d increase considerably. Based on these observations, we define a heuristic function based on our observations. We compute the priorities of nodes from their attributes by the heuristic function and preferentially keep high-priority nodes. The heuristic function h to compute the priority of node n is as follows:

$$h(n) = p_n \cdot \max\nolimits_{f \in \mathbb{F}} \left(\frac{t_{n,f}}{k}, \frac{1}{d_{n,f}} \right) \text{ if } t_{n,f} > 0. \tag{10}$$

This function outputs larger value when (1) a frontier is connected to at least one terminals and (2) the frontier is connected to a large number of terminals or (3) the frontier has a small number of uncertain edges. In the former case, the terminals are likely connected, and in the latter case, the terminals are likely disconnected. Low-priority nodes (i.e., n with small $h(n)$) are then deleted from an S^{2}BDD.

4.3.3 Sampling procedure. Our approach samples possible graphs so that it avoids sampling the possible graphs that are used to compute the lower and upper bounds of network reliability, for satisfying the requirements of the stratified sampling. We sample the possible graphs from the set of possible graphs that in which terminals are not connected/disconnected yet. We denote by $\mathbb{W}_u^{\mathcal{G}}$ such set of possible graphs, and the set is obtained from intermediate graphs corresponding to the deleted nodes and nodes in the S^{2}BDD. We employ dynamic programming for efficiently sampling possible graphs from $\mathbb{W}_u^{\mathcal{G}}$. In addition, we use the idea of the *stratified random sampling* [32] for determining the number of samples for subgroups that are partial $\mathbb{W}_u^{\mathcal{G}}$.

We first divide $\mathbb{W}_u^{\mathcal{G}}$ into subgroups and then randomly sample possible graphs from each subgroup. The number of samples for each subgroup is taken in proportion to the sum of the probabilities of the intermediate graphs in the subgroup. We here explain only how to divide the deleted nodes and how to decide the number of samples for them. As for the nodes in S^{2}BDD, each subgroup is the set of possible graphs obtained from the intermediate graph corresponding to the node, and the number of samples is computed from its probabilities.

We divide the set of intermediate graphs for deleted nodes into subgroups according to original BDD layers instead of the node itself. This is because probabilities of deleted nodes are typically quite small to decide the number of samples. $\mathbb{W}_u^{\mathcal{G}_l}$ and s_l are the set of intermediate graphs corresponding to the deleted nodes at layer l and the number of samples at layer l, respectively. s_l is computed by multiplying s and the total probabilities $\hat{p_{s_l}}$ of deleted nodes at layer l. We compute $\hat{p_{s_l}}$ from the attributes maintained by the S^{2}BDD by the following equation:

$$\hat{p_{s_l}} = 1 - \textstyle\sum_{i=1}^{l-1} p_{s_i} - p_{\mathbb{N}_{next}} - p_c - p_d, \tag{11}$$

where $p_{\mathbb{N}_{next}}$ denotes the sum of probabilities of $n \in \mathbb{N}_{next}$. $\hat{p_{s_l}}$ is the expected sum of probabilities of deleted nodes. This is because $\hat{p_{s_l}}$ indicates the sum of probabilities in $\mathbb{N}_l$ when the number of nodes at layer $l+1$ reaches the maximum size. The number of samples s_l at layer l becomes $s \cdot \hat{p_{s_l}}$. The dynamic programming and stratified random sampling improve the efficiency of sampling while keeping the unbiased sampling.

4.4 Complexity

We explain the time and space complexities of our approach.

Theorem 3. *Given the uncertain graph $\mathcal{G}$, the updated number of samples s', and the maximum width of S^{2}BDD w, the time and space complexities of our approach are $O(w^2 \log w + s'(|\mathbb{V}| + |\mathbb{E}|))$ and $O(w \log w + |\mathbb{V}| + |\mathbb{E}|)$, respectively.*

Proof: The time complexity of our approach is divided into two parts; constructing S^{2}BDD and sampling. To construct S^{2}BDD, our construction method compares attributes on each node each other for generating and merging procedures. The number of attributes on each node increases in proportion to the number of frontiers. The number of frontiers is $O(\log w)$ because the number of existent/non-existent edges is at most $\log w$. Thus, the time complexity for constructing S^{2}BDD is $O(w^2 \log w)$. The time complexity of sampling is $O(s'(|\mathbb{V}| + |\mathbb{E}|))$. Therefore, the time complexity of our approach is $O(w^2 \log w + s'(|\mathbb{V}| + |\mathbb{E}|))$.

The space complexity depends on the size of S^{2}BDD and the uncertain graphs. The size of S^{2}BDD is the number of nodes multiplied by the number of attributes on each node. Therefore, the space complexity is $O(w \log w + |\mathbb{V}| + |\mathbb{E}|)$. □

5 EXTENSION

The computation cost of our approach depends on the size of the uncertain graphs as well as the number of samples. The computation cost decreases as the size of the uncertain graphs decreases. Therefore, we propose an extension technique to efficiently reduce the size of graphs while preserving the accuracy. The extension technique preprocesses the uncertain graphs before sampling possible graphs and constructing an S^{2}BDD. It not only improves the efficiency but also improves the accuracy of the approximation. The extension technique uses *2-edge-connected components* for reducting the size of uncertain graphs [7].

Definition 3 (2-edge-connected component). *Given a graph $G = (\mathbb{V}, \mathbb{E})$, an edge is called a bridge if G is disconnected after the removal of the edge from $\mathbb{E}$. Vertices that are connected by bridges are called articulation points. A subgraph $C = (\mathbb{V}_C, \mathbb{E}_C)$ of G is a 2-edge connected component if C is still connected after the removal of any edges from $\mathbb{E}_C$. We denote the sets of bridges, articulation points, and 2-edge connected components by $\mathbb{B}$, $\mathbb{A}$, and $\mathbb{C}$, respectively*

The 2-edge-connected components, bridges, and articulation points provide sets of edges (and vertices) such that the uncertain graph is disconnected or still connected when the edges (and vertices) are deleted. Because we can compute 2-edge connected components only by using the network topology of a given uncertain graph, we precompute them as an index.

The extension technique consists of three phases; (1) pruning, (2) decomposing, and (3) transforming. In the pruning phase, we first compute $\mathcal{G}'$ such that $R[\mathcal{G}] = R[\mathcal{G}']$. The number of edges in $\mathcal{G}'$ is smaller than that in $\mathcal{G}$ by pruning edges and vertices that do not affect computing the network reliability. Next, in the decomposing phase, we compute the subgraphs $\mathcal{G}_1, \ldots, \mathcal{G}_m$ where $R[\mathcal{G}'] = \Pi_{i=1}^{m} R[\mathcal{G}_i]$. Finally, in the transforming phase, we compute $\mathcal{G}_i'$ such that $R[\mathcal{G}_i] = R[\mathcal{G}_i']$ for all $1 \leq i \leq m$. Since we transform the graph into a smaller graph, the number of edges in $\mathcal{G}_i'$ is smaller than that in $\mathcal{G}_i$.

Prune: We prune vertices and edges that do not affect the network reliability. A vertex (or an edge) is unnecessary if the graph is partitioned after the removal of the vertex (or edge) from $\mathcal{G}$ and one of the partitioned graphs does not include terminals. A naive approach deletes each articulation point and bridge, and then checks whether partitioned graphs include terminals or not. This approach incurs $O((|\mathbb{B}| + |\mathbb{A}|)(|\mathbb{V}| + |\mathbb{E}|))$ time complexity. To improve the efficiency, we reconstruct the uncertain graph based on the 2-edge connected components. To do so, we first unite the set of vertices and edges included in $C \in \mathbb{C}$ to form a single vertex v_c. We then set every articulation point included in C as vertex v_a and set edges between v_a and v_c. The other vertices and edges that are not included in $\mathbb{C}$ are still in the reconstructed graphs. Therefore, the vertices of the reconstructed graph indicate $\mathbb{C}$, $\mathbb{A}$, and the vertices that are not included in $\mathbb{C}$. If any vertex in C except for articulation points is a terminal, v_c is also a terminal. The reconstructed graph is structured as a tree structure because the 2-edge connected components are connected to the other components by a single edge. To compute the necessary vertices and edges, we compute the minimum Steiner tree for terminals in the reconstructed graph. The minimum Steiner tree includes only the necessary vertices and edges to compute the network reliability because it includes only the edges and vertices that all the terminals are connected. Its computation cost is $O(|\mathbb{V}|)$, because the minimum Steiner tree in a tree structure is computed by a depth first search from a terminal.

Decompose: We decompose the graph because the time complexity for computing the network reliability on decomposed graphs becomes smaller than that on that original uncertain graph. The decomposed graph has fewer edges than the original uncertain graph. We decompose the graph according to the following lemma:

Lemma 5.1. *Given an uncertain graph and a set of bridges, we obtain $R[\mathcal{G}, \mathbb{T}] = p_b \cdot \prod_{i=1}^{m} R[\mathcal{G}_i, \mathbb{T}_i]$, where $p_b = \prod_{e_b \in \mathbb{B}} p(e_b)$ and $\mathbb{T}_i$ is the set of terminals for $\mathcal{G}_i$.*

Proof: Given intermediate graph $\mathcal{G}_{\mathbb{E}}(\mathbb{E}_\exists, \mathbb{E}_\neg)$ and edge $e \in \mathbb{E} \backslash (\mathbb{E}_\exists \cup \mathbb{E}_\neg)$, the network reliability is computed using the Factoring Theorem [10]:

$$\begin{aligned} R[\mathcal{G}_{\mathbb{E}}(\mathbb{E}_\exists, \mathbb{E}_\neg)] \quad = \quad & p(e) \cdot R[\mathcal{G}_{\mathbb{E}}(\mathbb{E}_\exists \cup e, \mathbb{E}_\neg)] \\ & + (1 - p(e)) \cdot R[\mathcal{G}_{\mathbb{E}}(\mathbb{E}_\exists, \mathbb{E}_\neg \cup e)]. \quad (12) \end{aligned}$$

If we select bridge $e_b = (v, v') \in \mathbb{B}$ as e in Equation (12), $R[\mathcal{G}_{\mathbb{E}}(\mathbb{E}_\exists, \mathbb{E}_\neg \cup e)]$ is zero because terminals in $\mathcal{G}_{\mathbb{E}}(\mathbb{E}_\exists, \mathbb{E}_\neg \cup e)$ are disconnected. Therefore, we obtain the following equation:

$$R[\mathcal{G}_{\mathbb{E}}(\mathbb{E}_\exists, \mathbb{E}_\neg)] = p(e_b) \cdot R[\mathcal{G}_{\mathbb{E}}(\mathbb{E}_\exists \cup e_b, \mathbb{E}_\neg)]. \quad (13)$$

For connecting all the terminals, e_b must be existent, and thus we can decompose the intermediate graph $\mathcal{G}_{\mathbb{E}}$ into two graphs $\mathcal{G}_{\mathbb{E}_1}$ and $\mathcal{G}_{\mathbb{E}_2}$. We also divide the terminals $\mathbb{T}$ into $\mathbb{T}_1$ and $\mathbb{T}_2$ for $\mathcal{G}_{\mathbb{E}_1}$ and $\mathcal{G}_{\mathbb{E}_2}$, respectively; $\mathbb{T}_1$ includes $\{t \in \mathbb{T}, v, v' | t, v, v' \in \mathbb{V}_1\}$ (similarly, $\mathbb{T}_2$). Thus, $R[\mathcal{G}_{\mathbb{E}}] = p(e_b) \cdot R[\mathcal{G}_{\mathbb{E}_1}] R[\mathcal{G}_{\mathbb{E}_2}]$. $\mathcal{G}_{\mathbb{E}_1}$ and $\mathcal{G}_{\mathbb{E}_2}$ are decomposed in the same manner. Then, we obtain $R[\mathcal{G}] = p_b \cdot \prod_{i=1}^{m} R[\mathcal{G}_i, \mathbb{T}_i]$. □

We decompose the uncertain graph into several subgraphs based on the above lemma. Its computation cost is $O(|\mathbb{B}||\mathbb{V}|)$ because we check whether decomposed graphs include terminals or not for each bridge.

Transform: We transform the graph to reduce its size. We delete and add the following edges and vertices without sacrificing the exactness of the network reliability:

- Sequential edges ($e = (v, v'), e' = (v, v'')$): Delete v, e and e', and add a new edge with probability $p(e) \cdot p(e')$ between v' and v'', provided that v is not a terminal and its degree is two.
- Parallel edges ($e = (v, v'), e' = (v, v')$): Delete e and e', and add a new edge with probability $(1-(1-p(e) \cdot (1-p(e'))$ between v and v'.
- Loop : Delete the loop because loops do not contribute to the network reliability. Note that transforming sequential and parallel edges can generate loops.

We iteratively repeat this process until the graph does not change. The computation cost is $O(\gamma \cdot |\mathbb{V}| \cdot {d_{avg}}^2)$ where γ and d_{avg} are the number of repetitions and the average degree of the vertices, respectively.

Consequently, the extension technique effectively reduces the computation cost for computing the network reliability with a small preprocessing time. Furthermore, it improves the accuracy of the sampling technique.

Algorithm 1: Computing the approximate network reliability

```
input  : Uncertain graph $\mathcal{G}$, terminals $\mathbb{T}$, maximum BDD size $w$, size of
         samples $s$, 2-edge connected components $\mathbb{C}$, bridges $\mathbb{B}$, articulation
         points $\mathbb{A}$
output : Approximate network reliability $\hat{R}$
1  procedure our approach
2  set $\mathbb{T}$ to $\mathcal{G}$;
3  $\hat{R}$, $\mathcal{S}_{\mathcal{G}} \leftarrow$ Preprocess($\mathcal{G}, \mathbb{T}, \mathbb{C}, \mathbb{B}, \mathbb{A}$);
4  for $\mathcal{G}_i \in \mathcal{S}_{\mathcal{G}}$ do
5      $r \leftarrow$ Construction($\mathcal{G}_i, w, s$);
6      $\hat{R} \leftarrow \hat{R} \cdot r$;
7  return $\hat{R}$;
8  end procedure
```

Algorithm 2: Constructing S^{2}BDD

```
input  : Uncertain graph $\mathcal{G}$, maximum size $w$, number of samples $s$
output : Approximate network reliability $\hat{R}$
1  procedure Construction($\mathcal{G}, w, s$)
2  Ordering($\mathbb{E}$);
3  $p_c, p_d, \hat{p_{s_l}}, c \leftarrow 0$;        /* initialize probabilities and sampling
     count */
4  $s' \leftarrow s$;
5  $\mathbb{N} \leftarrow$ CreateRoot; $\mathbb{F} \leftarrow null$;
6  for $l$ for 1, . . . , $|\mathbb{E}|$ do
7      $p_{\mathbb{N}}, p_{s_i} \leftarrow 0$;
8      $\mathbb{F}' \leftarrow \mathbb{F}$; compute $\mathbb{F}$ based on $e_l$;
9      while $\mathbb{N}$ is empty do
10         $n \leftarrow \mathbb{N}.pop$;
11         for state $\in$ { non-existent, existent} do
12             set($n, \mathbb{F}', \mathbb{F}$, state, $\mathcal{G}, e_l$);
13             if $n$ is 0-sink then $p_d \leftarrow p_d + p_n$;
14             else if $n$ is 1-sink then $p_c \leftarrow p_c + p_n$;
15             else
16                 if hashmap[$n$] is not null then
17                     $p_{hashmap[n]} \leftarrow p_{hashmap[n]} + p_n$;
18                 else
19                     if $|\mathbb{N}_{next}| \leq w$ then
20                         $h_n \leftarrow h(n)$;
21                         $\mathbb{N}_{next}$.add($n$); hashmap[$n$] $\leftarrow n$;
                           $p_{\mathbb{N}_{next}} \leftarrow p_{\mathbb{N}_{next}} + p_n$;
22                     else
23                         $p_{s_i} \leftarrow p_{s_i} + p_n$;
24                         for $i$ for
                             1, . . . , $\lfloor s' \cdot (1 - \hat{p_{s_l}} - p_{\mathbb{N}_{next}} - p_c - p_d) \rfloor$
                             do
25                             if Sampling($\mathcal{G}, n$) then $c \leftarrow c + 1$;
26         if $c + \lfloor s' \cdot p_{\mathbb{N}_{next}} \rfloor \geq s'$ then
27             for $n \in \mathbb{N}$ do
28                 for $i$ for 1, . . . , $\lfloor s' \cdot p_{\mathbb{N}_{next}} \rfloor$ do
29                     if Sampling($\mathcal{G}, n$) then $c \leftarrow c + 1$;
30             break;
31         if $\mathbb{N}_n$ is empty then
32             break;
33         $\mathbb{N} \leftarrow \mathbb{N}_{next}$;
34         sort $\mathbb{N}$ in descending order of $h(n)$;
35         $\hat{p_{s_l}} \leftarrow \hat{p_{s_l}} + p_{s_i}$; compute $s'$; clear $\mathbb{N}_{next}$; clear hashmap;
36 compute $\hat{R}$ based on the sampling;
37 return $\hat{R}$;
38 end procedure
```

Theorem 4. *Given* $\mathcal{G}_1, \ldots, \mathcal{G}_m$ *such that* $R[\mathcal{G}] = p_b \cdot \Pi_{i=1}^{m} R[\mathcal{G}_i]$*, the variance of the network reliability decreases for* $0 < \hat{R} < 1$ *and* $0 < p_b < 1$.

Proof: The network reliability is denoted by $\hat{R} = p_b \cdot \Pi_{i=1}^{m} \hat{R}[\mathcal{G}_i]$. The valiance is computed as follows:

Algorithm 3: Extension technique

```
input  : Uncertain graph $\mathcal{G}$, terminals $\mathbb{T}$, 2-edge connected components $\mathbb{C}$,
         bridges $\mathbb{B}$, articulation points $\mathbb{A}$
output : Probability $p_b$, the set of decomposed graphs $\mathcal{S}_{\mathcal{G}}$
procedure Preprocess($\mathcal{G}, \mathbb{C}, \mathbb{B}, \mathbb{A}$)
/* Prune                                                          */
$\mathcal{G}_r \leftarrow$ Reconstruct($\mathcal{G}$);
Compute the minimum Steiner tree $\mathcal{T}$ for $\mathcal{G}_r$ and terminals;
Delete edges and vertices of $\mathcal{G}$ not included in $\mathcal{T}$;
/* Decompose                                                      */
$p_b \leftarrow \prod_{e_b \in \mathbb{B}} p(e_b)$;
Delete the set of bridges in $\mathcal{G}$;
$\mathcal{S}_{\mathcal{G}} \leftarrow$ the set of disconnected graphs;
/* Transform                                                      */
for $\mathcal{G}' \in \mathcal{S}_{\mathcal{G}}$ do
    while 1 do
        for $v \in \mathbb{V}$ of $\mathcal{G}'$ do
            if $v$ connects to edge $e = (v, v)$ then
                delete $e = (v, v)$;
            if $v \notin \mathbb{T}$ and $v$ connects to just two edges $e = (v, v')$ and
              $e' = (v, v'')$ then
                delete $e$ and $e'$ from $\mathcal{G}'$;
                add a new edge $(v', v'')$ with probability $p(e) \cdot p(e')$;
        for $v \in \mathbb{V}$ of $\mathcal{G}'$ do
            for $\forall$ pair of $u$ and $u' \in$ the set of neighbor vertices of $v$ do
                if $u = u'$ then
                    delete edge $e = (v, u)$ and $e' = (v, u')$;
                    add a new edge $(v, u)$ with probability
                      $(1 - (1 - p(e) \cdot (1 - p(e'))$;
        if The number of edges does not change then
            break;
return $p_b$, $\mathcal{S}_{\mathcal{G}}$;
end procedure
```

$$
\begin{aligned}
Var[\hat{R}] &= Var[p_b \cdot \Pi_{i=1}^{m} \hat{R}[\mathcal{G}_i]] \\
&= (Var[p_b] + {p_b}^2)(Var[\hat{R}[\mathcal{G}_1]] + \hat{R}[\mathcal{G}_1]^2) \cdots \\
&\quad (Var[\hat{R}[\mathcal{G}_m]] + \hat{R}[\mathcal{G}_m]^2) - {p_b}^2 \cdot \Pi_{i=1}^{m} \hat{R}[\mathcal{G}_i]^2 \\
&= {p_b}^2 \Pi_{i=1}^{m} (Var[\hat{R}[\mathcal{G}_i]] + \hat{R}[\mathcal{G}_i]^2) - {p_b}^2 \Pi_{i=1}^{m} \hat{R}[\mathcal{G}_i]^2 \\
&= {p_b}^2 \Pi_{i=1}^{m} \left(\frac{\hat{R}[\mathcal{G}_i](1-\hat{R}[\mathcal{G}_i])}{s} + \hat{R}[\mathcal{G}_i]^2 \right) - {p_b}^2 \Pi_{i=1}^{m} \hat{R}[\mathcal{G}_i]^2 \\
&= {p_b}^2 \Pi_{i=1}^{m} \hat{R}[\mathcal{G}_i] \left(\frac{(1+(s-1)\hat{R}[\mathcal{G}_i])}{s} \right) - {p_b}^2 \Pi_{i=1}^{m} \hat{R}[\mathcal{G}_i]^2 \\
&< \frac{{p_b}^2 \Pi_{i=1}^{m} \hat{R}[\mathcal{G}_i]}{s} - \frac{{p_b}^2 \Pi_{i=1}^{m} \hat{R}[\mathcal{G}_i]^2}{s} \\
&= p_b \frac{\hat{R}(1-\hat{R})}{s} < \frac{\hat{R}(1-\hat{R})}{s}
\end{aligned}
\qquad (14)
$$

Note that $Var[p_b] = 0$. $Var[\hat{R}]$ is smaller than the variance of the network reliability of the original graph. □

6 ALGORITHM OF OUR APPROACH

In this section, we explain the entire algorithm of our approach. Algorithm 1 shows the pseudo-codes. Our approach first preprocesses uncertain graphs and obtains decomposed uncertain graphs (line 3). For each decomposed graph, it then constructs an S^{2}BDD to compute the approximate network reliability of the decomposed graphs (lines 4–5). The product of the network reliability of each decomposed graph is the original network reliability (line 6).

Algorithm 2 shows the pseudo-codes for the construction of an S^{2}BDD. We process edges in a predefined order, and compute the set of frontiers (lines 6–8). For each node at layer l, we compute the nodes at layer $l + 1$ according to the states of the edges

Table 2: Dataset

Name	Abbr.	Type	#vertices	#edges	Avg. Deg	Avg. Prob
Zachary-karate-club	Karate	Social	34	78	4.59	0.527
American-Revolution	Am-Rv	Affiliation	141	160	2.27	0.528
DBLP before 2000	DBLP1	Coauthorship	25,871	108,459	8.38	0.222
DBLP after 2000	DBLP2	Coauthorship	48,938	136,034	5.56	0.203
Tokyo	Tokyo	Road network	26,370	32,298	2.45	0.391
New York City	NYC	Road network	180,188	208,441	2.31	0.294
Hit-direct	Hit-d	Protein	18,256	248,770	27.25	0.470

(lines 11–12). The set function (line 12) sets attributes on the new node to n and checks whether the terminals are connected or disconnected based on Lemmas 1 and 2. If the new node are 0-sink and 1-sink, we add p_n to p_d and p_c, respectively (line 13–14). Otherwise, we compute hash values for n, and if the hash of n is not null, we add the probability p_n to the node in the hash (lines 16–17). If the hash is null with respect to n, it inserts n into the set $\mathbb{N}_{next}$ of nodes at layer $l + 1$ and into the hash after computing their priorities (lines 19–21). If the number of nodes in $\mathbb{N}_n$ exceeds the maximum size w, we delete n and pick possible graphs as samples from n (lines 22–25). After sampling an enough number of possible graphs, we sample form the nodes in the S^{2}BDD (lines 26–29).

Algorithm 3 shows the pseudo-codes for the extension technique. The extension technique first reconstructs the uncertain graph (line 2). Then, it computes the minimum Steiner tree for the reconstructed graph and prunes the edges and vertices that are not included in the Steiner tree from the original uncertain graph (lines 3–4). To decompose the graph, we compute the product of the probabilities of bridges p_b (line 5). Then, we delete bridges from the uncertain graph, and the disconnected subgraphs are inserted into the set of decomposed uncertain graphs (lines 6–7). For each decomposed graph, it transforms vertices and edges that satisfy the transformation rules (lines 8–20).

7 EXPERIMENT

We evaluate our approach in terms of efficiency, accuracy, and memory usage.

7.1 Dataset

We summarize the datasets in Table 2. The first two datasets; Zachary-karate-club and American-revolution are small datasets for evaluating accuracy, which are extracted from KONECT[3]. We randomly assign probabilities based on the uniform distribution [9]. The other five datasets; DBLP before 2000, DBLP after 2000, Tokyo, New York City, and Hit-direct, are large datasets. Edge existence probabilities for each large dataset are assigned based on the attributes of the edges in each dataset. DBLP before 2000 and DBLP after 2000 are graphs extracted from DBLP[4], where vertices and edges are authors and co-author, respectively. We compute the edge existence probabilities by $\frac{\log(\alpha+1)}{\log(\alpha_M+2)}$, where α and α_M denote the number of co-authors and the maximum in each dataset, respectively [6]. The Tokyo and New York City datasets are road networks extracted from OpenStreetMap[5]. We compute the edge existence probabilities in the same manner as with the DBLP datasets, although we use road lengths instead of the number of co-authors. Note that both the Tokyo and New York City datasets are not planar graphs. Hit-direct is a protein-protein interaction network extracted from the Human Genome Center[6]. We use the interaction scores $\in (0, 1]$ of interactions as the edge existence probabilities.

7.2 Setting and Implementation

For each dataset, we generate 20 searches (except when we evaluate the accuracy, for which see Section 7.6). The terminals are selected randomly from vertices. We vary the number of terminals k, the number of samples s, and the maximum size of the S^{2}BDD w.

Because the existence probabilities of possible graphs can be very small, we use the Boost.Multiprecision library, with precision of 10,000 decimal points, for the large datasets. We compute the 2-edge-connected components using code provided by the authors [7]. We compare our approach with two existing approaches; the sampling-based and BDD-based approaches. The BDD-based approach uses the state-of-the art library, TdZDD. All algorithms are implemented in C++, and run on a server with an Intel Xenon E7-8860v4 at 2.20GHz with 256GB RAM.

7.3 Efficiency

We compare the efficiency of our approach with that of sampling-based and BDD-based approaches. Figure 3 shows the response time for each large dataset when the numbers of terminals k is set to 5, 10, and 20. DNF indicates that we cannot compute the network reliability due to the lack of memory space. We use Monte Carlo estimator for our approach and the sampling-based approach (denoted by Pro(MC) and Sampling(MC), respectively) and set s to 10,000. For our approach, we set w to 10,000. We also evaluate our approach without the extension technique denoted by Pro(MC)w/o ext. We here omit the results of Horvitz-Thompson estimator because they are almost equivalent to those of Monte Carlo estimator.

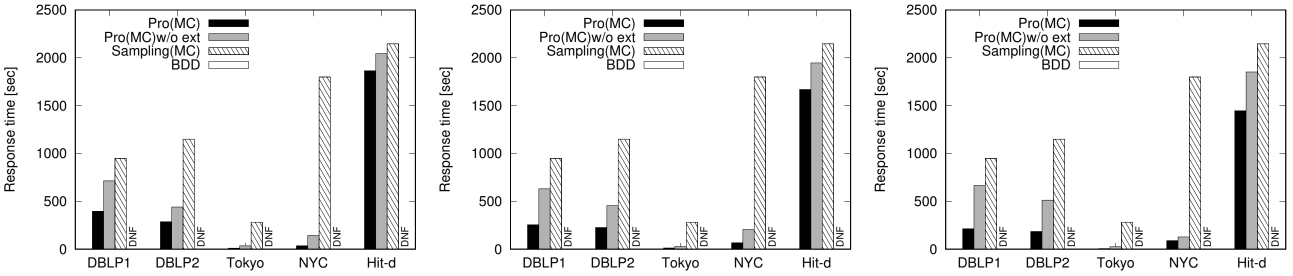
The results show that our approach is more efficient than both of the sampling-based and the BDD-based approaches for all k. The BDD-based approach cannot compute the network reliability because it runs out of memory. Our approach achieves higher efficiency than the sampling-based approach because it reduces the number of samples. Furthermore, we can see that the extension technique improves the efficiency. In particular, our approach works well on the Tokyo and NYC datasets. This is because the S^{2}BDD works well for planar-like graphs (even when they are not strictly planar graphs). In the Hit-direct dataset, the lower and upper bounds do not effectively become tight because the number of degrees is large. Nevertheless, our approach is more efficient than the sampling-based approach.

[3] http://konect.uni-koblenz.de/

[4] http://dblp.uni-trier.de/

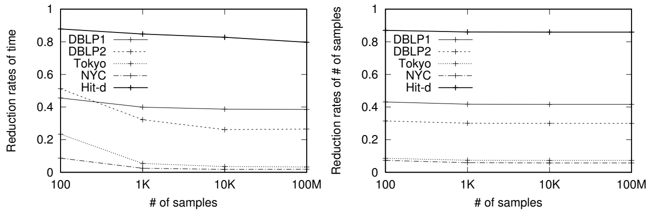
[5] https://www.openstreetmap.org

[6] http://hintdb.hgc.jp/htp/download.html.



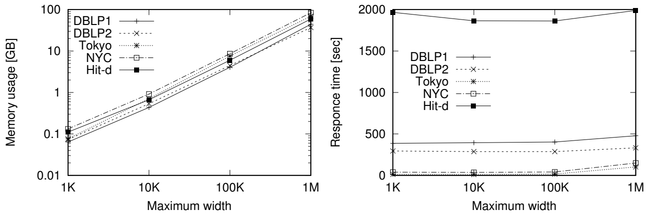
(a) $k = 5$ **(b)** $k = 10$ **(c)** $k = 20$

Figure 3: Overview of efficiency



(a) Response time **(b) # of samples**

Figure 4: Efficiency with varying the number of samples



(a) Memory usage **(b) Response time**

Figure 5: Efficiency with varying the maximum width

7.4 Effect of Number of Samples

We evaluate the effect of the given number of samples. Figure 4 shows (a) the rate of response time of our approach over that of the sampling-based approach and (b) the rate of updated samples s' over s, varying the number of samples. This figure shows that our approach becomes more efficient as the given number of samples increases. This is because the reduction of the number of samples is more effective when the given number of samples is large. Therefore, our approach more effectively works when we need a high accurate network reliability.

7.5 Effect of Maximum Width

We evaluate the effect of the given maximum width of S^{2}BDD. The maximum width w affects the memory usage and efficiency. Figure 5 shows (a) the memory usage and (b) the response time. From Figure 5(a), we can see that the memory usage increases as the maximum width increases. The memory usage depends on the maximum width but not depends on the size of graphs. Our approach can be used for large-scale graphs in terms of memory usage. From Figure 5(b), we can see that the response time does not largely depend on the maximum width. When the maximum width is large, our approach can reduce the number of samples but takes a large computation cost for constructing S^{2}BDD. Our approach is robust enough to the maximum width in terms of efficiency. Consequently, our approach effectively decreases the response time even for large-scale graphs.

7.6 Accuracy

We evaluate the accuracy of our approach compared with the sampling-based approaches. For both approaches, we use Horvits-Thompson estimator (denoted by Pro(HT) and Sampling(HT)) as well as Monte Carlo estimator. Since the network reliability problem is #P-complete, we cannot compute the exact answer for large datasets in terms of both response time and memory usage. We use the Karate and Am-Rv datasets which can be computed the exact network reliability. We evaluate the variance and the error rate to determine the accuracy of the approximation as follows: $variance = \frac{\Sigma_{i=1}^{q_1}\Sigma_{j=1}^{q_2}(R_i - \hat{R}_{i,j})^2}{q_1 \cdot q_2}$ and $error\ rate = \frac{\Sigma_{i=1}^{q_1}\Sigma_{j=1}^{q_2}|R_i - \hat{R}_{i,j}|}{q_1 \cdot q_2 \cdot R_i}$, where R_i and $\hat{R}_{i,j}$ denote the i-th exact network reliability and the j-th approximate network reliability for the i-th search, respectively. We generate 100 searches and compute the network reliability 100 times for each search (i.e., both q_1 and q_2 are 100).

Tables 3 and 4 show the accuracy on the Karate and Am-Rv datasets, respectively. Table 3 shows that our approach outperforms the sampling-based approaches in terms of both of the variance and error rate. Comparing the variance between the estimators, the Monte Carlo estimator is slightly better than the Horvits-Thompson sampling. This is because we sample possible graphs with replacement, and thus the Horvits-Thompson estimator is less effective. Table 4 shows that our approach always computes the exact network reliability on the Am-Rv dataset— its error rate is zero. Both of the existing sampling-based approaches have high error rates when $k = 20$ although their variances are small. Because the network reliability is very small, the sampling-based approaches rarely sample the possible graphs in which terminals are connected. Thus, the approximate network reliability is often zero, and the error rates are close to one. From these results, we conclude that our approach can achieve less variance and error rate with fewer samples than the other approaches and compute the exact answer for small-scale graphs.

7.7 Effect of Extension Technique

Finally, we evaluate the performance of the extension technique. The effect of the extension technique is detailed in Table 5 which shows the process time and the ratio of the maximum number of edges in decomposed graphs over the number of edges in the original uncertain graph. The results show that the extension technique requires a very small time compared with computing the network reliability. Thus, it effectively reduces the total response time. Since it reduces the size of uncertain graphs, it mitigates the computation cost for the S^{2}BDD. The extension technique is effective for improving the efficiency of our approach.

Table 3: Accuracy on Karate dataset

k	Method	Variance	Error rate
5	Pro(MC)	0.025	0.036
	Pro(HT)	0.025	0.036
	Sampling(MC)	0.025	0.037
	Sampling(HT)	0.029	0.042
10	Pro(MC)	0.013	0.058
	Pro(HT)	0.014	0.059
	Sampling(MC)	0.013	0.058
	Sampling(HT)	0.015	0.062
20	Pro(MC)	$0.76 \cdot 10^{-3}$	0.054
	Pro(HT)	$0.85 \cdot 10^{-3}$	0.057
	Sampling(MC)	$0.78 \cdot 10^{-3}$	0.056
	Sampling(HT)	$0.86 \cdot 10^{-3}$	0.057

Table 4: Accuracy on Am-Rv dataset

k	Method	Variance	Error rate
5	Pro(MC)	0	0
	Pro(HT)	0	0
	Sampling(MC)	$0.43 \cdot 10^{-4}$	0.061
	Sampling(HT)	$0.31 \cdot 10^{-4}$	0.059
10	Pro(MC)	0	0
	Pro(HT)	0	0
	Sampling(MC)	$0.099 \cdot 10^{-5}$	0.38
	Sampling(HT)	$0.12 \cdot 10^{-5}$	0.37
20	Pro(MC)	0	0
	Pro(HT)	0	0
	Sampling(MC)	$0.10 \cdot 10^{-3}$	1.00
	Sampling(HT)	$0.10 \cdot 10^{-3}$	1.00

Table 5: Effect of extension technique

Dataset	Process time [sec]	Reduced graph size
Karate	$0.0277 \cdot 10^{-3}$	0.757
Am-Rv	$0.310 \cdot 10^{-3}$	0.120
DBLP1	0.060	0.946
DBLP2	1.61	0.797
Tokyo	0.015	0.425
NYC	0.370	0.279
Hit-d	0.184	0.982

8 CONCLUSION

In this paper, we proposed an efficient sampling-based approach for computing the approximate network reliability. Our approach reduces the number of samples by using lower and upper bounds of the network reliability based on the stratified sampling. We developed scalable and sampling BDD, called S^{2}BDD, which efficiently computes the bounds. The S^{2}BDD preferentially searches for the possible graphs that highly improve the bounds. We further developed the extension technique of our approach to reduce the size of graphs. Experiments demonstrated that our approach is up to 51.2 times faster than the sampling-based approach with a higher accuracy.

ACKNOWLEDGEMENT

This research is partially supported by JST ACT-I Grant Number JPMJPR18UD and by JSPS KAKENHI Grant-in-Aid for Young Scientists (B) (JP15K21069), Japan.

REFERENCES

[1] Charu C. Aggarwal. 2009. *Managing and Mining Uncertain Data*. Vol. 35. Kluwer.

[2] Avinash Agrawal and A Satyanarayana. 1984. An O (|E|) time algorithm for computing the reliability of a class of directed networks. *Operations research* 32, 3 (1984), 493–515.

[3] S Hasanuddin Ahmad. 1988. Simple enumeration of minimal cutsets of acyclic directed graph. *IEEE transactions on reliability* 37, 5 (1988), 484–487.

[4] Saurabh Asthana, Oliver D King, Francis D Gibbons, and Frederick P Roth. 2004. Predicting protein complex membership using probabilistic network reliability. *Genome research* 14, 6 (2004), 1170–1175.

[5] Michael O Ball, Charles J Colbourn, and J Scott Provan. 1995. Network reliability. *Handbooks in operations research and management science* 7 (1995), 673–762.

[6] Matteo Ceccarello, Carlo Fantozzi, Andrea Pietracaprina, Geppino Pucci, and Fabio Vandin. 2017. Clustering uncertain graphs. *PVLDB* 11, 4 (2017), 472–484.

[7] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. 2013. Efficiently computing k-edge connected components via graph decomposition. In *SIGMOD*. 205–216.

[8] James Cheng, Zechao Shang, Hong Cheng, Haixun Wang, and Jeffrey Xu Yu. 2014. Efficient processing of k-hop reachability queries. *The VLDB Journal* 23, 2 (2014), 227–252.

[9] Yurong Cheng, Ye Yuan, Lei Chen, Guoren Wang, Christophe Giraud-Carrier, and Yongjiao Sun. 2016. DISTR: a distributed method for the reachability query over large uncertain graphs. *IEEE Transactions on Parallel and Distributed Systems* 27, 11 (2016), 3172–3185.

[10] Charles J Colbourn. 1987. *The combinatorics of network reliability*. Oxford University Press New York.

[11] George S Fishman. 1986. A comparison of four Monte Carlo methods for estimating the probability of st connectedness. *IEEE Transactions on reliability* 35, 2 (1986), 145–155.

[12] Christian Frey, Andreas Züfle, Tobias Emrich, and Matthias Renz. 2018. Efficient information flow maximization in probabilistic graphs. *TKDE* 30, 5 (2018), 880–894.

[13] R Hamer, G De Jong, E Kroes, and P Warffemius. 2005. The value of reliability in Transport–Provisional values for the Netherlands based on expert opinion. *Transport Research Centre of the Dutch Ministry of Transport* (2005).

[14] Gary Hardy, Corinne Lucet, and Nikolaos Limnios. 2007. K-terminal network reliability measures with binary decision diagrams. *IEEE Transactions on Reliability* 56, 3 (2007), 506–515.

[15] David G Harris and Aravind Srinivasan. 2018. Improved bounds and algorithms for graph cuts and network reliability. *Random Structures & Algorithms* 52, 1 (2018), 74–135.

[16] Johannes U Herrmann and Sieteng Soh. 2009. A memory efficient algorithm for network reliability. In *Asia-Pacific Conference*. 703–707.

[17] Ronald Jansen, Haiyuan Yu, Dov Greenbaum, Yuval Kluger, Nevan J Krogan, Sambath Chung, Andrew Emili, Michael Snyder, Jack F Greenblatt, and Mark Gerstein. 2003. A Bayesian networks approach for predicting protein-protein interactions from genomic data. *science* 302, 5644 (2003), 449–453.

[18] Ruoming Jin, Lin Liu, and Charu C Aggarwal. 2011. Discovering highly reliable subgraphs in uncertain graphs. In *SIGKDD*. 992–1000.

[19] Ruoming Jin, Lin Liu, Bolin Ding, and Haixun Wang. 2011. Distance-constraint reachability computation in uncertain graphs. *PVLDB* 4, 9 (2011), 551–562.

[20] Charles R Kalmanek and Y Richard Yang. 2010. The challenges of building reliable networks and networked application services. In *Guide to Reliable Internet Services and Applications*. 3–17.

[21] Jun Kawahara, Takeru Inoue, Hiroaki Iwashita, and Shinichi Minato. 2017. Frontier-based search for enumerating all constrained subgraphs with compressed representation. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 100, 9 (2017), 1773–1784.

[22] Arijit Khan, Francesco Bonchi, Aristides Gionis, and Francesco Gullo. 2014. Fast Reliability Search in Uncertain Graphs. In *International Conference on Extending Database Technology*. 535–546.

[23] Arijit Khan and Lei Chen. 2015. On uncertain graphs modeling and queries. *PVLDB* 8, 12 (2015), 2042–2043.

[24] Minh Lê, Max Walter, and Josef Weidendorfer. 2014. Improving the kuo-luyeh algorithm for assessing two-terminal reliability. In *European Dependable Computing Conference*. 13–22.

[25] Mitchell O Locks. 1987. A minimizing algorithm for sum of disjoint products. *IEEE Transactions on Reliability* 36, 4 (1987), 445–453.

[26] Takanori Maehara, Hirofumi Suzuki, and Masakazu Ishihata. 2017. Exact Computation of Influence Spread by Binary Decision Diagrams. In *International Conference on World Wide Web*. 947–956.

[27] Eugène Manzi, Martine Labbé, Guy Latouche, and Francesco Maffioli. 2001. Fishman's sampling plan for computing network reliability. *IEEE Transactions on Reliability* 50, 1 (2001), 41–46.

[28] Rajeev Motwani and Prabhakar Raghavan. 2010. *Randomized algorithms*. Chapman & Hall/CRC.

[29] William G Ortel. 1999. Broad band optical fiber telecommunications network. (Jan. 19 1999). US Patent 5,861,966.

[30] J Scott Provan. 1986. The complexity of reliability computations in planar and acyclic graphs. *SIAM J. Comput.* 15, 3 (1986), 694–702.

[31] J NoK Rao, HO Hartley, and WG Cochran. 1962. On a simple procedure of unequal probability sampling without replacement. *Journal of the Royal Statistical Society. Series B (Methodological)* (1962), 482–491.

[32] S Thompson. 2002. *Sampling*. Wiley.

[33] Leslie G Valiant. 1979. The complexity of enumeration and reliability problems. *SIAM J. Comput.* 8, 3 (1979), 410–421.

[34] Lucien DJ Valstar, George HL Fletcher, and Yuichi Yoshida. 2017. Landmark Indexing for Evaluation of Label-Constrained Reachability Queries. In *SIGMOD*. 345–358.

[35] Fu-Min Yeh, Shyue-Kung Lu, and Sy-Yen Kuo. 2002. OBDD-based evaluation of k-terminal network reliability. *IEEE Transactions on Reliability* 51, 4 (2002), 443–451.

[36] Bihai Zhao, Jianxin Wang, Min Li, Fang-Xiang Wu, and Yi Pan. 2014. Detecting protein complexes based on uncertain graph model. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 11, 3 (2014), 486–497.

[37] Junfeng Zhou, Shijie Zhou, Jeffrey Xu Yu, Hao Wei, Ziyang Chen, and Xian Tang. 2017. DAG reduction: Fast answering reachability queries. In *SIGMOD*. 375–390.

Scalable Parallelization of RDF Joins on Multicore Architectures

Dimitris Bilidas
National and Kapodistrian University of Athens
Greece
d.bilidas@di.uoa.gr

Manolis Koubarakis
National and Kapodistrian University of Athens
Greece
koubarak@di.uoa.gr

ABSTRACT

The RDF data model has emerged as the most prominent way to interlink and exchange data on the Web due to its simplicity in the form of subject predicate object statements, but this simplicity comes with the cost of having to execute a large number of joins in order to get the desirable query results. Numerous approaches exist that aim to treat this problem, mainly focusing on disk based storage. In this work we consider a main memory setting and present a physical design and query method aiming to exploit spatial locality for efficient in-memory processing. Our design is also amenable to straightforward parallelization, something crucial for main memory database systems. Specifically, we present a join implementation that allows to achieve any desired degree of parallelism on arbitrary join queries and RDF graphs stored in memory using compact vertical partitioning. We use an adaptive join processing approach, such that we take advantage of complete or even partial ordering of RDF data, which is compactly stored in order to increase spatial locality and keep memory consumption low, coupled with an ID-to-Position vector index used when ordering does not allow for efficient scanning of the input relation. We have implemented an in-memory prototype that experimentally shows the efficiency and scalability of our proposal, taking advantage of continuously growing sizes of main memory and multi-core environments of modern hardware. Specifically, we show that for a machine with 128 GB of main memory and 16 cores, which is a reasonable amount for an average modern server, our prototype can store and query RDF graphs with up to two billion triples, and it outperforms centralized and distributed state of the art approaches.

1 INTRODUCTION

The *Resource Description Framework* (RDF) is a data model recommended by the W3C for semantic data integration, sharing and linking across different organizations and applications on the Web. RDF provides flexible modeling of data coming from heterogeneous domains in the form of triples forming subject-predicate-object statements, facilitating the construction of Knowledge Graphs. Every component of such a triple is a resource uniquely identified by an IRI or a data value in the form of a literal. The latter can only be present in the object position. A set of such statements can be considered an RDF graph, where subjects and objects are nodes and there exists an arc labeled with the property name, connecting corresponding subject and object for each statement. Several organizations publish data in the RDF model, leading to interlinking information from different sources and automatic processing using software agents. As a result, as of 2018 the *Linked Open Data* (LOD) cloud [34] contains more than 1000 datasets and 60 billion triple statements, with DBpedia [8], a dataset that contains semantic information extracted from Wikipedia, taking up a central position with 3 billion triples and around 50 million links to other datasets.

The SPARQL query language is the W3C recommendation for querying RDF graphs. The basic building block of SPARQL queries are triple patterns. A triple pattern is similar to an RDF statement, with the exception that each component (subject, predicate or object) can be either a resource or a variable. The evaluation of a single triple pattern over an RDF graph consists of finding matches of the pattern on the graph such that variables are substituted by RDF resources. A *Basic Graph Pattern* (BGP) is a set of triple patterns. During evaluation of a BGP all triple patterns are matched to an RDF statement and common variables between triple patterns are substituted by the same resource. If we consider RDF storage on a single relational triples table, a BGP with n triple patterns corresponds to $n - 1$ self joins of the triples table.

Since the adoption of the RDF data model numerous systems and research prototypes have been developed aiming at efficient SPARQL query evaluation, focusing mainly on the evaluation of BGPs which proved to be extremely demanding. Centralized systems explored different physical storage options and query execution techniques. Main storage schemas include a single triples table, denormalized property tables, vertical partitioning, graph-based storage and storage based on bit arrays. Details and references to such systems are presented in the next section. As scalability became an issue with the continuously increasing size of several datasets, distributed approaches came into play, assisted by cloud technologies such as the MapReduce framework, its implementation Apache Hadoop and several Big Data processing systems built on top of it. Most of these systems use optimizations in order to minimize the execution cycles, which correspond to Hadoop jobs and involve data transfer between the workers. This is due to the synchronous nature of the MapReduce paradigm. As a result, depending on data partitioning and replication one can achieve evaluation completely in parallel for some queries, but for queries that require communication the overhead is important due to the synchronization step.

A number of in-memory distributed systems were later proposed such that their communication is based on custom asynchronous methods, mostly on the Message Passing Interface (MPI) standard. Trinity.RDF [42] is based on graph exploration and it was the first system to follow this design. TriAD [14] and the extension of the centralized main memory RDF store RDFox with a dynamic data exchange operator [26] also use an asynchronous execution model (In what follows we will refer to the system described in [26] as the dynamic exchange operator approach), but unlike Trinity.RDF they use relational-style joins, increasing the level of parallelism for large intermediate results over the graph-based approach. In order to do so, both of these systems use expensive graph partitioning before data loading. AdPart [3] tries to overcome this problem by using simple subject-based hash partitioning and then adaptively, based on the query load, replicates specific data fragments to the workers. As a result of the initial subject-based partitioning, expensive broadcast of intermediate result occurs in case of joins on objects.

Our query processing approach is inspired by the asynchronous execution model of main-memory distributed RDF stores, mainly of TriAD and the dynamic exchange operator approach. Both these approaches use expensive preprocessing in the form of graph partitioning in order to minimize communication between servers during query execution. Also, extra effort is needed in order to track the server that contains each resource. Most importantly, even in a centralized parallel environment these systems would require some form of inter-process or inter-thread communication and as a result some form of synchronization. For example, in case of rehashing, each worker of TriAD has to wait in order to receive and rehash all intermediate results from all other workers. Same kind of overheads occur in the dynamic exchange operator where each worker must hold a queue for each query atom, where incoming messages are put. This may lead to blocking execution until some other worker process results for a subsequent query atom. Also, in the dynamic exchange operator approach detecting termination is not trivial and requires a round of message exchanging. Our method ensures parallel execution without any form of communication or synchronization between the workers (in our case threads) while at the same time avoiding expensive preprocessing like graph partitioning. Furthermore, we adaptively decide to scan the corresponding partitions when it is preferable, instead of always using index-based nested loops as done by the dynamic exchange operator approach.

Regarding the physical data storage, our approach is inspired by *column-store* systems such as MonetDB [16] and C-Store [37], as we first use vertical partitioning [1] to create a separate table for each property, and then keep subjects and objects for each property in separate arrays so that each tuple can be reconstructed by relating entities at the same positions in these arrays, reminiscent of the virtual IDs of column stores. This way we achieve increased spatial locality during processing. Also, we allocate a single array position for each distinct subject or object as a simple form of column specific compression (reminiscent of the POS and PSO indexes used by Hexastore [39]) and we keep two replicas of each two-column table in different sort orders. The main contributions presented in this work are:

- A join processing approach with low memory consumption, able to efficiently parallelize evaluation of arbitrary multi-join BGPs without any communication.
- Physical design method which compactly stores RDF data in memory, in order to increase spatial locality during join processing. For example, for scale 10240 of the LUBM dataset with about 1.4 billion triples, excluding dictionary, the storage requirements are only 22 GB (50GB if we include the dictionary).
- A cache-friendly method which adaptively, during execution, decides to switch from binary search to scan in order to take advantage of existing (even partial) sorting of RDF triples, that further improves our join implementation. An auxiliary bit vector index can be used to avoid binary search and improve efficiency.
- An implementation and experimental evaluation for datasets up to 2 billion triples which shows that our proposal outperforms existing state of the art centralized systems. Also, based on published results for other systems, it is shown that for tested datasets, like LUBM 10240, our implementation running on a single 16-core server outperforms (mostly for complex queries) or performs close to the fastest state of the art asynchronous distributed in-memory systems deployed on a cluster of machines.

We proceed by first presenting related work. Then in Section 3 we present details of the physical data storage and in Section 4 we present details of the adaptive join method that allows for incorporating parallelism into processing. We present implementation details and experimental evaluation in Section 5 and we finally conclude and discuss future work.

2 RELATED WORK

RDF storage using relational technology has been a subject of research since the proposal of the RDF data model. BGP evaluation using a single triples table that contains the whole RDF graph involves expensive self joins over this large table. As a solution, some systems like Jena [40] proposed the usage of "flattened" property tables, which contain a larger number of columns, in an effort to simulate a relational schema and avoid joins as much as possible. Nevertheless, this design has some drawbacks, like for example a lot of NULL values for wide tables, the need for UNION during a single BGP processing and difficulty to handle multi-values attributes. [5] aims at efficient evaluation using an object-relational DBMS including a two-column representation for properties. Vertical partitioning[1] uses this representation in order to treat the drawbacks of the property tables. In this approach a separate two-column table is created for every property of the RDF graph. In this case, the number of joins is not reduced in comparison to the single triples table, but each join is between smaller tables and also tables not relevant to the query do not need to be accessed at all. Column stores are ideal candidates for RDF processing using vertical partitioning, as they provide compact storage and compression over each column.

Hexastore enhances the vertical partitioning by replicating the data through six different indexes, corresponding to all possible permutations of subject, predicate and object [39]. RDF-3X [23] also uses extensive indexing such that an index is created not only for all possible permutations but also for aggregated values, resulting in 15 indexes stored as clustered B+ trees. This schema along with several optimizations, such as skipping large parts of irrelevant data during merge joins using a form of *sideways information passing*, made RDF-3X one of the most efficient disk-based RDF stores, despite conceptually using the single triples-table approach. Our design follows the vertical partitioning approach, but as in Hexastore, we keep two different replicas for each property in different sort orders (corresponding to POS and PSO indexes) and we also compactly store only distinct subjects and objects. Also, our adaptive join optimization (Section 4.1) can be considered a way of skipping irrelevant data as in RDF-3X.

Regarding SPARQL query processing using cloud technologies, an initial approach using the MapReduce framework is presented in [30, 31]. In this work, the authors describe the query evaluation of Basic Graph Patterns of SPARQL using an iterative algorithm, such that every join in the query requires a separate MapReduce job. The RDF data is stored in plain files in the distributed file system. A similar iterative approach is also used in [20], but here the authors note that more than one triple patterns that share a variable can be joined together in the same MapReduce job. They use a greedy selection algorithm that chooses in every step the variable that appears in more triple patterns and they employ reduce-side joins to get the results. In [11] predicate-based hash partitioning is employed. The query is decomposed to subqueries using the same partitioning and in every node a local Sesame RDF store is used to evaluate each subquery. Instead of hash partitioning, in [15] the authors use a graph partitioning algorithm to assign triples to nodes and also they employ data replication for triples that are on the boundaries of each partition, in order to maximize the number of subqueries that can be executed without communication between the nodes. They stress the usefulness of a heuristic that finds the minimal number of subqueries because this corresponds to a minimal number of Hadoop jobs, and they split each query to a number of such subqueries using a brute-force method, which is suitable only for queries with few triple patterns.

A number of approaches store the RDF data into an existing system that has its own declarative language and then they transform the SPARQL queries into that language. For example, [32] uses Pig Latin[24] and performs some well known optimizations to the SPARQL query, like the early execution of filters and some selectivity estimations based on variable counting. During the translation to Pig Latin, [32] just uses multi joins when consecutive joins on the same variable are found, as this is an option that Pig Latin offers. RAPID+, a system which is also based on Pig Latin, is presented in [29]. Here the authors propose an intermediate algebra which is called Nested TripleGroup Algebra, in order to facilitate the grouping of join operators during the translation of the query to the execution plan in Pig Latin. The result is that each star join involving two or more triple patterns can be executed in one Map-Reduce job, using vertical partitioning.

H_2RDF+ [25] uses HBase[1] to store the RDF data. It takes advantage of the HBase key ordering for each table and it uses six tables, each one corresponding to an RDF triple permutation. In this way there is replication of the data, so that the system can perform fast merge joins when all triples are part of the initial RDF data. When some data is result of an intermediate step, the system first performs a sorting on this intermediate data. Another key feature of the system is that during the query planning it examines the option that the query will be executed in a centralized system. The rationale behind this is that if the query is simple, its evaluation in a centralized system can be preferable, because one can avoid the overhead of the MapReduce jobs and network communication. The system uses a greedy planner to decide about the order of the joins, based on a cost model and some index statistics that it has. In a similar manner, the system named Rya[27] uses Accumulo[2], to store indexes for permutations of subject, predicate and object in the row ID field of each corresponding table, but it only uses three indexes instead of all the possible ones. Rya supports range queries and regular expressions, multi-threaded join execution and also provides some limited inferencing capabilities. S2RDF [33] uses the in-memory system Spark to store the RDF data using vertical partitioning combined with semi-join materialization and then translates the SPARQL query to Spark SQL [7].

Regarding in memory join processing, a lot of research has been concentrated on cache friendly methods, such as the radix hash join [18], and also into taking advantage of hardware features such as the SIMD vectorized instructions for efficient parallel sort-merge joins [4, 17]. These works consider the setting of relational data with arbitrary number of columns, where a single join has to be performed on previously non indexed columns and sorting or hashing is a serious overhead that has to be performed in parallel. Instead, our work is tailored for RDF graphs, as it exploits initial ordering of both subject and object RDF columns and partial ordering of subsequent joins for pipelining multiway joins, such that it completely avoids hashing or sorting during query execution. Exploiting partial ordering of values in a column has been used by main memory systems in the form of zone-maps [28, 35] where additional statistics about each such zone have to be maintained in order to skip scanning certain areas. Adaptivity during run-time regarding the decision of scanning a base relation or use a secondary index has been studied in [9, 10] for disk-based systems.

Regarding centralized parallel in memory RDF processing, to the best of our knowledge there is no work concentrating on query processing. RDFox [21] and Inferray [38] are both systems that aim at parallel in memory computation and materialization of RDF inferences. This can be thought of as a preprocessing step prior to querying. Although RDFox offers query evaluation, it seems that is not the focal point of the system and for such queries there is no support for intra-query parallelism, that is each query is evaluated in a single thread. In [12] several variations of the disk based RDF-3X are presented, such that they allow parallel join evaluation. From the experimental results it is shown that depending on the query, there is no clear variation that has better performance, whereas for some queries the original version is better, as parallel evaluation prohibits the usage of the sideways information passing optimization in RDF-3X. Also, their approach works by parallelizing each join separately and demands communication and synchronization costs.

3 PHYSICAL DATA STORAGE

In this section we present our physical data storage and give an overview of the join method that allows incorporation of parallelism. First, following the common practice used by many systems, we use dictionary encoding, by assigning an integer value to each value encountered in the RDF data. We use common numbering for values appearing in the subject and object positions and a different numbering for values appearing in the property position, but for ease of presentation here we assume common numbering for all values. Thus, after parsing of an RDF dataset that contains N distinct values, our dictionary will contain integer IDs from 1 to N. Then, we apply vertical partitioning [1] to create a separate two-column table for each property defined in the data. We keep two replicas of each two-column table, the first sorted on subject and then on object, and the second sorted first on object and then on subject. Given that a property P is assigned to integer i from our dictionary encoding, we will refer to the first replica of two-column table for P as $prop_i$ and to the second replica as $prop_{i^-}$ and we will call the tables first sorted on subject *S-O tables* and tables first sorted in object *O-S tables*.

Consider for example the following RDF data (IRIs are omitted):

```
ProfessorA teaches Mathematics
ProfessorB teaches Chemistry
ProfessorC teaches Literature
ProfessorA teaches Physics
ProfessorA worksFor University1
ProfessorB worksFor University2
ProfessorC worksFor University2
```

[1] http://hbase.apache.org/

[2] http://accumulo.apache.org/

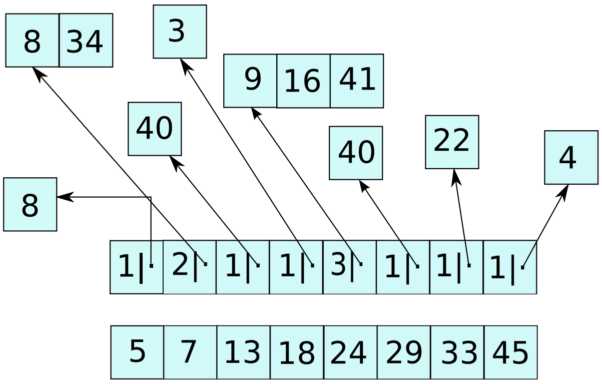


Figure 1: Example of Physical Data Storage for a Property Partition

The dictionary encoding of the data is given in Table 1. Using this encoding, the two-column tables $prop_2$ and $prop_9$ that correspond to properties $teaches$ and $worksFor$ will be created.

Table 1: Example of Dictionary Encoding

Integer	Value
1	ProfessorA
2	teaches
3	Mathematics
4	ProfessorB
5	Chemistry
6	ProfessorC
7	Literature
8	Physics
9	worksFor
10	University1
11	University2

For each table, we store a sorted integer array with the distinct subjects (for S-O tables) or distinct objects (for O-S tables). We also store a second array of same length with the first. Each position of this second array contains a pointer to a sorted integer array and an integer denoting the length of this array. This is a pointer to the objects (for S-O tables) or subjects (for O-S tables) that correspond to the subject (respectively object) located at the same position of the first array. The reason that we keep two separate arrays has simply to do with compactly storing the integers of the first array and improving spatial locality during the join processing. We also keep track of the length of the first array, using an array of length $2 * (number\ of\ properties)$ that contains this information for all properties. Getting this information involves a simple lookup at a specific position, for example, to get the number of subjects for $prop_7$, we should look at position $2 * 7$, whereas to get the number of objects for $prop_{7^-}$ we should look at position $(2 * 7) + 1$.

Figure 1 contains an example of physical storage for a property table. Given that the specific table is for property $prop_3$, then it contains the following triples: 5 $prop_3$ 8, 7 $prop_3$ 8, 7 $prop_3$ 34, 13 $prop_3$ 40, 18 $prop_3$ 3, 24 $prop_3$ 9, 24 $prop_3$ 16, 24 $prop_3$ 41, 29 $prop_3$ 40, 33 $prop_3$ 22, 45 $prop_3$ 4. Note that in order to avoid memory fragmentation, the different object arrays of this example can be allocated to a continuous memory area. In this case, instead of having different pointers for each position of the second array, we can keep a single pointer to the start of this memory area and only keep offsets in each position of the second array.

Our join method resembles an index-based nested loops join (or merge join when possible - this will be discussed later) that starts concurrently from different shards of the first table, and runs in parallel, by probing the next table to be joined for each tuple. In this way our method operates on left-deep query join trees as shown in Example 3.1.

Example 3.1. Consider a SPARQL query:

```
SELECT ?x ?y ?z
WHERE {
    ?x teaches ?z .
    ?x worksFor ?y .  }
```

Also suppose that the join order chosen by the optimizer (see Section 4.3) is the same with the order of the triples in the text of the query. This will be translated to a join $prop_2 \bowtie_{subject=subject} prop_9$. If there are two available threads, our algorithm will start concurrently scanning two different shards of $prop_2$. For each tuple encountered during this process, it will probe, using binary search, table $prop_9$. This process can be decomposed into completely independent tasks that start from different shards and operate on read-only common data, and thus it straightforward to be implemented using threads or separate processes with shared memory. It is even straightforward to be implemented on different machines using complete data replication and parallelize the query across machines without any communication.

Note that for the given query, the degree of parallelism depends on the number of different shards of the first table. For more selective queries a different strategy may be needed as shown in Example 3.2.

Example 3.2. Consider the following query, that contains an extra filter:

```
SELECT ?x ?z
WHERE {
    ?x teaches ?z.
    ?x worksFor University1 .   }
```

In this case, suppose that the optimizer chooses the inverse join order, as it is reasonable that the filter will limit the results of the second triple pattern. In this case, table $prop_9$ should be scanned first. One first observation is that instead of scanning the whole table, we can search for tuples where object is equal to 10. To do so it is better to use the replica that is first ordered by object. After we search $prop_{9^-}$ for $object = 10$, we obtain the vector of subjects that correspond to $object = 10$ (in our case it is only value 1). Then we start scanning this vector and probing table $prop_2$ using these values. In this way we do not obtain any level of parallelism for this query, as we start from a specific value of the first table. It is easy though to recover the parallelism, if we start scanning concurrently different shards of the vector that corresponds to $object = 10$. If the query contains a triple pattern with variable in the predicate position, then a union over all properties will be needed, but this is rarely encountered in real world queries[1]. In any case, if the number of distinct predicates encountered in the dataset is very large, an ID-Predicate index similar to the one use in [41] can be useful. Also note that the exact number of threads that will be used is independent of our physical data storage and can be decided on a per query basis after data loading in memory. In our current implementation (Section 5) we choose to execute each query with the same number of threads (optimally this should be equal to the number of available processing cores or greater in case hyper-threading is supported as shown in Section 5.2.3), but an extension such that very simple and selective queries could be executed with fewer resources is possible.

4 QUERY PROCESSING

The approach followed by RDF stores like RDF-3X and TriAD, is to take advantage of initial sorting of RDF triples, and perform merge joins when possible. Hash join is preferred when inputs are not sorted on the join key. On the other hand, the dynamic exchange operator approach always uses index-based nested loops aiming at low memory consumption and avoiding blocking operators. Our system uses a combination of these two approaches, by taking into consideration the following points:

- When both inputs are already sorted on the join key, merge join is preferable over hash join.
- For main memory systems, index-based nested loops (in our case in the form of binary searches over the inner table stored as an array) does not exploit data locality and also it is not amenable to efficient data prefetching due to conditional branching. Nevertheless, for very selective joins, it may still be faster than merge join.
- For RDF data processing, where the initial triples are sorted in all three subject, predicate and object columns, even if the whole input is not sorted on the join key of a subsequent join, large portions of the input can still be sorted as it is demonstrated in the following example.

Example 4.1. Consider the following SPARQL query:

```
SELECT ?x ?y
WHERE {
    ?x prop1 ?y .
    ?x prop2 ?z .
    ?z prop3 ?w .    }
```

If the selected join order is as shown in text of the query, S-O tables will be used for all properties. As shards of prop1 are scanned, for each thread of execution, *prop*2 will be probed for values sorted on ?x, but for the second join, probing *prop*3 will not in general be sorted on ?z. Nevertheless, for each distinct ?x, *prop*3 probing will still be sorted on ?z and if each subject of *prop*3 is connected to many objects, it may be more efficient to avoid binary search on *prop*3 and switch to scanning for each distinct ?x.

A single join operator has been implemented in our system, that adaptively during run-time, for each search key, decides if it will switch to binary search (a behavior similar to index-based nested loops) or keep scanning the input in the form of sequential search, continuing from the position that the cursor has been left from a previous search (a behavior similar to merge join).

4.1 Adaptive Join Processing

Given a left-deep join tree produced from the optimizer, each worker starts scanning a shard of the first relation, or a specific shard of an object/subject vector of the first S-O/O-S relation in case a filter exists, and searching the subsequent relations for each produced tuple. The search procedure is presented in Algorithm 1. The algorithm takes as input a pointer to current cursor position (*cursor_position*), which corresponds to the position of the last accessed element for the array, and decides if it will use binary or sequential search. The *cursor_position* is updated each time for both successful and unsuccessful searches inside *Sequential_Search* and *Binary_Search* functions.

Obtaining an exact cost-model in order to take the correct decision is an involved process that needs to take into consideration factors such as the exact cache hierarchy, the size and bandwidth estimation for each cache level for both sequential access (scanning) and random access, cache line size, the replacement of cache entries from operations other than the join under consideration (for example subsequent joins of the same query) and the existence in cache of relevant entries from previous operations (for example scanning of the same relation in a previous query). Obtaining such cost models for hierarchical memory systems has been studied in [19], where cost functions are defined for basic access patterns and then combinations of these functions can be used to derive the cost of complex compound access patterns. As a prerequisite, specific hardware measurements should be known, which can be obtained through a separate calibration program that estimates cache and CPU characteristics.

In our case, decision has to be made during runtime for each produced tuple and each join of the query. Instead of using an analytical cost model, we opt for a fast and lightweight method using two assumptions: a uniform distribution of integers in the first array of each table and that existing cache contents have an impact proportional to the cost of either binary search or scanning. The second assumption simply denotes that existing cache contents can improve both methods, but they will not change which the methods is more efficient in each case. For example, if binary search is preferable with completely empty cache, it will remain so independently of the cache contents and vice versa. As a result we base our decision on the difference between the last accessed element and the element that we are currently searching for. Specifically, we pass as argument to the algorithm a threshold which is computed during data loading for each table. This threshold takes into consideration an estimation about the maximum distance of the position of the last accessed element and the position of the element to be found in the array, in order for sequential search to be preferable. To switch from distance in the array to the actual arithmetic distance of the two numbers, we use the uniform distribution assumption, which leads to an estimation that the difference between an element and its subsequent one is $(array[size - 1] - array[0])/size$. Note that in Algorithm 1, if $Distance > Threshold$ then we could perform binary search using $CursorPosition$ instead of 0 as starting position, and if $Distance < -Threshold$ we could use $CursorPosition$ as the end position instead of $size$. In theory this reduces the steps needed from binary search, but in practice it is not efficient, as always performing binary search on the whole array leads to the array positions visited during the first steps to frequently occur in cache.

Regarding the determination of the threshold, a calibration process shown in Algorithm 2 is used. This process takes place after data loading, prior to query execution, and tries to determine a distance (called $WindowSize$) such that when searching for a value $ToFind$ in the $Array$ and the position of $ToFind$ is at distance $WindowSize$ from the position of the last accessed element ($CursorPosition$), then $BinarySearch$ and $SequentialSearch$ perform roughly the same. Specifically, the ratio of the larger to the smaller execution times of these two methods should be smaller than a value close to 1.0 which is specified in the input of the algorithm ($Threshold$). For each calibration step each process is called $NoOfSearches$ times, each time searching for a value estimated to be at distance equal to $CurrentWindowSize$ from the previous one. If the ratio is larger than the $Threshold$, calibration continues such that the window size is multiplied by this ratio (in case time spent on binary search is larger) or divided (otherwise). This calibration process is different from a calibration needed when using an analytical cost model, in the sense that we directly make an estimation for a value related to processing, instead of estimating values about several hardware characteristics. Once the calibration process terminates, we precompute the estimated value distance (corresponding to the position distance that we obtained) for each property, such that during query execution we only need to perform one integer subtraction, one absolute value computation and one comparison for each tuple (lines 2-3 of Algorithm 1.

Algorithm 1: Adaptively switching between binary and sequential search

```
1 Search ($Array, Value, CursorPosition, Threshold, Size$);
    Input  : $Array$: an array of integers (subjects of an S-O table
             or objects of an O-S table), $Value$: integer value to
             find, $CursorPosition$:pointer to current cursor
             position, $Threshold$: integer, $Size$: size of array
    Output : nonnegative integer corresponding to the position of
             $Value$ in $Array$ or a negative integer if $Value$ is not
             present in the $Array$
    Uses   : $Binary\_Search(Array, Value, CursorPosition, Size)$, $Sequential\_Search(Array, Value, CursorPosition, Size)$
2 $Distance := Array[CursorPosition] - Value$;
3 if $|Distance| <= Threshold$ then
4     return $Sequential\_Search(Array, Value, CursorPosition, Size)$;
5 else
6     return $Binary\_Search(Array, Value, CursorPosition, Size)$;
7 end
```

4.2 ID-to-Position Index

Our join method takes advantage of initial sorting and performs cache-friendly joins even when only a partial order of input triples is possible, but when ordering does not help we must resort to binary search. In this section we describe the structure of an ID-to-Position index that is used to avoid binary search and directly locate the position of a given integer on the property array. A separate such ID-to-Position index must be built for each S-O or O-S table, but its usage is auxiliary, in the sense that our system can operate without all or some of these indexes. Given an RDF dataset with N distinct values and a corresponding dictionary with IDs from 1 to N, in order to directly locate the position of a given value in a table, we need to store an integer array of length N, such that the value at index p denotes the exact position at the table where it is located the resource whose ID value according to the dictionary is p, or a special value to denote absence of the specific resource from the table.

For example, given the property shown in Figure 1 and supposing that the maximum ID contained in the dictionary is 45, we would need an array of integers with length 45, such that at position 5 of the array we would have the value 0, at position 7 the value 1, at position 13 the value 2 and so on for positions $18, 24, 29, 33$ and 45, and all other position of the array would have a value denoting absence. If we use M-byte integers, then for each table the memory requirement would be $M * N$ bytes. In order to save space, we use a different layout on out ID-to-Position index, such that we only use an integer to denote the position of the property table at specific intervals, and for all other positions we use a bit value to simply denote presence or absence of value from the property table. Finding the exact position for a value requires reading the previous integer and then counting bits set to 1 up to the position of the ID-to-Position Index corresponding to the value. For example, if we choose the interval to be equal to 8, then our index will store the integer -1 at start, followed by bit values $0, 0, 0, 0, 1, 0, 1, 0$, then integer value 1 and bit values $0, 0, 0, 0, 1, 0, 0, 0$, then integer value 2 and bit values $0, 1, 0, 0, 0, 0, 0, 1$, then integer value 4 and bit values $0, 0, 0, 0, 1, 0, 0, 0$, then integer value 5 and bit values $1, 0, 0, 0, 0, 0, 0, 0$ and finally integer value 6 and bit values $0, 0, 0, 0, 1$. If we want to find the position of value 29 at the property we can

Algorithm 2: Calibration Process

```
Calibrate ($Array, NoOfSearches, StartingWindowSize, Threshold$);
  Input  : $Array$: an array of integers (subjects of an S-O table
           or objects of an O-S table), $NoOfSearches$: number
           of times to run sequential and binary search in each
           calibration step, $StartingWindowSize$: initial
           window size used in first step of calibration,
           $Threshold$: A threshold ratio to stop calibration
  Output : integer corresponding to the window size such that
           if two values in array are longer apart then binary
           search is preferable
$NextWindowSize = StartingWindowSize$;
$AvgGap = (Array[Size - 1] - Array[0])/Size$;
do
    $WindowSize = NextWindowSize$;
    $TotalGap = AvgGap * WindowSize$;
    $PreviousSearchPosition = 0$;
    $StartTime = getTimeNow()$;
    $ToFind = Array[0]$;
    for $K \leftarrow 0$ to $NoOfSearches$ do
        $Binary\_Search(Array, ToFind, 0, \&PreviousSearchPosition)$;
        $ToFind+ = TotalGap$;
    end
    $TimeBinary = getTimeNow() - StartTime$;
    $toFind = Array[0]$;
    $PreviousSearchPosition = 0$;
    $StartTime = getTimeNow()$;
    for $k \leftarrow 0$ to $noOfSearches$ do
        $Sequential\_Search(array, toFind, \&PreviousSearchPosition)$;
        $ToFind+ = TotalGap$;
    end
    $TimeScan = getTimeNow() - StartTime$;
    $TimeDiff = |TimeBinary - TimeScan|$;
    if $TimeBinary > TimeScan$ then
        $Fraction = TimeBinary/TimeScan$;
        $NextWindowSize = WindowSize * Fraction$;
    else
        $Fraction = TimeScan/TimeBinary$;
        $NextWindowSize = WindowSize/Fraction$;
    end
while $Fraction > Threshold$;
return WindowSize;
```

directly check bit at position $((29 \div 8) + 1) * M * 8 + 29$. If bit is not set, then value is not present in property table. If bit is set we read integer value that starts at bit position $(29 \div 8) * M * 8 + (29 \div 8) * 8$ at the array and we add to this the number of bits that are set after this number for $29 \ mod \ 8$ positions. With this layout, given an interval A we only need $N/8 + ((N/A) * M)$ bytes. Also, given that the integer and the number of bits followed up to the next integer fit into a single cache line (with proper alignment of the index in the memory), we only need one memory access and some computation that can be done efficiently as a popcount operation in order to determine the position.

As an example, using the dataset LUBM 10240 described in Section 5, which contains about 1.4 billion triples, 17 distinct properties and about 336 million distinct resources, using 4-byte integers and choosing the interval to be 480 we only need 44.8 MB for each property, leading to a total memory usage of about 1.5 GB if we choose to create all possible indexes for $S - O$ and $O - S$ tables, in contrast to a memory requirement of 45.7 GB if we had used the simple layout.

Regarding modification of the join processing in case the ID-to-Position index is used, the only change that needs to be addressed is a different threshold resulted from calibration process. Specifically, since we anticipate that using the index will have better behavior in comparison with binary search, we need to estimate two different thresholds with regards as to when sequential search is preferable, with the threshold when ID-to-Position index is used being smaller than the threshold when binary search is used.

4.3 Join Ordering and Cost Estimation

As in RDF-3X and TriAD, we employ a bottom-up dynamic programming optimizer. As the level of parallelism during execution is determined by the number of threads, we assume that the benefit of each possible join order from parallelism will be a fixed proportion of its centralized cost, that is the execution cost if we consider that each property is consisting of a single shard. As a result of this assumption, we disregard parallelism during optimization. During cost estimation, we assume that a specific choice will be followed for all tuples of a join, either binary search or scanning. The latter will only take place when the join inputs are already fully sorted and it is estimated to be cheaper than binary search. Adaptivity during execution is expected to give a cost equal or lower to this estimation. For each property of a specific join order we choose to use the replica that leads to more selective results.

As selectivity estimation is not the focal point of this work, currently, in order to estimate the sizes of intermediate results we use equi-depth histograms. As it is known that often estimates based on such histograms may not be accurate especially in the case of RDF data [23], we precompute some cardinalities between pairs of properties during data loading and use these as a corrective step. We plan to implement more elaborate techniques for cardinality estimation in the future, like for example estimations based on characteristic sets [22] or RDF data summaries [36].

5 EXPERIMENTS

In-memory data storage and query processing for our prototype have been implemented in C as an extension of a SQLite, which is used as disk-based storage. Disk-based tables are created and saved during data import from RDF files. On application start-up the in-memory data structures are created reading from the tables. The dictionary can either be loaded in memory or kept in disk where for IRI-to-ID transformation (during query optimization) a clustered B+ tree on IRI is used and for ID-to-IRI transformation (during IRI construction of answer tuples) a clustered B+ tree on id. Our system is called through a wrapper written in Java, where also query parsing and optimization is implemented. We use the name *PARJ* for our implementation, which stands for Parallel Adaptive RDF Joins.

All experiments were conducted on a 16-core server with Intel E5-4603 processors at 2.20 GHz and 128 GB RAM running Debian 8. We used the popular Lehigh University Benchmark (LUBM) [13] and Waterloo SPARQL Diversity Test Suite (WatDiv) [6] benchmarks. All material required to reproduce the experiments is available online [3].

5.1 Setup

We use two sets of experiments: in the first one we test the efficiency of our approach in the single-thread setting. In this setup we use as competitors the in-memory RDF store RDFox (SVN version: 2776) and also RDF-3X [23] (version 0.3.8) for comparison with a state of the art disk-based system. The second setup is about multi-threaded execution. In the second setup we use as competitor the TriAD system which in [14] it is shown to outperform all competitors in the centralized parallel setting. We have used the optimized build for TriAD, as it is suggested in the installation manual.

Due to a hard-coded limit in the TriAD source code, we could not execute queries using more than 20 workers[4]. Note that in PARJ, each worker corresponds exactly to one thread, so given that hyper-threading is enabled, we found that the optimal performance was achieved when we used two threads for each processing core, resulting in 32 workers/threads in our testing machine. More details regarding the behavior of PARJ for different number of threads are given in Section 5.2.3. For TriAD it was not clear which number of workers should be the optimal, as this could be query depended. This is also the reason that we do not use TriAD in the single-thread setting. To have a better image and find the optimal setup, we executed TriAD with different number of workers, and we also modified the hard-coded limit and tried with up to 32 workers. For most queries, TriAD performance is degrading for more than 20 workers. From our testing we found that the overall best performance was achieved for 16 workers and this is the setup we used for TriAD in our experiments. Also, we present results for both TriAD settings: with summary mode enabled and disabled. For summary mode, we used the same number of partitions used in [14]: 200K for LUBM 10240 and 70K for WatDiv 1000.

Regarding result handling, as our intention is to concentrate in join processing, all systems were tested in the so called ”silent“ mode, that is we do not include the time for dictionary lookups and result tuple construction. In multi threaded execution this also means that we do not measure the time to aggregate the results together. Each query was executed 10 times and the average execution time is shown. We have deployed RDF-3X using an in-memory filesystem and as a result there is no need to report cold and warm cache times.

5.2 Results

We present results for scale 10240 of the LUBM benchmark in Table 2 (about 1.4 billion triples) and scale 1000 of the WatDiv benchmark (about 110 million triples). For WatDiv we used both

[3] https://github.com/dbilid/experiments

[4] This was verified with the TriAD implementors

basic test workload (Table 3) and incremental linear and mixed linear extensions of basic workload (Table 4). For WatDiv we generated all the queries proposed in the workloads. For LUBM we used the seven queries commonly used to test systems that do not perform reasoning tasks, which can be found in [42], and are labeled LUBM1-LUBM7, and we also used three extra queries from [26] (LUBM8-LUBM10). A timeout of 30 minutes was used for all queries.

Regarding single thread execution, we first observe that RDFox is comparable to PARJ for some queries, but for other queries, especially for queries from the WatDiv incremental and mixed linear extensions, is highly inefficient. This confirms that this system is not optimized for query answering, but instead, it aims at efficient parallel materialization of RDF implications. Regarding RDF-3X, we can see that it performs more than one order of magnitude slower from PARJ for most queries. The reason is that despite the fact that it is deployed in an in-memory filesystem, its processing is oriented towards optimizing disk access, as it is not aware that it operates in memory. For example, it uses B+ trees to minimize the number of disk pages needed, it skips records with its sideways-information passing optimization only when it reads a new disk-page into memory, it uses compression on a per page basis and also its cost estimation is based on disk access. Nevertheless, there are some queries, for example queries in the ML-2 set or LUBM8, where RDF-3X outperforms the single-threaded PARJ execution. These are queries with large intermediate results, but only few final answers, where the record skipping using sideways information passing in RDF-3X results in substantial gains.

Regarding multi-thread execution we can see that for most queries the summary mode of TriAD is inferior to the simple mode, sometimes by a large margin. For example, for query LUBM 3 in Table 2 the execution time increases from 2 seconds to more than 15 seconds. For the specific query we saw that execution over the summary graph takes up most of the execution time. In any case, the results show that for parallel execution on a centralized environment the pruning from the graph summaries does not contribute to an important improvement which can justify the overhead of graph partitioning.

A comparison of PARJ with the best TriAD mode shows that we outperform TriAD by more than an order of magnitude for the average execution time of the LUBM 10240 queries: from 838 milliseconds for PARJ to 13263 for TriAD (Table 2). For basic WatDiv testing (Table 3), though TriAD performs slightly better for simple queries, PARJ performs better overall with a total average execution time of 11.27 ms (geomean: 7.76) whereas TriAD has a total average execution time of 13.95 (geomean: 6.8). For the more complex queries of WatDiv extended workloads (Table 4) PARJ clearly outperforms TriAD. For some queries the difference is more than two orders of magnitude. As an example, for query ML1-7 the time increases from 7 ms to 2154. The specific query contains a series of subject-object joins, which leads TriAD to perform blocking data transfers between workers and rehashing over large intermediate results, though the final result is relatively small.

Regarding the difference between the silent mode and the full result handling, we have executed all queries with full result handling (except from printing) in PARJ. That is we include answer tuple construction, dictionary lookups and sending all results to the coordinating thread. We do not include these results, as we saw that for most queries, usually with results up to a few thousand tuples, the difference is not important, but for queries with many million results the difference can be significant. This can be seen especially for query 2 from the LUBM benchmark (about 10M results) where execution time in multi threaded execution increases from 151 milliseconds in silent mode to 610 milliseconds in full result handling. The same holds for queries C3 (about 4.3M results) and IL-3-5 to IL-3-10 from WatDiv which have more than 50M results. Query IL-3-8 has by far the largest number of results (about 1.6 billion tuples with 9 columns). This is the reason that TriAD runs out of memory for the specific query, since even in silent mode, each worker keeps in memory all the results instead of using an iterator to send the results to the master (or discard the results in silent mode) as they are produced, as it is the approach used by PARJ. Execution times for the full result handling mode of PARJ are included in the online material to reproduce experiments.

Table 5: Impact of Adaptive Processing for LUBM 10240 and WatDiv 1000 (times in ms)-1 thread

Query	Binary	AdBinary	Index	AdIndex
LUBM1	22186	15454	16557	15369
LUBM2	2877	2443	2535	2437
LUBM3	6562	5491	6415	5338
LUBM4	5	7	7	5
LUBM5	1	1	1	1
LUBM6	2	2	2	3
LUBM7	12246	11866	9197	9213
LUBM8	15725	9782	10420	9899
LUBM9	77468	63586	58171	58082
LUBM10	22359	14892	16217	14606
Avg	15943	12352	11952	11495
Geomean	1034	892	898	864
Watdiv1000 Avg	8439	8003	5013	4869
WatDiv 1000 Geomean	33	28	25	23

5.2.1 Effect of Runtime Join Optimization. In order to examine the effect of our adaptive join method, we have executed the queries of both datasets using four different strategies as shown in Table 5. For WatDiv benchmark we only report the average and geometric mean of all execution times. In the first (Binary) column we report the execution times when we always use binary search. In the second column (AdBinary) we use our adaptive join method in order to switch from binary to sequential search. In third column (Index) we always use the ID-to-Position index, whereas in the last column (AdIndex) we use the adaptive join method in order to switch from ID-to-position index to sequential search. One can observe that the impact of the adaptive join method is more important when binary search is employed (comparison of first and second column), whereas when the ID-to-Position index is used (comparison between third and fourth column) its contribution to better performance is smaller. This is in line with the result of our calibration method, where when binary search is used, the result threshold is about 200 positions, whereas when ID-to-Position index is used the threshold is about 20 positions. Also, it seems that the impact is more important for LUBM queries, where in case of binary search it leads to a decrease of 23% in average execution time. The reason for that is that the average execution time for WatDiv queries is heavily affected by the IL-3 queries, where the impact of the adaptive method is not important, as sequential search can rarely be used in these queries. This is also the reason for the great reduction in average execution time of WatDiv queries when the ID-to-Position index is used, as the aforementioned queries are greatly profit from the index.

Table 2: Results for LUBM 10240 (times in ms)

	Single Thread			Multi-Thread		
	PARJ	RDFox	RDF-3X	PARJ-32	TriAD	TriAD-SG 200K
LUBM1	15369	96677	1329510	800	4188	4467
LUBM2	2437	40368	21870	151	965	1101
LUBM3	5338	136554	23179	605	2004	15243
LUBM4	5	1	8	10	12	5
LUBM5	1	1	6	4	2	2
LUBM6	3	3	190	5	95	5
LUBM7	9213	31180	68769	473	13400	14125
LUBM8	9899	44144	6485	1336	2838	3906
LUBM9	58082	187192	208839	4014	42932	32982
LUBM10	14606	26690	51235	982	65925	41510
Avg	11495	56281	171009	838	13263	11334
Geomean	864	2536	5581	180	1071	881

Table 3: Results for WatDiv Basic Workload scale 1000 (times in ms)

	Single Thread			Multi-Thread		
	PARJ	RDFox	RDF-3X	PARJ-32	TriAD	TriAD-SG 200K
L1	5	5	40	10	3	5
L2	8	43	30	5	5	6
L3	2	244	13	4	2	3
L4	3	7	19	4	2	8
L5	9	57	40	6	3	46
Avg	5	71	28	6	3	14
Geomean	5	29	26	5	3	8
S1	49	1209	18	47	34	116
S2	3	284	27	3	4	17
S3	4	17	7	3	2	18
S4	4	153	10	5	5	29
S5	4	1*	14	4	4	20
S6	1	5	8	5	2	3
S7	1	695	7	5	2	3
Avg	9	338*	13	10	8	29
Geomean	4	61*	12	6	4	15
F1	5	24	15	6	5	19
F2	12	153	27	10	37	13
F3	3	59	73	9	29	74
F4	56	249	83	19	9	66
F5	3	10	108	7	40	58
Avg	16	99	61	10	24	46
Geomean	8	56	48	9	18	37
C1	21	50	140	12	39	598
C2	76	178	441	16	40	1574
C3	266	4810	127	45	43**	527**
Avg	121	1679	236	24	41**	900**
Geomean	75	350	199	21	41**	792**

* RDFox returns an empty result-set for query S5, whereas the correct answer is not empty.

** TriAD returns only distinct answers for query C3, even though modifier DISTINCT is not present in the SPARQL query. The number of results returned is only 8162 instead of 4335801.

5.2.2 Effect of ID-to-Position Index. We now proceed to describe the evaluation of our ID-to-Position Index compared to standard binary search using the LUBM 10240 dataset in the single-thread setting. Table 6 shows the number of binary searches and the number of sequential searches which were performed using the decision of our adaptive join method, using a threshold of about 200 computed with our calibration algorithm. The fact that sequential searches heavily outnumber binary searches provides a strong indication that ordering is present in the RDF dataset. In order to compare our index with binary search, we kept the threshold the same as computed in the case of binary search, and executed the queries by performing our index based lookup instead of binary search, measuring the exact number of total execution cycles used in the index lookup or binary search procedure each time, as well as the cache misses for each cache level. If we exclude queries no 1 and 3-6, as they nearly perform only sequential searches, we can see that our ID-to-Position index results in more than 30% decrease in total execution cycles and similar or larger decrease in the number of cache misses for all levels of cache hierarchy.

Table 4: Results for WatDiv Incremental and Mixed Linear Workloads scale 1000 (times in ms)

	Single Thread			Multi-Thread		
	PARJ	RDFox	RDF-3X	PARJ-32	TriAD	TriAD-SG 200K
IL-1 5	3	27617	1339	5	584	5082
IL-1 6	4	204898	1832	4	1482	11814
IL-1 7	8	669099	1272	7	1862	14950
IL-1 8	3	700199	1633	5	1615	21238
IL-1 9	26	728518	1396	11	630	23844
IL-1 10	29	734363	1923	9	618	25752
Avg	12	510782	1566	7	1132	17113
Geomean	8	335194	1546	6	1002	15068
IL-2 5	2	6574	1525	6	476	5340
IL-2 6	5	62149	2046	4	952	11156
IL-2 7	2	78211	1794	3	344	58749
IL-2 8	4	80453	1865	16	1148	62448
IL-2 9	9	86995	1998	6	1062	67045
IL-2 10	4	87872	1867	5	1093	70658
Avg	4	67042	1849	7	846	45899
Geomean	4	51948	1841	6	770	31807
IL-3 5	13259	187101	542948	1494	11195	17093
IL-3 6	58379	397964	357310	7070	13603	25492
IL-3 7	23208	342533	Timeout	1192	1809	23492
IL-3 8	71918	1214564	Timeout	4903	Out Of Memory	Out Of Memory
IL-3 9	26437	966919	Timeout	2082	7182	39462
IL-3 10	41867	951513	175247	1882	8118	46593
Avg	39178	676766		3104		
Geomean	33565	552681		2496		
ML-1 5	2	11481	163	2	56	374
ML-1 6	2	2	83	2	33	1152
ML-1 7	1	1	728	7	2154	4646
ML-1 8	2	1	824	4	103	2018
ML-1 9	5	98058	994	4	198	11766
ML-1 10	4	14111	1482	3	930	9841
Avg	3	20609	712	4	579	4966
Geomean	2	178	478	3	206	2786
ML-2 5	3175	1136335	936	201	413	1849
ML-2 6	2	12182	166	5	92	1041
ML-2 7	121	27151	678	15	296	895
ML-2 8	69	818424	2863	19	1996	24500
ML-2 9	4335	919541	282	259	330	1587
ML-2 10	52	849283	1952	9	728	32449
Avg	1292	627153	1146	85	643	10387
Geomean	151	249327	741	30	419	3599

Table 6: Number of binary searches and sequential searches for LUBM10240 chosen by out adaptive join method

Query	#Binary	#Sequential	Binary Search				ID-to-Position Index			
			Cycles	L1 Misses	L2 Misses	L3 Misses	Cycles	L1 Misses	L2 Misses	L3 Misses
LUBM1	1	107525748	2236	130	49	9	3135	102	43	8
LUBM2	204795	10854018	502M	26.7M	10.8M	3.5M	355M	18.3M	4.4M	543K
LUBM3	1	33169741	2401	140	50	8	4175	139	42	3
LUBM4	4	68	38745	666	368	235	16862	469	182	34
LUBM5	1	10	2423	94	29	0	2395	162	83	5
LUBM6	1	570	2033	106	26	0	2003	130	48	0
LUBM7	2257238	28768005	2.95B	254M	80.1M	2.30M	2.12B	211M	58.9M	1.08M
LUBM8	8645	84755793	17.4M	1.20M	682K	84.1K	11.2M	841K	351K	21.7K
LUBM9	409590	351307982	1.06B	53.6M	19.7M	2.92M	655.7M	39.1M	11.18M	639.7K
LUBM10	558279	116015419	1.22B	66.7M	24.2M	2.98M	798.2M	50.76M	12.7M	634.3K

5.2.3 Scalability. In this section we experimentally show the scalability of PARJ with regard to a varying number of threads and varying dataset size. As far as the first issue is concerned, we can already observe from Section 5.2 and specifically from Tables 2, 3 and 4, that running PARJ in multi threaded mode with 32 threads performs on average about 15 times better than the single thread version, but for the simple queries, when execution time is less than few tens milliseconds, multi-threaded execution does not seem to provide important gains. There are two reasons for that. The first one is the overhead of spawning multiple threads and

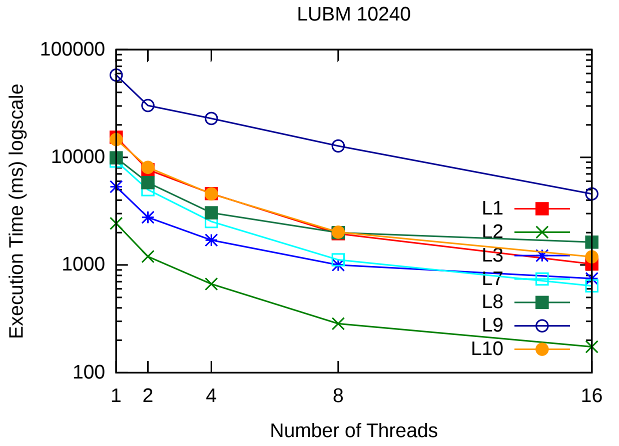


Figure 2: LUBM 10240 execution times in ms for different number of threads

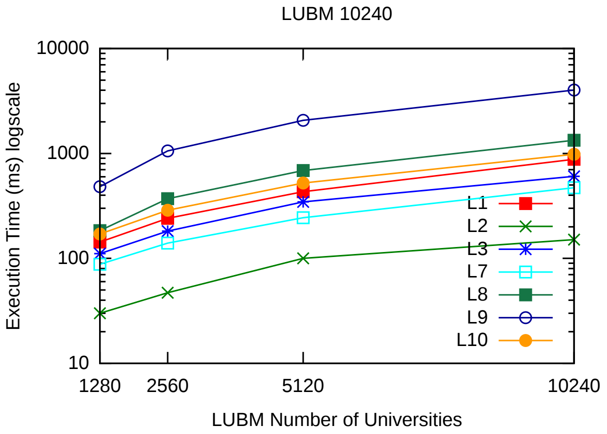


Figure 3: LUBM 32 threads execution times in ms for different dataset sizes

the second is that query parsing and optimization take up a large fragment of the total execution time, which cannot be avoided in multi-threaded execution. The best example of this is query S1 from WatDiv benchmark which is a star join query with 9 triple patterns and more than 40 milliseconds of the reported time of 49 milliseconds is spent on producing the join order in the optimizer.

In order to better examine the behavior of PARJ for a varying number of threads we have executed the queries from LUBM benchmark for scale 10240 with 1, 2, 4, 8 and 16 threads as shown in Figure 2. We exclude from this presentation simple and very selective queries L4, L5 and L6 that do not appear to improve from parallelism, since already in the single-threaded execution their execution time is only a few milliseconds, much of which is due to query parsing and optimization. On the other hand, complex queries L1, L3, and L7-L10, and also the simple but not selective query L2 show large and nearly linear improvement. The reason that we do not show results beyond 16 threads in Figure 2 has to do with the capabilities of our testing machine, which has exactly 16 processing cores. As stated before, best results were obtained with 32 threads as hyper-threading was enabled, but improvement from 16 to 32 threads cannot be evaluated and interpreted reliably for the specific scalability experiment, as here we aim to examine the behavior of PARJ for a varying number of threads given that the underlying hardware can provide full processing resources to each thread.

We have also examined the scalability of our system for a varying dataset size. Findings in Figure 3 show a similar situation for a varying number of universities in the execution with 32 threads, confirming the excellent scalability of PARJ.

5.2.4 Comparison With Distributed RDF Stores. A comparison of a parallel centralized system with distributed systems is not straightforward, as many factors come into play in order to have a result that will be as fair and complete as possible. In this section we attempt some first comparison of PARJ with existing RDF stores based on a recently published survey [2] and we plan to further investigate this issue experimentally in the future. The aforementioned survey presents an experimental comparison of 12 distributed systems designed for shared-nothing clusters, chosen as the most competitive and innovative from a variety of approaches and characteristics. The experiments were performed on a cluster with 12 servers, each with 148GB of memory and 24 cores, using, among others, the LUBM 10240 (only queries LUBM1-LUBM7) and WatDiv 1000 (only basic workload) benchmarks. For both these benchmarks the single server results of PARJ (in the full result handling mode) are comparable with the faster of the reported systems which is the non-adaptive version of AdPart (the adaptive version is not included in the results of [2]). Specifically, the average and geometric mean of execution times for first seven queries of LUBM 10240 are 918 and 75 milliseconds respectively (compared with 419 and 103 for PARJ in full result handling mode) whereas the geometric means for the 4 query categories of the basic workload of WatDiv 1000 are 9, 7, 160 and 111 milliseconds (compared with 9, 10, 12 and 48 for PARJ in full result handling mode).

6 CONCLUSIONS AND FUTURE WORK

We have presented a centralized in-memory system for parallelizing join processing on RDF graphs. We have shown that our design has excellent scaling capabilities and performance. For future work, we first plan to perform a more thorough experimental comparison with distributed RDF stores. As we mentioned, it is straightforward to extend PARJ to a "cluster" version through full replication, such that during query execution each worker start processing from different initial shard. We plan to implement and compare this version with the current state of the art distributed systems. We also want to further evaluate PARJ on a high-end server with larger available memory, in order to load and process larger RDF graphs. Based on the scaling capabilities presented during the experiments, we anticipate that our approach will be able to efficiently handle such datasets.

Furthermore, we plan to investigate the efficient incorporation of query answering with respect to class and property hierarchies into our join approach. RDF Schema (RDFS) as well as more expressive ontological languages like OWL-2 QL define ontological constraints on top of RDF graphs, such that SPARQL query answering must be extended by taking into consideration the corresponding semantics in order to provide the user with the complete answers. Deep and wide class and property hierarchies pose a serious performance issue for all systems that perform query answering with respect to such *entailment regimes*. Materializing all implied assertions, as it is the case in RDFS reasoning with forward chaining, with respect to these hierarchies may lead to data size many times larger than the original, something that may not be viable especially for an in-memory system. On the other hand, using RDFS reasoning with backward chaining may lead to complicated queries. We plan to extend our join method to handle such queries, by "unioning" tables during the pipelined join execution in order to provide complete answering with respect to hierarchies, without the need to materialize the implications.

ACKNOWLEDGMENTS

We would like to thank Stratos Idreos for his comments on an early version of this work and the anonymous reviewers for their input and corrections. The present work was co-funded the European Union's Horizon 2020 research and innovation programme under grant agreement No 825258 and by the European Union (ERDF) and Greek national funds through the Operational Program "Competitiveness, Entrepreneurship and Innovation", under the call "RESEARCH-CREATE-INNOVATE" (project code:T1EDK01000).

REFERENCES

[1] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. 2007. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. 411–422.

[2] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. 2017. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. *PVLDB* 10, 13 (2017), 2049–2060.

[3] Razen Al-Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. 2016. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDB J.* 25, 3 (2016), 355–380.

[4] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the VLDB Endowment* 5, 10 (2012), 1064–1075.

[5] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. 2001. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *SemWeb*.

[6] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified Stress Testing of RDF Data Management Systems. In *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*. 197–212.

[7] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD Conference*. ACM, 1383–1394.

[8] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. 2007. DBpedia: A Nucleus for a Web of Open Data. In *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*. 722–735.

[9] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. 2015. Smooth scan: Statistics-oblivious access paths. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 315–326.

[10] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. 2018. Smooth Scan: robust access path selection without cardinality estimation. *The VLDB Journal* (2018), 1–25.

[11] Jin-Hang Du, Haofen Wang, Yuan Ni, and Yong Yu. 2012. HadoopRDF: A Scalable Semantic Data Analytical Engine. In *Intelligent Computing Theories and Applications - 8th International Conference, ICIC 2012, Huangshan, China, July 25-29, 2012. Proceedings*. 633–641.

[12] Jinghua Groppe and Sven Groppe. 2011. Parallelizing join computations of SPARQL queries for large semantic web databases. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 1681–1686.

[13] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.* 3, 2-3 (2005), 158–182.

[14] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. 2014. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. 289–300.

[15] Jiewen Huang, Daniel J. Abadi, and Kun Ren. 2011. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB* 4, 11 (2011), 1123–1134.

[16] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35, 1 (2012), 40–45.

[17] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1378–1389.

[18] Stefan Manegold, Peter Boncz, and Martin Kersten. 2002. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering* 14, 4 (2002), 709–730.

[19] Stefan Manegold, Peter Boncz, and Martin L Kersten. 2002. Generic database cost models for hierarchical memory systems. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 191–202.

[20] Jaeseok Myung, Jongheum Yeon, and Sang-goo Lee. 2010. SPARQL basic graph pattern processing with iterative MapReduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*. ACM.

[21] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. 2015. RDFox: A highly-scalable RDF store. In *International Semantic Web Conference*. Springer, 3–20.

[22] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*. IEEE Computer Society, 984–994.

[23] Thomas Neumann and Gerhard Weikum. 2009. Scalable join processing on very large RDF graphs. In *SIGMOD Conference*. ACM, 627–640.

[24] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*. ACM, 1099–1110.

[25] Nikolaos Papailiou, Dimitrios Tsoumakos, Ioannis Konstantinou, Panagiotis Karras, and Nectarios Koziris. 2014. H_2RDF+: an efficient data management system for big RDF graphs. In *SIGMOD Conference*. ACM, 909–912.

[26] Anthony Potter, Boris Motik, Yavor Nenov, and Ian Horrocks. 2016. Distributed RDF Query Answering with Dynamic Data Exchange. In *International Semantic Web Conference (1) (Lecture Notes in Computer Science)*, Vol. 9981. 480–497.

[27] Roshan Punnoose, Adina Crainiceanu, and David Rapp. 2015. SPARQL in the cloud using Rya. *Inf. Syst.* 48 (2015), 181–195.

[28] Wilson Qin and Stratos Idreos. 2016. Adaptive data skipping in main-memory systems. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2255–2256.

[29] Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu. 2011. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *ESWC (2) (Lecture Notes in Computer Science)*, Vol. 6644. Springer, 46–61.

[30] Kurt Rohloff and Richard E. Schantz. 2010. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. In *PSI EtA*. ACM, 4.

[31] Kurt Rohloff and Richard E. Schantz. 2011. Clause-iteration with MapReduce to scalably query datagraphs in the SHARD graph-store. In *DICT@HPDC*. ACM, 35–44.

[32] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. 2011. PigSPARQL: mapping SPARQL to Pig Latin. In *SWIM*. ACM, 4.

[33] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. 2016. S2RDF: RDF Querying with SPARQL on Spark. *PVLDB* 9, 10 (2016), 804–815.

[34] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. 2014. Adoption of the Linked Data Best Practices in Different Topical Domains. In *Semantic Web Conference (1) (Lecture Notes in Computer Science)*, Vol. 8796. Springer, 245–260.

[35] Dominik Ślezak, Jakub Wróblewski, Victoria Eastwood, and Piotr Synak. 2008. Brighthouse: an analytic data warehouse for ad-hoc queries. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1337–1345.

[36] Giorgio Stefanoni, Boris Motik, and Egor V Kostylev. 2018. Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1043–1052.

[37] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *VLDB*. ACM, 553–564.

[38] Julien Subercaze, Christophe Gravier, Jules Chevalier, and Frederique Laforest. 2016. Inferray: fast in-memory RDF inference. *Proceedings of the VLDB Endowment* 9, 6 (2016), 468–479.

[39] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: sextuple indexing for semantic web data management. *PVLDB* 1, 1 (2008), 1008–1019.

[40] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. 2003. Efficient RDF Storage and Retrieval in Jena2. In *SWDB*. 131–150.

[41] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: a fast and compact system for large scale RDF data. *Proceedings of the VLDB Endowment* 6, 7 (2013), 517–528.

[42] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB* 6, 4 (2013), 265–276.

Pivoted Subgraph Isomorphism: The Optimist, the Pessimist and the Realist

Ehab Abdelhamid
Datometry Inc.
ehab@datometry.com

Ibrahim Abdelaziz
IBM Research
ibrahim.abdelaziz1@ibm.com

Zuhair Khayyat
Lucidya LLC
zuhair@lucidya.com

Panos Kalnis
KAUST
panos.kalnis@kaust.edu.sa

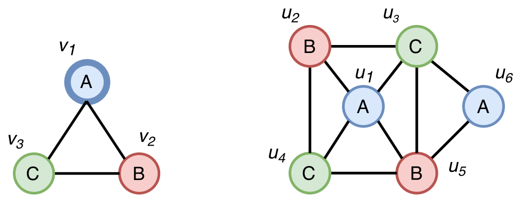
ABSTRACT

Given query graph Q with pivot node v, Pivoted Subgraph Isomorphism (*PSI*) finds all distinct nodes in an input graph G that correspond to v in all matches of Q in G. PSI is a core operation in many applications such as frequent subgraph mining, protein functionality prediction and in-network node similarity. Existing applications implement PSI as an instance of the general subgraph isomorphism algorithm, which is expensive and under-optimized. As a result, these applications perform poorly and do not scale to large graphs. In this paper, we propose SmartPSI; a system to efficiently evaluate PSI queries. We develop two algorithms, called optimistic and pessimistic, each tailored for different instances of the problem. Based on machine learning, SmartPSI builds on-the-fly a classifier to decide which of the two algorithms is appropriate for evaluating each graph node. SmartPSI also implements a machine learning-based optimizer to generate low-cost execution plans. Our experimental evaluation with large-scale real graphs shows that SmartPSI outperforms existing approaches by up to two orders of magnitude and is able to process significantly larger graphs. Moreover, SmartPSI is shown to achieve up to 6 times performance improvement when it replaces standard subgraph isomorphism in the state-of-the-art distributed frequent subgraph mining system.

1 INTRODUCTION

Graphs are widely used to model information in a variety of real world applications such as social networks [31, 32], chemical compounds [20] and protein-protein interactions [25]. Finding all valid bindings of a particular query node is an important step in various application domains such as frequent subgraph mining [4, 37], protein functionality prediction [12], neighborhood pattern mining [15], query recommendation [16] and in-network node similarity [39]. This step is called Pivoted Subgraph Isomorphism (*PSI*); for query Q with pivot node v, PSI finds the set of distinct matches of v in an input graph G. For example, PSI query S $(\underline{v_1}, v_2, v_3)$ in Figure 1 has two matches of the pivot node v_1 in G: u_1 and u_6.

Existing applications [4, 13, 15, 16, 25] depend on subgraph isomorphism [9, 17, 30] to evaluate PSI-like queries. They use subgraph isomorphism to find all matches of the query and then project distinct nodes that correspond to the pivot node. In the example of Figure 1, subgraph isomorphism generates 5 intermediate results: $(\underline{u_1}, u_2, u_3)$, $(\underline{u_1}, u_2, u_4)$, $(\underline{u_1}, u_5, u_4)$, $(\underline{u_1}, u_5, u_3)$ and $(\underline{u_6}, u_5, u_3)$), in order to project u_1 and u_6 as results for the PSI query. We show in Table 1 the number of PSI results compared to the number of intermediate isomorphic subgraphs for various real graphs and query sizes (see Section 5 for details). Observe that the number of isomorphic subgraphs grows exponentially with the query size even for small and sparse datasets like Yeast, whereas the actual PSI results are significantly fewer. Consequently, the performance of applications that depend on subgraph isomorphism degrades rapidly with complex queries and larger graphs.



(a) PSI query S

(b) Input graph G

Figure 1: (a) A PSI query with v_1 as pivot, (b) input graph. The bindings of v_1 are u_1 and u_6

A possible approach to reduce the exponential number of matches in subgraph ismorphism-based solutions is to stop the search once a match of the pivot node is found for each candidate graph node. We demonstrate the effect of this optimization by proposing TurboIso$^+$ (Section 5.2) as a modified version of TurboIso [17] which is a highly optimized subgraph isomorphism solution. We show in Table 2 that TurboIso$^+$ is significantly faster than TurboIso for all query sizes. However, it remains computationally expensive since it cannot avoid the overhead of the sophisticated data structures used to compile all occurrences of the given query. As such, using subgraph isomorphism to solve PSI queries is unnecessarily expensive. Moreover, to the best of our knowledge, there is no existing research that has been devoted to efficiently answer PSI queries.

In this paper, we formalize the pivoted subgraph isomorphism problem and introduce the *optimistic* and *pessimistic* approaches for evaluating PSI queries efficiently. The optimistic approach is highly optimized to confirm that a graph node matches the pivot node in the query. It uses a greedy depth-first guided search to minimize the number of graph explorations required to prove that a particular node is indeed a matching node. If no match exists, it would have unnecessarily wasted resources to discover non-matching nodes. The pessimistic approach, on the other hand, is highly optimized to verify that a graph node does not match the given query. It is based on a random, non-guided, search that prunes adjacent nodes early depending on their neighbourhood.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.32

Table 1: Number of subgraph query matches using PSI vs. standard subgraph isomorphism

Dataset	Query	Query Size						
		4	**5**	**6**	**7**	**8**	**9**	**10**
Yeast	PSI	7×10^4	6×10^4	5×10^4	5×10^4	4×10^4	3×10^4	3×10^4
	Subgraph Iso.	1.3×10^7	1.2×10^8	1.1×10^9	7.4×10^9	5.8×10^{10}	6.0×10^{11}	2.8×10^{12}
Cora	PSI	2.1×10^5	1.9×10^5	1.4×10^5	1.2×10^5	1.1×10^5	0.9×10^5	0.8×10^5
	Subgraph Iso.	1.6×10^8	1.2×10^{10}	1.1×10^{12}	8.4×10^{13}	1.5×10^{15}	8.9×10^{16}	NA
Human	PSI	1.4×10^5	1.3×10^5	1.2×10^5	1.1×10^5	1.1×10^5	1.0×10^5	1.1×10^5
	Subgraph Iso.	2.6×10^{10}	7.6×10^{12}	NA	NA	NA	NA	NA

Early pruning helps the pessimistic algorithm to avoid unnecessary graph traversals. It detects non-matching nodes significantly faster than the optimistic approach, however, it has a higher cost when used for matching nodes.

To maximize the gain from these two methods, the optimistic method has to be used to validate nodes that match the query while the pessimistic method should be used to confirm other non-matching nodes. Unfortunately, we do not know which graph node matches the given PSI query prior to query evaluation, while it is counter-intuitive to use the proposed methods randomly on the input graph. To solve this problem, we propose SmartPSI, a solution that relies on machine learning to predict which method is best to evaluate each graph node. In SmartPSI, a classification model is trained to predict the type of each graph node (i.e., whether a node does or does not match the given query). Based on the prediction result, the appropriate method is picked accordingly; the optimistic method is used for matching graph nodes (predicted valid nodes) and the pessimistic is used for non-matching nodes (predicted invalid nodes). Notice that SmartPSI is an exact solution since the correctness of the result is guaranteed even when using the opposite evaluation, this is due to the fact that both algorithms traverse the whole search space in the worst-case scenario. SmartPSI is also coupled with a novel query optimizer, that recommends an efficient search order plan for evaluating each graph node. The last row of table 2 gives a hint on how the combined ideas implemented in SmartPSI, which are solely designed for PSI queries, perform compared to the more general solutions of subgraph isomorphism.

In summary, our contributions are:

- We introduce two methods for evaluating PSI queries, each method is optimized for a particular type of the input graph nodes.
- We propose a solution based on machine learning that, for each graph node, predicts the node type, picks the corresponding optimized evaluation method, and selects an efficient plan for query evaluation.
- We also propose a preemptive query processor that employs a detection and recovery technique at run-time to reduce the impact of incorrect predictions.
- We experimentally compare the performance of SmartPSI against the state-of-the-art subgraph isomorphism techniques. We show that SmartPSI is orders of magnitude faster and it scales to much larger queries and input graphs. Furthermore, we empirically show that our optimizations significantly improve the performance of a cutting edge frequent subgraph mining system.

The rest of the paper is organized as follows: Section 2 formalizes the problem and presents various PSI applications. Section 3 introduces two novel PSI evaluation methods. Section 4 presents a two-threaded baseline solution and describes machine learning based optimizations that overcome the limitations of the baseline. Section 5 presents the experimental evaluation. Section 6 surveys the related work while Section 7 concludes.

Table 2: Performance of PSI solutions on Human dataset (details in Section 5). SmartPSI is our proposed approach.

Query size	4	5	6	7
TurboIso	5.4 hrs	>24 hrs	>24 hrs	>24 hrs
TurboIso$^+$	14 min	30 min	45 min	2.4 hrs
SmartPSI	27 sec	47 sec	2 min	4.3 min

2 PIVOTED SUBGRAPH ISOMORPHISM

2.1 Definition

A labeled graph $G = (V_G, E_G, L_G)$ consists of a set of nodes V_G, a set of edges E_G and a labeling function L_G that assigns labels to nodes and edges. A graph $S = (V_S, E_S, L_S)$ is a *subgraph* of a graph G iff $V_S \subseteq V_G$, $E_S \subseteq E_G$ and $L_S(v) = L_G(v)$ for all $v \in V_S \cup E_S$. Subgraph isomorphism is the task of finding all occurrences of S inside G. It is a computationally expensive problem, known to be NP-Complete [14]; it needs to validate a huge number of intermediate and final results. PSI, which is also NP-Complete [15], relaxes subgraph isomorphism by focusing on finding all node matches of a particular query node (v_p), which is called a *pivot* node, inside the input graph G.

Definition 2.1. A pivoted graph is a tuple $\eth$={G, v_p}, where G is a labeled graph and $v_p \in V_G$ is called the pivot node of the pivoted graph $\eth$.

The pivot node (v_p) is the node of interest, and it is usually set by the user who creates the query.

Definition 2.2. A *pivoted subgraph isomorphism* of $\eth_S$=(S, v_p) to $\eth_G$={G, u_p} where $S = (V_S, E_S, L_S)$, $G = (V_G, E_G, L_G)$, $v_p \in V_S$ and $u_p \in V_G$, is an injective function $M : V_S \longrightarrow V_G$ satisfying (*i*) $L_S(v) = L_G(M(v))$ for all nodes $v \in V_S$, (*ii*) $(M(u), M(v)) \in E_G$ and $L_S(u, v) = L_G(M(u), M(v))$ for all edges $(u, v) \in E_S$, and (*iii*) $M(v_p) = u_p$.

Rather than finding all matches in subgraph isomorphism, PSI finds at most one occurrence per graph node u_p, which significantly reduces computation and memory overheads. The input graph node u_p is called a *valid* node for query $\eth_S$ if it matches v_p in at least one occurrence of S in G, otherwise it is called an *invalid* node. PSI has an important role in several applications [4, 13, 15, 16, 25]. In the following discussion, we highlight some of these applications.

2.2 PSI Applications

Frequent Subgraph Mining (FSM). FSM is an important operation for graph analytics and knowledge discovery [4, 5, 13]. Given an input graph and a support threshold, FSM evaluates the frequencies of a large number of candidate subgraphs to determine which one is frequent. Typically, this evaluation relies on finding the distinct input graph nodes that match their corresponding candidate subgraph nodes [11]. Current solutions [4, 13] employ subgraph isomorphism to conduct this step. Instead of using the expensive subgraph isomorphism algorithms, PSI can be employed to efficiently evaluate the frequencies. Using PSI instead of subgraph isomorphism results in a significant improvement in the overall efficiency of FSM techniques (see Section 5.5).

Function prediction in PPI networks. In the domain of protein-protein-interaction (PPI) networks, it is common to discover new proteins with unknown functions [12]. Knowledge about their functionality is essential to analyze these networks and discover valuable insights. In order to predict these functions, a set of significant patterns are extracted from the PPI network. Each protein with unknown function is then matched against the extracted patterns. A label that is matched to the node with unknown function represents a predicted function for this node. Each significant pattern is represented by a different pivoted subgraph query, and the matching task resembles a PSI query evaluation.

Discovering Pattern Queries by Sample Answers. Knowledge bases are effective in answering questions. Nevertheless, it is challenging for a user to issue a query that follows the schema of the knowledge base. Han et al. [16] proposed a query discovery approach to assist users by recommending candidate queries. This approach is based on having a sample answer set (i.e., a set of nodes which the user thinks they match his query). The first step finds a set of queries that match the neighborhood around the given nodes. These queries represent recommended candidate queries. This step is conducted by a series of PSI operations which tries to filter out all queries that do not match any of the given answer nodes. Queries that pass the first step are then ranked and the top ones are recommended for the user.

Neighborhood Pattern Mining. Mining frequent neighborhood patterns [15] finds the set of frequent patterns that each originates from graph nodes with a particular label. Similar to FSM, candidate patterns are generated and evaluated to determine whether they are frequent. Though, the evaluation step is different. Given a specific label, each candidate pattern is evaluated by PSI to know the number of graph nodes that satisfy this pattern. Based on this process, interesting knowledge can be obtained regarding the common connectivity patterns found around each node label.

In-network Node Similarity. In many applications, it is very important to measure the similarity among graph nodes. These applications include role discovery [28], objects similarity [21] and node clustering [19]. Two nodes are similar if they have similar neighborhoods. There are several techniques for calculating nodes similarity. A recent approach [39] proposed a unified framework for measuring the similarity between two nodes. One of the proposed metrics is the maximum common pivoted subgraph that exists around the two nodes. This metric is extended into a more general similarity metric, which is used to compare the common pivoted subgraphs occurring in the neighborhoods of any two nodes. Such approach is a direct application of PSI.

3 THE OPTIMIST AND THE PESSIMIST

In this section, we propose an optimist and a pessimist evaluation mechanisms to improve PSI evaluation. The *optimistic* method is optimized to confirm that a candidate node is valid, i.e., matches the query pivot node. On the other hand, the *pessimistic* method is effective in proving that a candidate node is invalid. Since both methods require access to a neighborhood signature, we will first describe what a neighborhood signature is and how it is calculated (Sections 3.1 and 3.2). Then, we introduce each PSI method in Sections 3.3 and 3.4.

3.1 Neighborhood Signature

The *neighborhood signature* of a graph node represents how this node interacts with its neighbors. A good neighborhood signature should be concise and effective in pruning/matching of a graph node. The literature introduces several signature definitions [7, 23, 42, 43]. We propose to use a label propagation technique inspired by the work of proximity pattern mining [23]. This technique assigns a list of weights to each graph node. Each weight corresponds to a label and reflects the proximity of that label to the graph node.

Definition 3.1. Let u be a node in the graph G. The neighborhood signature of u within a distance D is the set of pairs:

$$NS_u{}^D = \{(l_1, w_1^D), (l_2, w_2^D)...(l_n, w_n^D)\}$$

where each l_i is a node label and each weight $w_i^D \in R$.

Label weights are propagated from nearby nodes such that for each label l_i, its weight w_i^D is calculated as:

$$w_i^D = \sum_{d=0}^{D} 2^{-d} \times C_u(l_i, d)$$

where D is the maximum propagation depth, and $C_u(l_i, d)$ is the number of nodes with label l_i having a shortest distance d from node u. Notice that the signature $NS_u{}^D$ is small since its size depends on the number of distinct labels that appear in the graph. It is also effective as the used weights reflect how neighbors are structured around the node. In this work, we use the same maximum propagation depth for query graphs and the input graph. Thus, we will omit the use of the superscript D where it is not needed.

Example: Consider the data graph G in Figure 1, G contains three different labels; $\{A, B, C\}$. Assuming that the maximum propagation depth is set to 2, the signature of node u_1 is calculated as follows: (*i*) first, the label of u_1, which is 'A', is given a weight = 1. Consequently, the list of weights from distance 0 is: $\{(A, 1), (B, 0), (C, 0)\}$. (*ii*) For the first level of neighbors, there exist four nodes, two nodes are labeled 'B' and the other two are labeled 'C'. Therefore, the weight of 'B' in this case is 2×0.5 and the weight of 'C' = 2×0.5. Consequently, the list of propagated weights from this level is $\{(A, 0), (B, 1), (C, 1)\}$. (*iii*) There is only one node that is two hops away from u_1; i.e. u_6, which has label 'A', and thus the weight of 'A' = $1 \times 0.25 = 0.25$. Consequently, the propagated weights from this level = $\{(A, 0.25), (B, 0), (C, 0)\}$. (*iv*) Finally, $NS_{u_1}^2$ is the sum of weights from all levels, hence, $NS_{u_1}^2 = \{(A, 1.25), (B, 1), (C, 1)\}$.

Signature Computation. Traditional approaches [23] follow an exploration-based strategy to compute node signatures. It executes a breadth-first search algorithm iteratively till the maximum propagation depth is met. This approach is very simple to implement but has an exponential computational complexity ($O(|N|.|L|.d^D)$), where |**N**| is the number of nodes in the graph, |**L**| is the number of distinct node labels, **d** is the average node degree and **D** is the maximum traversal depth.

Optimization. To improve the performance of computing neighbourhood signatures, we propose a significantly faster approach based on matrix multiplication to compute all neighborhood signatures of the input graph. In comparison to the exploration-based technique [23], the computational complexity of our proposal is $O(|N|.|L|.d.D)$. The first step is to create a matrix NS^0 : $|N| \times |L|$. Each row $NS^0(n)$ represents the neighborhood signature of a node n and is initialized as: $NS^0(n) = [b_1, b_2, ...b_L]$ where $b_l = 1$ if $l \in L(n)$ and 0 otherwise. Then for the subsequent D iterations, the corresponding row for each node n is updated as follows:

$$NS^i(n) = NS^{i-1}(n) + \frac{1}{2}Adj(n) \times NS^{i-1}$$

where $Adj(n)$ is the adjacency matrix row that corresponds to node n. Notice that the label weights obtained by the iterative matrix multiplications may differ from the ones generated by the exploration based approach. This is because the iterative matrix approach considers neighbors labels multiple times through different paths, compared to one time through the shortest path in the previous approach. However, these weights are still based on the proximity of the labels that exist around the node under consideration.

Example: Consider the PSI query in Figure 2. Assuming a maximum propagation depth of 2, we first initialize NS^0 and Adj to be:

$$NS^0 = \begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

$$Adj = \begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{ccccc} v_0 & v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Then, we apply two iterations of the matrix multiplication to compute the results for NS^1 and NS^2:

$$NS^1 = \begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 1 & 1/2 & 0 & 0 \\ 1/2 & 3/2 & 1/2 & 0 \\ 0 & 3/2 & 1/2 & 0 \\ 0 & 1 & 1 & 1/2 \\ 0 & 0 & 1/2 & 1 \end{pmatrix} \end{array}$$

$$NS^2 = \begin{array}{c} \\ v_0 \\ \mathbf{v_1} \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 5/4 & 5/4 & 1/4 & 0 \\ \mathbf{1} & \mathbf{3} & \mathbf{5/4} & \mathbf{1/4} \\ 1/4 & 11/4 & 5/4 & 1/4 \\ 1/4 & 13/4 & 2 & 1 \\ 0 & 1/2 & 1 & 5/4 \end{pmatrix} \end{array}$$

Notice that row v_1 in NS^2 is the neighborhood signature of the query graph pivot node in Figure 2(a).

PSI query evaluation benefits from neighborhood signatures in two ways. First, graph traversals are guided towards the relevant graph nodes in order to efficiently reach a match for a given query. Second, significant pruning is achieved during the evaluation of graph nodes that do not satisfy the query.

3.2 Neighborhood Signature Satisfaction

The neighborhood signatures of two nodes can be utilized to quickly judge if they have the same neighbourhood and hence could be a match. A signature NS_i is said to satisfy another signature NS_j if for every $(l_j, w_j) \in NS_j$, there exists $(l_i, w_i) \in NS_i$ where $l_i = l_j$ and $w_i \geq w_j$. In Figure 1(b), u_1 has a neighborhood signature $NS^2_{u_1} = \{(A, 1.25), (B, 1), (C, 1)\}$. For query S in Figure 1(a), $NS_{v_1} = \{(A, 1), (B, 0.5), (C, 0.5)\}$. NS_{u_1} satisfies NS_{v_1} since the weights of all labels in NS_{u_1} are larger than their corresponding ones in NS_{v_1}.

Proposition 3.2. *Given a graph G, a graph node u and a pivoted query $\eth$={S, v_p}, where S is a subgraph and v_p is the pivot node, if NS_u does not satisfy NS_v, then u is an invalid node given the pivoted query $\eth$.*

Proof: We will use a proof by contradiction approach. Let NS_u do not satisfy NS_v. Let's assume a graph node u be valid for the pivot node v of subgraph S. Since NS_u do not satisfy NS_v. This implies that there exists at least one pair $(l_i, w_u) \in NS_u$ and a pair $(l_j, w_v) \in NS_v$, where $l_i = l_j$ and $w_u < w_v$. For w_u to be less than w_v, the contribution to the weight w_u from neighbor nodes with label l_i should be less than of neighbors of node v with the same label. The weight contribution relies on the number of neighbors labeled l_i and their distances. Accordingly, either the number of neighbor nodes of u with label l_i is less than that for neighbors of node v with the same label, or their distances are larger than their corresponding ones in the query. The last statement contradicts with the assumption that v is a valid node of u since this requires that each neighbor of v to have a distinct corresponding match with a neighbor of u. □

3.3 The Optimist

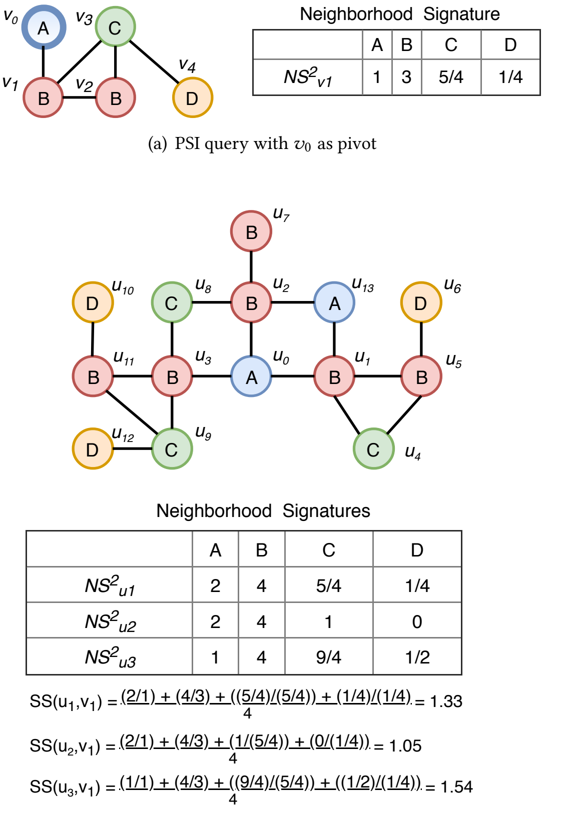
The optimistic algorithm uses a greedy depth-first guided search to quickly find a match for the given query. Its name is derived from the fact that it assumes that input candidate nodes are valid and optimizes the evaluation accordingly. The algorithm starts by generating a satisfiability score for each neighbor of the candidate node and then it sorts them based on the calculated scores descendingly. The satisfiability score measures the likelihood of a graph node being a match for its corresponding query node, it is calculated as:

$$SS(u, v) = avg_{(l,w_l) \in NS_v}(NS_u(l)/w_l)$$

For example, the satisfiability score of u_1 in G and v_1 in S from Figure 1, where $NS_{u_1} = \{(A, 1.25), (B, 1), (C, 1)\}$ and $NS_{v_1} = \{(A, 1), (B, 0.5), (C, 0.5)\}$, is calculated as follows:

$$\frac{(1.25/1) + (1/0.5) + (1/0.5)}{3} = 1.75$$

We show more examples of satisfiability scores in Figure 2. Moreover, Figure 2 shows how to use the computed neighborhood signatures of v_1, u_1, u_2 and u_3 (based on Section 3.1) to determine the satisfiability scores of node pairs (u_1, v_1), (u_2, v_1) and (u_3,



Neighborhood Signature

	A	B	C	D
NS^2_{v1}	1	3	5/4	1/4

(a) PSI query with v_0 as pivot

Neighborhood Signatures

	A	B	C	D
NS^2_{u1}	2	4	5/4	1/4
NS^2_{u2}	2	4	1	0
NS^2_{u3}	1	4	9/4	1/2

$$SS(u_1,v_1) = \frac{(2/1) + (4/3) + ((5/4)/(5/4)) + (1/4)/(1/4)}{4} = 1.33$$

$$SS(u_2,v_1) = \frac{(2/1) + (4/3) + (1/(5/4)) + (0/(1/4))}{4} = 1.05$$

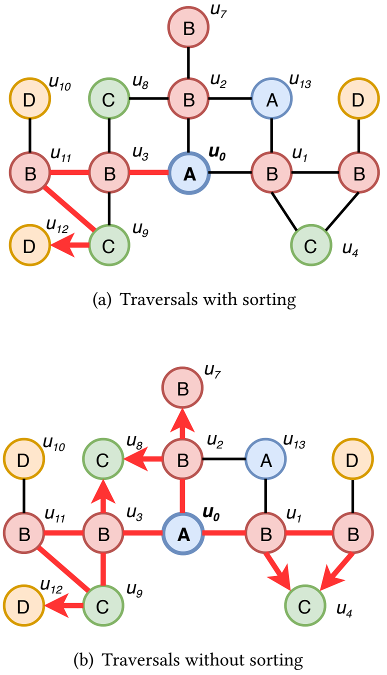
$$SS(u_3,v_1) = \frac{(1/1) + (4/3) + ((9/4)/(5/4)) + ((1/2)/(1/4))}{4} = 1.54$$

(b) Input graph G

Figure 2: (a) PSI query S, showing the neighbors signature for v_1 (b) Input graph G with satisfiability scores with regard to query node v_1

v_1). Note that a higher neighbourhood signature of label l for v_i means that v_i is connected with many nodes having label l which increases the chance for finding a match for u. As a result, the larger the satisfiability score of a graph node, the higher the chances that it satisfies a corresponding node in the PSI query. Once the satisfiability scores are calculated for all neighbors, the optimistic algorithm sorts them according to their scores and traverses those with higher scores first. The idea is to prioritize nodes with high likelihood to reach a result over other nodes with less chances, and thus the algorithm can finish faster. This step is repeated recursively by calculating the satisfiability scores of the new neighboring nodes and change paths until a match is found or all possible graph traversals are exhausted. The optimistic algorithm is useful only when used with valid nodes; i.e. nodes that match the pivot node. Otherwise, it suffers from performance degradation caused by the overhead of calculating the satisfiability scores for candidate nodes and sorting them, which is an unnecessary overhead for the case of invalid nodes.

Figure 3(a) shows an example of how the optimistic algorithm employs the satisfiability scores obtained in Figure 2(b) to reach a solution quickly. Consider evaluating the neighbors of u_0 for matching with query node v_1, the optimistic algorithm is going to traverse the graph starting from u_3, as shown in Figure 3(a), since it has the highest score ($SS(u_3) = 1.54$). As a result, the optimistic algorithm is able to find that u_0 is valid using the least number of traversals by avoiding the traversal of u_1 and u_2. A random traversal would have exhausted up to 3 different traversals from u_0, as shown in Figure 3(b), since u_1 and u_2 do not lead to an occurrence.



(a) Traversals with sorting

(b) Traversals without sorting

Figure 3: The behaviour of the optimistic algorithm starting from u_0 with (a) sorting and (b) without sorting

Algorithm 1: PSI Evaluation

```
   Input: G the input graph, S query subgraph, M Mapping, T
          Method type; Optimistic or Pessimistic
   Output: R true if a result is found, false otherwise
1  if M is full mapping then Return true
2  $v_{next}$ ←GetNextQueryNode(S)
3  C ←GetNextCandidateNodes(G, S, $v_{next}$)
4  if Super Optimistic then C ←GetLimitedCandidates(C,
   Max)
5  if T is Optimistic then C ←SortBySatisfiabilityScore(C)
6  foreach $c \in C$ do
7      if T is Pessimistic And ¬Satisfy(c, $v_{next}$) then Continue
8      $M_{new} \leftarrow M + (v_{next}, c)$
9      if ValidMapping(M) then
10         valid ←PSI(G, S, $M_{new}$, T)
11         if valid is true then Return true
12 Return false
```

Algorithm 1 shows the steps required for evaluating PSI queries. To enable the optimistic algorithm, the input flag T is set to optimistic. Note that this algorithm is recursive where it stops whenever a result is found (Line 1). The crux of the optimistic method is the sorting of graph nodes based on their satisfiability scores (Line 5). The sorting function introduces extra overhead, which might be unacceptable when evaluating nodes with high degrees. To avoid such problem and benefit from situations where a match can be quickly found, we propose a super optimistic step, where only a small portion of the neighbor nodes are checked. The difference between the "super" optimistic and the original optimistic method is in Line 4. For the super optimistic version, the number of candidates is limited by a maximum value (we use 10 in our experiments). Therefore, the overhead of sorting is significantly minimized and less intermediate graph nodes are visited. For each graph node, the optimistic approach proceeds with the super optimistic version. If a result is found, it returns that result without further processing. Otherwise, the normal optimistic method is used.

3.4 The Pessimist

In this method, we exploit aggressive pruning to quickly reach a decision. The pessimistic algorithm evaluates the label, degree and the neighborhood signature of the candidate nodes against the corresponding query node. Then, it prunes non-matching candidate nodes without carrying out further graph traversals. In comparison to a typical subgraph isomorphism evaluation, this early pruning reduces the number of intermediate results since it does not go deeper in the search plan for pruned nodes. This pruning results in a significant improvement, especially for graphs with high degree nodes. Although pruning has a notable computation price due to its excessive calculations per evaluated node, it is worth exploiting it for invalid nodes to reduce the cost of unproductive traversals. On the contrary, using this pruning for valid nodes introduces unnecessary overhead to reach the conclusion that a node is valid.

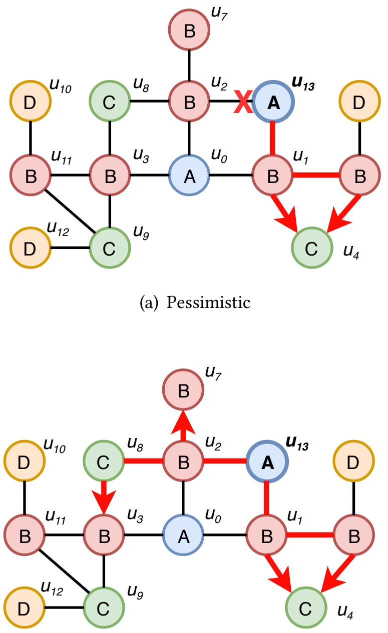
The utilization of the neighborhood signature in pruning is important since it captures wider properties of the area surrounding each graph node. Proposition 3.2 allows pruning of a node when its neighborhood signature cannot satisfy the neighborhood signature of the query node. Based on this proposition, a significant number of graph nodes are pruned but with a small number of false positive nodes (i.e., nodes that satisfy the corresponding neighborhood signature but are non-candidates because they do not match the query graph). Algorithm 1 follows the pessimistic approach by setting the flag T to Pessimistic. It is similar to the optimistic method except Line 7 which applies pruning on nodes using Proposition 3.2 and the neighborhood signature.

We show in Figure 4 the difference between the pessimistic and optimistic algorithms when evaluating an invalid node, i.e., u_{13}, against v_0 in Figure 2(a). In this example, the candidate mappings of query node v_1 are u_1 and u_2. Notice that both nodes satisfy label and direct neighbors requirements; both have label "B" and are connected to nodes labeled "B" and "C". When looking at the neighborhood signatures of both nodes, we discover that u_1 satisfies the neighborhood signature of v_1, i.e., its features have weights higher than or equal to those of the features of v_1. On the other hand, the neighborhood signature of u_2 does not satisfy that of v_1 because weights corresponding to labels "C" and "D" are both less than those for v_1. As a result, the pessimistic algorithm prunes u_2 early in the evaluation and traverse u_1 for further evaluation as shown in Figure 4(a). The optimistic algorithm, if employed, would explore both paths, u_1 and u_2, as shown in Figure 4(b).

4 THE REALIST: SMARTPSI

4.1 Two-threaded PSI Baseline

In order to benefit from the optimistic and pessimistic methods, we need to know the type of each graph node; i.e., valid or invalid, and pick the appropriate method accordingly. However, such knowledge is not available beforehand. Therefore, we propose a simple baseline, based on threading, that leverages the two algorithms for PSI evaluation. Figure 5 shows how this baseline works. For each graph node, two threads are executed concurrently. One thread runs the optimistic method while the other runs the pessimistic method. The thread that finishes its evaluation first stops the other thread and returns the result. This approach guarantees that each node is evaluated in almost the least wall-clock time of both methods (optimistic and pessimistic). However, it suffers from two issues: (*i*) under-utilization, as two threads are doing the job of a single task, and (*ii*) initiating and stopping millions of threads to evaluate the candidate graph nodes leads to a huge unnecessary cumulative overhead.



(a) Pessimistic

(b) Optimistic

Figure 4: The behaviour of (a) the pessimistic vs. (b) the optimistic algorithm starting from u_{13}

4.2 SmartPSI

This section discusses SmartPSI; our proposed PSI solution that employs machine learning techniques to avoid the limitations of the above-mentioned two-threaded baseline. We show in Figure 6 the architecture of SmartPSI which integrates the utilization of both PSI methods with two independent classification models. The first model (Model α) is a node type classifier that identifies whether a graph node is valid or invalid. Compared to the two-threaded approach, this model allows SmartPSI to benefit from the optimized PSI methods, i.e., the optimistic and the pessimistic, without sacrificing extra resources. The second classification model (Model β) is trained to predict an efficient query execution plan for each candidate node.

SmartPSI starts by loading the entire input graph in-memory, then it computes the neighborhood signatures for each graph node. When receiving a query, it extracts the candidate nodes from the input graph. A small percentage of the candidate nodes (around 10% up to a maximum value) is randomly selected to train

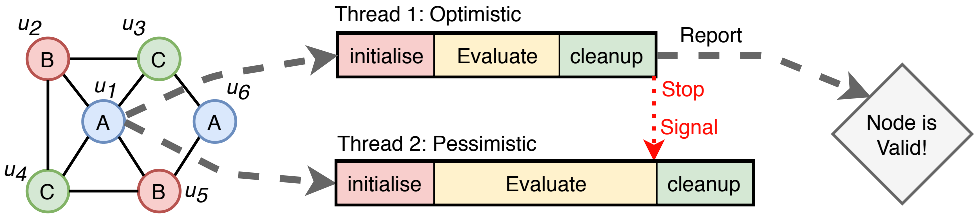


Figure 5: The two-threaded PSI solution

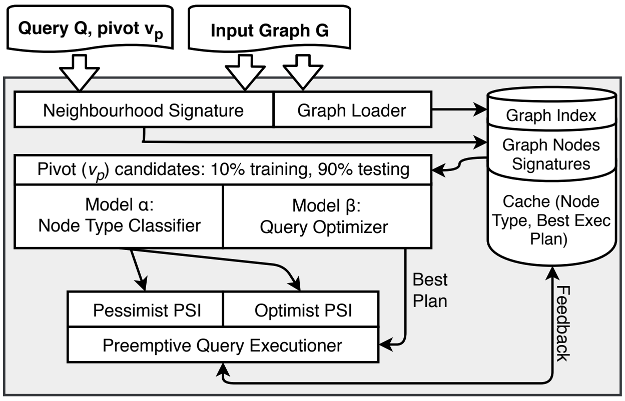


Figure 6: SmartPSI System Architecture

both models (α and β) using the Random Forest classification algorithm [10]. Our empirical evaluation shows that using this simple classifier is effective since it provides both a lightweight training time as well as a decent prediction accuracy (see Section 5.4). However, utilizing other machine learning classifiers; e.g. Neural Networks or Support Vector Machines, is orthogonal to our work. The remaining candidate nodes are passed to the trained models to predict their types and evaluation plans based on their neighborhood signatures. SmartPSI uses these predictions to evaluate each node against the query and the result of this evaluation is cached for future use. In the rest of this section, we describe in more details each one of these steps.

4.2.1 *Predicting the Node Type (Model α).*

Correctly predicting the candidate node type (as *valid* or *invalid*) helps SmartPSI to decide the node's most efficient graph matching method (optimistic or pessimistic). Valid nodes are best evaluated using the optimistic algorithm while the pessimistic algorithm is much better at evaluating invalid nodes. To select the best method for each graph node, we utilize a binary classification model to predict the node's type (class) based on its neighborhood signature, and accordingly decide which method to use for evaluation.

Training: The neighborhood signature of each graph node is used to build the feature vectors for our classifier. Each label in the neighborhood signature represents a feature and the value associated with this label in the signature is considered as the feature's weight. Each training node is evaluated using the pessimistic method and is labeled based on its confirmed type. The pessimistic method is used during model training since, on average, it is more stable and performs better than the optimistic method.

Prediction: For the remaining graph nodes, the trained model is used to predict the node type, either valid or invalid. Nodes predicted as valid are passed to the optimistic method while invalid nodes are given to the pessimistic method.

4.2.2 *Predicting an Optimal Plan (Model β).*

Selecting the order in which query nodes are evaluated is important for efficient evaluation. A bad order results in an excessive number of intermediate results and consequently poor performance. Existing techniques [17, 18, 30] employ heuristic approaches to prioritize the evaluation of query nodes that have higher selectivity. However, these approaches may not consistently provide good performance since their proposed plans are not adapted to the local features of each graph node. We describe next how SmartPSI trains a classifier to predict an efficient locality-aware query plan for each graph node based on its neighborhood signature.

Training: Model β is a *multi-class* classifier; its training phase uses the same set of graph nodes used for model α. For each query, a set of plans are generated and evaluated for each training node u_t. The plan that results in the least time is selected as the optimal plan for u_t. This plan is considered the class of u_t and is fed along with u_t feature vector to the multi-class classifier.

The training time can be very large especially for big queries with a considerably large number of possible plans. For example, a subgraph with 6 nodes can have up to 6! (or 720) plans. It is not practical and very expensive to evaluate all these plans. To mitigate this problem, we only train using a small sample of these plans to minimize the training time. Furthermore, we avoid evaluating very expensive plans by enforcing a configurable time limit when evaluating each one of the sample plans. We first set the time limit to a relatively small value. Evaluations of subsequent plans are not allowed to exceed that time limit. If no plan is able to finish within the allotted time, the time limit is gradually increased. This process repeats until at least one plan can finish within the time limit.

Prediction: For each remaining candidate node, SmartPSI uses the created model β to predict a good query execution plan based on its neighborhood signature. Then, this plan is fed to the corresponding PSI method for evaluation.

4.2.3 *Prediction Caching.*

As shown in Figure 6, correct predictions of both models are cached to improve the run-time performance of SmartPSI. The cache module stores the node signature of already evaluated nodes. Upon checking next candidate nodes, SmartPSI checks if a similar node has been evaluated before. If exists, the PSI evaluation model and execution plan decisions are looked up from the cache without consulting the classifiers. Using caching improves the efficiency of SmartPSI since the time needed to consult the prediction model and possible wrong predictions are avoided. This is because nodes having the same neighborhood signature are deemed similar since they have similar graph structures around them. Thus, they are expected to have the same optimal method and execution plan.

4.3 Preemptive Query Execution

As in any classifier, the trained model may result in incorrect predictions. In our case, there are two types of prediction errors; incorrect plan prediction and incorrect node type prediction. SmartPSI, however, employs a *detection and recovery* technique to minimize the impact of wrong predictions. As shown in Figure 6, the preemptive query processor monitors the evaluation of a node u_i and an incorrect prediction is detected if the time of this evaluation exceeds a maximum value. We calculate this value as follows:

$$MaxTime(u_i) = 2 \times AvgT(PSIMethod(u_i), Plan(u_i))$$

$PSIMethod(u_i)$ returns the used PSI method for u_i while $Plan(u_i)$ is the currently used plan. Note that $AvgT(X, Y)$ returns the average time needed for the selected PSI method X and plan Y during the training phase. To be able to recover from bad predictions, the query processor goes through three states; (*i*) it evaluates candidate nodes using predicted plan and PSI method with a time limit. If the execution timed-out, (*ii*) the processor uses the opposite PSI method and restarts the evaluation with a time limit. This step overcomes wrong predictions in the first model (Model α). If the execution timed-out again, (*iii*) the processor restarts the node evaluation, without time limits, by using the original predicted PSI method with a standard execution plan generated by selectivity-based heuristics. This step uses a best guess for the execution plan because Model β was not able to provide a credible suggestion. Since the accuracy of our classification models is high (as we show later in Section 5.4), the majority of candidate nodes should be able to finish before hitting the first timeout.

5 EVALUATION

In this section, we experimentally evaluate the performance of SmartPSI and compare it against existing competitors using several real large-scale datasets. Specifically, we show the following: (*i*) SmartPSI significantly outperforms the state-of-the-art subgraph isomorphism solutions for solving PSI queries (Section 5.2). (*ii*) Our proposed optimizations provide significant improvements including the optimized matrix-based signature generation and the proposed machine-learning approach compared to optimistic-only, pessimistic-only and the two-threaded baseline search techniques (Section 5.3). (*iii*) Our machine-learning model achieves both high accuracy and minimal run-time overhead (Section 5.4). Finally, (*iv*) we show a significant improvement in the efficiency of ScaleMine [4]; the state-of-the-art distributed frequent subgraph mining system; when subgraph isomorphism is replaced by pivoted subgraph isomorphism (Section 5.5).

5.1 Experimental setup

Datasets: We use six real graphs to evaluate the performance of SmartPSI and compare it to existing techniques. These graphs are widely used in the subgraph isomorphism and frequent subgraph mining literature [4, 9, 17]. Table 3 shows the details of each dataset. *Yeast* [8] and *Human* [26] are protein-protein interaction networks where nodes represent proteins and edges represent the interactions among them. *Cora* [1] is a citation graph where each node represents a publication with a label representing a machine learning area. In the *YouTube* [3] graph, each node represents a YouTube video and is labeled with its category, while edges connect similar videos. *Twitter* [2] models the Twitter social network where each node represents a user and edges represent follower-followee relationships. The original Twitter graph is unlabeled, we follow the same approach used in [4] to assign node labels. Finally, *Weibo* [40] is a social network crawled from Sina Weibo micro-blogging website. Each node represents a user and is labeled with the city that user lives in, while each edge represents a follower-followee relationship.

Table 3: Datasets and their characteristics

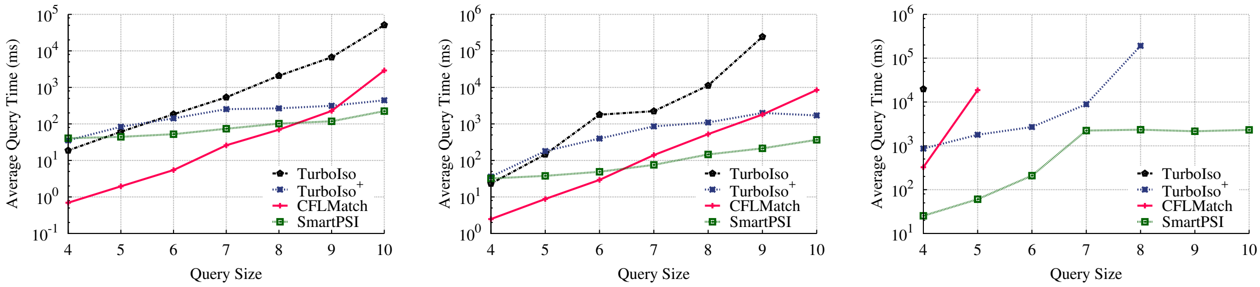
Dataset	Nodes	Edges	#node labels
Yeast [8]	3,112	12,519	71
Cora [1]	2,708	5,429	7
Human [26]	4,674	86,282	44
Youtube [3]	5,101,938	42,546,295	25
Twitter [2]	11,316,811	85,331,846	25
Weibo [40]	1,655,678	369,438,063	55

Query Graphs: From each input graph, we extract a set of random connected subgraphs as query graphs. Then, a random node is assigned as a pivot node for each query. Similar to the related work in [9, 17], a random walk with restart algorithm is used to extract 1000 query graphs for each size. Query sizes range from four to ten nodes. The resulted queries span a wide range of query complexities including paths, trees, stars and other complex shapes. Thus, they cover query loads for a wide variety of PSI applications. Unless stated otherwise, in all experiment, we use 1000 queries per query size.

Hardware Setup: SmartPSI and its competitors are deployed on a single Linux machine with 148GB RAM and two 2.1GHz AMD Opteron 6172 CPUs, each with 12 cores. We also use a distributed hardware setting for deploying ScaleMine [4] and its PSI version. In particular, we conduct ScaleMine experiments (see Section 5.5) on a Cray XC40 supercomputer which has 6,174 dual sockets compute nodes based on 16 cores Intel processors running at 2.3GHz with 128GB of RAM. We used up to 32 compute nodes with a total of 1024 cores. In all experiments, the maximum allowed time is 24 hours, any task which exceeds this limit is aborted.

5.2 Comparison with Existing Systems

SmartPSI is evaluated against three state-of-the-art subgraph isomorphism solutions: (*i*) CFL-Match [9]; the fastest reported subgraph isomorphism solution, (*ii*) TurboIso [17]; a recent subgraph ismorphism system that utilizes the query structure to combine the evaluation of similar parts of the query in a single task. and (*iii*) TurboIso$^+$; a modified version of TurboIso. Since TurboIso is a subgraph isomorphism algorithm, it finds all matches of a given query regardless of the pivot node. To optimize the performance for PSI queries, we configured TurboIso$^+$ to start evaluating the given queries using candidate matches of the pivot nodes and stop the evaluation once a pivot node match is found. Since the source code of CFL-Match is not publicly available, we could not follow a similar approach to provide a modified version of CFL-Match that is optimized for PSI queries. We also tried to



(a) Yeast Dataset (b) Cora Dataset (c) Human Dataset

Figure 7: Query Performance of SmartPSI vs. state-of-the-art Subgraph Isomorphism Systems

compare against TurboIso-Boost [30]; however, similar to what mentioned in [9], we encountered several run-time problems and we could not resolve them even after communicating with the authors. Thus, we had to omit its results. Notice that SmartPSI and all its competitors are single-machine in-memory systems. Moreover, for each input graph, we use a maximum of 1000 nodes to train the classification models in SmartPSI.

This experiment shows how available solutions (i.e., subgraph isomorphism techniques) perform when solving pivoted subgraph isomorphism queries in comparison to SmartPSI. In order to evaluate PSI queries, these algorithms use subgraph isomorphism to find all matches, then generates the list of distinct graph nodes that correspond to the pivot query node. Figure 7 shows the results on Yeast, Cora and Human datasets. The x-axis shows the query size (in number of nodes) and the y-axis shows the processing time. For Yeast (Figure 7(a)), subgraph isomorphism is a comparably easy task since this dataset is relatively small. As a result, existing techniques outperform SmartPSI on queries of size four. As the query size increases, subgraph isomorphism techniques become slower than SmartPSI. The optimizations proposed by CFLMatch allows it to outperform other systems up to queries of size 8, however, SmartPSI starts gaining momentum and becomes the fastest for larger queries. Notice that TurboIso$^+$ is significantly faster than TurboIso due to its optimization step, however, it is still slower than SmartPSI on larger queries.

For Cora dataset (Figure 7(b)), SmartPSI significantly outperforms other systems with up to two orders of magnitude. CFLMatch's optimizations worked well for small queries; it is the fastest for queries of size 4, 5 and 6. As the query size grows, similar to the Yeast dataset, CFLMatch generates huge amounts of intermediate query matches and becomes worse than SmartPSI. For queries with size 10, TurboIso fails to finish query evaluation within 24 hours. Finally for Human dataset (Figure 7(c)), both TurboIso and CFLMatch fail to evaluate most of the queries as this dataset is significantly larger and denser. TurboIso$^+$ could only complete its evaluation for queries of size up to 8. Compared to all systems, SmartPSI is able to evaluate all queries on the Human dataset with performance improvements of up to two orders of magnitude. This experiment highlights the unsuitability of the state-of-the-art subgraph isomorphism solutions to solve PSI queries even for small graphs. It also shows SmartPSI's significant improvements over the existing solutions.

5.3 SmartPSI Optimizations

Neighborhood Signature Overhead: Figure 8 compares between the matrix-based and exploration-based methods in terms of the total time for populating the neighborhood signatures (see Section 3.1). As shown in Figure 8, the processing time of both methods increases as the input graph gets larger. Exploration-based method, however, suffers from being computationally expensive especially for large datasets. For Twitter, exploration-based could not finish generating the nodes' neighborhood signatures within 24 hours. On the other hand, the matrix-based method reduces the overhead of exploration-based method significantly. It requires around 400 seconds, which is more than two orders of magnitude faster than the exploration-based method. This significant improvement is a direct result of the difference in complexities between the two methods.

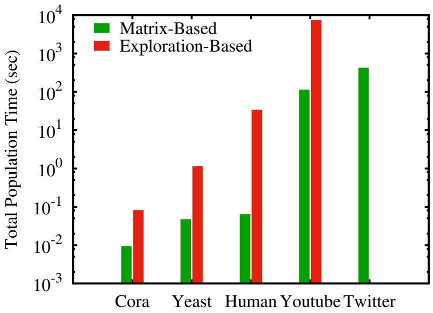
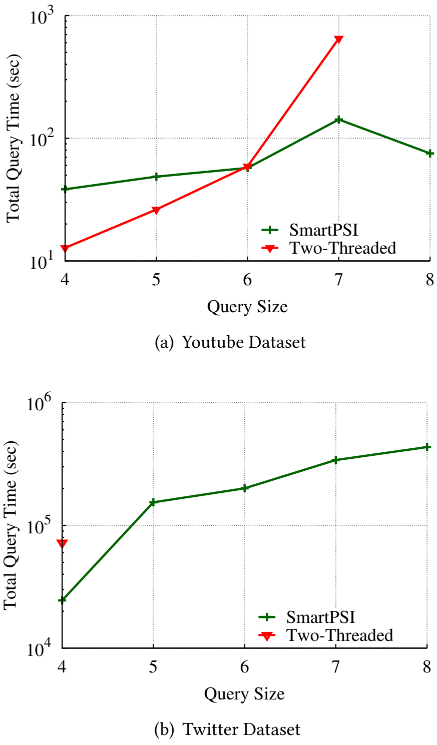


Figure 8: Exploration vs. Matrix Based Approach for Populating Neighborhood Signatures of various datasets.

SmartPSI vs. the two-threaded baseline: Figure 9 shows the run-time overhead of evaluating 100 queries on SmartPSI compared to the two-threaded solution using YouTube and Twitter datasets. The X-axis shows the query size in terms of number of nodes, while the Y-axis shows the total time to evaluate the corresponding queries. Only 100 queries are used since evaluating 1000 queries takes too much time for the two threaded approach. Since the two-threaded baseline utilizes two parallel threads, for a fair comparison, only in this experiment we run a modified version of SmartPSI that uses two concurrent threads to evaluate two different graph nodes in parallel. Using YouTube datatset, Figure 9(a) shows that the two-threaded baseline outperforms SmartPSI for small queries because it does not possess the overhead of training and prediction. Nevertheless, the two-threaded solution becomes slower than SmartPSI as queries grow in size till it exceeds the time limit on queries larger than 7 nodes. This is also true for Twitter dataset in Figure 9(b), where the two-threaded baseline can only evaluate queries of size 4. SmartPSI



(a) Youtube Dataset

(b) Twitter Dataset

Figure 9: SmartPSI (2 threads) vs. two-threaded Baseline

has an upper hand since the two-threaded baseline underutilizes the available resources by using two threads to finish a task that can be conducted by one thread. Moreover, the heuristic-based query execution plans, which are used in the two-threaded solution, are far less competent than the optimized plans of SmartPSI.

SmartPSI vs. optimistic and pessimistic: Figure 10 compares SmartPSI against *Pessimistic*-only and *Optimistic*-only using 10 queries for each query size on the Twitter dataset. In particular, the Pessimistic solution uses the pessimistic PSI method regardless of the type of the graph node. Likewise, Optimistic solution always use the optimistic PSI method. Moreover, the *Pessimistic* and *Optimistic* solutions use a heuristic-based query evaluation plan. SmartPSI significantly outperforms the optimistic and pessimistic methods. Furthermore, the two competitors fail to evaluate queries of size eight. The node type prediction in SmartPSI allows it to avoid exploring the full search space (compared to *Pessimistic*) and to reduce the overhead of calculating scores and sorting graph nodes accordingly (compared to *Optimistic*). Moreover, SmartPSI is able to predict the best evaluation plan for the candidate nodes, which is not achievable for the other two solutions.

5.4 Prediction Accuracy and Training Overhead

Machine Learning Models: We tested several machine learning models to build SmartPSI's classifier including Random Forest (RF), Support Vector Machines (SVM) and Neural Networks (NN). In our experiments, we found that the RF classifier provides the best accuracy. For example, it always achieves more than 95% accuracy on Human dataset compared to SVM (90%) and NN (92%). At the same time, RF is two times faster in building the model and getting predictions. We also observed similar performance results for the other datasets. Therefore, we use RF as it provides consistently good performance in terms of accuracy and training time.

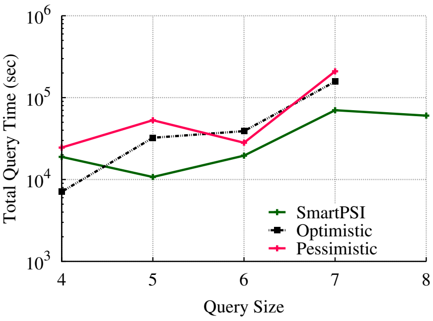


Figure 10: SmartPSI vs. Optimistic and Pessimistic on Twitter Dataset

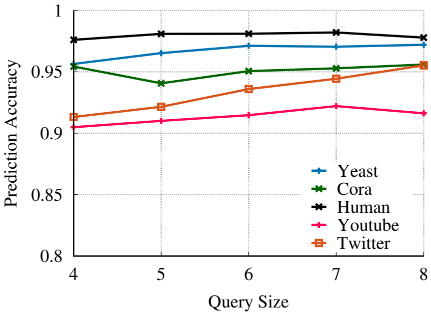


Figure 11: Prediction Accuracy

Model accuracy: We conducted a set of experiments to measure the prediction accuracy of the node type prediction model using Yeast, Human, Cora, YouTube and Twitter datasets. Accuracy is calculated by comparing the result of the model's prediction to the ground truth result obtained by node evaluation using subgraph isomorphism, i.e., the number of correctly predicted node types divided by the total number of examined nodes. Figure 11 shows that the prediction accuracy of the proposed classification model is always higher than 90% for the used datasets. Furthermore, it also shows that prediction accuracy is effective and stable where the variation in predictions quality among the different datasets is small regardless of the query size.

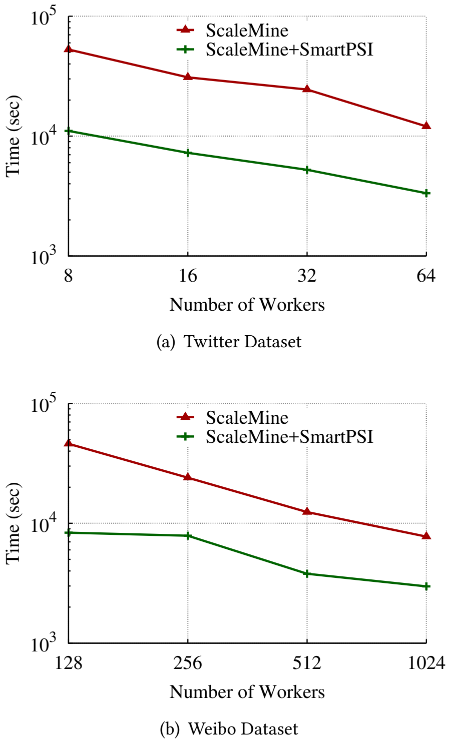
Training overhead: Table 4 shows the overhead of models training/prediction compared to the total query evaluation time. This overhead includes model building, node type prediction and suggesting a good execution plan. In this experiment, we ran 100 queries per query size for each dataset. For small datasets and queries, the training overhead is relatively large compared to the total time. For example in Human dataset, the prediction overhead exceeds the PSI evaluation time for small queries. The reason is that query evaluation on small graphs is usually comparably fast. For larger query sizes in Human dataset, PSI evaluation becomes more expensive which reduces the relative overhead of the classification model. For larger datasets; e.g., YouTube and Twitter, the training time is very insignificant compared to PSI evaluation time. This shows that high accuracy of our classification model comes at a low computation cost, especially on large graphs.

Table 4: The overhead of model training and prediction to the total time of SmartPSI

Dataset	4	5	6	7	8
Human	75.41%	57.94%	50.85%	34.88%	33.05%
YouTube	1.13%	1.19%	1.20%	1.79%	2.83%
Twitter	1.98%	5.33%	20.15%	2.35%	5.19%

5.5 PSI Application: Frequent Subgraph Mining

This experiment highlights the advantage of using SmartPSI to boost the efficiency of an example PSI-dependent application; Frequent Subgraph Mining (FSM). For this experiment, we use ScaleMine[1] [4]; the state-of-the-art distributed FSM system. Note that ScaleMine uses subgraph isomorphism for finding occurrences for each candidate subgraph. We implemented ScaleMine+SmartPSI, a variation of ScaleMine that employs SmartPSI's techniques instead of the traditional subgraph isomorphism for computing the frequency of each candidate subgraph. Figures 12(a) and 12(b) show the performance of ScaleMine compared to the optimized version, ScaleMine+SmartPSI, using Twitter and Weibo datasets on frequency thresholds of 155K and 460K, respectively. The X-axis shows the total number of compute nodes while the Y-axis shows the overall time required to finish the mining task. For the Weibo dataset, the maximum size of allowed frequent subgraphs is set to six edges. In this experiment, ScaleMine+SmartPSI outperforms ScaleMine significantly for both datasets. For Twitter, ScaleMine+SmartPSI is up to 5X faster than ScaleMine. As for Weibo, ScaleMine+SmartPSI is up to 6X faster. By replacing the traditional subgraph isomorphism, ScaleMine+SmartPSI generates significantly less intermediate results and quickly reaches the final answer.



(a) Twitter Dataset

(b) Weibo Dataset

Figure 12: ScaleMine vs. ScaleMine+SmartPSI

6 RELATED WORK

6.1 Subgraph Isomorphism

Subgraph isomorphism is the bottleneck of many graph operations since it is an NP-complete problem. Therefore, many research efforts attempted to reduce its overhead in practice. The first practical algorithm that addresses this problem follows a backtracking approach [35]. Since then, several performance enhancements were proposed, ranging from CSP-based techniques [29], search order optimization [18], indexing [38] and parallelization [33]. As reported in [26], GraphQL [18] is considered one of the best subgraph isomorphism techniques, though its performance is not stable over the different datasets. GraphQL prunes the search space by using local and global graph information. Moreover, it utilizes a search order optimization technique based on a global cost model.

TurboIso [17] is a recent approach that outperforms previous techniques by grouping similar parts of the query and process them at once. Moreover, TurboIso adapts its search order plan according to each input graph node. BoostIso [30] is a plugin that enhances the efficiency of other approaches. It is able to avoid duplicate computations by exploiting the relationship among the input graph nodes. CFLMatch [9] is the current state-of-the-art technique for subgraph isomorphism. It decomposes the query graph into a core and multiple trees. Such decomposition allows better search order optimization. Moreover, CFLMatch avoids redundant evaluations of tree leaf nodes by representing the mappings of these nodes in a compressed way. Although these techniques lead to relatively significant improvements in the domain of subgraph isomorphism, the algorithm itself is impractical especially for large graphs. The main reason behind this limitation of subgraph isomorphism is the excessive number of results it generates.

The closest approach to our work is Ψ-framework [22]. It utilizes both isomorphic query re-writings and existing alternative algorithms in parallel to improve the performance of subgraph isomorphism. Ψ-framework is similar to our two-threaded prototype (Section 4.1) where it requires more processing power to achieve a reasonable performance. Unlike SmartPSI, Ψ-framework is unable to decide on which approach is better to use during runtime.

6.2 Neighborhood Signature

A graph node can be represented by a signature that reflects the neighborhood structures around it. This neighborhood signature can be represented as a feature vector and used in many important applications such as abnormal nodes detection [6] and predicting missing links [27].

Neighborhood signatures are also used for indexing and pruning. GADDI [41] is a subgraph matching solution that builds an index based on graph structures that exist between any pair of vertices. Based on this index,a two way pruning technique is used to efficiently prune candidate matches. GADDI suffers from the excessive cost required for creating its index. NOVA [43] and DSI [24] maintain the paths to and from surrounding nodes. These paths are used to filter out unqualified graph nodes. Both Nova and DSI rely on maintaining a huge number of paths. As such, both technique comprise excessive overhead, especially

[1] https://github.com/ehab-abdelhamid/ScaleMine

when processing large-scale dense graphs. (NNT) [36] is a more effective index that relies on finding the trees around each graph node. NNT requires the use of tree matching algorithms which makes pruning and matching more complex.

The above approaches are expensive, especially for large-scale dense graphs. Other simpler indexes were proposed such as SMS [42] which maintains the list of labels and node degrees that appear in the neighborhood. Although this indexing technique is simple, it significantly outperforms NOVA and GADDI. Tale [34] is a system that finds approximate matches of query subgraphs in an input graph. It relies on an index, called NH-Index, which maintains information about the direct neighbors of each graph node. NH-Index maintains information such as node label, node degree, neighbor nodes and how they are connected. Both SMS and Tale focus on the direct neighbors. Thus, their pruning and descriptive power is limited. More recently, k-hop label [7] extends its indexing by considering all neighbor nodes within K-hops. Such representation extends beyond direct neighbors, but it lacks effective representation of the surrounding structure. Finally, label propagation [23] captures the neighborhood structure around each graph node using a list of labels along with their corresponding weights. Weights represent how labels are placed and connected around the node under consideration. Thus, more informative structural information is captured.

7 CONCLUSION

In this paper, we propose SmartPSI; an efficient machine-learning based system for evaluating Pivoted Subgraph Isomorphism (PSI) queries. It is based on two effective PSI algorithms; each is optimized for a certain graph node type. SmartPSI is also coupled with a machine-learning model to predict the graph node type and call the appropriate PSI algorithm accordingly. It also uses a machine-learning based query optimizer to select execution plans that reduce the total query run-time by minimizing intermediate query results. Our experiments show that SmartPSI is efficient and able to scale PSI evaluation where it is up to two orders of magnitude faster compared to existing subgraph isomorphism techniques. Furthermore, when applied to Frequent Subgraph Mining (FSM), PSI improves the performance of the state-of-the-art distributed FSM system by up to a factor of 6X.

ACKNOWLEDGMENTS

This research made use of the Supercomputing Laboratory resources at King Abdullah University of Science & Technology (KAUST), Saudi Arabia.

REFERENCES

[1] 2017. Cora Dataset. http://linqs.umiacs.umd.edu/projects/projects/lbc/. (2017).

[2] 2017. Twitter Dataset. http://socialcomputing.asu.edu/datasets/Twitter. (2017).

[3] 2017. YouTube Dataset. http://netsg.cs.sfu.ca/youtubedata/. (2017).

[4] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. 2016. Scalemine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph. In *Proceedings of SC.*

[5] E. Abdelhamid, M. Canim, M. Sadoghi, B. Bhattacharjee, Y. Chang, and P. Kalnis. 2017. Incremental Frequent Subgraph Mining on Large Evolving Graphs. *IEEE Transactions on Knowledge and Data Engineering* 29, 12 (December 2017), 2710–2723.

[6] Leman Akoglu, Mary McGlohon, and Christos Faloutsos. 2010. Oddball: Spotting anomalies in weighted graphs. In *Proceedings of PAKDD.*

[7] Pranay Anchuri, Mohammed J Zaki, Omer Barkol, Shahar Golan, and Moshe Shamy. 2013. Approximate Graph Mining with Label Costs. In *Proceedings of SIGKDD.*

[8] Saurabh Asthana, Oliver D King, Francis D Gibbons, and Frederick P Roth. 2004. Predicting Protein Complex Membership using Probabilistic Network Reliability. *Genome Research* 14, 6 (2004).

[9] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of SIGMOD.*

[10] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32.

[11] Björn Bringmann and Siegfried Nijssen. 2008. What is frequent in a single graph?. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining.* Springer, 858–863.

[12] Young-Rae Cho and Aidong Zhang. 2010. Predicting Protein Function by Frequent Functional Association Pattern Mining in Protein Interaction Networks. *IEEE Transactions on Information Technology in Biomedicine* 14, 1 (2010).

[13] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. GraMi: Frequent subgraph and pattern mining in a single large graph. *PVLDB* 7 (2014).

[14] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co.

[15] Jialong Han and Ji-Rong Wen. 2013. Mining frequent neighborhood patterns in a large labeled graph. In *Proceedings of CIKM.*

[16] Jialong Han, Kai Zheng, Aixin Sun, Shuo Shang, and Ji-Rong Wen. 2016. Discovering Neighborhood Pattern Queries by sample answers in knowledge base. In *Proceedings of ICDE.*

[17] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *Proceedings of SIGMOD.*

[18] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: Auery Language and Access Methods for Graph Databases. In *Proceedings of SIGMOD.*

[19] Keith Henderson, Brian Gallagher, Tina Eliassi-Rad, Hanghang Tong, Sugato Basu, Leman Akoglu, Danai Koutra, Christos Faloutsos, and Lei Li. 2012. Rolx: Structural Role Extraction & Mining in Large Graphs. In *Proceedings of SIGKDD.* ACM.

[20] CA James, D Weininger, and J Delany. 2003. Daylight Theory Manual Daylight Version 4.82. Daylight Chemical Information Systems. (2003).

[21] Glen Jeh and Jennifer Widom. 2002. SimRank: A Measure of Structural-context Similarity. In *Proceedings of SIGKDD.* ACM.

[22] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. 2017. Subgraph querying with parallel use of query rewritings and alternative algorithms. (2017).

[23] Arijit Khan, Xifeng Yan, and Kun-Lung Wu. 2010. Towards Proximity Pattern Mining in Large Graphs. In *Proceedings of SIGMOD.*

[24] Yubo Kou, Yukun Li, and Xiaofeng Meng. 2010. DSI: A Method for Indexing Large Graphs using Distance Set. In *Proceedings of WAIM.*

[25] Michael Lappe and Liisa Holm. 2004. Unraveling Protein Interaction Networks with Near-optimal Efficiency. *Nature Biotechnology* 22, 1 (2004).

[26] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *PVLDB* 6, 2 (2012).

[27] Ryan N Lichtenwalter, Jake T Lussier, and Nitesh V Chawla. 2010. New Perspectives and Methods in Link Prediction. In *Proceedings of SIGKDD.* ACM.

[28] Andrew McCallum, Xuerui Wang, and Andrés Corrada-Emmanuel. 2007. Topic and Role Discovery in Social Networks with Experiments on Enron and Academic Email. *Journal of Artificial Intelligence Research* 30 (2007).

[29] J J McGregor. 1979. Relational Consistency Algorithms and their Application in Finding Subgraph and Graph Isomorphisms. *Information Sciences* 19 (1979).

[30] Xuguang Ren and Junhu Wang. 2015. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *PVLDB* 8 (2015).

[31] Sherif Sakr, Faisal Moeen Orakzai, Ibrahim Abdelaziz, and Zuhair Khayyat. 2016. *Large-Scale graph processing using Apache giraph.* Springer.

[32] Tom AB Snijders, Philippa E Pattison, Garry L Robins, and Mark S Handcock. 2006. New Specifications for Exponential Random Graph Models. *Sociological Methodology* 36, 1 (2006).

[33] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *PVLDB* 5, 9 (May 2012).

[34] Yuanyuan Tian and Jignesh M Patel. 2008. Tale: A Tool for Approximate Large Graph Matching. In *Proceedings of ICDE.*

[35] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *Journal of ACM* 23 (1976). Issue 1.

[36] Changliang Wang and Lei Chen. 2009. Continuous Subgraph Pattern Search over Graph Streams. In *Proceedings of ICDE.*

[37] Xifeng Yan and Jiawei Han. 2002. gspan: Graph-based Substructure Pattern Mining. In *Proceedings of ICDM.*

[38] Xifeng Yan, Philip S. Yu, and Jiawei Han. 2004. Graph indexing: A Frequent Structure-based Approach. In *Proceedings of SIGMOD.*

[39] Yu Yang, Jian Pei, and Abdullah Al-Barakati. 2017. Measuring In-network Node Similarity Based on Neighborhoods: A Unified Parametric Approach. *Knowledge and Information Systems* (2017).

[40] Jing Zhang, Biao Liu, Jie Tang, Ting Chen, and Juanzi Li. 2013. Social Influence Locality for Modeling Retweeting Behaviors. In *Proceedings of IJCAI.*

[41] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: Distance Index Based Subgraph Matching in Biological Networks. In *Proceedings of EDBT.*

[42] Weiguo Zheng, Lei Zou, and Dongyan Zhao. 2011. Answering Subgraph Queries over Large Graphs. In *Proceedings of WAIM.* Springer.

[43] Ke Zhu, Ying Zhang, Xuemin Lin, Gaoping Zhu, and Wei Wang. 2010. Nova: A Novel and Efficient Framework for Finding Subgraph Isomorphism Mappings in Large Graphs. In *Proceedings of DASFAA.*

MinoanER: Schema-Agnostic, Non-Iterative, Massively Parallel Resolution of Web Entities

Vasilis Efthymiou*
IBM Research
vasilis.efthymiou@ibm.com

George Papadakis
University of Athens
gpapadis@di.uoa.gr

Kostas Stefanidis
Univeristy of Tampere
konstantinos.stefanidis@tuni.fi

Vassilis Christophides
INRIA-Paris and Univ. of Crete
vassilis.christophides@inria.fr

ABSTRACT

Entity Resolution (ER) aims to identify different descriptions in various Knowledge Bases (KBs) that refer to the same entity. ER is challenged by the *Variety*, *Volume* and *Veracity* of entity descriptions published in the Web of Data. To address them, we propose the MinoanER framework that simultaneously fulfills *full automation*, support of *highly heterogeneous* entities, and *massive parallelization* of the ER process. MinoanER leverages a token-based similarity of entities to define a new metric that derives the similarity of neighboring entities from the most important relations, as they are indicated only by statistics. A *composite blocking method* is employed to capture different sources of matching evidence from the content, neighbors, or names of entities. The search space of candidate pairs for comparison is compactly abstracted by a novel *disjunctive blocking graph* and processed by a non-iterative, massively parallel matching algorithm that consists of four generic, schema-agnostic matching rules that are quite robust with respect to their internal configuration. We demonstrate that the effectiveness of MinoanER is comparable to existing ER tools over real KBs exhibiting *low Variety*, but it outperforms them significantly when matching KBs with *high Variety*.

1 INTRODUCTION

Even when data integrated from multiple sources refer to the same real-world entities (e.g., persons, places), they usually exhibit several quality issues such as *incompleteness* (i.e., partial data), *redundancy* (i.e., overlapping data), *inconsistency* (i.e., conflicting data) or simply *incorrectness* (i.e., data errors). A typical task for improving various data quality aspects is *Entity Resolution* (ER). In the Web of Data, ER aims to facilitate interlinking of data that describe the same real-world entity, when unique entity identifiers are not shared across different Knowledge Bases (KBs) describing them [8]. To resolve entity descriptions we need (a) *to compute effectively the similarity of entities*, and (b) *to pair-wise compare entity descriptions*. Both problems are challenged by the three Vs of the Web of Data, namely Variety, Volume and Veracity [10]. Not only does the *number of entity descriptions* published by each KB never cease to increase, but also the *number of KBs* even for a single domain, has grown to thousands (e.g., there is a x100 growth of the LOD cloud size since its first edition[1]). Even in the same domain, KBs are *extremely heterogeneous* both regarding how they semantically structure their data, as well as how diverse properties are used to describe even substantially similar entities (e.g., only 109 out of ~2,600 LOD vocabularies are shared by more than one KB). Finally, KBs are of *widely differing quality*, with significant differences in the coverage, accuracy and timeliness of data provided [9]. Even in the same domain, various inconsistencies and errors in entity descriptions may arise, due to the limitations of the automatic extraction tools [34], or of the crowd-sourced contributions.

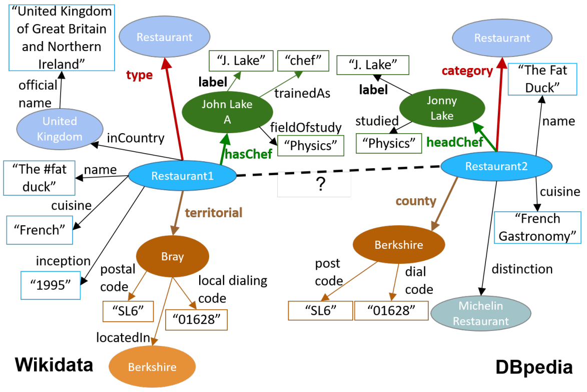


Figure 1: Parts of entity graphs, representing the Wikidata (left) and DBpedia (right) KBs.

The Web of Data essentially calls for novel ER solutions that relax a number of assumptions underlying state-of-the-art methods. The most important one is related to the notion of similarity that better characterizes *entity descriptions* in the Web of Data - we define an entity description to be a URI-identifiable set of attribute-value pairs, where values can be literals, or the URIs of other descriptions, this way forming an *entity graph*. Clearly, Variety results in extreme *schema heterogeneity*, with an unprecedented number of attribute names that cannot be unified under a global schema [15]. This situation renders all schema-based similarity measures that compare specific attribute values inapplicable [15]. We thus argue that similarity evidence of entities within and across KBs can be obtained by looking at the bag of strings contained in descriptions, regardless of the corresponding attributes. As this **value-based similarity** of entity pairs may still be weak, due to a highly heterogeneous description content, we need to consider additional sources of matching evidence; for instance, the **similarity of neighboring entities**, which are interlinked via various semantic relations.

Figure 1 presents parts of the Wikidata and DBpedia KBs, showing the entity graph that captures connections inside them. For example, Restaurant2 and Jonny Lake are neighbor entities in this graph, connected via a "headChef" relation. If we compare John Lake A to Jonny Lake based on their values only, it is easy

*Work conducted during the Ph.D research of the author at ICS-FORTH.
[1] https://lod-cloud.net

Series ISSN: 2367-2005

10.5441/002/edbt.2019.33

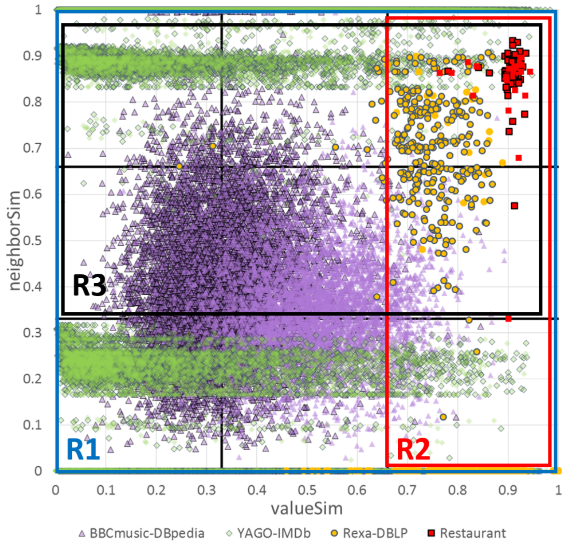


Figure 2: Value and neighbor similarity distribution of matches in 4 datasets (see Table 1 for more details).

to infer that those descriptions are matching; they are *strongly similar*. However, we cannot be that sure about Restaurant1 and Restaurant2, if we only look at their values. Those descriptions are *nearly similar* and we have to look further at the similarity of their neighbors (e.g, John Lake A and Jonny Lake) to verify that they match.

Figure 2 depicts both sources of similarity evidence (*valueSim*, *neighborSim*) for entities known to match (i.e., ground truth) in four benchmark datasets that are frequently used in the literature (details in Table 1). Every dot corresponds to a matching pair of entities, and its shape denotes its origin KBs. The horizontal axis reports the normalized value similarity (weighted Jaccard coefficient [21]) based on the tokens (i.e., single words in attribute values) shared by a pair of descriptions, while the vertical one reports the maximum value similarity of their neighbors. The value similarity of matching entities significantly varies across different KBs. For **strongly similar entities**, e.g., with a value similarity > 0.5, existing duplicate detection techniques work well. However, a large part of the matching pairs of entities is covered by **nearly similar entities**, e.g., with a value similarity < 0.5. To resolve them, we need to additionally exploit evidence regarding the similarity of neighboring entities.

This also requires revisiting the blocking (aka indexing) techniques used to reduce the number of candidate pairs [7]. To avoid restricting candidate matches (i.e., descriptions placed on the same block) to strongly similar entities, we need to assess both *value* and *neighbor* similarity of candidate matches. In essence, rather than a unique indexing function, we need to consider a *composite blocking* that provides matching evidence from different sources, such as the content, the neighbors or even the names (e.g., `rdfs:label`) of entities. Creating massively parallelizable techniques for processing the search space of candidate pairs formed by such composite blocking is an open research challenge.

Overall, the main requirements for Web-scale ER are: *(i)* identify both strongly and nearly similar matches, *(ii)* do not rely on a given schema, *(iii)* do not rely on domain experts for aligning relations and matching rules, *(iv)* develop non-iterative solutions to avoid late convergence, and *(v)* scale to massive volumes of data. None of the existing ER frameworks proposed for the Web of Data (e.g., LINDA [4], SiGMa [21] and RiMOM [31]) simultaneously fulfills all these requirements. In this work, we present the MinoanER framework for a Web-scale ER[2]. More precisely, we make the following contributions:

• We leverage a token-based similarity of entity descriptions, introduced in [27], to define a new metric for the similarity of a set of neighboring entity pairs that are linked via important relations to the entities of a candidate pair. Rather than requiring an *a priori* knowledge of the entity types or of their correspondences, we rely on *simple statistics over two KBs to recognize the most important entity relations* involved in their neighborhood, as well as, *the most distinctive attributes that could serve as names of entities* beyond the `rdfs:label`, which is not always available in descriptions.

• We exploit several indexing functions to place entity descriptions in the same block either because they share a common token in their values, or they share a common name. Then, we introduce a novel abstraction of multiple sources of matching evidence regarding a pair of entities (from the content, neighbors, or the names of their descriptions) under the form of a *disjunctive blocking graph*. We present an efficient algorithm for weighting and then pruning the edges with low weights, which are unlikely to correspond to matches. As opposed to existing disjunctive blocking schemes [3, 18], *our disjunctive blocking is schema-agnostic and requires no (semi-)supervised learning*.

• We propose a *non-iterative matching process* that is implemented in Spark [36]. Unlike the data-driven convergence of existing systems (e.g., LINDA [4], SiGMa [21], RiMOM [31]), the matching process of MinoanER involves a specific number of predefined generic, schema-agnostic matching rules (R1-R4) that traverse the blocking graph. First, we identify matches based on their name (R1). This is a very effective method that can be applied to all descriptions, regardless of their values or neighbor similarity. Unlike the schema-based blocking keys of relational descriptions usually provided by domain experts, *MinoanER automatically specifies distinctive names of entities from data statistics*. Then, the value similarity is exploited to find matches with many common and infrequent tokens, i.e., strongly similar matches (R2). When value similarity is not high, nearly similar matches are identified based on both value and neighbors' similarity using a *threshold-free rank aggregation function* (R3), as opposed to existing works that combine different matching evidence into an aggregated score. Finally, *reciprocal evidence* of matching is exploited as a verification of the returned results: only entities that are mutually ranked in the top positions of their unified ranking lists are considered matches (R4). Figure 2 abstractly illustrates the type of matching pairs that are covered by each matching rule.

• We experimentally compare the effectiveness of MinoanER against state-of-the-art methods using established benchmark data that involve real KBs. The main conclusion drawn from our experiments is that MinoanER achieves at least equivalent performance over KBs exhibiting a *low Variety* (e.g., those originating from a common data source like Wikipedia) even though the latter make more assumptions about the input KBs (e.g., alignment of relations); yet, MinoanER significantly outperforms state-of-the-art ER tools when matching KBs with high Variety. The source code and datasets used in our experimental study are publicly available[3].

The rest of the paper is structured as follows: we introduce our value and neighbor similarities in Section 2, and we delve into the

[2] A preliminary, abridged version of this paper appeared in [13].

[3] http://csd.uoc.gr/~vefthym/minoanER

blocking schemes and the blocking graph that lie at the core of our approach in Section 3. Section 4 describes the matching rules of our approach along with their implementation in Spark, while Section 5 overviews the main differences with the state-of-the-art ER methods. We present our thorough experimental analysis in Section 6 and we conclude the paper in Section 7.

2 BASIC DEFINITIONS

Given a KB $\mathcal{E}$, an entity description with a URI identifier i, denoted by $e_i \in \mathcal{E}$, is a set of attribute-value pairs about a real-world entity. When the identifier of an entity description e_j appears in the values of another entity description e_i, the corresponding attribute is called a *relation* and the corresponding value (e_j) a *neighbor* of e_i. More formally, the relations of e_i are defined as $relations(e_i) = \{p|(p,j) \in e_i \wedge e_j \in \mathcal{E}\}$, while its neighbors as $neighbors(e_i) = \{e_j|(p,j) \in e_i \wedge e_j \in \mathcal{E}\}$. For example, for the Wikidata KB in the left side of Figure 1 we have: $relations(Restaurant1) = \{$hasChef, territorial, inCountry$\}$, and $neighbors(Restaurant1) = \{$John Lake A, Bray, United Kingdom$\}$.

In the following, we exclusively consider clean-clean ER, i.e., the sub-problem of ER that seeks matches among two duplicate-free (clean) KBs. Thus, we simplify the presentation of our approach, but the proposed techniques can be easily generalized to more than two clean KBs or a single dirty KB, i.e., a KB that contains duplicates.

2.1 Entity similarity based on values

Traditionally, similarity between entities is computed based on their values. In our work, we apply a similarity measure based on the number and the frequency of common words between two values[4].

Definition 2.1. Given two KBs, $\mathcal{E}_1$ and $\mathcal{E}_2$, the **value similarity** of two entity descriptions $e_i \in \mathcal{E}_1, e_j \in \mathcal{E}_2$ is defined as: $valueSim(e_i,e_j)= \sum_{t \in tokens(e_i) \cap tokens(e_j)} \frac{1}{log_2(EF_{\mathcal{E}_1}(t) \cdot EF_{\mathcal{E}_2}(t)+1)}$, where $EF_{\mathcal{E}}(t) = |\{e_l|e_l \in \mathcal{E} \wedge t \in tokens(e_l)\}|$ stands for "Entity Frequency", which is the number of entity descriptions in $\mathcal{E}$ having token t in their values.

This value similarity shares the same intuition as TF-IDF in information retrieval. If two entities share many, infrequent tokens, then they have high value similarity. On the contrary, very frequent words (resembling stopwords in information retrieval) are not considered an important matching evidence, when they are shared by two descriptions, and therefore, they only contribute insignificantly to the $valueSim$ score. The number of common words is accounted by the number of terms that are considered in the sum and the frequency of those words is given by the inverse Entity Frequency (EF), similar to the inverse Document Frequency (DF) in information retrieval.

Proposition 1. $valueSim$ is a *similarity metric*, since it satisfies the following properties [5]:

- $valueSim(e_i,e_i) \geq 0$,
- $valueSim(e_i,e_j) = valueSim(e_j,e_i)$,
- $valueSim(e_i,e_i) \geq valueSim(e_i,e_j)$,
- $valueSim(e_i,e_i) = valueSim(e_j,e_j) = valueSim(e_i,e_j) \Leftrightarrow e_i{=}e_j$,
- $valueSim(e_i,e_j) + valueSim(e_j,e_z) \leq valueSim(e_i,e_z) + valueSim(e_j,e_j)$.

PROOF. Please refer to the extended version of this paper[5]. □

Note that $valueSim$ has the following properties: *(i)* it is not a normalized metric, since it can take any value in $[0,+\infty)$, with 0 denoting the existence of no common tokens in the values of the compared descriptions. *(ii)* The maximum contribution of a single common token between two descriptions is 1, in case this common token does not appear in the values of any other entity description, i.e., when $EF_{\mathcal{E}_1}(t) \cdot EF_{\mathcal{E}_2}(t) = 1$. *(iii)* It is a *schema-agnostic similarity metric*, as it disregards any schematic information[6].

2.2 Entity similarity based on neighbors

In addition to value similarity, we exploit the relations between descriptions to find the matching entities of the compared KBs. This can be done by aggregating the value similarity of all pairs of descriptions that are neighbors of the target descriptions.

Given the potentially high number of neighbors that a description might have, we propose considering only the most valuable neighbors for computing the neighbor similarity between two target descriptions. These are neighbors that are connected with the target descriptions via important relations, i.e., relations that exhibit high *support* and *discriminability*. Intuitively, high support for a particular relation p indicates that p appears in many entity descriptions, while high discriminability for p indicates that it has many distinct values. More formally:

Definition 2.2. The **support of a relation** $p \in \mathcal{P}$ in a KB $\mathcal{E}$ is defined as: $support(p) = \frac{|instances(p)|}{|\mathcal{E}|^2}$, where $instances(p) = \{(i,j)|e_i,e_j \in \mathcal{E} \wedge (p,j) \in e_i\}$.

Definition 2.3. The **discriminability of a relation** $p \in \mathcal{P}$ in a KB $\mathcal{E}$ is defined as: $discriminability(p) = \frac{|objects(p)|}{|instances(p)|}$, where $objects(p) = \{j|(i,j) \in instances(p)\}$.

Overall, we combine support and discriminability via their harmonic mean in order to locate the most important relations.

Definition 2.4. The **importance of a relation** $p \in \mathcal{P}$ in a KB $\mathcal{E}$ is defined as: $importance(p) = 2 \cdot \frac{support(p) \cdot discriminability(p)}{support(p)+discriminability(p)}$.

On this basis, we identify the most valuable relations and neighbors for every single entity description (i.e., *locally*). We use $topNrelations(e_i)$ to denote the N relations in $relations(e_i)$ with the maximum importance scores. Then, the **best neighbors** for e_i are defined as: $topNneighbors(e_i) = \{ne_i|(p,ne_i) \in e_i \wedge p \in topNrelations(e_i)\}$.

Intuitively, strong matching evidence (high value similarity) for the important neighbors leads to strong matching evidence for the target pair of descriptions. Hence, we formally define neighbor similarity as follows:

Definition 2.5. Given two KBs, $\mathcal{E}_1$ and $\mathcal{E}_2$, the **neighbor similarity** of two entity descriptions $e_i \in \mathcal{E}_1, e_j \in \mathcal{E}_2$ is:

$$neighborNSim(e_i,e_j)= \sum_{\substack{ne_i \in topNneighbors(e_i)\\ ne_j \in topNneighbors(e_j)}} valueSim(ne_i,ne_j).$$

Proposition 2. $neighborNSim$ is a similarity metric.

[4] We handle numbers and dates in the same way as strings, assuming string-dominated entities.

[5] http://csd.uoc.gr/~vefthym/DissertationEfthymiou.pdf

[6] Note that our value similarity metric is crafted for the token-level noise in literal values, rather than the character-level one. Yet, our overall approach is tolerant to character-level noise, as verified by our extensive experimental analysis with real datasets that include it. The reason is that it is highly unlikely for matching entities to have character-level noise in all their common tokens.

Proof. Given that *neighborNSim* is the sum of similarity metrics (*valueSim*), it is a similarity metric, too [5]. □

Neither *valueSim* nor *neighborNSim* are normalized, since the number of terms that contribute in the sums is an important matching evidence that can be mitigated if the values were normalized.

Example 2.6. Continuing our example in Figure 1, assume that the best two relations for $Restaurant1$ and $Restaurant2$ are:
$top2relations(Restaurant1)$ = {hasChef, territorial} and
$top2relations(Restaurant2)$ = {headChef, county}. Then,
$top2neighbors(Restaurant1)$ = {John Lake A, Bray} and
$top2neighbors(Restaurant2)$ = {Jonny Lake, Berkshire}, and
$neighbor2Sim(Restaurant1, Restaurant2)$ =
$valueSim(Bray, JonnyLake)$+$valueSim(John\ Lake\ A, Berkshire)$
+$valueSim(Bray, Berkshire)$+$valueSim(John\ Lake\ A, Jonny\ Lake)$.
Note that since we don't have a relation mapping, we also consider the comparisons (Bray, JonnyLake) and (John Lake A, Berkshire).

Entity Names. From every KB, we also derive the *global* top-k attributes of highest importance, whose *literal* values act as **names** for any description e_i that contains them. Their support is simply defined as: $support(p) = |subjects(p)|/|\mathcal{E}|$, where $subjects(p) = \{i|(i, j) \in instances(p)\}$ [32]. Based on these statistics, function $name(e_i)$ returns the names of e_i, and $\mathcal{N}_x$ denotes all names in a KB $\mathcal{E}_x$. In combination with $topNneighbors(e_i)$, this function covers both local and global property importance, exploiting both the rare and frequent attributes that are distinctive enough to designate matching entities.

3 BLOCKING

To enhance performance, *blocking* is typically used as a preprocessing step for ER in order to reduce the number of unnecessary comparisons, i.e., comparisons between descriptions that do not match. After blocking, each description can be compared only to others placed within the same block. The desiderata of blocking are [6]: (i) to place matching descriptions in common blocks (*effectiveness*), and (ii) to minimize the number of suggested comparisons (*time efficiency*). However, efficiency dictates skipping many comparisons, possibly yielding many missed matches, which in turn implies low effectiveness. Thus, the main objective of blocking is to achieve a good trade-off between minimizing the number of suggested comparisons and minimizing the number of missed matches [7].

In general, blocking methods are defined over key values that can be used to decide whether or not an entity description could be placed in a block using an *indexing function* [7]. The 'uniqueness' of key values determines the number of entity descriptions placed in the same block, i.e., which are considered as *candidate matches*. More formally, the building blocks of a blocking method can be defined as [3]:

- An *indexing function* $h_{key} : \mathcal{E} \rightarrow 2^B$ is a unary function that, when applied to an entity description using a specific blocking key, it returns as a value the subset of the set of all blocks B, under which the description will be indexed.
- A *co-occurrence function* $o_{key} : \mathcal{E} \times \mathcal{E} \rightarrow \{true, false\}$ is a binary function that, when applied to a pair of entity descriptions, it returns 'true' if the intersection of the sets of blocks produced by the indexing function on its arguments, is non-empty, and 'false' otherwise; $o_{key}(e_k, e_l) = true$ iff $h_{key}(e_k) \cap h_{key}(e_l) \neq \emptyset$.

In this context, each pair of descriptions whose co-occurrence function returns 'true' shares at least one common block, and the distinct union of the block elements is the input entity collection (i.e., all the descriptions from a set of input KBs). Formally:

Definition 3.1. Given an entity collection $\mathcal{E}$, **atomic blocking** is defined by an indexing function h_{key} for which the generated blocks, B^{key}=$\{b_1^{key}, \ldots, b_m^{key}\}$, satisfy the following conditions:

(i) $\forall e_k, e_l \in b_i^{key} : b_i^{key} \in B^{key}, o_{key}(e_k, e_l) = true$,
(ii) $\forall (e_k, e_l) : o_{key}(e_k, e_l)$=$true, \exists b_i^{key} \in B^{key}, e_k, e_l \in b_i^{key}$,
(iii) $\bigcup_{b_i^{key} \in B^{key}} b_i^{key} = \mathcal{E}$.

Given that a single key is not enough for indexing loosely structured and highly heterogeneous entity descriptions, we need to consider several keys that the indexing function will exploit to build different sets of blocks. Such a composite blocking method is characterized by a disjunctive co-occurrence function over the atomic blocks, and it is formally defined as:

Definition 3.2. Given an entity collection $\mathcal{E}$, **disjunctive blocking** is defined by a set of indexing functions H, for which the generated blocks $B = \bigcup_{h_{key} \in H} B^{key}$ satisfy the following conditions:

(i) $\forall e_k, e_l \in b : b \in B, o_H(e_k, e_l) = true$,
(ii) $\forall (e_k, e_l) : o_H(e_k, e_l) = true, \exists b \in B, e_k, e_l \in b$,
where $o_H(e_k, e_l) = \bigvee_{h_{key} \in H} o_{key}(e_k, e_l)$.

Atomic blocking can be seen as a special case of composite blocking, consisting of a singleton set, i.e., $H = \{h_{key}\}$.

3.1 Composite Blocking Scheme

To achieve a good trade-off between effectiveness and efficiency, our *composite blocking scheme* assesses the name and value similarity of the candidate matches in combination with similarity evidence provided by their neighbors on important relations. We consider the blocks constructed for all entities $e_i \in \mathcal{E}$ using the indexing function $h_i(\cdot)$ both over entity names ($\forall n_j \in names(e_i) : h_N(n_j)$) and tokens ($\forall t_j \in tokens(e_i) : h_T(t_j)$). The composite blocking scheme O of MinoanER is defined by the following disjunctive co-occurrence condition of any two entities $e_i, e_j \in \mathcal{E}$:
$O(e_i, e_j) = o_N(e_i, e_j) \vee o_T(e_i, e_j) \vee$
$(\bigvee_{(e_i', e_j') \in topNneighbors(e_i) \times topNneighbors(e_j)} o_T(e_i', e_j'))$, where o_N, o_T is the co-occurrence function applied on names and tokens, respectively. Intuitively, two entities are placed in a common block, and are then considered candidate matches, if at least one of the following three cases holds: (i) they have the same name, which is not used by any other entity, in which case the common block contains only those two entities, or (ii) they have at least one common word in any of their values, in which case the size of the common block is given by the product of the Entity Frequency (EF) of the common term in the two input collections, or (iii) their top neighbors share a common word in any of their values. Note that token blocking (i.e., h_T) allows for deriving $valueSim$ from the size of blocks shared by two descriptions. As a result, no additional blocks are needed to assess neighbor similarity of candidate entities: token blocking is sufficient also for estimating $neighborNsim$ according to Definition 2.5.

3.2 Disjunctive Blocking Graph

The disjunctive blocking graph G is an abstraction of the disjunctive co-occurrence condition of candidate matches in blocks. Nodes represent candidates from our input entity descriptions, while edges represent pairs of candidates for which at least one

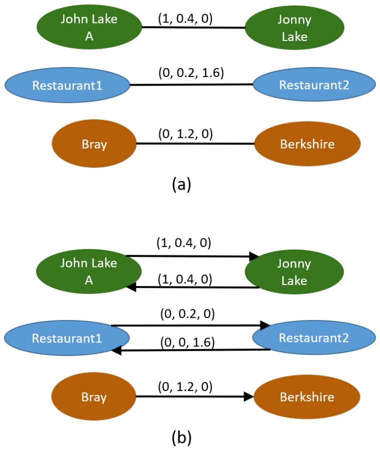


Figure 3: (a) Parts of the disjunctive blocking graph corresponding to Figure 1, and (b) the corresponding blocking graph after pruning.

of the co-occurrence conditions is 'true'. Each edge is actually labeled with three weights that quantify similarity evidence on names, tokens and neighbors of candidate matches. Specifically, the disjunctive blocking graph of MinoanER is a graph $G = (V, E, \lambda)$, with λ assigning to each edge a label (α, β, γ), where α is '1' if $o_N(e_i, e_j)$ is true and the name block in which e_i, e_j co-occur is of size 2, and '0' otherwise, $\beta = valueSim(e_i, e_j)$, and $\gamma = neighborNSim(e_i, e_j)$. More formally:

Definition 3.3. Given a block collection $B = \bigcup_{h_{key} \in H} B^{key}$, produced by a set of indexing functions H, the **disjunctive blocking graph** for an entity collection $\mathcal{E}$, is a graph $G = (V, E, \lambda)$, where each node $v_i \in V$ represents a distinct description $e_i \in \mathcal{E}$, and each edge $<v_i, v_j> \in E$ represents a pair $e_i, e_j \in \mathcal{E}$ for which $O(e_i, e_j) = `true'$; $O(e_i, e_j)$ is a disjunction of the atomic co-occurrence functions o^k defined along with H, and $\lambda : E \longrightarrow R^n$ is a labeling function assigning a tuple $[w^1, \ldots, w^n]$ to each edge, where w^k is a weight associated with each co-occurrence function o^k of H.

Definition 3.3 covers the cases of an entity collection $\mathcal{E}$ being composed of one, two, or more KBs. When matching k KBs, assuming that all of them are clean, the disjunctive blocking graph is k-partite, with each of the k KBs corresponding to a different independent set of nodes, i.e., there are only edges between descriptions from different KBs. The only information needed to match multiple KBs is to which KB every description belongs, so as to add it to the corresponding independent set. Similarly, the disjunctive blocking graph covers dirty ER, as well.

Example 3.4. Consider the graph of Figure 3(a), which is part of the disjunctive blocking graph generated from Figure 1. John Lake A and Jonny Lake have a common name ("J. Lake"), and there is no other entity having this name, so there is an edge connecting them with $\alpha = 1$. Bray and Berkshire have common, quite infrequent tokens in their values, so their similarity (β in the edge connecting them) is quite high ($\beta = 1.2$). Since Bray is a top neighbor of Restaurant1 in Figure 1, and Berkshire is a top neighbor of Restaurant2, there is also an edge with a non-zero γ connecting Restaurant1 with Restaurant2. The γ score of this edge (1.6) is the sum of the β scores of the edges connecting Bray with Berkshire (1.2), and John Lake A with Jonny Lake (0.4).

3.3 Graph Weighting and Pruning Algorithms

Each edge in the blocking graph represents a suggested comparison between two descriptions. To reduce the number of comparisons suggested by the disjunctive blocking graph, we keep for each node the K edges with the highest β and the K edges with the highest γ weights, while *pruning* edges with trivial weights (i.e., (α, β, γ)=(0,0,0)), since they connect descriptions unlikely to match. Given that nodes v_i and v_j may have different top K edges based on β or γ, we consider each undirected edge in G as two directed ones, with the same initial weights, and perform pruning on them.

Example 3.5. Figure 3(b) shows the pruned version of the graph in Figure 3(a). Note that *the blocking graph is only a conceptual model, which we do not materialize; we retrieve any necessary information from computationally cheap inverted indices.*

The process of weighting and pruning the edges of the disjunctive blocking graph is described in Algorithm 1. Initially, the graph contains no edges. We start adding edges by checking the name blocks B_N (Lines 5-9). For each name block b that contains exactly two entities, one from each KB, we create an edge with α=1 linking those entities (note that in Algorithm 1, b^k, $k \in \{1, 2\}$, denotes the sub-block of b that contains the entities from $\mathcal{E}_k$, i.e., $b^k \subseteq \mathcal{E}_k$). Then, we compute the β weights (Lines 10-14) by running a variation of Meta-blocking [27], adapted to our value similarity metric (Definition 2.1). Next, we keep for each entity, its connected nodes from the K edges with the highest β (Lines 15-18). Line 20 calls the procedure for computing the top in-neighbors of each entity, which operates as follows: first, it identifies each entity's *topNneigbors* (Lines 36-43) and then, it gets their reverse; for every entity e_i, we get the entities $topInNeighbors[i]$ that have e_i as one of their $topNneighbors$ (Lines 44-47). This allows for estimating the γ weights according to Definition 2.5. To avoid re-computing the value similarities that are necessary for the γ computations, we exploit the already computed βs. For each pair of entities $e_i \in \mathcal{E}_1$, $e_j \in \mathcal{E}_2$ that are connected with an edge with $\beta > 0$, we assign to each pair of their $inNeighbors$, (in_i, in_j), a partial γ equal to this β (Lines 20-27). After iterating over all such entity pairs e_i, e_j, we get their total neighbor similarity, i.e., $\gamma[i, j] = neighborNsim(e_i, e_j)$. Finally, we keep for each entity, its K neighbors with the highest γ (Lines 28-33).

The time complexity of Algorithm 1 is dominated by the processing of value evidence, which iterates twice over all comparisons in the token blocks B_T. In the worst-case, this results in one computation for every pair of entities, i.e., $O(|\mathcal{E}_1| \cdot |\mathcal{E}_2|)$. In practice, though, we bound the number of computations by removing excessively large blocks that correspond to highly frequent tokens (e.g., stop-words). Following [27], this is carried out by Block Purging [26], which ensures that the resulting blocks involve two orders of magnitude fewer comparisons than the brute-force approach, without any significant impact on recall. This complexity is higher than that of name and neighbor evidence, which are both linearly dependent on the number of input entities. The former involves a single iteration over the name blocks B_N, which amount to $|\mathcal{N}_1 \cap \mathcal{N}_2|$, as there is one block for every name shared by $\mathcal{E}_1$ and $\mathcal{E}_2$. For neighbor evidence, Algorithm 1 checks all pairs of N neighbors between every entity e_i and its K most value-similar descriptions, performing $K \cdot N^2 \cdot (|\mathcal{E}_1| + |\mathcal{E}_2|)$ operations; the cost of estimating the top in-neighbors for each entity is lower, dominated by the ordering of all relations in $\mathcal{E}_1$ and $\mathcal{E}_2$ (i.e., $|R_{max}| \cdot log|R_{max}|$), where $|R_{max}|$ stands for the maximum number of relations in one of the KBs.

Algorithm 1: Disjunctive Blocking Graph Construction.

```
Input: $\mathcal{E}_1$, $\mathcal{E}_2$, the blocks from name and token blocking, $B_N$ and $B_T$
Output: A disjunctive blocking graph $G$.
procedure getCompositeBlockingGraph($\mathcal{E}_1$, $\mathcal{E}_2$, $B_N$, $B_T$)
    $V \leftarrow \mathcal{E}_1 \cup \mathcal{E}_2$;
    $E \leftarrow \emptyset$;
    $W \leftarrow \emptyset$ ;                                  // init. to $(0, 0, 0)$
    // Name Evidence
    for $b \in B_N$ do
        if $|b^1| \cdot |b^2| = 1$ then        // only 1 comparison in $b$
            $e_i \leftarrow b^1.get(0)$, $e_j \leftarrow b^2.get(0)$ ;    // entities in b
            $E \leftarrow E \cup \{< v_i, v_j >\}$;
            $W \leftarrow W.set(\alpha, < v_i, v_j >, 1)$;
    // Value Evidence
    for $e_i \in \mathcal{E}_1$ do
        $\beta[] \leftarrow \emptyset$ ;        // value weights wrt all $e_j \in \mathcal{E}_2$
        for $b \in B_T \wedge b \cap e_i \neq \emptyset$ do
            for $e_j \in b^2$ do                        // $e_j \in \mathcal{E}_2$
                $\beta[j] \leftarrow \beta[j] + 1/log_2(|b^1| \cdot |b^2| + 1)$ ;     // valueSim
        $ValueCandidates \leftarrow getTopCandidates(\beta[], K)$;
        for $e_j \in ValueCandidates$ do
            $E \leftarrow E \cup \{< v_i, v_j >\}$;
            $W \leftarrow W.set(\beta, < v_i, v_j >, \beta[j])$;
    for $e_i \in \mathcal{E}_2$ do ...;            // ...do the same for $\mathcal{E}_2$
    // Neighbor Evidence
    $inNeighbors[] \leftarrow getTopInNeighbors(\mathcal{E}_1, \mathcal{E}_2)$;
    $\gamma[][] \leftarrow \emptyset$ ;     // neighbor weights wrt all $e_i, e_j \in V$
    for $e_i \in \mathcal{E}_1$ do
        for $e_j \in \mathcal{E}_2$, s.t. $W.get(\beta, < v_i, v_j >) > 0$ do
            for $in_j \in inNeighbors[j]$ do
                for $in_i \in inNeighbors[i]$ do        // neighborNSim
                    $\gamma[i][j] \leftarrow \gamma[i][j] + W.get(\beta, < n_i, n_j >)$;
    for $e_i \in \mathcal{E}_2$ do ...;            // ...do the same for $\mathcal{E}_2$
    for $e_i \in \mathcal{E}_1$ do
        $NeighborCandidates \leftarrow getTopCandidates(\gamma[i][], K)$;
        for $e_j \in NeighborCandidates$ do
            $E \leftarrow E \cup \{< v_i, v_j >\}$;
            $W.set(\gamma, < v_i, v_j >, \gamma[i][j])$;
    for $e_i \in \mathcal{E}_2$ do ...;            // ...do the same for $\mathcal{E}_2$
    return $G = (V, E, W)$;
procedure getTopInNeighbors($\mathcal{E}_1$, $\mathcal{E}_2$)
    $topNeighbors[] \leftarrow \emptyset$ ;      // one list for each entity
    $globalOrder \leftarrow$ sort $\mathcal{E}_1$'s relations by importance;
    for $e \in \mathcal{E}_1$ do
        $localOrder(e) \leftarrow relations(e).sortBy(globalOrder)$;
        $topNrelations \leftarrow localOrder(e).topN$;
        for $(p, o) \in e$, where $p \in topNrelations$ do
            $topNeighbors[e].add(o)$;
    for $e_i \in \mathcal{E}_2$ do ...;            // ...do the same for $\mathcal{E}_2$
    $topInNeighbors[] \leftarrow \emptyset$;      // the reverse of topNeighbors
    for $e \in \mathcal{E}_1 \cup \mathcal{E}_2$ do
        for $ne \in topNeighbors[e]$ do
            $topInNeighbors[ne].add(e)$;
    return $topInNeighbors$;
```

4 NON-ITERATIVE MATCHING PROCESS

Our matching method receives as input the disjunctive blocking graph G and performs four steps – unlike most existing works, which involve a data-driven iterative process. In every step, a matching rule is applied with the goal of extracting new matches from the edges of G by analyzing their weights. The functionality of our algorithm is outlined in Algorithm 2. Next, we describe its rules in the order they are applied:

Algorithm 2: Non-iterative Matching.

```
   Input: $\mathcal{E}_1$, $\mathcal{E}_2$, The pruned, directed disjunctive blocking graph $G$.
   Output: A set of matches $M$.
1  $M \leftarrow \emptyset$;                            // The set of matches
   // Name Matching Value (R1)
2  for $< v_i, v_j > \in G.E$ do
3      if $G.W.get(\alpha, < v_i, v_j >) = 1$ then
4          $M \leftarrow M \cup (e_i, e_j)$;
   // Value Matching Value (R2)
5  for $v_i \in G.V$ do
6      if $e_i \in \mathcal{E}_1 \setminus M$ then        // check the smallest KB for
        efficiency
7          $v_j \leftarrow argmax_{v_k \in G.V} G.W.get(\beta, < v_i, v_k >)$ ;    // top
            candidate
8          if $G.W.get(\beta, < v_i, v_j >) \geq 1$ then
9              $M \leftarrow M \cup (e_i, e_j)$;
   // Rank Aggregation Matching Value (R3)
10 for $v_i \in G.V$ do
11     if $e_i \in \mathcal{E}_1 \cup \mathcal{E}_2 \setminus M$ then
12         $agg[] \leftarrow \emptyset$;       // Aggregate scores, init. zeros
13         $valCands \leftarrow G.valCand(e_i)$ ; // nodes linked to $e_i$
            in decr. $\beta$
14         $rank \leftarrow |valCands|$;
15         for $e_j \in valCands$ do
16             $agg[e_i].update(e_j, \theta \cdot rank/|valCands|)$;
17             $rank \leftarrow rank - 1$;
18         $ngbCands \leftarrow G.ngbCand(e_i)$ ; // nodes linked to $e_i$
            in decr. $\gamma$
19         $rank \leftarrow |ngbCands|$;
20         for $e_j \in ngbCands$ do
21             $agg[e_i].update(e_j, (1 - \theta) \cdot rank/|ngbCands|)$;
22             $rank \leftarrow rank - 1$;
23     $M \leftarrow M \cup (e_i, getTopCandidate(agg[e_i]))$;
   // Reciprocity Matching Value (R4)
24 for $(e_i, e_j) \in M$ do
25     if $< v_i, v_j > \notin G.E \vee < v_j, v_i > \notin G.E$ then
26         $M \leftarrow M \setminus (e_i, e_j)$;
27 return $M$;
```

Name Matching Rule (R1). The matching evidence of R1 comes from the entity names. It assumes that *two candidate entities match, if they, and only they, have the same name n.* Thus, R1 traverses G and for every edge $<v_i, v_j>$ with $\alpha = 1$, it updates the set of matches M with the corresponding descriptions (Lines 2-4 in Alg. 2). All candidates matched by R1 are not examined by the remaining rules.

Value Matching Rule (R2). It presumes that *two entities match, if they, and only they, share a common token t, or, if they share many infrequent tokens.* Based on Definition 2.1, R2 identifies pairs of descriptions with high value similarity (Lines 5-9). To this end, it goes through every node v_i of G and checks whether the corresponding description stems from the smaller in size KB, for efficiency reasons (fewer checks), but has not been matched yet. In this case, it locates the adjacent node v_j with the maximum β weight (Line 7). If $\beta \geq 1$, R2 considers the pair (e_i, e_j) to be a match. Matches identified by R2 will not be considered in the sequel.

Rank Aggregation Matching Rule (R3). This rule identifies further matches for candidates whose value similarity is low ($\beta < 1$), yet their neighbor similarity (γ) could be relatively high. In this respect, the order of candidates rather than their absolute similarity values are used. Its functionality appears in Lines 10-23 of Algorithm 2. In essence, R3 traverses all nodes of G that correspond to a description that has not been matched yet. For every such node v_i, it retrieves two lists: the first one contains adjacent edges with a non-zero β weight, sorted in descending order (Line 13), while

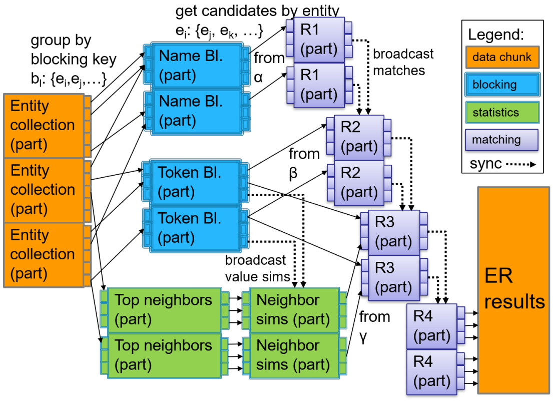


Figure 4: The architecture of MinoanER in Spark.

the second one includes the adjacent edges sorted in decreasing non-zero γ weights (Line 18). Then, R3 aggregates the two lists by considering the normalized ranks of their elements: assuming the size of a list is K, the first candidate gets the score K/K, the second one $(K-1)/K$, while the last one $1/K$. Overall, each adjacent node of v_i takes a score equal to the weighted summation of its normalized ranks in the two lists, as determined through the trade-off parameter $\theta \in (0,1)$ (Lines 16 & 21): the scores of the β list are weighted with θ and those of the γ list with 1-θ. At the end, v_i is matched with its top-1 candidate match v_j, i.e., the one with the highest aggregate score (Line 23). Intuitively, R3 *matches e_i with e_j, when, based on all available evidence, there is no better candidate for e_i than e_j.*

Reciprocity Matching Rule (R4). It aims to clean the matches identified by R1, R2 and R3 by exploiting the reciprocal edges of G. Given that the originally undirected graph G becomes directed after pruning (as it retains the best edges per node), a pair of nodes v_i and v_j are reciprocally connected when there are two edges between them, i.e., an edge from v_i to v_j and an edge from v_j to v_i. Hence, R4 aims to improve the precision of our algorithm based on the rationale that two entities are unlikely to match, when one of them does not even consider the other to be a candidate for matching. Intuitively, *two entity descriptions match, only if both of them "agree" that they are likely to match.* R4 essentially iterates over all matches detected by the above rules and discards those missing any of the two directed edges (Lines 24-26), acting more like a filter for the matches suggested by the previous rules.

Given a pruned disjunctive blocking graph, every rule can be formalized as a function that receives a pair of entities and returns true (T) if the entities match according to the rule's rationale, or false (F) otherwise, i.e., $Rn : \mathcal{E}_1 \times \mathcal{E}_2 \longrightarrow \{T, F\}$. In this context, we formally define the MinoanER matching process as follows:

Definition 4.1. The **non-iterative matching** of two KBs $\mathcal{E}_1$, $\mathcal{E}_2$, denoted by the Boolean matrix $M(\mathcal{E}_1, \mathcal{E}_2)$, is defined as a filtering problem of the pruned disjunctive blocking graph G: $M(e_i, e_j) = (\text{R1}(e_i, e_j) \vee \text{R2}(e_i, e_j) \vee \text{R3}(e_i, e_j)) \wedge \text{R4}(e_i, e_j)$.

The time complexity of Algorithm 2 is dominated by the size of the pruned blocking graph G it receives as input, since R1, R2 and R3 essentially go through all directed edges in G (in practice, though, R1 reduces the edges considered by R2 and R3, and so does R2 for R3). In the worst case, G contains $2K$ directed edges for every description in $\mathcal{E}_1 \cup \mathcal{E}_2$, i.e., $|V|_{max} = 2 \cdot K \cdot (|\mathcal{E}_1| + |\mathcal{E}_2|)$. Thus, the overall complexity is linear with respect to the number of input descriptions, i.e., $O(|\mathcal{E}_1| + |\mathcal{E}_2|)$, yielding high scalability.

4.1 Implementation in Spark

Figure 4 shows the architecture of MinoanER implementation in Spark. Each process is executed in parallel for different chunks of input, in different Spark workers. Each dashed edge represents a synchronization point, at which the process has to wait for results produced by different data chunks (and different Spark workers).

In more detail, Algorithm 1 is adapted to Spark by applying name blocking simultaneously with token blocking and the extraction of top neighbors per entity. Name blocking and token blocking produce the sets of blocks B_N and B_T, respectively, which are part of the algorithm's input. The processing of those blocks in order to estimate the α and β weights (Lines 5-9 for B_N and Lines 10-18 for B_T) takes place during the construction of the blocks. The extraction of top neighbors per entity (Line 20) runs in parallel to these two processes and its output, along with the β weights, is given to the last part of the graph construction, which computes the γ weights for all entity pairs with neighbors co-occuring in at least one block (Lines 21-33).

To minimize the overall run-time, Algorithm 2 is adapted to Spark as follows: R1 (Lines 2-4) is executed in parallel with name blocking and the matches it discovers are broadcasted to be excluded from subsequent rules. R2 (Lines 5-9) runs after both R1 and token blocking have finished, while R3 (Lines 10-23) runs after both R2 and the computation of neighbor similarities have been completed, skipping the already identified (and broadcasted) matches. R4 (Lines 24-26) runs in the end, providing the final, filtered set of matches. Note that during the execution of every rule, each Spark worker contains only the partial information of the disjunctive blocking graph that is necessary to find the match of a specific node (i.e., the corresponding lists of candidates based on names, values, or neighbors).

5 RELATED WORK

To the best of our knowledge, there is currently no other Web-scale ER framework that is fully automated, non-iterative, schema-agnostic and massively parallel, at the same time. For example, WInte.r [22] is a framework that performs multi-type ER, also incorporating the steps of blocking, schema-level mapping and data fusion. However, it is implemented in a sequential fashion and its solution relies on a specific level of structuredness, i.e., on a schema followed by the instances to be matched. Dedoop [20] is a highly scalable ER framework that comprises the steps of blocking and supervised matching. However, it is the user that is responsible for selecting one of the available blocking and learning methods and for fine-tuning their internal parameters. This approach is also targeting datasets with a predefined schema. Dedupe [16] is a scalable open-source Python library (and a commercial tool built on this library) for ER; however, it is not fully automated, as it performs active learning, relying on human experts to decide for a first few difficult matching decisions. Finally, we consider progressive ER (e.g., [1]) orthogonal to our approach, as it aims to retrieve as many matches as possible as early as possible.

In this context, we compare MinoanER independently to state-of-the-art matching and blocking approaches for Web data.

Entity Matching. Two types of similarity measures are commonly used for entity matching [21, 31]. (i) *Value-based similarities* (e.g., Jaccard, Dice) usually assess the similarity of two descriptions based on the values of specific attributes. Our value similarity is a variation of ARCS, a Meta-blocking weighting scheme [12], which disregards any schema information and considers each entity description as a bag of words. Compared to

ARCS, though, we focus more on the *number than the frequency of common tokens between two descriptions*. (ii) *Relational similarity* measures additionally consider neighbor similarity by exploiting the value similarity of (some of) the entities' neighbors.

Based on the nature of the matching decision, ER can be characterized as *pairwise* or *collective*. The former relies exclusively on the value similarity of descriptions to decide if they match (e.g., [20]), while the latter iteratively updates the matching decision for entities by dynamically assessing the similarity of their neighbors (e.g., [2]). Typically, the starting point for this similarity propagation is a set of seed matches identified by a value-based blocking.

For example, SiGMa [21] starts with seed matches having identical entity names. Then, it propagates the matching decisions on the 'compatible' neighbors, which are linked with pre-aligned relations. For every new matched pair, the similarities of the neighbors are recomputed and their position in the priority queue is updated. LINDA [4] differs by considering as compatible neighbors those connected with relations having similar names (i.e., small edit distance). However, this requirement rarely holds in the extreme schema heterogeneity of Web data. RiMOM-IM [23, 31] is a similar approach, introducing the following heuristic: if two matched descriptions e_1, e_1' are connected via aligned relations r and r' and all their entity neighbors via r and r', except e_2 and e_2', have been matched, then e_2 and e_2' are also considered matches.

All these methods employ Unique Mapping Clustering for detecting matches: they place all pairs into a priority queue, in decreasing similarity. At each iteration, the top pair is considered a match, if none of its entities has been already matched. The process ends when the top pair has a similarity lower than t.

MinoanER employs Unique Mapping Clustering, too. Yet, it differs from SiGMa, LINDA and RiMOM-IM in five ways: (i) the matching process iterates over the disjunctive blocking graph, instead of the initial KBs. (ii) MinoanER employs statistics to automatically discover distinctive entity names and important relations. (iii) MinoanER exploits different sources of matching evidence (values, names and neighbors) to statically identify candidate matches already from the step of blocking. (iv) MinoanER does not aggregate different similarities in one similarity score; instead, it uses a disjunction of the different evidence it considers. (v) MinoanER is a static collective ER approach, in which all sources of similarity are assessed only once per candidate pair. By considering a composite blocking not only on the value but also on the neighbors similarity, we discover in a non-iterative way most of the matches returned by the data-driven convergence of existing systems, or even more (see Section 6).

PARIS [33] uses a probabilistic model to identify matches, based on previous matches and the functional nature of entity relations. A relation is considered *functional* if, for a source entity, there is only one destination entity. If $r(x, y)$ is a function in one KB and $r(x, y')$ a function in another KB, then y and y' are considered matches. The functionality of a relation and the alignment of relations along with previous matching decisions determine the decisions in subsequent iterations. *Unlike MinoanER, PARIS cannot deal with structural heterogeneity, while it targets both ontology and instance matching*.

Finally, [30] parallelizes the collective ER approach of [2], relying on a black-box matching and exploits a set of heuristic rules for structured entities. It essentially runs multiple instances of the matching algorithm in subsets of the input entities (similar to blocks), also keeping information for all the entity neighbors, needed for similarity propagation. Since some rules may require the results of multiple blocks, an iterative message-passing framework is employed. *Rather than a block-level synchronization, the MinoanER parallel computations in Spark require synchronization only across the 4 generic matching rules* (see Figure 4).

Regarding the matching rules, the ones employed by MinoanER based on values and names are similar to rules that have already been employed in the literature individually (e.g., in [21, 23, 31]). In this work, we use a combination of those rules for the first time, also introducing a novel rank aggregation rule to incorporate value and neighbor matching evidence. Finally, the idea of reciprocity has been applied to enhance the results of Meta-blocking [28], but was never used in matching.

Blocking. Blocking techniques for relational databases [6] rely on blocking keys defined at the schema-level. For example, the *Sorted Neighborhood* approach orders entity descriptions according to a sorting criterion and performs blocking based on it; it is expected that matching descriptions will be neighbors after the sorting, so neighbor descriptions constitute candidate matches [17]. Initially, entity descriptions are lexicographically ordered based on their blocking keys. Then, a window, resembling a block, of fixed length slides over the ordered descriptions, each time comparing only the contents of the window. An adaptive variation of the sorted neighborhood method is to dynamically decide on the size of the window [35]. In this case, adjacent blocking keys in the sorted descriptions that are significantly different from each other, are used as boundary pairs, marking the positions where one window ends and the next one starts. Hence, this variation creates non-overlapping blocks. In a similar line of work, the sorted blocks method [11] allows setting the size of the window, as well as the degree of desired overlap.

Another recent schema-based blocking method uses Maximal Frequent Itemsets (MFI) as blocking keys [19] – an itemset can be a set of tokens. Abstractly, each MFI of a specific attribute in the schema of a description defines a block, and descriptions containing the tokens of an MFI for this attribute are placed in a common block. Using frequent itemsets to construct blocks may significantly reduce the number of candidates for matching pairs. However, since many matching descriptions share few, or even no common tokens, further requiring that those tokens are parts of frequent itemsets is too restrictive. The same applies to the requirement for a-priori schema knowledge and alignment, thus resulting in many missed matches in the Web of Data.

Although blocking has been extensively studied for tabular data, the proposed approaches cannot be used for the Web of Data, since their blocking keys rely on the existence of a global schema. However, the use of schema-based blocking keys is inapplicable to the Web of Data, due to its extreme schema heterogeneity [15]: entity descriptions do not follow a fixed schema, as even a single description typically uses attributes defined in multiple LOD vocabularies. In this context, schema-agnostic blocking methods are needed instead. Yet, the schema-agnostic functionality of most blocking methods requires extensive fine-tuning to achieve high effectiveness [29]. The only exception is token blocking, which is completely parameter-free [26]. Another advantage of token blocking is that it allows for computing value similarity from its blocks, as they contain entities with *identical* blocking keys – unlike other methods like Dedoop [20] and Sorted Neighborhood [17], whose blocks contain entities with *similar* keys.

SiGMa [21] considers descriptions with at least two common tokens as candidate matches, which is more precise than our token blocking, but misses more matches. The missed matches will be considered in subsequent iterations, if their neighbor similarity is

Table 1: Dataset statistics.

	Restau-rant	Rexa-DBLP	BBCmusic-DBpedia	YAGO-IMDb
$\mathcal{E}_1$ entities	339	18,492	58,793	5,208,100
$\mathcal{E}_2$ entities	2,256	2,650,832	256,602	5,328,774
$\mathcal{E}_1$ triples	1,130	87,519	456,304	27,547,595
$\mathcal{E}_2$ triples	7,519	14,936,373	8,044,247	47,843,680
$\mathcal{E}_1$ av. tokens	20.44	40.71	81.19	15.56
$\mathcal{E}_2$ av. tokens	20.61	59.24	324.75	12.49
$\mathcal{E}_1/\mathcal{E}_2$ attributes	7 / 7	114 / 145	27 / 10,953	65 / 29
$\mathcal{E}_1/\mathcal{E}_2$ relations	2 / 2	103 / 123	9 / 953	4 / 13
$\mathcal{E}_1/\mathcal{E}_2$ types	3 / 3	4 / 11	4 / 59,801	11,767 / 15
$\mathcal{E}_1/\mathcal{E}_2$ vocab.	2 / 2	4 / 4	4 / 6	3 / 1
Matches	89	1,309	22,770	56,683

Table 2: Block statistics.

	Restaurant	Rexa-DBLP	BBCmusic-DBpedia	YAGO-IMDb
$\lvert B_N \rvert$	83	15,912	28,844	580,518
$\lvert B_T \rvert$	625	22,297	54,380	495,973
$\lvert\lvert B_N \rvert\rvert$	83	$6.71 \cdot 10^7$	$1.25 \cdot 10^7$	$6.59 \cdot 10^6$
$\lvert\lvert B_T \rvert\rvert$	$1.80 \cdot 10^3$	$6.54 \cdot 10^8$	$1.73 \cdot 10^8$	$2.28 \cdot 10^{10}$
$\lvert\mathcal{E}_1\rvert \cdot \lvert\mathcal{E}_2\rvert$	$7.65 \cdot 10^5$	$4.90 \cdot 10^{10}$	$1.51 \cdot 10^{10}$	$2.78 \cdot 10^{13}$
Precision	4.95	$1.81 \cdot 10^{-4}$	0.01	$2.46 \cdot 10^{-4}$
Recall	100.00	99.77	99.83	99.35
F1	9.43	$3.62 \cdot 10^{-4}$	0.02	$4.92 \cdot 10^{-4}$

strong, whereas *MinoanER identifies such matches from the step of blocking*. RiMOM-IM [31] computes the tokens' TF-IDF weights, takes the top-5 tokens of each entity, and constructs a block for each one, along with the attribute this value appears. *Compared to the full automation of MinoanER, this method requires attribute alignment.* [25] iteratively splits large blocks into smaller ones by adding attributes to the blocking key. This leads to a prohibitive technique for voluminous KBs of high Variety.

Disjunctive blocking schemes have been proposed for KBs of high [18] and low [3] levels of schema heterogeneity. Both methods, though, are of limited applicability, as they require labelled instances for their supervised learning. *In contrast, MinoanER copes with the Volume and Variety of the Web of Data, through an unsupervised, schema-agnostic, disjunctive blocking*.

Finally, LSH blocking techniques (e.g., [24]) hash descriptions multiple times, using a family of hash functions, so that similar descriptions are more likely to be placed into the same bucket than dissimilar ones. This requires tuning a similarity threshold between entity pairs, above which they are considered candidate matches. *This tuning is non-trivial, especially for descriptions from different domains, while its effectiveness is limited for nearly similar entities* (see Figure 2).

6 EXPERIMENTAL EVALUATION

In this section, we thoroughly compare MinoanER to state-of-the-art tools and a heavily fine-tuned baseline method.

Experimental Setup. All experiments were performed on top of Apache Spark v2.1.0 and Java 8, on a cluster of 4 Ubuntu 16.04.2 LTS servers. Each server has 236GB RAM and 36 Intel(R) Xeon(R) E5-2630 v4 @2.20GHz CPU cores.

Datasets. We use four established benchmark datasets with entities from real KBs. Their technical characteristics appear in Table 1. All KBs contain relations between the described entities.

Restaurant[7] contains descriptions of restaurants and their addresses from two different KBs. It is the smallest dataset in terms of the number of entities, triples, entity types[8], as well as the one using the smallest number of vocabularies. We use it for two reasons: *(i)* it is a popular benchmark, created by the Ontology Alignment Evaluation Initiative, and *(ii)* it offers a good example of a dataset with very high value and neighbor similarity between matches (Figure 2), involving the easiest pair of KBs to resolve.

Rexa-DBLP[9] contains descriptions of publications and their authors. The ground truth contains matches between both publications and authors. This dataset contains strongly similar matches in terms of values and neighbors (Figure 2). Although it is relatively easy to resolve, Table 1 shows that it exhibits the greatest difference with respect to the size of the KBs to be matched (DBLP is 2 orders of magnitude bigger than Rexa in terms of descriptions, and 3 orders of magnitude in terms of triples).

BBCmusic-DBpedia [14] contains descriptions of musicians, bands and their birthplaces, from BBCmusic and the BTC2012 version of DBpedia[10]. In our experiments, we consider only entities appearing in the ground truth, as well as their immediate in- and out-neighbors. The most challenging characteristic of this dataset is the high heterogeneity between its two KBs in terms of both schema and values: DBpedia contains ~11,000 different attributes, ~60,000 entity types, 953 relations, the highest number of different vocabularies (6), while using on average 4 times more tokens than BBCmusic to describe an entity. The latter feature means that all normalized, set-based similarity measures like Jaccard fail to identify such matches, since a big difference in the token set sizes yields low similarity values (see Figure 2). A thorough investigation has shown that in the median case, an entity description in this dataset contains only 2 words in its values that are used by both KBs [14].

YAGO-IMDb [33] contains descriptions of movie-related entities (e.g., actors, directors, movies) from YAGO[11] and IMDb[12]. Figure 2 shows that a large number of matches in this dataset has low value similarity, while a significant number has high neighbor similarity. Moreover, this is the biggest dataset in terms of entities and triples, challenging the scalability of ER tools, while it is the most balanced pair of KBs with respect to their relative size.

Baselines. In our experiments, we compare MinoanER against four state-of-the-art methods: SiGMa, PARIS, LINDA and RiMOM. PARIS is openly available, so we ran its original implementation. For the remaining tools, we report their performance from the original publications[13]. We also consider BSL, a custom baseline method that receives as input the disjunctive blocking graph *G*, before pruning, and compares every pair of descriptions connected by an edge in *G*. The resulting similarities are then processed by Unique Mapping Clustering. Unlike MinoanER, though, BSL disregards all evidence from entity neighbors, relying exclusively on value similarity. Yet, it optimizes its performance through a series of well-established string matching methods that undergo extensive fine-tuning on the basis of the ground-truth.

In more detail, we consider numerous configurations for the four parameters of BSL in order to maximize its F1: *(i)* The schema-agnostic representation of the values in every entity. BSL uses token *n*-grams for this purpose, with $n \in \{1, 2, 3\}$, thus representing every resource by the token uni-/bi-/tri-grams that appear

[7] http://oaei.ontologymatching.org/2010/im/

[8] Extracted using the attribute w3.org/1999/02/22-rdf-syntax-ns#type.

[9] oaei.ontologymatching.org/2009/instances/

[10] datahub.io/dataset/bbc-music, km.aifb.kit.edu/projects/btc-2012/

[11] www.yago-knowledge.org

[12] www.imdb.com

[13] RiMOM-IM [31] is openly available, but no execution instructions were made available to us.

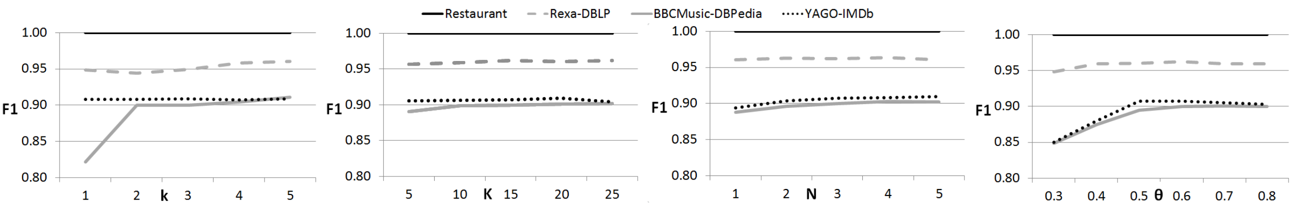


Figure 5: Sensitivity analysis of the four configuration parameters of our MinoanER.

in its values. *(ii)* The weighting scheme that assesses the importance of every token. We consider TF and TF-IDF weights. *(iii)* The similarity measure, for which we consider the following well-established similarities: Cosine, Jaccard, Generalized Jaccard and SiGMa (which applies exclusively to TF-IDF weights [21]). All measures are normalized to $[0, 1]$. *(iv)* The similarity threshold that prunes the entity pairs processed by Unique Mapping Clustering. We use all thresholds in $[0, 1)$ with a step of 0.05. In total, we consider 420 different configurations, reporting the highest F1.

6.1 Effectiveness Evaluation

Blocks Performance. First, we examine the performance of the blocks used by MinoanER (and BSL). Their statistics appear in Table 2. We observe that the number of comparisons in token blocks ($||B_T||$) is at least 1 order of magnitude larger than those of name blocks ($||B_N||$), even if the latter may involve more blocks ($|B_N|>|B_T|$ over YAGO-IMDb). In fact, $||B_N||$ seems to depend linearly on the number of input descriptions, whereas $||B_T||$ seems to depend quadratically on that number. Nevertheless, the overall comparisons in $B_T \cup B_N$ are at least 2 orders of magnitude lower than the Cartesian product $|E_1| \cdot |E_2|$, even though recall is consistently higher than 99%. On the flip side, both precision and F-Measure (F1) remain rather low.

Parameter Configuration. Next, we investigate the robustness of our method with respect to its internal configuration. To this end, we perform a *sensitivity analysis*, using the following meaningful values for the four parameters of MinoanER: $k \in \{1, 2, 3, 4, 5\}$ (the number of most distinct predicates per KB whose values serve as names), $K \in \{5, 10, 15, 2, 25\}$ (the number of candidate matches per entity from values and neighbors), $N \in \{1, 2, 3, 4, 5\}$ (the number of the most important relations per entity), and $\theta \in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8\}$ (the trade-off between value- vs neighbor-based candidates). Preliminary experiments demonstrated that the configuration $(k, K, N, \theta) = (2, 15, 3, 0.6)$ yields a performance very close to the average one. Therefore, we use these parameter values as the default ones in our sensitivity analysis.

In more detail, we sequentially vary the values of one parameter, keeping the others fixed to their default value, so as to examine its effect on the overall F1 of MinoanER. The results appear in the diagrams of Figure 5. We observe that MinoanER is quite robust in most cases, as small changes in a parameter value typically lead to an insignificant change in F1. This should be attributed to the composite functionality of MinoanER and its four matching rules,in particular: even if one rule is misconfigured, the other rules make up for the lost matches. There are only two exceptions:

(i) There is a large increase in F1 over BBCmusic-DBpedia when k increases from 1 to 2. The former value selects completely different predicates as names for the two KBs, due to the schema heterogeneity of DBpedia, thus eliminating the contribution of the name matching rule to F1. This is ameliorated for k=2.

(ii) F1 is significantly lower over BBCmusic-DBpedia and YAGO-IMDb for $\theta < 0.5$. This should be expected, since Figure 2 demonstrates that both datasets are dominated by nearly-similar matches, with the value similarity providing insufficient evidence for detecting them. Hence, θ should promote neighbor-similarity at least to the same level as the value-similarity (i.e., $\theta \geq 0.5$).

Table 3: Evaluation of MinoanER in comparison to the state-of-the-art methods and the heavily fine-tuned baseline, BSL.

		Restau-rant	Rexa-DBLP	BBCmusic-DBpedia	YAGO-IMDb
SiGMa [21]	Prec.	99	**97**	-	**98**
	Recall	94	90	-	85
	F1	97	94	-	91
LINDA [4]	Prec.	**100**	-	-	-
	Recall	63	-	-	-
	F1	77	-	-	-
RiMOM [23]	Prec.	86	80	-	-
	Recall	77	72	-	-
	F1	81	76	-	-
PARIS [33]	Prec.	95	93.95	19.40	94
	Recall	88	89	0.29	90
	F1	91	91.41	0.51	**92**
BSL	Prec.	**100**	96.57	85.20	11.68
	Recall	**100**	83.96	36.09	4.87
	F1	**100**	89.82	50.70	6.88
MinoanER	Prec.	**100**	**96.74**	**91.44**	91.02
	Recall	**100**	**95.34**	**88.55**	**90.57**
	F1	**100**	**96.04**	**89.97**	90.79

As a result, next, we can exclusively consider the configuration $(k, K, N, \theta) = (2, 15, 3, 0.6)$ for MinoanER. This is the suggested global configuration that works well over all datasets, but parameter tuning per individual dataset may yield better results.

Comparison with Baselines. Table 3 shows that MinoanER offers competitive performance when matching KBs with few attributes and entity types, despite requiring no domain-specific input. Specifically, it achieves 100% F1 in Restaurant, which is 3% higher than SiGMa, 9% higher than PARIS, and ~20% higher than LINDA and RiMOM. BSL also achieves perfect F1, due to the strongly similar matches (see Figure 2). In Rexa-DBLP, MinoanER also outperforms all existing ER methods. It is 2% better than SiGMa in F1, 4.6% better than PARIS, 20% better than RiMOM, and 6% better than BSL. In YAGO-IMDb, MinoanER achieves similar performance to SiGMa (91% F1), with more identified matches (91% vs 85%), but lower precision (91% vs 98%). Compared to PARIS, its F1 is 1% lower, due to 3% lower precision, despite the 1% better recall. The high structural similarity between the two KBs make this dataset a good use case for PARIS. BSL exhibits the worst performance, due to the very low value similarity of matches in this KB. Most importantly, MinoanER achieves the best performance by far over the highly heterogeneous KBs of BBCmusic-DBpedia. PARIS struggles to identify the matches, with BSL performing significantly better, but still poorly in absolute numbers. In contrast, MinoanER succeeds in identifying 89% of matches with 91% precision, achieving a 90% F1.

Comparing the performance of MinoanER to that of its input blocks, precision raises by several orders of magnitude at the cost of slightly lower recall. The lower recall is caused by missed

Table 4: Evaluation of matching rules.

		Restaurant	Rexa-DBLP	BBCmusic-DBpedia	YAGO-IMDb
R1	Precision	**100**	**97.36**	**99.85**	97.55
	Recall	68.54	87.47	66.11	66.53
	F1	81.33	92.15	79.55	79.11
R2	Precision	**100**	96.15	90.73	**98.02**
	Recall	**100**	30.56	37.01	69.14
	F1	**100**	46.38	52.66	81.08
R3	Precision	98.88	94.73	81.49	90.51
	Recall	98.88	**94.73**	**81.49**	**90.50**
	F1	98.88	**94.73**	**81.49**	**90.50**
¬R4	Precision	100	96.03	89.93	90.58
	Recall	100	96.03	89.93	90.57
	F1	100	96.03	89.93	90.58
No Neighbors	Precision	100	96.59	89.22	88.05
	Recall	100	95.26	85.36	87.42
	F1	100	95.92	87.25	87.73

matches close to the lower left corner of Figure 2, i.e., with very low value and neighbor similarities. This explains why the impact on recall is larger for BBCmusic-DBpedia and YAGO-IMDb.

Evaluation of Matching Rules. Table 4 summarizes the performance of each matching rule in Algorithm 2, when executed alone, as well as the overall contribution of neighbor similarity evidence.

• *Name Matching Rule (R1).* This rule achieves both high precision (> 97%) and a decent recall (> 66%) in all cases. Hence, given no other matching evidence, R1 alone yields good matching results, emphasizing precision, with only an insignificant number of its suggested matches being false positives. To illustrate the importance of this similarity evidence in real KBs, we have marked the matches with identical names in Figure 2 as bordered points. Thus, we observe that matches may agree on their names, regardless of their value and neighbor similarity evidence.

• *Value Matching Rule (R2).* This rule is also very precise (> 90% in all cases), but exhibits a lower recall (> 30%). Nevertheless, even this low recall is not negligible, especially when it complements the matches found from R1. In the case of strongly similar matches as in Restaurant, this rule alone can identify all the matches with perfect precision.

• *Rank Aggregation Matching Rule (R3).* The performance of this rule varies across the four datasets, as it relies on neighborhood evidence. For KBs with low value similarity (left part of Figure 2), this rule is the only solution for finding matches having no/different names. In BBCmusic-DBpedia and YAGO-IMDb, it has the highest contribution in recall and F1 of all matching rules, with the results for YAGO-IMDb being almost equivalent to those of Table 3 (YAGO-IMDb features the lowest value similarities in Figure 2). For KBs with medium value similarity (middle part of Figure 2), but not enough to find matches with R2, aggregating neighbor with value similarity is very effective. In Rexa-DBLP, R3 yields almost perfect results. Overall, R3 is the matching rule with the greatest F1 in 3 out of 4 datasets.

• *Reciprocity Matching Rule (R4).* Given that R4 is a filtering rule, i.e., it does not add new results, we measure its contribution by running the full workflow without it. Its performance in Table 4 should be compared to the results in Table 3. This comparison shows that this rule increases the precision of MinoanER, with a small, or no impact on recall. Specifically, it increases the precision of BBCmusic-DBpedia by 1.51%, while its recall is decreased by 1.38%, and in the case of YAGO-IMDb, it improves precision by 0.44% with no impact on recall. This results in an increase of 0.04% and 0.21% in F1. Overall, R4 is the weakest matching rule, yielding only a minor improvement in the results of MinoanER.

Contribution of neighbors. To evaluate the contribution of neighbor evidence in the matching results, we have repeated Algorithm 2, without the rule R3. Note that this experiment is not the same as our baseline; here, we use all the other rules, also operating on the pruned disjunctive blocking graph, while the baseline does not use our rules and operates on the unpruned graph. The results show that *neighbor evidence plays a minor or even no role in KBs with strongly similar entities*, i.e., Restaurant and Rexa-DBLP, *while having a bigger impact in KBs with nearly similar matches*, i.e., BBCmusic-DBpedia and YAGO-IMDb (see Figure 2). Specifically, compared to the results of Table 3, the use of neighbor evidence improves precision by 2.22% and recall by 3.19% in BBCmusic-DBpedia, while, in YAGO-IMDB, precision is improved by 2.97% and recall by 3.15%.

6.2 Efficiency Evaluation

To evaluate the scalability of matching in MinoanER, we present in Figure 6 the running times and speedup of matching for each dataset, as we change the number of available CPU cores in our cluster, i.e., the number of tasks that can run at the same time. In each diagram, the left vertical axis shows the running time and the right vertical axis shows the speedup, as we increase the number of available cores (from 1 to 72) shown in the horizontal axis[14]. Across all experiments, we have kept the same total number of tasks, which was defined as the number of all cores in the cluster multiplied by a parallelism factor of 3, i.e., 3 tasks are assigned to each core, when all cores are available. This was to ensure that each task would require the same amount of resources (e.g., memory), regardless of the number of available cores.

We observe that the running times decrease as more cores become available, and this decrease is steeper when using a small number of cores. For example, resolving Rexa-DBLP with 6 cores is 6 times faster than with 1 core, while it is 10 times faster with 12 cores than with 1 core (top-right of Figure 6). We observe a sub-linear speedup in all cases, which is expected when synchronization is required for different steps (see Section 4.1). Though, the bigger datasets have a speedup closer to linear than the smaller ones, because the Spark overhead is smaller with respect to the overall running time in these cases. We have also measured the percentage of time spent for the matching phase (Algorithm 2) compared to the total execution time of MinoanER. In Restaurant and Rexa-DBLP, matching takes 45% of the total time, in BBCmusic-DBpedia 30% and in YAGO-IMDb 20%. Thus, in all cases, matching takes less than half the execution time, while it takes smaller percentage of time as the tasks get bigger.

It is not possible to directly compare the efficiency of MinoanER with the competitive tools of Table 3; most of them are not publicly available, while the available ones do not support parallel execution using Spark. The running times reported in the original works are about sequential algorithms executed in machines with a different setting than ours. However, we can safely argue that our fixed-step process, as opposed to the data-iterative processes of existing works, boosts the efficiency of MinoanER at no cost in (or, in most cases, with even better) effectiveness. Indicatively, the running time of MinoanER for Rexa-DBLP was 3.5 minutes (it took PARIS 11 minutes on one of our cluster's servers - see Experimental Setup - for the same dataset), for BBCmusic-DBpedia it was 69 seconds (it took PARIS 3.5 minutes on one of

[14] We could not run MinoanER on the YAGO-IMDb dataset with only 1 core, due to limited space in a single machine, so we report its running time starting with a minimum of 4 cores. This means that the linear speedup for 72 tasks would not be 72, but 18 (72/4).

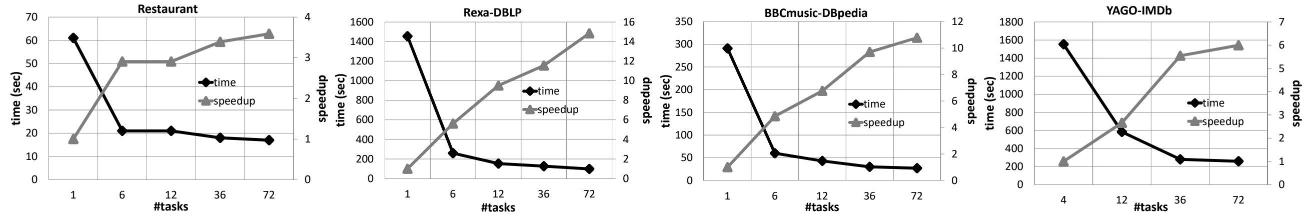


Figure 6: Scalability of matching in MinoanER w.r.t. running time (left vertical axis) and speedup (right vertical axis) as more cores are involved.

our cluster's servers), while the running time for YAGO-IMDb was 28 minutes (SiGMa reports 70 minutes, and PARIS reports 51 hours). In small datasets like Restaurant, MinoanER can be slower than other tools, as Spark has a setup overhead, which is significant for such cases (it took MinoanER 27 seconds to run this dataset, while PARIS needed 6 seconds).

7 CONCLUSIONS

In this paper, we have presented MinoanER, a fully automated, schema-agnostic and massively parallel framework for ER in the Web of Data. To resolve highly heterogeneous entities met in this context, MinoanER relies on schema-agnostic similarity metrics that consider both the content and the neighbors of entities. It exploits these metrics in a composite blocking scheme and conceptually builds a disjunctive blocking graph - a novel, comprehensive abstraction that captures all sources of similarity evidence. This graph of candidate matches is processed by a non-iterative algorithm that consists of four generic, schema-agnostic matching rules, with linear cost to the number of entity descriptions and robust performance with respect to their internal configuration.

The results show that neighbor evidence plays a minor role in KBs with strongly similar entities, such as Restaurant and Rexa-DBLP, while having a big impact in KBs with nearly similar entities, such as in BBCmusic-DBpedia and YAGO-IMDb. MinoanER achieves at least equivalent performance with state-of-the-art ER tools over KBs exhibiting low Variety, but outperforms them to a significant extent when matching KBs with high Variety. The employed matching rules (R1, R2, R3, R4) manage to cover a wide range of matches, as annotated in Figure 2, but there is still room for improvement, since the recall of blocking is better than that of matching. As an improvement, we will investigate how to create an ensemble of matching rules and how to set the parameters of pruning candidate pairs dynamically, based on the local similarity distributions of each node's candidates.

REFERENCES

[1] Yasser Altowim, Dmitri V. Kalashnikov, and Sharad Mehrotra. 2018. ProgressER: Adaptive Progressive Approach to Relational Entity Resolution. *ACM Trans. Knowl. Discov. Data* 12, 3 (2018), 33:1–33:45.

[2] Indrajit Bhattacharya and Lise Getoor. 2007. Collective entity resolution in relational data. *TKDD* 1, 1 (2007), 5.

[3] Mikhail Bilenko, Beena Kamath, and Raymond J. Mooney. 2006. Adaptive Blocking: Learning to Scale Up Record Linkage. In *ICDM*. 87–96.

[4] Christoph Böhm, Gerard de Melo, Felix Naumann, and Gerhard Weikum. 2012. LINDA: distributed web-of-data-scale entity matching. In *CIKM*. 2104–2108.

[5] Shihyen Chen, Bin Ma, and Kaizhong Zhang. 2009. On the similarity metric and the distance metric. *Theor. Comput. Sci.* 410, 24-25 (2009), 2365–2376.

[6] Peter Christen. 2012. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer.

[7] Peter Christen. 2012. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE TKDE* 24, 9 (2012), 1537–1555.

[8] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. 2015. *Entity Resolution in the Web of Data*. Morgan & Claypool Publishers.

[9] Jeremy Debattista, Christoph Lange, Sören Auer, and Dominic Cortis. 2018. Evaluating the quality of the LOD cloud: An empirical investigation. *Semantic Web* 9, 6 (2018), 859–901.

[10] Xin Luna Dong and Divesh Srivastava. 2015. *Big Data Integration*. Morgan & Claypool Publishers.

[11] Uwe Draisbach and Felix Naumann. 2011. A generalization of blocking and windowing algorithms for duplicate detection. In *ICDKE*. 18–24.

[12] Vasilis Efthymiou, George Papadakis, George Papastefanatos, Kostas Stefanidis, and Themis Palpanas. 2017. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Inf. Syst.* 65 (2017), 137–157.

[13] Vasilis Efthymiou, George Papadakis, Kostas Stefanidis, and Vassilis Christophides. 2018. Simplifying Entity Resolution on Web Data with Schema-agnostic Non-iterative Matching. In *ICDE*.

[14] Vasilis Efthymiou, Kostas Stefanidis, and Vassilis Christophides. 2015. Big data entity resolution: From highly to somehow similar entity descriptions in the Web. In *IEEE Big Data*. 401–410.

[15] Behzad Golshan, Alon Y. Halevy, George A. Mihaila, and Wang-Chiew Tan. 2017. Data Integration: After the Teenage Years. In *PODS*. 101–106.

[16] Forest Gregg and Derek Eder. 2017. Dedupe. (2017). https://github.com/dedupeio/dedupe

[17] Mauricio A. Hernàndez and Salvatore J. Stolfo. 1995. The Merge/Purge Problem for Large Databases. In *SIGMOD*. 127–138.

[18] Mayank Kejriwal. 2016. Disjunctive Normal Form Schemes for Heterogeneous Attributed Graphs. *CoRR* abs/1605.00686 (2016).

[19] Batya Kenig and Avigdor Gal. 2013. MFIBlocks: An effective blocking algorithm for entity resolution. *Inf. Syst.* 38, 6 (2013), 908–926.

[20] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Dedoop: Efficient Deduplication with Hadoop. *PVLDB* 5, 12 (2012), 1878–1881.

[21] Simon Lacoste-Julien, Konstantina Palla, Alex Davies, Gjergji Kasneci, Thore Graepel, and Zoubin Ghahramani. 2013. SIGMa: simple greedy matching for aligning large knowledge bases. In *KDD*. 572–580.

[22] Oliver Lehmberg, Christian Bizer, and Alexander Brinkmann. 2017. WInte.r - A Web Data Integration Framework. In *ISWC (Posters, Demos & Industry Tracks)*.

[23] Juanzi Li, Jie Tang, Yi Li, and Qiong Luo. 2009. RiMOM: A Dynamic Multistrategy Ontology Alignment Framework. *IEEE Trans. Knowl. Data Eng.* 21, 8 (2009), 1218–1232.

[24] Pankaj Malhotra, Puneet Agarwal, and Gautam Shroff. 2014. Graph-Parallel Entity Resolution using LSH & IMM. In *EDBT/ICDT Workshops*. 41–49.

[25] W.P. McNeill, Hakan Kardes, and Andrew Borthwick. 2012. Dynamic record blocking: efficient linking of massive databases in mapreduce. In *QDB*.

[26] George Papadakis, Ekaterini Ioannou, Themis Palpanas, Claudia Niederée, and Wolfgang Nejdl. 2013. A Blocking Framework for Entity Resolution in Highly Heterogeneous Information Spaces. *IEEE Trans. Knowl. Data Eng.* 25, 12 (2013), 2665–2682.

[27] George Papadakis, Georgia Koutrika, Themis Palpanas, and Wolfgang Nejdl. 2014. Meta-Blocking: Taking Entity Resolutionto the Next Level. *IEEE Trans. Knowl. Data Eng.* 26, 8 (2014), 1946–1960.

[28] George Papadakis, George Papastefanatos, Themis Palpanas, and Manolis Koubarakis. 2016. Boosting the Efficiency of Large-Scale Entity Resolution with Enhanced Meta-Blocking. *Big Data Research* 6 (2016), 43–63.

[29] George Papadakis, Jonathan Svirsky, Avigdor Gal, and Themis Palpanas. 2016. Comparative Analysis of Approximate Blocking Techniques for Entity Resolution. *PVLDB* 9, 9 (2016), 684–695.

[30] Vibhor Rastogi, Nilesh N. Dalvi, and Minos N. Garofalakis. 2011. Large-Scale Collective Entity Matching. *PVLDB* 4, 4 (2011), 208–218.

[31] Chao Shao, Linmei Hu, Juan-Zi Li, Zhichun Wang, Tong Lee Chung, and Jun-Bo Xia. 2016. RiMOM-IM: A Novel Iterative Framework for Instance Matching. *J. Comput. Sci. Technol.* 31, 1 (2016), 185–197.

[32] Dezhao Song and Jeff Heflin. 2011. Automatically Generating Data Linkages Using a Domain-Independent Candidate Selection Approach. In *ISWC*. 649–664.

[33] Fabian M. Suchanek, Serge Abiteboul, and Pierre Senellart. 2011. PARIS: Probabilistic Alignment of Relations, Instances, and Schema. *PVLDB* 5, 3 (2011), 157–168.

[34] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. 2015. Data X-Ray: A Diagnostic Tool for Data Errors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, 1231–1245.

[35] Su Yan, Dongwon Lee, Min-Yen Kan, and C. Lee Giles. 2007. Adaptive sorted neighborhood methods for efficient record linkage.. In *JCDL*. 185–194.

[36] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud*.

Extending Cross-Domain Knowledge Bases with Long Tail Entities using Web Table Data

Yaser Oulabi, Christian Bizer
Data and Web Science Group, University Mannheim
B6 26, 68159 Mannheim, Germany
{yaser,chris}@informatik.uni-mannheim.de

ABSTRACT

Cross-domain knowledge bases such as YAGO, DBpedia, or the Google Knowledge Graph are being used as background knowledge within an increasing range of applications including web search, data integration, natural language understanding, and question answering. The usefulness of a knowledge base for these applications depends on its completeness. Relational HTML tables from the Web cover a wide range of topics and describe very specific long tail entities, such as small villages, less-known football players, or obscure songs.

This systems and applications paper explores the potential of web table data for the task of completing cross-domain knowledge bases with descriptions of formerly unknown entities. We present the first system that handles all steps that are necessary for this task: schema matching, row clustering, entity creation, and new detection. The evaluation of the system using a manually labeled gold standard shows that it can construct formerly unknown instances and their descriptions from table data with an average F1 score of 0.80. In a second experiment, we apply the system to a large corpus of web tables extracted from the Common Crawl. This experiment allows us to get an overall impression of the potential of web tables for augmenting knowledge bases with long tail entities. The experiment shows that we can augment the DBpedia knowledge base with descriptions of 14 thousand new football players as well as 187 thousand new songs. The accuracy of the facts describing these instances is 0.90.

1 INTRODUCTION

Cross-domain knowledge bases like YAGO [18], DBpedia [20], Wikidata [30], or the Google Knowledge Graph are being employed for an increasing range of applications, including natural language processing, web search, and question answering.

The YAGO, DBpedia, and Wikidata knowledge bases all rely on data that has been extracted from Wikipedia and as a result cover mostly head instances that fulfill the Wikipedia notability criteria. Their coverage of less well known instances from the long tail is rather low [11]. As the usefulness of a knowledge base often increases with its completeness, adding long tail instances to an existing knowledge base is an important task.

Web tables [8], which are relational HTML tables extracted from the Web, contain large amounts of structured information, covering a wide range of topics, and describe very specific long tail instances. Web tables are thus a promising source of information for the task of augmenting cross-domain knowledge bases.

Augmenting knowledge bases with descriptions of long tail instances requires, on the one hand, identifying instances of a specific class that are not yet part of a knowledge base, and, on the other hand, compiling descriptions of the new instances according to the schema of the knowledge base. Two areas of related work are relevant for this task: Existing work on slot filling [11, 22, 23, 27, 29] focuses on adding missing facts describing existing instances to a knowledge base. The methods do not attempt to discover new instances. In contrast, existing research on set expansion [24, 31, 32] focuses on determining the names of new instances and is not concerned with compiling structured descriptions of those instances according to a schema of a knowledge base. As most set expansion methods disambiguate new instances solely based on names, they miss the potential of exploiting additional features for disambiguation.

As a result, no viable methods that are able to automatically augment a knowledge base with new instances and their descriptions exist. In this work, we close this gap by introducing and evaluating the first system that is able to generate descriptions of formerly unknown long-tail entities given a corpus of relational web tables. The system exploits the synergies between the task of identifying new instances and compiling descriptions of these instances using an iterative approach. The contributions of the paper are as follows:

- We introduce the first system that is able to generate descriptions of new instances given the set of all instances of a class from a knowledge base and a corpus of relational web tables.
- We evaluate our system using a manually built gold standard of annotated web tables and report the lessons learned from this experiment.
- We run our system over a large corpus of web tables which allows us to profile the general potential of web table data for augmenting knowledge bases with descriptions of long tail instances.

Figure 1 gives an overview of the overall process performed by our system. The process consists of four main steps which are executed in two iterations. We first apply schema matching methods to match web tables and attribute columns of those tables to classes and properties in the knowledge base. Second, a row clustering method identifies rows that describe the same entity. From these row clusters, the entity creation component creates entity descriptions according to the schema of the knowledge base. Finally, the new detection component determines whether an entity already exists in the knowledge base. We iterate over the pipeline a second time using the row clusters and entity-to-instance correspondences from the first run in order to refine the schema mapping. After the second run of the pipeline, entities identified as new are added to the knowledge base.

The paper is structured as follows: First, we describe the profile of the knowledge base, the web table corpus, and the gold standard that are used for the experiments throughout the paper. Section 3 describes and evaluates the individual steps of the overall process. Section 4 discusses the overall performance of the pipeline on the gold standard, while in Section 5 we run our

Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.34

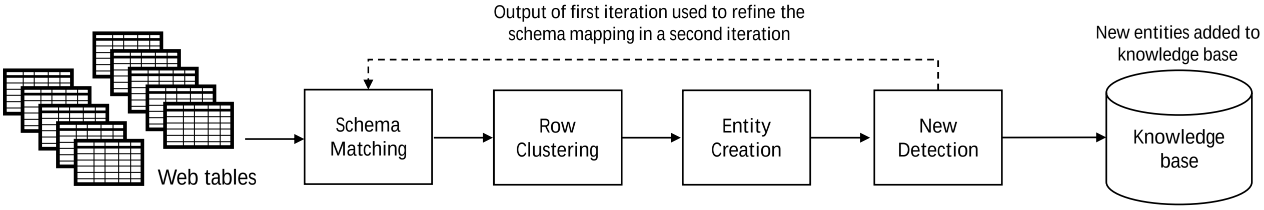

Output of first iteration used to refine the schema mapping in a second iteration
New entities added to knowledge base
Web tables
Schema Matching
Row Clustering
Entity Creation
New Detection
Knowledge base

Figure 1: Overview of the overall pipeline.

Table 1: Number of instances and facts for selected DBpedia classes.

Class	Instances	Facts
GF-Player	20,751	137,319
Song	52,533	315,414
Settlement	468,986	1,444,316

system on the large corpus of web tables in order to profile the overall potential of web tables for the task at hand. Section 6 compares our system to the related work.

2 EXPERIMENTAL SETUP

This section describes the datasets that we use to evaluate our system. We will first describe the knowledge base and the selection of classes we aim to extend. We then describe the web table corpus in which we hope to find new instances. We finally describe the gold standard that we use throughout this work.

2.1 Knowledge Base and Classes

We employ DBpedia [20] as the target knowledge base to be extended. It is extracted from Wikipedia and especially Wikipedia infoboxes. As a result, the covered instances are limited to those identified as notable by the Wikipedia community.

From DBpedia we selected three classes on which we focus throughout this work. To ensure that information covered by the classes is diverse, we selected each from a different first-level class, i.e. Agent, Work, and Place. To ensure that the classes are not too broad, we preferred classes further down in the hierarchy. As a result we chose the following three classes: GridironFootballPlayer (GF-Player), Song and Settlement. The class Song includes all instances of the class Single. Tables 1 and 2 provide an overview of the number of instances and facts, and the property densities of those classes. We only consider properties that have an initial density of at least 30 %. We use the 2014 release of DBpedia, as this release has been used in related work [22, 23, 26, 27], and its release date is also closer to the extraction of the web table corpus used in this work.

Table 1 shows that DBpedia already covers tens of thousands of instances for the profiled classes. This could indicate that most of the well-known instances are already covered, so that we are especially interested in finding instances from the long tail.

Table 2 also reveals that the density differs significantly from property to property. Only the properties of class Song have consistently high densities larger than 60 %. The football player class has many properties, but half of them have a density below 50 %. The class Settlement suffers from both, a small number of properties, and low densities for some of them.

Table 2: Number of facts and property densities for selected DBpedia properties.

Class	Property	Facts	Density
GF-Player	**birthDate**	20,218	97.43 %
GF-Player	**college**	19,281	92.92 %
GF-Player	**birthPlace**	17,912	86.32 %
GF-Player	**team**	13,349	64.33 %
GF-Player	**number**	11,430	55.08 %
GF-Player	**position**	11,240	54.17 %
GF-Player	**height**	10,059	48.47 %
GF-Player	**weight**	10,027	48.32 %
GF-Player	**draftYear**	7,947	38.30 %
GF-Player	**draftRound**	7,932	38.22 %
GF-Player	**draftPick**	7,924	38.19 %
Song	**genre**	47,040	89.54 %
Song	**musicalArtist**	45,097	85.85 %
Song	**recordLabel**	43,053	81.95 %
Song	**runtime**	42,035	80.02 %
Song	**album**	40,666	77.41 %
Song	**writer**	33,942	64.61 %
Song	**releaseDate**	31,696	60.34 %
Settlement	**country**	433,838	92.51 %
Settlement	**isPartOf**	416,454	88.80 %
Settlement	**populationTotal**	292,831	62.44 %
Settlement	**postalCode**	154,575	32.96 %
Settlement	**elevation**	146,618	31.26 %

2.2 Web Table Corpus

In this work, we utilize the english-language relational tables set of the Web Data Commons 2012 Web Table Corpus.[1] The set consists of 91.8 million tables. Table 3 gives an overview of the general characteristics of tables in the corpus. We can see that the majority of tables are rather short, with an average of 10.4 rows and a median of 2, whereas the average and median number of columns are 3.5 and 3. As a result, a table on average describes 10 instances with 30 values, which likely is a sufficient size and potentially useful for finding new instances and their descriptions. In [27] we have profiled the potential of the same corpus for the task of slot filling, meaning to find missing values for existing DBpedia instances.

For every table we assume that there is one attribute that contains the labels of the instances described by the rows. The remaining columns contain values, which potentially can be used to generate descriptions according to the knowledge base schema.

[1] http://webdatacommons.org/webtables/#toc3

Table 3: Characteristics of the web table corpus.

	Average	Median	Min	Max
Rows	10.37	2	1	35,640
Columns	3.48	3	2	713

Table 4: Number of tables and value correspondences for selected classes.

Class	Tables	V_{Matched}	$V_{\text{Unmatched}}$
GF-Player	10,432	206,847	35,968
Song	58,594	1,315,381	443,194
Settlement	11,757	82,816	13,735

For the three evaluated classes, Table 4 shows the result of matching the table corpus to existing instances and properties in DBpedia, using a method from previous work [26, 27]. The first column shows the number of matched tables that have at least one matched attribute column. Rows of those tables were matched directly to existing instances of DBpedia. From the second and third columns we see how many values were matched to existing instances and how many values remained unmatched. While more values were matched, the number of unmatched values is still large, especially for the song class.

2.3 Gold Standard

For the purpose of this work we built a publicly available gold standard of annotated web tables. We first annotated clusters of rows that describe the same instance. Additionally, we annotated if these clusters describe new instances and for clusters that overlap with existing instances in DBpedia, we also annotated the clusters with the correspondence to the existing instance. We annotated attribute-to-property correspondences, where table columns are mapped to properties in DBpedia. Finally, we annotate facts for all cluster and property combinations for which a candidate value exist in the annotated web tables.

The gold standard contains tables with instances of varying degree of popularity, so that they describe head and long tail instances. We also prioritized tables with rows that are unlikely to have a match in DBpedia and ensured that for some labels, we select at least five rows to be able to form large enough clusters.

Table 5 provides an overview of the annotations per class. In the first three columns we see the number of table, attribute and row annotations. On average, we have 1.85 attribute annotations per table, not counting the label attribute. The two following columns show the number of annotated clusters, followed by the number of values within those clusters that match a knowledge base property. The second to last column shows the number of overall value groups, i.e. the number of cluster and property combinations for which at least one candidate value exists. For all groups we included facts, i.e. the correct value given the group's cluster and property. The last column shows for how many of the groups, the correct value is contained among the candidate values. We annotated 271 clusters, of which 39 % are new. On average, each cluster has approximately 3.42 rows, 7.69 values, 3.17 value groups and their facts, and 2.88 groups where the correct value is present in the web tables.

We use the gold standard for learning and testing. For this, we split the data into three folds and performed cross-validation. We ensured that we evenly split new clusters and homonym groups, which are groups of clusters with highly similar labels. All clusters of a homonym group were always placed in one fold.

The gold standard, along with the code and other data, is publicly available.[2] The results of this work are therefore replicable.

3 METHODOLOGY

In this section we present our system and evaluate alternative approaches for the individual components of the pipeline. As shown in Figure 1, the pipeline begins with the web tables and ends with entities being added to the knowledge base as new instances. In between, the pipeline consists of four components: schema matching, row clustering, entity creation and new detection.

We iterate over the pipeline twice. During the second run, we utilize the output of the row clustering and the new detection to generate a refined schema mapping. The attribute-to-property correspondences derived by the schema matching are important, because they allow us to extract for a row a set of values which correspond to the schema of the knowledge base. These values are utilized by the row clustering and new detection components with a positive impact on performance. More importantly, these values are required to create descriptions for new instances.

During the schema matching phase, we also match each table to a class in the knowledge base. Afterwards we run the remainder of the pipeline for each class separately.

3.1 Schema Matching

The first step in the pipeline is to create a mapping between the schemata of the individual web tables and the schema of the knowledge base. As the web tables have heterogeneous schemata, this task is non-trivial. Overall there are four steps necessary: (1) data type detection, (2) label attribute detection, (3) table-to-class matching and (4) attribute-to-property matching.

Data Type Detection

Throughout our pipeline we utilize a number of data types to type individual values, facts, attribute columns or knowledge base properties. Each type has a corresponding similarity function, and an equivalence threshold, which is used to determine if the compared values are equal. We employ overall six data types:

- **Text:** string, where two strings do not have to be exactly equal to be similar, e.g. label of an instance.
- **Nominal String:** string, where two strings are either completely equal or unequal, e.g. ISO code of a country.
- **Instance Reference:** reference to an instance, e.g. team of an athlete or musical artist of a song.
- **Date:** date with two possible granularities: year or specific day, e.g. release date of song, or birth date of a person.
- **Quantity:** numeric quantity, where numeric closeness has a semantic relevance, e.g. population of a settlement.
- **Nominal Integer:** integer, where numbers close to each other are not semantically related. This include e.g. numbers or draft rounds of athletes. Typing nominal integers in addition to nominal strings allows some components, especially the attribute-to-property matcher, to use methods tailored for this type.

We run a data-type detection algorithm [26] that assigns to each table attribute one of the following types: text, date and quantity. Detecting the other three types requires an understanding of the

[2] http://data.dws.informatik.uni-mannheim.de/expansion/LTEE/

Table 5: Overview of the gold standard.

Class	Tables	Attributes	Rows	Existing Clusters	New Clusters	Matched Values	Value Groups	Correct Value Present
GF-Player	192	572	358	81	19	1,207	475	444
Song	152	248	193	34	63	425	231	212
Settlement	188	162	376	49	25	451	152	124

actual semantics of an attribute, so that they are assigned by the attribute-to-property matcher, after an attribute has successfully been matched to a knowledge base property.

The data type detection is performed using manually defined regular expressions. We decide the data type of an attribute based on the majority data type among its values.

Label Attribute Detection

For each table we assign one column as the label attribute, which contains natural language labels for the entities described in the table rows [26]. For this we find the column with the data type text and the highest number of unique values. In case there is a tie between multiple columns, we choose the column that is furthest to the left [26].

Table-to-Class Matching

We utilize an approach that performs both row-to-instance and attribute-to-property matching to find the class of a table.

We first extract from the label attribute a label for each row, and use the label to find candidate instances from the knowledge base. A class, for which many rows of a table have a candidate instance, is chosen as a possible candidate class of that table. We assign the number of rows with a match as a score to that class.

Given these candidate classes, we then evaluate how well their properties match. We compare the values in the rows with facts of their candidate instance in the knowledge base, to find if they match a certain property of the candidate class. We block comparisons based on data type as detected above. Using the matched values we are able to perform duplicate-based attribute-to-property matching [5], where we chose the property with the highest number of matched cells as the property of the attribute, and assign the number as the score of the correspondence.

Per candidate class, we aggregate all scores to compute a ranked list of candidate classes. We choose the class with the highest score as the class of the table. This approach was proposed and evaluated by Ritze et al., where authors find that it can achieve an F1 score of 0.97 on a web table corpus [26].

Attribute-to-Property Matching

Our attribute-to-property matching approach consists of three steps. We first select candidate properties from the knowledge base schema based on data types. For text attributes, we choose all properties with types instance reference, nominal string and text, for quantity attributes we choose properties with types quantity and nominal integer and finally for date attributes, we choose properties with types date, quantity and nominal integer as candidates. After matching, the data type of the attribute is changed to the data type of the matched property and the values are accordingly normalized.

Secondly, we use various matchers, described further below, to compute matching scores. Given a candidate knowledge base property, a matcher finds a score from 0.0 to 1.0 that measures the likelihood that the attribute matches the property. Scores of multiple matchers are then aggregated based on a weighted average, where weights are learned for each class individually.

Table 6: Attribute-to-property matching performance by iteration.

Iteration	P	R	F1
First	0.929	0.608	0.735
Second	0.924	0.916	0.920
Third	0.929	0.916	0.922

We then utilize thresholds on the aggregated scores to determine if a certain candidate property matches an attribute. The thresholds are learned per property of the knowledge base schema. An attribute is matched to a property if it is both, a property that achieves a score above the property-specific threshold, and the property with the highest aggregated score.

Overall we implement five matchers, three of which exploit the knowledge base. `KB-Overlap` computes the proportion of values in the attribute that generally fit the candidate property in the knowledge base. `KB-Label` compares the label in the attribute header row to the labels of the candidate property in the knowledge base. `KB-Duplicate` computes the proportion of values in the attribute that is equal to the fact of the candidate property in the knowledge base, based on the instance correspondences generated by the new detection component.

We further implement two matchers that exploit the large web table corpus. For this, we first match attributes using the above described matchers for a preliminary mapping. We then rerun the matching using two additional matchers that exploit the preliminary mapping and the web table corpus. `WT-Label` utilizes the column headers of columns matched in the preliminary run, to derive label-to-property scores, where the score represents the likelihood that an attribute with a certain header row label corresponds to a certain candidate property of the knowledge base. `WT-Duplicate` knows through a previous row clustering run which rows in the tables describe the same instances. Using the preliminary mappings, we can find values in the corpus that are matched to same instance and property. This matcher measures and returns the proportion of values in an attribute, for which an equal value matched to same instance exists.

Table 6 shows by iteration the performance of an attribute-to-property matching method that aggregates all matchers. The duplicate-based methods are not included in the first iteration, as they require output from the other pipeline components. We evaluated the methods on the attribute annotations in the gold standard. We split the annotations first into a learning and a testing set, where the learning set contains two third of annotations.

From the table we can see that a second iteration and the utilization of the output of the row clustering and the new detection components have a large positive effect on schema matching performance. The table also shows that a third iteration has only a marginal positive effect, so that two iterations suffice.

To determine the usefulness of each individual matcher, we evaluate the weights assigned in the aggregated method of the second iteration. As we learn weights per class, the following weights are averages. The duplicate-based matchers have a combined weight of 0.43, where the `KB-Duplicate` matcher with a weight of 0.25 is more important. The label-based matchers achieve a higher combined weight of 0.46 where the `WT-Label` with a weight of 0.25 is very effective. Finally the `KB-Overlap` method is the least important method with a weight of 0.10. Additionally, the distribution of weights for the individual classes were similar to the here mentioned averages.

From the weights we can first of all see, that the attribute label is quite an effective method for schema matching. More importantly, we can conclude, that the most effective approach is one that combines various matchers and thereby exploits the highest number of individual signals for schema matching.

3.2 Row Clustering

After matching tables to classes and table attributes to properties of the knowledge base, we cluster rows that describe the same instance together. This step is especially important, as it reveals the overall number of unique instances described in the tables.

Our row clustering methods consist of a row similarity metric, which measures the likelihood that two rows describe the same instance, and a clustering algorithm, that utilizes the similarity metric to create clusters of rows.

In earlier work [25–27], we determined which rows describe the same instance by matching them to existing instances in the knowledge base. As a result, our earlier methods were unable to cluster rows of new instances, unlike in this work, where we perform clustering independently from existing instances.

Clustering Algorithm

In the context of this work, a required feature of the clustering algorithm is its ability to determine the number of clusters. In case of a perfect clustering, this number would correspond to the exact number of instances described by the rows of all tables.

Correlation clustering approaches [1, 2, 6, 10] fulfill this requirement. Clustering here is viewed as an optimization problem that aims to find the optimal partitioning of a set of vertices by maximizing a fitness function that aggregates similarities within a partition and dissimilarities between partitions.

Due to the large number of rows that need to be clustered (see Table 11), we need clustering methods that scale. As correlation clustering approaches try to find the globally optimum solution, they do not scale for the task at hand. We therefore utilize a greedy correlation clustering algorithm [15, 16], which solves the optimization problem locally for each decision made by the clusterer. The algorithm employs a row similarity function with a normalized output from −1.0 to 1.0 and sequentially tries to assign each row to the optimum cluster by summing the similarity scores of the current row with all individual rows in already created clusters in order to compute aggregated row-to-cluster scores. If there are scores larger than zero, the row is assigned to the cluster with the highest score. If no score is larger than zero, a new cluster is created, and the row is assigned to it. While every row assignment or cluster creation would maximize the fitness function locally, this does not ensure global optimization.

To achieve further scalability, we perform the row assignment in parallel instead of sequentially. While this is much faster, it can results in errors during clustering. We therefore run the Kernighan-Lin with joins (KLj) [19] clustering algorithm as a secondary step. The algorithm improves an existing preliminary clustering, in our case the output of the parallelized greedy clustering, by comparing cluster pairs and attempting to move individual rows between those clusters or merging the clusters fully. Similarly, each cluster is compared with an empty set to find whether splitting a cluster increases the fitness function locally. The operations are repeated until no further operation is able to increase the local fitness function.

As a result, we are able to quickly build a preliminary clustering with complete parallelization, while with a second step, we ensure that clustering quality is still high. Scalability is further ensured by the blocking approach described below.

Row Similarity Metrics

A row similarity metric compares two rows and returns a score that measures the likelihood that the two rows describe the same instance. Depending on the exact implementation, other input, e.g. from the knowledge base or the web table corpus, might be utilized. We implement six different similarity metrics:

- **`LABEL`:** We use the label attribute of a table to derive labels for all rows to then derive a similarity score by comparing labels using the Monge-Elkan similarity with Levenshtein as the inner similarity function.
- **`BOW`:** For each row we create a bag-of-words binary term vector that contains the terms that occur in all cells of a row. For this, cell values are cleaned, normalized and tokenized. To compare two rows, we compute the cosine similarity of their vectors.
- **`PHI`:** This approach allows us to compare two rows by comparing their tables. It derives a similarity between two tables using the PHI correlation of row labels, which we first compute using the following formula:

$$PHI(x,y) = \frac{n \times n_{xy} - n_x \times n_y}{\sqrt{n_x \times n_y \times (n - n_x) \times (n - n_y)}},$$

$$\begin{aligned} \text{where } n &= \text{total number of unique labels,} \\ n_{ab} &= \text{occurrence of labels a and b in same table,} \\ n_a &= \text{occurrence of label a in a table.} \end{aligned}$$

  For each label we therefore have a vector that measures its correlation with all other labels in the corpus. We then create such a vector for each table, by averaging the vectors of the table's row labels. With this we attempt to derive a vector that captures semantic information about all rows described in the table. When comparing two rows, we return the cosine similarity of the vectors of their tables.
- **`ATTRIBUTE`:** Using the attribute-to-property correspondences we can derive for each row values matched to the knowledge base schema. This allows us to perform value normalization and apply data-type-specific similarity functions to compare row values. If the two rows being compared have overlapping value pairs, so that both values are matched to same property, we use the data type similarity function to determine if those values are equal, assigning them a score of either 1.0 or 0.0. As there are possibly more than one overlapping value pair, the similarity returned equals to the average similarity scores of all pairs. Additionally, a confidence score is attached that

equals the number of pairs compared. This confidence score is used by the aggregation methods described below.

- **IMPLICIT_ATT:** Many tables have rows that describe instances that are similar, e.g. cities in Germany or athletes drafted in 2010. This information is not stated explicitly in any of the row cells. Using the following approach, we attempt to derive for a table implicit property-value combinations that apply to all instances described by the table. We can then use these implicit property-value combinations to compare rows with each other.
  We first use the row labels to find candidate instances for all rows, and then for each row all property-value combinations that exist for at least one candidate in the knowledge base. For each property-value combination we then derive a score for the whole table, which equals the proportion of rows that have this combination. We keep only combinations with a score above a certain threshold. The remaining combinations are the implicit attributes of a table and their score is assigned as a confidence score.
  Given two rows we compare the implicit attributes of one row with overlapping implicit attributes and column attributes of the other row and vice versa. We return the average of the similarities of all compared pairs and the sum of the implicit attribute scores as the confidence.
- **SAME_TABLE:** This metric builds on the observation that rows in a single table usually describe different entities. The metric assigns two rows of the same table a similarity of 0.0, otherwise 1.0.

Similarity Score Aggregation

We implement two approaches to aggregate the row similarity scores. We first utilize a weighted average, where the weights assigned to each metric are learned. In this case, attached confidence scores are not considered. We also learn a threshold, where scores above the threshold indicate that the rows describe the same instance. This threshold is used to normalize the similarity metric to −1.0 and 1.0. To learn the weights, we model the data in the learning set as row-pairs that either match or not, i.e. with scores of either 1.0 or 0.0. When learning weights we utilize a genetic algorithm that attempts to maximize the matching performance on the learning set.

As a second, alternative aggregation approach, we use random forest regression tree [7], where as features we include both similarity and confidence scores. We again model the data as row-pairs, where non-matching row-pairs are assigned a score of −1.0, while matching pairs a score of 1.0. To learn the random forest regression tree we utilize the WEKA library. We learn the hyperparameters of the algorithm by using the out-of-bag error with different out-of-bag rates on the learning set.

As a third aggregation approach, we combine both aggregation methods using a weighted average, where the weights are also learned as described above. In all cases we upsample to balance the number of matching and non-matching row pairs.

Blocking

To ensure the scalability of the row clustering for large web table corpora, we implement a blocking algorithm. We block comparisons by first limiting the number of clusters a row is compared to during the parallel greedy clustering, and, secondly, limiting the cluster pairs that are compared with each other during the KLj clustering.

Table 7: Average clustering performance and metric importance scores for alternative row clustering methods.

Run	PCP	AR	F1	MI
LABEL	0.71	0.83	0.76	0.33
+ BOW	0.73	0.84	0.78	0.18
+ PHI	0.74	0.84	0.78	0.05
+ ATTRIBUTE	0.75	0.85	0.80	0.21
+ IMPLICIT_ATT	0.78	0.87	0.82	0.17
+ SAME_TABLE	0.79	0.87	0.83	0.07

We utilize the row labels for the blocking mechanism. We first normalize the labels of all rows and use them to build a Lucene index. Each label in the index forms a block, which includes all rows with that exact label. For each row we use the index to retrieve a number of labels similar to the row's label, and assign their blocks to the row.

During the parallelized greedy clustering, we compare a row only to clusters with which the row shares a block. The blocks of a cluster are the union of the blocks of all rows in that cluster. Similarly, during the KLj clustering, two clusters are only compared when they share a block.

Evaluation

To evaluate the performance of the row clustering, we employ the evaluation approach proposed by Hassanzadeh et al. [17]. We use the set of clusters annotated in the gold standard, denoted as G, and the set of clusters returned by our method, denoted as C, to first compute a one-to-one mapping between the clusters in G and the clusters in C. We map a cluster in C to a cluster in G, if it contains the highest fraction of rows that are from that cluster in G. In case two clusters in C have the same proportion, we take the cluster with the highest absolute number of overlapping rows. We denote the mapping as M.

Using this mapping we compute average recall, penalized clustering precision and their F1 score. Average recall is the average of the individual recalls of the clusters in G. The recall of a cluster in G is equal to the ratio of rows in the mapped cluster from C that are also in G to the number of total rows in G. If no cluster from C was mapped to a cluster in G, the recall of that cluster is zero.

To compute the clustering precision we compute the precision of all pairs of rows that are part of the same cluster in C. A pair is determined to be correct if both rows are part of the same cluster in G. Unlike the average recall, this does not measure how well we find cluster, but how well we place rows in the same cluster.

As finding the correct number of unique instances described in the web tables is important, finding the correct number of clusters is important. We therefore penalize the clustering precision, as is also suggested by [17], if the number of returned clusters deviates from the correct number of clusters. We penalize by multiplying the clustering precision by a penalizing factor. This factor is computed by finding the sizes of C, G or M and dividing lowest by the highest size. We take this penalized clustering precision as the main precision-based score to evaluate our clustering.

Results and Lessons Learned

We will first evaluate the effectiveness of the individual row similarity metrics, and afterwards take a look at the effect of different aggregation methods as well as the blocking.

Table 7 shows the average performance of various row clustering methods using the third aggregation method, which combines random forest and weighted average. The first row contains the results when using only the LABEL metric. For every following row we aggregate one additional similarity metric. The last column of the table shows the metric importance, which is the average of the relative importance of the metric inside the learned random forest regression tree and the weights in the learned weighted average function. The importance scores shown are derived for the method that aggregates all metrics, i.e. the one that corresponds to the last row of the table.

The table shows that the similarity of row labels is the best indicator if two rows describe the same instance, as it has the highest average metric importance of 0.33 and with it we are able to achieve a moderate F1 of 0.76. At the same time, it alone is not enough and all other similarity metrics positively impact the row clustering performance when aggregated. This applies especially to the metrics BOW, ATTRIBUTE and IMPLICIT_ATT, which increase the F1 score by 2 percentage points each.

Both PHI and SAME_TABLE have smaller effects, as both only increase precision by 1 percentage point without an effect on recall. PHI likely achieves a low impact because it does not measure the similarity of two rows directly, but rather compares their tables. On the other hand, the same applies to IMPLICIT_ATT and it has a significantly higher impact. This shows the likely benefit of utilizing the knowledge base as background knowledge.

From the overall results, we can conclude that the best approach is to aggregate multiple metrics, thereby combining the different signals exploited by the individual metrics. The LABEL method utilizes the output of the label attribute detection, while the ATTRIBUTE method utilizes the knowledge base as background knowledge to semantically understand the attributes of the table. The IMPLICIT_ATT also exploits a knowledge base, but to assign semantic property-value combinations to its rows. Finally, the BOW method includes all information of a row, potentially covering information that couldn't be mapped to a schema.

The last row in Table 7 shows the clustering method that aggregates all metrics using a combination of random forest and weighted average. Applying both aggregation methods separately would have achieved an F1 score of 0.81 for weighted average and 0.82 for random forest.

Finally, the blocking yields no decrease in F1, which shows that it is an effective approach with minimal loss in recall.

3.3 Entity Creation

The entity creation component receives clusters of rows and transforms each cluster into an entity. First, an entity consists of one or more labels, which we extract from the label attribute of the entity's rows. More importantly, an entity contains a set of values mapped to the properties of the knowledge base. Given that at the row-level, each row can have multiple values matched to certain knowledge base properties, and that we have multiple rows in a cluster, there are likely to be multiple candidate values for one property when creating an entity. We therefore apply the following four-step method to fuse candidate values:

(1) **Scoring:** We score candidate values using one of the three alternative approaches described below.
(2) **Grouping:** We group equal values together. This is done using the data type specific similarity functions.
(3) **Selection:** We then select the group with the highest sum of individual candidate value scores.
(4) **Fusion:** We fuse a group into a fused value by using data type specific fusers. For text and instance reference types we utilize the majority value in a group, whereas for quantity and date types we use a weighted median approach. For nominal string and nominal integer, no fusion is necessary, as all values in a group will be equal.

We test three different scoring approaches. VOTING assigns all candidate values equal scores of 1.0. In the KBT [13] approach we measure for a certain table attribute the correctness of its overlapping values, i.e. those matched to an existing fact in the knowledge base, to estimate the trustworthiness of the whole attribute. Finally, the MATCHING approach utilizes the scores attached to a column by the attribute-to-property component. We measure the effect of the scoring approach using the output at the end of the pipeline. The results are discussed in Section 4.2.

The methods presented here are similar to those in our earlier works [22, 23, 27]. In this work, we additionally apply scores derived from the schema matching component for fusion.

3.4 New Detection

After creating entities from row clusters, we now determine whether a created entity describes a new instance, not yet present in the knowledge base. This is done by attempting to match the entities to existing instances by exploiting various features through entity-to-instance similarity metrics. If there are no instances found or the distance between an entity and an instance is large enough, the entity is determined to be new.

Additionally, for entities not classified as new, we attempt to match them to an existing instance in the knowledge base. These correspondences to existing instances are fed back into a second iteration of the pipeline to refine the schema mapping.

Our new detection approach consists of three steps:

Candidate Selection: We find a list of candidate instances from the knowledge base using a Lucene index built from the labels of knowledge base instances. To search candidates for an entity we utilize the labels attached to the entity in the entity creation component. Additionally candidates found must be of the class of the created entity or share one parent class.

Similarity Score Computation: We compute a score to measure the similarity between the created entity and a candidate instance. Multiple entity-to-instance similarity metrics are implemented and tested below.

Classification: If the highest similarity for any candidate instance is lower than a learned threshold, we classify the entity as new. Otherwise we find the candidate instance with the highest similarity score, and in case its score is higher than another threshold, the entity is classified as existing and a correspondence from the entity to that instance is generated.

In earlier work we presented methods that are solely concerned with matching rows to existing instances of a knowledge base, whereas with the methods presented here, we also determine whether rows have a match at all. Additionally, we do not match rows directly, but first create entities from row clusters, which allows us to exploit more information for matching.

Entity-To-Instance Similarity Metrics

As described above, the following metrics compute a similarity score between a created entity, and a one candidate instance of the knowledge base. We implement overall six different metrics:

- **LABEL:** We compute the similarity between the labels of the created entity and the labels of the candidate instance using Monge-Elkan with Levenshtein as the inner similarity function.

- **TYPE:** In DBpedia every class is part of a hierarchy with a certain number of parent classes. We compute the overlap of the classes of the candidate instance with the class of the entity and its parent classes.
- **BOW:** We create a bag-of-words binary term vector for the entity by combining the vectors of all its rows, which themselves are created as described in the row clustering approach. We then create a vector for the candidate instance in the knowledge base, using its labels, abstract and facts. We return the cosine similarity of both vectors.
- **ATTRIBUTE:** For each property, where a fact exists in both the created entity and the candidate instance in the knowledge base, we determine if the fused fact is equal to the fact in the knowledge base. As there could be multiple overlapping properties, we return an average similarity and a corresponding confidence score, which equals the number of overlapping properties.
- **IMPLICIT_ATT:** We utilize the implicit attributes derived for tables, as described in Section 3.2, to derive implicit attributes for a created entity. We sum up the confidence scores of equal implicit attributes for the tables of all rows in the entity and divide by the total number of rows to compute an entity-level confidence score. We then compare these property-value combinations at the entity level with overlapping facts of a candidate instance.
- **POPULARITY:** We use a dataset of Wikipedia page links to rank all candidate instances of an entity by their number of incoming page links. A similarity score is assigned to each candidate based on its rank. If an entity has only one candidate instance, we assign it a score of 1.0.

Similarity Score Aggregation

We aggregate various similarity scores using the same aggregation approaches utilized for row clustering.

Evaluation

We evaluate the new detection component using the clusters annotated in the gold standard. Before we run new detection on those clusters, we create entities from them as outlined in Section 3.3 above. From the gold standard we know whether a certain created entity describes a new instance or not. In case it describes an existing instance, we additionally know from the gold standard the exact instance it describes.

When running the new detection component on those entities, we receive a set of entities classified as new, and another set classified as existing with additional correspondences to existing instances in the knowledge base. We can now use the gold standard to determine the accuracy of those classifications, which equals the fraction of correctly classified entities. Existing entities must additionally be matched to the correct instance in the knowledge base to be counted as correctly classified.

As the accuracy measures the classification performance for both the new and existing instances, we additionally evaluate both separately using F1. The precision of the new entities equals the fraction of entities returned with a new classification that were correctly classified as new, whereas recall is equal to the fraction of total new entities in the gold standard that were correctly classified as new. The same applies to the existing entities, with a second condition that the entity must be matched to the correct instance in the knowledge base as well.

Table 8: Average performance and metric importance scores for alternative new detection methods.

Run	ACC	$F1_{\text{Existing}}$	$F1_{\text{New}}$	MI
LABEL	0.69	0.66	0.67	0.20
+ TYPE	0.79	0.75	0.82	0.26
+ BOW	0.85	0.84	0.83	0.17
+ ATTRIBUTE	0.85	0.86	0.84	0.20
+ IMPLICIT_ATT	0.88	0.87	0.89	0.11
+ POPULARITY	0.89	0.88	0.88	0.06

Results and Lessons Learned

Table 8 shows the performance of various new detection methods. All numbers in table are averages of all classes and folds. The first row shows a method that only utilizes the label. For each following row we aggregate an additional similarity metric into the method using the combined aggregation approach. The metric importance shown in the last column reflects a score derived from the random forest and the weights in a method that aggregates all metrics, i.e., the method described in the last row.

From the table we can see that with an accuracy of 0.89, the final aggregated method performs quite well. It achieves a performance considerably better than the LABEL method, which only has an accuracy of 0.69. Additionally we can see that all similarity metrics are important and contribute positively to the overall performance. This shows that the combined approach is able to leverage the different features exploited by the individual metrics. The LABEL metric does however have a high importance score, even though the candidate selection already only returns candidates with similar labels.

The later a metric is aggregated, the more difficult it is to yield a large absolute increase in performance. As such, the metrics TYPE and BOW increase accuracy by 10 and 6 percentage points respectively, which is much larger than for metrics added later. At the same time we see, that ATTRIBUTE, which has a higher importance score than BOW, does not increase accuracy at all. It does however increase the individual F1 scores.

Noticeable is also the increase achieved by the IMPLICIT_ATT metric. It is able to improve the accuracy by further 3 percentage points, even though it is the second to last metric to be aggregated. This means, that we are successfully able to leverage the knowledge base as background knowledge to derive semantically relevant property-value combinations per table and entity.

The POPULARITY metric only impacts performance for the Settlement class. This means that, given just the name of a settlement, it is safe to assume that the most well-known settlement is meant. This makes sense, as this assumption is typically made when speaking about cities in general.

The last row shows the performance of a new detection method that aggregates all metrics using a combined approach of weighted average and random forest. When using the aggregation methods separately, we achieve an accuracy of 0.85 and 0.86 respectively. The combined approach is therefore able to exploit both aggregation methods to achieve a higher performance.

4 OVERALL RESULTS ON THE GOLD STANDARD

In this section, we choose the best methods for clustering and new detection, and evaluate the output of a complete run of our system using our gold standard. We will first evaluate how

Table 9: Results of new instances found evaluation.

Class	Clust.	New Det.	P	R	F1
GF-Player	GS	ALL	0.89	0.95	0.91
GF-Player	ALL	ALL	0.82	0.95	0.87
Song	GS	ALL	0.92	0.88	0.90
Song	ALL	ALL	0.72	0.72	0.72
Settlement	GS	ALL	0.84	0.90	0.87
Settlement	ALL	ALL	0.74	0.87	0.80
Average	ALL	ALL	0.76	0.85	0.80

many of the new instances were correctly found, while in the second part, we will evaluate how many correct facts were found. Throughout this evaluation we utilize three fold cross-validation.

4.1 New Instances Found Evaluation

To evaluate how well new instances were found, we utilize precision and recall. First, we determine the number of new instances annotated in the testing sets, for which an entity was correctly returned by the system. For this, three conditions must be met. First, a majority of the rows of an entity must correspond to the same new instance in the gold standard, while at the same time the entity must also contain the majority of the rows that actually describe that instance. Lastly, the entity must be classified as new by the new detection component. Based on this, recall is defined as the fraction of new instances in the gold standard for which a correct entity was returned. Precision, on the other hand, is the fraction of entities returned by the system as new, that correctly match an instance in the gold standard.

Table 9 shows the performance of our system for the three classes separately. To evaluate the individual impact of row clustering and new detection, we once evaluate using the clustering from the gold standard, denoted GS, and once with the aggregated clustering method containing all similarity metrics, denoted ALL. For new detection we run in both cases the aggregated method containing all similarity metrics, also denoted ALL.

Generally, we achieve good performance for football players and settlements, while for songs the performance is less convincing. This is likely, because for songs, the homonym problem is much larger. It is much more likely that there exist songs of the same name by various artists. Sometimes these homonyms even represent cover versions, so that they are highly similar in their descriptions, e.g. in runtime or writer.

When investigating how much performance is lost by the individual components, we find that for football players and settlements the new detection component is causing a larger decrease than the row clustering. More specifically, the drop is mainly caused by classifying entities as new even though they should be matched to existing instances. This is confirmed by the recall being higher than precision. For songs, the clustering causes a much larger decrease in performance, showing that the clustering task is more difficult for songs, and that we require more sophisticated clustering methods.

4.2 Facts Found Evaluation

In this section we evaluate how well we can generate facts from web tables for new entities. Again, we need a mapping between entities returned and instances in the gold standard, for which we utilize the approach described above. For wrongly created entities, or entities incorrectly determined to be new, i.e. they could not be mapped to a new cluster in the gold standard, their facts are counted as wrong and therefore reduce precision. To determine if returned facts for matched entities are correct, they are compared to the facts in the gold standard using data type specific similarity functions and a learned tolerance range. To measure recall, we utilize the number of annotated facts for which a correct value is present in the web tables, as seen in Table 5.

Table 10: Results of the facts found evaluation.

		New	F1	F1	F1
Class	Clust.	Det.	`VOTING`	`KBT`	`MATCHING`
GF-Player	GS	GS	0.82	0.82	0.82
GF-Player	GS	ALL	0.81	0.81	0.81
GF-Player	ALL	ALL	0.81	0.81	0.81
Song	GS	GS	0.80	0.81	0.81
Song	GS	ALL	0.74	0.73	0.74
Song	ALL	ALL	0.67	0.69	0.68
Settlement	GS	GS	0.98	0.98	0.98
Settlement	GS	ALL	0.93	0.93	0.93
Settlement	ALL	ALL	0.91	0.91	0.91
Average	ALL	ALL	0.80	0.80	0.80

Table 10 shows fusion performance per class. In order to measure the individual impact of the new detection and the row clustering on the overall performance, we again perform multiple runs. For the first run, we utilize for both components the correct annotations from the gold standard. In the following run we use our new detection methods, while for the third run we additionally use our clustering methods, in both cases using the methods that aggregate all similarity metrics. Additionally we test performance for the different fusion scoring methods, `VOTING`, `KBT` and `MATCHING`, described in Section 3.3.

From the table we can see that for football players and songs we lose 18 and 19 percentage points of F1 score even when row clustering and new detection are perfect. We looked at a sample of errors and were able to identify two main causes. The largest amount of errors is due to the attribute-to-property matching component, where the proportion of errors caused by wrong or missing column matches makes up 43 % of all errors. This is followed by 35 % of errors that occurred as a result of wrong or outdated data in the tables.

In addition, we evaluate various fusion scoring approaches. We find that all performances are very close, so that the choice of scoring approach is of low relevance.

Finally, we can deduce from the table the individual impacts of the pipeline components on the performance. The largest negative impact is due to errors in finding facts as described above, of which the attribute-to-property matching is a large contributor. Not as large are the individual impacts of the row clustering and new detection components, but they are still significant ranging from 1 to 13 percentage points. Overall, the way errors compound throughout the pipeline shows the difficult nature of the task at hand. Every individual component has to perform very well for there to be good performance at the end of the pipeline.

5 LARGE-SCALE PROFILING

In this section we try to estimate the potential of web tables for extending knowledge bases with new instances, by running our system on not only the tables of the gold standard, but on all tables of a specific class within the whole corpus (see Table 4). We are especially interested in how many new instances we can correctly add per class and how that compares to the number of existing instances in the knowledge base. We additionally are interested in the descriptions of these new instances, the number and accuracy of facts, and property densities of the new instances.

Evaluation

From the entities returned as new by the system, we pull a stratified sample of 50 entities for each class. We group the returned entities by the number of facts generated for each entity. We then pull from each group a number of entities proportional to the size of the group in relation to the total number of new entities.

The accuracy of new entities equals the fraction of entities that were correctly identified as new when compared to the 2014 release of DBpedia, while the accuracy of facts equals the proportion of facts within those new entities that are correct.

Results and Lessons Learned

Table 11 shows the results of the large-scale profiling per class. The second column lists the number of rows of all tables matched to the given class. The three columns afterwards describe existing entities found, to how many unique instances in the knowledge base they were matched, and the ratio of the two numbers. The remaining columns contain the number of new entities, their facts, the relative increases when compared to Table 1 and the accuracies of new entities and facts of the extracted sample.

First of all, we find that the ratio of existing entities to matched instances in the knowledge base differs by class. While for players and settlements the number is good, it is less so for songs. Song was the class with the worst performance at row clustering, i.e. identifying the exact number of unique instances. This shows, that we need to implement more sophisticated row clustering methods or, alternatively, perform deduplication after clustering.

For the class Song we have a very large number of new entities and facts, even if we would correct the number of new entities by the ratio in the fifth column. For Settlement, there are in comparison very few new entities. When considering that only 26 % of them are correct, we would actually achieve a relative increase in knowledge base instances of 0.3 %. The difference can be explained by understanding the notability rules of Wikipedia.

There are a very large number of obscure songs. It is very common, even for well-known artists, to release for example only a few songs from an album as singles, which are the only ones that become popular. And here the notability rules compound the issue, as songs only receive their own Wikipedia article if they are notable, e.g. because they were independently released.

For Settlement, almost the opposite is true. While there are many small villages, they are never irrelevant, as there are always enough people living in them, who might contribute to a Wikipedia article. More importantly, Wikipedia deems any place notable if it has legal recognition. As a result, Wikipedia covers a lot of settlements, and it is difficult to find new ones.

Football players are in the middle of both classes. There are not as many obscure football players, as, theoretically speaking, the number of teams is limited, but there are many that are obscure enough, not to be covered in a Wikipedia article. And while the absolute number of newly added players is not high, compared to the number of existing instances in the knowledge base, we achieve an increase of 67 % for instances and 32 % for facts.

Table 12 shows the property densities for new entities. As one would expect, the properties are not as dense as in Table 2. More importantly, the distribution of densities differs significantly. For football players, personal properties like birthDate and birthPlace have a very low density for new instances, but high for the knowledge base. This might be, because in Wikipedia one is interested in describing a person, whereas in web tables, the games, teams and drafts are more in focus. For those tables, a property like position might be more relevant, which explains why its density is even higher than in the knowledge base.

For songs, the properties writer, genre, and record label have very low densities compared to the knowledge base. It is likely that for genre, this is a column matching issue, as song genres are not always objectively defined. For writer and recordLabel there could be two causes. First, they might be uninteresting properties, and secondly, there are often multiple correct facts. The record label might even differ by country. This makes these properties difficult to match, and, more importantly, unlikely to be included in a table. We can confirm the latter, as these properties occurred very rarely in the tables we annotated for the gold standard.

When looking at the accuracies of new entities we also find differences per class. We achieve a moderate accuracy for songs, a sub-par accuracy for players and a low accuracy for settlements.

The primary reason for the low accuracy for settlements are conflicting values in an entity of an existing instance and the instance in the knowledge base. This includes outdated population numbers, but also isPartOf values, where the values in the entity and in the knowledge base are both correct, but different, preventing the entity from matching. This problem makes up 36 % of all errors. 25 % of errors are because the new entity does not describe a settlement, but a different place, like a region or a mountain. This error is caused by incorrect table-to-class matching. These problems are magnified because there are so few new entities to begin with, so that these corner cases make up a huge proportion of the new entities returned.

For GF-Player, the sources of errors include bad column-to-attribute matching, entities not being football players due to bad table-to-class matching, and incomplete information in DBpedia. The latter happened primarily when a football athlete was not assigned the correct class in DBpedia. The accuracy for entities with a higher number of values is however much higher. If we do not consider entities with one value, the accuracy of new entities rises to 0.72. If we further do not consider entities with two values, we achieve an accuracy of 0.85. This would mean excluding 6,360 entities, but also that with an accuracy of 0.85 we can add 7,623 entities with 34,922 facts to the knowledge base, an increase of 37 % for instances, and 25 % for facts.

For songs, the sources of errors are versatile. The main contributors are bad new detection, incorrect table-to-class matching, and bad clustering. The latter meaning that an entity was incorrectly detected as new, as a result of being created from rows that describe different instances.

We generally notice that the performance does not correlate with the performance on the gold standard. This might indicate that the gold standard either does not completely reflect the nature of the task, or the gold standard is not large enough. On the other hand, we achieve a consistently high accuracy for new facts, similar to the high performance on the gold standard, as seen in Section 4.2. This means that when it comes to finding descriptions, our performance is quite good, even if the density is lower when compared to the knowledge base.

Table 11: Results and evaluation of a system run on all tables matched to a class.

Class	Total Rows	Existing entities	Matched KB instances	Matching Ratio	New Entities	New Facts	N. Entities Accuracy	N. Facts Accuracy
GF-Player	648,741	30,074	24,889	1.21	13,983 (+67%)	43,800 (+32%)	0.60	0.95
Song	2,173,536	40,455	29,140	1.39	186,943 (+356%)	393,711 (+125%)	0.70	0.85
Settlement	1,472,865	28,628	27,365	1.05	5,764 (+1%)	7,043 (+0%)	0.26	0.94

Table 12: Property densities for new entities returned by the full run.

Class	Property	Facts	Density
GF-Player	position	9,204	65.82%
GF-Player	team	7,637	54.62%
GF-Player	college	6,849	48.98%
GF-Player	weight	5,915	42.30%
GF-Player	height	4,253	30.42%
GF-Player	number	2,951	21.10%
GF-Player	birthDate	2,537	18.14%
GF-Player	draftPick	2,404	17.19%
GF-Player	draftRound	1,538	11.00%
GF-Player	draftYear	386	2.76%
GF-Player	birthPlace	126	0.90%
Song	musicalArtist	143,656	76.84%
Song	runtime	115,652	61.86%
Song	album	52,664	28.17%
Song	releaseDate	47,377	25.34%
Song	genre	23,814	12.74%
Song	recordLabel	10,278	5.50%
Song	writer	270	0.14%
Settlement	isPartOf	2,889	50.12%
Settlement	postalCode	1,605	27.85%
Settlement	country	1,232	21.37%
Settlement	populationTotal	1,214	21.06%
Settlement	elevation	103	1.79%

Overall we find that for some classes there is high potential for finding new instances using web tables. Additionally, we also find that the performance of our system for these classes is generally good. Yet, it is clear that more sophisticated approaches are necessary for row clustering, new detection and table-to-class matching.

6 RELATED WORK

This section compares our method to the related work. As we investigate a new problem, we compare to research on similar tasks as well as on specific subtasks, including slot filling, set expansion, schema matching and identity resolution.

Slot Filling

Many works that exploit web table data for knowledge base augmentation [11, 25–29] focus on the task of slot filling, i.e. adding missing facts for existing instances. One prominent state of the art work by Dong et al. [11] introduces a probabilistic approach that exploits background knowledge, in their case Freebase, to construct a large knowledge base using web data, including web tables. The extracted facts however only describe instances that already exist in Freebase [11, 12].

While slot filling is a different task, we can still generally compare the numbers of generated facts. In a previous work, where we use web tables to perform slot filling for DBpedia [27], we are able to find 378, 892 facts, 64, 237 of which are new facts for existing instances. We reach an F1 score of 0.71. Compared to that work, we are able to achieve better numbers and accuracy. In the work by Dong et al., the authors are able to find 271 M facts for instances in Freebase with an expected correctness higher than 90 %. Of those facts, 90 M are new facts for existing instances. While the amount of facts that we discover are much smaller, we are only dealing with three classes and looking at facts only for new instances. Additionally we only use web tables to extract new facts, while Dong et al. also use free text, HTML DOM trees and schema.org annotations. Our average accuracy of 0.91 for facts of new entities is comparable.

Set Expansion

Set expansion is a task, where new instances are retrieved to complete a set [24, 31, 32]. Set expansion methods, however, only focus on finding the labels of new instances. The methods rely on seeds from that set to perform set expansion. Both, the complete sets and the number of seeds, are often small. In contrast, we focus on a scenario in which large sets of entities, already contained in the knowledge base, are extended with potentially large sets of new instances. Finally, most set expansion methods make use of ranked evaluation, where the precision of the top k instances is measured. In contrast, our evaluation not only focuses on precision, but also considers recall.

One set expansion method exploits a corpus of relational tables to augment an incomplete relational table with new instances and their descriptions [33]. The methodology differs significantly from ours. To find more instances, the method uses sets of 1 to 5 seed instances to first search for candidates in the tables. For this, it exploits the labels of the seeds and the caption of the seed table. Candidates are then ranked based on how often they co-occur with the seeds and how similar their tables' captions are. Similarly, the method searches for candidate columns in the corpus and ranks them based on how well they fit the seed table. The method then returns a fixed number of entities, where the authors use 256 as their cut-off. While this approach does generate descriptions, it does not resolve any of the following problems of set expansion. Their approach still ranks candidate entities based on their similarity to the seeds, and, more importantly, always returns a fixed number of instances. Especially the latter makes this approach not applicable to our task, as we are interested in generating as many new instances as possible.

To compare our work with works of set expansion, we need to utilize ranked evaluation and therefore need to implement a ranking algorithm for new entities. We rank based on the similarity scores returned by the new detection. These scores measure the distance between one or more existing instance in the knowledge base, and an entity generated from the web tables. We rank new entities higher, the higher their lowest distance to the closest existing instance is. Using this, we achieve a MAP, with a cut-off at 256, of 0.88, while related works achieve 0.63 [33], 0.95 [32] and 0.78 [31]. For precision at 5 and 20 we achieve 0.84 and 0.78 respectively, while a related work achieves 0.94 and 0.91 [33]. We find that our performance is comparable, even though the task we are solving is more difficult.

Schema Matching and Identity Resolution

Throughout the components of the pipeline, we apply approaches for which a large corpus of related work exists. This includes schema matching methods, which are surveyed in [3]. For row clustering and new detection we essentially exploit identity resolution methods, which are extensively surveyed in [9, 14].

For the specific use case of matching web table attributes to DBpedia properties, authors from our research group were able to achieve an F1 score of 0.81 [25]. While our performance of 0.92 is higher, we consider a smaller number of classes and properties.

There exists a large corpus of research on the task of matching web table rows to existing knowledge base instances. While this task is not a primary objective of this paper, we are able to evaluate our pipeline by comparing how well we are able to perform this task. We achieve an average F1 score of 0.83, compared to 0.80 [25] and 0.87 [34] in the related work, and we achieve an accuracy of 0.78, compared to 0.83 [21] and 0.93 [4] in the related work. While for F1, our performance is comparable to the related work, it is lower when it comes to accuracy.

7 CONCLUSION

This paper explored the potential of web table data for extending a cross-domain knowledge base with new long tail entities and their descriptions according to the schema of the knowledge base. To the best of our knowledge, this specific task has to this date not been handled in related research. For this task, we present and evaluate a complete system. It consists of a pipeline with multiple components, including schema matching, row clustering, entity creation and new detection.

We evaluated our pipeline using a manually annotated gold standard of web tables. We find that the task is non-trivial, as it requires good performance in all steps of the process. For all components of the pipeline we implemented and evaluated multiple alternative methods. We find that aggregating the similarity scores of multiple metrics that exploit different features yields the best results. We also find that metrics that make use of label similarity, while highly important, are not sufficient to yield a good performance. Additionally, we are able to show that metrics that use the knowledge base as background knowledge, e.g. to semantically understand cell values or to derive semantic information about web tables, have a positive impact on performance. Finally, we are able to utilize the output of the pipeline in a second iteration to achieve a large improvement in schema matching performance, while any further iteration has negligible impact.

We are successfully able to utilize our pipeline and our proposed implementations of the pipeline's components to find new instances and their descriptions from the web tables. At the same time, there remains room to further improve the quality of the generated data.

Finally we run the method on the complete web table corpus to profile the overall potential of web tables in augmenting a knowledge base class with new instances. We find that this potential differs per class, but at the same time, we find that for some classes a large number of instances and facts with a high accuracy can successfully be added to the knowledge base.

REFERENCES

[1] Nir Ailon, Moses Charikar, and Alantha Newman. 2008. Aggregating Inconsistent Information: Ranking and Clustering. *JACM* 55, 5 (2008), 23:1–23:27.

[2] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. 2004. Correlation Clustering. *Machine Learning* 56, 1 (2004), 89–113.

[3] Philip A. Bernstein, Jayant Madhavan, and Erhard Rahm. 2011. Generic Schema Matching, Ten Years Later. *VLDB* 4, 11 (2011), 695–701.

[4] Chandra Sekhar Bhagavatula, Thanapon Noraset, and Doug Downey. 2015. TabEL: Entity Linking in Web Tables. In *ISWC '15*.

[5] Alexander Bilke and Felix Naumann. 2005. Schema Matching using Duplicates. In *ICDE '05*. 69–80.

[6] Francesco Bonchi, David Garcia-Soriano, and Edo Liberty. 2014. Correlation Clustering: From Theory to Practice. In *KDD '14*.

[7] Leo Breiman. 2001. Random Forests. *Machine Learning* 45 (2001), 5–32.

[8] Michael J. Cafarella, Alon Y. Halevy, Yang Zhang, Daisy Zhe Wang, and Eugene Wu. 2008. Uncovering the Relational Web. In *WebDB '08*.

[9] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. 2015. *Entity Resolution in the Web of Data.* Morgan & Claypool Publishers.

[10] Erik D. Demaine, Dotan Emanuel, Amos Fiat, and Nicole Immorlica. 2006. Correlation clustering in general weighted graphs. *Theoretical Computer Science* 361, 2 (2006), 172 – 187.

[11] Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. 2014. Knowledge Vault: A Web-scale Approach to Probabilistic Knowledge Fusion. In *KDD '14*.

[12] Xin Luna Dong. 2016. How Far Are We from Collecting the Knowledge in the World. In *ICWE'16 Keynote*.

[13] Xin Luna Dong, Evgeniy Gabrilovich, Kevin Murphy, Van Dang, Wilko Horn, Camillo Lugaresi, Shaohua Sun, and Wei Zhang. 2015. Knowledge-based Trust: Estimating the Trustworthiness of Web Sources. *VLDB* 8, 9 (2015), 938–949.

[14] Xin Luna Dong and Divesh Srivastava. 2015. Record Linkage. In *Big Data Integration*. Morgan & Claypool Publishers.

[15] Micha Elsner and Eugene Charniak. 2008. You Talking to Me? A Corpus and Algorithm for Conversation Disentanglement. In *ACL-08: HLT*.

[16] Micha Elsner and Warren Schudy. 2009. Bounding and Comparing Methods for Correlation Clustering Beyond ILP. In *ILP '09*.

[17] Oktie Hassanzadeh, Fei Chiang, Hyun Chul Lee, and Renée J. Miller. 2009. Framework for Evaluating Clustering Algorithms in Duplicate Detection. *VLDB* 2, 1 (2009), 1282–1293.

[18] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. 2013. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artificial Intelligence* 194 (2013), 28–61.

[19] Margret Keuper, Evgeny Levinkov, Nicolas Bonneel, Guillaume Lavoue, Thomas Brox, and Bjorn Andres. 2015. Efficient Decomposition of Image and Mesh Graphs by Lifted Multicuts. In *ICCV '15*.

[20] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. 2015. DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195.

[21] Girija Limaye, Sunita Sarawagi, and Soumen Chakrabarti. 2010. Annotating and Searching Web Tables Using Entities, Types and Relationships. *VLDB* 3, 1-2 (2010), 1338–1347.

[22] Yaser Oulabi and Christian Bizer. 2017. Estimating missing temporal meta-information using Knowledge-Based-Trust. In *KDWeb '17*.

[23] Yaser Oulabi, Robert Meusel, and Christian Bizer. 2016. Fusing Time-dependent Web Table Data. In *WebDB '16*.

[24] Patrick Pantel, Eric Crestan, Arkady Borkovsky, Ana-Maria Popescu, and Vishnu Vyas. 2009. Web-scale Distributional Similarity and Entity Set Expansion. In *EMNLP '09*.

[25] Dominique Ritze and Christian Bizer. 2017. Matching Web Tables To DBpedia - A Feature Utility Study. In *EDBT '17*.

[26] Dominique Ritze, Oliver Lehmberg, and Christian Bizer. 2015. Matching HTML Tables to DBpedia. In *WIMS '15*.

[27] Dominique Ritze, Oliver Lehmberg, Yaser Oulabi, and Christian Bizer. 2016. Profiling the Potential of Web Tables for Augmenting Cross-domain Knowledge Bases. In *WWW '16*.

[28] Yoones A. Sekhavat, Francesco Di Paolo, Denilson Barbosa, and Paolo Merialdo. 2014. Knowledge Base Augmentation using Tabular Data. In *LDOW '14*.

[29] Mihai Surdeanu and Heng Ji. 2014. Overview of the English Slot Filling Track at the TAC 2014 Knowledge Base Population Evaluation. In *TAC '14*.

[30] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: A Free Collaborative Knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85.

[31] Chi Wang, Kaushik Chakrabarti, Yeye He, Kris Ganjam, Zhimin Chen, and Philip Bernstein. 2015. Concept Expansion Using Web Tables. In *WWW '15*.

[32] Richard C. Wang and William W. Cohen. 2007. Language-Independent Set Expansion of Named Entities Using the Web. In *ICDM '07*.

[33] Shuo Zhang and Krisztian Balog. 2017. EntiTables: Smart Assistance for Entity-Focused Tables. In *SIGIR '17*.

[34] Ziqi Zhang. 2017. Effective and Efficient Semantic Table Interpretation using TableMiner(+). *Semantic Web* 8, 6 (2017), 921–957.

Continuous Deployment of Machine Learning Pipelines

Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Tilmann Rabl, and Volker Markl

DFKI GmbH Technische Universität Berlin

{behrouz.derakhshan, alireza.rm, tilmann.rabl, volker.markl}@dfki.de

ABSTRACT

Today machine learning is entering many business and scientific applications. The life cycle of machine learning applications consists of data preprocessing for transforming the raw data into features, training a model using the features, and deploying the model for answering prediction queries. In order to guarantee accurate predictions, one has to continuously monitor and update the deployed model and pipeline. Current deployment platforms update the model using online learning methods. When online learning alone is not adequate to guarantee the prediction accuracy, some deployment platforms provide a mechanism for automatic or manual retraining of the model. While the online training is fast, the retraining of the model is time-consuming and adds extra overhead and complexity to the process of deployment.

We propose a novel continuous deployment approach for updating the deployed model using a combination of the incoming real-time data and the historical data. We utilize sampling techniques to include the historical data in the training process, thus eliminating the need for retraining the deployed model. We also offer online statistics computation and dynamic materialization of the preprocessed features, which further reduces the total training and data preprocessing time. In our experiments, we design and deploy two pipelines and models to process two real-world datasets. The experiments show that continuous deployment reduces the total training cost up to 15 times while providing the same level of quality when compared to the state-of-the-art deployment approaches.

1 INTRODUCTION

In machine learning applications, a pipeline, a series of complex data processing steps, processes a labeled training dataset and produces a machine learning model. The model then has to be deployed into a deployment platform where it answers prediction queries in real-time. To properly preprocess the prediction queries, typically the pipeline has to be deployed alongside the model.

A deployment platform must be robust, i.e., it should accommodate many different types of machine learning models and pipelines. Moreover, it has to be simple to tune. Finally, the platform must maintain the quality of the model by further training the deployed model when new training data becomes available.

Online deployment of machine learning models is one method for maintaining the quality of a deployed model. In the online deployment approach, the deployment platform utilizes online learning methods to further train the deployed model [13]. In online learning, the model is updated based on the incoming training data. Online learning adapts the model to the new training data and provides an up-to-date model. However, online learning is sensitive to noise and outliers which may result in an increase in the prediction error rate. Therefore, to guarantee a high level of quality, one has to tune the online learning method to the specific use case [22, 23]. Thus, effective online deployment of machine learning models cannot provide robustness and simplicity.

To solve the problem of degrading model quality, periodical deployment approach is utilized. In the periodical deployment approach, the platform, in addition to utilizing simple online learning, periodically retrains the deployed model using the historical data. One of the challenges in many real-world use cases is the size of the training datasets. Typically, training datasets are extremely large and require hours or days of data preprocessing and training to result in a new model. Despite this drawback, in some applications, retraining the model is still critical, as even a small increase in the quality of the deployed model can have a large impact. For example, in the domain of ads click-through rate (CTR) prediction, even a 0.1% accuracy improvement yields hundreds of millions of dollars in revenue [21]. In the periodical deployment approach, while the model is being retrained, new prediction queries and training data are still arriving at the deployment platform. However, the platform has to answer the prediction queries using the currently deployed model. Moreover, the platform appends the new training data to the historical data. By the time the retraining process is over, enough training data is accumulated which requires the deployment platform to perform another retraining. As a result, the deployed model quickly becomes stale.

Although periodical deployment is robust and easy to tune, it cannot maintain the quality of the deployed model without incurring a high training cost. We propose a deployment platform that eliminates the need for retraining, thus significantly reducing the training cost while achieving the same level of quality as the periodical deployment approach. Our deployment platform is robust, i.e., it accommodates many different types of machine learning models and pipelines. Moreover, similar to the periodical deployment, the tuning process of our deployment platform is simple and requires the same amount of user interaction as the periodical deployment.

Our deployment platform continuously updates the model using a combination of the historical and incoming training data. Similar to existing deployment platforms, our platform also utilizes online learning methods to update the model based on the incoming training data. However, instead of the periodical retraining, our deployment platform performs regular updates to the model based on samples of the historical data. Our deployment platform offers the following two features:

Proactive training. Proactive training is the process of utilizing samples of the data to update the deployed model. First, the deployment platform processes a given sample using the pipeline, then it computes a partial gradient and updates the deployed model based on the partial gradient. The updated model is immediately ready for answering prediction queries. Our experiments show that proactive training reduces the training time by one order of magnitude while providing the same level of quality when compared to the periodical deployment approach.

Online Statistics Computation and Dynamic Materialization. Before updating the model using proactive training, the pipeline has to preprocess the training data. Every component of the pipeline needs to scan the data, updates the statistics (for example the mean and the standard deviation of the standard scaler component), and finally, transform the data. Computing these statistics and transforming the data are time consuming processes. Aside from the proactive training, our deployment platform also employs online

Series ISSN: 2367-2005

10.5441/002/edbt.2019.35

learning methods to update the model in real-time. During the online learning, we compute the required statistics and transform the data. The deployment platform stores the updated statistics for every pipeline component and materializes the transformed features by storing them in memory or disk. In presence of a limited storage capacity, the platform removes the older transformed features, and only re-materializes them when needed (through a process called *dynamic materialization*). By reusing the computed statistics and the materialized features during the proactive training, we eliminate the data preprocessing steps of the pipeline and further decrease the proactive training time.

In summary, our contributions are:

- A platform for continuously training deployed machine learning models and pipelines that adapts to the changes in the incoming data. The platform accommodates different types of machine learning models and pipelines. In our experiments, we design and deploy two different machine learning pipelines.
- Proactive training of the deployed models and pipelines that frequently updates the model using samples of the data which completely eliminates the need for periodical retraining while providing the same level of model quality.
- Efficient pipeline processing and model training by online statistics computation and dynamic materialization, which provides up-to-date models for answering prediction queries.

The rest of this paper is organized as follows: In Section 2, we provide background information on the optimization strategy we utilize in our continuous deployment platform and tuning mechanism of the existing deployment platforms. Section 3 describes the details of our continuous training approach. In Section 4, we introduce the architecture of our deployment platform. In Section 5, we evaluate the performance of our continuous deployment platform. Section 6 discusses the related work. Finally, Section 7 presents our conclusion and the future work.

2 BACKGROUND

To continuously train the deployed model, we compute partial updates based on the current model parameters and a combination of the incoming and existing data. To compute the partial updates, we utilize Stochastic Gradient Descent (SGD) [36]. SGD has several parameters (typically referred to as hyperparameters) and in order to work effectively, they have to be tuned. In this section, we describe the details of SGD and its hyperparameters and discuss the effect of the hyperparameters on training machine learning models.

2.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an optimization strategy utilized by many machine learning algorithms for training a model. SGD is an iterative optimization technique where in every iteration, one data point or a sample of the data points is utilized to update the model. SGD is suitable for large datasets as it does not require scanning the entire data in every iteration [5]. SGD is also suitable for online learning scenarios, where new training data becomes available one at a time. Many different machine learning tasks such as classification [23, 36], clustering [6], and matrix factorization [19] utilize SGD in training models. SGD is also the most common optimization strategy for training neural networks on large datasets [12].

To explain the details of SGD, we describe how it is utilized to train a simple linear regression model. In linear regression, the goal is to find the weight vector(w) that minimizes the least-squares cost function ($J(w)$):

$$J(w) = \frac{1}{2}\sum_{i=1}^{N}(x^i w - y^i)^2 \tag{1}$$

where N is the size of the training dataset. To utilize SGD for finding the optimal w, we start from initial random weights. Then in every iteration, we update the weights based on the gradient of the loss function:

$$w^{t+1} = w^t + \eta \sum_{i \in S}(y^i - x^i w)x^i \tag{2}$$

where η is the learning rate hyperparameter and S is the random sample in the current iteration. The algorithm continues until convergence, i.e., when the weight vector does not change after an iteration.

Learning Rate. An important hyperparameter of stochastic gradient descent is the learning rate. The learning rate controls the degree of change in the weights during every iteration. The most trivial approach for tuning the learning rate is to initialize it to a small value and after every iteration decrease the value by a small factor. However, in complex and high-dimensional problems, the simple tuning approach is ineffective [27]. Adaptive learning rate methods such as Momentum [26], Adam [18], Rmsprop [31], and AdaDelta [35] have been proposed. These methods adaptively adjust the learning rate in every iteration to speed up the convergence rate. Moreover, some of the learning rate adaptation methods perform per coordinate modification, i.e., every parameter of the model weight vector is adjusted separately from the others [18, 31, 35]. In many high-dimensional problems, the parameters of the weight vector do not have the same level of importance, therefore each parameter must be treated differently during the training process.

Sample Size. Another hyperparameter of stochastic gradient descent is the sample size (sometimes referred to as the mini-batch size). Given proper learning rate tuning mechanism, SGD eventually converges to a solution regardless of the sample size. However, the sample size can greatly affect the time that is required to converge. Two extremes of the sample size are 1 (every iteration considers 1 data item) and N (similar to batch gradient descent, every iteration scans the entire dataset). Setting the sample size to 1 increases the model update frequency but results in noisy updates. Therefore, more iterations are required for the model to converge. Using the entire data in every iteration leads to more stable updates. As a result, the model training process requires fewer iterations to converge. However, because of the size of the data, individual iterations require more time to complete. A common approach is mini-batch gradient descent. In mini-batch gradient descent, the sample size is selected in such a way that each iteration is fast. Moreover, the training process requires fewer iterations to converge.

2.2 Tuning the Periodical Deployment

Typically, two groups of hyperparameters affect the efficiency of the periodical deployment approach. The first group (the deployment hyperparameters) control the frequency and amount of data for every retraining. The second group (the training hyperparameters) tune the algorithm for retraining procedure. In this work, we are targeting training algorithms based on Stochastic gradient descent. Therefore, the hyperparameters are the learning rate and the sample size.

There are several existing approaches for tuning the training hyperparameters, such as grid search, random search, and sequential model based search [4, 17]. The deployment hyperparameters, however, are typically selected to fit the specific use case. For example, in many of the real-world use cases, one retrains the deployed model using the entire historical data (hyperparameter for the amount of data for every retraining) on a daily basis (hyperparameter for the frequency of the retraining). In the next sections, we describe how we tune the deployment and training hyperparameters of our deployment framework.

3 CONTINUOUS TRAINING APPROACH

In this section, we describe the details of our continuous training approach. Figure 1 shows the workflow of our proposed platform. The platform processes the incoming training data through 5 stages:

1. Discretizing the data: To efficiently preprocess the data and update the model, the platform transforms the data into small chunks and stores them in the storage unit. The platform assigns a timestamp to every chunk indicating its creation time. The timestamp acts as both a unique identifier and an indicator of the recency of the chunk.

2. Preprocessing the data: The platform utilizes the deployed pipeline to preprocess the raw training data chunks and transform them into feature chunks. Then, the platform stores the feature chunks along with a reference to the originating raw data chunk in the storage unit. When the storage unit becomes full, the platform starts removing the oldest feature chunks and only keep the reference to the originating raw data chunks. In case the later stages of the deployment platform request a deleted feature chunk, the platform can recreate the feature chunk by utilizing the referenced raw data chunk. During the preprocessing stage, we utilize *online statistics computation* to compute the required statistics for the different pipeline components. These statistics speed up the data processing in later stages.

3. Sampling the data: A sampler unit samples the feature chunks from the storage. Different sampling strategies are available to address different use-case requirements.

4. Materializing the data: Depending on the size of the storage unit, some preprocessed feature chunks (results of step 2) are not materialized. If the sampler selects unmaterialized feature chunks, the platform recreates these feature chunks by utilizing the deployed pipeline through a process called *dynamic materialization.*

5. Updating the model: By utilizing the preprocessed feature chunks, the platform updates the deployed model through a process called *proactive training.*

In the rest of this section, we first describe the details of the online statistics computation the platform performs during the preprocessing step. Then we introduce the dynamic materialization approach and the effects of different sampling strategies on the materialization process. Finally, we describe the details of the proactive training method.

3.1 Online Statistics Computation

Some components of the machine learning pipeline, such as the standard scaler or the one-hot encoder, require some statistics of the dataset before they process the data. Computing these statistics requires scans of the data. In our deployment platform, we utilize online training as well as proactive training. During the online update of the deployed model, we compute all the necessary statistics for every component. Every pipeline component first reads the incoming data. Then it updates its underlying statistics. Finally, the component transforms and forwards the data to the next component. Online computation of the required statistics eliminates the need to recompute the same statistics during the dynamic materialization and proactive training.

Online statistics computation is only applicable to certain types of pipeline components. The support for stateless pipeline components is trivial as they do not rely on any statistics before transforming the data. For stateful operations, since the statistics update occurs during the online data processing, the platform can only update the statistics that can be computed incrementally. Many of the well-known data preprocessing components (such as standardization and one-hot encoding) require statistics that can be computed incrementally (such as mean, standard deviation, and hash table). However, some pipeline components require statistics that cannot be updated incrementally (such as percentile) or the algorithm utilized by the pipeline component is non-incremental (such as PCA). As a result, our deployment platform does not support such components. Fortunately, recent bodies of work are devoted to developing novel techniques for online feature engineering [32, 33] and approximate machine learning [25] that offer fast and incremental alternatives with theoretical error bounds to non-incremental algorithms.

The platform can also facilitate the online statistics computation for user-defined pipeline components. In Section 4, we describe how users can incorporate this feature into their custom pipeline components.

3.2 Dynamic Materialization

In order to update the statistics of the pipeline components, each component must first transform the data and then forwards the transformed data to the next component. At the end of this process, the pipeline has transformed the data chunks into feature chunks that the model will utilize during the training process. In our continuous deployment platform, we repeatedly sample the data chunks to update the model. Storing the chunks as materialized features greatly reduces the processing time as the entire data preprocessing steps can be skipped during the model update. However, in the presence of a limited storage capacity, one has to consider the effect of storing the materialized feature chunks.

To address the storage capacity issue, we utilize dynamic materialization. While creating the feature chunks, the platform assigns a unique identifier (the creation timestamp) and a reference to the originating raw data chunk. In dynamic materialization, when the size of the stored feature chunks exceeds the storage capacity, the platform removes the content of the oldest materialized feature chunks from the storage and only keeps the unique identifier and the reference to the raw data chunk (similar to cache eviction). The next time the sampler selects one or more of the evicted feature chunks, the platform re-materializes each feature chunk from the raw data chunk by reapplying the deployed pipeline to the raw data chunk. Figure 2 shows the process of dynamic materialization in two possible scenarios. For both scenarios, there are a total 6 data chunks (raw and feature) available in the storage (with timestamps t_0 to t_5). The sampling operation selects the chunks at t_0, t_2, and t_5. In Scenario 1, all the feature chunks are materialized. Therefore, the platform directly utilizes them to update the model. In Scenario 2, the platform has previously evicted some of the materialized feature chunks due to the limit on the storage capacity. In this scenario, the platform first re-materializes the evicted chunks using the deployed pipeline components before updating the model.

It is important to note that the continuous training platform assumes the raw data chunks are always stored and are available for re-materialization. If some of the raw data chunks are not available, the platform ignores these chunks during the sampling operation. A similar issue arises in the periodical deployment approach. If there is not enough space to store all the historical and incoming data, at every retraining, the platform only utilizes the available data in the storage.

3.2.1 Storage requirement for materialized feature chunks. In order to estimate the storage requirement for the preprocessed feature chunks, we investigate the size complexity of different pipeline components in terms of the input size (raw data chunks). Table 1 shows the categories of the pipeline components and their characteristics. Let us assume the total number of the values in a

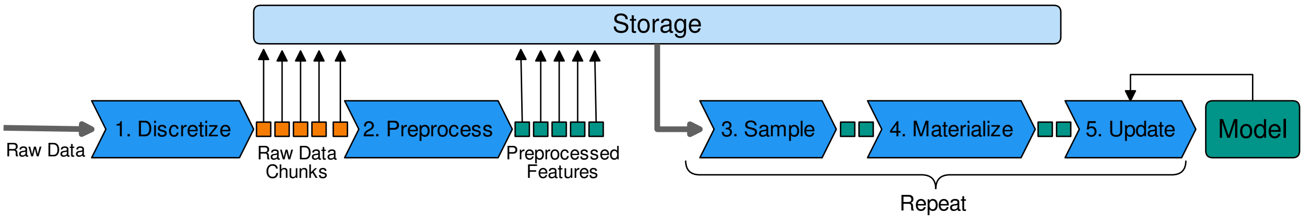


Figure 1: Workflow of our continuous deployment approach. (1) The platform converts the data into small units (2) The platform utilizes the deployed pipeline to preprocess the data and transform the raw data into features and store them in the storage. (3) The platform samples the data from the storage. (4) The platform materializes the sampled data (5) Using the sampled data, the deployment platform updates the deployed model.

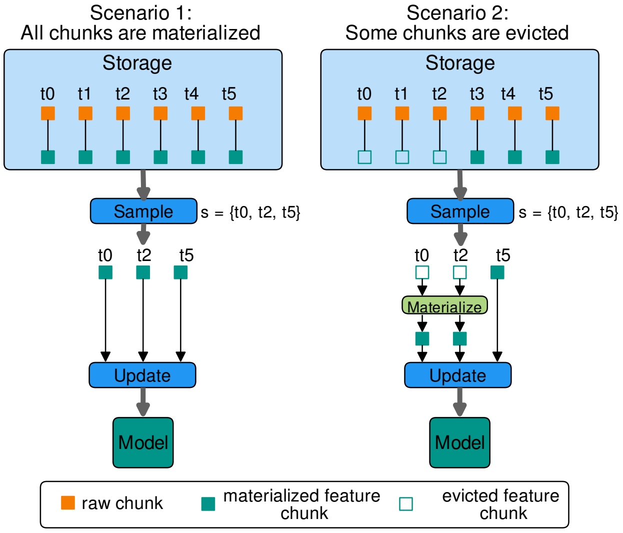


Figure 2: Dynamic Materialization process

Component type	Unit of work	Characteristics
data transformation	data point (row)	filtering or mapping
feature selection	feature (column)	selecting some columns
feature extraction	feature (column)	generating new columns

Table 1: Description of the pipeline component types. Unit of work indicates whether the component operates on a row or a column.

dataset where $\mathcal{R}$ represents the rows and C represents columns is p, where $p = |\mathcal{R}| \times |C|$. Data transformation and feature selection operations either perform a one-to-one mapping (e.g., normalization) or remove some rows or columns (e.g., anomaly filtering and variance thresholding). Therefore, the complexity of data transformation and feature selection operations is linear in terms of the input size ($O(p)$). The case for feature extraction is more complicated as there are different types of feature extraction operations. In many cases, the feature extraction process creates a new feature (column) by combining one or more existing features (such as summing or multiplying features together). This results in a complexity of $O(p)$ as the increase in size is linear with respect to the input size. However, in some case, the feature extraction process generates many features (columns) from a small subset of the existing features. Prominent examples of such operations are one-hot encoding and feature hashing. One-hot encoding converts a column of the data with categorical values into several columns (1 column for each unique value). For every value in the original column, the encoded representation has the value of 1 in the column the value represents and 0 in all the other columns. Consider the case when we are applying the one-hot encoding operation to every column $\forall c \in C$. Furthermore, let us assume $q = \max_{\forall c \in C} |\mathcal{U}(c)|$, where $\mathcal{U}$ is the function that returns the unique values in a column ($\mathcal{U}(x) \in [1, |\mathcal{R}|]$). Thus, the complexity of the one-hot encoding operation is $O(pq)$ (each existing value is encoded with at most q binary values). Based on the value of q, two scenarios may occur:

- if $q \ll |\mathcal{R}| \Rightarrow O(pq) = O(p)$
- if $q \approx |\mathcal{R}| \Rightarrow O(pq) = O(p|\mathcal{R}|) = O(p\dfrac{p}{|C|}) = O(p^2)$

The second scenario represents the worst-case scenario where almost every value is unique and we have very few columns (if the number of columns is large then the complexity is lower than $O(p^2)$). A quadratic growth rate, especially in the presence of large datasets, is not desirable and may render the storage of even a few feature chunks impossible. However, both one-hot encoding and feature hashing produce sparse data where for every encoded data point, only one entry is 1 and all the other entries are 0. Therefore, by utilizing sparse vector representation, we guarantee a complexity of $O(p)$.

Since the complexity is in worst-case scenario linear with respect to the size of the input data and the eviction policy gradually dematerializes the older feature chunks, the platform ensures the size of the materialized features will not unexpectedly exceed the storage capacity.

3.2.2 Effects of sampling strategies on the dynamic materialization. Our platform offers three sampling strategies, namely, uniform, window-based, and time-based (Section 4.2). The choice of the sampling strategy affects the efficiency of the dynamic materialization. Here, we analyze the effects of dynamic materialization in reducing the data processing overhead.

We define N as the maximum number of the raw data chunks, n as the number of existing raw chunks during a sampling operation, m as the maximum number of the materialized feature chunks (corresponds to the size of the dedicated storage for the materialized feature chunks), and s as the sample size (in each sampling operation, we are sampling s chunks out of the available n chunks)[1]. Let us define MS as the number of materialized feature chunks in a sampling operation. The variable MS follows a hypergeometric distribution[2] (sampling without replacement) where the number of success states is m, and the number of draws is s. Therefore, the expected value of MS for a sampling operation with n chunks is:

$$E_n[MS] = s\frac{m}{n}$$

To quantify the efficiency of the dynamic materialization, we introduce the materialization utilization rate with n raw chunks, which indicates the ratio of the materialized feature chunks:

$$\mu_n = \frac{E_n[MS]}{s}$$

Finally, the average materialization utilization rate for the dynamic materialization process is:

$$\mu = \frac{\sum_{n=1}^{N} \mu_n}{N} \tag{3}$$

[1] The value N corresponds to the size of the storage unit dedicated for raw data chunks which bounds the variable n. If we assume n is unbounded, then as $lim_{n \to \infty}$, the probability of sampling materialized feature chunks becomes 0.

[2] https://en.wikipedia.org/wiki/Hypergeometric_distribution

μ indicates the ratio of the feature chunks that do not require re-materialization before updating the model (a μ of 0.5 shows on average half of the sampled chunks are materialized). To simplify the analysis, we assume the platform performs one sampling operation after every incoming data chunk. In reality, a scheduler component governs the frequency of the sampling operation (Section 4.1). Next, we describe how the sampling strategy affects the computation of μ.

Random Sampling: For the random sampling strategy, we compute μ_n as:

$$\mu_n = \begin{cases} 1, & \text{if } n \le m \\ \dfrac{E_n[MS]}{s} = \dfrac{s\frac{m}{n}}{s} = \dfrac{m}{n}, & \text{otherwise} \end{cases}$$

Since for the first m sampling operations the number of raw chunks (n) is smaller than the total size of the materialized chunks (m), μ_n is 1.0 (every sampled chunk is materialized).

$$\begin{aligned} \mu = \frac{\sum\limits_{n=1}^{N} \mu_n}{N} &= \frac{m \times 1.0 + \sum\limits_{n=m+1}^{N} \frac{m}{n}}{N} \\ &= \frac{m + m(\frac{1}{m+1} + \frac{1}{m+2} + \ldots + \frac{1}{N})}{N} \\ &= \frac{m(1 + (H_N - H_m))}{N} \\ &\approx \frac{m(1 + ln(N) - ln(m))}{N} \end{aligned} \tag{4}$$

The highlighted section corresponds to the Harmonic numbers [30]. The t-th harmonic number is:

$$H_t = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{t} \approx ln(t) + \gamma + \frac{1}{2t} - \frac{1}{12t^2}$$

where $\gamma \approx 0.5772156649$ is the Euler-Mascheroni constant. In our analysis, since t is sufficiently large (more than 1000), we ignore $\frac{1}{2t} - \frac{1}{12t^2}$.

Window-based Sampling: In the window-based sampling, we have an extra parameter w which indicates the number of chunks in the active window. If $m \ge w$ then $\mu = 1$, as all the feature chunks in the active window are always materialized. However, when $m < w$:

$$\mu_n = \begin{cases} 1, & \text{if } n \le m \\ \dfrac{E_n[MS]}{s} = \dfrac{m}{n}, & \text{if } m < n \le w \\ \dfrac{E_w[MS]}{s} = \dfrac{m}{w}, & \text{if } w < n \end{cases}$$

therefore:

$$\begin{aligned} \mu = \frac{\sum\limits_{n=1}^{N} \mu_n}{N} &= \frac{m + \sum\limits_{n=m+1}^{w} \frac{m}{n} + (N-w)\frac{m}{w}}{N} \\ &\approx \frac{m + m(H_w - H_m) + (N-w)\frac{m}{w}}{N} \\ &= \frac{m(1 + ln(w) - ln(m) + \frac{N-w}{w})}{N} \end{aligned} \tag{5}$$

Time-based Sampling: For the time-based sampling strategy, there is no direct approach for computing the expected value of MS (the number of the materialized chunks in the sample). However, we are assigning a higher sampling probability to the recent chunks. As a result, we guarantee the time-based sampling has a higher average materialization utilization rate than the uniform sampling. In the experiments, we empirically show the average materialization utilization rate.

In our experiments, we execute a deployment scenario with a total of 12,000 chunks ($N = 12000$), where each chunk is around 3.5 MB (a total of 42 GB). For the uniform sampling strategy, in order to achieve $\mu = 0.91$, using Formula 4, we set the maximum number of the materialized chunks to 7,200 ($m = 7200$). This shows that, in the worst-case scenario (when uniform sampling is utilized), by materializing around 25 GB of the data, we ensure the deployment platform does not need to re-materialize the data 91% of the time.

3.3 Proactive Training

Updating the model is the last step of our continuous deployment platform. We update the model through the proactive training process. Unlike, the full retraining process that is triggered by a certain event (such as a drop in the quality or certain amount of time elapsed since the last retraining), proactive training continuously updates the deployed model. The proactive training utilizes the mini-batch stochastic gradient descent to update the model incrementally. Each instance of the proactive training is analogous to an iteration of the mini-batch SGD. Algorithm 1 shows the pseudocode of the mini-batch SGD algorithm. In mini-batch SGD, we first initialize the model (Line 1). Then, in every iteration, we randomly sample points from the dataset (Line 3), compute the gradient of the loss function J (Line 4), and finally update the model based on the value of the gradient and the learning rate (Line 5). Since the platform executes the proactive training in arbitrary intervals, we must ensure each instance of the proactive training is independent of the previous instances. According to the mini-batch SGD algorithm, each iteration of the SGD only requires the model (m_{i-1}) and the learning rate (η_{i-1}) of the previous iteration (Lines 4 and 5). Given these parameters, iterations of SGD are conditionally independent of each other. Therefore, to execute the proactive training, the deployment platform only needs to store the model weights and the learning rate. By proactively training the deployed model, the platform ensures the model stays up-to-date and provides accurate predictions.

Algorithm 1 mini-batch Stochastic Gradient Descent

```
Input: $D$ = training dataset
Output: $m$ = trained model
1: initialize $m_0$
2: for $i = 1 \ldots n$ do
3:     $s_i$ = sample from $D$
4:     $g = \nabla J(s_i, m_{i-1})$
5:     $m_i = m_{i-1} - \eta_{i-1} g$
6: end for
7: return $m_n$
```

Proactive training is a form of incremental training [14] which is limited to SGD-based models. In our deployment platform, one can replace the proactive training with other forms of incremental training. However, we limit the platform's support to SGD for two reasons. First, SGD is simple to implement and is used for training a variety of machine learning models in different domains [6, 19, 23]. Second, since the combination of the data sampling and the proactive training is similar to the mini-batch SGD procedure, proactive training provides the same regret bound on the convergence rate as the existing stochastic optimization approaches [18, 36].

4 DEPLOYMENT PLATFORM

Our proposed deployment platform comprises of five main components: pipeline manager, data manager, scheduler, proactive trainer, and execution engine. Figure 3 gives an overview of the architecture of our platform and the interactions among its components. At the center of the deployment platform is the pipeline manager. The pipeline manager monitors the deployed pipeline and model, manages the processing of the training data and prediction queries, and enables the continuous update of the deployed model. The data manager and the scheduler enable the pipeline manager to perform proactive training. The proactive trainer component manages the execution of the iterations of SGD on the deployed model. The execution engine is responsible for executing the actual data transformation and model training components of the pipeline.

4.1 Scheduler

The scheduler is responsible for scheduling the proactive training. The scheduler instructs the pipeline manager when to execute the proactive training. The scheduler accommodates two types of scheduling mechanisms, namely, *static* and *dynamic*. The static scheduling utilizes a user-defined parameter that specifies the interval between executions of the proactive training. This is a simple mechanism for use cases that require constant updates to the deployed model (for example, every minute). The dynamic scheduling tunes the scheduling interval based on the rate of the incoming predictions, prediction latency, and the execution time of the proactive training. The scheduler uses the following formula to compute the time when to execute the next proactive training:

$$T' = S * T * pr * pl \tag{6}$$

where T' is the time in seconds when the next proactive training is scheduled to execute, T is the total execution time (in seconds) of the last proactive training, pl is the average prediction latency (second per item), and pr is the average number of prediction queries per second (items per second). S is the slack parameter. Slack is a user-defined parameter to hint the scheduler about the possibility of surges in the incoming prediction queries and training data. During a proactive training, a certain number of predictions queries arrive at the platform ($T * pr$) which requires $T * pr * pl$ seconds to be processed. The scheduler must guarantee that the deployment platform answers all the queries before executing the next proactive training ($T' > T * pr * pl$). A large slack value (≥ 2) results in a larger scheduling interval, thus allocating most of the resources of the deployment platform to the query answering component. A small slack value ($1 \leq S \leq 2$) results in smaller scheduling intervals. As a result, the deployment platform allocates more resources for training the model.

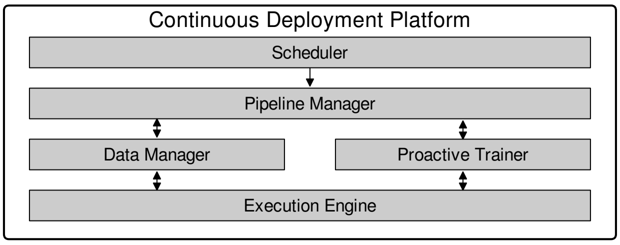


Figure 3: Architecture of the Continuous Deployment Platform

4.2 Data Manager

The data manager component is responsible for the storage of historical data and materialized features, receiving the incoming training data, and providing the pipeline manager with samples of the data. The data manager has four main tasks. First, the data manager discretizes the incoming training data into chunks, assigns a timestamp (which acts as a unique identifier) to them, and stores them in the storage unit. Second, it forwards the data chunks (and the prediction queries) to the pipeline manager for further processing. Third, after the pipeline manager transforms the data into feature chunks, the data manager stores the transformed feature chunks in the storage unit along with a reference to the originating raw data chunk (i.e., the timestamp of the raw data chunk). If the storage unit reaches its limit, the data manager removes old feature chunks. Finally, upon the request of the pipeline manager, the data manager samples the data for proactive training.

During the sampling procedure, the data manager randomly selects a set of chunks by using their timestamp as key. Then, the data manager proceeds as follows. For every sampled timestamp, if the transformed feature chunk exists in the storage, then the data manager forwards it to the pipeline manager. However, if the data manager has previously removed the transformed feature chunks from the storage unit, the data manager forwards the raw data chunk to the pipeline manager and notifies the pipeline manager to re-transform the raw data chunk (i.e., the dynamic materialization process).

The data manager provides three sampling strategies, namely, uniform, time-based, and window-based. The uniform sampling strategy provides a random sample from the entire data where every data chunk has the same probability of being sampled. The time-based sampling strategy assigns weights to every data chunk based on their timestamp such that recent chunks have a higher probability of being sampled. The window-based sampling strategy is similar to the uniform sampling, but instead of sampling from the entire historical data, the data manager samples the data from a given time range. Based on the specific use-case, the user chooses the appropriate sampling strategy. In many real-world use cases (e.g., e-commerce and online advertising), the deployed model should adapt to the more recent data. Therefore, the time-based and window-based sampling provide more appropriate samples for training. However, in some use cases, the incoming training data is not time-dependent (e.g., image classification of objects). In these scenarios, the window-based and the time-based sampling strategies may fail to provide a non-biased sample. In Section 5, we evaluate the effect of the sampling strategy on both the total deployment cost and the quality of the deployed model.

4.3 Pipeline Manager

The pipeline manager is the main component of the platform. It loads the pipeline and the trained model, transforms the data into features using the pipeline, enables the execution of the proactive training, and exposes the deployed model to answer prediction queries.

Each pipeline component must implement two methods: *update* and *transform*. Furthermore, every pipeline component has an internal state for storing the statistics (if needed). During the online training, when new training data becomes available, the pipeline manager first invokes the *update* method which enables the component to update its internal statistics using the incoming data. Then, the pipeline manager invokes the *transform* method, which transforms the data. After forwarding the data through every component of the pipeline, the pipeline manager sends the transformed features to the data manager for storage.

When the scheduler component informs the pipeline manager to execute proactive training, the pipeline manager requests the data manager to provide it with a sample of the data chunks for the next proactive training. If some of the sampled data chunks are not materialized, the pipeline manager re-materializes the chunks by invoking the transform methods of the pipeline components. Then, it provides the proactive trainer with the current model parameters and the materialized sample of the features. Once the proactive training is over, the pipeline manager receives the updated model.

The data manager also forwards the prediction queries to the pipeline manager. Similar to the training data, the pipeline manager sends the prediction queries through the pipeline to perform the necessary data preprocessing (by only invoking the *transform* method of every pipeline component). Using the same pipeline to process both the training data and the prediction queries guarantees that the same set of transformations are applied to both types of data. As a result, the pipeline manager prevents inconsistencies between training and inference that is a common problem in the deployment of machine learning pipelines [3]. Finally, the pipeline manager utilizes the deployed model to make predictions.

4.4 Proactive Trainer

The proactive trainer is responsible for training the deployed model by executing iterations of SGD. In the training process, the proactive trainer receives a training dataset (sampled materialized features) and the current model parameters from the pipeline manager. Then, the proactive trainer performs one iteration of SGD and returns the updated model to the pipeline manager. The proactive trainer utilizes advanced learning rate adaptation techniques such as Adam, Rmsprop, and AdaDelta to dynamically adjust the learning rate parameter when training the model.

In order for the proactive training to update the deployed model, the machine learning model component of the deployed pipeline must implement an *update* method, which is responsible for computing the gradient. To provide support for other types of incremental training approaches, one needs to implement the training logic in the *update* method of the model. However, as described in Section 3.3, the proactive training with data sampling can guarantee convergence only when the SGD optimization is utilized.

4.5 Execution Engine

The execution engine is responsible for executing the SGD and the prediction answering logic. In our deployment platform, any data processing platform capable of processing data both in batch mode (for proactive training) and streaming mode (online learning and answering prediction queries) is a suitable execution engine. Platforms such as Apache Spark [34], Apache Flink [7], and GoogleDataFlow [2] are distributed data processing platforms that support both stream and batch data processing.

5 EVALUATION

To evaluate the performance of our deployment platform, we perform several experiments. Our main goal is to show that the continuous deployment approach maintains the quality of the deployed model while reducing the total training time. Specifically, we answer the following questions:
1. How does our continuous deployment approach perform in comparison to online and periodical deployment approaches with regards to model quality and training time?
2. What are the effects of the learning rate adaptation method, the regularization parameter, and the sampling strategy on the continuous deployment?
3. What are the effects of online statistics computation and dynamic materialization optimizations on the training time?

To that end, we first design two pipelines each processing one real-world dataset. Then, we deploy the pipelines using different deployment approaches.

5.1 Setup

Pipelines. We design two pipelines for all the experiments.

URL pipeline. The URL pipeline processes the URL dataset for classifying URLs, gathered over a 121 days period, into malicious and legitimate groups [22]. The pipeline consists of 5 components: input parser, missing value imputer, standard scaler, feature hasher, and an SVM model. To evaluate the SVM model, we compute the misclassification rate on the unseen data.

Taxi Pipeline. The Taxi pipeline processes the New York taxi trip dataset and predicts the trip duration of every taxi ride [8]. The pipeline consists of 5 components: input parser, feature extractor, anomaly detector, standard scaler, and a Linear Regression model. We design the pipeline based on the solutions of the top scorers of the New York City (NYC) Taxi Trip Duration Kaggle competition[3]. The input parser computes the actual trip duration by first extracting the pickup and drop off time fields from the input records and calculating the difference (in seconds) between the two values. The feature extractor computes the haversine distance[4], the bearing[5], the hour of the day, and the day of the week from the input records. Finally, the anomaly detector filters the trips that are longer than 22 hours, smaller than 10 seconds, or the trips that have a total distance of zero (the car never moved). To evaluate the model, we use the Root Mean Squared Logarithmic Error (RMSLE) measure. RMSLE is also the chosen error metric for the NYC Taxi Trip Duration Kaggle competition.

Deployment Environment. We deploy the URL pipeline on a single laptop running a macOS High Sierra 10.13.4 with 2,2 GHz Intel Core i7, 16 GB of RAM, and 512GB SSD and the Taxi pipeline on a cluster of 21 machines (Intel Xeon 2.4 GHz 16 cores, 28 GB of dedicated RAM per node). In our current prototype, we are using Apache Spark 2.2 as the execution engine. The data manager component utilizes the Hadoop Distributed File System (HDFS) 2.7.1 for storing the historical data [28]. We leverage the SVM, LogisticRegression, and the GradientDescent classes of the machine learning library in Spark (MLlib) to implement the proactive training logic. We represent both the raw data and the feature chunks as RDDs. Therefore, we can utilize the caching mechanism of Apache Spark to simply materialize/dematerialize feature chunks.

Datasets. Table 2 describes the details of the datasets such as the size of the raw data for the initial training, and the amount of data for the prediction queries and further training after deployment. For the URL pipeline, we first train a model on the first day of the data (day 0). For the Taxi pipeline, we train a model using the data from January 2015. For both datasets, since the entire data fits in the memory of the computing nodes, we use batch gradient descent (sampling ratio of 1.0) during the initial training. We then deploy the models (and the pipelines). We use the remaining data for sending prediction queries and further training of the deployed models.

Dataset	size	# instances	Initial	Deployment
URL	2.1 GB	2.4 M	Day 0	Day 1-120
Taxi	42 GB	280 M	Jan15	Feb15 to Jun16

Table 2: Description of Datasets. The Initial and Deployment columns indicate the amount of data used during the initial model training and the deployment phase (prediction queries and further training data)

Evaluation metrics. For experiments that compare the quality of the deployed model, we utilize the prediction queries to compute the cumulative prequential error rate of the deployed models over time [11]. For experiments that capture the cost of the deployment, we measure the time the platforms spend in updating the model, performing proactive training (retraining for the periodical deployment scenario), and answering prediction queries.

Deployment process. The URL dataset does not have timestamps. Therefore, we divide every day of the data into chunks of 1 minute which results in a total of 12000 chunks, each one with the size of roughly 200KB. The deployment platform first uses the chunks for prequential evaluation and then updates the deployed

[3] https://www.kaggle.com/c/nyc-taxi-trip-duration/

[4] https://en.wikipedia.org/wiki/Haversine_formula

[5] https://en.wikipedia.org/wiki/Bearing_(navigation)

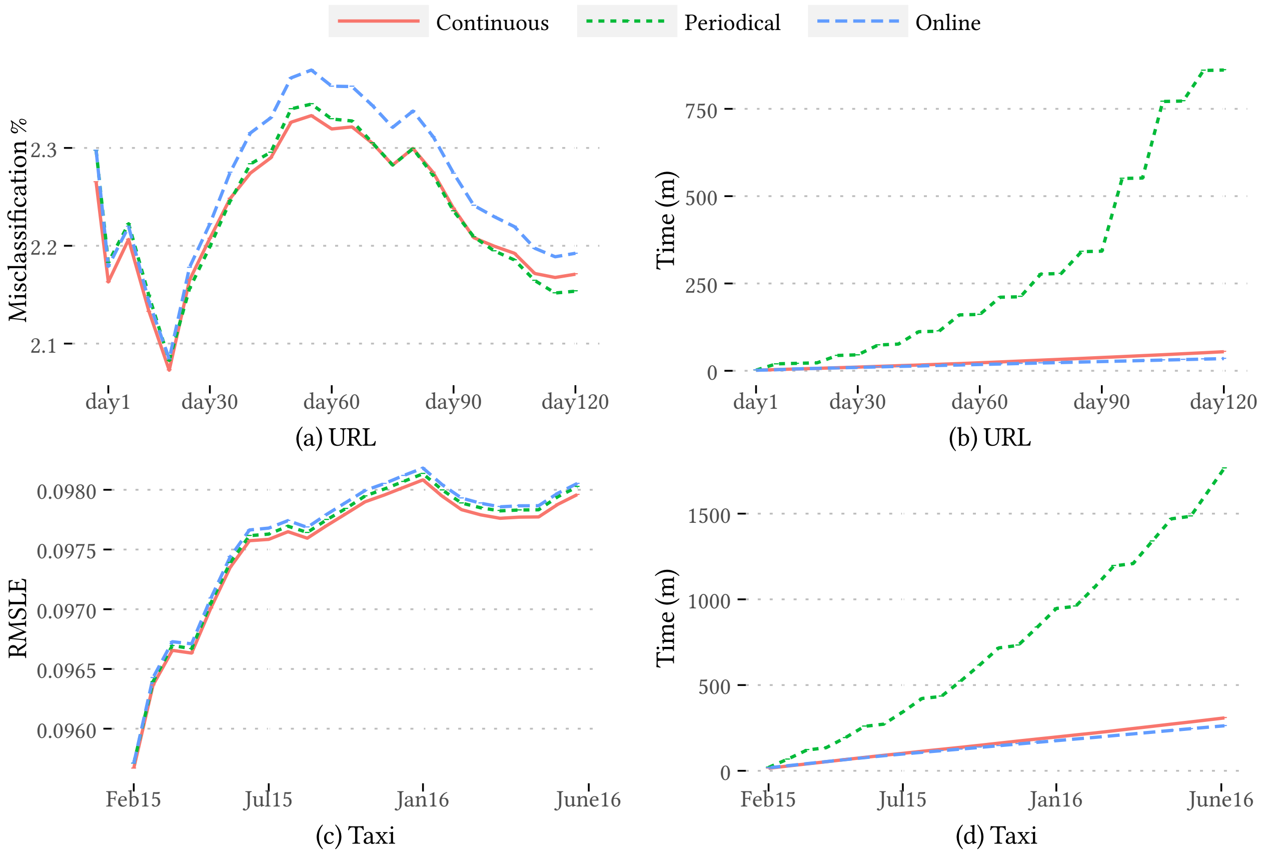


Figure 4: Model Quality and Training cost for different deployment approaches

model. The Taxi dataset includes timestamps. In our experiments, each chunk of the Taxi dataset contains one hour of the data, which results in a total of 12382 chunks, with an average size of 3MB per chunk. The deployment platform processes the chunks in order of the timestamps (from 2015-Feb-01 00:00 to 2016-Jun-30 24:00, an 18 months period).

5.2 Experiment 1: Deployment Approaches

In this experiment, we investigate the effect of our continuous deployment approach on model quality and the total training time. We use 3 different deployment approaches.

- Online: deploy the pipeline, then utilize online gradient descent with Adam learning rate adaptation method for updating the deployed model.
- Periodical: deploy the pipeline, then periodically retrain the deployed model.
- Continuous: deploy the pipeline, then continuously update the deployed model using our platform.

The periodical deployment initiates a full retraining every 10 days and every month for URL and Taxi pipelines, respectively. Since the rate of the incoming training and prediction queries are known, we use static scheduling for the proactive training. Based on the size and rate of the data, our deployment platform executes the proactive training every 5 minutes and 5 hours for the URL and Taxi pipelines, respectively. To improve the performance of the periodical deployment, we utilize the warm starting technique, used in the TFX framework [3]. In warm starting, each periodical training uses the existing parameters such as the pipeline statistics (e.g., standard scaler), model weights, and learning rate adaptation parameters (e.g., the average of past gradients used in Adadelta, Adam, and Rmsprop) when training new models.

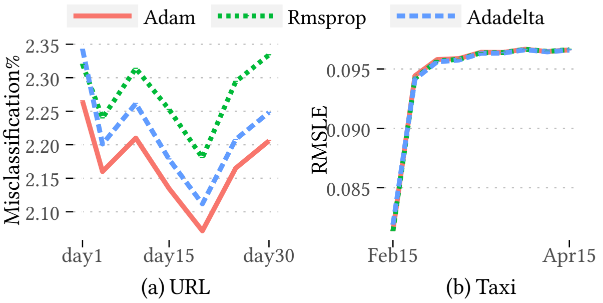
Figure 4 (a) and (c) show the cumulative error rate over time for the different deployment approaches. For both datasets, the continuous and the periodical deployment result in a lower error rate than the online deployment. Online deployment visits every incoming training data point only once. As a result, the model updates are more prone to noise. This results in a higher error rate than the continuous and periodical deployment. In Figure 4 (a), during the first 110 days of the deployment, the continuous deployment has a lower error rate than the periodical deployment. Only after the final retraining, the periodical deployment slightly outperforms the continuous deployment. However, from the start to the end of the deployment process, the continuous deployment improves the average error rate by 0.3% and 1.5% over the periodical and online deployment, respectively. In Figure 4 (c), for the Taxi dataset, the continuous deployment always attains a smaller error rate than the periodical deployment. Overall, the continuous deployment improves the error rate by 0.05% and 0.1% over the periodical and online deployment, respectively.

When compared to the online deployment, periodical deployment slightly decreases the error rate after every retraining. However, between every retraining, the platform updates the model using online learning. This contributes to the higher error rate than the continuous deployment, where the platform continuously trains the deployed model using samples of the historical data.

In Figure 4 (b) and (d), we report the cumulative cost over time for every deployment platform. We define the deployment cost as the total time spent in data preprocessing, model training, and performing prediction. For the URL dataset (Figure 4 (b)), online deployment has the smallest cost (around 34 minutes) as it only scans each data point once (around 2.4 million scans). The continuous deployment approach scans 45 million data points. However, the total cost at the end of the deployment is only 50% larger than the online deployment approach (around 54 minutes). Because of the online statistics computation and the dynamic materialization optimizations, a large part of the data preprocessing time is avoided. For the periodical deployment approach, the cumulative deployment cost starts similar to the online deployment approach. However, after every offline retraining, the deployment cost substantially increases. At the end of the deployment process, the total cost for the periodical deployment is more than 850 minutes which is 15 times more than the total cost of the continuous deployment approach. Each data point in the URL dataset has more than 3 million features. Therefore, the convergence time for each retraining is very high. The high data-dimensionality and repeated data preprocessing contribute to the large deployment cost of the periodical deployment.

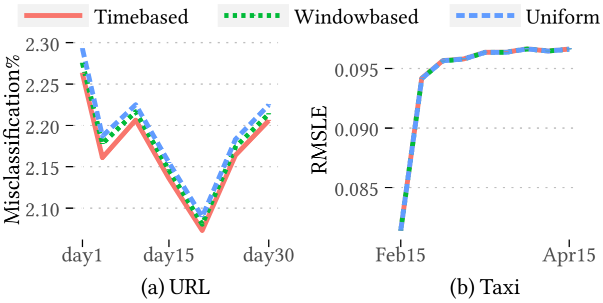
For the Taxi dataset (Figure 4 (d)), the cost of online, continuous, and periodical deployments are 262, 308, and 1765 minutes, respectively. Similar to the URL dataset, continuous deployment only adds a small overhead to the deployment cost when compared with the online deployment. Contrary to the URL dataset, the feature size of the Taxi dataset is 11. Therefore, offline retraining converges faster to a solution. As a result, for the Taxi dataset, the cost of the periodical deployment is 6 times larger than the continuous deployment (instead of 15 times for URL dataset).

5.3 Experiment 2: System Tuning



(a) URL (b) Taxi

Figure 5: Result of hyperparameter tuning during the deployment



(a) URL (b) Taxi

Figure 6: Effect of different sampling methods on quality

In this experiment, we investigate the effect of different parameters on the quality of the models after deployment. As described in Section 3.3, proactive training is an extension of the stochastic gradient descent to the deployment phase. Therefore, we expect the set of hyperparameters with the best performance during the initial training also performs the best during the deployment phase.

Proactive Training Parameters. Stochastic gradient descent is heavily dependent on the choice of learning rate and the regularization parameter. To find the best set of hyperparameters for the initial training, we perform a grid search. We use advanced learning rate adaptation techniques (Adam, Adadelta, and Rmsprop) for both initial and proactive training. For each dataset, we divide the initial data (from Table 2) into a training and evaluation set. For each configuration, we first train a model using the training set and then evaluate the model using the evaluation set. Table 3 shows the result of the hyperparameter tuning for every pipeline. For the URL dataset, Adam with regularization parameter $1E$-3 yields the model with the lowest error rate. The Taxi dataset is less complex than the URL dataset and has a smaller number of feature dimensions. As a result, the choice of different hyperparameter does not have a large impact on the quality of the model. The Rmsprop adaptation technique with the regularization parameter of $1E$-4 results in a slightly better model than the other configurations.

After the initial training, for every configuration, we deploy the model and use 10 % of the remaining data to evaluate the model after deployment. Figure 5 shows the results of the different hyperparameter configurations on the deployed model. To make the deployment figure more readable, we avoid displaying the result of every possible combination of hyperparameters and only show the result of the best configuration for each learning rate adaptation technique. For the URL dataset, similar to the initial training, Adam with regularization parameter $1E$-3 results in the best model. For the Taxi dataset, we observe a similar behavior to the initial training where different configurations do not have a significant impact on the quality of the deployed model.

This experiment confirms that the effect of the hyperparameters (learning rate and regularization) during the initial and proactive training are the same. Therefore, we tune the parameters of the proactive training based on the result of the hyperparameter search during the initial training.

Sampling Methods. The choice of the sampling strategy also affects the proactive training. Each instance of the proactive training updates the deployed model using the provided sample. Therefore, the quality of the model after an update is directly related to the quality of the sample. We evaluate the effect of three different sampling strategies, namely, time-based, window-based, and uniform, on the quality of the deployed model. The sample size is similar to the sample size during the initial training ($16k$ and $1M$ for URL and Taxi data, respectively). Figure 6 shows the effect of different sampling strategies on the quality of the deployed model. For the URL dataset, time-based sampling improves the average error rate by 0.5% and 0.9% over the window-based and uniform sampling, respectively. As new features are added to the URL dataset over time, the underlying characteristics of the dataset gradually change [22]. A time-based sampling approach is more likely to select the recent items for the proactive training. As a result, the deployed model performs better on the incoming prediction queries. The underlying characteristics of the Taxi dataset are known to remain static over time. As a result, we observe that different sampling strategies have the same effect on the quality of the deployed model. Our experiments show that for datasets that gradually change over time, time-based sampling outperforms other sampling strategies. Moreover, time-based sampling performs similarly to window-based and uniform sampling for datasets with stationary distributions.

5.4 Experiment 3: Optimizations Effects

In this experiment, we analyze the effect of the system optimizations, i.e., online statistics computation and the dynamic materialization on the total deployment cost. We define the materialization rate (i.e., $\frac{m}{n}$, as described in Section 3.2) as the ratio of the number of materialized chunks over the total number of chunks (both URL and Taxi have around 12,000 chunks in total). For both datasets, the materialization rates of 0.0, 0.2, 0.6, and 1.0 indicates that 0, 2400,

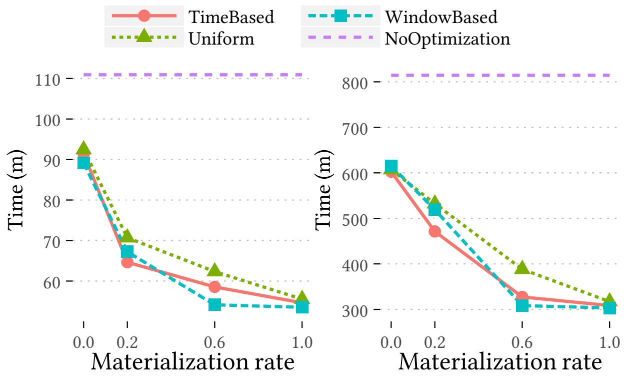
	URL			Taxi		
Adaptation	1E-2	1E-3	1E-4	1E-2	1E-3	1E-4
Adam	0.030	**0.026**	0.035	0.09553	0.09551	**0.09551**
RMSProp	0.030	**0.027**	0.034	0.09552	0.09552	**0.09550**
Adadelta	0.029	**0.028**	0.034	**0.09609**	0.09610	0.09619

Table 3: Hyperparameter tuning during initial training (bold numbers show the best results for each adaptation techniques)

7200, and 12000 chunks are materialized. For the window-based sampling strategy, we set the window size to 6,000 chunks (half of the total chunks). In this experiment, we assume the raw data is always stored in memory. The total size of the datasets after materialization is 5.2 GB and 59 GB for the URL and Taxi datasets, respectively. Therefore, when setting the materialization rate to a specific value, we must ensure we have enough memory capacity to store both the materialized and the raw data. Table 4 shows the empirical values and theoretical estimates of μ for different settings. For both the uniform and time-based sampling, the empirical and analytical computation yield similar values. Moreover, the empirical computation shows that the time-based strategy performs better than the uniform sampling strategy. When the number of materialized feature chunks is 0 or 12000, the design of the deployment platform guarantees that μ is 0.0 and 1.0, respectively. Therefore, we do not report those results in the table.

	URL		Taxi	
Sampling	$\frac{m}{n}$=0.2	$\frac{m}{n}$=0.6	$\frac{m}{n}$=0.2	$\frac{m}{n}$=0.6
Uniform	0.52 (**0.52**)	0.91 (**0.91**)	0.51 (**0.52**)	0.90 (**0.91**)
Window-based	0.58 (**0.58**)	1.0 (**1.0**)	0.57 (**0.58**)	1.0 (**1.0**)
Time-based	0.68	0.97	0.65	0.97

Table 4: Empirical computation and theoretical estimates (bold numbers) of μ for different sampling strategies and materialization rates ($\frac{m}{n}$). We omit the materialization rates 0.0 and 1.0 since both the empirical and theoretical estimates of μ are 0.0 and 1.0 for every sampling strategy.



(a) URL (b) Taxi

Figure 7: Effect of the online statistics computation and dynamic materialization on the deployment cost

To examine the effect of μ on the deployment cost, we plot the total deployment cost using different sampling strategies and materialization rates ($\frac{m}{n}$) for the URL and Taxi deployment scenarios in Figure 7. When the materialization rate is 0.0 or 1.0, the sampling strategies have similar effects on the deployment cost. Therefore, the total deployment cost for every sampling strategy is 90 minutes for URL and 600 minutes for Taxi deployment scenario, when the materialization rate is 0.0. Similarly, the deployment cost is 54 minutes for URL and 308 minutes for Taxi, when the materialization rate is 1.0 (an improvement of 40% for URL and 49% for Taxi deployment scenarios).

For the URL deployment scenario, when the materialization rate is 0.2, time-based, window-based, and uniform sampling improve the deployment cost by 30%, 25%, and 23% in comparison with the materialization rate of 0.0. Similarly, in the Taxi deployment scenario, time-based, window-based, and uniform sampling improve the deployment cost by 22%, 16%, and 12%, respectively. Time-based sampling performs better since it has a higher μ value than the other two sampling strategies (Table 4). When the materialization rate is 0.2, the rate of the decrease in the deployment cost for the URL scenario is greater than the Taxi scenario. We attribute this difference in the decrease in the deployment cost to two reasons. First, the number of sampled chunks in the Taxi deployment scenario is larger than the URL (720 for Taxi and 100 for URL). Before updating the model, we utilize the *context.union* operation of Spark, to combine all the non-materialized and materialized chunks. The union operation incurs a larger overhead when the number of underlying chunks is bigger. Second, we execute the URL deployment scenario on a single machine with SSD. Since materializing data that resides on an SSD is faster than an HD, we observe a larger decrease in the deployment cost.

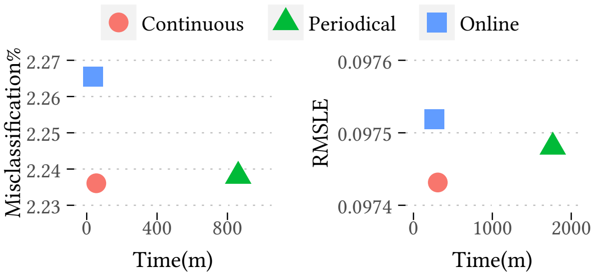
When the materialization rate is 0.6, window-based sampling has the best performance. Since the size of the window is smaller than the number of the materialized feature chunks, every sampled feature chunk is materialized. For the URL deployment scenario, window-based, time-based, and uniform sampling improves the performance by 40%, 36%, and 33%, respectively. For the Taxi deployment scenario, window-based, time-based, and uniform sampling improves the performance by 49%, 46%, and 37%, respectively. Similar phenomena explain the difference in performance improvement at materialization rate of 0.6 between the Taxi and the URL deployment scenarios. At materialization rate of 0.6, more than 90% of the chunks are materialized. Therefore, the Taxi deployment scenario gains relatively more than the URL deployment scenario from a smaller number of disk I/O operations.

To analyze the effect of the online statistics computation on the deployment cost, we also execute the deployment scenarios without the online statistics computation and the dynamic materialization optimizations. In this case, the deployment platform first accesses the sampled raw data chunk directly from the disk. Then, the platform recomputes the required statistics of every component by scanning the data. Finally, it transforms the raw data chunk into the preprocessed feature chunks by utilizing the deployed pipeline. Without the optimizations, the choice of the sampling strategy does not affect the total deployment time (similar to the materialization rate of 0.0). Therefore, when the optimizations are disabled, we only show the results for the time-based sampling (depicted as NoOptimization in Figure 7). The extra disk access and data processing result in an increase of %110 for the URL (Figure 7a) and %170 for the Taxi deployment scenarios (Figure 7b) when compared with a fully optimized execution (with online statistics computation and materialization rate of 1.0). Similar to the dynamic materialization case, we observe a larger increase in the deployment cost of the Taxi deployment scenario due to the larger overhead of disk I/O.

The result of this experiment shows that even under limited storage we can benefit from the dynamic materialization, especially for the time-based and window-based sampling strategies. Furthermore, online statistics computation can improve the total deployment cost, especially when the expected amount of incoming data is large.

5.5 Discussion

Trade-off between quality and training cost. In many real-world use cases, even a small improvement in the quality of the deployed model can have a significant impact [21]. Therefore, one can employ more complex pipelines and machine learning training algorithms to train better models. However, during the deployment where prediction queries and training data become available at a high rate, one must consider the effect of the training time. To ensure the model is always up-to-date, the platform must constantly update the model. Long retraining time may have a negative impact on the prediction accuracy as the deployed model becomes stale. Figure 8 shows the trade-off between the average quality and the total cost of the deployment. By utilizing continuous deployment, we reduce the total cost of the deployment 6 to 15 times when compared with the periodical deployment, while providing the same quality (even slightly outperforming the periodical deployment by 0.05% and 0.3% for the Taxi and URL datasets, respectively.)



(a) URL (b) Taxi

Figure 8: Trade-off between average quality and deployment cost

Staleness of the model during the periodical deployment. In the experiments of the periodical deployment approach, we pause the inflow of the training data and prediction queries. However, in real-world scenarios, the training data and the prediction queries constantly arrive at the platform. Therefore, the periodical deployment platform pauses the online update of the deployed model and answers the prediction queries using the currently deployed model (similar to how Velox operates [9]). As a result, the error rate of the deployed model may increase during the retraining process. However, in our continuous deployment platform, the average time for the proactive training is small (200 ms for the URL dataset and 700 ms for the Taxi dataset). Therefore, the continuous deployment platform always performs the online model update and answers the predictions queries using an up-to-date model.

6 RELATED WORK

Traditional machine learning systems focus solely on training models and leave the task of deploying and maintaining the models to the users. It has only been recently that some platforms, for example LongView [1], Velox [9], Clipper [10] , and TensorFlow Extended [3] have proposed architectures that also consider model deployment and query answering.

LongView integrates predictive machine learning models into relational databases. It answers predictive queries and maintains and manages the models. LongView uses techniques such as query optimization and materialized view selection to increase the performance of the system. However, it only works with batch data and does not provide support for real-time queries. As a result, it does not support continuous and online learning. In contrast, our platform is designed to work in a dynamic environment where it answers prediction queries in real-time and continuously updates the model.

Velox is an implementation of the common periodical deployment approach. Velox supports online learning and can answer prediction queries in real-time. It also eliminates the need for the users to manually retrain the model offline. Velox monitors the error rate of the model using a validation set. Once the error rate exceeds a predefined threshold, Velox initiates a retraining of the model using Apache Spark. However, Velox has four drawbacks. First, retraining discards the updates that have been applied to the model so far. Second, the process of retraining on the full dataset is resource intensive and time-consuming. Third, the platform must disable online learning during the retraining. Lastly, the platform only deploys the final model and does not support the deployment of the machine learning pipeline. Our approach differs from Velox as it exploits the underlying properties of SGD to integrate the training process into the platform's workflow. Our platform replaces the offline retraining with proactive training. As a result, our deployment platform maintains the model quality with a small training cost. Moreover, our deployment platform deploys the machine learning pipeline alongside the model.

Clipper is another machine learning deployment platform that focuses on producing high-quality predictions by maintaining an ensemble of models. For every prediction query, Clipper examines the confidence of every deployed model. Then, it selects the deployed model with the highest confidence for answering the prediction query. However, it does not update the deployed models, which over time leads to outdated models. On the other hand, our deployment platform focuses on maintenance and continuous update of the deployed models.

TensorFlow Extended (TFX) is a platform that supports the deployment of machine learning pipelines and models. TFX automatically stores new training data, performs analysis and validation of the data, retrains new models, and finally redeploys the new pipelines and models. Moreover, TFX supports the warm starting optimization to speed up the process of training new models. TFX aims to simplify the process of design and training of machine learning pipelines and models, simplify the platform configuration, provide platform stability, and minimize the disruptions in the deployment platform. For use cases that require months to deploy new models, TFX reduces the time to production from the order of months to weeks. Although TFX uses the term "continuous training" to describe the deployment platform, it still periodically retrains the deployed model on the historical dataset. On the contrary, our continuous deployment platform performs more rapid updates to the deployed model. By exploiting the properties of SGD optimization technique, our deployment platform rapidly updates the deployed models (seconds to minutes instead of several days or weeks) without increasing the overhead. Our proactive training component can be integrated into the TFX platform to speed up the process of pipeline and model update.

Weka [15], Apache Mahout [24], and Madlib [16] are systems that provide the necessary toolkits to train machine learning models. All of these systems provide a range of training algorithms for machine learning methods. However, they do not support the management and deployment of machine learning models and pipelines. Our platform focuses on continuous deployment and management of machine learning pipelines and models after the initial training.

MLBase [20] and TuPaq [29] are model management systems. They provide a range of training algorithms to create machine learning models and mechanism for model search as well as model management. They focus on training high-quality models by performing automatic feature engineering and hyper-parameter search. However, they only work with batch datasets. Moreover, the users have to manually deploy the models and make them available for answering prediction queries. On the contrary, our deployment platform focuses on the continuous deployment of pipelines and models.

7 CONCLUSIONS

We propose a deployment platform for continuously updating machine learning pipelines and models. After a machine learning pipeline is designed and initially trained on a dataset, our platform deploys the pipeline and makes it available for answering prediction queries. To ensure that the model maintains an acceptable error rate , existing deployment platforms periodically retrain the deployed model. However, periodical retraining is a time-consuming and resource-intensive process. As a result of the lengthy training process, the platform cannot produce fresh models. This results in model-staleness which may decrease the quality of the deployed model.

We propose a training approach, called proactive training, that utilizes samples of the historical data to train the deployed pipeline. Proactive training replaces the periodical retraining, which provides the same level of model quality without the lengthy retraining process. We also propose online statistics computation and dynamic materialization of the preprocessed features which further decreases the training time. We propose a modular design that enables our deployment platform to be integrated with different scalable data processing platforms.

We implement a prototype using Apache Spark to evaluate the performance of our deployment platform. In our experiments, we develop two pipelines with two machine learning models to process two real-world datasets. We discuss how to tune the deployment platform based on the available historical data. Our experiments show that our continuous deployment reduces the total deployment cost by a factor of 6 and 15 for the Taxi and URL datasets, respectively. Moreover, continuous deployment platform provides the same level of quality for the deployed model when compared with the periodical deployment approach.

Currently, we provide support for anomaly and concept drift detection through components of the machine learning pipeline. However, given the large impact of the concept drift and anomalies on the performance of a deployed model, in the future work, we plan to extend our platform to provide native support for both concept drift and anomaly detection and alleviation. Furthermore, we plan to integrate more complex machine learning pipelines and models (e.g., neural networks) into our deployment platform.

Acknowledgments. This work has been supported by the European Commission through Streamline (ref. 688191), by the German Ministry for Education and Research through Berlin Big Data Center BBDC (funding mark 01IS14013A) and Berliner Zentrum für Maschinelles Lernen BZML (funding mark 01IS18037A), and by Zalando.

REFERENCES

[1] M. Akdere, U. Cetintemel, M. Riondato, E. Upfal, and S. Zdonik. 2011. The Case for Predictive Database Systems: Opportunities and Challenges.. In *CIDR*. 167–174.

[2] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[3] D. Baylor, E. Breck, H. Cheng, N. Fiedel, et al. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1387–1395.

[4] J. Bergstra and Y. Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, Feb (2012), 281–305.

[5] L. Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 177–186.

[6] L. Bottou, Y. Bengio, et al. 1995. Convergence properties of the k-means algorithms. *Advances in neural information processing systems* (1995), 585–592.

[7] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, et al. 2015. Apache flink: Stream and batch processing in a single engine. *Data Engineering* (2015), 28.

[8] O. Chapelle. [n. d.]. NYC Taxi & Lomousine Commision Trip Record Data. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml. [Online; accessed 10-April-2018].

[9] D. Crankshaw, P. Bailis, J. Gonzalez, H. Li, et al. 2014. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809* (2014).

[10] D. Crankshaw, X. Wang, G. Zhou, M. Franklin, et al. 2016. Clipper: A Low-Latency Online Prediction Serving System. *arXiv preprint arXiv:1612.03079* (2016).

[11] P. Dawid. 1984. Present position and potential developments: Some personal views: Statistical theory: The prequential approach. *Journal of the Royal Statistical Society. Series A (General)* (1984), 278–292.

[12] J. Dean, G. Corrado, R. Monga, K. Chen, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[13] J. Duchi, E. Hazan, and Y. Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12, Jul (2011), 2121–2159.

[14] A. Gepperth and B. Hammer. 2016. Incremental learning algorithms and applications. In *European Symposium on Artificial Neural Networks (ESANN)*.

[15] M. Hall, E. Frank, G. Holmes, B. Pfahringer, et al. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

[16] J. Hellerstein, C. Ré, F. Schoppmann, D. Wang, et al. 2012. The MADlib analytics library: or MAD skills, the SQL. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1700–1711.

[17] F. Hutter, H. Hoos, and K. Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*. Springer, 507–523.

[18] D. Kingma and J. Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[19] Y. Koren, R. Bell, C. Volinsky, et al. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009), 30–37.

[20] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, et al. 2013. MLbase: A Distributed Machine-learning System.. In *CIDR*, Vol. 1. 2–1.

[21] X. Ling, W. Deng, C. Gu, H. Zhou, et al. 2017. Model Ensemble for Click Prediction in Bing Search Ads. In *Proceedings of the 26th International Conference on World Wide Web Companion*. International World Wide Web Conferences Steering Committee, 689–698.

[22] J. Ma, L. Saul, S. Savage, and G. Voelker. 2009. Identifying suspicious URLs: an application of large-scale online learning. In *Proceedings of the 26th annual international conference on machine learning*. ACM, 681–688.

[23] H. McMahan, G. Holt, D. Sculley, M. Young, et al. 2013. Ad Click Prediction: a View from the Trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*.

[24] S. Owen, R. Anil, T. Dunning, and E. Friedman. 2011. *Mahout in Action*. Manning Publications Co., Greenwich, CT, USA.

[25] Y. Park, J. Qing, X. Shen, and B. Mozafari. 2018. BlinkML: Approximate Machine Learning with Probabilistic Guarantees. (2018).

[26] N. Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural networks* 12, 1 (1999), 145–151.

[27] T. Schaul, S. Zhang, and Y. LeCun. 2013. No more pesky learning rates. *ICML (3)* 28 (2013), 343–351.

[28] K. Shvachko, H. Kuang, S. Radia, and R.Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. IEEE, 1–10.

[29] E. Sparks, A. Talwalkar, M. Franklin, M. Jordan, and T. Kraska. 2015. TuPAQ: An efficient planner for large-scale predictive analytic queries. *arXiv preprint arXiv:1502.00068* (2015).

[30] Z. Sun. 2012. Arithmetic theory of harmonic numbers. *Proc. Amer. Math. Soc.* 140, 2 (2012), 415–428.

[31] T. Tieleman and G. Hinton. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning* 4, 2 (2012), 26–31.

[32] I. Tsamardinos, G. Borboudakis, P. Katsogridakis, P. Pratikakis, and V. Christophides. 2018. A greedy feature selection algorithm for Big Data of high dimensionality. *Machine Learning* (2018), 1–54.

[33] K. Yu, X. Wu, W. Ding, and J. Pei. 2016. Scalable and accurate online feature selection for big data. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 11, 2 (2016), 16.

[34] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. 2010. Spark: cluster computing with working sets. *HotCloud* 10 (2010), 10–10.

[35] M. Zeiler. 2012. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* (2012).

[36] T. Zhang. 2004. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the twenty-first international conference on Machine learning*. ACM, 116.

Discovering Order Dependencies through Order Compatibility

Cristian Consonni
University of Trento
cristian.consonni@unitn.it

Paolo Sottovia
University of Trento
ps@disi.unitn.it

Alberto Montresor
University of Trento
alberto.montresor@unitn.it

Yannis Velegrakis
Utrecht University and University of Trento
velgias@disi.unitn.eu

ABSTRACT

A relevant task in the exploration and understanding of large datasets is the discovery of hidden relationships in the data. In particular, functional dependencies have received considerable attention in the past. However, there are other kinds of relationships that are significant both for understanding the data and for performing query optimization. Order dependencies belong to this category. An *order dependency* states that if a table is ordered on a list of attributes, then it is also ordered on another list of attributes. The discovery of order dependencies has been only recently studied. In this paper, we propose a novel approach for discovering order dependencies in a given dataset. Our approach leverages the observation that discovering order dependencies can be guided by the discovery of a more specific form of dependencies called *order compatibility dependencies.* We show that our algorithm outperforms existing approaches on real datasets. Furthermore, our algorithm can be parallelized leading to further improvements when it is executed on multiple threads. We present several experiments that illustrate the effectiveness and efficiency of our proposal and discuss our findings.

1 INTRODUCTION

In the big data era, the volume and complexity of available datasets has grown so much that data engineers are having a hard time interpreting the information contained in them. In such a reality, the ability to discover hidden dependencies in some automatic way is fundamental. Dependencies across different parts of the data play a significant role in query optimization, since redundant information may be ignored making the query evaluation faster. Furthermore, parts of the data may be replaced with others that are easier to manipulate, without affecting the final result. Data profiling may help with data quality since it highlights constraints that may exist in the data but are not fully satisfied and have not been enforced when designing the database.

Dependency discovery is not a new challenge. Functional and inclusion dependencies are the most common type of dependencies and have been studied extensively [14]. A *functional dependency* states that if two different data elements sharing a common structure have the same part A, then some other part B should also have the same value. An *inclusion dependency* states that the values of the data elements in some part A must be a subset of the values in a subpart B of some other portion of the dataset.

An example of functional dependency can be seen in Table 1, that shows a relational table with data regarding yearly incomes, savings and taxes. Assume that the tax system is a progressive one that categorizes the different incomes into brackets, each of them characterized by a tax percentage. Thus, there is a functional dependency from the income amount to the tax brackets, i.e., `income` → `bracket`. Since for every income range the percentage is fixed, there are two other functional dependencies from the income to the tax amount and vice-versa, i.e., `income` → `tax` and `tax` → `income`. Using the transitive property of functional dependencies a new one can be inferred, i.e., `tax` → `bracket`.

A closer look at Table 1 can illustrate another, stronger, form of dependency: as the income is increasing, bracket and tax amount are increasing as well. In other words, if we were to order the table based on the income column, each one of the bracket and the tax amount columns will also end up being ordered. This form of dependency is known as an *order dependency* and is typically noted with the ↦ symbol, i.e., `income` ↦ `tax`, which is read as: `income` *orders* `tax`.

The knowledge encoded by order dependencies can be applied to various tasks during the entire data life-cycle [3]: in the design phase, order dependencies can be exploited to assist schema design [21] or for selecting indexes [7]; if data are extracted from unstructured sources, order dependencies can aid knowledge discovery, to find hidden properties of the data; in the context of data profiling [13], data integration and cleansing [5], order dependencies can be used to describe a dataset; for data quality [8], order dependencies can be used as requirements or constraints [1].

The most important application of order dependencies is their use in the optimization of queries; in particular, they can be used to rewrite the `ORDER BY` clauses in SQL queries in ways similar to that of functional dependencies for the `GROUP BY` statements [17, 22]. Consider the following query:

```
SELECT income, bracket, tax
FROM TaxInfo
ORDER BY income, bracket, tax
```

TaxInfo

name	income	savings	bracket	tax
T. Green	35,000	3,000	1	5,250
J. Smith	40,000	4,000	1	6,000
J. Doe	40,000	3,800	1	6,000
S. Black	55,000	6,500	2	8,500
W. White	60,000	6,500	2	9,500
M. Darrel	80,000	10,000	3	14,000

Table 1: A relational table with financial information.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.36

Given that the order dependencies `income` ↦ `tax` and `income` ↦ `bracket` hold, the query optimizer can infer that sorting by `income` makes the ordering on the other two columns redundant, so the `ORDER BY` clause can be simplified to `ORDER BY income`.

The concept of order dependency in the context of database systems first appeared under the name of *point-wise order* [9–11]. A point-wise ordering specifies that a *set* of columns orders another *set* of columns. In the example of Table 1, the point-wise order dependency `income,tax` ⇝ `bracket` holds because if both of the tuples (`income`, `tax`) and (`tax`, `income`) are lexicographically ordered, then the column `bracket` is ordered in the same way. A new definition for order dependency was later introduced [21] to represent an order-preserving mapping between *lists* of attributes. In contrast to point-wise ordering, the new definition was distinguishing tuples with attributes in different order, thus having lists of attributes instead of sets.

There are cases where two lists of attributes order each other when taken together. This property is known as *order compatibility* and is denoted with the symbol ~. In Table 1, e.g., it holds that

$$(\texttt{income}, \texttt{savings}) \mapsto (\texttt{savings}, \texttt{income}) \qquad \text{and}$$
$$(\texttt{savings}, \texttt{income}) \mapsto (\texttt{income}, \texttt{savings})$$

and thus `income` ~ `savings`. Another way to see an order compatibility dependency between two columns is that their values are both monotonically non-decreasing when they are considered pairwise.

Dependencies are typically derived from design specifications, from the context of queries or from other known dependencies using inference rules. Discovering dependencies by analyzing the data is a process known as *dependency discovery* [14]. It conceptually requires to check for all potential dependencies if they hold in the database instance under examination, which may be time consuming. Thus, there is interest in developing strategies that limits the number of combinations to be checked. The task becomes even more challenging in the case of order dependencies, where the order of attributes matters, leading to a search space much larger than that of functional dependencies.

In this work we study ways for efficiently discovering order dependencies. We follow a bottom-up approach in which we start by checking short lists of columns and progressively check longer and longer lists. In this process, once an order dependency between two lists of attributes is found not to hold, we prune the search space by ignoring larger lists that include them. In this way, many of the combinations that would have normally been checked are avoided.

We advocate that this whole process can be significantly improved by framing the discovery of order dependencies in the context of order compatibility dependencies. This is based on a recently introduced theorem [21] that established that an order dependency holds if and only if a functional and an order compatibility dependency hold between the two attribute lists of the order dependency. We illustrate in details how the order compatibility dependencies can be exploited to find order dependencies and propose a new algorithm for finding them.

Recently, two algorithms to automatically detect order dependencies in relational data have been proposed: ORDER, proposed by Langer and Naumann [13], and FASTOD proposed by Szlichta et al. [18]. ORDER explores a lattice of order dependency candidates, in a level-wise fashion reminiscent of the TANE algorithm [12]. After building a dependency candidate, ORDER checks its validity against the data and then it applies pruning rules to reduce the search space over the lattice. ORDER has been shown to be incomplete [18], i.e. it does not find the complete set of order dependencies. In particular, this approach is unable to discover dependencies with repeated attributes, for example, the order dependency (`income`, `savings`) ↦ `savings` of Table 1 cannot be discovered. Dependencies of this form, however, may not be inferred from other dependencies and are useful in the case of queries that involve ordering with multi-column indexes. In the example of Table 1, an index over (`income`, `savings`) can be used to simplify the clause `ORDER BY savings`. FASTOD [18] is based on a different axiomatization of order dependencies that allows mapping dependencies between lists of attributes to dependencies between sets of attributes written in a canonical form. In this way, several order dependencies are mapped to the same set-based canonical form. FASTOD explores the space of order dependencies of this set-based canonical form, still retaining the ability to find a complete set of dependencies. While we have reproduced the results presented in the original work, we have found that an implementation error of the original work produces wrong results over simple datasets, this vitiates the validity of their results and the comparison with our approach.

The approach we present in this paper is able to provide a complete set of dependencies that is based on the idea that the whole process of order dependency discovery could be performed through the search of *order compatibility dependencies.* While our approach has a higher worst-case complexity than FASTOD, it outperforms all the state-of-the-art approaches [13, 18] when tested over real datasets.

In particular, our contributions are the following:

- we introduce a definition of minimality for a set of order compatibility dependencies that we show being complete in the sense that it can recover all valid order compatibility dependencies that hold over a given instance of relational data;
- we propose a novel algorithm for finding order dependencies that is complete and can perform the detection of order dependencies in parallel.
- we perform an exhaustive experimental evaluation that shows the performance of our algorithm in comparison with existing works, including a study of its scalability over big datasets and multiple threads.
- we discuss possible solutions for the discovery of the most important order dependencies in the case of dataset that could not be managed (too many columns) in a reasonable amount of time.

The paper is structured as follows: in Section 2 we review the relevant definitions and theorems that formalize the connection between order dependencies and order compatibility dependencies. In Section 3 we prove that order dependency discovery can be guided by order compatibility dependencies without losing completeness. Our novel algorithm is presented in Section 4, while Section 5 contains the discussion of our experimental evaluation. Finally, a thorough review of the related work can be found in Section 6 and we present our conclusions in Section 7.

2 BACKGROUND

To provide the background of our problem, we first review the definition of order dependency and its axiomatization, then we describe the formal relation between order dependencies (ODs), functional dependencies (FDs) and order compatibility dependencies (OCDs).

2.1 Notational Conventions and Definitions

We adopt the notational conventions summarized in Table 2, consistently with the literature [21]. Let $\boldsymbol{R}$ be a relation over the set of attributes $\mathcal{U}$ and $\boldsymbol{r}$ be a table instance of $\boldsymbol{R}$, i.e. a set of tuples under $\boldsymbol{R}$'s schema. A tuple p can be projected over a single attribute A, over a set of attributes $\mathcal{X}$ and over a list of attributes $\mathbf{X}$ by subscripting the tuple as follows: $p_A, p_{\mathcal{X}}, p_{\mathbf{X}}$.

We assume that a total ordering is defined over each of the attributes, denoted $\leq_A$; in the following, however, we will drop the attribute specification and use $\leq$, as the attribute will be always clear from the context. Order dependencies are defined based on the operator $\preccurlyeq$, which is equivalent to a lexicographical ordering over a list of attributes, and is defined by:

Definition 2.1 (operator $\preccurlyeq$). Given a list of attributes $\mathbf{X} := [A|\mathbf{T}]$ and two tuples $p, q \in \boldsymbol{r}$, the operator $\preccurlyeq$ (and its associated operator $\prec$) are defined as follows:

$$
\begin{aligned}
p_{\mathbf{X}} \preccurlyeq q_{\mathbf{X}} \quad &\Leftrightarrow \quad \big(p_A < q_A\big) \vee \\
&\qquad \big((p_A = q_A) \wedge (\mathbf{T} = [\cdot] \vee p_{\mathbf{T}} \preccurlyeq q_{\mathbf{T}})\big) \qquad (1) \\
p_{\mathbf{X}} \prec q_{\mathbf{X}} \quad &\Leftrightarrow \quad p_{\mathbf{X}} \preccurlyeq q_{\mathbf{X}} \wedge p_{\mathbf{X}} \neq q_{\mathbf{X}}
\end{aligned}
$$

The $\preccurlyeq$ operator reproduces the ordering clause `ORDER BY ASC` in SQL [21].

Based on the comparison operator of Definition 2.1, we can introduce the concept of *order dependency* [21].

Definition 2.2 (Order dependency (OD)). Given a relation $\boldsymbol{R}$ and two lists $\mathbf{X}$ and $\mathbf{Y}$, $\mathbf{X} \mapsto \mathbf{Y}$ is an *order dependency* if, for any instance $\boldsymbol{r}$ of $\boldsymbol{R}$ and for every pair of tuples $p, q \in \boldsymbol{r}$, the following implication holds:

$$p_{\mathbf{X}} \preccurlyeq q_{\mathbf{X}} \Rightarrow p_{\mathbf{Y}} \preccurlyeq q_{\mathbf{Y}} \qquad (2)$$

If both $\mathbf{X} \mapsto \mathbf{Y}$ and $\mathbf{Y} \mapsto \mathbf{X}$ hold, we say that $\mathbf{X}$ and $\mathbf{Y}$ are *order equivalent* and we write $\mathbf{X} \leftrightarrow \mathbf{Y}$. If $\mathbf{XY} \leftrightarrow \mathbf{YX}$ we say that $\mathbf{X}$ and $\mathbf{Y}$ are *order compatible* and we write $\mathbf{X} \sim \mathbf{Y}$. We will discuss the latter relation in Section 2.2.

Order dependencies satisfy the set of axioms $\mathcal{J}_{\mathrm{OD}}$, introduced by Szlichta et al. [21], which are reported in Table 3. These axioms are analogous to the Armstrong axioms for functional dependencies [2].

Relations:

- $\boldsymbol{R}$, written as a capital letter in bold italics, is a *relation* over a set of attributes $\mathcal{U}$;
- $\boldsymbol{r}$, written as a lowercase letter in bold italics, is a *table instance* over $\boldsymbol{R}$, i.e. a *set of tuples*;
- Single *attributes* are represented with capital letters: A, B, and C;
- *Tuples* are represented with lowercase letters: p, q, s, and t.

Lists:

- Bold capital letters are *lists* of attributes: $\mathbf{X}$,$\mathbf{Y}$, and $\mathbf{Z}$. they can represent the empty list $[\cdot]$;
- A list is denoted with square brackets $[A,B,C]$. A list $[A|\mathbf{T}]$ is composed by a *head* A and a *tail* $\mathbf{T}$;
- $\mathbf{XY}$ is a shorthand for $\mathbf{X} \circ \mathbf{Y}$, $\mathbf{X}A$ and $A\mathbf{X}$ are shorthands for $\mathbf{X} \circ [A]$ and $[A] \circ \mathbf{X}$ respectively, AB denotes $[A,B]$.

Table 2: Notational conventions

AX1: *Reflexivity*

$$\mathbf{XY} \mapsto \mathbf{X}$$

AX2: *Prefix*

$$\frac{\mathbf{X} \mapsto \mathbf{Y}}{\mathbf{ZX} \mapsto \mathbf{ZY}}$$

AX3: *Normalization*

$$\mathbf{WXYXV} \leftrightarrow \mathbf{WXYV}$$

AX4: *Transitivity*

$$\frac{\mathbf{X} \mapsto \mathbf{Y} \qquad \mathbf{Y} \mapsto \mathbf{Z}}{\mathbf{X} \mapsto \mathbf{Z}}$$

AX5: *Suffix*

$$\frac{\mathbf{X} \mapsto \mathbf{Y}}{\mathbf{X} \leftrightarrow \mathbf{YX}}$$

AX6: *Chain*

$$\frac{\begin{array}{c}\mathbf{X} \sim \mathbf{Y}_1 \\ \forall_{i \in [1,n-1]} \mathbf{Y}_i \sim \mathbf{Y}_{i+1} \\ \mathbf{Y}_n \sim \mathbf{Z} \\ \forall_{i \in [1,n]} \mathbf{Y}_i\mathbf{X} \sim \mathbf{Y}_i\mathbf{Z}\end{array}}{\mathbf{X} \sim \mathbf{Z}}$$

Table 3: The set of axioms $\mathcal{J}_{\mathrm{OD}}$ for order dependencies [21]

2.2 Decomposing Order Dependencies

We show how an order dependency can be decomposed in a pair composed of one functional dependency and one order compatibility dependency.

Functional dependencies. Functional dependencies encode the fact that, in a relation, one attribute determines completely another attribute.

Definition 2.3 (Functional dependency (FD)). Given a relation $\boldsymbol{R}$ and two sets of attributes $\mathcal{X}$ and $\mathcal{Y}$, $\mathcal{X} \rightarrow \mathcal{Y}$ is a *functional dependency* if, for any instance $\boldsymbol{r}$ of $\boldsymbol{R}$ and for every pair of tuples $p, q \in \boldsymbol{r}$, the following implication holds:

$$p_{\mathcal{X}} = q_{\mathcal{X}} \Rightarrow p_{\mathcal{Y}} = q_{\mathcal{Y}} \qquad (3)$$

Order Compatibility Dependencies. Order compatibility dependencies encode the fact that in a relation two lists of attributes show the same monotonicity. If we order either combination of the lists in non-decreasing order, they end up both ordered such their values are both monotonically non-decreasing.

Definition 2.4 (order compatibility dep. (OCD)). Given a relation $\boldsymbol{R}$ and two lists of attributes $\mathbf{X}$ and $\mathbf{Y}$ in $\boldsymbol{R}$, $\mathbf{X} \sim \mathbf{Y}$ is a *order compatibility dependency* if, for any instance $\boldsymbol{r}$ of $\boldsymbol{R}$ and for every pair of tuples $p, q \in \boldsymbol{r}$, the following implications hold:

$$p_{\mathbf{XY}} \preccurlyeq q_{\mathbf{XY}} \Leftrightarrow p_{\mathbf{YX}} \preccurlyeq q_{\mathbf{YX}} \qquad (4)$$

Order dependencies are a stricter relation between two attributes with respect to functional and order compatibility dependencies; furthermore, when an order dependency between two lists of attributes $\mathbf{X}$ and $\mathbf{Y}$ holds, an order compatibility dependency between $\mathbf{X}$ and $\mathbf{Y}$ holds as well. In this sense, order dependencies combine the fact that an attribute functionally determines and have the same monotonicity of another when ordered together.

We present previous results [21] that formally highlight the nature of the relationship between these dependencies, showing that when an order dependency does not hold there are only two possible scenarios called *split* and *swap*. When $\mathbf{X}$ does not order $\mathbf{Y}$, i.e. when the order dependency between $\mathbf{X}$ and $\mathbf{Y}$ does not hold, we write $\mathbf{X} \not\mapsto \mathbf{Y}$.

	A	B
t_1	1	4
t_2	2	5
t_3	3	6
t_4	3	7
t_5	4	1

Table 4: A relational table containing both a split and a swap between the two attributes *A* and *B*.

Split. A split indicates the case where there exists a pair of tuples that have the same values when projected over the attributes X, but have different values over the attributes Y. Formally:

Definition 2.5 (Split). Two tuples s,$t \in \boldsymbol{r}$ form a split over two lists of attributes X, Y iff $s_{\mathrm{X}} = t_{\mathrm{X}}$ but $s_{\mathrm{Y}} \neq t_{\mathrm{Y}}$, or equivalently:

$$\exists s, t \in \boldsymbol{r} \,:\, s_{\mathrm{X}} = t_{\mathrm{X}} \wedge s_{\mathrm{Y}} > t_{\mathrm{Y}}$$

When an OD between two attribute lists is valid, then a FD is valid as well (Theorem 15, [21])

Theorem 2.6 (ODs subsume FDs). *For every instance* $\boldsymbol{r}$ *of relation* $\boldsymbol{R}$*, if the OD* $\mathrm{X} \mapsto \mathrm{Y}$ *holds, then the FD* $\mathcal{X} \to \mathcal{Y}$ *holds.*

Whereas if there is a split between two lists of attributes X and Y, there is no guarantee that ordering data will result in ordered tuples over Y and XY; in other words, both the ODs $\mathrm{X} \mapsto \mathrm{Y}$ and $\mathrm{X} \mapsto \mathrm{XY}$ do not hold. Furthermore, a split falsifies the functional dependency $\mathcal{X} \to \mathcal{Y}$ as well.

The relationship between order and functional dependencies is formalized as follows (Theorem 13, [21]):

Theorem 2.7 (FD and OD correspondence). *For every instance* $\boldsymbol{r}$ *of a relation* $\boldsymbol{R}$*, the functional dependency* $\mathcal{X} \to \mathcal{Y}$ *holds iff* $\mathrm{X} \mapsto \mathrm{XY}$ *holds for all lists* X *that order the attributes of* $\mathcal{X}$ *and all lists* Y *that order the attributes of* $\mathcal{Y}$.

In Table 4, tuples t_3 and t_4 form a split for the attributes A and B, thus $A \not\mapsto B$ and $A \not\mapsto AB$. The functional dependency $A \to B$ is not valid, as well.

Swap. A *swap* indicates the case where there exists a pair of tuples whose values projected over two lists of attributes X and Y are swapped, i.e. they are sorted differently when they are ordered with respect to X or Y. Formally:

Definition 2.8 (Swap). Two tuples s,$t \in \boldsymbol{R}$ form a swap over two list of attributes X and Y, iff $s_{\mathrm{X}} \prec t_{\mathrm{X}}$ but $t_{\mathrm{Y}} \prec s_{\mathrm{Y}}$, or equivalently:

$$\exists s, t \in \boldsymbol{r} \,:\, s_{\mathrm{X}} \prec t_{\mathrm{X}} \wedge s_{\mathrm{Y}} \succ t_{\mathrm{Y}}$$

Swaps between X and Y falsify the ODs of the form $\mathrm{X} \mapsto \mathrm{Y}$, $\mathrm{Y} \mapsto \mathrm{X}$, and $\mathrm{XY} \leftrightarrow \mathrm{YX}$. For example, tuples t_1 and t_5 in Table 4 form a swap for attributes A and B, thus $A \not\mapsto B$, $AB \not\mapsto B$, and $AB \not\mapsto BA$.

Splits and swaps establish a correspondence between order dependencies, functional dependencies and order compatibility dependencies, as in the following theorem (Theorem 15, [21]):

Theorem 2.9 (OD = FD + OCD). $\mathrm{X} \mapsto \mathrm{Y}$ *holds iff* $\mathcal{X} \to \mathcal{Y}$ *(*$\mathrm{X} \mapsto \mathrm{XY}$*) and* $\mathrm{X} \sim \mathrm{Y}$ *(*$\mathrm{XY} \leftrightarrow \mathrm{YX}$*) hold.*

In summary, when an order dependency between two lists of attributes X and Y holds:

- a functional dependency $\mathcal{X} \to \mathcal{Y}$ holds, which implies the absence of split conditions;
- an order compatibility dependency between X and Y holds, which implies the absence of swap conditions.

We exploit this relation to guide our discovery algorithm as established in Section 4.2.

3 ORDER DEPENDENCY THROUGH ORDER COMPATIBILITY

This section introduces the concepts that lay the foundations to our approach.

3.1 Minimality of Discovered Dependencies

Similarly to what has been done for functional dependencies, we introduce the notion of minimality of a set of order compatibility dependencies. In principle, minimality is the property for which a set of dependencies equipped with inference rules can recover all the dependencies that are valid in a given instance of relational data. Axioms AX1-AX6 presented in Table 3 provide inference rules for order dependencies.

The concept of minimality for order dependencies, as for other types of dependencies, has several uses. First of all, it allows reasoning about the validity of pruning rules, e.g., to show that they do not lead to loss of information about valid dependencies in the relation.

Minimality serves also the purpose of compressing information to a manageable size: in fact, if we take a relation with n attributes, in the worst case where each of which is order equivalent to every other, the minimal set of ODs would contain $n - 1$ dependencies ($A \leftrightarrow B, A \leftrightarrow C$, etc.), while the set of all valid dependencies would contain $\mathcal{O}((n!)^2)$ elements: all the possible combinations of attributes on the left-hand and right-hand sides, a prohibitively large number.

Finally, real applications may not need the whole list of dependencies: for example, in knowledge discovery, redundant dependencies do not add value to the properties discovered and too many dependencies can cause the most important ones to be missed; in query optimization, the only useful dependencies are those that can be applied to the queries to be performed.

Any pruning rule applied by a dependency discovery algorithm needs to respect minimality, in the sense that it should allow the recovery of the full set of valid dependencies. A complete algorithm must find at least all *minimal* order dependencies over an instance $\boldsymbol{r}$ of a relation $\boldsymbol{R}$.

To introduce the concept of minimality for ODs, we start by presenting the concepts of closure and equivalence of sets of order dependencies.

Definition 3.1 (Closure). The *closure* of the set of ODs $\mathcal{M}$, denoted $\mathcal{M}^+$, is the set of ODs that are logically implied from $\mathcal{M}$ by the axioms $\mathcal{J}_{\mathrm{OD}} = \{\mathrm{AX1} - \mathrm{AX6}\}$ defined in Table 3.

$$\mathcal{M}^+ = \{\mathrm{X} \mapsto \mathrm{Y} \mid \mathcal{M} \vdash_{\mathcal{J}_{\mathrm{OD}}} \mathrm{X} \mapsto \mathrm{Y}\}$$

Definition 3.2 (Equivalence of sets of ODs). Two sets $\mathcal{M}_1$ and $\mathcal{M}_2$ of order dependencies are *equivalent* if and only if they have the same closure $\mathcal{M}_1^+ = \mathcal{M}_2^+$.

The closure of a set of minimal dependencies is the set of all the dependencies that are valid over $\boldsymbol{r}$. We build this definition first showing which lists of attributes are in minimal form and then when an OCD is minimal. Finally, we prove that our definition of minimality is complete.

Order compatibility dependencies employ attribute lists instead of attribute sets, thus we introduce the concepts of *minimal* attribute list. An attribute list is minimal if it has no embedded order dependency, i.e., the list of attributes is the shortest possible.

Definition 3.3 (minimal attribute list). An attribute list $\mathbf{X}$ is minimal if there is no other list $\mathbf{X}'$ such that:

- $\mathbf{X}'$ is smaller than $\mathbf{X}$, and
- $\mathbf{X}$ and $\mathbf{X}'$ are order equivalent.

For example, the attribute list *ABA* is never minimal, in fact, by the Normalization axiom (AX3) we know that $ABA \leftrightarrow AB$ and *AB* is a shorter list than *ABA*. Instead, *AB* is a minimal attribute list, unless $A \leftrightarrow B$.

An OCD is minimal if both sides are given as minimal attribute lists and there are no repeated attributes:

Definition 3.4 (minimal OCD). An OCD $\mathbf{X} \sim \mathbf{Y}$ is minimal if:

- $\mathbf{X}$ and $\mathbf{Y}$ are minimal attribute lists;
- $\mathcal{X} \cap \mathcal{Y} = \varnothing$.

In the following theorem, we show that OCDs with repeated attributes can be derived from OCDs without repeated attributes:

Theorem 3.5 (Completeness of minimal OCD). *Order compatibility dependencies with repeated attributes can be derived from OCD without repeated attributes.*

Proof. The proof of this theorem is split in three cases:

(1) OCDs of the form $\mathbf{XY} \sim \mathbf{XZ}$ can be derived from $\mathbf{Y} \sim \mathbf{Z}$;
(2) OCDs of the form $\mathbf{XY} \sim \mathbf{MY}$ can be derived from $\mathbf{XY} \sim \mathbf{M}$ and $\mathbf{X} \sim \mathbf{MY}$;
(3) OCDs of the form $\mathbf{XY} \sim \mathbf{MYN}$ can be derived from $\mathbf{X} \sim \mathbf{M}$, $\mathbf{XY} \sim \mathbf{M}$, $\mathbf{X} \sim \mathbf{MY}$ and $\mathbf{XY} \sim \mathbf{MN}$;

which are covered respectively by Theorems 3.10, 3.11 and 3.12 which are presented separately for clarity in Section 3.3. □

The following result extends the Downward Closure theorem (Theorem 12, [21]):

Theorem 3.6. *Downward closure for OCD*

$$\frac{\mathbf{XY} \sim \mathbf{Z}V}{\mathbf{X} \sim \mathbf{Z}}$$

Theorems 3.5 and 3.6 provide the justification for the structure of the search tree used in our approach, as explained in Section 4.2. In particular, we derive the following pruning rule:

Theorem 3.7 (Pruning rule for OCD).

$$\frac{\mathbf{X} \nsim \mathbf{Z}}{\mathbf{XY} \nsim \mathbf{Z}V}$$

Proof. This theorem is the contronominal preposition of Theorem 3.6. □

Theorem 3.5 justifies the reduction of the search space that we highlight in the following section.

3.2 Dimension of the Search Space

The number of valid ODs over a relation $\boldsymbol{R}$ is vast: in fact, if $\boldsymbol{R}$ has n attributes, then all permutations of length k of n elements must be considered both for the left-hand side and the right-hand side of the OD. We address a limitation of the previous work by Langer and Naumann [13] and we show that some order dependencies with repeated attributes cannot be derived from other dependencies without repeated attributes.

For example, in Table 5 (a) we have that $AB \not\mapsto B$, instead in Table 5 (b) we have $AB \mapsto B$ and $A \sim B$. For both tables the dependencies $A \not\mapsto B$ and $B \not\mapsto A$ do not hold, thus the validity of $AB \mapsto B$ and $A \sim B$, in this case, cannot be inferred from shorter ODs.

	(a)		(b)	
	A	B	A	B
t_1	1	4	1	4
t_2	2	5	2	5
t_3	3	6	3	6
t_4	3	7	3	7
t_5	4	1	4	7

Table 5: Two relations where the ODs $A \not\mapsto B$ and $B \not\mapsto A$ do not hold; furthermore in (a) $AB \not\mapsto B$ while in (b) $AB \mapsto B$ and $A \sim B$.

Finding all valid order dependencies thus requires, in principle, the need for checking all combinations $\mathbf{X} \mapsto \mathbf{Y}$ where both $\mathbf{X}$ and $\mathbf{Y}$ can be permutations of length k of the n attributes in $\boldsymbol{R}$ with $1 \leq k \leq n$. If we denote with $S(n)$ the number of k-permutations of n elements we have:

$$S(n) = \lfloor e \cdot n! \rfloor - 1$$

Excluding trivial ODs of the form $\mathbf{X} \mapsto \mathbf{X}$, the number of candidates that needs to be checked would be:

$$C(n) = \big(S(n) - 1\big) \cdot \big(S(n) - 1\big) - \big(S(n) - 1\big) \propto \mathcal{O}\big((n!)^2\big) \quad (5)$$

With $n = 10$, there are more than $97 \cdot 10^{12}$ candidates ODs.

In contrast to general order dependencies, OCDs candidates with repeated attributes, i.e., $\mathbf{X} \rightarrow \mathbf{Y}$ or $\mathbf{X} \sim \mathbf{Y}$ where $\mathcal{X} \cap \mathcal{Y} \neq \varnothing$, are redundant in the sense that their validity can be inferred from the validity of other dependencies of the same type without repeated attributes and with shorter attribute lists.

Theorem 3.8. $\mathbf{X} \sim \mathbf{Y}$ *iff* $\mathbf{XY} \mapsto \mathbf{Y}$

Proof. We prove the implication in each direction:

⇒ By definition $\mathbf{X} \sim \mathbf{Y}$ implies that both the order dependencies $\mathbf{XY} \mapsto \mathbf{YX}$ and $\mathbf{YX} \mapsto \mathbf{XY}$ are valid. By Reflexivity (AX1) $\mathbf{YX} \mapsto \mathbf{Y}$ and thus by Transitivity (AX4) the order dependency $\mathbf{XY} \mapsto \mathbf{Y}$ is valid.

⇐ Conversely, if $\mathbf{XY} \mapsto \mathbf{Y}$, by Suffix (AX5) $\mathbf{XY} \mapsto \mathbf{YXY}$ and Normalization (AX3) $\mathbf{XY} \mapsto \mathbf{YX}$. □

□

This means that ODs of the form $\mathbf{XY} \mapsto \mathbf{Y}$ and OCDs of the form $\mathbf{X} \sim \mathbf{Y}$ are equivalent. We are thus enabled to solve the problem by considering only OCDs without repeated attributes, and thus the dimension of the search space is reduced to $\mathcal{O}(n!)$.

As shown in Theorem 2.9, if $\mathbf{X} \mapsto \mathbf{Y}$ then $\mathbf{X} \sim \mathbf{Y}$ is valid; we can thus derive the following theorem, which will provide the foundation for the pruning rules detailed in Section 4.2.1.

Theorem 3.9.

$$\frac{\mathbf{X} \mapsto \mathbf{Y}}{\mathbf{XZ} \sim \mathbf{Y}} \quad (6)$$

Proof. By the Augmentation theorem [21], $\mathbf{X} \mapsto \mathbf{Y}$ implies $\mathbf{XZ} \mapsto \mathbf{Y}$. By Theorem 2.9 of Section 2.2, $\mathbf{XZ} \mapsto \mathbf{Y}$ implies $\mathbf{XZ} \sim \mathbf{Y}$. □

3.3 Completeness of Minimal OCD

We divide the proof of the completeness of our definition of minimality for OCDs in three parts: first, in the following theorem we prove that attribute lists with repeated attributes at the beginning are redundant:

Theorem 3.10 (Completeness of minimal OCD - 1).

$$\frac{\mathbf{Y} \sim \mathbf{Z}}{\mathbf{XY} \sim \mathbf{XZ}}$$

Proof. By the Shift theorem [21] and the fact that $\mathbf{X} \leftrightarrow \mathbf{X}$ by Reflexivity (AX1):

$$\frac{\begin{array}{c}\mathbf{YZ} \mapsto \mathbf{ZY} \\ \mathbf{X} \leftrightarrow \mathbf{X}\end{array}}{\mathbf{XYZ} \mapsto \mathbf{XZY}}$$

by Normalization (AX3) and Replace [21] $\mathbf{XYXZ} \mapsto \mathbf{XZXY}$. Analogously by the Shift theorem [21] starting from $\mathbf{ZY} \mapsto \mathbf{YZ}$ we obtain $\mathbf{XZXY} \mapsto \mathbf{XYXZ}$. Thus $\mathbf{XYXZ} \leftrightarrow \mathbf{XZXY}$, i.e., $\mathbf{XY} \sim \mathbf{XZ}$ □

The following theorem proves that attribute lists with repeated attributes at the end are also redundant:

Theorem 3.11 (Completeness of minimal OCD - 2).

$$\frac{\begin{array}{c}\mathbf{X} \sim \mathbf{Y} \\ \mathbf{XZ} \sim \mathbf{Y} \\ \mathbf{X} \sim \mathbf{YZ}\end{array}}{\mathbf{XZ} \sim \mathbf{YZ}}$$

Proof.
(1) using $\mathbf{XY} \leftrightarrow \mathbf{YX}$ and $\mathbf{XZY} \leftrightarrow \mathbf{YXZ}$, by Normalization (AX3) $\mathbf{XZY} \leftrightarrow \mathbf{XZYZ}$ and by Replace [21] $\mathbf{YXZ} \leftrightarrow \mathbf{XZYZ}$;
(2) using $\mathbf{XY} \leftrightarrow \mathbf{YX}$ and $\mathbf{XYZ} \leftrightarrow \mathbf{YZX}$, by Normalization (AX3) $\mathbf{YZX} \leftrightarrow \mathbf{YZXZ}$, by Replace [21] $\mathbf{YXZ} \leftrightarrow \mathbf{YZX}$ and by Transitivity (AX4) $\mathbf{YXZ} \leftrightarrow \mathbf{YZXZ}$;

By Transitivity (AX4) $\mathbf{YXZ} \leftrightarrow \mathbf{XZYZ}$ and $\mathbf{YXZ} \leftrightarrow \mathbf{YZXZ}$ imply $\mathbf{XZYZ} \leftrightarrow \mathbf{YZXZ}$, i.e., $\mathbf{XZ} \sim \mathbf{YZ}$. □

Finally, the following theorem proves that attribute lists with repeated attributes in the middle are redundant:

Theorem 3.12 (Completeness of minimal OCD - 3).

$$\frac{\begin{array}{c}\mathbf{X} \sim \boldsymbol{M} \\ \mathbf{XY} \sim \boldsymbol{M} \\ \mathbf{X} \sim \boldsymbol{MY} \\ \mathbf{XY} \sim \boldsymbol{MN}\end{array}}{\mathbf{XY} \sim \boldsymbol{MYN}}$$

Proof.

(1) from $\mathbf{XY} \sim \mathbf{MN}$, by Normalization (AX3) $\mathbf{XYMYN} \leftrightarrow \mathbf{MNXY}$;
(2) from $\mathbf{XY} \sim \mathbf{M}$ and $\mathbf{X} \sim \mathbf{MY}$, using $\mathbf{X} \sim \mathbf{M}$ and Replace [21] we get $\mathbf{MYX} \leftrightarrow \mathbf{XYM}$ and $\mathbf{MXY} \leftrightarrow \mathbf{MYX} \leftrightarrow \mathbf{XYM}$;
(3) from (2), by the Shift theorem [21] with $\mathbf{MY} \leftrightarrow \mathbf{MY}$ and $\mathbf{MNXY} \leftrightarrow \mathbf{XYMMYN}$ we get $\mathbf{MYMNXY} \leftrightarrow \mathbf{MYXYMMYN}$;
(4) by Normalization (AX3) $\mathbf{MYMNXY} \leftrightarrow \mathbf{MYNXY}$;
(5) from $\mathbf{MYXYMMYN}$, using $\mathbf{MYX} \leftrightarrow \mathbf{XYM}$ and Normalization (AX3) we get $\mathbf{XYMYMYN}$ and finally $\mathbf{XYMYN}$;

From points (3), (4) and (5) we finally get $\mathbf{MYNXY} \leftrightarrow \mathbf{XYMYN}$, i.e., $\mathbf{XY} \sim \mathbf{MYN}$. □

4 THE OCDDISCOVER ALGORITHM

We present now the details of our algorithm, called ocddiscover, by first examining its search strategy to cover all the possible combinations and then presenting an implementation in pseudocode.

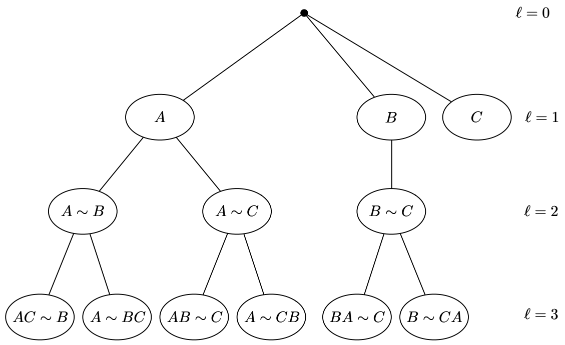


Figure 1: Permutation tree for a table with $n = 3$ attributes.

4.1 Column Reduction

Given that the search space grows with the number of columns, we start our discovery algorithm focusing on the columns showing special properties and we perform two operations: (a) the removal of constant columns; (b) the reduction of order-equivalent columns. The dependencies provided by these operations are an integral part of the results provided by our algorithm.

Removal of constant columns. Constant columns generate a huge amount of ODs; in fact, over an instance $\boldsymbol{r}$ a constant column C is ordered by any other attribute list $\mathbf{X}$.[1] Thus, we remove all constant columns and we collect the corresponding dependencies.

Reduction of order-equivalent columns. Order-equivalent columns as $A \leftrightarrow B$ describe a relation in which both the directions of the order dependency hold. By the Replace theorem (Theorem 6, [21]), we can replace any order dependency where A appears with another dependency with any instance of A replaced with B, that is:

$$\mathbf{X}A\mathbf{Y} \mapsto \mathbf{M}A\mathbf{N} \Leftrightarrow \mathbf{X}B\mathbf{Y} \mapsto \mathbf{M}B\mathbf{N}$$

We check any combination of order-equivalent dependencies, i.e. for all $A, B \in \mathcal{U}$ we verify the validity of $A \mapsto B$ and $B \mapsto A$, and we build the equivalence classes of columns using the Tarjan algorithm [25].

We choose a representative from each of these equivalence classes; we then remove all other columns. We store this information to later recover the redundant dependencies.

4.2 Search Tree

We use a breadth-first search strategy for identifying OCD relations in $\boldsymbol{r}$; in this way, shorter minimal dependencies are discovered before longer ones. At the first level, we consider the set of all pairs of single attributes. Given that OCDs are commutative, we build this set by enumerating all the attributes with $A_1, A_2, \ldots, A_n$ and taking all the pairs (A_i, A_j) such that $\{(A_i, A_j) \mid A_i, A_j \in \mathcal{U}, i < j\}$.

Figure 1 shows the tree $\mathcal{T}$ of generated candidates for a relation $\boldsymbol{r}$ with attributes $\mathcal{U} = \{A,B,C\}$ where all possible candidates are generated.

Each OCD candidate $\mathbf{X} \sim \mathbf{Y}$ is checked for order compatibility; we are then confronted with two possibilities:

[1] If C is constant column, the following property holds for any tuple $\boldsymbol{p}$, $\boldsymbol{q}$ in any instance $\boldsymbol{r}$ of $\boldsymbol{R}$: $p_{\mathbf{X}} \leq q_{\mathbf{X}} \Rightarrow p_C = q_C$, where the second part of the implication is always true by definition of constant column.

- if the candidate is not order compatible, we do not generate any other candidate starting from it, as stated by Theorem 3.7 in Section 3.1;
- if the candidate is order compatible, we generate new OCD candidates in the following way: for each attribute not already present in the OCD, for each $A \in \mathcal{U} \setminus \{\mathcal{X} \cup \mathcal{Y}\}$, we add it to the right of each attribute list, i.e. $\mathbf{X}A \sim \mathbf{Y}$ and $\mathbf{X} \sim \mathbf{Y}A$, then we apply further pruning rules as explained in Section 4.2.1.

4.2.1 Pruning Rules. When we find a new OCD $\mathbf{X} \sim \mathbf{Y}$, we further check the validity of the OD $\mathbf{X} \mapsto \mathbf{Y}$ and $\mathbf{Y} \mapsto \mathbf{X}$.

From Theorem 3.9 we derive the following pruning rules:

- if $\mathbf{X} \mapsto \mathbf{Y}$ we do not generate the candidates of the form $\mathbf{XZ} \overset{?}{\sim} \mathbf{Y}$, i.e. the left-hand children candidates of $\mathbf{X} \sim \mathbf{Y}$ are pruned;
- if $\mathbf{Y} \mapsto \mathbf{X}$, we do not generate the candidates of the form $\mathbf{X} \overset{?}{\sim} \mathbf{YZ}$, i.e. the right-hand children candidates of $\mathbf{X} \sim \mathbf{Y}$ are pruned;

If both dependencies are valid we prune all the subtree from the given OCD candidate.

With reference to Figure 1, if the order compatible dependency $A \sim B$ is valid:

- if $A \mapsto B$, the children candidate $AC \overset{?}{\sim} B$ is pruned;
- if $B \mapsto A$, the children candidate $A \overset{?}{\sim} BC$ is pruned.

4.2.2 Parallelizability. ocddiscover explores the tree of candidates breadth-first. Each branch of the tree can be visited independently since each OCD candidate is independent from the others; furthermore, a candidate is generated for each level if and only if its father in the tree was a valid order dependency. We exploit this structure to parallelize the execution of ocddiscover by assigning candidates from different branches to different queues; each queue is then processed by a different thread. With reference to Algorithm 1, if we have K cores available, we can have K independent subtrees $\mathcal{T}_\ell^1, \mathcal{T}_\ell^2, \ldots, \mathcal{T}_\ell^K$ (lines 7 — 12) each one containing the OCD candidates belonging to a different branch of the tree. The number of queues used by the algorithm can be chosen as a run-time parameter provided by the user.

4.3 Order Checking

One of the most important steps in our approach is the check of order compatibility candidates.

Single check. Given an OCD candidate in the form $\mathbf{X} \overset{?}{\sim} \mathbf{Y}$, we need to verify if it holds. The general definition of order compatibility states that $\mathbf{X} \sim \mathbf{Y} \equiv \mathbf{XY} \leftrightarrow \mathbf{YX}$, i.e. $\mathbf{X}$ and $\mathbf{Y}$ are order compatible if $\mathbf{XY} \mapsto \mathbf{YX}$ and $\mathbf{YX} \mapsto \mathbf{XY}$; however, with the following theorem, we can reduce the problem of checking the validity of an OCD to a single check.

Theorem 4.1. $\mathbf{XY} \mapsto \mathbf{YX}$ *is valid iff* $\mathbf{X} \sim \mathbf{Y}$.

Proof.

$\Rightarrow$ we have to prove that $\mathbf{XY} \mapsto \mathbf{YX} \Rightarrow \mathbf{YX} \mapsto \mathbf{XY}$. If, by contradiction, $\mathbf{YX} \not\mapsto \mathbf{XY}$, then:

$$\exists\, p,q \mid p_{\mathrm{YX}} \le q_{\mathrm{YX}} \Rightarrow p_{\mathrm{XY}} > q_{\mathrm{XY}} \tag{7}$$

thus since $p_{\mathrm{XY}} > q_{\mathrm{XY}}$ we can distinguish two cases:

– if $p_{\mathrm{X}} > q_{\mathrm{X}}$, we can conclude that:

$$q_{\mathrm{XY}} < p_{\mathrm{XY}} \Rightarrow q_{\mathrm{YX}} > p_{\mathrm{YX}}$$

thus $\mathbf{XY} \not\mapsto \mathbf{YX}$;

– if $p_{\mathrm{X}} = q_{\mathrm{X}} \wedge p_{\mathrm{Y}} > q_{\mathrm{Y}}$, we always obtain a contradiction with the condition expressed in Eq. 7;

thus both $\mathbf{XY} \mapsto \mathbf{YX}$ and $\mathbf{YX} \mapsto \mathbf{XY}$ are valid and $\mathbf{X} \sim \mathbf{Y}$;

$\Leftarrow$ by definition if $\mathbf{X} \sim \mathbf{Y}$ then both $\mathbf{YX} \mapsto \mathbf{XY}$ and $\mathbf{XY} \mapsto \mathbf{YX}$ are valid;

□

Checking with Indexes. To compute if an order dependency candidate holds, we sort the relation by the left-hand side attributes of the candidate. We build an index that contains only the position of each tuple in the order in which it appears. Then, we iterate over the tuples following this index, and we check if the attributes on the right-hand side violate the ordering, specifically if we detected a pair of tuples forming a swap. We break the detection loop as soon as we find a violation and return true otherwise. In the worst case, the number of comparisons to be made is $O(m + m \log m)$ where m is the total number of tuples in the relation.

`NULL` Values. In real-world datasets, and in many of the test cases that will be analyzed in Section 5, data contains `NULL` values, which destroy the total ordering assumption because they can not be compared with the other values. We use the standard SQL semantics given by `set ansi nulls ON`, i.e. `NULL` equals `NULL`, and `NULLS FIRST` for sorting.

4.4 Description of the Algorithm

Algorithm 1 is the main algorithm that implements our approach. The input is given by the instance of relational data $\boldsymbol{r}$ and its set of attributes $\mathcal{U}$.

Algorithm 1 ocddiscover

```
Input: r: a relational instance
Input: U: the set of attributes associated with r
 1: function ocddiscover(r, U)
 2:     $\ell \leftarrow 2$
 3:     $\mathcal{U}' \leftarrow$ columnsReduction($\mathcal{U}$)
 4:     $\mathcal{T}_1 \leftarrow \{(A,B) \mid A,B \in \mathcal{U}', B > A\}$
 5:     while $\mathcal{T}_\ell \neq \emptyset$ do                          ▷ Main loop
 6:         $\mathcal{T}_{\ell+1} \leftarrow \emptyset$
 7:         for each $(\mathrm{X},\mathrm{Y}) \in \mathcal{T}_\ell$ do
 8:             if checkCandidate(XY, YX, r) then
 9:                 emit $\mathrm{X} \sim \mathrm{Y}$
10:                 $\mathcal{T}_{\ell+1} \leftarrow \mathcal{T}_{\ell+1} \cup$ generateNextLevel($\mathrm{X}, \mathrm{Y}, \mathcal{U}'$)
11:             end if
12:         end for
13:         $\ell \leftarrow \ell + 1$
14:     end while
15: end function
```

In Line 3, function columnReduction() (not shown here) is called to apply the operations described in Section 4.1: the removal of constant attributes and the reduction of order-equivalent columns. This function prints a list of order-equivalence relations and a list of constant attributes and returns a reduced set of attributes $\mathcal{U}'$ where (a) the constant columns are removed; and (b) for each class of order-equivalent attributes, one representative is chosen.

The initial tree of OCD candidates is built in Line 4; by the commutativity of OCDs, only half of the combinations are added.

Figure 1 shows that the second level of the tree ($\ell = 2$) contains only the initial candidates $A \overset{?}{\sim} B$, $A \overset{?}{\sim} C$ and $B \overset{?}{\sim} C$.

Then, the algorithm continues with the main loop where each OCD candidate $\mathbf{X} \overset{?}{\sim} \mathbf{Y}$ is tested against the data in $\boldsymbol{r}$ in the form of an OD candidate $\mathbf{XY} \overset{?}{\mapsto} \mathbf{YX}$ using the function checkCandidate(), which is described in Algorithm 2.

The function checkCandidate() iterates over the index built on the left-hand side of the candidate with generateIndex() and checks that the values over the attributes in the right-hand side are in the same order. The loop is terminated early if a violation is detected.

Algorithm 2 checkCandidate

```
Input: X, Y: an OD candidate
Input: r: the instance of relational data
Output: true if $\mathbf{X} \mapsto \mathbf{Y}$, false otherwise.
function checkCandidate(X, Y, r)
    $l_r \leftarrow$ len(r)
    $index \leftarrow$ generateIndex(X,Y,r)
    for $i \leftarrow 1$ to $l_r - 1$ do
        for each $A \in \mathbf{Y}$ do
            if $r[index[i], A] > r[index[i+1], A]$ then
                return false
            else if $r[index[i], A] < r[index[i+1], A]$ then
                return true
            end if
        end for
    end for
    return true
end function
```

If the candidate is a valid OCD, we emit it as a result and generate the new candidates through generateNextLevel(), which is described in Algorithm 3.

The function generateNextLevel() builds a set containing all the OCD candidates of the form $\mathbf{X}A \sim \mathbf{Y}$ and $\mathbf{X} \sim \mathbf{Y}A$, where A is an attribute that does not already belong to the lists $\mathbf{X}$ and $\mathbf{Y}$, this corresponds to creating a new level of the tree presented in Section 4.2 using the pruning rules of Section 4.2.1. The function further checks if the ODs $\mathbf{X} \overset{?}{\mapsto} \mathbf{Y}$ or $\mathbf{Y} \overset{?}{\mapsto} \mathbf{X}$ hold. If so, it applies the pruning rules and returns the remaining candidates. We emit the valid ODs found in Lines 9 and 16 of Algorithm 3.

The new candidates are added to the queue of candidates to check for the next level in line 10 of Algorithm 1. The loop terminates when there are no candidates left.

5 EXPERIMENTAL EVALUATION

In this section, we evaluate the results of our approach, with particular emphasis on its scalability in the number of rows and columns, and we compare it with previous work on order dependency detection. We analyze the results of ocddiscover over 6 real-world datasets and 5 synthetic datasets.

Our algorithm is implemented in Java 1.7 and is designed to work on the Metanome data profiling framework [15] as a multi-threaded program.

All experiments were run on a `i686` Intel Xeon E52440 2.40 GHz machine with 12 cores in hyper-threading and 128 GB RAM, over a Linux kernel `v4.15.0`. The execution environment is a 64-bit Oracle JDK version `1.8.0_171`, with the JVM heap space limited to 110 GB.

Algorithm 3 generateNextLevel

```
Input: X, Y: an OCD candidate
Input: $\mathcal{U}'$: the set of reduced attributes of relation R
Output: C: the candidate OCD generated from $\mathbf{X} \sim \mathbf{Y}$
 1: function generateNextLevel(X, Y, $\mathcal{U}'$)
 2:     $C \leftarrow \varnothing$
 3:     $\mathcal{A}^+ \leftarrow \mathcal{U}' -$ set(X) $-$ set(Y)
 4:     if ¬checkCandidate(X, Y, r) then                 ▷ $\mathbf{X} \not\mapsto \mathbf{Y}$
 5:         for each $A \in \mathcal{A}^+$ do
 6:             $C.add((\mathbf{X}A, \mathbf{Y}))$
 7:         end for
 8:     else                                             ▷ $\mathbf{X} \mapsto \mathbf{Y}$
 9:         emit $\mathbf{X} \mapsto \mathbf{Y}$
10:     end if
11:     if ¬checkCandidate(Y, X, r) then                 ▷ $\mathbf{Y} \not\mapsto \mathbf{X}$
12:         for each $A \in \mathcal{A}^+$ do
13:             $C.add((\mathbf{X}, \mathbf{Y}A))$
14:         end for
15:     else                                             ▷ $\mathbf{Y} \mapsto \mathbf{X}$
16:         emit $\mathbf{Y} \mapsto \mathbf{X}$
17:     end if
18:     return C
19: end function
```

5.1 Datasets

We use the datasets provided by the Information Systems Group of Hasso-Plattner-Institut.[2] These datasets are the same used by the previous work on order dependency discovery by Langer and Naumann [13]. We have also created three simple additional synthetic datasets, called `YES`, `NO`, and `NUMBERS` created to highlight the differences of our approach with previous works. In particular, `YES` and `NO` reproduce, respectively, the examples in Tables 5 (a) and 5 (b), while `NUMBERS` is shown in Table 7.

Table 6 presents the datasets and their properties; for each dataset, the table reports: the dataset name, the number of rows $|\boldsymbol{r}|$, the number of attributes $|\mathcal{U}|$, the number of functional dependencies discovered by the fastfds algorithm [26] $|\mathcal{F}_d|$, the number of ODs discovered $|\mathcal{O}_d|$ by order. For fastod we provide: (a) the number of FDs discovered $|\mathcal{F}_d|$, (b) the number of ODs discovered $|\mathcal{O}_d|$. For ocddiscover we provide: (a) the number of OCDs discovered $|\mathcal{O}_c|$, which are missed by order [13], since they are order dependencies with repeated attributes; (b) the number of ODs discovered $|\mathcal{O}_d|$; and (c) the total number of dependency candidates checked during the execution of the algorithm.

Execution time is averaged across 5 independent runs and we set a time threshold at 5 hours. When the time limit is reached, for order, and fastod we are unable to present the number of dependencies discovered so far, while for ocddiscover we report the number of dependencies discovered and the number of checks made until the limit.

5.2 Comparison with Previous Work

We discuss the results of the extensive comparison with the previous state-of-the-art algorithms for detecting order dependencies. The code used for the comparison has been provided by the respective authors.[3]

[2] https://hpi.de/naumann/projects/repeatability/data-profiling/fd-algorithms.html

[3] the source code for order is available at: https://hpi.de/naumann/projects/repeatability/data-profiling/fds.html#c168192 while the source of fastod has been provided to us by the authors through direct communication.

Dataset properties			FASTFDS [26]	ORDER [13]		FASTOD [18]			OCDDISCOVER			
Dataset	$\lvert \boldsymbol{r} \rvert$	$\lvert \mathcal{U} \rvert$	$\lvert \mathcal{F}_d \rvert$	$\lvert \mathcal{O}_d \rvert$	**time (ms)**	$\lvert \mathcal{O}_d \rvert$	$\lvert \mathcal{F}_d \rvert$	**time (ms)**	$\lvert \mathcal{O}_d \rvert$	$\lvert \mathcal{O}_c \rvert$	**time (ms)**	**checks**
DBTESMA	250,000	30	89,571	$-^*$	5 h*	400	89,571	4,641,485	138	0	**337,289**	4,118
DBTESMA_1K	1000	30	11,099	$-^*$	5 h*	30	11,099	5,799	138	0	**1,835**	4,118
FLIGHT_1K	1,000	109	$-^*$	$-^\dagger$	$-^\dagger$	$-^*$	$-^*$	5 h*	3,216,069*	29,404,555*	5 h*	7,473,951
HEPATITIS	155	20	8,250	0	**182**	32,717	8,250	211,903	0	5	361	556
HORSE	300	29	128,727	31	46,907	$-^*$	$-^*$	5 h*	31	7	**618**	1,185
LETTER	20,000	17	61	0	**1,215**	$-^*$	$-^*$	5 h*	0	0	1,720	272
LINEITEM	6,001,215	16	$-^*$	1	**982,075**	$-^*$	$-^*$	5 h*	1	0	1,039,517	255
NCVOTER_1K	1,000	19	758	18	**796**	2,333	758	90,000	18	1	872	338
NO	5	2	1	0	323	0	1	24	0	0	**4**	2
YES	5	2	0	0	329	1	0	28	1	1	**3**	2
NUMBERS	7	4	4	0	331	6	4	325	0	0	**28**	12

Table 6: Datasets and execution statistics for the OCDDISCOVER, ORDER [13], and FASTOD [18] algorithms. "*" indicates that the execution has reached the time limit of 5 hours, while "†" that it has exceeded the memory limit of 110GB. When the time limit is reached, for OCDDISCOVER we present partial results.

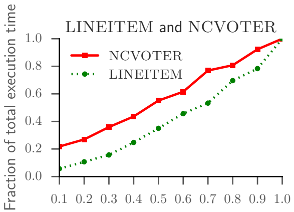


Figure 2: Normalized execution times for row scalability.

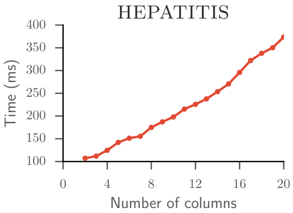


Figure 3: Execution times for column scalability for HEPATITIS.

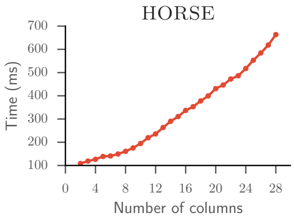


Figure 4: Execution times for column scalability for HORSE.

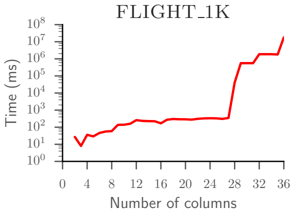


Figure 5: Execution times for column scalability for FLIGHT_1K.

A	B	C	D
1	3	1	1
2	2	3	2
2	3	2	2
2	5	2	2
3	1	2	3
4	4	4	2
4	5	3	2

Table 7: NUMBERS dataset.

To compare our algorithm with ORDER and FASTOD we need to transform OCDs back to ODs. In fact, in a relation $\boldsymbol{R}$ over attributes A, B and C where $A \leftrightarrow B$, $A \sim C$ and $B \sim C$, the set of OCDs $\mathcal{O} = \{A \sim B, A \sim C, B \sim C\}$ is minimal following Definition 3.4. However, COLUMNSREDUCTION() would discover the order-equivalence $A \leftrightarrow B$ and choose one attribute as a representative, e. g. A. Thus OCDDISCOVER would return as valid OCDs only $\mathcal{O} = \{A \sim C\}$. From this information, we infer the remaining dependency $B \sim C$ using the axioms $\mathcal{J}_{\text{OD}}$. We perform this expansion and compare the results produced by OCDDISCOVER, ORDER and FASTOD. The function performing the expansion is not shown in Algorithm 1, but the times reported in Table 6 include it. This step did not impact the running time of OCDDISCOVER.

5.2.1 Comparison with Langer and Naumann [13]. For ORDER, dependencies are considered to be *completely non-trivial*, if their left- and right-hand side attribute lists are disjoint. However, we argue that limiting the discovery of order dependencies to candidates where the left-hand side and the right-hand side are completely disjoint gives incomplete results. Our algorithm, instead, is complete.

Following Theorem 3.8 in Section 3.2, order dependencies of the form $\mathbf{XY} \mapsto \mathbf{Y}$ can be inferred from order compatibility dependencies of the form $\mathbf{X} \sim \mathbf{Y}$. Note that these dependencies have repeated attributes between its left- and right-hand sides.

We show the difference between our approach and previous work with the YES and NO datasets. As reported in Table 6, the ORDER algorithm does not find any order dependency in either of the YES and NO datasets. OCDDISCOVER, instead, finds correctly the order compatibility dependency $A \sim B$, i.e., the order equivalence $AB \leftrightarrow BA$, in YES.

Our approach detects all the dependencies found by ORDER, and additional dependencies on HEPATITIS, NCVOTER_1K, and HORSE. For FLIGHT_1K and NCVOTER_ALLC we found several dependencies but we were not able to compare the results with ORDER because the latter does not report the discovered dependencies when the time limit is reached.

Provided that the order compatibility dependencies found by OCDDISCOVER are translated to the corresponding OD in the form $\mathbf{XY} \mapsto \mathbf{Y}$, OCDDISCOVER effectively discovers a minimal set of ODs even following the definition of minimality provided by Langer and Naumann [13].

When a candidate dependency is found to be false, pruning rules are applied. For this reason, notwithstanding the factorial dimension of the search space, datasets with several columns, such as datasets HEPATITIS and HORSE, are successfully and completely tested. When pruning cannot be applied, the generation of candidates grows – e.g., more than 7 million candidates are generated in FLIGHT_1K. For this dataset, OCDDISCOVER detects more than 32 million ODs. In this particular case, the number of checks is smaller than the number of discovered dependencies because we also count the dependencies inferred from constant and order-equivalent columns reported by the COLUMNSREDUCTION() function.

Furthermore, Table 6 shows that using order compatibility dependencies does not hinder the performance of the detection. In all dataset tested, the performance of our algorithm with respect to the execution time is comparable to order; in some cases, we obtain significant speedups up to a factor of 75, e.g. in HORSE.

5.2.2 Comparison with Szlichta et al. [18]. As shown in Table 6, ocddiscover and fastod compute a different number of order dependencies. We claim that this difference, which also affects also the results published in [18], is due to an implementation error in the code. Table 7 presents an instance of a relational table where the implementation of fastod we received finds several order dependencies that are not actually present in the data, e.g. $[B] \mapsto [AC]$. Other datasets were also affected by this issue, but, unfortunately, we were not able to isolate and resolve the root cause of this incorrect behavior. In addition, fastod considers all columns as if they contain data of type `String`, thus using a lexicographical ordering, while order and ocddiscover perform type inference over the datasets provided, and use the natural ordering for real and integer numbers. We have also implemented for ocddiscover the possibility of forcing lexicographical ordering, i.e., treat all data as if they were of type `String`, but we do not report these results since this change does not affect the execution time of our approach.

Furthermore, the scalability experiments reported by Szlichta et al. [18] used trimmed-down versions of the datasets. For the row scalability experiments, the datasets were reduced to 1,000 rows, while for the column scalability experiments columns were chosen at random. For this reason, we report in Table 6 both the DBTESMA and DBTESMA_1K datasets, which are respectively the full dataset with 250,000 rows, and a trimmed-down version with the first 1,000 rows. For column scalability, we tested ocddiscover and fastod against HEPATITIS, FLIGHT_1K, and NCVOTER_1K, which were used in full, so we were able to compare the performance of the two algorithms.

As shown in Table 6, ocddiscover gains significant speed-ups. This highlights the fact that, while the worst-case complexity of ocddiscover is $(O)(|\boldsymbol{r}|!)$, which is greater than that of fastod, $O(2^{|\boldsymbol{r}|})$, the execution time depends on the actual number of dependencies contained in the dataset.

5.3 Performance

In the following, we analyze the scalability of our approach with respect to the number of rows, the number of columns, and the number of parallel threads used.

5.3.1 Scalability in the number of rows. We performed our analysis on the synthetic dataset LINEITEM with 6,001,215 rows and 16 columns. We also test the algorithm on the NCVOTER dataset. This dataset has 938,084 rows and 94 columns, but since our algorithm did not terminate on the full datasets, we consider only a subset of 20 randomly chosen columns. Figure 2 shows the results for the scalability experiment on LINEITEM and NCVOTER. Ten samples of the original dataset have been created, ranging from 10% to 100% of the rows with a step of 10%. Five repetitions have been performed for each of the samples and the average is reported. Variance is very small and thus not shown.

The experiments show that the algorithm scales almost linearly with the number of rows, and it is able to find a complete set of OCDs over datasets with millions of rows.

The execution time would be expected to grow log-linearly with respect to the number of rows, due to the indexing phase; but an increasing number of rows may correspond to a smaller number of dependencies; thus, the pruning phase could reduce the number of checks to be computed.

Previous work, instead, has shown the ability to scale linearly on the number of rows performing the check of dependency candidates with sorted partitions computed from the data. This method could have been re-implemented in our approach as well, but it would have been out of the scope of this paper.

5.3.2 Scalability in the number of columns. Scalability over columns is the key challenge in detecting dependencies in relational data since in many cases the dimension of the search scales with the number of columns.

We choose the HORSE and HEPATITIS datasets, that are well-suited to evaluate the influence of an increasing number of columns, given that their execution completes. We also consider FLIGHT_1K that has a very high number of columns and does not terminate.

The evaluation approach is as follows: we start with two random columns from each dataset, and we incrementally add more randomly-chosen columns, until the total number of columns in the dataset is reached.

To avoid skewing the results, we generate 50 samples of each dataset with the process described above and we run our algorithm over these samples. We average the execution time of ocddiscover of each sample with c columns over all the 50 samples.

Figures 3 and 4 show the results of the column scalability experiment on the HEPATITIS and HORSE datasets.

Figure 5 shows how the algorithm behaves when the number of dependencies discovered in the data grows on a single run. Times on the y-axis are in logarithmic scale. ocddiscover is very susceptible to columns that are quasi-constant, i.e., attributes with very few distinct values, but not constant. In this case, ocddiscover cannot eliminate these columns during the column-reduction phase. As argued in Section 4.1, constant columns are ordered by any other attribute; quasi-constant columns are associated with a large number of valid OCDs and consequently the size of the tree to be explored grows enormously.

In fact, the slowdown corresponding to the sample with 28 columns in Figure 5 is caused by the addition of a column with 3 distinct values. This column appears on the right-hand side of more than 94% of the dependencies found in that sample.

5.3.3 Scalability over parallel threads. As described in Section 4.2.2, ocddiscover can be run over multiple threads.

Figure 6 shows the results of the multithreading scalability experiments on the LETTER, LINEITEM, and DBTESMA datasets. On the y-axis times are normalized to the runtime over a single thread, which is the maximum for each case. Table 8 reports the executions times.

	Time (s) vs number of threads			
Dataset	**1**	**2**	**4**	**8**
LETTER	5.7	4.6	3.6	3.4
LINEITEM	2,848.8	1,770.0	1,243.5	1,040.0
DBTESMA	2,228.9	1,240.0	686.0	414.0

Table 8: Execution time of ocddiscover versus number of threads.

As it is shown in Figure 6, using multiple parallel threads shortens the execution time of our dependency discovery algorithm.

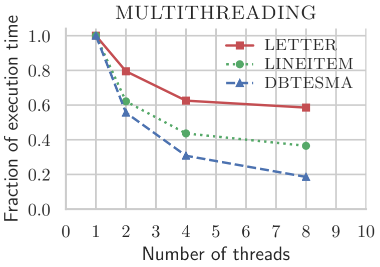


Figure 6: Execution times for `LETTER` (red straight line), `LINEITEM` (green dotted line), `DBTESMA` (blue dashed line), when executed over multiple threads, normalized over the runtime over a single thread.

The amount of this improvement varies based on the characteristics of each dataset and in particular, depends on the of OCD candidates checked.

If we compare `LETTER` and `LINEITEM`, we see that while the number of checks performed on each dataset is comparable (272 for `LETTER` versus 255 for `LINEITEM`), the number of their rows differ by several orders of magnitude (~ $20k$ lines for `LETTER`, versus ~ $6M$ for `LINEITEM`). This implies that checking the validity of an OCD candidate for `LINEITEM` takes longer than checking a candidate for `LETTER`. Thus the relative gain when splitting the work over multiple threads is greater for `LINEITEM`.

Instead, comparing `LINEITEM` and `DBTESMA` we can see that for the latter dataset the number of checks performed is much higher, thus the workload of candidate checking can be spread over multiple threads, leading to greater relative improvements.

5.4 Quasi-constant Columns

The quasi-constant column scenario is challenging for our algorithm. We further develop the idea of measuring how varied the values in a column are by measuring its *entropy*.

Definition 5.1 (Entropy). Given an attribute $A \in \mathcal{U}$ of an instance $\boldsymbol{r}$ of a relation $\boldsymbol{R}$, the entropy of A is defined as:

$$H(A) = -\sum_{[a]} p_{[a]} \log\left(p_{[a]}\right) \tag{8}$$

where $[a]$ are the equivalence classes of distinct values in A and $p_{[a]}$ is the probability of extracting an instance of class a, computed as the relative frequency of instances class $[a]$ over the total number of tuples in $\boldsymbol{r}$:

$$p_{[a]} = \frac{|\{t \in \boldsymbol{r} \mid t_A \in [a]\}|}{|\boldsymbol{r}|}$$

For constant columns there is only one equivalence class and $p_{[a]} = 1$, thus $H(A) = 0$. If all values are distinct, for each $[a]$, $|[a]| = 1$ and $p_{[a]} = 1/|\boldsymbol{r}|$:

$$H(A) = -\sum_{[a]} \frac{1}{|\boldsymbol{r}|} \log\left(1/|\boldsymbol{r}|\right) = \log\left(|\boldsymbol{r}|\right)$$

We test the idea that progressively less diverse columns cause the slowdown of OCDDISCOVER by taking the `FLIGHT_1K` dataset and running it over multiple samples build with the following criteria: we calculate the entropy of each column in `FLIGHT_1K` and then we build samples of increasing size in the number of columns by adding progressively the columns with decreasing entropy, i.e., we start with the columns with the greatest number of distinct values and we progressively add columns with less distinct values. Eventually, the constant columns are added.

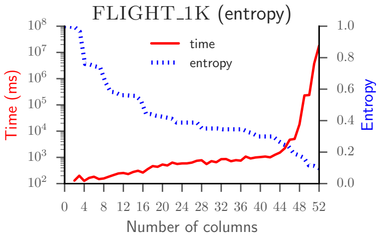


Figure 7: Execution times (red straight line) for the `FLIGHT_1K` when adding columns of decreasing entropy (blue dotted line). On the y-axis, times (left-hand side) are in logarithmic scale, entropy (right-hand side) is normalized with respect to the maximum value over the dataset.

The result of the execution on OCDDISCOVER over this set of samples is reported in Figure 7. With 50 columns the OCDDISCOVER completes in 4 minutes, adding the 51st column the execution time grows by an order of magnitude to over 1 hour. With the addition of the 52nd column, the algorithm reaches the time limit of 5 hours. The 50th, 51st, and 52nd columns have respectively 4, 2, and 2 distinct values respectively.

With respect to applications, this insight could be exploited to develop algorithms that return results for the most diverse columns, which can be the most interesting with respect to other properties of the data such as unique column combinations (UCC). Detection of unique column combinations is usually performed to find primary keys candidates that may be also interesting candidates from the point of view of ordering and query optimization.

In summary, the column scalability experiments show that OCDDISCOVER can find a complete set of ODs over datasets with tens of columns. Furthermore, OCDDISCOVER can be easily adapted to perform the detection over a set of interesting columns, where the interestingness of an attribute can be determined providing a function measuring the properties chosen by the user.

6 RELATED WORK

Functional Dependencies: Theory, Discovery and Applications. Literature on functional dependencies is vast [2, 12, 14, 16]. Applications of functional dependencies span several fields from query optimization [6], to data cleaning [5], to data quality management [8]. Given this variety of applications, several algorithms for the discovery of functional dependencies have been developed. The first algorithm to be proposed was TANE [12], which has served as inspiration for many subsequent efforts [14, 16]. Research on better functional dependency discovery algorithms is still ongoing [15]. Functional dependencies have been extended in several ways: from conditional functional dependencies [4], to approximate (or partial) functional dependencies [14].

Order Dependencies Theory and Applications. In the 1980's, Ginsburg and Hull were the first to consider the idea of analyzing orderings between the attributes of a relation as a kind of dependency [9–11]. They introduced the concept of point-wise ordering [11], that is a relation where a set of attributes orders another set of attributes. Recently, Szlichta et al. introduced the concept of order dependency [21], which is the one used throughout this paper. Order dependencies are defined over lists of attributes, and can be formalized in a similar way to functional dependencies [21, 23]. In this paper we focused on dependencies where the attributes are all ordered in the same direction, also called "unidirectional" order dependencies; however order dependencies can be generalized to "polarized" or "bidirectional" ODs where a different direction of the ordering can be specified for each attribute on either side of the dependency [20]. One of the main theoretical problems concerning dependencies in relational data is the problem of inference. For order dependencies this problem is shown to be co-NP-complete [24].

Applications of Order Dependencies. Notwithstanding their recent theoretical formulation, order dependencies have been already used in several applications, such as query optimization.

Sorting is a fundamental database operation. Since the seminal works [10], research has focused on developing optimization strategies for dealing with queries with an `ORDER BY` clause [17]. Order dependencies can be used for this purpose, as it has been shown with an implementation of a query optimizer in IBM DB2. Optimizing queries with order dependencies yields significant speedups in execution times over the well-known TPC-DS benchmark and on queries taken from real-world scenarios [22].

Discovery Algorithms for Order Dependencies. These applications are driving the need for discovering order dependencies in existing datasets. The first work proposing an algorithm to solve this problem is the one by Langer and Naumann [13]. Their approach, called ORDER, follows the path of TANE, computing the potential order dependency candidates from the permutations of attributes and traversing the lattice in a level-wise, bottom-up manner. Pruning rules are applied to reduce the number of candidates to check, with the caveat of eliminating only redundant relations that can be later inferred from the dependencies discovered together with the axioms. However, as shown in this paper, this approach does not consider all the possible order dependency candidates, discarding repeated attributes. While this gives a significant advantage in execution time, reducing the worst-case time complexity of the algorithm to $O(|\boldsymbol{r}|!)$, its major drawback is the possibility of losing completeness. More recently Szlichta et al. [18] proposed an algorithm called FASTOD that is complete and faster than ORDER. This algorithm exploits a novel polynomial mapping that transforms ODs with lists of attributes into canonical forms of ODs that are established between sets of attributes. FASTOD has exponential worst-case time complexity, $O(2^{|\boldsymbol{r}|})$, in the number of attributes. Recently, this work was extended in order to discover bidirectional order dependencies [19].

7 CONCLUSIONS

In this work, we presented a novel method for discovering order dependencies (ODs). We based our approach on the fact that an order dependency is valid if and only if both a functional dependency (FD) and an order compatibility dependency (OCD) are valid. Thus, we designed a novel and efficient algorithm – called OCDDISCOVER – where the search for ODs is guided by checking the validity of OCDs. Our approach outperforms existing two state-of-the-art algorithms, ORDER [13] and FASTOD [18] with respect to ORDER, we are complete, meaning that we detect order dependencies that are ignored. We have shown that these dependencies cannot be inferred by other detected dependencies. While the worst-case complexity of OCDDISCOVER is greater than FASTOD, the execution time on real datasets depends on the actual number of dependencies found, thus our algorithm outperforms FASTOD. Furthermore, we presented an extensive set of experiments that illustrate that our approach can be executed in parallel over multiple threads. We have also suggested that considering the entropy of attributes can lead to further developments in discovery the most interesting order dependencies. As a future work, we would like to consider dynamic inputs, where additional rows and columns may be added at runtime.

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling relational data: a survey. *VLDBJ* 24, 4 (2015), 557–581.

[2] William Ward Armstrong. 1974. Dependency Structures of Data Base Relationships.. In *IFIP congress*, Vol. 74. Geneva, Switzerland, 580–583.

[3] Sören Auer, Jens Lehmann, Axel-Cyrille Ngonga Ngomo, and Amrapali Zaveri. 2013. Introduction to linked data and its lifecycle on the web. In *Reasoning Web. Semantic technologies for intelligent data access*. Springer, 1–90.

[4] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2007. Conditional functional dependencies for data cleaning. In *ICDE*. IEEE, 746–755.

[5] Xu Chu, Ihab F Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *ICDE*. IEEE, 458–469.

[6] Hugh Darwen and C Date. 1989. The role of functional dependencies in query decomposition. *Relational Database Writings* 1991 (1989), 133–154.

[7] Jirun Dong and Richard Hull. 1982. Applying approximate order dependency to reduce indexing space. In *SIGMOD*. ACM, 119–127.

[8] Wenfei Fan and Floris Geerts. 2012. Foundations of data quality management. *Synthesis Lectures on Data Management* 4, 5 (2012), 1–217.

[9] Seymour Ginsburg and Richard Hull. 1981. Ordered Attribute Domains in the Relational Model. In *XP2 Workshop on Relational Database Theory, June 22-24 1981, The Pennsylvania State University, PA, USA*.

[10] Seymour Ginsburg and Richard Hull. 1983. Order dependency in the relational model. *Theoretical computer science* 26, 1 (1983), 149–195.

[11] Seymour Ginsburg and Richard Hull. 1986. Sort sets in the relational model. *Journal of the ACM (JACM)* 33, 3 (1986), 465–488.

[12] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal* 42, 2 (1999), 100–111.

[13] Philipp Langer and Felix Naumann. 2016. Efficient order dependency detection. *The VLDB Journal* 25, 2 (2016), 223–241.

[14] Jixue Liu, Jiuyong Li, Chengfei Liu, Yongfeng Chen, et al. 2012. Discover dependencies from data – A review. *IEEE Transactions on Knowledge and Data Engineering* 24, 2 (2012), 251–264.

[15] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. 2015. Data profiling with Metanome. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1860–1863.

[16] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional dependency discovery: An experimental evaluation of seven algorithms. *PVLDBE* 8, 10 (2015), 1082–1093.

[17] David Simmen, Eugene Shekita, and Timothy Malkemus. 1996. Fundamental techniques for order optimization. *ACM SIGMOD Record* 25, 2 (1996), 57–67.

[18] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. 2017. Effective and Complete Discovery of Order Dependencies via Set-based Axiomatization. *PVLDB* 10, 7 (2017), 721–732. https://doi.org/10.14778/3067421.3067422

[19] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. 2018. Effective and complete discovery of bidirectional order dependencies via set-based axioms. *VLDB J.* 27, 4 (2018), 573–591. https://doi.org/10.1007/s00778-018-0510-0

[20] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. 2012. Chasing Polarized Order Dependencies.. In *AMW*. 168–179.

[21] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. 2012. Fundamentals of order dependencies. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1220–1231.

[22] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, Wenbin Ma, Weinan Qiu, and Calisto Zuzarte. 2014. Business-Intelligence Queries with Order Dependencies in DB2.. In *EDBT*. 750–761.

[23] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. 2013. Axiomatic System for Order Dependencies. In *Proceedings of the 7th Alberto Mendelzon International Workshop on Foundations of Data Management*. http://ceur-ws.org/Vol-1087/shortpaper1.pdf

[24] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. 2013. Expressiveness and complexity of order dependencies. *PVLDBE* 6, 14 (2013), 1858–1869.

[25] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. https://doi.org/10.1137/0201010

[26] Catharine Wyss, Chris Giannella, and Edward Robertson. 2001. Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract. In *DaWaK*. Springer, 101–110.

Scalable Kernel Density Estimation-based Local Outlier Detection over Large Data Streams*

Xiao Qin[1], Lei Cao[2], Elke A. Rundensteiner[1] and Samuel Madden[2]
[1]Department of Computer Science, Worcester Polytechnic Institute
[2]CSAIL, Massachusetts Institute of Technology
[1]{xqin,rundenst}@cs.wpi.edu [2]{lcao,madden}@csail.mit.edu

ABSTRACT

Local outlier techniques are known to be effective for detecting outliers in skewed data, where subsets of the data exhibit diverse distribution properties. However, existing methods are not well equipped to support modern high-velocity data streams due to the high complexity of the detection algorithms and their volatility to data updates. To tackle these shortcomings, we propose local outlier semantics that operate at an abstraction level by leveraging kernel density estimation (KDE) to effectively detect local outliers from streaming data. A strategy to continuously detect top-N KDE-based local outliers over streams is designed, called **KELOS** – the first linear time complexity streaming local outlier detection approach. The first innovation of KELOS is the abstract kernel center-based KDE (*a*KDE) strategy. *a*KDE accurately yet efficiently estimates the data density at each point – essential for local outlier detection. This is based on the observation that a cluster of points close to each other tend to have a similar influence on a target point's density estimation when used as kernel centers. These points thus can be represented by one abstract kernel center. Next, the KELOS's inlier pruning strategy early prunes points that have no chance to become top-N outliers. This empowers KELOS to skip the computation of their data density and of the outlier status for every data point. Together *a*KDE and the inlier pruning strategy eliminate the performance bottleneck of streaming local outlier detection. The experimental evaluation demonstrates that KELOS is up to 6 orders of magnitude faster than existing solutions, while being highly effective in detecting local outliers from streaming data.

1 INTRODUCTION

Motivation. The growth of digital devices coupled with their ever-increasing capabilities to generate and transmit live data presents an exciting new opportunity for real time data analytics. As the volume and velocity of data streams continue to grow, automated discovery of insights in such streaming data is critical. In particular, finding *outliers* in streaming data is a fundamental task in many online applications ranging from fraud detection, network intrusion monitoring to system fault analysis. In general, outliers are data points situated away from the majority of the points in the data space. For example, a transaction of a credit card in a physical location far away from where it has normally been used may indicate fraud. Over 15.4 million U.S residents were victims of such fraud in 2016 according to [3]. On the other hand, as more transactions take place in this new location, the previous transaction may appear legitimate as it begins to conform to the increasingly expected behavior exemplified by the new data. Thus, in streaming environments, it is critical to design a mechanism to efficiently identify outliers by monitoring the statistical properties of the data relative to each other as it changes over time.

State-of-the-Art. To satisfy this need, several methods [20, 21] have been proposed in recent years that leverage the concept of *local outlier* [6] to detect outliers from data streams. The local outlier notion is based on the observation that real world datasets tend to be skewed, where different subspaces of the data exhibit different distribution properties. It is thus often more meaningful to decide on the outlier status of a point based on its difference with the points in its local neighborhood as opposed to using a global density [9] or frequency [5] cutoff threshold to detect outliers [11]. More specifically, a point x is considered to be a *local outlier* if the *data density* at x is low *relative* to that at the points in x's local neighborhood. Unfortunately, existing streaming local outlier solutions [20, 21] are not scalable to high volume data streams. The root cause is that they measure the data density at each point x based on the point's distance to its k nearest neighbors (kNN). Unfortunately, kNN is very sensitive to data updates, meaning that the insertion or removal of even a small number of points can cause the kNN of many points in the dataset to be updated [20]. Since the complexity of the kNN search [6] is quadratic in the number of the points, significant resources may be wasted on a large number of unnecessary kNN re-computations. Therefore, those approaches suffer from a high response time when handling high-speed streams. For example, it takes [20, 21] 10 minutes to process just 100k tuples as shown by their experiments. Intuitively, kernel density estimation (KDE) [26], an established probability density approximation method, could be leveraged for estimating the data density at each point [16, 23, 27]. Unlike kNN-based density estimation that is sensitive to data changes, KDE estimates data density based on the statistical properties of the dataset. Therefore, it tends to be more robust to gradual data changes and thus a better fit for streaming environments. However, surprisingly, to date no method has been proposed that utilizes KDE to tackle the local outlier detection problem for data streams.

Challenges. Effectively leveraging KDE in the streaming context comes with challenges. Similar to kNN search, the complexity of KDE is quadratic in the number of points [26]. While the computational costs can be reduced by running the density estimation on kernel centers sampled from the input dataset, sampling leads to a trade-off between accuracy and efficiency. Although a low sampling rate can dramatically reduce the computational complexity, one must be cautious because the estimated data density at each point may be inaccurate due to an insufficient number of kernel centers. On the other hand, a higher sampling rate will certainly lead to a better estimation of the data density. However, the computational costs of KDE increase quadratically with more kernel centers. With a large number of kernel centers, KDE

*This work was partly supported by the National Science Foundation under grants IIS-1815866 and IIS-1560229.

would be at risk of becoming too costly to satisfy the stringent response time requirements of streaming applications. Due to this accuracy versus efficiency trade-off, to the best of our knowledge, no method has successfully adapted KDE to function efficiently on streaming data to date.

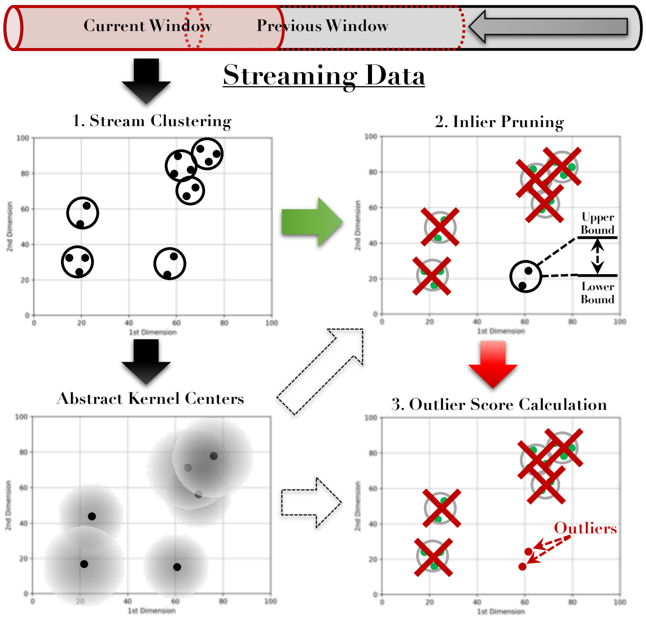


Figure 1: An illustration of KELOS approach.

Proposed Solution. In this work, we propose a scalable KDE-based strategy (Fig. 1) for detecting top-N local outliers over streams, or in short **KELOS**. KELOS provides the first practical solution for local outlier detection on streaming data. Our key contributions are given below.

- New KDE-based semantics are proposed for the continuous detection of the top promising outliers from data streams. This establishes a foundation for the design of a scalable streaming local outlier detection method.
- A notion of the *abstract kernel center* is introduced to solve the accuracy versus efficiency trade-off of KDE. This leverages the observation that kernel centers close to each other tend to have a similar strength of influence on the densities at other points. These nearby points thus can be clustered together and considered as one abstract kernel center weighted by the amount of data it represents. Compared to the traditional sampling-based KDE, our strategy achieves accurate density estimation using *much fewer kernel centers*. This in turn speeds up the quadratic complexity process of local density estimation. This notion of abstract kernel centers by itself could be applied to a much broader class of density estimation related stream mining tasks beyond local outlier detection.
- Unlike existing techniques [20, 21], which detect outliers by computing the data density and then the outlierness score for every data point, KELOS quickly prunes the vast majority of the data points that have no chance to become outliers. The more expensive KDE method itself is only used thereafter to evaluate the remaining much smaller number of potential outlier candidates.
- Putting these optimizations together, we obtain the first linear time complexity streaming local outlier detection approach that outperforms the state-of-the-arts by up to 6 orders of magnitudes in speed confirmed by our experiments on real world datasets.

2 PRELIMINARIES

2.1 Local Outlier

Given a point x_i, it is a *local outlier* if the *data density* at x_i (e.g. inverse of average distances to its kNN) is significantly lower than the densities at x_i's neighbors.

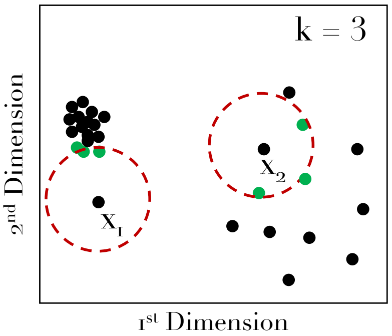
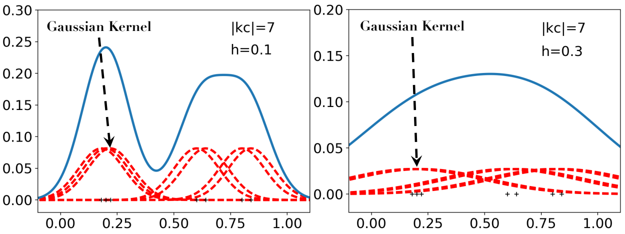


Figure 2: Local outlier detection using local densities.

As illustrated in Fig. 2, although the densities at x_1 and x_2 are both low, the density at x_1 is quite different than the densities at the locations of its neighbors. However, the densities at the neighbors of x_2 is similar to x_2. Therefore, x_1 is more likely to be an outlier than x_2 due to its *relatively low density* in contrast to those at its neighbors. Therefore, conceptually measuring a point x_i's status of being a **local outlier** corresponds to the two-steps:
(1) Estimate the density at x_i and the densities at its neighbors;
(2) Compute the outlierness score of x_i based on the deviation of the density at x_i in contrast to those at its neighbors.

2.2 Kernel Density Estimation



(a) Bandwidth h = 0.1 **(b) Bandwidth h = 0.3**

Figure 3: An example of univariate kernel density estimator using Gaussian kernel with different bandwidth.

Kernel density estimation (KDE) is a non-parametric method to estimate the probability density function (PDF) of a dataset $X = \{x_1, \cdots, x_n\}$. Given a point x_i, the kernel density estimator of X computes how likely x_i is drawn from X. This computed probability can be interpreted as the "data density" at x_i in X.

The density at x_i in X is:

$$\tilde{f}(x_i) = \frac{1}{m} \sum_{j=1}^{m} K_h(|x_i - kc_j|). \tag{1}$$

Kernel Centers. $kc_j \in \mathbb{KC}$ where $1 \leq j \leq m$ are called the kernel centers in the estimator. Typically, kc_j is a point sampled from X. The selected set of kernel centers must be sufficient to represent the data distribution of X [26]. Each kernel center kc_j carries a kernel function K_h. The *density contribution* by a kernel center kc_j is calculated based upon the distance from kc_j to the target point x_i. The density at x_i is estimated by the average density contribution by all kernel centers. For example, in Fig. 3(a), there are 7 kernel centers. Each of them carries a kernel function (red dashed curve). The shape of the overall density function across all kernels is represented by the blue solid line. Given a dataset X with n points and m kernel centers, the time complexity of computing the densities of all $x_i \in X$ is $O(nm)$.

Kernel Function. A wide range of kernel functions can be used in kernel density estimation [26]. The most commonly used ones are the *Gaussian* and *Epanechnikov* kernel functions [11]. In this study, we adopt the *Gaussian* kernel:

$$K_{gauss}(u) = \frac{1}{(\sqrt{2\pi})h} e^{(-\frac{1}{2}\frac{u^2}{h^2})}, \tag{2}$$

$$K_{epanechnikov}(u) = \frac{3}{4h}\left(1 - \frac{u^2}{h^2}\right), \tag{3}$$

where u represents the distance from a kernel center kc_j to the target point x_i and h is an important smoothing factor, called *bandwidth*. The *bandwidth h* controls the smoothness of the shape of the estimated density function. The greater the value h, the smoother the shape of the density function $\tilde{f}$. As shown in Figs. 3(a) and (b), using the same set of kernel centers but different bandwidth values, the estimated PDFs (the blue lines) are significant different from each other. Therefore, an appropriate bandwidth is critical to the accuracy of the density estimation.

Balloon Kernel. In *balloon kernel* [30], when estimating the density at a target point x_i, only the k nearest kernel centers of x_i denoted as kNN(x_i,$\mathbb{KC}$) are utilized in the estimator. This provides each point x_i a customized kernel density estimator that adapts to the distribution characteristics of x_i's surrounding area, hence also called *local density*. Therefore, Balloon kernel fits the local outlier that detects outliers based on the local distribution properties as shown in [23]:

$$\tilde{f}(x_i) = \frac{1}{k}\sum_{j=1}^{k} K_h(|x_i - kc_j|) \text{ where } kc_j \in k\text{NN}(x_i, \mathbb{KC}). \tag{4}$$

Multi-dimensional Kernel. We adopt the *product kernel* [24] as the form of the kernel function. The product kernel is typical in density estimation for multi-dimensional data. Given a d dimensional target point x_i and a kernel center kc_j, for each dimension l the product kernel (Eq. 5) first computes the density contribution of kc_j to x_i based on the distance on dimension l. The final density contribution by kc_j to x_i is the product of the density contributions by kc_j on all dimensions, so called product kernel. As we will show in Sec. 7, this creates opportunities for the design of constant time density update operation in the streaming context. Moreover, product kernel allows the bandwidth to be customized for each dimension, resulting in more accurate estimation [24].

$$\tilde{f}(x_i) = \sum_{j=1}^{k}\prod_{l=1}^{d} K_{h^l}(|x_i^l - kc_j^l|) \text{ where } kc_j \in k\text{NN}(x_i, \mathbb{KC}). \tag{5}$$

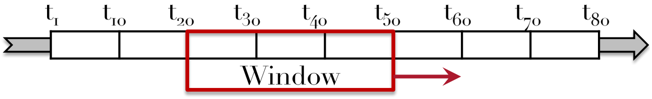


Figure 4: A data stream in the form of sliding windows.

3 PROPOSED OUTLIER SEMANTICS

Next, we propose our semantics of KDE-based streaming local outliers. We first introduce the notion of top-N local outliers that captures the most extreme outliers in the input dataset. We then apply this concept to sliding windows to characterize outliers in data streams.

3.1 Top-N KDE-based Local Outliers

We first define a new outlierness measure, KDE-based local outlierness measure (KLOME). The goal is to reduce the computation costs of the existing KDE-based outlier semantics [23], while still effective in detecting outliers.

Definition 3.1. **KLOME.** Given a set of data points $X = \{x_1, \cdots, x_n\}$ and a set of kernel centers $\mathbb{KC} = \{kc_1, \cdots, kc_m\}$, the KLOME score of a target point $x_i \in X$ is defined as KLOME(x_i) = *z-score*($\tilde{f_b}(x_i)$, $\{\tilde{f_b}(kc_j) \mid \forall kc_j \in k\text{NN}(x_i, \mathbb{KC})\}$).

Here *z-score*$(s, S) = (s$ - $\bar{S})/\sigma_S$ [33] indicates how many standard deviations a value s is above or below the mean of a set of values S. In this definition, KLOME(x_i) measures how different the local density at x_i is from the average local density at x_i's nearest kernel centers denoted as kNN(x_i, $\mathbb{KC}$). A negative KLOME score of a target point x_i indicates that the local density at x_i is smaller than the local densities at its neighbors' locations. The *smaller* the KLOME score of a point x_i is, the *larger* the possibility that x_i is an outlier.

The key property of our KLOME semantics is that the density at x_i is compared against the densities at its kNN in the kernel center set $\mathbb{KC}$ (kNN(x_i, $\mathbb{KC}$)) instead of its actual kNN in the dataset.

The intuition is as below. The kernel centers sufficient to recover the distribution of the original dataset can well capture every local phenomenon. The density at x_i is estimated based on its location *relative to* the selected kernel centers kc_j $\in k$NN(x_i, $\mathbb{KC}$). Naturally kc_j can serve as the local neighbor of x_i in the density deviation computation of x_i (*z-score*). In other words, kNN(x_i, $\mathbb{KC}$) effectively models the local neighborhood of a point x_i. This in turn significantly reduces the computational complexity compared to working with x_i's kNNs in the much larger input dataset X.

Next, we define the top-N KDE-based local outlier:

Definition 3.2. Given a set of data points $X = \{x_1, \cdots, x_n\}$ and a count threshold N, the **top-N KDE-based local outliers** are a set of N data points, denoted by *Top*-KLOME(X, N) such that $\forall x_i \in$ *Top*-KLOME(X, N) and $\forall x_j \in X \setminus$ *Top*-KLOME(X, N), KLOME(x_i) $\leq$ KLOME(x_j).

3.2 Local Outlier Detection in Sliding Window

We work with periodic sliding window semantics, illustrated in Fig. 4, commonly adopted to model a finite substream of interest from the otherwise infinite data stream [4]. Such semantics can be either time or count-based. Each data point x_i has an associated time stamp denoted by $x_i.time$. The window size and slide size of a stream S are denoted as $S.win$ and $S.slide$ correspondingly.

Each window W_c has a starting time $W_c.T_{start}$ and an ending time $W_c.T_{end} = W_c.T_{start} + S.win$. Periodically the current window W_c slides, causing $W_c.T_{start}$ and $W_c.T_{end}$ to increase by $S.slide$ respectively. For count-based windows, a fixed number (count) of data points corresponds to the window size $S.win$. Accordingly $S.slide$ is also measured by number of data points. $\mathcal{S}^{W_c}$ denotes the set of data points falling into the current window W_c. The local outliers are then always detected in the current active window W_c. An outlier in the current window might turn into an inlier in the next window.

Next, we define the problem of continuously detecting Top-KLOME(X, N) over sliding windows:

Definition 3.3. Given a stream S, a window size $S.win$, a slide size $S.slide$, and an integer N, continuously compute Top-KLOME$(\mathcal{S}^{W_c}, N)$ over sliding windows.

Next, we introduce our KELOS framework for supporting continuous local outlier detection over windows stream with our proposed KLOME semantics.

4 THE KELOS FRAMEWORK

KELOS framework, depicted in Fig. 5, consists of three main components, namely, **stream data abstractor**, **density estimator** and **outlier detector**.

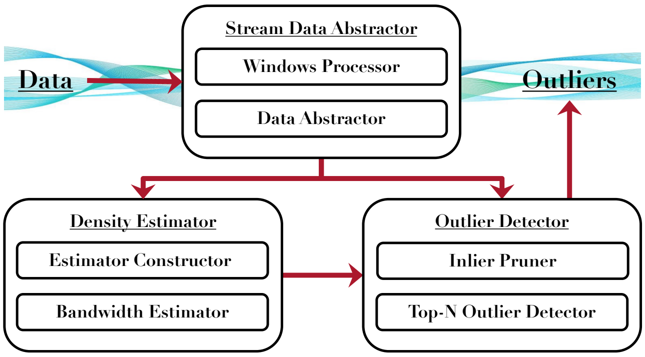


Figure 5: The KELOS framework.

The **stream data extractor** is composed of the *window processor* and the *data abstractor*. The *window processor* feeds the latest data that falls into the current stream window to the system. By leveraging a lightweight stream clustering approach, the *data abstractor* dynamically organizes the data points in each window into evolving clusters based on their affinity. It then generates and maintains statistics that reflect the key distribution properties of each cluster. These statistics are essential to performing density estimation and the inlier pruning during outlier detection.

The *estimator constructor* in the **density estimator** builds kernel density estimators utilizing *abstract kernel centers* where each abstract kernel center represents one data cluster. The *bandwidth estimator* leverages the statistics associated with each cluster to approximate an optimal bandwidth for each density estimator customized for each target. The constant time complexity of the bandwidth estimator ensures that the bandwidth can be continuously updated online to best fit the data.

The **outlier detector** continuously computes the top-N outliers, that is, the N points with the highest outlierness scores. It avoids having to compute the density and the outlierness score for each and every point by pruning clusters that as a whole do not have a chance to contain any outlier. This leverages the *stable density* property of a tight cluster and the characteristics of local outliers (Sec. 6.1).

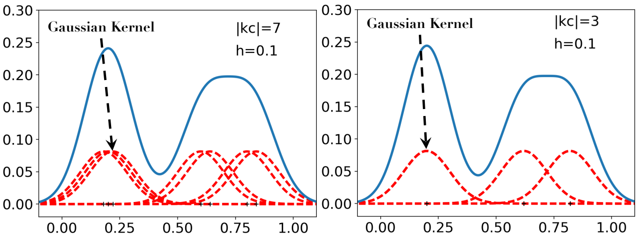
In Sec. 5, we present the key strategies of our **density estimator**, namely abstract kernel center-based KDE. In Sec. 6, we introduce the techniques core to our **outlier detector**. It efficiently identifies and prunes the points that are guaranteed not to be outliers. Finally, in Sec. 7, we introduce our **stream data abstractor**. It features a low complexity dual-purpose clustering algorithm that continuously constructs data clusters, while at the same time generating the statistics needed to support the density estimator and outlier detector.

5 DENSITY ESTIMATOR

In this section, we propose our abstract kernel center-based KDE strategy (*a*KDE). It solves the problem of accurately yet efficiently estimating the density at a given point. In contrast to the traditional sampling-based KDE approach [26], our density estimation is performed on top of a set of clusters (Fig. 1) that succinctly summarize the distribution characteristics of the dataset. This approach is inspired by our *abstract kernel center* observation below.

Abstract Kernel Center Observation. In KDE, the density at a given point x_i is determined by the *additive influences* of the kernel centers, while the influence from one center kc_j is determined by the distance between kc_j and x_i. The centers close to each other tend to have similar influence on the target point x_i. Using them redundantly instead of representing them as a whole perplexes the density estimation by unnecessarily enlarging the center space. To obtain succinct while informative representatives as kernel centers, KELOS first performs a lightweight clustering that groups close points together. The centroid of the cluster weighted by the cluster's data cardinality, called abstract kernel center (AKC) is then selected as a kernel center to perform density estimation.

Fig. 6(b) shows an example estimation using the abstract kernel centers. The original 7 points in Fig. 3(a) are abstracted into three clusters. The estimations (blue line) in Fig. 6(b) with 3 centers and Fig. 6(a) using all 7 points as kernel centers are similar.



(a) All points as kernel centers **(b) Abtract kernel centers**

Figure 6: Local kernel density estimator.

On the performance side, real world data sets tend to be skewed. Therefore, typically most points can be clustered into a small number of tight clusters. Correspondingly, the number of the abstract kernel centers tends to be much smaller than the number of sampled kernel centers that would be sufficient to represent the overall data distribution of the dataset. Since the bottleneck of local density estimation is on the computation of the k nearest kernel centers for each to be estimated point x_i, the small number of abstract kernel centers promises to reduce the complexity of the successive density estimation process.

Furthermore, the abstract kernel centers allow us to use a small k while establishing a diversified neighborhood – hence a comprehensive density estimator for each point. This not only reduces the complexity of the kNN search and kernel density computation, but also alleviates the problem of selecting an appropriate k because of the reduced range of possible k.

Definition 5.1. Given a stream window $S^{W_c} = \{x_1, \cdots, x_n\}$, the abstract kernel centers of S^{W_c} are a set of pairs $\mathbb{AKC}(S^{W_c}) = \{\langle c_{c_1}, |c_1|\rangle, \cdots, \langle c_{c_m}, |c_m|\rangle\}$, where c_{c_i} $(1 \leq i \leq m)$ corresponds to the centroid of the respective data cluster c_i and $|c_i|$ the number of points in c_i. Here $\bigcup_{i=1}^{m} c_i = S^{W_c}$ and $\forall i, j, i \neq j$ $c_i \cap c_j = \emptyset$.

Weighted Kernel Density Estimator. Intuitively, each abstract kernel center represents the centroid of a cluster of points close to each other along with the data cardinality of this cluster. Utilizing these abstract kernel centers, we construct a weighted kernel density estimator [13], where the kernel centers correspond to the centroids in $\mathbb{AKC}(S^{W_c})$ (the first component of $\mathbb{AKC}$) and the weight corresponds to the cardinality of the data cluster represented by the centroid (the second component). Therefore, the weighted kernel density estimator reflects the distribution characteristics of the entire dataset by utilizing only a small number of kernel centers. The formula is shown below:

$$\tilde{f}_{\mathbb{AKC}(S^{W_c})}(x_i) = \sum_{j=1}^{k} \omega(c_{c_j}) \prod_{l=1}^{d} K_{h^l}(|x_i^l - c_{c_j}^l|) \tag{6}$$

where

$$\omega(c_{c_j}) = \frac{|c_j|}{\sum_{m=1}^{k} |c_m|}, \tag{7}$$

and $c_{c_m} \in k\text{NN}(x_i, \mathbb{AKC}(S^{W_c}))$. Here $\tilde{f}_{\mathbb{AKC}(S^{W_c})}(x_i)$ in Eq. 6 corresponds to a weighted product kernel estimator that computes the local density at x_i and $k\text{NN}(x_i, \mathbb{AKC}(S^{W_c}))$ corresponds to the k nearest centroids of x_i in the abstract kernel centers.

Bandwidth Estimation. One additional step required to make the weighted kernel density estimator work is to establish an appropriate bandwidth for the kernel on each dimension. Here we show that the data driven strategy introduced in [25] (Eq. 8) can be efficiently applied here by leveraging the abstract kernel centers.

$$h^l = 1.06\sigma^l k^{-1/(d+1)}. \tag{8}$$

In Eq. 8, d denotes the data dimension. σ^l denotes the weighted standard deviation of the kernel centers on the lth dimension computed by:

$$\sigma^l = \sqrt{\sum_{m=1}^{k} \omega(c_{c_m})(c_{c_m}^l - \mu^l)^2}, \tag{9}$$

where

$$\mu^l = \frac{\sum_{m=1}^{k} \omega(c_{c_m}) c_{c_m}^l}{k}, \tag{10}$$

and $c_{c_m} \in k\text{NN}(x_i, \mathbb{AKC}(S^{W_c}))$.

Efficiency. The time complexity of KDE is $O(nm)$, where n is number of data points and m is the number of kernel centers. Since *a*KDE dramatically reduces the number of kernel centers, it significantly speeds up the KDE computation. On the other hand, data clustering introduces extra computation overhead. In this work, we apply the low complexity *micro-clustering* [2] strategy that processes each point only once. This overhead is significantly outweighed by the saved KDE computation costs. Therefore, overall *a*KDE is much faster than the traditional KDE – as shown in Sec. 8.2.

6 OUTLIER DETECTOR

Our outlier detector fully utilizes the data clusters produced for *a*KDE by leveraging our *stable density* observation described below.

Stable Density Observation. Data points in a tight cluster are close to each other. Therefore, they tend to share the same kernel centers and have similar local densities. By the definition of local outliers, the outlierness score of a point x depends on the relative density at x in contrast to those at its neighbors. Therefore, these points tend to have similar outlierness scores. Since outliers only correspond to small subset of points with the highest outlierness scores, it is likely that most of the data clusters do not contain any outlier.

Assume we have a method to approximate the highest (upper bound) and lowest (lower bound) outlierness scores for the points in each data cluster. Using these bounds, the data clusters that have no chance to contain any outlier can be quickly identified and pruned from outlier candidate set without any further investigation. More specifically, if the upper bound outlierness score of a data cluster c_i is smaller than the lower bound outlierness score of a data cluster c_j, then the whole c_i can be pruned (under the trivial premise that c_j has at least N points). This is so because there are at least N points in the dataset whose outlierness scores are larger than any point in c_i.

Leveraging this observation, we now design an efficient local outlier detection strategy. The overall process is given in Alg. 1. We first rank and then prune data clusters based on their upper KLOME score bounds. As shown in Sec. 3.1, a small KLOME score indicates large outlier possibility. Therefore, the upper KLOME bound corresponds to the lower outlierness score bound. Similarly, the lower KLOME bound corresponds to the upper outlierness score bound. Therefore, if the lower KLOME bound of a cluster c_i is higher than the upper KLOME bound of another cluster c_j, all points in c_i can be pruned immediately. Only the clusters with a small lower KLOME bound (large outlierness score upper bound) are subject to further investigation. The densities and KLOME scores at the data point-level are computed only for the data points in these remaining clusters. Finally, the top-N results are selected among these points by maintaining their KLOME scores in a priority queue.

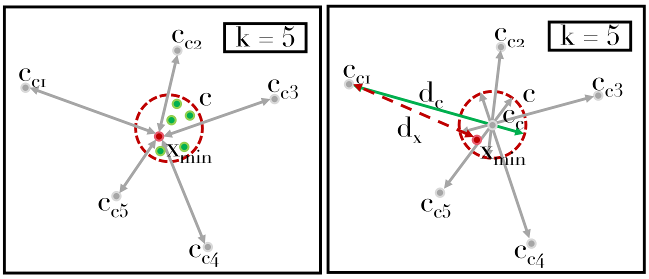
6.1 Bounding the KLOME Scores

Next, we present an efficient strategy to establish the upper and lower KLOME bounds for each given data cluster.

By Def. 3.1, the KLOME score of a point x_i corresponds to *z-score*($\tilde{f}(x_i)$, S), where S refers to the local densities at x_i's kernel centers. Since the points in the same cluster c_i typically share the same kernel centers, the data point $x_{min} \in c_i$ with the minimal density determines the lower bound KLOME score of the entire cluster c_i. Similarly the upper bound is determined by the point x_{max} with the maximal density. Obviously it is not practical to figure out the lower/upper bound by computing the densities at all points and thereafter finding x_{min} and x_{max}.

Algorithm 1 : Top-N Outlier Computation.

```
Input: Clusters $C$.
Output: Top-N Outliers.
$PriorityQueue$<Cluster> $P$ of size $N$ /*by upperbound in ascending order*/
$P \leftarrow$ first $N$ in $C$
for rest of the cluster $c$ in $C$ do
    if $KLOME_{low}(c) > KLOME_{up}(P.peek)$ then
        prune $c$
    else if $KLOME_{up}$(c) < $KLOME_{low}(P.peek)$ then
        $P$.poll & $P$.add($c$)
    else
        $P$.add($c$)
$PriorityQueue$<Data> $R$ of size $N$ /*by $KLOME$ in ascending order*/
for cluster $c$ in $P$ do
    for data $d$ in $c$ do
        compute $KLOME$ of $d$
        $R$.add($d$)
return $R$
```



(a) Lowest KLOME in c **(b) Lower bound of c**

Figure 7: An example of lower KLOME bound.

Lower bound. We now show that by utilizing the statistical property of each data cluster – more specifically the *radius*, the bounds can be derived in constant time. Here we use the lower bound as example to demonstrate our solution (Fig. 7).

Lemma 6.1. *Given a data cluster c_i, its k nearest kernel centers $\{c_{c_1}, \cdots, c_{c_k}\}$ and the data point x_{min} which has the minimum density among all points in c_i, $\tilde{f}_{min}(c_i) \le \tilde{f}(x_{min})$, where $\tilde{f}_{min}(c_i) = \sum_{j=1}^{k} \omega(c_{c_j}) K_h(|c_{c_i} - c_{c_j}| + r)$. Here r is the radius of c_i and c_{c_i} is the centroid of c_i.*

Proof. The density contribution $K_h(|x_i - c_{c_j}|)$ is inversely proportional to the distance between the evaluated point x_i and the kernel center c_{c_j}. The longer the distance, the smaller the density contribution is from the kernel center. The radius r of a cluster c_i is the distance from c_i's centroid c_{c_i} to the furthest possible points in c_i. The longest possible distance from a kernel center c_{c_j} to any point in c_i is denoted as $d_c = |c_{c_i} - c_{c_j}| + r$. The distance from c_{c_1} to x_{min} is denoted as $d_x = |c_{c_j} - x_{min}|$. $d_c \ge d_x$ by the triangle inequality. Therefore $K_h(d_x) \le K_h(d_c)$. This holds for any kernel center c_{c_j}. Therefore $\tilde{f}_{min}(c_i) = \sum_{j=1}^{k} \omega(c_{c_j}) K_h(|c_{c_j} - c_{c_i}| + r) \le \tilde{f}(x_{min})$. □

Intuitively, the density at a data point is measured by the summation of the density contributions of all relevant kernel centers. The summation of the density contribution from each kernel center c_{c_j} to the point x_j that is the point furthest to c_{c_j} in c_i is guaranteed to be smaller or equal to the density at point x_{min}. This is so because the distance from x_{min} to each kernel center c_{c_j} cannot be larger than the distance between c_{c_j} and x_j.

According to Lemma 6.1, given the radius of a data cluster c_i and its k nearest kernel centers $c_{c_1} \cdots c_{c_k}$, the **lower KLOME bound** of cluster c_i is computed as:

$$KLOME_{low}(c_i) = z\text{-}score(\tilde{f}_{min}(c_i), \{\tilde{f}(c_{c_1}) \cdots \tilde{f}(c_{c_k})\}). \quad (11)$$

Upper Bound. Similarly, we can show that the maximal local density at a cluster c_i, denoted by $\tilde{f}_{max}(c_i)$, can be obtained based on the shortest distance from each kernel center to the points in c_i.

$$\tilde{f}_{max}(c_i) = \sum_{j=1}^{k} \omega(c_{c_j}) K_h(|c_{c_j} - c_{c_i}| - r). \quad (12)$$

Accordingly, the upper KLOME bound of each cluster c_i $KLOME_{up}(c_i)$ is derived based on $\tilde{f}_{max}(c_i)$.

$$KLOME_{up}(c_i) = z\text{-}score(\tilde{f}_{max}(c_i), \{\tilde{f}(c_{c_1}) \cdots \tilde{f}(c_{c_k})\}). \quad (13)$$

7 THE EFFICIENT STREAM DATA ABSTRACTOR

The stream data abstractor adapts a lightweight stream clustering algorithm similar to [2, 8] that clusters the extremely close data points together. As the clusters are continuously constructed and incrementally maintained, the statistics needed by both *a*KDE and inlier pruning, namely the *cardinality*, the *centroid*, and the *radius* of the cluster, must also be continuously generated. We thus refer to this as dual-purpose clustering. The dual-purpose clustering is based on two key ideas: *additive meta data* and *pane-based meta data maintenance.*

The **additive meta data** is inspired by *micro-clustering* [2] – a popular stream clustering approach. The idea is that by maintaining meta data that satisfies the additive properties, the statistics required by both the density estimator and the outlier detector can be computed in constant time whenever the window evolves.

Definition 7.1. A **cluster** c_i in a d-dimensional data set $S_{W_c} = \{x_1, \cdots, x_m\}$ corresponding to the data in the current window W_c of stream S is represented as a 4-tuple set $\{M, LS, R_{min}, R_{max}\}$ where M denotes the cardinality of the cluster, $LS =< \sum_{i=1}^{m} x_i^1, \cdots, \sum_{i=1}^{m} x_i^d >$ is the linear sum of the points by dimension, $R_{min} =< x_{min}^1, \cdots, x_{min}^d >$ and $R_{max} =< x_{max}^1, \cdots, x_{max}^d >$ are the minimum and maximum values of the points in each dimension.

Cardinality and Centroids for *a*KDE. In Def. 7.1, M refers to data cardinality of cluster c_i. M is used to compute the *weight* (Eq. 7) and the *centroid* of the abstract kernel center. The linear sum LS is used to compute the *centroid* of cluster $c_i = \frac{LS}{M}$.

The Radius for Inlier Pruning. R_{min} and R_{max} representing the minimal and maximal values in each dimension are utilized to compute the radius of cluster c_i. Radius is a key statistic needed by our outlier detector to quickly prune the clusters from the outlier candidate set.

Since the radius is defined as the distance from the centroid c_{c_i} to its furthest point in cluster c_i, the radius changes whenever the centroid changes. All points in c_i then have to be re-scanned

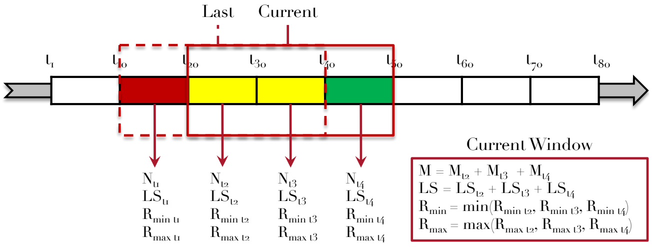


Figure 8: An example of an evolving cluster.

to find the point "furthest" from the new centroid. This, being computational expensive, is not acceptable in online applications.

The remedy comes from our carefully selected product kernel function (Eq. 5). In the product kernel, each dimension has its own customized bandwidth. Accordingly, we only need the radius on each single dimension to estimate the bandwidth instead of the radius over the multi-dimensional data space. Since updating the minimum or maximum value per insertion or deletion to an array is constant cost, the cost of radius maintenance for k separated dimensions (R_{min} and R_{max}) is constant as well.

Pane-based meta data maintenance. The pane-based meta data maintenance strategy [17] is utilized to effectively update the meta data for each cluster as the window slides. Given the window size $S.win$ and slide size $S.slide$, a window can be divided into $\frac{S.win}{gcd(S.win,S.slide)}$ small panes where *gcd* refers to greatest common divisor. The meta data of a cluster c_i is maintained at the pane granularity instead of maintaining one meta data structure for the whole window. Since the data points in the same pane arrive and expire at the same slide pace, the meta data can be quickly computed by aggregating the meta data structures maintained for the unexpired panes as the window moves. This process is illustrated in Fig. 8. Since the meta data satisfies the additive property, the computation can be done in constant time. In this way, no explicit operation is required to handle the expiration of outdated data from the current window. Therefore, our stream clustering algorithm only needs to exclusively deal with the new arrivals.

The Dual-Purpose Stream Clustering Algorithm. Once a new data point x arrives, The algorithm first finds its nearest cluster according to the distance of x to all the centroids. If the distance from x to its nearest cluster c_i denoted as $dist(x, c_{c_i})$ is smaller than a radius threshold θ, x is inserted into c_i. The corresponding *4-tuple* meta data is updated accordingly. On the other hand, if $dist(x, c_{c_i}) > \theta$, a new cluster will be created. This logic is described in Alg. 2.

7.1 Time Complexity Analysis of KELOS

The complexity of the clustering comes from the nearest centroid search. The complexity is $O(mc)$ with m the number of new arrivals and c the number of centroids. In the density estimation step, each point has to find its k nearest kernel centers from the c centroids. Therefore, in worst case the complexity is $O(c)$ for each point. In the outlier detection step, the cluster-based pruning takes $O(c^2)$.

One more pass is required for the remaining points to compute the density and the outlierness score. Assuming the number of remaining points is l, the density computation takes $O(lc)$, while the outlierness score computation takes $O(l \log N)$, where $O(\log N)$ comes from the priority queue operation for maintaining the top-N outlierness score points. Therefore, the overall computation costs for a batch of newly arriving data points is $O(mc) + O(c^2) + O(lc) + O(l \log N)$. In summary, the time complexity of our KDE based outlier detection approach is linear in the number of points. Since typically $N \ll c \ll l \ll m$, the complexity is dominated by the clustering step.

Algorithm 2 : Dual-Purpose Stream Clustering

```
Input: Data batch $\mathcal{D}$ at time $t_i$, window size $S.win$ and threshold $\theta$.
Output: Clusters $C$.
Array<Cluster> C
for every data $d$ in $\mathcal{D}$ do
    $dist_{min} \leftarrow +\infty$
    Cluster $c_n$
    for every cluster $c$ in $C$ do
        $dist \leftarrow distance(d, c.centroid_{t_{i-S.win+1} \rightarrow t_i})$
        if dist $< dist_{min}$ then
            $dist_{min} \leftarrow dist$
            $c_n \leftarrow c$
    if $dist_{min}$ < threshold $\theta$ then
        $c_n$.insert($d$,$t_i$)
    else
        Cluster $c_{new}$
        $c_{new}$.insert($d$,$t_i$)
        $C$.add($c_{new}$)
return $C$
```

8 EXPERIMENTAL EVALUATION

8.1 Experimental Setup & Methodologies

In this section, we compare the efficiency and effectiveness of KELOS against the state-of-art local outlier detection approaches for data streams. All experiments are conducted on an Ubuntu server with 56 Intel(R) Xeon(R) 2.60GHz cores and 512GB memory. Overall, our KELOS is 1-6 orders of magnitude faster, while still at least as accurate as the alternative approaches.

We utilize the public as well as synthetic datasets generated using the data mining framework ELKI [1] to measure the efficiency of KELOS. We also work with real labeled datasets so that we can evaluate the effectiveness of KELOS.

Real Datasets. We work with 3 labeled public datasets. The **HTTP** dataset [10] contains in total 567,479 network connection records with 2,211 among them being outliers. The labeled outliers correspond to different types of network intrusions including DOS, R2L, U2R, etc. Three numerical attributes, namely duration, src_bytes and dst_bytes are utilized in our experiments. The points in the HTTP dataset are ordered by their arrival time. Therefore we can generate a data stream simply by enforcing a sliding window on it.

The Yahoo! Anomaly Dataset (**YAD**) [15] is considered as one of the industry standards for outlier detection evaluation. It is composed of 4 distinct data sets. In this work we utilize Yahoo! A1 and Yahoo! A2. Yahoo! A1 is based on the real production traffic to some of the Yahoo! services. The anomalies are marked by Yahoo! domain experts. Yahoo! A2 is a synthetic data set containing time-series with random seasonality, trend and noise. Yahoo! A1 and Yahoo! A2 contain 94,866 points with 1,669 outliers and 142,101 points with 312 outliers respectively. Each data point has three attributes: timestamp, value, and label.

Synthetic Datasets. We generate synthetic datasets to measure the efficiency of KELOS under various scenarios with different distribution properties. We first create static datasets containing different number of data clusters and then utilize these datasets to simulate windowed streams. For example, to simulate a windowed stream with three clusters and some outliers, we first create a static dataset containing three clusters. The size and shape of the clusters are controlled by different parameter settings in ELKI such as type of distribution, standard deviation, etc. When generating a sliding window data stream, different windows correspond to different datasets with different cluster densities.

Comparative Methods. We compare KELOS against five baselines, namely sLOF, sKDEOS, pKDEOS, iLOF [20] and MiLOF [21]. LOF [6] is the seminal and most popular local outlier detection method. KDEOS [23] leverages KDE in local density estimation which is then used to compute an outlierness score for each point. Since their performance bottleneck is the kNN search, we implemented the skyband stream kNN search algorithm [29] to speed up their outlier detection process for windowed streams and named the modified methods as sLOF and sKDEOS. iLOF [20] and MiLOF [21] are two incremental LOF algorithms specifically designed for *landmark* windows. iLOF computes the LOF score for each inserted data point and update the LOF score of the affected data points following the reverse kNN relationships. MiLOF improves iLOF in a scenario where the memory is limited. When the memory limit is reached, part of the existing data points are summarized into small clusters as references for the future LOF approximation. Since iLOF and MiLOF do not handle data expiration, they have to start from scratch for each new window. In the original KDEOS, every data point in the input dataset is used as kernel center. We also implemented a sampling-based sKDEOS called pKDEOS, since it is the common practice for KDE to use only the data points uniformly sampled from the input dataset as kernel centers. All methods continuously return the N points with the highest outlierness scores as outliers in each window.

Efficiency Measures. We measure the end-to-end execution time.

Effectiveness Measures. We measure the effectiveness using the *Precision@N* (*P@N*) metric typical for the evaluation of outlier detection techniques [7].

$$P@N = \frac{\text{\# of True Outliers}}{N}. \tag{14}$$

Intuitively, $P@N$ measures the true positive of the detected top-N outliers. An ideal $P@N$ equals to 1, where all outliers are found and no inlier is returned as result. Here we measure the $P@N$ metric window by window and report the average $P@N$ over all windows. Following [7], we replace N with $|O|$ in the $P@N$ computation for each window, where $|O|$ corresponds to the total number of labeled (ground truth) outliers falling in this window. Only the top-$|O|$ points out of the top-N outlier list are used in the evaluation. Therefore, the $P@|O|$ for n consecutive stream windows is:

$$P@|O| = \frac{\sum_{i=1}^{n} \text{\# of True Outliers in top-}|O|_i}{\sum_{i=1}^{n} |O|_i}, \tag{15}$$

where $|O|_i$ denotes the number of the true outliers in the ith window.

To investigate the quality of the ranking in the Top-$|O|$ outlier list, we also measure the *average precision* (AP) [31]:

$$Average\,Precision(AP) = \frac{1}{|O|} \sum_{o \in O} P@rank(o). \tag{16}$$

Here P@rank(o) measures the $P@N$, where N is set as rank(o) representing the position at which a true outlier o appears. AP measures how well the true outliers are ranked among all data points.

8.2 Efficiency Evaluation

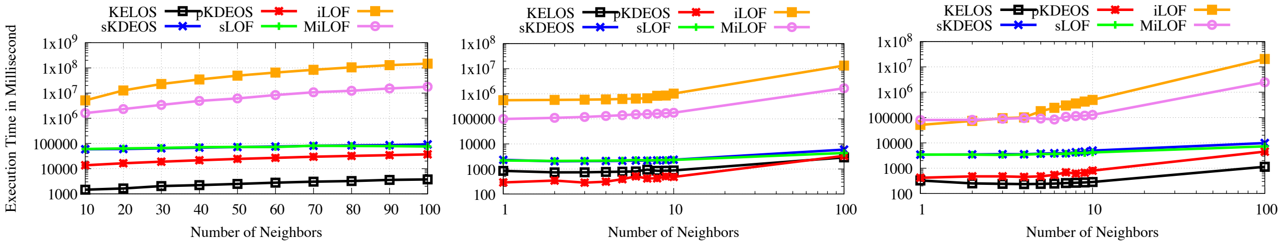
We evaluate the end-to-end execution time and memory consumption by varying the number of the neighbors k, the window size and the window overlap rate.

Number of Neighbors k. The k parameter defines the number of the neighbors to be considered in the computation of outlierness score for each point. We first report the execution time of all methods on real datasets with varied k from 10 to 100. The radius threshold θ of KELOS appropriate for HTTP, Yahoo! A1, and Yahoo! A2 are set as 0.095, 0.1 and 40 (See Sec.8.4 for parameter tuning). The window size of HTTP is set to 6,000 and window sizes of Yahoo! A1 and Yahoo! A2 are set as 1,415, and 1,412 based on the instruction of the data provider for the effectiveness of outlier detection. The sampling rates of pKDEOS is set as 10% which is a relatively high sampling rate ensuring that pKDEOS always has more than k kernel centers to use as k increases. For MiLOF, we configured it to keep 10% of the data in memory for LOF score approximation.

As shown in Fig. 9(a), KELOS is about 2-6 orders of magnitude faster than the alternatives. Among all alternative methods, iLOF and MiLOF are the slowest. To reduce the influence of the new arrivals they use a *point-by-point* processing strategy that computes the LOF score for each new point and update the LOF score of the affected data points immediately after a single insertion. It wastes significant CPU time on unnecessarily updating the LOF scores of some points that are modified again later due to the insertion of other new arrivals. sLOF and sKDEOS are 1-2 orders of magnitude faster than the previous two, as they compute the outlierness score only once for each data point in the entire window. pKDEOS is faster than sKDEOS and sLOF, because pKDEOS only utilizes the sampled points as kernel centers. Searching for the k nearest kernel centers from the sampled kernel center set is much faster than searching among all points in each window. However, pKDEOS is still at least 1 order of magnitude slower than KELOS on HTTP. This is because in order to satisfy the accuracy requirement, the number of the sampled kernel centers has to be large enough to represent the distribution of the data stream. While the aKDE approach of KELOS only uses the centroid of each cluster as abstract kernel center. Therefore, the number of the clusters tends to be much smaller than the number of the sampled kernel centers. Furthermore, KELOS effectively prunes most of the inliers without conducing the expensive density estimation, while in contrast, others have to compute the outlierness score for each and every data point.

As shown in Fig. 9(b), although pKDEOS is faster than KELOS on Yahoo! A1 due to the smaller population of the sampled kernel centers, KELOS outperforms pKDEOS in the effectiveness measurements (Tab. 1). On average, KELOS keeps slightly more kernel centers in the memory than pKDEOS for Yahoo! A2 (Fig. 12(c)). However, KELOS still outperforms pKDEOS on execution time because of our inlier pruning strategy.

Window Size. To evaluate how window size affects the efficiency of KELOS, we measure and compare the average window

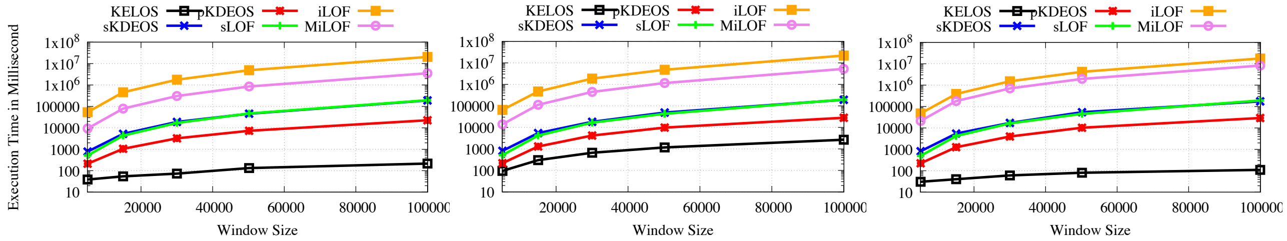


(a) HTTP

(b) Yahoo! A1

(c) Yahoo! A2

Figure 9: Execution time of varied number of neighbors k. Note the maximum k that each method can reach is different. For LOF-based methods, it depends on the total number of data points. For KDE-based methods, it depends on the number of the kernel centers available.

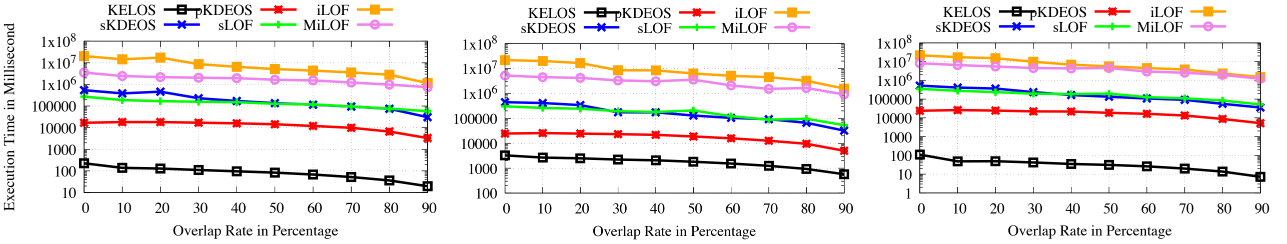


(a) One-Cluster Stream

(b) Three-Cluster Stream

(c) Six-Cluster Stream

Figure 10: Execution time on synthetic datasets of varied window size.



(a) One-Cluster Stream

(b) Three-Cluster Stream

(c) Six- Cluster Stream

Figure 11: Execution time on synthetic datasets of varied overlap rate.

processing time of each method by varying the window size of synthetic datasets from 5,000 to 100,000. For these experiments, we created three data streams with one, three and six Gaussian clusters with various parameter configurations (mean, variance and etc.) across the timeline and 10 outliers for each window. The parameters of different all the methods are configured such as they achieve the same accuracy ($P@|O| > 0.95$) with as little memory consumption as possible while k=10. As shown in Fig. 10, in general, the execution time increases with the increase of the window size (average data points within a single window). KELOS constantly outperforms all other baselines by 1-5 orders of magnitude.

Window Overlap Rate. We evaluate the efficiency of KELOS by varying synthetic streams' window overlap rate from 0% to 90%. The synthetic streams are created in the same fashion to the previous experiments. The parameters of different all the methods are configured such as they achieve the same accuracy ($P@|O| > 0.95$) with as little memory consumption as possible and k=10. Based on this controlled accuracy, we evaluate the average window processing time.

When the overlap rate increases, the execution time of all six approaches decrease as demonstrated in Fig. 11. This is because iLOF and MiLOF are designed to process the data incrementally. Although iLOF and MiLOF are not capable of handling data pruning, in these experiments, we simply assume that the overlapped data are already processed and we report the their processing times for the rest of the window. sKDEOS, pKDEOS and sLOF all leverage the skyband stream kNN search that incrementally computes the kNN for each point as window slides, while the clustering algorithm used by KELOS is incremental by nature as shown in Sec. 7. As shown in Fig. 11, KELOS is 2-5 orders of magnitude faster than iLOF, MiLOF, sKDEOS, sLOF and pKDEOS in all settings on all three streams. iLOF and MiLOF are the slowest. sLOF and sKDEOS are faster than the previous two but slower than pKDEOS. Although they all can save computation by avoiding some unnecessary reprocessing of the existing data, the reason of the performance difference is the same as we already explained in the previous experiments.

Memory Consumption. The memory consumption shown in Fig. 12 is evaluated by counting the number of the kernel centers and data points kept in the memory by each approach. sKDEOS, iLOF and sLOF (represented by sLOF in Fig. 12) utilizes all points as kernel centers or reference points, while pKDEOS and MiLOF (represented by pKDEOS in Fig. 12) dramatically reduces the number of the kernel centers by sampling and the number of the references by clustering. KELOS uses the smaller number of kernel centers as compared to sLOF which facilitate more accuracte density estimation. The number of the kernel centers is equivalent to the number of the clusters that tends to be small. sLOF and iLOF measures the local density at each point by computing the local reachability density (LRD). Since the LRD computation requires the access to all points falling in each window, it is equivalent to using all points as kernel centers similar to sKDEOS. On synthetic data stream, Similar to the memory consumption on the real HTTP dataset, KELOS uses the least number of kernel centers. pKDEOS and MiLOF uses fewer kernel centers or references than sKDEOS, iLOF and sLOF because of the sampling and clustering.

8.3 Effectiveness Evaluation

We report the accuracies of our KELOS, sKDEOS (≈ pKDEOS) and sLOF methods on the real data streams. Tab. 1 (with pKDEOS included) shows the peak $P@|O|$ and AP for each approach on each dataset. KELOS outperforms all other approaches in all cases. We are not reporting the results of iLOF and MiLOF, since they perform equally to sLOF in effectiveness. Similarly, for the KDE-based methods we only report the results on sKDEOS for various configurations.

Number of Neighbors k. The number of the neighbors k is the most influential factor that affects the detection accuracies of all methods. The parameter settings are the same to the efficiency evaluation when varying k.

Fig. 13& 14 demonstrate the trend of $P@|O|$ and AP as k varies. The line of KELOS stops at 800 in Fig. 13(a)&14(a), because KELOS uses cluster-based aKDE approach. The number of the kernel centers is restricted by the number of the clusters. Fig. 13(a) shows the results on the HTTP dataset. For our KELOS, as k increases, the $P@|O|$ increases until k reaches 80. It then starts decreasing after k is larger than 100. Overall sKDEOS and sLOF show the similar trend. Compared to KELOS they have to use a much larger k to get relative high accuracy. The trends on the Yahoo A1 and A2 datasets are different from that on the HTTP dataset as shown in Fig. 13(b)% 13(c). The $P@|O|$ continuously increases and gets stable after k reaches certain value. This confirms our observation that using as many as possible kernel centers in the density estimator does not always lead to more accurate density estimation. This justifies our decision of adopting the balloon kernel that only takes the close kernels into consideration when estimating the density at a point x.

The trends of AP are similar to the trends of $P@|O|$ on all datasets as shown in Fig. 14. Overall, KELOS is as accurate or more accurate than alternative approaches. Furthermore, compared to the alternatives, KELOS uses a smaller k to achieve high accuracy. This also contributes to the performance gain of KELOS in execution time.

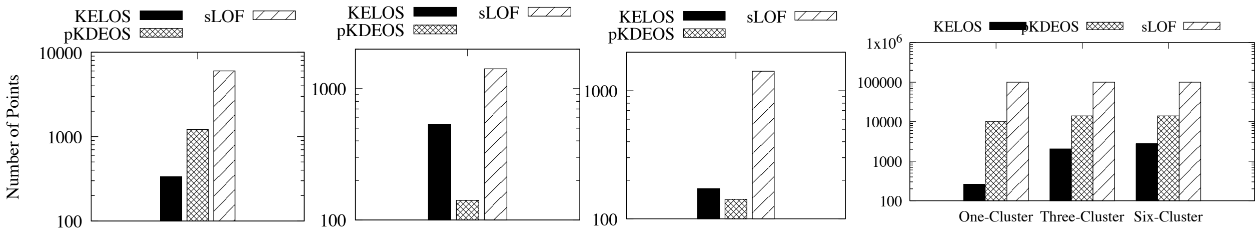
8.4 Hyper Parameter Tuning

In KELOS, the radius threshold θ defines the maximum possible radius of the formed clusters, that is, the tightness of the clusters. Since the effectiveness of our aKDE approach (Sec. 5) and the pruning strategy (Sec. 5) rely on the tightness of the clusters, θ is important for the accuracy of KELOS. In this set of experiments, we vary θ from small to large on the large HTTP dataset that contains multiple data clusters. As shown in Fig. 15(a), when θ is at 0.1, the $P@|O|$ is at the peak. Then $P@|O|$ and AP of KELOS starts to decrease gradually as θ increases. Large θ results in a small number of clusters that have large radius. Potentially the centroid of a large radius cluster might not precisely represent all points in the cluster. This leads to inaccurate density estimation. Furthermore, a larger radius causes looser upper and lower KLOME bound. This makes the inlier pruning less effective. However, a smaller radius θ inevitably leads a large number of clusters. This increases the computation costs of both stream clustering and the cluster-based aKDE method as shown in Fig. 15(b). Therefore, KELOS will achieve the best performance when radius θ is set to the largest value that is still 'small' enough to generate tight data clusters.

Tuning Radius Threshold θ. Micro-clustering utilizes the radius threshold θ to make sure only the extremely similar points fall into the same cluster and produce tight clusters. The smaller the θ is, the tighter the formed clusters are. However, as shown in Fig. 15(a), although small θ achieves high accuracy in density estimation, it also slows down the overall speed of KELOS. Therefore, it is important to find an appropriate θ threshold that can balance the speed and the accuracy. Instead of trying to acquire an optimal θ value at the beginning by exploring some expensive preprocessing, in streaming context we recommend that the θ threshold could be dynamically adapted to the optimal value. More specifically, one can start by initializing the system using a relatively small θ value to ensure the accuracy of the results. Then the θ value can be gradually adjusted to larger values as the data stream evolves as long as the accuracy is still reasonably good based on the user feedback. As concept drift occurs when stream data evolves, the θ is adjusted again using the same principle.

9 RELATED WORK

Local Outlier Factor. Local outlier detection has been extensively studied in the literature since the introduction of the Local Outlier Factor (LOF) semantics [6]. A detailed survey of LOF and its variations can be found in [7]. The concept of local outlier, LOF in particular, has been applied in many applications [7]. However, LOF requires kNN search for every data point and needs multiple iterations over the entire dataset to compute these LOF values. For this reason, to support continuously evolving streaming data, iLOF was proposed [20] to quickly find the points whose LOF scores are influenced by new arrivals. This avoids re-computing the LOF score for each point as the window slides. However as the velocity of the stream increases, most of the points in a window will be influenced. Therefore this approach does not scale to high volume streaming data. In [21] an approximation approach MiLOF was designed to support LOF in streaming data that focuses on the memory efficiency. However, MiLOF only considers new incoming data. It does not offer any efficient strategy to handle the update caused by data expiration. Therefore, it is not efficient when handing windowed streams. As evaluated



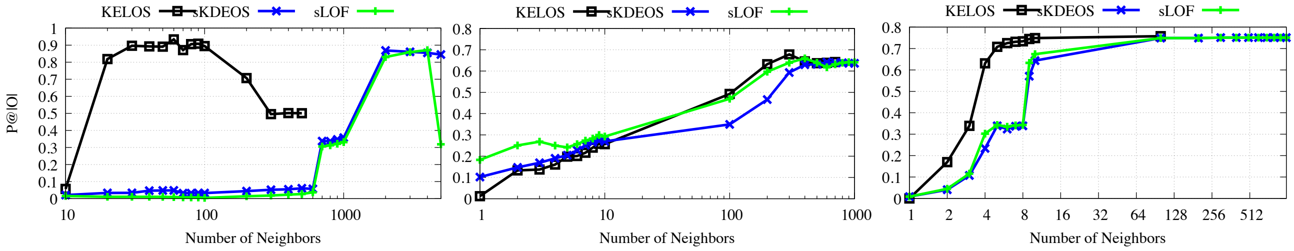
(a) HTTP

(b) Yahoo! A1

(c) Yahoo! A2

(d) Synthetic

Figure 12: Memory consumption for real and synthetic data streams (window size as 100,000).

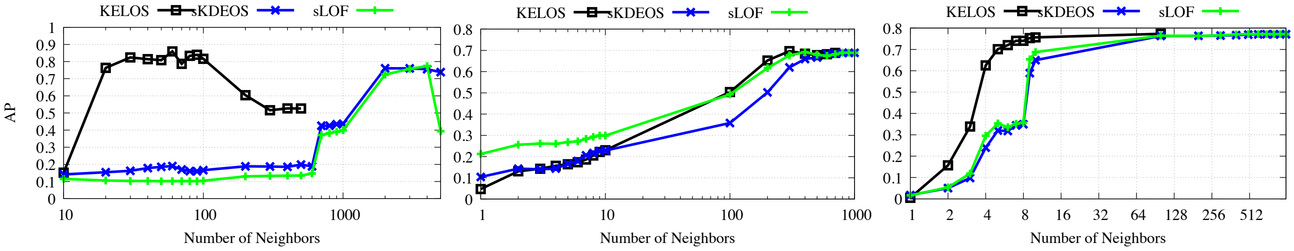


(a) HTTP

(b) Yahoo! A1

(c) Yahoo! A2

Figure 13: $P@|O|$ of varied number of neighbors k.



(a) HTTP

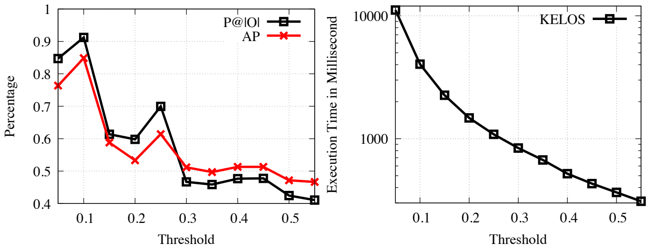
(b) Yahoo A1

(c) Yahoo A2

Figure 14: AP of varied number of neighbors k.

Table 1: Peak accuracies among various k.

	P@|O|			AP		
	HTTP	Yahoo! A1	Yahoo! A2	HTTP	Yahoo! A1	Yahoo! A2
sLOF	87.06%	65.97%	75.11%	77.34%	69.16%	77.19%
sKDEOS	86.88%	64.17%	75.11%	76.06%	68.84%	76.95%
pKDEOS	87.43%	37.39%	74.89%	77.54%	36.43%	77.10%
KELOS	93.40%	67.83%	75.75%	85.92%	69.64%	77.30%



(a) Accuracy

(b) Execution Time

Figure 15: Varying radius threshold θ.

in our experiments, iLOF and MiLOF are much slower than our skyband-based streaming LOF implementation sLOF.

Efficient Kernel Density Estimation. Kernel density estimation is considered as a quadratic process $O(nm)$ with n the total number of data points and m the number of kernel centers. Previous efforts have aimed to accelerate this process while still providing accurate estimation, such as utilizing sampling [26]. [14, 32] designed a method that incrementally maintains a small, fixed size of kernel centers to perform density estimation over data streams. However, to ensure the accuracy of density estimation over skewed datasets, the sample size has to be large. Therefore it cannot solve the efficiency problem of KDE in our context. [12] studied the density-based classification problem. It proposed a pruning method that correctly classifies the data without estimating the density for each point by utilizing a user-defined density threshold. However, this pruning method can not be applied to solve our problem, since a point with low density is not necessarily an outlier based on the local outlier semantics we target on.

Outlier Detection using KDE. For each point in the current window of a sliding window stream, [27] utilizes KDE to approximate the number of its neighbors within a certain range. This information is then utilized to support distance-based outlier detection and LOCI [19]. It directly applies off-the-shelf KDE method on each window. No optimization technique is proposed to speed up KDE in the streaming context. [16] is the first work that studied how to utilize KDE to detect local outliers in static datasets. This later was improved by [23] to be better aligned with LOF semantics. Each data point's density is estimated based upon the surrounding kernel centers only, therefore called local density. Instead of considering outliers only based on their density value, data points are measured based on the density in contrast to their neighbors. However, this work does not improve the efficiency of KDE. Nor does it consider streaming data. As confirmed in our experiment (Sec. 8.2), it is indeed orders of magnitude slower than our KELOS.

Other Streaming Outlier Detection Approaches. LEAP [9] and Macrobase [5] scale distance-based and statistical-based outlier detection respectively to data streams. They rely on the absolute density at each point to detect outliers, while we work with the local outlier method which determines whether a point is an outlier based on the density relative to its neighbors. It tends to be more effective than the absolute density-based methods [11]. Streaming HS-Trees [28] detects outliers by using a classification forest containing a set of randomly constructed trees. The points falling in the leafs that contain a small number of points are considered as outliers. Similar to [5, 9], this method also relies on absolute density of each point to detect outliers. RS-Hash [22] proposed an efficient outlier detection approach using sample-based hashing and ensemble. However, different from our local outlier mining problem, it focuses on subspace outlier detection, that is, detecting outliers hidden in different subspaces of a high dimensional dataset. Similar to iLOF [20] and MiLOF [21], DILOF [18] processes the data stream in a point-by-point fashion and incrementally detects outliers and sequential outliers from *landmark* windows. It does not handle data expiration which is a required operation in sliding windows scenario.

10 CONCLUSION

We present KELOS – the first solution for continuously monitoring top-N KDE-based local outliers over sliding window streams. First, we propose the KLOME semantics to continuously capture the *n* points that have the highest outlierness scores in the streaming data. Second, a continuous detection strategy is designed that efficiently supports the KLOME semantics by leveraging the key properties of KDE. Using real world datasets we demonstrate that KELOS is 2-6 orders of magnitude faster than the baselines, while being highly effective in detecting outliers from data streams.

REFERENCES

[1] Elke Achtert, Hans-Peter Kriegel, and Arthur Zimek. 2008. ELKI: A Software System for Evaluation of Subspace Clustering Algorithms. In *Scientific and Statistical Database Management, 20th International Conference, SSDBM 2008, Hong Kong, China, July 9-11, 2008, Proceedings.* 580–585.

[2] Charu C Aggarwal, Jiawei Han, Jianyong Wang, and Philip S Yu. 2003. A framework for clustering evolving data streams. In *VLDB*. VLDB Endowment, 81–92.

[3] Sarah Miller Al Pascual, Kyle Marchini. 2017. Identity Fraud: Securing the Connected Life. (2017).

[4] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and Issues in Data Stream Systems. In *PODS*. 1–16.

[5] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. 2017. MacroBase: Prioritizing Attention in Fast Data. In *SIGMOD*. 541–556.

[6] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. 2000. LOF: Identifying Density-based Local Outliers. In *SIGMOD*. 93–104.

[7] Guilherme O Campos, Arthur Zimek, Jörg Sander, Ricardo JGB Campello, Barbora Micenková, Erich Schubert, Ira Assent, and Michael E Houle. 2016. On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study. *Data Mining and Knowledge Discovery* 4, 30 (2016), 891–927.

[8] Feng Cao, Martin Estert, Weining Qian, and Aoying Zhou. 2006. Density-based clustering over an evolving data stream with noise. In *SDM*. SIAM, 328–339.

[9] Lei Cao, Di Yang, Qingyang Wang, Yanwei Yu, Jiayuan Wang, and Elke A Rundensteiner. 2014. Scalable distance-based outlier detection over high-volume data streams. In *ICDE*. IEEE, 76–87.

[10] The Third International Knowledge Discovery and Data Mining Tools Competition. 1999. KDD Cup Dataset. *http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html* (1999).

[11] Charu C. Aggarwal: Second Edition. 2017. *Outlier Analysis*. Springer.

[12] Edward Gan and Peter Bailis. 2017. Scalable Kernel Density Classification via Threshold-Based Pruning. In *SIGMOD*. 945–959.

[13] Francisco J Goerlich Gisbert. 2003. Weighted samples, kernel density estimators and convergence. *Empirical Economics* 28, 2 (2003), 335–351.

[14] C. Heinz and B. Seeger. 2008. Cluster Kernels: Resource-Aware Kernel Density Estimators over Streaming Data. *TKDE* 20, 7 (July 2008), 880–893.

[15] Nikolay Laptev, Saeed Amizadeh, and Ian Flint. 2015. Generic and Scalable Framework for Automated Time-series Anomaly Detection. In *SIGKDD*. 1939–1947.

[16] Longin Jan Latecki, Aleksandar Lazarevic, and Dragoljub Pokrajac. 2007. Outlier detection with kernel density functions. In *MLDM*. Springer, 61–75.

[17] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005 (SIGMOD 2005)*. 311–322.

[18] Gyoung S Na, Donghyun Kim, and Hwanjo Yu. 2018. DILOF: Effective and Memory Efficient Local Outlier Detection in Data Streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 1993–2002.

[19] Spiros Papadimitriou, Hiroyuki Kitagawa, Phillip B. Gibbons, and Christos Faloutsos. 2003. LOCI: Fast Outlier Detection Using the Local Corr. Integral. In *ICDE*. 315–326.

[20] Dragoljub Pokrajac, Aleksandar Lazarevic, and Longin Jan Latecki. 2007. Incremental local outlier detection for data streams. In *CIDM*. IEEE, 504–515.

[21] Mahsa Salehi, Christopher Leckie, James C. Bezdek, Tharshan Vaithianathan, and Xuyun Zhang. 2016. Fast Memory Efficient Local Outlier Detection in Data Streams. *TKDE* 28, 12 (2016), 3246–3260.

[22] Saket Sathe and Charu C Aggarwal. 2016. Subspace outlier detection in linear time with randomized hashing. In *ICDM*. IEEE, 459–468.

[23] Erich Schubert, Arthur Zimek, and Hans-Peter Kriegel. 2014. Generalized outlier detection with flexible kernel density estimates. In *SDM*. SIAM, 542–550.

[24] David W Scott. 2015. *Multivariate density estimation: theory, practice, and visualization*. John Wiley & Sons.

[25] Simon J Sheather and Michael C Jones. 1991. A reliable data-based bandwidth selection method for kernel density estimation. *J. Royal Stat. Soc.* (1991), 683–690.

[26] Bernard W Silverman. 1986. *Density estimation for statistics and data analysis*. Vol. 26. CRC press.

[27] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos. 2006. Online Outlier Detection in Sensor Data Using Non-parametric Models. In *VLDB*. VLDB Endowment, 187–198.

[28] Swee Chuan Tan, Kai Ming Ting, and Tony Fei Liu. 2011. Fast anomaly detection for streaming data. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, Vol. 22. 1511.

[29] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. 2002. Continuous Nearest Neighbor Search. In *VLDB*. 287–298.

[30] George R Terrell and David W Scott. 1992. Variable kernel density estimation. *The Annals of Statistics* (1992), 1236–1265.

[31] Ethan Zhang and Yi Zhang. 2009. Average precision. In *Encyclopedia of database systems*. Springer, 192–193.

[32] Aoying Zhou, Zhiyuan Cai, Li Wei, and Weining Qian. 2003. M-kernel merging: Towards density estimation over data streams. In *DASFAA*. IEEE, 285–292.

[33] Dennis Zill, Warren S Wright, and Michael R Cullen. 2011. *Advanced engineering mathematics*. Jones & Bartlett Learning.

RDF graph summarization: principles, techniques and applications

Haridimos Kondylakis
Institute Of Computer Science, FORTH
Heraklion, Greece
kondylak@ics.forth.gr

Dimitris Kotzinos
Lab. ETIS UMR 8051, University of Paris-Seine, University of Cergy-Pontoise, ENSEA, CNRS
Pontoise, France
Dimitrios.Kotzinos@u-cergy.fr

Ioana Manolescu
Inria and LIX (UMR 7161, CNRS and Ecole polytechnique)
Palaiseau, France
ioana.manolescu@inria.fr

ABSTRACT

The explosion in the amount of the RDF on the Web has lead to the need to explore, query and understand such data sources. The task is challenging due to the complex and heterogeneous structure of RDF graphs which, unlike relational databases, do not come with a structure-dictating schema. *Summarization* has been applied to RDF data to facilitate these tasks. Its purpose is to extract concise and meaningful information from RDF knowledge bases, representing their content as faithfully as possible. There is no single concept of RDF summary, and not a single but many approaches to build such summaries; the summarization goal, and the main computational tools employed for summarizing graphs, are the main factors behind this diversity.

This tutorial presents a structured analysis and comparison existing works in the area of RDF summarization; it is based upon a recent survey which we co-authored with colleagues [3]. We present the concepts at the core of each approach, outline their main technical aspects and implementation. We conclude by identifying the most pertinent summarization method for different usage scenarios, and discussing areas where future effort is needed.

1 INTRODUCTION

The explosion in the amount of the RDF on the Web has lead to the need to explore, query and understand such data sources. This need arises both for computer scientists and for scientists and practitioners in the many areas where Open Data is produced - ranging from agriculture to education, from cultural artefacts to criminality statistics. All users who need to tame, understand and analyze such complex RDF graphs are faced with several challenges.

Firstly, RDF graphs are often large compared with the human ability to understand and analyze them; even a “tiny” graph of e.g. 10.000 nodes is challenging for humans to comprehend. Secondly, unlike relational databases which come equipped with a prescriptive schema, RDF graphs lack regular structure or many times this structure exists but is unknown. Thirdly, size of the is challenging both for humans and for automated data processing tools. Fourthly, while RDF graphs may come equipped with ontologies, which specify the known relationships between the properties and classes present in the graph, the ontology itself is sometimes a source of complexity, especially if it is very large. In the presence of ontologies, graphs may contain implicit information, i.e. facts that hold in the graph despite not being physically present there. Reflecting the implicit facts of the ontology is in itself a challenge. Additionally specific parts of the ontology might not be used at all or very little in the specific Knowledge Base (KB).

Summarization has been applied to RDF data to facilitate these tasks. Its purpose is to extract concise and meaningful information from RDF knowledge bases, representing their content as faithfully as possible. There is no single concept of RDF summary, and not a single but many approaches to build such summaries.

Summarizing semantic graphs is a multifaceted problem with many dimensions, and thus many algorithms, methods and approaches have been developed to cope with it. As a result, there is now a confusion in the research community about the terminology in the area, further increased by the fact that certain terms are often used with different meanings in the relevant literature, denoting similar, but not identical research directions or concepts. We believe that this lack of terminology and classification hinders scientific development in this area.

Following up on a recent survey which we co-authored with colleagues [3], in this tutorial we present the main conceptual tools behind graph summarization, including some techniques developed prior to the advent of RDF, and show how all these techniques have been applied to the problems of summarizing semantic graphs. The goal of our tutorial is to acquaint the audience with the literature in this area, help them identify the tools and techniques most suited to the summarization problems they might have, and point out areas of interest for future work.

2 SCOPE

The tutorial aims at a broad range of researchers, students, IT professionals and practitioners, and developers. Anyone working with semantic graphs and RDF more specifically will benefit from this tutorial. Students and researchers will not only get a good introduction to the topic with a complete coverage of the state-of-the-art, but will also find a number of challenging research problems in these emerging technologies on which they may decide to focus their future research efforts. Practitioners will get a good overview of what the summarization algorithms, techniques and systems can offer nowadays and learn how they can use them to enhance their understanding of their available datasets. Developers of systems relying on semantic graphs will get helpful information that will help them improve further their products, enhancing query execution, data visualization and understanding.

Other tutorials [10, 15] considers the broad topic of summarizing large graphs, mostly using data mining tools, and with an approach tailored specifically to social networks; our tutorial focuses on the particularities of RDF graphs (and of their summarization).

Series ISSN: 2367-2005

10.5441/002/edbt.2019.38

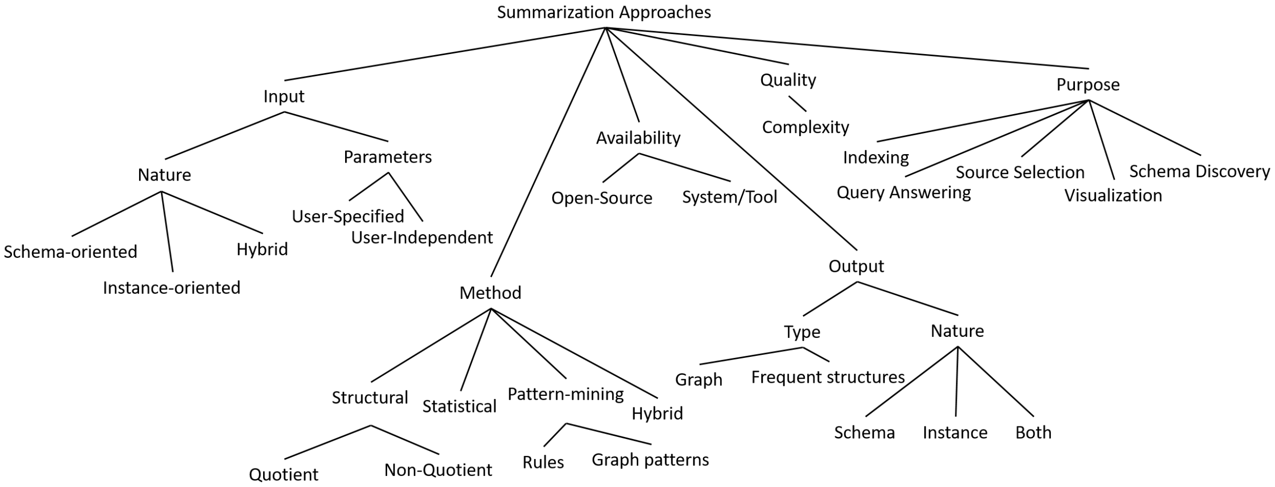


Figure 1: A taxonomy of the works in the area [3].

2.1 Tutorial goal

The goal of this tutorial is to introduce summarization notions and tools which are useful in order to concrete RDF data management applications. A broad set of techniques will be presented covering summarization of general RDF graphs, that contain or not ontological information, independent of their application domain.

3 OUTLINE

Our tutorial will be organized as follows.

3.1 Introduction and preliminaries

We will recall the basics of RDF graphs, RDFS and OWL ontologies, RDF queries (focusing in particular on conjunctive queries, the most frequently used in practice) and inference in RDF knowledge graphs in the presence of an ontology.

3.2 Applications

Next, we will present the main classes of application contexts which have justified the need for RDF summaries:

Indexing: The first and foremost application RDF summaries has been brought by the necessity of efficiently querying large and complex graphs. In this context, sets of nodes which are likely to be used together by queries are grouped together and their IDs are associated to a given summary node. Then, query processing proceeds in two stages: first, the summary nodes relevant to a given query are identified; then, from the summary node, an index lookup gives access directly to the respective data nodes.

Estimating the size of query results: To the same direction with indexing, summaries can be used to identify directly when no nodes are available for a specific query. More than this, summaries can also store statistical information on the available nodes, leading query optimizers to start query evaluation from the most selective conditions.

Source selection: Query evaluation across several graphs, in particular in a distributed setting where multiple graphs are each accessible behind its individual endpoint, can greatly benefit from the knowledge that one source (or one graph) does not have results matching a (sub)query. This is one facet of source selection: restricting query evaluation to avoid datasets on which it is guaranteed to have no answers. A variant consists of ordering the graphs to be explored (queried), so that in a finite time budget, the most interesting graphs are sure to be visited.

Graph visualization and schema discovery: Summarized information is easier to be visualized and comprehended. On the other hand when an ontology is not present, it can be extracted out of the available data, augmenting user understanding on the available information.

3.3 Classification & Dimensions

Then, we will present a classification of the available approaches according to the main algorithmic idea behind the summarization approach. We identify the following main categories; we also indicate a few of the relevant references (out of the 122 present in our survey [3]):

(1) **Structural methods** are those which summarize semantic graphs, based mostly on the graph structure, i.e. the paths and sub-graphs available in the RDF graph. Techniques in this category can be further categorized as quotient and non-quotient.
   - **Quotient** approaches are based on the idea of characterizing selected graph nodes as "equivalent" in a certain way, and then summarizing the graph by assigning a representative summary node to each class of equivalent graph nodes; further, each edge between two graph nodes leads to the corresponding edge being present between the two graph node's representatives. In a quotient summary, each property (edge label) from the input RDF graph is guaranteed present in the summary graph; more a quotient summary is guaranteed to be at most as large as the original graph. Different quotient summaries result from different notions of equivalence among the RDF graph nodes. Sample works in this area include [2, 4, 11, 22, 27], while [17] and [9] are important sources of inspiration from before the RDF age;
   - **Non-quotient** approaches are the remaining methods that are based mostly, on specific measures according to which the most important nodes are identified and then linked to formulate the presented summary. Pioneered by Dataguides [6, 20], the area features many recent works such as [21, 24, 29, 30]; [28] has an information retrieval approach, as it aims at extracting the most interesting triples to be shown to a user about a

subject; [32] summarizes ontologies found on the web, through the prism of the salience (interestingness) of their concepts.

(2) **Pattern mining methods** employee mining techniques to identify patterns appearing in the semantic graph. A pattern might be a set of instances having a certain set of properties, which are in exact or approximate terms representative of the graph or provide enough information on the graph using some cost function to determine that. We consider also as patterns the discovery of rules that can be used to reconstruct the graph and thus represent it adequately. Those patterns, together, compose the summary. Representative works in this area are, e.g., [8, 25, 33];
(3) **Statistical methods** on the other hand try to qualitatively summarize the contents of a graph counting occurrences, building histograms, measuring frequencies and other statistical measures out of the available semantic graph. This class comprises notably works such as [5, 7, 23, 31];
(4) Finally, several works combine techniques from several of the main areas listed above; these are **hybrid methods**, e.g., [1, 27] first and foremost aim at estimating the cardinality of query patterns, [26] summarizes RDF graphs and ontologies through the prism of statistics, while [19] aims at graph compression with bounded error, that is: a core (most regular) part of the graph is identified as comprising several copies of a same data pattern, and compressed into a single copy of this, whereas the rest is ignored from the summary and considered to be the summarization error (which the authors seek to minimize under certain constraints).

Further, we will characterize each of these proposal along a set of other informative dimensions:

(1) **Input:** An interesting dimension of analysis is the input required by each summarization method, as different approaches have usually different requirements for the dataset they get as input. RDF data graphs are usually accepted, RDF/S and/or OWL are considered for some of the works for specifying graph semantics whereas very few works consider DL models. Some works are based only on the ontology part whereas others consider only instances. Hybrid approaches are also available consuming both instances and the ontology for producing summaries. In addition, many works in the area require additional user input of fine tuning (e.g. summary size, weights, equivalence relations etc.) whereas some others are completely user independent.
(2) **Output:** Besides input, the available works might have also different output. The summary for example can be a graph or a selection of the most frequent structures such as nodes, paths, rules or queries. In addition we distinguish summaries that only output instances from those that output schema information as well.
(3) **Availability:** Several approaches are available by the authors as complete system/tool and some others provide only the corresponding algorithms/theory. Finally some systems are available online and can be readily tested.
(4) **Purpose:** As already explained in the applications of the summarization techniques, summaries can be build for indexing, source selection, visualisation, schema discovery or for facilitating query answering.
(5) **Quality:** Finally an important dimension of study, for each summarization algorithm is its completeness in terms of coverage, precision and recall of the result if an "ideal" summary is available as golden standard and its corresponding computational complexity.

Figure 1 presents the various dimensions that will be used in order to present the works available in each category.

Natural connections exist between the families of RDF summaries and the applications they are best suited for. Structural quotient summaries are most applicable to indexing and query answering through graph reduction; this holds especially for quotients built through equivalence relations such as bisimilarity (possibly bounded). Non-quotient summaries mostly target visualization, schema discovery and data understanding. Pattern mining summaries provide in many cases logical rules besides the summary graph as part of the final result, so could be possibly more useful in RDF graph compression scenarios. Summaries could also be really useful in data integration scenarios [14], where instead of generating mappings [16], [18] between data source schemas, summaries could be used to drive the definition of the mapping. Extending this to a scenario where the sources can also evolve [13], [12], summaries can play a key role in schema understanding and mapping redefinition.

3.4 Open issues and future research directions

RDF graph summaries can be useful in different application and research scenarios. Each scenario brings each own specific requirements and the possibility of having more than one items being suitable is present. One open issue in this respect is whether one could use the provided taxonomy to further automate the selection of the appropriate algorithms in the different use cases.

Identifying the *quality* of the RDF summary is also a difficult and not really widely addressed problem. The main problem that remains is how could one compare the summaries produced by the different algorithms and take into account the specificities of the problem at hand and provide an RDF summary with some guarantees. Given that even human experts do not agree on the quality of different summaries in many cases, this remains a challenging task.

Finally, one important problem that has been looked up very little is the *updates* of the RDF summaries produced, given the dynamic nature of most RDF datasets as well as their size. It is an open issue how one could update the summary without having to recompute the whole summary every time; and this problem has also a temporal dimension since one should answer not only how but also when this update is pertinent.

4 PRESENTERS

Haridimos Kondylakis is a scientific collaborator at the Institute of Computer Science, FORTH. His research interests span the following areas: Semantic Integration; Knowledge Evolution; Big Data Management; Data Series Indexing and Querying. He has extensive experience in participating in more than 16 European Projects and he also acts as a regular reviewer and a PC member for a number of premier journals and conferences. He has more than 110 publications in international conferences, books and journals including ACM SIGMOD, VLDB, VLDB Journal, JWS, KER, EDBT, ISWC, ESWC etc.

Dimitris Kotzinos is a Professor at the Department of Computer Science of the University of Cergy – Pontoise, member of the ETIS Lab and head of the MIDI team of the lab. His main research interests include data management algorithms, techniques and tools; development of methodologies, algorithms and tools for web-based information systems, portals and web services; and the understanding of the meaning (semantics) of interoperable data and services on the web.

Ioana Manolescu is a senior researcher, and the lead of the CEDAR team, joint between Inria Saclay and the LIX lab (UMR 7161) of Ecole polytechnique, in France. The CEDAR team research focuses on rich data analytics at cloud scale. She is a member of the PVLDB Endowment Board of Trustees, and a co-president of the ACM SIGMOD Jim Gray PhD dissertation committee. Recently, she has been a general chair of the IEEE ICDE 2018 conference, an associate editor for PVLDB 2017 and 2018, and the program chair of SSDBBM 2016. She has co-authored more than 130 articles in international journals and conferences, and contributed to several books. Her main research interests include data models and algorithms for computational fact-checking, performance optimizations for semistructured data and the Semantic Web, and distributed architectures for complex large data.

REFERENCES

[1] Anas Alzogbi and Georg Lausen. 2013. Similar Structures inside RDF-Graphs. In *Proceedings of the WWW2013 Workshop on Linked Data on the Web, Rio de Janeiro, Brazil, 14 May, 2013.*

[2] Šejla Čebirić, François Goasdoué, Pawel Guzewicz, and Ioana Manolescu. 2017. *Compact Summaries of Rich Heterogeneous Graphs.* Research Report RR-8920. INRIA Saclay ; Université Rennes 1. https://hal.inria.fr/hal-01325900

[3] Sejla Cebirić, François Goasdoué, Haridimos Kondylakis, Dimitris Kotzinos, Ioana Manolescu, Georgia Troullinou, and Mussab Zneika. 2018. Summarizing Semantic Graphs: A Survey. *The VLDB Journal* (2018). https://hal.inria.fr/hal-01925496 Accepted for publication, to appear.

[4] Mariano P Consens, Valeria Fionda, Shahan Khatchadourian, and Giuseppe Pirro. 2015. S+ EPPs: construct and explore bisimulation summaries, plus optimize navigational queries; all on existing SPARQL systems. *Proceedings of the VLDB Endowment* 8, 12 (2015), 2028–2031.

[5] Marek Dudás, Vojtech Svátek, and Jindrich Mynarz. 2015. Dataset Summary Visualization with LODSight. In *The Semantic Web: ESWC 2015 Satellite Events - ESWC 2015 Satellite Events Portorož, Slovenia, May 31 - June 4, 2015, Revised Selected Papers.* 36–40.

[6] Roy Goldman and Jennifer Widom. 1997. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece.* 436–445.

[7] Katja Hose and Ralf Schenkel. 2012. Towards benefit-based RDF source selection for SPARQL queries. In *Proceedings of the 4th International Workshop on Semantic Web Information Management, SWIM 2012, Scottsdale, AZ, USA, May 20, 2012.* 2.

[8] Amit Krishna Joshi, Pascal Hitzler, and Guozhu Dong. 2013. Logical Linked Data Compression. In *The Semantic Web: Semantics and Big Data, 10th International Conference, ESWC 2013, Montpellier, France, May 26-30, 2013. Proceedings.* 170–184.

[9] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. 2002. Covering indexes for branching path queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002.* 133–144.

[10] Arijit Khan, Sourav S. Bhowmick, and Francesco Bonchi. 2017. Summarizing Static and Dynamic Big Graphs. *PVLDB* 10, 12 (2017), 1981–1984.

[11] Shahan Khatchadourian and Mariano P. Consens. 2010. ExpLOD: Summary-Based Exploration of Interlinking and RDF Usage in the Linked Open Data Cloud. In *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part II.* 272–287.

[12] Haridimos Kondylakis and Dimitris Plexousakis. 2011. Ontology Evolution in Data Integration: Query Rewriting to the Rescue. In *Conceptual Modeling - ER 2011, 30th International Conference, ER2011, Brussels, Belgium, October 31 - November 3, 2011. Proceedings.* 393–401.

[13] Haridimos Kondylakis and Dimitris Plexousakis. 2012. Ontology Evolution: Assisting Query Migration. In *Conceptual Modeling - 31st International Conference ER 2012, Florence, Italy, October 15-18, 2012. Proceedings.* 331–344.

[14] Haridimos Kondylakis and Dimitris Plexousakis. 2013. Ontology evolution without tears. *J. Web Sem.* 19 (2013), 42–58.

[15] Shou-De Lin, Mi-Yen Yeh, and Cheng-Te Li. 2013. Sampling and Summarization for Social Networks (tutorial).

[16] Yannis Marketakis, Nikos Minadakis, Haridimos Kondylakis, Konstantina Konsolaki, Georgios Samaritakis, Maria Theodoridou, Giorgos Flouris, and Martin Doerr. 2017. X3ML mapping framework for information integration in cultural heritage and beyond. *Int. J. on Digital Libraries* 18, 4 (2017), 301–319.

[17] Tova Milo and Dan Suciu. 1999. Index Structures for Path Expressions. In *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings.* 277–295.

[18] Nikos Minadakis, Yannis Marketakis, Haridimos Kondylakis, Giorgos Flouris, Maria Theodoridou, Gerald de Jong, and Martin Doerr. 2015. X3ML Framework: An Effective Suite for Supporting Data Mappings. In *Proceedings of the Workshop on Extending, Mapping and Focusing the CRM co-located with 19th International Conference on Theory and Practice of Digital Libraries (2015), Poznań, Poland, September 17, 2015.* 1–12.

[19] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. 2008. Graph summarization with bounded error. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008.*

[20] Svetlozar Nestorov, Jeffrey D. Ullman, Janet L. Wiener, and Sudarshan S. Chawathe. 1997. Representative Objects: Concise Representations of Semistructured, Hierarchical Data. In *ICDE.*

[21] Alexandros Pappas, Georgia Troullinou, Giannis Roussakis, Haridimos Kondylakis, and Dimitris Plexousakis. 2017. Exploring Importance Measures for Summarizing RDF/S KBs. In *The Semantic Web - 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28 - June 1, 2017, Proceedings, Part I.* 387–403.

[22] François Picalausa, Yongming Luo, George H. L. Fletcher, Jan Hidders, and Stijn Vansummeren. 2012. A Structural Approach to Indexing Triples. In *The Semantic Web: Research and Applications - 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings.*

[23] Carlos Eduardo S. Pires, Paulo Orlando Queiroz-Sousa, Zoubida Kedad, and Ana Carolina Salgado. 2010. Summarizing ontology-based schemas in PDMS. In *Workshops Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA.* 239–244.

[24] Paulo Orlando Queiroz-Sousa, Ana Carolina Salgado, and Carlos Eduardo S. Pires. 2013. A Method for Building Personalized Ontology Summaries. *JIDM* 4, 3 (2013), 236–250.

[25] Qi Song, Yinghui Wu, and Xin Luna Dong. 2016. Mining Summaries for Knowledge Graph Search. In *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain.* 1215–1220.

[26] Blerina Spahiu, Riccardo Porrini, Matteo Palmonari, Anisa Rula, and Andrea Maurino. 2016. ABSTAT: Ontology-driven Linked Data Summaries with Pattern Minimalization. In *SumPre.*

[27] Giorgio Stefanoni, Boris Motik, and Egor V. Kostylev. 2018. Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018.* 1043–1052. https://doi.org/10.1145/3178876.3186003

[28] Marcin Sydow, Mariusz Pikula, and Ralf Schenkel. 2013. The notion of diversity in graphical entity summarisation on semantic knowledge graphs. *J. Intell. Inf. Syst.* 41, 2 (2013), 109–149.

[29] Georgia Troullinou, Haridimos Kondylakis, Evangelia Daskalaki, and Dimitris Plexousakis. 2017. Ontology understanding without tears: The summarization approach. *Semantic Web* 8, 6 (2017), 797–815.

[30] Georgia Troullinou, Haridimos Kondylakis, Kostas Stefanidis, and Dimitris Plexousakis. 2018. Exploring RDFS KBs Using Summaries. In *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part I.* 268–284.

[31] Gang Wu, Juanzi Li, Ling Feng, and Kehong Wang. 2008. Identifying Potentially Important Concepts and Relations in an Ontology. In *The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings.* 33–49.

[32] Xiang Zhang, Gong Cheng, Weiyi Ge, and Yuzhong Qu. 2009. Summarizing Vocabularies in the Global Semantic Web. *J. Comput. Sci. Technol.* 24, 1 (2009), 165–174.

[33] Mussab Zneika, Claudio Lucchese, Dan Vodislav, and Dimitris Kotzinos. 2016. Summarizing Linked Data RDF Graphs Using Approximate Graph Pattern Mining. In *EDBT 2016.* 684–685.

Schemas And Types For JSON Data

Mohamed-Amine Baazizi
Sorbonne Université, LIP6 UMR 7606
France
baazizi@ia.lip6.fr

Dario Colazzo
Université Paris-Dauphine, PSL Research University
France
dario.colazzo@dauphine.fr

Giorgio Ghelli
Dipartimento di Informatica
Università di Pisa
Pisa, Italy
ghelli@di.unipi.it

Carlo Sartiani
DIMIE
Università della Basilicata
Potenza, Italy
sartiani@gmail.com

ABSTRACT

The last few years have seen the fast and ubiquitous diffusion of JSON as one of the most widely used formats for publishing and interchanging data, as it combines the flexibility of semistructured data models with well-known data structures like records and arrays. The user willing to effectively manage JSON data collections can rely on several schema languages, like JSON Schema, JSound, and Joi, or on the type abstractions offered by modern programming languages like Swift or TypeScript.

The main aim of this tutorial is to provide the audience with the basic notions for enjoying all the benefits that schemas and types can offer while processing and manipulating JSON data. This tutorial focuses on four main aspects of the relation between JSON and schemas: (1) we survey existing schema language proposals and discuss their prominent features; (2) we review how modern programming languages support JSON data as first-class citizens; (3) we analyze tools that can infer schemas from data, or that exploit schema information for improving data parsing and management; and (4) we discuss some open research challenges and opportunities related to JSON data.

1 INTRODUCTION

The last two decades have seen a dramatic change in the data processing landscape. While at the end of the last century data were usually very structured and managed inside relational DBMSs, nowadays they have very different characteristics: they are big, usually semistructured or even unstructured, without a rigid and predefined schema, and hosted and produced in data processing platforms that do not embrace the relational model. In this new scenario, where data come without a schema, and multiple data models coexist, JSON is affirming as a useful format for publishing and exchanging data, as it combines the flexibility of XML with well-known data structures like records and arrays. JSON is currently employed for publishing and sharing data in many application fields and for many different purposes: for instance, JSON is used as the result format for many web site APIs (e.g., Twitter, New York Times), as a common format for the remote interaction of modern web applications (e.g., Facebook's GraphQL is entirely based on JSON), as a common format for exchanging scientific data as well as public open data (e.g., the U.S. Government's open data platform: `https://www.data.gov`).

Given the wide diffusion of JSON and its use in scientific as well as mainstream applications, the need to directly manipulate JSON data inside applications rapidly emerged. To this aim, a schema language, specifically designed for JSON, has been introduced, but its adoption is not growing at a fast pace, since its specification is somewhat complex and many modern programming languages, like Swift and TypeScript, directly support JSON data through their own, simple type systems; furthermore, Walmart Labs endowed JavaScript, which is inherently untyped, with a powerful schema language for JSON objects by means of JavaScript function calls.

In this tutorial proposal, we will present and discuss existing schema and type languages for JSON data, and compare their features; we will also discuss several schema-related tools, with a particular focus on approaches for schema inference. The main aim of this tutorial is to provide the audience and developers with the basic notions for enjoying all the benefits that schemas and types can offer while processing, analyzing, and manipulating JSON data.

Outline. This *1.5-hour* tutorial is split into five main parts:

(1) **JSON primer (∼ 10 min.)**. In this very introductory part of the tutorial, we review the basic notions about JSON together with its JavaScript legacy, and present a few examples, coming from publicly available datasets, that we will use throughout the remaining parts of the tutorial.
(2) **Schema languages (∼ 20 min.)**. In this part of the tutorial we focus on existing schema languages for JSON data collections and discuss their most prominent features.
(3) **Types in Programming Languages (∼ 15 min.)**. In this part of the tutorial we review how modern programming languages support JSON data as first class citizens. In particular, we focus on programming and scripting languages for web and/or mobile applications, where JSON data interchange is a crucial task.
(4) **Schema Tools (∼ 30 min.)**. In this part of the tutorial we analyze tools that exploit schema information for improving JSON data processing. We focus on the problem of inferring a meaningful schema for schemaless JSON collections, as well as on the exploitation of schema information for improving data parsing and management.
(5) **Future Opportunities (∼ 10 min.)**. Finally, we outline open research problems as potential directions for new research in this area.

In what follows we describe at a very high level the technical content covered in each of the last four aforementioned parts.

2 SCHEMA LANGUAGES

In this part of the tutorial we will focus our attention on several schema languages for JSON data, with particular emphasis on JSON Schema [4] and Walmart Labs Joi [6].

JSON Schema emerged in the academic community and has been developed without a specific programming or scripting language in mind. JSON Schema allows the programmer to specify a schema for any kind of JSON values, and supports traditional type constructors, like union and concatenation, as well as very powerful constructors like negation types.

JSON Schema has already been studied. Indeed, in [21], motivated by the need of laying the formal foundations for the JSON Schema language [4], Pezoa et al. present the formal semantics of that language, as well as a theoretical study of its expressive power and validation problem. Along the lines of [21], Bourhis et al. [15] have recently laid the foundations for a logical characterization and formal study for JSON schema and query languages.

On the contrary, Joi has been developed by Walmart as a tool for (i) creating schemas for JSON objects and (ii) ensuring the validation of objects inside an untyped scripting language like JavaScript; furthermore, to the best of our knowledge, Joi has not been studied so far. Joi only allows the designer to describe the schema for JSON objects, but it still provides the ability to specify co-occurrence and mutual exclusion constraints on fields, as well as union and value-dependent types.

We will analyze the most prominent features of these languages, and compare their capabilities in a few scenarios. We will also briefly discuss JSound [5], an alternative, but quite restrictive, schema language, as well as a few other schema-related proposals, such that described in [24], where Wang et al. present a framework for efficiently managing a schema repository for JSON document stores. The proposed approach relies on a notion of JSON schema called *skeleton*. In a nutshell, a skeleton is a collection of trees describing structures that frequently appear in the objects of a JSON data collection. In particular, the skeleton may totally miss information about paths that can be traversed in some of the JSON objects.

3 TYPES IN PROGRAMMING LANGUAGES

Unlike XML, which found no space as a first class citizen in programming languages, with the obvious and notable exception of XQuery, JSON has been designed starting from the object language of an existing scripting language. Therefore, given its wide use in web and mainstream application development, JSON support has been introduced in several strongly typed programming and/or scripting languages.

To directly and naturally manage JSON data a programming language should incorporate the ability to express record types, sequence types, and union types. While record and sequence types can be easily found in many programming languages, union types are quite rare and usually confined to functional languages only.

In this part of the tutorial we will discuss the support for JSON objects inside the type systems of TypeScript [9] and Swift [8]. TypeScript is a typed extension of JavaScript, while Swift is an Apple-backed programming language that is rapidly becoming the language of choice for developing applications in the Apple ecosystem (iOS + macOS). These languages show similar features, but also very significant differences in the treatment of JSON objects.

We will also compare the features offered by these languages with those of the schema languages we presented in the second part of the tutorial.

4 SCHEMA TOOLS

In this part of the tutorial we will present several schema-related tools for JSON data. We will first discuss existing approaches for inferring a schema starting from a dataset and then move to parsing tools that are able to exploit dynamic type information to speed-up data parsing.

4.1 Schema Inference

Several schema inference approaches for JSON data collections have been proposed in the past. In [10–12] authors describe a distributed, parametric schema inference approach capable of inferring schemas at different levels of abstraction. In the context of Spark, the Spark Dataframe schema extraction [7] is a very interesting tool for the automated extraction of a schema from JSON datasets; this tool infers schemas in a distributed fashion, but, unlike the technique described in [10–12], its inference approach is quite imprecise, since the type language lacks union types, and the inference algorithm resorts to `Str` on strongly heterogeneous collections of data. Other systems, like Jaql [13], exploit schema information for inferring the output schema of a query, but still require an externally supplied schema for input data, and perform output schema inference only locally on a single machine.

There are also a few inference tools for data stored in NoSQL systems and RDBMSs. Indeed, in the context of NoSQL systems (e.g. MongoDB), recent efforts have been dedicated to the problem of implementing tools for JSON schema inference. A JavaScript library for JSON, called mongodb-schema, is presented in [22]. This tool analyzes JSON objects pulled from MongoDB, and processes them in a streaming fashion; it is able to return quite concise schemas, but it cannot infer information describing field correlation. Studio 3T [19] is a commercial front-end for MongoDB that offers a very simple schema inference and analysis feature, but it is not able to merge similar types, and the resulting schemas can have a huge size, which is comparable to that of the input data. In [23], a python-based tool is described, called Skinfer, which infers JSON Schemas from a collection of JSON objects. Skinfer exploits two different functions for inferring a schema from an object and for merging two schemas; schema merging is limited to record types only, and cannot be recursively applied to objects nested inside arrays. Couchbase, finally, is endowed with a schema discovery module which classifies the objects of a JSON collection based on both structural and semantic information [3]. This module is meant to facilitate query formulation and select relevant indexes for optimizing query workloads.

When moving to RDBMSs, in [16] Abadi and al. deal with the problem of automatically transforming denormalised, nested JSON data into normalised relational data that can be stored in a RDBMS; this is achieved by means of a schema generation algorithm that learns the *normalised, relational* schema from data. This approach ignores the original structure of the JSON input dataset and, instead, depends on patterns in the attribute data values (functional dependencies) to guide its schema generation.

4.2 Parsing

There are a few novel parsing tools for JSON data that take into account dynamic type information for improving the efficiency of the applications relying on them.

In a recent work [20], Li et al. present streaming techniques for efficiently parsing and importing JSON data for analytics tasks; these techniques are then used in a novel C++ JSON parser, called Mison, that exploits AVX instructions to speed up data parsing and discarding unused objects. To this end, it infers structural information of data on the fly in order to detect and prune parts of the data that are not needed by a given analytics task.

In [14], Bonetta and Brantner present Fad.js, a *speculative*, JIT-based JSON encoder and decoder designed for the Oracle Graal.js JavaScript runtime. It exploits data access patterns to optimize both encoding and decoding: indeed, Fad.js relies on the assumption that most applications never use all the fields of input objects, and, for instance, skips unneeded object fields during JSON object parsing.

5 FUTURE OPPORTUNITIES

We finally discuss several open challenges and opportunities related to JSON schemas, including the following ones.

Schema Inference and ML. While all schema inference approaches covered in the previous part of the tutorial are based on traditional techniques, a recent work by Gallinucci et al. [17] shows the potential benefits of ML approaches in schema inference; furthermore, ML-based inference techniques have already been used for non-JSON data, as shown by Halevy et al. in [18]. Hence, a promising research direction is to understand how these methods can be efficiently applied to large collections of data and whether they can overcome some limitations of previous approaches.

Schema-Based Data Translation. While JSON is very frequently used for exchanging and publishing data, it is hardly used as internal data format in Big Data management tools, that, instead, usually rely on formats like Avro [1] and Parquet [2]. When input datasets are heterogeneous, schemas can improve the efficiency and the effectiveness of data format conversion. Therefore, a major opportunity is to design schema-aware data translation algorithms that are driven by schema information and use it to improve the quality of the translation.

6 INTENDED AUDIENCE AND COVERAGE

Our goal is to present a coherent starting point for EDBT attendees who are interested in understanding the foundations and applications of schemas and types for JSON data processing. We will not assume any background in JSON schema languages, but will introduce them starting from the roots, giving broad coverage of many of the key ideas, making it appropriate for graduate students seeking new areas to study and researchers active in the field alike.

7 BIOGRAPHICAL SKETCHES

Mohamed-Amine Baazizi (Ph.D.) is an assistant professor at Sorbonne Université. He received his PhD from Université of Paris-Sud and completed his postdoctoral studies in Télécom Paristech. His research focuses on exploiting schema information for optimizing the processing of semi-structured data.

Dario Colazzo (Ph.D.) is Full Professor in Computer Science at LAMSADE - Université Paris-Dauphine. He received his PhD from Università di Pisa, and he completed his postdoctoral studies at Università di Venezia and Université Paris Sud. His main research activities focus on static analysis techniques for large scale data management.

Giorgio Ghelli (Ph.D.) is Full Professor in Computer Science, at Università di Pisa. He was Visiting Professor at École Normale Supérieure Paris, at Microsoft Research Center, Cambridge (UK), and at Microsoft Co. (Redmond, USA), member of the W3C XML Query Working Group, member of the board of the EAPLS. He worked on database programming languages and type systems for these languages, especially in the fields of object oriented and XML data models.

Carlo Sartiani (Ph.D.) is an assistant professor at Università della Basilicata. He received his PhD from Università di Pisa, and he completed his postdoctoral studies at Università di Pisa. He worked on database programming languages and data integration systems, and his current research activities focus on semistructured and big data.

REFERENCES

[1] Apache Avro. `https://avro.apache.org`.
[2] Apache Parquet. `https://parquet.apache.org`.
[3] Couchbase auto-schema discovery. https://blog.couchbase.com/auto-schema-discovery/.
[4] JSON Schema language. `http://json-schema.org`.
[5] JSound schema definition language. `http://www.jsoniq.org/docs/JSound/html-single/index.html`.
[6] Object schema description language and validator for JavaScript objects. https://github.com/hapijs/joi.
[7] Spark Dataframe. `https://spark.apache.org/docs/latest/sql-programming-guide.html`.
[8] Swift. `https://swift.org`.
[9] TypeScript. `https://www.typescriptlang.org`.
[10] Mohamed Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Schema Inference for Massive JSON Datasets. In *EDBT '17*.
[11] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Counting types for massive JSON datasets. In *Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017*. 9:1–9:12.
[12] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Parametric schema inference for massive JSON datasets. *The VLDB Journal* (2019). https://doi.org/10.1007/s00778-018-0532-7
[13] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene J. Shekita. 2011. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB* 4, 12 (2011), 1272–1283.
[14] Daniele Bonetta and Matthias Brantner. 2017. FAD.js: Fast JSON Data Access Using JIT-based Speculative Optimizations. *PVLDB* 10, 12 (2017), 1778–1789. http://www.vldb.org/pvldb/vol10/p1778-bonetta.pdf
[15] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. 2017. JSON: Data model, Query languages and Schema specification. In *PODS '17*. 123–135.
[16] Michael DiScala and Daniel J. Abadi. 2016. Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data. In *SIGMOD '16*. 295–310.
[17] Enrico Gallinucci, Matteo Golfarelli, and Stefano Rizzi. 2018. Schema profiling of document-oriented databases. *Inf. Syst.* 75 (2018), 13–25.
[18] Alon Y. Halevy, Flip Korn, Natalya Fridman Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. 2016. Goods: Organizing Google's Datasets. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 795–806.
[19] 3T Software Labs. 2017. Studio 3T. `https://studio3t.com`.
[20] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. *PVLDB* 10, 10 (2017), 1118–1129.
[21] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON Schema. In *WWW '16*. 263–273.
[22] Peter Schmidt. 2017. mongodb-schema. `https://github.com/mongodb-js/mongodb-schema`.
[23] scrapinghub. 2015. Skinfer. `https://github.com/scrapinghub/skinfer`.
[24] Lanjun Wang, Shuo Zhang, Juwei Shi, Limei Jiao, Oktie Hassanzadeh, Jia Zou, and Chen Wangz. 2015. Schema Management for Document Stores. *Proc. VLDB Endow.* 8, 9 (May 2015), 922–933.

Influence Maximization Revisited: The State of the Art and the Gaps that Remain

Akhil Arora
EPFL Lausanne
akhil.arora@epfl.ch

Sainyam Galhotra
UMass Amherst
sainyam@cs.umass.edu

Sayan Ranu
IIT Delhi
sayanranu@cse.iitd.ac.in

ABSTRACT

The steady growth of graph data from social networks has resulted in wide-spread research on the influence maximization (IM) problem. This results in extension of the state-of-the-art almost every year. With the recent *explosion* in the application of IM in solving *real-world* problems, it is no longer a theoretical exercise. Today, IM is used in a plethora of real-world scenarios, with *OnePlus*[1] series of mobile phones, *Hokey Pokey*[2] ice-creams, and *galleri5 influencer marketplace*[3] being the most prominent industrial use-cases. Given this scenario, navigating the *maze* of IM techniques to get an in-depth understanding of their utilities is of *prime* importance. In this tutorial, we address this *paramount* issue and solve the *dilemma* of "*Which IM technique to use and under What scenarios*"? "*What does it really mean to claim to be the state-of-the-art*"?

This tutorial builds upon our benchmarking study [1], and will provide a concise and intuitive overview of the most important IM techniques, which is usually lost in the technical literature. Specifically, we will unearth a series of *incorrect claims* made by prominent IM papers, disseminate the inherent *deficiencies* of existing approaches, and surface the *open challenges* in IM even after a decade of research.

1 MOTIVATION

Influence maximization (IM) has been one of the most actively studied areas of data management research over the past decade. With this, almost every year, a new IM technique has been published that claims to be the state-of-the-art. However, IM is no longer a theoretical problem. We rely on *Facebook* and *WhatsApp* to communicate with friends. *Twitter* is used to disseminate information such as traffic-news, emergency-services, etc. IM is used by companies to publicize their products or shape opinions (Ex: *OnePlus*, *galleri5*, and *HokeyPokey* [19]).

On the academic front, researchers are interested in classical IM [3, 7–10, 16–18, 21, 24, 25, 31] as well as more application-specific models such as IM under *competition* [22], *time and opinion-aware* IM [6, 12] etc. Undoubtedly, this extensive research has promoted prosperity of the family of IM techniques. However, it also raises several questions that are not adequately addressed. Given this widespread applicability, it is important to understand the following questions from a *neutral standpoint*.

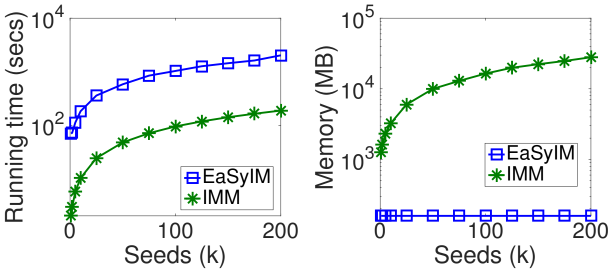


Figure 1: Comparing IMM ($\epsilon = 0.5$) with EaSyIM ($iter = 100$) under IC ($W(u,v) = 0.1$) on the YouTube dataset.

- *Which IM technique should one use given the resources in hand? How to choose the most appropriate IM technique in a given specific scenario?*
- *What does it mean to claim to be the state-of-the-art? More fundamentally, Is there really a single state-of-the-art technique as is often claimed?*
- *Are the claims made by the recent papers true?*
- *What are the unsolved challenges in the field?*

To highlight the ambiguity that plagues the current maze of IM techniques, we provide a concrete example[4] to motivate the need for answering the questions stated above.

What does it mean to be the state of the art? While many techniques claim to be the state of the art, in reality, they are often the state of the art in only one aspect of the IM problem. Consider Figs. 1a-1b, where EaSyIM [12] and IMM [30] scale better with respect to memory and running time respectively. Thus, neither technique can be termed as better than the other.

1.1 Relevance and Timeliness

• First, EDBT is an appropriate platform to present a tutorial on IM from a *neutral standpoint* since reproducibility tests and benchmarking have always been a key area of interest of the database community at large. Moreover, as shown in Fig. 2, of late database conferences have become the venue of choice for authors conducting research in the field of influence maximization, since ensuring scalability (while maintaining quality guarantees) has become central to the problems identified in this area.

• Second, to ensure a streamlined growth of the field, this tutorial, in addition to surveying existing IM techniques, serves as a *timely* and *relevant* avenue to *disseminate answers of the questions stated above to the data management community*. Overall, the tutorial will build upon our benchmarking study [1] and present our musings over almost a decade long literature (along with the recent advances) in the field of IM from top publication venues, and *unravel* many interesting and unknown avenues in the well-studied area of influence maximization.

[1] https://oneplusstore.in
[2] http://www.hokeypokey.in
[3] https://galleri5.com/aboutus

[4] A detailed analysis on more such examples may be found in [1].

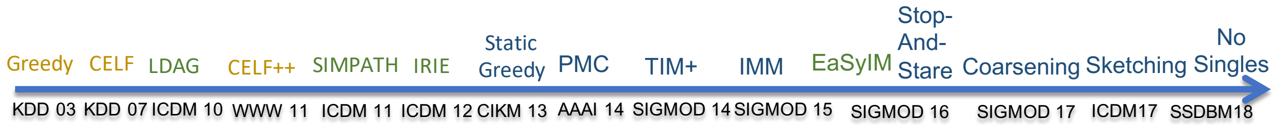


Figure 2: Timeline showcasing the Evolution of IM techniques.

- Lastly, as mentioned previously, IM is no longer a theoretical problem. It is regularly used by companies to publicize their products or shape opinions. *OnePlus*, *galleri5*, and *HokeyPokey* [19] for example rely completely on IM through social networks. To this end, the insights presented in this tutorial would definitely be advantageous to a broad audience at EDBT/ICDT ranging from theorists to researchers who are more interested in understanding and harnessing the practical power of IM in social networks.

2 INTENDED AUDIENCE, PREREQUISITE KNOWLEDGE AND LENGTH

The tutorial is aligned to the general area of data management and the web, thereby being relevant for a broad audience at EDBT: including students, academic researchers, and industrial experts specifically interested in benchmarking, data mining, social-network analysis, large-scale analytics, and performance tuning. No prior knowledge beyond basic probability and graph theory is expected. Familiarity with information-diffusion concepts would help, but not needed. The tutorial is self-contained and possesses introduction of most of the foundational concepts.

The key take away would be knowledge of the gaps including mis-claims and myths, leading to some of the never unraveled aspects of the IM problem, thereby enabling a more streamlined advancement in IM research. Since IM is a hot topic, we expect around 50 participants.

3 OUTLINE OF THE TUTORIAL

3.1 Introduction (20 minutes)

The first part of the tutorial will involve the formal definition of the IM problem along with an analysis of the fundamental information diffusion models: IC and LT [18]. The other aspect here would be to motivate the importance of IM, by citing real-world applications, in order to bridge the gap between theoretical models and real-world information diffusion.

Moving ahead, we will explain in detail the various challenges faced in designing effective solutions for the IM problem. We will analyze various properties of IM, namely – NP-hardness, submodularity etc., and also present the scenarios where exact estimation of influence is possible.

3.2 Summary of IM Algorithms (30 minutes)

First, we will present a categorized overview [1] of existing IM algorithms. This will enable the attendees to grasp the broad spectrum of IM techniques as portrayed in Table 1 in an intuitive and concise manner. Next, we will delve into a detailed description of each category.

Given that the IM problem is NP hard, Kempe et al. [18] leverage submodularity to propose a Greedy algorithm that provides the best approximation on the *quality* of obtained *spread*. Later, CELF [21] and CELF++ [15] were proposed to maintain the same quality of *spread* with an attempt to improve the efficiency by applying several optimizations over the Greedy algorithm.

Next, we will present the *heuristics* IMRank/IRIE [8, 17] and LDAG/SIMPATH [7, 16] that improve the *efficiency* and *scalability* aspect of IM, and perform well for the WC and LT models respectively. The caveat with these techniques is that they work well in practice, however lack any theoretical backing on the *quality* of the obtained *spread*. We will also introduce our techniques – *ASIM* [13] and *EaSyIM* [12], which are better both empirically and theoretically when compared to other heuristics.

Lastly, we will present a recent class of techniques that use *sampling* [9, 24, 25, 30, 31] to portray superior *efficiency* while retaining *quality* guarantees. These techniques either maintain reverse reachable (RR) sets of nodes or snapshots of cascades, and try to estimate influential nodes by sampling nodes from the original network. The caveat here is that most of these techniques are not scalable owing to their *exorbitantly* high memory footprint [1].

Table 1 summarizes the techniques discussed above, while stating their key highlights and the respective state-of-the-arts. This will enable the attendees to understand the representative techniques in the literature and the different aspects they address. It will also facilitate the attendees to appreciate as to why "*One Size Doesn't Fit All!*".

Finally, we will analyze why there does not exist any algorithm capable of simultaneously excelling in all the three fronts: (1) efficiency, (2) scalability, and (3) quality?

3.3 Myths, Mis-Claims and Insights (20 minutes)

In continuation to the overview of the techniques, here, we present our findings and provide recommendations to answer the question(s) posed by us in Section 1. We firmly establish that several *claims* from highly cited papers are *incorrect* (our experiments have been marked *SIGMOD Reproducible*), the evaluation procedure adopted by various techniques could produce misleading results, and expose a series of *myths* that could potentially alter the way we approach IM research. All the *insights* would be supported by *empirical* results, presented to the audience using a *python interface* to the publicly available implementations[5] of the discussed techniques.

3.4 Open Challenges and Future Directions (20 minutes)

The last part of the tutorial would focus towards *summarizing* the key insights discussed previously to eventually *shortlist* the best technique(s) and the corresponding scenarios in which they are the best. To this end, a decision-tree (Fig. 3b) will be presented as a tool to the audience. Next, we would delve into a detailed discussion on the open

[5] For details please visit our project page: https://sigdata.github.io/infmax-benchmark

Type	Theoretical Guarantee?	Highlights	State-of-the-Art
Greedy and Optimizations	Yes	Superior Quality and Scalability but low Efficiency	CELF/CELF++ [15, 21]
Heuristics	No	Superior Efficiency and Scalability at the cost of Quality	*EaSyIM*, IRIE, & LDAG/SIMPATH [7, 12, 16, 17]
Sampling Snapshots/RR sets	Yes	Superior Efficiency and Quality at the cost of Scalability	PMC, Stop-and-stare, Coarsening, Sketching, and NoSingles [23–27]

Table 1: The spectrum of IM techniques.

challenges in the field of information propagation, thereby providing a streamlined view of future research directions.

- The most important research direction is the development of a *scalable* and *efficient* algorithm with error *guarantees* (Fig. 3a), which still remains as the holy grail of influence maximization. While recent efforts by Popova et al. [27], Ohsaka et al. [26], and Nguyen et al. [23] are steps in this direction to improve scalability of the class of memory-intensive sampling algorithms [10, 24, 30, 31], more work is needed to achieve true scalability. To this end, there is a need for a generic framework inspired by classical data management systems which are shown to perform well for managing graph data [11], or the use of modern data management technologies that rely on distribution and parallelization to improve scalability and efficiency.
- Another compelling and novel research direction lies in validating the correctness/effectiveness of the classical information diffusion models proposed in [18] using real-world social media data capturing cascades from retweets or mentions as ground truth. The advantage of this exercise would be two fold: (1) Exploring the most unfathomed area in the field provides tremendous scope for advancement of the state-of-the-art, and (2) Curating a benchmark dataset for all the follow-up research. Such efforts would also attract further research on scalably learning influence probabilities from real-world interaction data extending on the works of [14] and [20].
- The challenges involved in scaling up influence maximization to massive networks under classical information diffusion models also cascade to the recent research activities around development of sophisticated diffusion models like opinion-aware [12], topic-aware [5] etc., which too are deprived of scalable algorithms. To this end, there is a need for devising a generic and unified framework for scaling up influence maximization under classical and various sophisticated real-world scenarios.

4 RELATED TUTORIALS

Multiple tutorials have been presented in the broad field of information propagation and IM at major data-centric venues [2, 4, 28, 29]. However, all these tutorials possess a common theme, i.e., each of them have provided an overview of models for information diffusion in networks and associated algorithms for influence analysis. While [2, 4, 29] were based on an algorithmic and data-mining perspective of the broad area of information diffusion, [28] focused on machine learning methods, specifically encircling the problems of network inference, influence estimation and control. In sum, the main focus of these tutorials was towards dissemination of the mathematical, technological, and algorithmic innovations to all-and-sundry, thereby enabling a step forward for sound analysis of research problems in the field of information propagation.

The proposed tutorial has the following key differentiations:

- First, the state-of-the-art has never been discussed from a neutral standpoint. More fundamentally, all the previous tutorials have given overviews of the existing literature, however, this exercise has not been from a perspective of a critic. We are the first to present some of the *highly controversial and ground-breaking discoveries*, thereby unraveling several *mis-claims and myths* in the existing IM research. This in-depth focus of our tutorial enables a more streamlined advancement in IM research with possible redefinition of the state-of-the-art.
- Second, IM is still a niche problem and provides many avenues to devise real world models and scalable algorithms capable of tackling competitive, time aware, opinion aware settings and many more. The knowledge gathered from this tutorial would facilitate more informed extensions to these settings.

5 TUTOR BIOGRAPHY AND EXPERTISE

Akhil Arora is a doctoral researcher at EPFL. His research interests include large scale data mining, databases, and machine learning. He is a recipient of the prestigious "EDIC Doctoral Fellowship" for the academic year 2018-19, and the "Most Reproducible Paper" award at SIGMOD 2018. He has published his research in prestigious data mining and database conferences, served as a reviewer, and co-organized workshops in these conferences. Further information available at https://www.cse.iitk.ac.in/~aarora.

Sainyam Galhotra is a graduate student at UMass Amherst. His research interests include graph analysis, data mining and integration. He is the recipient of the "Best Paper Award" at FSE 2017 and the "Most Reproducible Award" at SIGMOD 2018. He is the first recipient of the "Krithi Ramamritham Scholarship" at UMass for contribution to research in databases. He has published in top data mining, database and machine learning conferences. Further information available at https://people.cs.umass.edu/~sainyam.

Sayan Ranu is an assistant professor in the Computer Science department at IIT Delhi. His research interests include graph mining, spatio-temporal data analytics, and bioinformatics. He was a recipient of the Best Paper Award at WISE 2016 and the "Most Reproducible Paper" award at SIGMOD 2018. Sayan regularly serves in the program committees of conferences and journals including KDD, ICDE, WWW, ICDM, TKDE, VLDB Journal. Further information available at http://www.cse.iitd.ac.in/~sayan/.

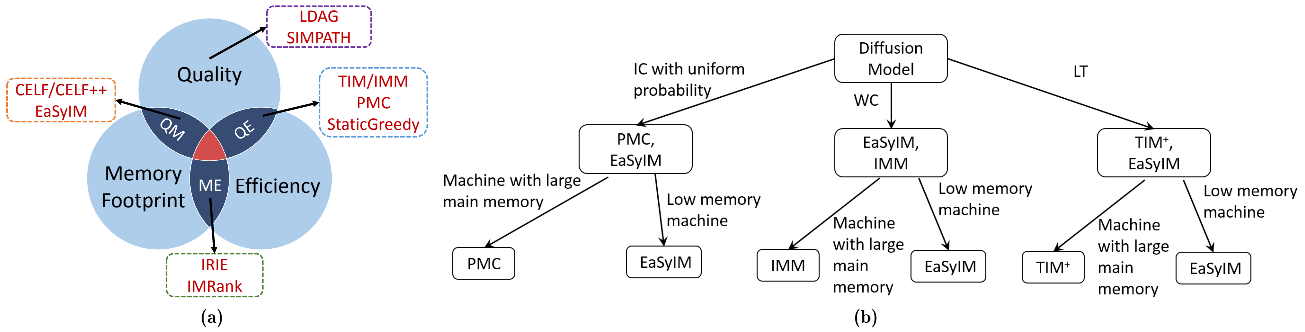


Figure 3: (a) Summarizing the spectrum of Influence Maximization (IM) techniques based on their strengths. (b) The decision tree for choosing the most appropriate IM algorithm.

5.1 Tutorials given by Authors

The authors possess adequate experience of delivering tutorials at reputed venues as indicated below:

- **Akhil Arora, Sainyam Galhotra, Sayan Ranu**, Shourya Roy: "Influence Maximization Revisited", COMAD 2018.
- **Sayan Ranu** and Ambuj Singh: "Indexing and mining topological patterns for drug discovery", in EDBT 2012.
- **Sayan Ranu** and Ambuj Singh: "Topological Indexing and Mining of Chemical Compounds", in BCB 2011.

5.2 Previous Edition of this Tutorial

An overview of the state-of-the-art IM techniques was presented at ACM CoDS-COMAD 2018. The current proposal for EDBT 2019 would include the following extensions:

- We will build upon our benchmarking framework to present detailed insights about the state-of-the-art IM algorithms in real-world scenarios. We will unravel several *myths and ambiguities that plague the current maze of IM techniques.*
- We will discuss in detail about the *open-challenges* that remain in the field of influence maximization and provide concrete pointers to important research questions in order to facilitate streamlined advancement of the field.

REFERENCES

[1] Akhil Arora, Sainyam Galhotra, and Sayan Ranu. 2017. Debunking the Myths of Influence Maximization: An In-Depth Benchmarking Study. In *SIGMOD*.

[2] Cigdem Aslay, Laks Lakshmanan, Wei Lu, and Xiaokui Xiao. 2018. Influence Maximization in Online Social Networks. In *WSDM*.

[3] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Brendan Lucier. 2014. Maximizing Social Influence in Nearly Optimal Time. In *SODA*.

[4] Carlos Castillo, Wei Chen, and Laks V. S. Lakshmanan. 2012. Information and Influence Spread in Social Networks. In *KDD*.

[5] Shuo Chen, Ju Fan, Guoliang Li, Jianhua Feng, Kian-lee Tan, and Jinhui Tang. 2015. Online Topic-aware Influence Maximization. *PVLDB* 8 (2015).

[6] Wei Chen, Wei Lu, and Ning Zhang. 2012. Time-Critical Influence Maximization in Social Networks with Time-Delayed Diffusion Process. *CoRR* abs/1204.3074 (2012).

[7] Wei Chen, Yifei Yuan, and Li Zhang. 2010. Scalable influence maximization in social networks under the linear threshold model. In *ICDM*.

[8] Suqi Cheng, Huawei Shen, Junming Huang, Wei Chen, and Xueqi Cheng. 2014. IMRank: influence maximization via finding self-consistent ranking. In *SIGIR*.

[9] Suqi Cheng, Huawei Shen, Junming Huang, Guoqing Zhang, and Xueqi Cheng. 2013. Staticgreedy: solving the scalability-accuracy dilemma in influence maximization. In *CIKM*.

[10] Edith Cohen, Daniel Delling, Thomas Pajor, and Renato F. Werneck. 2014. Sketch-based Influence Maximization and Computation: Scaling up with Guarantees. In *CIKM*.

[11] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. 2015. The Case Against Specialized Graph Analytics Engines. In *CIDR*.

[12] Sainyam Galhotra, Akhil Arora, and Shourya Roy. 2016. Holistic Influence Maximization: Combining Scalability and Efficiency with Opinion-Aware Models. In *SIGMOD*.

[13] Sainyam Galhotra, Akhil Arora, Srinivas Virinchi, and Shourya Roy. 2015. ASIM: A Scalable Algorithm for Influence Maximization Under the Independent Cascade Model. In *WWW*.

[14] Amit Goyal, Francesco Bonchi, and Laks V. S. Lakshmanan. 2011. A Data-based Approach to Social Influence Maximization. *PVLDB* 5 (2011).

[15] Amit Goyal, Wei Lu, and Laks V.S. Lakshmanan. 2011. CELF++: Optimizing the Greedy Algorithm for Influence Maximization in Social Networks. In *WWW*.

[16] Amit Goyal, Wei Lu, and Laks VS Lakshmanan. 2011. Simpath: An efficient algorithm for influence maximization under the linear threshold model. In *ICDM*.

[17] Kyomin Jung, Wooram Heo, and Wei Chen. 2012. IRIE: Scalable and Robust Influence Maximization in Social Networks. In *ICDM*.

[18] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the Spread of Influence Through a Social Network. In *KDD*.

[19] V. Kumar, Vikram Bhaskaran, Rohan Mirchandani, and Milap Shah. 2013. Creating a Measurable Social Media Marketing Strategy: Increasing the Value and ROI of Intangibles and Tangibles for Hokey Pokey. *Marketing Science* 32, 2 (2013).

[20] Konstantin Kutzkov, Albert Bifet, Francesco Bonchi, and Aristides Gionis. 2013. STRIP: Stream Learning of Influence Probabilities. In *KDD*.

[21] Jure Leskovec, Andreas Krause, Carlos Guestrin, and Christos Faloutsos. 2007. Cost-effective Outbreak Detection in Networks. In *KDD*.

[22] Hui Li, Sourav S. Bhowmick, Jiangtao Cui, Yunjun Gao, and Jianfeng Ma. 2015. GetReal: Towards Realistic Selection of Influence Maximization Strategies in Competitive Networks. In *SIGMOD*.

[23] Hung T. Nguyen, Tri P. Nguyen, NhatHai Phan, and Thang N. Dinh. 2017. Importance Sketching of Influence Dynamics in Billion-Scale Networks. In *ICDM*.

[24] Hung T. Nguyen, My T. Thai, and Thang N. Dinh. 2016. Stop-and-Stare: Optimal Sampling Algorithms for Viral Marketing in Billion-scale Networks. In *SIGMOD*.

[25] Naoto Ohsaka, Takuya Akiba, Yuichi Yoshida, and Kenichi Kawarabayashi. 2014. Fast and Accurate Influence Maximization on Large Networks with Pruned Monte-Carlo Simulations. In *AAAI*.

[26] Naoto Ohsaka, Tomohiro Sonobe, Sumio Fujita, and Ken-ichi Kawarabayashi. 2017. Coarsening Massive Influence Networks for Scalable Diffusion Analysis. In *SIGMOD*.

[27] Diana Popova, Naoto Ohsaka, Ken-ichi Kawarabayashi, and Alex Thomo. 2018. NoSingles: a space-efficient algorithm for influence maximization. In *SSDBM*.

[28] Manuel Rodriguez and Le Song. 2015. Diffusion in Social and Information Networks: Research Problems, Probabilistic Models and Machine Learning Methods. In *KDD*.

[29] Jimeng Sun and Jie Tang. 2014. Models and algorithms for social influence analysis. In *WWW*.

[30] Youze Tang, Yanchen Shi, and Xiaokui Xiao. 2015. Influence Maximization in Near-Linear Time: A Martingale Approach. In *SIGMOD*.

[31] Youze Tang, Xiaokui Xiao, and Yanchen Shi. 2014. Influence Maximization: Near-optimal Time Complexity Meets Practical Efficiency. In *SIGMOD*.

Finding Meaningful Contrast Patterns for Quantitative Data

Rohan Khade
George Mason University
Fairfax, VA, USA
rkhade@gmu.edu

Jessica Lin
George Mason University
Fairfax, VA, USA
jessica@gmu.edu

Nital Patel
Intel Corporation
Chandler, AZ, USA
nital.s.patel@intel.com

ABSTRACT

Contrast set mining identifies patterns that can best distinguish between two groups of data. While many machine learning models share the same goal, contrast set mining focuses on data understanding and interpretability. Most existing work in contrast set mining focuses on categorical data. In this work, we propose an algorithm that discovers contrast patterns on *mixed* data (datasets that contain both categorical and continuous attributes). Our algorithm is able to discover multivariate interactions using a supervised adaptive binning strategy. The binning strategy identifies meaningful bin boundaries in continuous attributes based on their relationships with other attributes. This in turn allows us to form better and more meaningful contrast patterns than traditional techniques that use global, pre-binning approaches. We propose various pruning strategies to reduce the search space, and show the utility of our algorithm on simulated data, several datasets from the UCI repository, as well as real manufacturing data.

ACM Reference Format:
Rohan Khade, Jessica Lin, and Nital Patel. 2019. Finding Meaningful Contrast Patterns for Quantitative Data. In *Proceedings of 22nd International Conference on Extending Database Technology (EDBT), Lisbon, Portugal, March 26-29, 2019 (EDBT 2019),* 12 pages.
https://doi.org/

1 INTRODUCTION

The work presented in this paper was motivated by a desire to reliably detect factors resulting in failure at final test during the semiconductor packaging and test process and is the outcome of a multi-year research grant funded by Intel, Corporation to develop a solution that can be applied to their manufacturing facilities. As the industry moves to more complex packages that involve complex process flows, the amount of data collected during the processing increases, and the signals being detected (as related to cause of test failures) become more diluted. At the same time, the cost of missing these signals increases, and hence there is a growing need to develop machine learning algorithms that can quickly detect the potential cause of part failures and deliver timely feedback to the engineers so that adjustments in the manufacturing line can be made to avoid generating scrap. Note that packaging can contribute up to 50% of the cost to manufacture a CPU, hence any scrap avoidance is highly desirable. As an example, during the baking stage of the manufacturing line, if the ovens are run at a higher temperature than usual, resulting in low yield, a timely notice could minimize potential loss. The behavior of manufacturing data is often predictable; however, at times there exist anomalies such as low yield for a batch. To find potential causes of this low yield, one could create a classification model comparing good chips and bad chips. Apart from a few models such as decision trees, most models are not interpretable to the user and hence non-actionable. Even though models like decision trees can be used to find explainable patterns, usually there is a single global model built for the whole dataset using a greedy strategy. To find all patterns, the user would need to build all possible trees which can take exponential time. Therefore, while decision trees are good for classification, they are not suitable if the goal is to detect patterns in the dataset. The matter becomes more complicated when we want to capture multivariate relationships between attributes (e.g. XOR data), which require more computational time.

The analysis of semiconductor manufacturing data is nontrivial because the numbers of attributes and instances are large, and the engineer needs considerable amount of time for analysis. Our intent is to learn the patterns ("contrast sets") that distinguish two groups, e.g. a normal group and an anomalous group, automatically, without external knowledge. We note that the main goal here is data understanding and exploration, rather than prediction.

Pattern mining algorithms are often used during the initial stages of the data mining process to understand relationships among features, or in a decision making stage. A major concern is displaying results that misconstrue relationships between attributes or giving incorrect insights to make decisions. For example, due to the large number of relationships between attributes to be considered, there is a high probability of discovering uninteresting or potentially spurious patterns. A large amount of existing work in pattern mining has focused on reducing the number of such uninteresting patterns, which is also one of the goals in this paper. Another concern relating to pattern mining is the time and space complexity. This can usually be reduced by either building a more compact representation of the data or pruning the search space. In this paper, we try to reduce the search space by pruning uninteresting regions.

Contrast set mining is a set of algorithms under the pattern mining paradigm to find patterns for which the supports differ significantly among groups. It is closely related to, and can be directly compared [21] to subgroup discovery and emerging pattern mining. There has been a lot of work in the area of contrast set mining [4, 14, 15, 26, 29]; however, there remain some issues that need to be addressed. First, in the manufacturing of semiconductors, many attributes of interest are continuous. Most of the existing work in contrast set mining, emerging patterns and subgroup discovery focus on improving the efficiency of the algorithms to find categorical contrasts, i.e. by reducing the search space and the number of database scans. Continuous attributes are typically handled by either computing some statistics (such as mean) that meaningfully differ among groups, or by using a binning technique as a preprocessing step and then treating the attribute as a categorical one. The latter can potentially provide more information since it identifies local patterns as ranges of values in the continuous attributes that can be actionable. Therefore, in this work we focus on binning-based approaches. A software suite Cortana has many state of the art subgroup discovery algorithms developed. It also has an implementation of an adaptive discretization method that we compare to in the experimental section. This approach, however, is a greedy approach and may miss (local) multivariate interactions between continuous features which we often find in semiconductor manufacturing data. As will be seen later, the patterns found using these algorithms seem to be redundant and cumbersome to interpret.

Binning or discretization is a fundamental and well-studied topic in data mining. Garcia et al. [10] published a detailed survey on discretization techniques, as well as a tool (KEEL) that contains implementations of 30 popular discretization algorithms. We applied these discretization algorithms extensively on various datasets, but we were not able to find an algorithm that satisfies all our requirements. Specifically, for our application (or applications containing continuous attributes in general), the discretizer has to be able to handle **multivariate data**, be **adaptive** (local bins with respect to a subset of attributes), and **dynamic** (tightly coupled algorithm with the end goal of finding the most meaningful contrast patterns). In addition, the algorithm needs to detect not only global correlations, but (local) multivariate interactions between features. Unfortunately, existing algorithms, including those implemented in Cortana, typically miss one or more of our requirements.

Another important aspect of the proposed algorithm is to show the user the most meaningful contrasts. The authors in [13, 27] have defined patterns that will most likely be interesting to a user. More specifically, a meaningful contrast is a pattern that is not redundant, is productive and independently productive. We define what each term means in the context of contrast pattern mining and extend it to cases where the patterns have only continuous, or have mixed features.

Our contributions are summarized as follows:

(1) We propose an algorithm, SDAD-CS (Supervised Dynamic and Adaptive Discretization for Contrast Sets), to find contrast patterns for datasets containing continuous (and categorical) attributes.
(2) Our binning technique is supervised, dynamic, and adaptive, and therefore finds better quality and meaningful bins as compared to the state of the art.
(3) The binning technique detects multivariate interactions and hence higher order contrasts can be detected.
(4) We introduce several pruning strategies to reduce the search space, which also results in finding more meaningful contrast patterns.
(5) We use statistical measures to find non-redundant, productive and independently productive contrast patterns.

2 RELATED WORK

A number of work on contrast set mining have been proposed [4, 14, 15, 26, 29]. In their pioneering work [4], the authors proposed an algorithm, STUCCO (Searching and Testing for Understandable Consistent COntrasts), that finds contrast sets in groups. STUCCO employs efficient search for contrast sets based on another rule mining algorithm, Max-Miner [5]. To assess the meaningfulness of the difference in support values across groups, the authors use a chi-square test on the null hypothesis that the support value is independent of group membership. In another work [26], the authors observe that existing commercial rule-finding system, Magnum Opus [25], can successfully perform the contrast-set mining task. The authors conclude that contrast-set mining is a special case of the more general rule discovery task. The techniques discussed above are only applicable to categorical data. A good survey on contrast sets, emerging patterns and subgroup discovery algorithms is provided in [21]. The authors also discuss how the interest measures (such as difference in support and WRACC) are compatible, i.e. the interest can be used interchangeably between communities.

Techniques derived from decision tree learning can be used; however, the authors in [16, 18, 23] explain some limitations of decision-tree-based method for our application. A number of work have been proposed to find subgroups in numerical domains. The authors in [20, 23] discretize numerical data into bins to find subgroups. The algorithm is implemented in an open source tool Cortana. Such techniques typically use an initial discretization method and then merge spaces based on an interest measure. We have compared against this approach in the experimental section. An interesting algorithm described in [11] also discretizes continuous attributes into bins for the problem of subgroup discovery. The algorithm uses optimistic estimates and horizontal pruning to prune the search space. This technique heavily relies on pruning based on the top-k subgroups, since the interest measure can be updated as soon as the algorithm reaches k subgroups. Finding all initial split points (exhaustive search in [11]) is expensive but if the initial partitions are not exhaustive, e.g. frequency or entropy based, the algorithm may miss interesting patterns that occur lower down the tree due to multivariate interactions. The current trend of recent algorithms tend to use sampling and user feedback to improve efficiency and quality of rules [6–8, 17]. This is an interesting direction; however, our goal is to develop an accurate and efficient discretizer in order to find contrast patterns. These algorithms can certainly be used in conjunction with our algorithm.

Quantitative association rule mining is well-studied and could potentially be used for contrast mining. Srikant proposed a discretization technique that partitions the range of the continuous attribute into *n* equal-frequency partitions, and assigns the partitions to consecutive integers [22]. If the supports for any consecutive partitions fall below the *minsup* threshold, they are merged. The problem, however, is setting the initial number of partitions *n* and handling multivariate interactions. If *n* is too small, it results in large partitions and potential information loss since elements in the same partition are indistinguishable. On the other hand, if *n* is too large, the algorithm becomes computationally expensive. In [2], the authors find extraordinary behavior by partitioning the antecedent of the rule into bins and finding the statistics of the consequent. The algorithm cannot handle multivariate interactions between continuous attributes. The algorithm described in [24] is a bottom-up merging algorithm. It merges contiguous parts of a feature based on the improvement of an interest measure. We also use a bottom-up approach to merge spaces; however, as will be shown later, our algorithm can handle multivariate interactions between continuous features and does not need initial small bins. MVD [3] proposed by Bay is able to detect multivariate interactions for continuous attributes. This algorithm is also a bottom-up approach which merges contiguous spaces if they are not statistically different.

3 PRELIMINARIES

Let *DB* be a dataset with *m* rows $R = \{r_1, r_2, \dots, r_m\}$ and *n* attributes $A = \{a_1, a_2, \dots, a_n\}$. An attribute can be either categorical or continuous. A categorical attribute can have multiple values, i.e.

for a categorical attribute a_i having l unique values, domain(a_i) = {v_{i1}, ...,v_{il}}. For continuous attributes, its values consist of real numbers i.e. domain(a_i) = $\mathbb{R}$. An item in DB is either a value in a categorical attribute, $a_i = v_{ix}$ where $v_{ix} \in$ domain(a_i), or range in a continuous attribute, $a_i \in [v_l, v_r]$ where $v_l \leq v_r$, $v_l \in \mathbb{R}$ and $v_l \in \mathbb{R}$. Using the above definitions, we note that items in a continuous attribute can have overlapping ranges. An *itemset c* is a combination of items in *DB*. Apart from having *n* attributes, *DB* has an extra attribute that contains the group information for each row (instance). Let $G = \{g_1, g_2, \dots, g_k\}$ be a set of groups. Each row belongs to exactly one group and multiple rows can be a part of a single group. If $|g_k|$ is the number of instances in group k and $count_k(c)$ is the number of rows that contain itemset c in group k then **support** $supp_k(c)$ is:

$$supp_k(c) = \frac{count_k(c)}{|g_k|} \tag{1}$$

Bay [4] formally defines contrast set mining as follows. An itemset is a contrast between 2 groups i and j if the support difference between the 2 groups is **large** and **significant**. The support difference is large if

$$supp_i(c) - supp_j(c) > \delta \tag{2}$$

and significant if

$$\chi^2_{ij}(c) < \alpha \tag{3}$$

where α and δ are user-defined parameters.

Contrasts should be non-redundant. In the context of frequent itemset mining, an itemset c is redundant if it contains a proper subset d that has the same support as i where i is an itemset of d [27]. An example given in [27] explains that any superset of itemset female, pregnant is unlikely to be interesting since female subsumes pregnant, i.e. these itemsets are functionally dependent.

An itemset c is said to be productive if for every partition d, where $d \subset c$, $supp(c) > supp(d) * supp(c \backslash d)$. Since the dataset is usually a sample of the population, statistical test such as fisher's exact test and chi-squared tests are performed on each partition of an itemset to check for significance of the product. Although this makes the algorithms computationally more expensive, it is an important step to determine productivity and finding meaningful patterns.

Another requirement is that a pattern should be independently productive. To be independently productive, an itemset should not be explained by any of its supersets apart from being productive and not redundant. Statistical tests are also performed at this step, usually as a postprocessing step.

An optimistic estimate (*oe*) of an itemset is the maximum possible value of an interest measure in any of the itemsets specializations[12]. If X' is the specialization of X and Int(X') is the calculated interest measure of X' then

$$Int(X') \leq oe(X) \tag{4}$$

optimistic estimates are used to prune the search space by calculating the upper bound of the children nodes of each explored node in the search tree.

Top-k pattern mining algorithms display the best 'k' patterns to the user based on some user defined interest measure. The advantage is two-fold. First, it removes the need for the user to enter a minimum threshold for the interest measure. For example, support-based pruning is the first stage of the Apriori algorithm [1] for association rule mining. Determining the best minimum support (*minsup*) is non-trivial, if *minsup* is too high or too low, it may find too few or too many patterns, respectively. The other advantage is that it helps prune the search space even if the interest measure is not monotonically decreasing based on optimistic estimates.

We use the following strategies to reduce the search space. An itemset is pruned if (1) it does not have support over δ in any group (*minimum deviation size* pruning); (2) its expected occurrence is less than 5, since statistical tests are not significant at that level; and (3) If the optimistic estimate of the χ squared value for the itemset's children is less than the current threshold. Also, to reduce the number of false positives, the value of α is adjusted according to Bonferroni's adjustment as explained in [4].

4 QUANTITATIVE CONTRAST SETS

4.1 Methodology

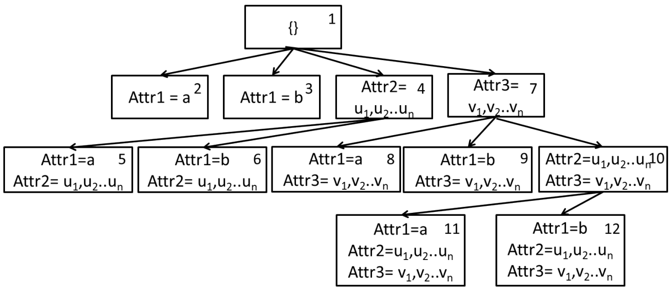

{} 1
Attr1 = a 2
Attr1 = b 3
Attr2= u1,u2..un 4
Attr3= v1,v2..vn 7
Attr1=a Attr2= u1,u2..un 5
Attr1=b Attr2= u1,u2..un 6
Attr1=a Attr3= v1,v2..vn 8
Attr1=b Attr3= v1,v2..vn 9
Attr2=u1,u2..un Attr3= v1,v2..vn 10
Attr1=a Attr2=u1,u2..un Attr3= v1,v2..vn 11
Attr1=b Attr2=u1,u2..un Attr3= v1,v2..vn 12

Figure 1: Search Tree for Mixed Data

To find combination of attributes (itemsets to be explored), any search algorithm such as breadth first search or depth first search can be used. Depth first search is not the preferred choice of search algorithm since it reduces the amount of pruning possible. More specifically, it may try to combine attributes which other algorithms may have found to be "non-combinable" early on. For example. if the support of a subset of an itemset is below the threshold, depth first search will not prune it. Breadth first search, on the other hand, can maximize pruning. However, the storage overhead at each level may be high. We use a search strategy [28] shown in Figure 1 since it can maximize pruning, and it requires less storage overhead than breadth first search. The figure shows how itemsets are combined, and the number on each node indicates the order in which it is explored. If an itemset contains only categorical attributes, calculating the support for each group is straightforward. When a combination containing at least one continuous attribute is encountered in the tree, our proposed algorithm, SDAD-CS, is called.

To explore contrast patterns for mixed (or exclusively continuous) itemsets, such as in nodes 4 to 12 in Figure 1, we propose SDAD-CS (Supervised Dynamic and Adaptive Discretization for Contrast Sets), a quantitative contrast-set mining algorithm. Given an itemset c containing 0 or more categorical items, and 1 or more continuous attributes $ca = \{a_1 \dots a_n\}$ where n > 0, SDAD-CS finds itemsets that are contrasts between the groups. The contrasts found should also have the interest measure (such as difference in support) greater than the current minimum. Each contrast pattern returned should contain items from all the attributes (categorical and continuous) specified by the calling function.

The core idea behind SDAD-CS is to first divide the space (range) of a continuous attribute in a top-down fashion, calculate the interest measures and determine whether to stop searching or to explore further. After that, it merges similar contiguous spaces in a bottom-up fashion and refines the space.

Let input group DB be all the rows and columns of the dataset containing the groups of interest. Let c and ca be the categorical itemset and continuous attributes to be explored, respectively. The pseudocode for SDAD-CS is shown in Algorithm 1. The α and δ are user input parameters (α is adjusted during execution). Though not shown specifically in the algorithm, α and δ are used each time there is a check for significance and largeness, respectively. The parent measure, which is initially set to 0, stores the parent's interest measure (such as difference in support). Min support is set to the current minimum support in the current list of top-k contrasts. If the list does not have k contrasts, min support is set to δ. Starting with the top-down part, $partition(ca)$ (Line 4 of the Algorithm) divides each continuous attribute at the median or mean (we use median) into spaces. For example, a continuous attribute with a range 0 to 100 with median 35 will be divided into [0-35] and (35-100]. Next, $find\_combs(p)$ (Line 5) finds combinations of spaces between continuous attributes. For example, if there are two continuous attributes, dividing each by its median creates four rectangles (spaces) on a scatter plot. Each space together with the categorical itemset creates a candidate itemset. If *cont* is the number of continuous attributes, then the number of spaces is 2^{cont}. These spaces define our initial bin boundaries.

The algorithm iterates over each space created. It checks if the space can be pruned (Line 7). This is performed by either checking a lookup table or by performing some calculation and saving the information in a lookup table, as will be described later. We use a hash map with the itemset as the key. More space-efficient data structures such as a hierarchical hash map can be used if space is an issue. Our pruning strategies are explained in detail in a later section; however, at this point it suffices to note that a space is pruned if it is found in the lookup table.

The next step is to calculate the support of the itemset in each group in the current space r (Line 10). SDAD-CS then calculates the interest measure – in our case the difference in support (and Surprising Factor) (Line 11). The algorithm then needs to make a decision whether to explore the current space further. This is determined by calculating the optimistic estimates for the child space. If the current database contains n groups, the optimistic estimate is calculated as follows.

Let r be the current space being explored, cca_r be the itemset found at space r and $count_k(cca_r)$ be the number of instances of group k in space r. If $|g_1|$, $|g_2|$ ... $|g_n|$ are the number of instances in group *1*, *2* ... *n* respectively, then,

$$supp_1(cca_r) = \frac{count_1(cca_r)}{|g_1|} \tag{5}$$

is the support of itemset cca in space r in group 1. Similar definition follows for the support of the same itemset in group n.

Let *level* be the current level in the recursive tree of SDAD-CS, $|ca|$ be the number of continuous attributes, then

$$max\_instances\_child = \frac{|DB|}{2^{level+1} * |ca|} \tag{6}$$

indicates the maximum number of instances in the child spaces created by a recursive call of SDAD-CS. This comes from the fact that the continuous space is split at the median and hence distributes the data points among all child spaces equally. It should be noted that the assumption is that the data is real-valued, and each reading is unique. Some care has to be taken if the number of unique values is far less than the number of data points.

The maximum support for itemset cca_r in group 1 in any of the child spaces is

$$max\_supp\_g1 = min(\frac{max\_instances\_child}{|g_1|}, supp_1(cca_r)) \tag{7}$$

The first part of equation 7 calculates the maximum support possible in a child space for group 1. We note here that the median is calculated based on all the given instances and the groups can be imbalanced. We see that dividing the space may not reduce the supports proportionally in all groups. If the number of possible instances in the child space is greater than the number of instances in group 1, then the first value inside the 'max' function is greater than 1. This is not possible and is taken care of in the second part of the equation. Also, support is monotonically decreasing as the space reduces, and hence if the support of the current space is less than the maximum **possible** support of the child space, the maximum support of the child space is the current support. A similar argument can be made for the other groups.

We can calculate minimum support by following Eq. 6-8:

$$other\_instances\_g1 = |DB| - count_1(cca_r) \tag{8}$$

$other\_instances\_g1$ is the number of instances of the other groups apart from g1 in the current space r.

Let

$$min\_instances\_g1 = max\_instances\_child - other\_instances\_g1 \tag{9}$$

which will be negative if the majority of elements are not g1.

$$min\_supp\_g1 = max(0, \frac{min\_instances\_g1}{|g1|}) \tag{10}$$

Finally, the optimistic estimate for the child space is given by

$$oe(cca_r) = max(\forall i \forall j, i \neq j, max\_supp\_gi - min\_supp\_gj) \tag{11}$$

i,j = 1..n

If the optimistic estimate calculated is greater than the minimum support, the child spaces are recursively explored (Lines 12-13). If a better contrast pattern is found in the child space, it is added to the current list of contrast patterns (Lines 14-15), else if the current contrast pattern is large and significant, then it is either added to the current list of contrasts D or D_{temp} (Lines 16-21). The current itemset is added to D if the interest measure is greater than its parents. However, if it is not, the algorithm waits until all the spaces are explored and adds it if at least the interest measure in one space is greater than that of its parent (Lines 22-23).

After finding contrast spaces, the algorithm merges similar and contiguous spaces to get more general and comprehensible contrasts in a bottom up fashion (Lines 26-30). To merge partitions, the spaces are sorted in increasing order of size. We observe that, SDAD-CS finds fewer and more meaningful itemsets since there is more opportunity of merging smaller itemsets. If we plot the continuous attributes on a scatter plot, the spaces created by two continuous attributes is a rectangle and the size is the area of the rectangle; by plotting 3 continuous attributes the space

Algorithm 1: Algorithm SDAD-CS

```
   Input: DB with group attribute, categorical items c in
          itemset, continuous attributes ca, $\delta$, $\alpha$, min support,
          parent measure pm
   Output: Set of contrast patterns
1  begin
2      $D \leftarrow List\ of\ itemsets\ that\ are\ contrasts$ (Initially set to
        empty )
3      $D_{temp} \leftarrow List\ of\ itemsets\ that\ may\ be\ contrasts$
        (Initially set to empty )
4      $p = partition(ca)$ $\%partition\ each\ continuous$
        $attribute\ at\ median$
5      $r = find\_combs(p)$ $\%\ find\ all\ combinations$
        $of\ ranges\ found\ by\ p$
6      for each space in r do
7          if can_prune($cca_r$) then
8              Add itemset to pruned list
9              continue
10         Calculate s($cca_r$) for each group
11         Calculate int($cca_r$) user defined interest measure
            (such as difference in support) between each group
12         if oe($cca_r$) > min support then
13             D$_{child}$ = SDAD_CS(DB$_r$, ca, $\delta$, $\alpha$, min support,
                int($cca_r$))
14         if $D_{child}$ not empty and then
15             Append(D,D$_{child}$)
16         else
17             if $cca_r$ large and significant then
18                 if $cca_r$ greater than pm then
19                     Append (D,$cca_r$)
20                 else
21                     Append (D$_{temp}$,$cca_r$)
22     if len(D)>0 then
23         append(D,D$_{temp}$)
24     else
25         return []
26     if level==1 then
27         FIS = SORT(All spaces from smallest to largest)
            while No space left to combine in FIS do
28             Check 2 contiguous spaces if combination is
                possible if Comb possible then
29                 Combine itemsets; update contrast set
30     Return D
```

is a cuboid and the size is the volume of the cuboid. In general, hyper-planes create hyper-cubes and the size is n-volume.

Lines 28-29 loop through the spaces and try to merge contiguous and similar ones. Again, similarity is tested using a chi square test with α_r and the resulting contrast is still large and significant. If the itemsets are merged, the support, *PR* (to be defined later), hyper volume and bin boundaries are updated accordingly. More specialized itemsets are deleted and the new itemset is inserted in the appropriate sorted place.

4.2 Interest Measures

The default interest measure we use in the quantitative analysis is the difference in support; however, we find that looking at the homogeneity of a space while searching can help us find interesting patterns. We define an interest measure, *purity ratio (PR)*, which describes how homogeneous the current region is with respect to the group. In general, any interest measure, such as entropy, can also be used here depending on the problem definition. Our data is highly imbalanced and working with just supports of the group eradicates this issue. For *purity ratio (PR)*, a value closer to 1 indicates that the current space contains mostly data from the same group. Suppose i and j are the groups we are contrasting, c is the itemset with discretized quantitative attributes, s_{ic} is the support of group i in the space of itemset c, we define *PR* as:

$$PR(c) = 1 - \frac{min(s_{ic}, s_{jc})}{max(s_{ic}, s_{jc})} \qquad (12)$$

One limitation with purity ratio is that it does not take the size of the itemset involved into consideration. For example, consider two itemsets: c_1 with supports of 0.02 and 0.04 in groups i and j respectively and c_2 with supports 0.30 and 0.60. Both have equal purity ratio. However, we notice that c_2 should be considered more interesting since it covers more instances. On the other hand, difference in support has another issue. Suppose we find two itemsets: c_1 with supports of 0.9 and 0.8 in groups i and j respectively and c_2 with supports 0.20 and 0.10, and we notice that both have similar support difference. However, c_2 (for our application) is more interesting, i.e. given the contrast c_2 the likelihood c_2 to be in group i is double that of j but there is almost a equal likelihood for c_1 to be from either of the groups. To overcome this, we define **SurPRising Measure**:

$$SurprisingMeasure(c) = PR(c) * Diff(c) \qquad (13)$$

By multiplying difference in support (Diff) in each group to purity (PR) it takes the size of the contrast into consideration while giving equal weights to both groups.

The optimistic estimate for Surprising Measure is the same as Equation 11, since in the best case, PR will always be 1 in any partition (PR = 1 if there is only one instance in a partition).

4.3 Pruning

For itemsets containing only categorical attributes, we use the same pruning methods as in [4], i.e. minimum deviation size, expected value and chi-square bounds. This can be directly applied to itemsets containing both continuous and categorical or only continuous items once the bins are formed. Apart from the above technique, we try to prune redundant contrasts. An itemset is redundant if the support of the itemset is equal to the support of one of its subsets. The rationale behind this can be explained using an example. Consider an itemset $\{female\ \&\ pregnant\}$. Female subsumes pregnant i.e. the support of $\{female\ \&\ pregnant\}$ is equal to support of $\{pregnant\}$. Any contrast that is a superset of $\{female\ \&\ pregnant\}$ is likely redundant.

If an itemset is redundant, the support difference will be the same as its ancestors. Not expanding this itemset will reduce the number of redundant contrasts and search space. We note that the itemset should be redundant in all groups. In many real world datasets, there might be missing values, or incorrectly entered values. In addition, highly correlated features also tend to have many redundant contrasts. Hence, we loosen the requirement of

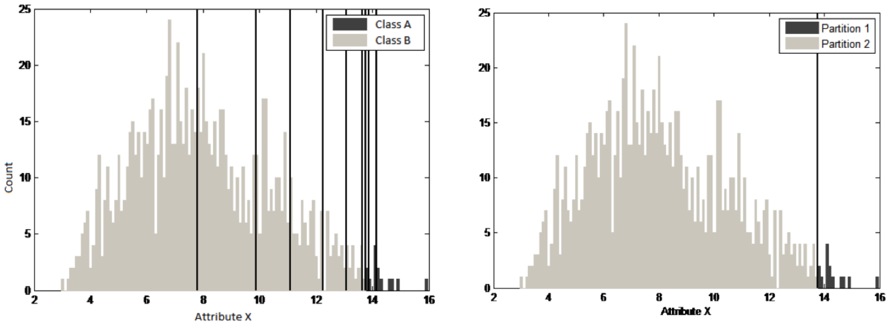


Figure 2: (Left) Vertical lines: all splits before merging. (Right) Final result after merging.

total subsumation and test whether the difference is statistically the same.

The datasets tested upon are samples of the population, and hence to make decisions for the population, statistical tests are needed. To check if two itemsets have statistically the same differences in support in the population, we use the central limit theorem. We choose this because we can assume that the difference in support for multiple samples of the population tend to follow a normal distribution. Extending the definition of central limit theorem to difference in support, it states that “Given no other samples, the best approximation of the mean of the difference in support for the population is the difference in support in the current sample."

Let α be the significance level, $|gx|$ and $|gy|$ be the sizes of groups x and y, respectively, $diff_{curr}$ be the current difference in the groups, $diff_{subset}$ be the difference of the subset, and $supp_x(c)$ and $supp_y(c)$ be the supports of itemset c in group x and y, respectively. For each subset, we calculate the bounds of the difference $diff_{bound}$. Let

$$a = \frac{supp_x(c) * (1 - supp_x(c))}{|gx|} \tag{14}$$

and,

$$b = \frac{supp_y(c) * (1 - supp_y(c))}{|gy|} \tag{15}$$

$$diff_{bound} = diff_{subset} \pm \alpha * \sqrt{a + b} \tag{16}$$

If $diff_{curr}$ is within the range of $diff_{bound}$, the difference support for the current itemset is statistically the same as its subset and hence may not be interesting and is pruned. Itemsets that are supersets of the current itemset will also not be meaningful.

Another case for redundancy for contrast patterns would be if there is a contrast found with support = 0 in a group but greater than δ in the other, then adding another item to the itemset may result in a redundant itemset. Extending this to itemsets containing continuous items, and looking at our definition of PR, we notice that when $PR = 1$ in a space, only one group is present in that space. Adding another item to the itemset would result in redundant contrasts. For example, consider a dataset containing attributes height and current country with groups toddler and adult. Consider we find a contrast height ∈]60,75] (inches) has support(adult)=0.8 and support(toddler)=0. Now adding current country to height may also result in a large and significant contrast, but it is clearly redundant between these groups.

A contrast pattern ’c’ is productive if for every subset ’a’ and ’c\a’,

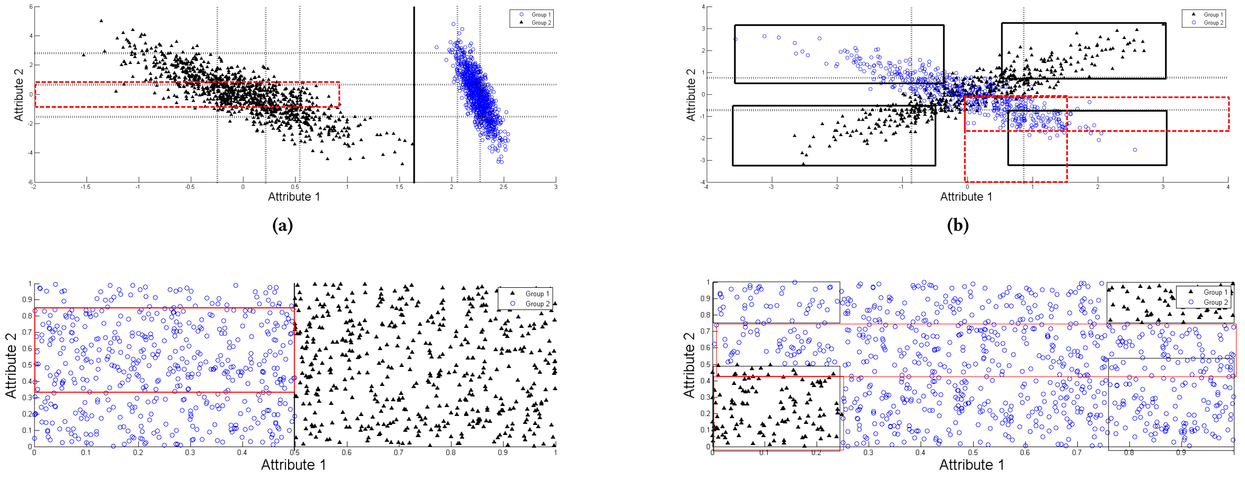
$$diff_c > supp_x(a) * supp_x(c \setminus a) - supp_y(a) * supp_y(c \setminus a) \tag{17}$$

if |g_x| > |g_y|.

If $diff_c$ is less than the product on the right-hand side of the equation for even one of the subsets, the contrast is clearly not productive. However, if it is greater, a statistical test is needed to confirm if it is indeed productive. We use chi-square test to check productivity. It should be noted that this formula is related to leverage in association rule mining which checks statistical dependence between variables.

At the end of the mining process, a check is performed to see if the contrasts are independently productive. Independently productive itemsets are meaningful, independent of their children or ancestor itemsets. For example, consider a dataset with two groups of days when a hurricane "develops" and "not develops," and a user wants to study the differences in the groups. There are a few necessary conditions for a hurricane to develop, e.g. temperature of water > 80 degrees Fahrenheit, depth of water > 200 feet and low wind shear. Considering these 3 features, the number of contrasts which will be found is 7. However, the only contrasts that the user might be interested in are the ones with all 3 conditions in it. Independently productive patterns provides the users with a compact set of patterns which are likely to be meaningful.

To check whether an itemset is independently productive, a check is performed on each superset of the itemset present in the final list. For example, consider itemset $\{A \,\&\, B\}$ and itemset $\{A\}$ in the list of contrasts found. Let r(A) be the indices of rows that item A is present, r(B) be the indices of rows that item B is present and r(A ∩ B) be the indices of rows that item A and item B are present. Now if itemset $\{A\}$ is independently productive then rows r(A) - r(A ∩ B) should also be a contrast, otherwise the contrast is found only because of itemset $\{B\}$. To check whether an itemset is a contrast, a chi-square test is performed to check significance difference in the groups. We note that an itemset may have multiple supersets and the check is performed only on supersets present in the final list. It is easy to see why this is the case by simple observation. If the superset is not a contrast, then it cannot be the case that the other features present in the superset caused the current itemset to be a contrast.



(a) (b) (c) (d)

Figure 3: (a) Simulated Dataset 1 (b) Simulated Dataset 2 (c) Simulated Dataset 3 (d) Simulated Dataset 4

4.4 Example

In Figure 2 we show an example of discretizing an itemset $c = \{X\}$, where X is a continuous attribute. Let G be the group attribute with value "A" and "B". The figures shows the histograms of X, and the different shades of gray denote the two groups "A" and "B". The darker shade denotes G = "A". Suppose 2% of all records belong in group "A", and the rest belong in group "B". X is first divided into two spaces at the median m, and SDAD-CS notices that PR in the left space is 1 since there are no instances of $\{X < m, Y = \text{"}A\text{"}\}$. This is a pure space and does not need to be split further. In the right space, however, the PR is $1 - (48/98)/(2/2) = 0.51$ and the optimistic estimate is $1 - 23/98 = 0.76$. The algorithm continues dividing the space and the new PR becomes $1 - (23/98)/(2/2) = 0.76$. All the partitions are shown in Figure 2(Left). Spaces are then merged from smallest to largest. The final partition after merging contiguous regions is in Figure 2(Right).

5 EXPERIMENTAL EVALUATION

SDAD-CS is compared with 3 other popular algorithms, MVD, Fayyad's entropy based method [9], and Subgroup Discovery interval binning [20] implemented in the Cortana software suite.

Experimental Setup: For all experiments, initial $\alpha = 0.05$ and $\delta = 0.1$. The search tree was stunted to have a maximum of 5 levels. For the simulated dataset, we use Surprising Factor as our interest measure since it results in the best contrasts qualitatively. For the quantitative analysis, we compare all the algorithms with SDAD-CS NP (No Pruning) . This was to level the playing field since many redundant and non-productive contrasts have a high interest measure and are pruned out by our algorithm. We use mean difference in support as the interest measure since the other algorithms are not developed to optimize Surprising Factor or Purity and hence would not be a fair comparison. These experiments however show the utility of our algorithm compared to the state of the art. We also discuss the scale and effect of non meaningful patterns found by not using our pruning methods.

We compare our algorithm to MVD [3] with initial $\alpha = 0.05$ and $\delta = 0.01$ of the size of the dataset. For MVD, the datasets were initially discretized to have 100 instances per small bin as in [3]. For Fayyad's discretizer, the Group attribute is treated as the Class. For subgroup discovery using Cortana, we use the WRACC measure (equivalent to finding support difference in groups [21]) with a minimum value of 0.01 with beam search and use the 'intervals' option for continuous attributes. The other settings for Cortana include keeping the target as nominal, search width 100, maximum time to infinity, maximum subgroups to k (100 in experiments), minimum coverage to 2 and maximum coverage to the entire dataset. Although Cortana is suite of algorithms, these settings seem to be the fairest comparison to our algorithm, and from here on we will refer to these settings as 'Cortana'. For Cortana we ran the algorithms twice, once for each subgroup, and then used all the subgroups found as the contrast set. The first part of this section qualitatively analyzes some of the contrasts found, and later we quantitatively compare the algorithms. Please visit https://zenodo.org/badge/latestdoi/8891484 to access the application version.

5.1 Simulated Dataset 1

As a litmus test, we first conduct experiments on 4 simulated datasets, to check the validity of the bins found. The first simulated data consist of 2 attributes as shown in Figure 3a. The bold line indicate the bins found by SDAD-CS and the dotted lines are the bins found by MVD.

The only split point SDAD-CS finds is with Attribute 1. Since $PR = 1$ for both contrasts we cannot do "better" by adding another attribute hence we prune these spaces i.e SDAD-CS will not find a contrast between Attribute 1 and Attribute 2. Although we see that there is some interaction (correlation) between the 2 attributes, which is detected by MVD, the goal here is to find a boundary that separates the groups and there is no interest in the underlying relationships in this case. MVD misses this splitting point. The entropy based method and Cortana finds the same contrast as SDAD-CS, however Cortana also finds a bin outlined by the red box which seems meaningless.

5.2 Simulated Dataset 2

This experiment shows the algorithm's ability to find meaningful contrast patterns in multivariate data. This dataset consists of two multivariate Gaussians in the shape of an "X" as shown in Figure 3b. Each Gaussian has a group attribute associated with it indicated by the markers.

The bin boundaries found by SDAD-CS are indicated by the rectangles in Figure 3b. We note that there is no rule found when we run SDAD-CS on each attribute individually. The contrasts found by MVD are similar to our algorithm. However, the entropy based method does not find any bins for this dataset. Cortana does not find the best bins which are shown by the red dotted boxes.

5.3 Simulated Dataset 3

In this experiment, we generated two variables uniformly distributed in the range 0 to 1. The only relationship in this dataset is that Attribute 1 in range 0 to 0.5 belongs to Group 2 and the rest Group 1. SDAD-CS finds contrasts at only level 1; however, Cortana finds meaningless contrasts at higher levels, shown in the red box. MVD finds similar contrasts as SDAD-CS.

5.4 Simulated Dataset 4

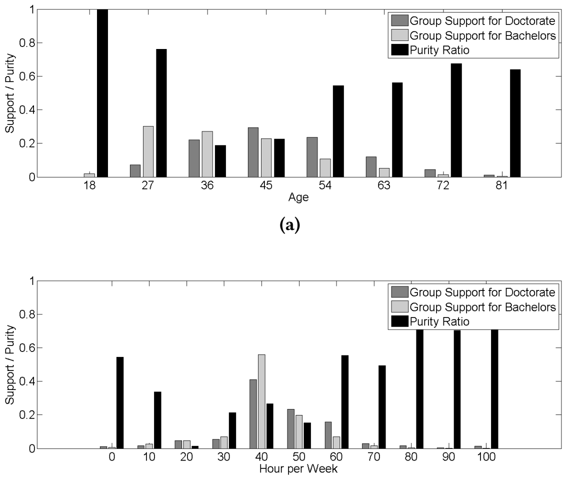
In Simulated dataset 4 we see interactions between attributes at level 2 of the search tree. We notice that during the search stage, there will be contrasts in the range 0 to 0.25 and 0.75 to 1 for Attribute 1 and 0 to 0.5 and 0.75 to 1 for Attribute 2. However, when the attributes are combined, the lower level contrasts are not independently productive and hence pruned by SDAD-CS. SDAD-CS finds a total of 6 contrasts. On the other hand, Cortana misses the top right contrasts and finds some meaningless regions shown in the middle of the figure.

5.5 Adult Dataset

5.5.1 Analyzing Bin Boundaries for Numeric Features. In this section, the differences in the contrasts found by the 5 algorithms on the Adult census dataset from the UCI Machine Learning repository [19] are shown. This experiment is a comparison between the 'Doctorate' and 'Bachelor' groups. We focus on Age and hours per week worked attributes since they highlight the differences between the algorithms.

Some of the quantitative contrasts found are shown in Table 1. Figure 4 shows the group support and the *PR* in each equi-frequency bin for Age and hours-per-week. The labels on the X-axis denote the bin boundary used and the Y-axis denotes the group support.

Looking at the contrasts found by SDAD-CS, we observe strong contrasts in the ranges 19–26 and 47–90 of the age attribute when we use PR as the interest measure to optimize. Looking at Figure 4a, we notice that, in the range 27–45, the supports for both groups are similar and hence it has a low *PR*. However, in the other ranges, there is clearly a dominant group. Similarly, in the range 50–100 for the hours-per-week attribute shown in Figure 4b, we see the majority belonging to the Doctorate group. The Bachelor group usually work less than 40 hours per week. The fifth contrast pattern discretizes {age, hours-per-week} which produces a better contrast (higher purity) than the contrasts found in the lower-order ones. This suggests that there is a multivariate relationship between these 2 attributes. We also notice the bin boundaries of {age, hours-per-week} change as compared to when they are discretized independently. This contrast shows that a global discretization may not work.



(a)

(b)

Figure 4: (a) Histogram comparing Age supports and purity ratio (b) Histogram comparing Hours per week supports and purity ratio

Cortana and SDAD-CS with support difference does not detect very good split points qualitatively. For example, in the range 19–26 in the age attribute we find only the group Bachelors. However, they find bigger bin boundaries. By looking at Figure 4a we notice the difference in supports and purity between ages 27 and 45 is small, however since the overall support in that space is much higher, the 2 algorithms find this as a large contrast. This is not surprising since the goal is to maximize the interest measure. A similar argument can be made for hours per week attribute. The contrasts found by Cortana when age and hours per week are combined are not *productive* according to the definition earlier. If we compare contrast 6 of Cortana and SDAD-CS with PR, Cortana finds a purer space. However, we also notice that in the range 19–26 of the age attribute, the support for the Doctorate group is close to 0, so *PR* is almost 1 for this contrast. Thus, we can prune this space for higher order combinations. An example of a contrast found without pruning this space is 19 ≤ Age ≤ 25 and 1 ≤ hours-per-week ≤ 40, which has support of 0 for the Doctorate group, and support of 0.10 for the Bachelors group. If this space is not pruned, SDAD would have found a purer contrast than Cortana, however, this clearly is a redundant space.

Fayyad's entropy discretizer and MVD detects level 1 interactions and finds strong contrasts, but fails to find any interaction between the attributes when combined. For the Doctorate group MVD forms a bin for Ages 48–59, which seems reasonable, however from 60–90 even though support difference is low, the purity (homogeneity) in favor of the Doctorate group is similar. Looking at the bar graph in Figure 4a, at around age 40, the support for both groups are similar, and as the age increases, we notice a higher support for the Doctorate group. MVD is not able to find the interaction between age and Hours worked

5.5.2 Analyzing top Patterns found. We now take a look at the top patterns found by Cortana (similar ones found by SDAD-CS without pruning). The setting for Cortana are as explained earlier; however, the depth of the tree was set as 2 for this discussion. The top 5 contrasts are shown in Table 3 (Cortana also displays contrasts such as $sex = Male\ and\ occupation = Prof\ specialty$

Table 1: Contrast Sets for Adult Dataset

S.No	Contrast Set	Supp (Doc.)	Supp (Bach.)
	Contrasts Using SDAD-CS with PR		
1	$18 < Age <= 26$	0	0.16
2	$47 < Age <= 90$	0.48	0.22
3	$1 < hour\_per\_week <= 40$	0.45	0.60
4	$50 < hour\_per\_week <= 99$	0.28	0.14
5	$49 < Age <= 69$ *and* $50 < hour\_per\_week <= 99$	0.13	0.03
6	$25 < Age <= 39$ *and* $1 < hour\_per\_week <= 39$	0.11	0.26
	Contrasts Using SDAD-CS with Support Difference		
1	$18 < Age <= 39$	0.26	0.57
2	$38 < Age <= 90$	0.76	0.46
3	$1 < hour\_per\_week <= 48$	0.55	0.73
4	$40 < hour\_per\_week <= 99$	0.55	0.40
	Contrasts Using Subgroup Discovery with Cortana		
1	$39 < Age <= 80.0$	0.74	0.43
2	$-inf < Age <= 39$	0.26	0.57
3	$6 < hour\_per\_week <= 49$	0.53	0.72
4	$49 < hour\_per\_week < inf$	0.45	0.28
5	$32 < Age <= 69$ *and* $49 < hour\_per\_week < inf$	0.41	0.19
6	$-inf < Age <= 43$ *and* $6 < hour\_per\_week <= 49$	0.20	0.50
	Contrasts Using Fayyad Entropy Binning		
1	$18 < Age <= 26$	0	0.16
2	$26 < Age <= 32$	0.08	0.19
3	$46 < Age <= 90$	0.24	0.51
4	$0 < hour\_per\_week <= 55$	0.91	0.78
	Contrasts Using MVD		
1	$18 < Age <= 24$	0	0.13
2	$47 < Age <= 58$	0.32	0.15
3	$39 < hour\_per\_week <= 40$	0.30	0.43
4	$50 < hour\_per\_week <= 99$	0.28	0.14

Table 2: Datasets

Dataset	Groups	No. of instance per group	No. of Features/ Continuous Features
Adult	Bachelors/Doctorate	8025/594	13/5
Spambase	Spam/No Spam	1813/2788	57/57
Breast Cancer	Benign/Malignant	458/241	10/10
Mammography	Severe/Not Severe	445/516	5/5
Transfusion	Donated/Not Donated	570/178	4/4
Shuttle	Rad Flow/High	45586/8903	9/9
Credit Card	No/Yes	23363/6635	24/23
Census Income	Below 50K/Above 50K	187141/12382	39/11
Ionosphere	g/b	225/126	34/34
covtype	Spruce-Fir/Lodgepole Pine	211840/283301	54/10

which is clearly the same as contrast number 4 in the table and is not considered here). We notice that the top 5 contrasts has one item in common $occupation = Prof\ specialty$. The question arises if all the contrast are meaningful.

Itemsets i, ii and iii in Table 3 are singular itemsets required for calculation of the expected support for the top 5 itemsets shown as a, b and c in table. Looking at Table 3, we see itemsets 1, 4 and 5 are not meaningful since the difference in support is not statistically different from the expected difference in support. Itemset 2 is clearly redundant and functionally dependent to itemset 3. Hence, of the top 5 contrasts found by Cortana, only contrast 3 would be displayed by SDAD-CS. It should be noted that these itemsets are seeds to higher order itemsets (with 3 and more items) which further exacerbates the problem. Later on we discuss the pervasiveness of this in all the datasets we encounter.

5.6 Quantitative Analysis

In this section, we compare the mean difference in support. We compare the algorithms based on difference of support since it is shown to be compatible with WRACC [21] (they are directly proportional). It should be noted that SDAD-CS finds significantly better contrasts with respect to Surprising Factor, however, it would not be a fair comparison since Cortana is not optimized for this interest measure.

Table 3: Top Contrast Sets for Adult Dataset with Cortana

S.No	Contrast Set	Supp (Doc.)	Supp (Bach.)
	Top 5 Contrasts found by Cortana		
1	$occupation = Prof\ specialty\ and\ 28 <= Age < 80$	0.74	0.21
2	$occupation = Prof\ specialty\ and\ 19302 <= fnlwgt < 606111$	0.76	0.28
3	$occupation = Profspecialty$	0.76	0.28
4	$occupation = Prof\ specialty\ and\ sex = Male$	0.61	0.17
5	$occupation = Prof\ specialty\ and\ class = > 50K$	0.55	0.11
	Required Itemsets with 1 item		
i	$28 <= Age < 80$	0.98	0.8
ii	$sex = Male$	0.81	0.69
iii	$class = > 50K$	0.73	0.41
	Expected Supports for itemsets		
a	$occupation = Prof\ specialty\ and\ 28 <= Age < 80$	0.75	0.22
b	$occupation = Prof\ specialty\ and\ sex = Male$	0.61	0.19
c	$occupation = Prof\ specialty\ and\ class = > 50K$	0.55	0.11

Each algorithm finds different number of contrasts. To have a meaningful comparison, we only compare the top k contrasts where k is decided by the algorithm that finds the least number of contrasts or 100, whichever is smaller. The itemsets are sorted on the interest measure which is used to compare the algorithms in the experiment i.e. the itemsets are sorted in decreasing order on difference. The datasets are from the UCI repository and are shown in Table 2.

The * and - in Table 4 indicate that the distributions are not significantly different from SDAD-CS according to the Wilcoxon Mann Whitney test, and that the experiment was not able to be completed, respectively. The results indicate that on average, SDAD-CS NP finds the best results followed by Cortana. However, many of the contrast found are redundant when analyzed qualitatively. For example, in the Shuttle dataset, SDAD-CS seems to find very bad contrasts compared to Cortana. Further analysis of the patterns show that $Attr\_1$ in (-inf, 54.0] has probabilities 0.91 and 0.01 in the 2 groups respectively and $Attr\_9$ in (-inf, 2.0] has probabilities 0.77 and 0. Cortana then finds another pattern $Attr\_1$ in (-inf, 54.0] and $Attr\_9$ in (-inf, 62.0] with probabilities 0.91 and 0.01 which is clearly not improving the pattern found in the previous level. However, these strong patterns contribute towards increasing the average interest measure. Comparing the results manually indicate the SDAD-CS finds all the non-redundant contrasts. Moreover, if we restrict the algorithms to find only patterns at the first level, SDAD-CS finds stronger patterns. Additional experiments were conducted to validate our algorithm on semiconductor manufacturing data and initial results indicate SDAD-CS found the most interesting patterns qualitatively.

We compare the time cost for MVD, SDAD-CS and SDAD-CS NP in Table 5. It should be noted that the time observed is only representative and may not be an accurate comparison. The implementation standards were kept similar, however, it is possible that the algorithms could be made faster by some implementation optimizations. In general SDAD-CS explores more spaces but that may not correlate to time taken. This may be because at each space, MVD is more computationally expensive. SDAD-CS with pruning is the fastest in general.

For each dataset we show the number of Redundant, Unproductive and Independently Productive Contrasts in the top 100 patterns without applying the filter. The results are shown in Table 6. As shown in the table the majority of the contrasts may not be interesting to the user.

6 CASE STUDY: ANALYSIS OF MANUFACTURING DATA

The previous section showed the ability of our algorithm to find better contrast patterns than other state of the art algorithms, however, through this section, we show the utility of contrast pattern mining in a real world scenario. We demonstrate that contrast patterns have the capability to find insightful information in a dataset from a high-volume semiconductor packaging factory. Note that the data has been encoded and normalized for intellectual property reasons. The patterns shown here can be easily interpreted by engineers which may not be possible with other machine learning paradigms. There are many examples where we can apply our algorithm in the semiconductor manufacturing domain, such as, analyzing the difference between machines or finding contrasts between a high yield and a low yield batch.

A large amount of information is collected on a per package level as material moves through the packaging and test process. The segment of processing in the manufacture of CPUs which is of interest to us, lies between the wafer test and final test operations. Wafer test is the test performed on an entire wafer before it gets singulated and packaged. Final test occurs after the packaging process, and is used to ensure the product is going to perform as designed under specified operating conditions. The data collected are tied to the part identifier and can consist of variables that have continuous, as well as, discrete values. One has parameters that correspond to contextual information related to, for example, the sequence of equipment that processed the part, including relevant subentities (e.g. test heads, pick and place heads, oven lanes, bond heads etc.), material information, along with parametric measurement information from sensors on process tools (such as temperatures and pressures), along with parametric measurements from test, as well as, and categorical data related to device performance. The data volumes are quite substantial when one looks at the data collected across the entire manufacturing flow and to show viability of the methods presented in this paper, a limited data set was normalized and used for testing. The intent of the activity is to use the methods to quickly identify manufacturing conditions that are resulting

Table 4: Quantitative Analysis of Contrast Sets

Dataset	SDAD-CS NP	MVD	Entropy	Cortana-Interval
		Mean Support Difference		
Adult	0.26	0.16	0.18	**0.27***
Spambase	**0.60**	0.42	0.36	**0.60***
Breast	0.86	0.46	0.51	0.87*
Mammography	**0.54**	0.36	0.52*	0.43
Transfusion	0.34	0.12	0.29	**0.35***
Shuttle	0.87	0.24	0.45	**0.92***
Credit Card	**0.26**	0.17	-	0.19
Census Income	0.48	0.32	-	**0.49***
Ionosphere	**0.76**	0.43	0.35	0.75*
Covtype	**0.49**	0.41	-	0.45

Table 5: Time taken by SDAD-CS and MVD

Dataset	Time in Seconds			Number of Partitions Evaluated		
	SDAD-CS	MVD	SDAD-CS NP	SDAD-CS	MVD	SDAD-CS NP
Adult	11.11	22.92	13.28	742	171	1024
Spambase	899.97	1901.88	1909.02	121604	592	283714
Breast	0.59	3.38	1.98	72	30	376
Mammography	0.71	0.88	0.86	188	19	248
Transfusion	0.42	0.69	0.40	86	23	84
Shuttle	45.80	80.82	105.95	302	382	1018
Credit Card	441.82	873.88	639.22	12126	3260	17202
Census Income	1490.34	2256.63	4127.39	594	2566	19516
Ionosphere	960.54	983.21	1169.56	117199	122854	7371104

Table 6: Number of Meaningful Contrasts

Dataset	Count (Meaningful Contrasts)	Count (Meaningless Contrasts)
Adult	3	97
Spambase	12	88
Breast Cancer	5	95
Mammography	11	37
Transfusion	7	23
Shuttle	9	91
Credit Card	1	99
Census Income	8	92
Ionosphere	10	90
Covtype	3	97

in failures at final test to prevent generation of additional scrap material. Note that these failures are typically sporadic and the upstream signals often get diluted with increasing process complexity.

For this experiment we took a sample of the entire population and compared it with parts that failed a particular test. The data consists of 148 attributes including around 30 continuous attributes. A quick look at the contrasts indicate some information of the failing parts. These insights allow engineers to tweak or change things that are a probable cause of the failure for the test. In Table 7 we see categorical contrasts which suggest that most of the problems occur on a particular placement tool and pick head on a specific chip attach module (CAM) and most of the issues usually occur on the back row of the tray holding the parts. Both the location of the impacted parts in the trays and the specific placement tool point to a potential issue with the rear lane of the module. We also see in Table 7 that the time the impacted parts are spending above the solder liquidus temperature in the reflow oven is unusually higher. Another issue noted in 7 indicates that the average peak reflow temperature for the chips that failed the test seem to be higher as well. These results indicate an issue with the temperature control in the rear lane of the reflow oven on that specific module. With this information, feedback and changes along the manufacturing line can be made in a timely manner, including blocking any additional processing on that specific equipment/location until the issue has been addressed. Other algorithms however give a large number of contrasts and are sometimes hard to interpret and act upon.

In any practical scenario the scaling of the algorithm is very important. Apart from introducing some pruning mechanisms in the previous sections in a real world scenario the data usually does not fit in main memory. A usual way to handle this situation

Table 7: Contrast Sets for Manufacturing data

Contrast Set	Supp. Diff	Supp. (Population)	Supp. (Sample)
CAM entity-SCE	0.27	0.28	0.55
Placement tool-JVF	0.27	0.28	0.55
10.5106<=CAM peak temp std<=10.6534	0.18	0.45	0.62
67.1875<=Die temp above std<=67.2486	0.17	0.13	0.3
CAM row location -Rear	0.16	0.34	0.5
92.0373<=CAM time above liquidus<=92.8009	0.16	0.04	0.21
254.1609 <= CAM Peak temperature <= 256.8191	0.14	0.24	0.37

is by parallelizing the algorithm by using multiple machines (in a cluster). It should be noticed that SDAD-CS is run on combinations of features (itemsets) and can be run parallel of each other. Intermittent results can be used to prune the next stages. There are multiple strategies proposed in the literature of association rule mining (or search tree algorithms) to find candidate itemsets in parallel. A simple strategy is find contrast patterns at each level of the tree in parallel and then use those results to prune the next level of the tree. There is some loss of pruning of the search space across subtrees, but by using this strategy, we can treat each problem at the computing nodes as an independent problem, and use the pruning strategies discussed earlier within subtrees. The times taken to complete the experiments are 18, 106 and 225 minutes for samples containing 100000, 500000 and 1000000 instances respectively with 120 features.

7 CONCLUSION

In this paper we propose a method to find contrast sets in mixed data. Using a binning strategy that automatically determines the size and number of bins for the continuous attributes, we find meaningful contrasts even with the presence of multivariate interactions. Our algorithm is capable of finding meaningful contrasts which can potentially be more interesting to users by finding productive, independently productive and non-redundant patterns. We discuss strategies to reduce the search space. The experimental results show the utility of our algorithm in real datasets and how it finds better contrasts compared to existing techniques. The algorithm introduced in this work provides insights for analyzing data that fits in the main memory. Manufacturing data, as well as data in many application domains are very large. We discussed a way to scale up the algorithm in a parallel environment. This can be potentially used to provide more accurate and real time patterns to engineers.

REFERENCES

[1] Rakesh Agarwal, Ramakrishnan Srikant, and others. 1994. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conference*. 487–499.

[2] Yonatan Aumann and Yehuda Lindell. 1999. A statistical theory for quantitative association rules. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 261–270.

[3] Stephen D Bay. 2001. Multivariate discretization for set mining. *Knowledge and Information Systems* 3, 4 (2001), 491–512.

[4] Stephen D Bay and Michael J Pazzani. 2001. Detecting group differences: Mining contrast sets. *Data mining and knowledge discovery* 5, 3 (2001), 213–246.

[5] Roberto J Bayardo Jr. 1998. Efficiently mining long patterns from databases. *ACM Sigmod Record* 27, 2 (1998), 85–93.

[6] Guillaume Bosc, Chedy Raïssy, Jean-François Boulicaut, and Mehdi Kaytoue. 2016. Any-time diverse subgroup discovery with monte carlo tree search. *arXiv preprint arXiv:1609.08827* (2016).

[7] Vladimir Dzyuba and Matthijs van Leeuwen. 2017. Learning what matters–Sampling interesting patterns. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 534–546.

[8] Vladimir Dzyuba, Matthijs van Leeuwen, and Luc De Raedt. 2017. Flexible constrained sampling with guarantees for pattern mining. *Data Mining and Knowledge Discovery* 31, 5 (2017), 1266–1293.

[9] Usama Fayyad and Keki Irani. 1993. Multi-interval discretization of continuous-valued attributes for classification learning. (1993).

[10] Salvador Garcia, Julian Luengo, José Antonio Sáez, Victoria Lopez, and Francisco Herrera. 2013. A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning. *IEEE Transactions on Knowledge and Data Engineering* 25, 4 (2013), 734–750.

[11] Henrik Grosskreutz and Stefan Rüping. 2009. On subgroup discovery in numerical domains. *Data mining and knowledge discovery* 19, 2 (2009), 210–226.

[12] Henrik Grosskreutz, Stefan Rüping, and Stefan Wrobel. 2008. Tight optimistic estimates for fast subgroup discovery. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 440–456.

[13] Wilhelmiina Hämäläinen and Geoffrey I Webb. 2017. Specious rules: an efficient and effective unifying method for removing misleading and uninformative patterns in association rule mining. In *Proceedings of the 2017 SIAM International Conference on Data Mining*. SIAM, 309–317.

[14] Franciso Herrera, Cristóbal José Carmona, Pedro González, and María José Del Jesus. 2011. An overview on subgroup discovery: foundations and applications. *Knowledge and information systems* 29, 3 (2011), 495–525.

[15] Robert J Hilderman and Terry Peckham. 2005. A statistically sound alternative approach to mining contrast sets. In *Proceedings of the 4th Australia Data Mining Conference (AusDM-05)*. 157–172.

[16] Rohan Khade, Jessica Lin, and Nital Patel. 2015. Frequent Set Mining for Streaming Mixed and Large Data. In *Machine Learning and Applications (ICMLA), 2015 IEEE 14th International Conference on*. IEEE, 1130–1135.

[17] Rohan Khade, Jessica Lin, and Nital Patel. 2018. Finding Contrast Patterns for Mixed Streaming Data. In *Proceedings of the 21st International Conference on Extending Database Technology (EDBT),*. 632–641.

[18] Rohan Khade, Nital Patel, and Jessica Lin. 2015. Supervised Dynamic and Adaptive Discretization for Rule Mining. In *SDM Workshop on Big Data and Stream Analytics*.

[19] M. Lichman. 2013. UCI Machine Learning Repository. (2013). http://archive.ics.uci.edu/ml

[20] Michael Mampaey, Siegfried Nijssen, Ad Feelders, and Arno Knobbe. 2012. Efficient algorithms for finding richer subgroup descriptions in numeric and nominal data. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*. IEEE, 499–508.

[21] Petra Kralj Novak, Nada Lavrač, and Geoffrey I Webb. 2009. Supervised descriptive rule discovery: A unifying survey of contrast set, emerging pattern and subgroup mining. *Journal of Machine Learning Research* 10, Feb (2009), 377–403.

[22] Ramakrishnan Srikant and Rakesh Agrawal. 1996. Mining quantitative association rules in large relational tables. In *Acm Sigmod Record*, Vol. 25. ACM, 1–12.

[23] Matthijs van Leeuwen and Arno Knobbe. 2012. Diverse subgroup set discovery. *Data Mining and Knowledge Discovery* 25, 2 (2012), 208–242.

[24] Ke Wang, Soon Hock William Tay, and Bing Liu. 1998. Interestingness-Based Interval Merger for Numeric Association Rules.. In *KDD*, Vol. 98. 121–128.

[25] Geoffrey I Webb. 1995. OPUS: An efficient admissible algorithm for unordered search. *Journal of Artificial Intelligence Research* 3 (1995), 431–465.

[26] Geoffrey I Webb, Shane Butler, and Douglas Newlands. 2003. On detecting differences between groups. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 256–265.

[27] Geoffrey I Webb and Jilles Vreeken. 2014. Efficient discovery of the most interesting associations. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 8, 3 (2014), 15.

[28] Geoffrey I Webb and Songmao Zhang. 2005. K-optimal rule discovery. *Data Mining and Knowledge Discovery* 10, 1 (2005), 39–79.

[29] Gangyi Zhu, Yi Wang, and Gagan Agrawal. 2015. SciCSM: novel contrast set mining over scientific datasets using bitmap indices. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*. ACM, 38.

Predicting “What is Interesting” by Mining Interactive-Data-Analysis Session Logs

Tova Milo
Tel Aviv University
milo@post.tau.ac.il

Chai Ozeri
Tel Aviv University
chaiozeri@mail.tau.ac.il

Amit Somech
Tel Aviv University
amitsome@mail.tau.ac.il

ABSTRACT

Assessing the interestingness of data analysis actions has been the subject of extensive previous work, and a multitude of interestingness measures have been devised, each capturing a different facet of the broad concept. While such measures are a core component in many analysis platforms (e.g., for ranking association rules, recommending visualizations, and query formulation), choosing the most adequate measure for a specific analysis task or an application domain is known to be a difficult task.

In this work we focus on the choice of interestingness measures particularly for Interactive Data Analysis (IDA), where users examine datasets by performing sessions of analysis actions. Our goal is to determine the most suitable interestingness measure that adequately captures the user’s current interest *at each step of an interactive analysis session.*

We propose a novel solution that is based on the mining of IDA session logs. First, we perform an offline analysis of the logs, and identify unique characteristics of interestingness in IDA sessions. We then define a classification problem and build a predictive model that can select the best measure for a given a state of a user session. Our experimental evaluation, performed over real-life session logs, demonstrates the sensibility and adequacy of our approach.

1 INTRODUCTION

Assessing the potential *interestingness* of the output generated by a data analysis action has attracted considerable attention both in research and in the industry, and was proven highly useful for tasks such as association rules ranking [18], choosing data visualizations [31], data summaries [6], query formulation [27], etc.

Consequently, a multitude of interestingness measures has been suggested in previous work, each measure attempting to capture a different aspect of the broad “interestingness” concept. For example, *diversity measures* favor data patterns in which elements differ significantly from one another, *peculiarity measures* favor data patterns that display anomalous behavior. Other measures capture *conciseness*, *novelty*, and so on. Consequently, an important (and still open) question is *how can one choose which interestingness measure to employ?* To tackle this exact question, several comprehensive empirical evaluations have been conducted (e.g. [12, 17, 18, 22, 29]). These excellent surveys conclude that (1) there is no single measure that consistently outperforms the rest and (2) the adequacy of specific measures depends heavily on the task at hand and on the application domain.

Whereas most previous work examines the interestingness of specific, singular analysis actions, our work focuses on the interestingness notion within the entire process of Interactive Data Analysis (IDA). There is a growing understanding in the industry and research communities that users analyze datasets *interactively* by issuing a *sequence* of analysis actions of different types (e.g., OLAP, visualizations, mining). Notable and ubiquitous IDA tasks are data exploration, business intelligence (BI), and fraud detection. Typically in IDA, users interact with a dataset by executing a series of analysis actions, referred to as *session.* After issuing an action (e.g. group-and-aggregate, filter, plot, cluster), the user examines its results output (which we call *display*) then decides if and which action to issue next.

Our goal is to predict, at each step of a user’s analysis session, what is the most suitable interestingness measure that adequately captures the user’s current interest.

If successful, such a predictive model may be highly useful in several analysis “meta” tasks, such as facilitating an evaluation method for analysts’ effectiveness, improving existing (and future) analysis recommender systems (e.g. [16, 25, 31]) and enhancing systems for automatic data exploration e.g. ([9, 24]).

To our knowledge, our work is the first to consider dynamically changing interestingness in the context of IDA. Therefore, Our first intent is to demonstrate that interestingness in IDA has different characteristics than the ones assumed in previous work, posing both challenges and opportunities. We identified the following key characteristics, based on an in-depth analysis of real-life session logs:

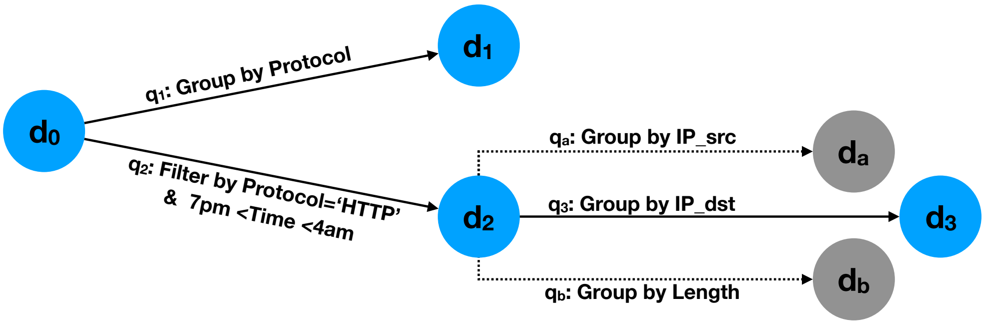
1. There is not one measure that holistically captures “what is interesting” in IDA sessions. When using a single interestingness measure, even if it is the most prevalent one, our experimental evaluation shows that it is inadequate for more than *two thirds* of all our examined cases. Also, many valuable, interesting actions obtain high scores w.r.t. one measure, and low to medium scores by others, hence, different measures need to be employed in different cases.

2. Interestingness (and correspondingly, the measure used to capture it) changes dynamically even in the same user session. We empirically show that within a single user session the “most adequate” interestingness measure changes every 2.2 analysis actions on average.

3. Interestingness is *contextual*. Namely, the analysis context, comprising of previous actions in the same session and their result displays, is, to some extent, correlated with the interestingness preferences of the user (and the measures capturing them).

The following example illustrates the dynamically changing nature of the interestingness notion in a typical IDA scenario.

Example. Clarice is a cyber security analyst assigned to examine inbound network traffic data of a large organization, with the goal of searching for back-door communication channels. She loads the dataset to an IDA interface and performs a sequence of actions, as illustrated in Figure 1 (the sequence of actions is depicted on the upper part of the figure, and the bottom tables depict the actions’ results) First, she performs a *group-by* on the field “Protocol” in order to view the amount of traffic of each



d_0				
Time	IP_src	IP_dst	Protocol	Length
13:00:37	10.1.31.15	8.8.4.4	DNS	76
13:00:41	10.1.31.15	74.25.22.7	TCP	62
...	...	...	...	...
23:56:41	10.2.22.8	8.8.8.8	DNS	72
23:56:41	10.2.22.8	5.8.63.3	SSL	277

d_1	
Protoco	Count
HTTP	492910
SSL	22479
...	...
SMB	96
NBNS	4

d_2				
Time	IP_src	IP_dst	Protocol	Length
19:00:00	10.2.26.3	5.8.63.6	HTTP	549.0
19:00:00	10.2.28.12	5.8.63.6	HTTP	350.0
...	...	...	...	...
03:57:42	10.2.23.13	5.8.63.6	HTTP	279.0
03:58:01	10.1.14.11	5.8.63.2	HTTP	471.0

d_3	
IP_dst	Count
5.8.63.6	100371
5.8.63.2	50538

d_a	
IP_src	Count
10.2.22.5	1624
10.2.28.12	1528
...	...
10.1.15.2	1316
10.1.14.11	1422

d_b	
Length	Count
231	971
238	1966
...	...
723	124
472	631

Figure 1: An Example IDA Session

network protocol. She then returns to the previous display, and issues a *filter* action in order to examine HTTP packets transmitted after business hours (i.e.,between 7pm to 4am). Last, she performs another group-by action on the attribute 'Destination IP', in order to obtain a summary of the uncommon packets (transmitted after business hours) categorized by their destination IP address.

Since Clarice is an expert analyst, assume that all her actions yield interesting result displays. However, *each action is considered interesting according to different measures*: For instance, her first group-by action results in a display (as depicted in Figure 1) that summarizes all traffic according to the network protocol. If we use a *diversity*-based measure (such as Variance [15] or Simpson [15]) to assess the interestingness of this display, it would rank high - as the amount of packets greatly differs between the different protocols. However, if assessed by *peculiarity* based measures (e.g. [19, 28]), which consider displays showing anomalous/extreme patterns as more interesting, this display may obtain a low interestingness score.

In contrast, the results display of the second (filter) action contains rather unusual packets transmitted after business hours (e.g., having a very small length or issued from IP addresses that are uncommon in the dataset). Therefore, this display may be ranked as highly interesting by *peculiarity* measures. However, it may yield lower scores from *diversity* measures (since the attributes of the unusual HTTP packets are rather evenly distributed).

Last, her third (group-by) action results in a compact summary of the unusual HTTP packets, grouped by their destination IP address. This display is considered interesting according to *conciseness* based measures (e.g. [6]) that favor displays conveying a small, human-readable number of rows that summarize a large number of elements. Indeed, all unusual HTTP packets are outgoing merely two different destination addresses. However, this display obtains a low score from *diversity* and *peculiarity* based measures. ■

As illustrated in the example, **although the expert analyst makes actions resulting in interesting displays - each action is supported (i.e., given a high score) by a different interestingness measure, and obtains low to medium scores by others**.

However, attempting to predict the most adequate interestingness measure at each point in an IDA session poses immediate challenges and questions: (1) How can we derive the "ground truth"? Manual labeling may be possible yet time-consuming and costly. (2) Even the simple task of examining a single action and determining which measure finds it more interesting than others is quite difficult, since the different measures capture different facets of interestingness and have different value ranges and distributions.

To overcome these challenges we propose mining analysis session-logs, containing previous analysis actions performed by the same or other users. Given the current user's state in a session, we search the repository to find points in other sessions that are *similar* to the user's state. We analyze what were the relevant measures for these previous sessions and use them to predict the most adequate measure for the current user.

The key contributions of our work are as follows:

1. A simple yet generic data model for interestingness in an IDA environment. Our model is compatible with different types of interactive analysis platforms, from traditional SQL to OLAP and modern web-based interfaces (such as Splunk and Tableau). Our generic model supports a wide range of existing interestingness measures and can be easily extended to support user-defined measures as well.

2. A-posteriori, offline interestingness analysis. The session logs do not provide any information about what parts of the session were interesting and which measures adequately capture it. We therefore devise methods for deriving the most adequate measure at each point in a *past* session by analyzing the interestingness of the *next-action* performed within the same session. By using new techniques for computing *relative interestingness*, we can properly compare the scores of this action (given by different measures) and determine which one best captures its interestingness.

3. Online Interestingness Prediction. Using the results of the above offline analysis, we define a classification problem and build a predictive model that can select the most adequate measure for a current session-state, without knowing its continuation. This model can be used for a dynamic, context-aware selection of interestingness measures in an *ongoing* session, and

Class	Measure	Definition	Reference
Diversity	Variance	$\frac{\sum_{j=1}^{m}(p_j-\bar{q})^2}{m-1}$	[15]
Diversity	Simpson	$\sum_{j=1}^{m} {p_j}^2$	[15]
Dispersion	Schutz	$1 - \frac{\sum_{j=1}^{m} p_j-\bar{q}}{2m\bar{q}}$	[15]
Dispersion	MacArthur	$1 + \sum_{j=1}^{m} \frac{p_j+\bar{q}}{2} log_2 \frac{p_j+\bar{q}}{2} - \frac{log_2 m-\sum_{j=1}^{m} p_j log_2 p_j}{2}$	[15]
Peculiarity	Outlier Score Function	See [19]	[19]
Peculiarity	Deviation	$\delta_{KL}(\{p'_j\}\mid\{p_j\})$	[31]
Conciseness	Compaction Gain	$\frac{\lvert O \rvert}{m}$	[6]
Conciseness	Log-Length	$1 - \frac{min(\log m, c)}{c}$	Following [26]

Table 1: A Partial List of Interestingness Measures

when combined with analysis assistance tools, can aid users in discovering interesting patterns in the data, compose meaningful visualizations, and so on.

4. Experimental evaluation. We evaluated our framework on real-life IDA session logs [1] acquired from over 50 experienced analysts in the domain of cyber security. Our empirical results show that our system can predict, with high accuracy, the correct measure to be used at each point in a user session.

The paper is organized as follows. In Section 2 we describe our data model for IDA interestingness and articulate the problem of interestingness measure prediction. Section 3 describes our framework, comprising the offline interestingness analysis an the online predictive model. Our experiments are detailed in Section 4. Last, we overview related work in Section 5, and conclude in Section 6.

2 BACKGROUND & PROBLEM DEFINITION

We first present a simple yet generic formal model for the IDA process, then describe the different notions of interestingness that we use. Last, we define the problem of *dynamic interestingness measure selection.*

2.1 IDA: Model and Definitions

An IDA session begins when a user loads a dataset, denoted $\mathcal{D}$, to an analysis UI (could be SQL, OLAP or a visualization-driven interface such as Tableau). Then, the user executes a series of *analysis actions* (e.g. SQL queries or visualization actions) $q_1, q_2, \ldots,$ examining the obtained results after each one. The results-set of action q_t, executed at step t is called a *display* (representing the results "screen") and denoted by d_t. The preliminary display is d_0, representing the dataset before any action was performed.

IDA session works intuitively like website navigation - at each point the user may invoke an action or backtrack to a previous display and take an alternative navigation path. We thus model an analysis session as an ordered labeled tree[1], denoted S. The nodes represent displays, and the edges outgoing from each node are labeled by the executed action and lead to the resulting display node.

We use S_t to denote the session after step t, namely the *state* in which the user examines the results display d_t, before deciding whether to execute a next action q_{t+1} or conclude the analysis.

Figure 1 illustrates an *analysis session tree* that corresponds to our running example, in which a user interacts with a dataset of network packets (ignore, for now, the dashed and gray parts). The directed edges represent actions q_1-q_3, and the nodes d_1-d_3 represent their corresponding results displays. The root node, d_0 represents the first display of the dataset before any action was invoked.

Last, we assume throughout this work that past analysis sessions are recorded in a *session log*. We denote by $\mathcal{R}$ a repository of such recorded sessions.[2]

2.2 Interestingness Notions for IDA

A typical interestingness measure, denoted i, takes as input an action q and its results display d, and returns a real number $i(q, d) \in \mathbb{R}$ indicating how interesting are the results (d) of the action q (higher score indicates a more interesting action).[3] For brevity, when d is clear from context, we omit it and simply refer to the *interestingness of action q* by $i(q)$.

While a multitude of different interestingness measures exist (See Section 5 for a discussion), w.l.o.g. we focus our attention in this work to eight common measures from the literature that correspond to four different facets of interestingness, following the categorization in [12] and [15].

The formal definitions of the considered measures (along with a corresponding reference) are provided in Table 1. Next, we intuitively describe each measure, then provide several examples.

Diversity. Diversity measures, e.g. *Simpson* and *Variance* [15] rank higher displays whose elements demonstrate notable differences in values. The definitions of the example measures that we use here are stated in Table 1. The notations are borrowed from [15], assuming an *aggregated* results display: m is the number of groups, v_j is an aggregated value for group j, $p_j = \frac{v_j}{\sum_{k=1}^{m} v_k}$ and $\bar{q} = \frac{1}{m}$. Example below.

Dispersion. In contrast to diversity, dispersion measures e.g. *Schutz* and *MacArthur* [15] favor displays consisting of relatively similar elements.[4]

[1] If the same display is generated twice (yet on different paths) it is represented by two different nodes

[2] Analysis sessions may either be recorded by the IDA platform, or, when it does not provide such a service, reconstructed from standard query logs by methods e.g. [32].

[3] Some measures consider more information, such as a reference display or a model of the user's prior belief. While our framework can be naturally extended to support such measures we omit them for the simplicity of presentation.

[4] In some cases the inverse score of a diversity measure can be used to evaluate dispersion, and vice versa.

Class	Measure (i)	Interestingness Scores				Relative Scores	
		$i(q_1)$	$i(q_3)$	$i(q_a)$	$i(q_b)$	$\bar{i}(q_3)$ (RB)	$\bar{i}(q_3)$ (N)
Diversity	Simpson	0.65	0.55	0.15	0.63	1	-0.07
Dispersion	Schutz	0.28	0.83	0.91	0.52	1	1.37
Peculiarity	Outlier Score Function	19.37	1.76	1.02	7.05	1	-0.74
Conciseness	Compaction Grain	51176	75454	39819	25706	2	2.2

Table 2: Interestingness Scores

Peculiarity. A display is peculiar if it presents or contains anomalous patterns. An example is a *Deviation*-based measure [31] that ranks a display higher if it demonstrates a difference from some reference display (e.g. the root display d_0); the p_j notation in the formal definition (Table 1) is the same as above, $\{p_j\}$ denotes the discrete distribution of p_j values, and $\{p'_j\}$ denotes the distribution of the aggregated values in the reference display. $\delta_{KL}(A|B)$ is the Kullback-Leibler divergence distance of the two distributions. Another peculiarity measure is the *Outlier Score Function* [19] (OSF) that focuses on the peculiarity of a single element (i.e, a single tuple, group, or cube cell) within the examined display. The final peculiarity score is simply the *maximum* of the elements' individual scores (See [19] for full details).

Conciseness. Such measures consider the *size* of the display, i.e. the number of elements it contains. Intuitively, displays that convey thousands of rows are difficult to interpret, therefore are considered less interesting. *Log-Length* scores a display proportionally to the log of its size, bounded by a constant c. *Compaction-Gain* (CG) compares the size of the particular display to the number of tuples in the original dataset (denoted O in the formula in Table 1).

In Section 5 we discuss other types of interestingness measures e.g. *surprisingness, actionability* and how they can also be incorporated in our framework.

As the reader can observe, each measure values different properties of the data and may rank a given display differently. The following example illustrates the interestingness evaluation according to the measure types presented above.

Example 2.1. Consider again the IDA session described in Figure 1, and the different interestingness measures described in Table 1. Let us assess the interestingness evaluation of actions q_1 (Group by 'Protocol') compared to q_3 (Group by 'Destination IP'). In Table 2, we report the interestingness scores of q_1 and q_3 according to four different measures, one from each interestingness type (we do not exemplify the calculation of each score, for that we refer the reader to the original papers as depicted in Table 1). Let us examine some of the scores:

1. Diversity, Dispersion: The results of q_1 are considered more interesting than of q_3 as $i(q_1) = 0.65$ and $i(q_3) = 0.55$ (See Table 1). This is due to the larger deviation in the groups' size in d_1 than what appears in d_3 which only contains two groups that are rather even in size. In terms of Dispersion, in which displays with less variations yield higher scores, q_3 is indeed more interesting than q_1 ($i(q_1) = 0.28$ and $i(q_3) = 0.83$)

2. Conciseness: In terms of the Compactness Grain measure, which considers the ratio between the number of tuples to the number of elements (e.g. groups) that covers them, we can see that $i(q_1) = 51,176$ while $i(q_3) = 75,454$. q_3 obtains a higher score than q_1 as its results-display d_3 covers a high number of packets in merely two groups (IP addresses), while d_1 covers all packets with a larger number of groups, one for each network protocol.

Interestingness Measure Prediction. Given a predefined set of interestingness measures $\mathcal{I}$, and a user session state S_t after t steps, our goal is to predict which measure in $\mathcal{I}$ adequately captures "what is interesting" at this point of the session. **Our main hypothesis in this work is that interestingness (and therefore the adequacy of measures) is *contextual*, hence correlated with previous actions taken by the user in the same session.** We illustrate this with our running example.

Example 2.2. Consider the Example Session in Figure 1, at state S_2, i.e. when the user examines Display d_2, before invoking the next action q_3. Our goal, as stated above, is to predict which measure from $\mathcal{I}$ (e.g., Diversity-based, Peculiarity, Conciseness, etc.) best captures the interestingness at this moment. While this seems like a challenging task, examining the previous actions in the session provides some intuition regarding which measure is preferable: In q_2 the user filters all packets to focus on *unusual HTTP traffic occurring after business hours.* As she examines a long list of anomalous elements, it is likely that she is interested in a more *concise* display that summarizes the data, rather than a display that demonstrates another peculiar pattern or one with high Dispersion.

Following this premise, we form a supervised multi-class classification problem that considers session-states as "samples" and assign them a "label" corresponding to interestingness measures in $\mathcal{I}$. In other words, we will assemble a training set containing labeled samples of the form $\langle S_t, i\rangle$ and build a predictive model $F(S_t) \approx i$ that best fits the training data.

To properly define this process, one needs to (1) develop means to determine "what is interesting" in a current state of an IDA session and which measure captures it best. (2) Once this is determined, develop a classification model for predicting what measure best captures the interestingness for the current session state.

We address the two issues in the following section, where we explicitly define the predictive task and model.

3 INTERESTINGNESS PREDICTION FRAMEWORK

We next describe our solution for interestingness measure prediction in IDA sessions. Before we describe our predictive model, we provide (Section 3.1) a set of techniques for *offline interestingness analysis*, in which we retrospectively derive what was the most suitable measure $i \in \mathcal{I}$ for a session state S_t, using the next action q_{t+1}. Then we describe in Section 3.2 how these techniques are used when constructing the training set and building the predictive model.

3.1 Offline Interestingness Analysis

Theoretically speaking, there are several possible ways to examine a session state S_t and determine what is the most suitable measure for it.

First, one can perform *manual labeling* by using expert analysts, familiar with interestingness measures, that can label a session state S_t with the most suitable measure as they see it. The problem with this approach is that it requires, first of all, a great manual effort that is not easily transfered to other contexts (e.g. IDA session logs performed on different datasets or for different purposes). Also, existing measures are often not intuitive even to expert analysts. A second approach could be prompting for user feedback at each session-state, gathering details about the user's intention, goals, and details regarding if and why the current display is interesting (w.r.t. each facet captured by the measures in $\mathcal{I}$). But this method, like the previous one, requires a considerable manual effort (this time by the users performing the sessions).

In contrast to these approaches, we devise means for deriving the adequate measure for a session state S_t solely by examining the continuation of the session. Assume for a moment that our repository only contains actions resulting in interesting displays. Then our key assumption is that for a session state S_t, **if the next action in the same session q_{t+1} is interesting *enough*** (we explain how this is determined in the sequel), **then we can use it to derive what is the measure that best captures the interestingness at S_t**.

A simplistic implementation of the above assumption is to simply choose, given a session state S_t, the measure $i \in \mathcal{I}$ that produces the highest score$i(q_{t+1})$. However, since the measures in $\mathcal{I}$ may produce scores of different value ranges and distributions, this method must be refined.

We therefore devise two interestingness *comparison methods* that are largely impartial to such biases. In each comparison method, we first compute, w.r.t. each measure i, the *relative interestingness score* of an action q denoted $\bar{i}(q)$, which is *comparable* to the relative scores $\overline{i'}(q)$ obtained by other interestingness measures $i' \in \mathcal{I}$. Once we have the unbiased, relative scores, we can simply choose a measure yielding the *maximal* relative interestingness as one that best captures the interestingness of an action q_{t+1}, hence is suitable for S_t. We call a measure *dominant* w.r.t. action q, denoted $i^{\star}(q)$, if it yields the maximal relative interestingness, i.e. $i^{\star}(q) = argmax_{i \in \mathcal{I}}(\bar{i}(q))$.

Last, we note that in real-life analysts are imperfect, hence IDA sessions may contain erroneous/redundant actions or simply uninteresting ones. We will show in the sequel how our analysis is used to eliminate such actions and minimize their negative effect.

We start by describing the Reference-Based Comparison method, which is comprehensive but expensive to compute. To overcome this, we then present an alternative method, the Normalized Comparison, which also reduces the score bias but requires a lower computational cost.

Reference-Based Comparison. Our first method for unbiased interestingness comparison, denoted Reference-Based comparison, examines the score of an action q as obtained by a particular measure and compares it to the scores achieved when employing *alternative actions*. Hence, instead of comparing individual scores of different measures for an action q, we first calculate how "high" each measure ranks q compared to a reference set of alternative action, denoted $R(q)$ (in Section 4 we explain how we generate $R(q)$ in our prototype implementation). Then we can simply derive that the measure i which ranks q the highest, is the one that best captures its interestingness (in case there is a tie, all measures that yield the highest relative interestingness are returned).

Algorithm 1 $ReferenceBasedComparison(\mathcal{I}, \langle q, p, d \rangle, R(q))$

```
1: for $q' \in R(q)$ do
2:     $d' \leftarrow$ The results of action q' on display $p$
3:     for $i \in \mathcal{I}$ do
4:         Compute the score $i(q', d')$
5: for $i \in \mathcal{I}$ do
6:     Compute the score $i(q, d)$
7:     $\bar{i}(q) \leftarrow |\{q' \in R(q) \mid i(q', d') \leq i(q, d)\}|$
8: return $argmax_{i \in \mathcal{I}}(\bar{i}(q))$
```

The Reference-Based comparison is depicted in Algorithm 1. It takes as input a set of measure $\mathcal{I}$, a tuple $\langle q, p, d \rangle$ which includes an action q together with its *parent* display p (i.e., the display on which q was employed), and its results display d, and last, a set of *alternative* actions, denoted $R(q)$. It then works as follows.

First, we execute each action in the reference set $R(q)$ (from the same parent display p of action q) then calculate its interestingness scores w.r.t. each measure in $\mathcal{I}$ (Lines 1-4). Next, we calculate the raw scores for q w.r.t. each measure $i \in \mathcal{I}$ (Line 6), then derive the *relative interestingness* score of q , denoted $\bar{i}(q)$, by counting the number of actions in $R(q)$ that obtained a lower interestingness score than q (Line 7). Last, we return the dominant measure(s) $i^{\star}(q)$ that produced the highest relative interestingness score (Line 8).

Example 3.1. We use the Reference-Based method to assess which is the dominant interestingness measure for action q_3 in our example session (Figure 1). While we can see from Table 2 that d_3 has high Conciseness score, it is also rather disperse. However is it more *disperse* than *concise*, or vice versa?

Let $R(q_3)$ be the set of alternative actions $\{q_a, q_b\}$ (The dashed edges and Grey displays in Figure 1). Their interestingness scores w.r.t. the different measures appear in Table 2. The relative interestingness scores for q_3 appears in the middle section of Table 2. For instance, in terms of Dispersion, since q_3 has a higher score than q_b yet a lower score than q_a its relative score is 1. However, in terms of Conciseness, since the score of q_3 is higher than both q_a and q_b, its relative interestingness (Conciseness) is 2. Indeed, Conciseness yield the highest relative score for q_3, hence is chosen as the dominant measure $i^{\star}(q_3)$ that best captures the interestingness of action q_3.

While this method completely eliminates the score biases of the different measures, note that it is rather expensive to compute (we demonstrate this in Section 4) as it requires to execute, at each comparison, all alternative actions in the reference set and compute their interestingness scores. Consequently, we devise a second, more efficient comparison method which significantly reduces the score biases using statistical analysis.

Normalized Comparison. The second interestingness comparison method, denoted *Normalized* Comparison, eliminates the score bias due to differences in the range and value distributions, by applying a two-staged normalization process to each measure: (1) to tackle the differences in the measures' value distributions we apply a Box-Cox [5] power transformation that makes the values resemble a normal-distribution. (2) To tackle the differences in the value ranges, we calculate the mean and standard deviation of each measure's (transformed) value distribution then employ z-score standardization [30] so that each computed value now represents the number of standard deviations the original value differs from the mean. The transformed and standardized score of each action q is defined to be its *relative interestingness score* $\bar{i}(q)$, and can now be compared to the relative scores produced by other measures in $\mathcal{I}$.

The Normalized Comparison, depicted in Algorithm 2, is divided into two parts.

First, Stage (1) of the normalization process is applied, in a preprocessing manner, to a sample of the score distribution of each measure. The function PreProcess (Lines 1-8) takes as input the set of measures $\mathcal{I}$ and a set QD of actions and their corresponding result displays, i.e. pairs of the form $\langle q, d \rangle$ (such a set can be extracted from the session repository $\mathcal{R}$). It then calculates the interestingness score w.r.t. each measure in $\mathcal{I}$ (Line 4) and transforms its value using the Box-Cox method (Line 5). Last, the mean and standard deviation of the transformed interestingness scores of each measure are returned (Line 8).

Once the preprocessing is done, the Normalized Comparison function (Lines 9-15) can be employed. It takes as input an action q with its results display d, a set $\mathcal{I}$ of interestingness measures and the mean and standard deviation of the transformed scores of each measure, and computes the dominant measure $i^{\star}(q)$ as follows: first, we calculate the interestingness scores $i(q, d)$ w.r.t. each measure in $\mathcal{I}$ and apply the Box-Cox transformation to it (Lines 11-12). Then the mean and standard deviation are updated (Note that Lines 11 to 13 can be skipped if theses values were already computed in the preprocessing phase), and the relative score is calculated by applying the z-score standardization (Line 14). Finally, we return the measure(s) that obtained the highest relative interestingness score $i^{\star}(q)$, as in the Reference-Based method.

Example 3.2. To illustrate the computation, we demonstrate how the Normalized method can be used to assess the measure $i^{\star}(q_3)$ which gives the highest relative score to action q_3 in our running example (depicted in Figure 1).

Assume we performed the preprocessing routine and calculated the scores of all other actions in the log, then performed the transformation and standardization described above. The normalized relative scores for action q_3 are depicted in the right-hand section of Table 2. Consistently with the previous example, the highest normalized score for q_3 is given by the *Conciseness* measure, as its score deviates more than 2.2 standard deviation from the mean conciseness scores.

Also, observe that similar scores of different measures can obtain significantly different results after the standardization process. For instance, the absolute scores of q_3 obtained by the *Dispersion* and *Peculiarity* measures are 0.74 and 0.71 (resp.), however their standardized scores are very different (2.39, -0.2 resp.)

In Section 4 we examine the correlation between the Reference-Based and the Normalized methods and compare their execution times. In what comes next, we describe how these methods are used to construct the training set and the predictive model.

3.2 Online Interestingness Prediction

We developed a predictive kNN-based model for selecting the most suitable measure at a particular session-state.

First we discuss what information is used to describe a session state S_t, then how the training set is constructed and the mechanism of the kNN-based classification model.

Algorithm 2 *NormalizedComparison*

```
1:  function PreProcess($\mathcal{I}$, $QD$)
2:      for $i \in \mathcal{I}$ do
3:          for $\langle q, d \rangle \in QD$ do
4:              Compute the score $i(q, d)$
5:              $\tilde{i}(q, d) \leftarrow$ BOX-COX($i(q, d)$)
6:      for $i \in \mathcal{I}$ do
7:          Calculate $\tilde{\mu}_i$ and $\tilde{\sigma}_i$, the mean and SD of all $\tilde{i}(q, d)$.
8:      Return $\tilde{\mu}_{\mathcal{I}}, \tilde{\sigma}_{\mathcal{I}}$, containing $\tilde{\mu}_i, \tilde{\sigma}_i \forall i \in \mathcal{I}$
9:  function NormalizedComparison($\mathcal{I}, \langle q, d \rangle, \tilde{\mu}_{\mathcal{I}}, \tilde{\sigma}_{\mathcal{I}}$)
10:     for $i \in \mathcal{I}$ do
11:         Compute $i(q, d)$
12:         $\tilde{i}(q, d) \leftarrow$ BOX-COX($i(q, d)$)
13:         Update $\tilde{\mu}_i$ and $\tilde{\sigma}_i$
14:         $\bar{i}(q) \leftarrow$ Z-SCORE($\tilde{i}(q, d), \tilde{\mu}_i, \tilde{\sigma}_i$)
15:     return $argmax_{i \in \mathcal{I}}(\bar{i}(q))$
```

Describing Session States. Recall that a session state S_t, is the subtree of S containing the first t actions and their result displays.

However, as older actions in the sessions may be of less importance to the classification model, we follow [25] and consider in our predictive model only the n most recent actions and displays, which we call the *n-context* of S_t, denoted c_t. More formally, c_t is defined as the minimal subtree of S that covers the most recent $min(n, 2t + 1)$ elements (i.e., displays and actions) up to step t (inclusive).

As an example, the 3-context at step $t = 2$ in our example session in Figure 1 includes Displays d_0 and d_2 and the action q_2.

In Section 4 we evaluate the predictive performance of the model when using n-contexts of various sizes.

Training Set Construction. Building a training set for a given session repository $\mathcal{R}$ and a set $\mathcal{I}$ of interestingness measures is performed as follows:

(1) Extracting n-contexts from the session repository. For each session state S_t in every session S in the repository we first compute its n-context. As we assume that the sessions in $\mathcal{R}$ are already represented as trees, deriving the n-context for a session state S_t can be done by a DFS-like traversal: Starting from display d_t, we process the nodes (i.e., displays) in reverse to the order of execution of their corresponding actions, considering only actions executed before step t until the size of the induced subtree (nodes+edges) reaches n, which is a configurable parameter in our framework.

For each session state S_t we keep a pair $\langle c_t, q_{t+1} \rangle$ comprising its corresponding n-context and the consecutive action q_{t+1} which will be used to derive its label (i.e., the suitable measure for that session state). For space efficiency, it is sufficient to store for each n-context only pointers to the original actions in the log rather than duplications. In Section 4 we explain how n-contexts are extracted and stored in our implementation.

(2) Assigning labels to n-contexts. For each pair $\langle c_t, q_{t+1} \rangle$, we use either one of the two comparison methods described above to find the dominant measure $i^{\star}(q_{t+1})$. This measure is assigned as a label to the n-context c_t, representing the most suitable measure for the corresponding session state S_t (recall that more than one measure may be qualified).

(3) Omitting globally "non-interesting" samples. Naturally, some of the actions in the session repository may be erroneous or simply non-interesting. Consequently, we want to eliminate from the training set samples whose consecutive action q_{t+1} is not interesting enough w.r.t. any of the measures in $\mathcal{I}$, hence can not be used for deriving the right measure. This is done by using a configurable threshold for the maximal relative interestingness score of q_{t+1}, denoted θ_I. If the relative interestingness obtained by the dominant measure $i^\star(q_{t+1})$ is lower than θ_I then we discard the sample $\langle c_t, q_{t+1}\rangle$ from the training set.

kNN-Based Classification. Once the training set containing labeled n-contexts is constructed, a classification model can be used, given n-context, to predict the *dominant* measure.

In principle, there are multiple techniques for performing supervised classification, however many of them requires a numeric vector representation for the samples (n-contexts in our case). However, we are not aware of such numeric representation of analysis sessions, yet numerous previous works [3, 11, 13, 25] define a notion of distance/similarity for analysis sessions (or a part thereof). For example the measure suggested in [25] uses tree edit distance to compare two session trees, together with two ground metrics that compare individual actions and displays. Alternatively, in [3] the authors suggest a measure based on local sequence alignment. We harness such a distance notion to form a simple kNN classifier: Given an n-context c_t, we search the training set for its k nearest-neighbors, then employ a majority-vote and return the most common label among the nearest neighbors. Last, it may be that some of the k nearest n-contexts may be too distant from c_t. To avoid the negative effect of such cases on the model's output, we use a distance threshold, denoted θ_δ, which is used to enforce a maximal distance (i.e., a minimal degree of similarity) between the kNN set and the given n-context. If the nearest neighbors retrieved are not similar enough, the model does not yield a prediction.

We intuitively explain this using our running example.

Example 3.3. Assume that our session repository $\mathcal{R}$ comprises only the example session depicted in Figure 1. We first extract n-contexts of e.g. size 3 from $\mathcal{R}$: For each $1 \leq t \leq 3$ we create the 3-context c_t: c_1 contains the single node d_0, c_2 contains d_0, q_1, d_1 and c_3 contains d_0, q_2, d_2. Recall from the previous examples that Compaction Gain measure (Conciseness) is the dominant measure for action q_3, therefore is used as a label for c_3.

Assume we are given another user's session on a different network log, however with the following last action q_u: "filter by protocol='SSL' & 10pm < Time<3am" (which is intuitively similar to q_2 in our example session in Figure 1). After extracting the n-context (containing q_u, its results, and its parent display) from the new session state, we predict what is the adequate measure using the kNN model: If using $k = 1$, then the most similar n-context in the repository is c_3, hence Compaction Gain will be return as prediction.

In Section 4 we explain how we evaluated the predictive model in various settings and compared its performance to several baseline approaches.

4 EXPERIMENTAL EVALUATION

We applied our offline interestingness analysis methods as well as the predictive model on an IDA session log containing real life analysis actions. We begin by describing the session log and our implementation choices, then describe our findings from the offline analysis. Last we evaluate the accuracy of our predictive model compared to other baselines, then test the effect on performance induced by the model's hyper parameters.

REACT-IDA: A repository of real-life IDA sessions. We used the only publicly available (to our knowledge) collection of recorded analysis sessions performed by real users on real-life datasets [1]. The sessions were collected as part of the experimental evaluation of an existing IDA recommender system [25], developed by some of the authors of this work (we discuss this system in more details in Section 5). The repository contains sessions performed by 56 network security analysts, recruited via dedicated forums, security firms, etc. The participating analysts were asked to explore 4 different network-logs datasets using REACT-UI [23], a dedicated, web-based analysis platform with an easy to use interface supporting data filtering, grouping and aggregation. Each dataset contains raw network logs that may reveal a distinct security event, e.g. malware communication hidden in network traffic, hacking activity inside a local network, an IP range/port scan, etc. After completing an analysis sessions, REACT-UI prompts the user to type a short summary of the findings. Sessions corresponding to summaries that successfully reveal the underlying security event are marked as *successful.* The repository contains a total of 454 sessions comprising 2460 distinct analysis actions, out of which 122 sessions (comprising 757 actions) are successful. The REACT-IDA session database in [1] also contains the original datasets used in the analysis, and the means for regenerating the actions and inspect their result displays, so that the each recorded session can be fully reconstructed.

4.1 Offline Analysis Evaluation

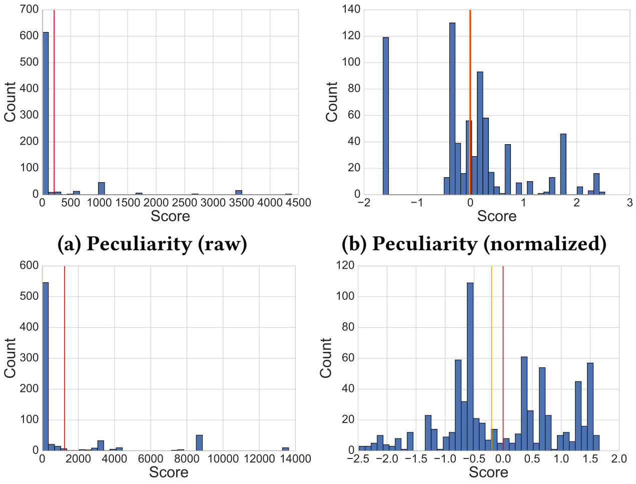
We next describe the application of our offline interestingness analysis methods on the REACT-IDA sessions database, accompanied by selected findings.

Computing interestingness scores. We re-executed the recorded actions in the REACT-IDA database and computed their interestingness scores w.r.t. all measures presented in table 1. Next, to form an unbiased set of measures $\mathcal{I}$, i.e., that does not contain dependent/highly correlated measures we computed the Pearson Correlation Coefficient for every pair of measures. While the average correlation score was 0.3 we saw that measures from *different types* (e.g. Dispersion, Peculiarity, Conciseness) have an average correlation of 0.071 compared to an average score of 0.543 obtained by measures of the *same type.* Consequently, we experimented with 16 different configuration of $\mathcal{I}$, containing one measure from each type. .

Applying offline comparisons. We applied both comparison methods to the REACT-IDA session database in order to calculate the dominant interestingness measure $i^\star(q)$ for each action q. As for the Reference-Based Comparison, recall that it compares the interestingness scores of an individual action to the interestingness scores of alternative actions. We constructed the reference action set as follows: For each action q with a parent display p we considered all actions in the databases from the same type (e.g. group-by, filter), omitting actions that when executed from display p result in displays comprising less than two rows.

As for the Normalized Comparison, we calculated the statistics over all actions (and their results) in the REACT-IDA database by applying the Box-Cox transformation and z-standardization to their raw interestingness values as described in Section 3.1. Each series of interestingness values (corresponding to a particular measure) was first shifted by a constant in order to eliminate negative scores (this is often required for power transformations). The configurable power parameter λ, which is used as the exponent for the values to be transformed, was determined using maximum-likelihood estimation, as is standard for such transformations.

As an example, Figure 2 depicts the scores histograms of the Outlier Score Function (Peculiarity) and the Compactness Grain (Conciseness) measures, before and after normalization. The red line in each figure represents the mean score, and the orange line represents the median. While the non-normalized scores are skewed towards zero, we can see that the normalized values distribute much more evenly, resembling a normal distribution.



(a) Peculiarity (raw) **(b) Peculiarity (normalized)**

(c) Conciseness (raw) **(d) Conciseness (normalized)**

Figure 2: Interestingness Scores Histograms

Understanding users' interestingness preferences. We studied the output of the interestingness comparison methods, namely the dominant measure w.r.t. each recorded action, in order to empirically validate our first two hypotheses presented in the introduction:

(1) Is there one measure/interestingness type that is sufficient to capture "what is interesting" in IDA? If indeed so, then it is sufficient to choose a-priori a single interestingness measure and apply it to all IDA tasks. To answer this question we counted how many actions in the REACT-IDA were labeled with the same dominant measure.

Figure 3 depicts the proportion of actions labeled with measures from the same interestingness type (averaged over all settings of $\mathcal{I}$), when using both the Reference-Based and the Normalized comparison methods.

We can see that in both comparison methods, the most common measure is dominant w.r.t. only 41% of the recorded actions, and the proportions of the rest of the measures types are rather evenly distributed. Due to ties, see that the sum of proportions is slightly larger than 1, i.e. where more than one interestingness measure was found dominant for the same action. Also, note that there are differences in some of the classes' size when a different comparison method is used (mainly in the Peculiarity and Conciseness classes). This is due to slight differences in the comparison base for each method: The Reference-Based method is affected by the parent display of the examined action (from which it was executed), whereas the Normalized is affected by the interestingness scores of other recorded actions, regardless of their parent displays and context.

The above result indicates that *there is no single interestingness measure that can be used for all actions in the repository.* However, as users' IDA sessions are performed on different datasets and for different purposes, one may still ask whether *one interestingness type is sufficient to capture interestingness within the same session?* If true, it may be sufficient to match a single interestingness measure with certain IDA tasks or datasets rather than dynamically choose a measure for each state in a user session. We therefore examined the relative interestingness scores of actions *of the same session.* We found that *on the course of a single session in the REACT-IDA repository, the dominant measure is changed every* 2.2 *steps* on average. This demonstrates that the interestingness preferences of the user, as well as the measures capturing them, are dynamically changing even within the same IDA session. In Section 4.2 we empirically validate our third hypothesis arguing that *interestingness is contextual* i.e., that one can successfully predict the right measure for a given session state S_t based on its corresponding n-context c_t.

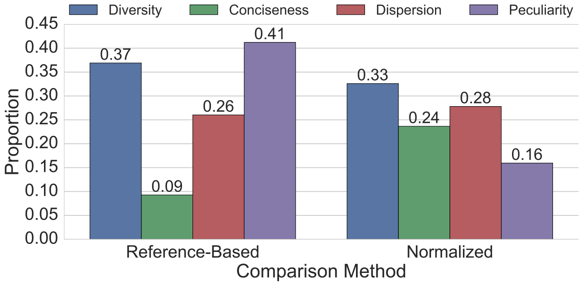


Figure 3: Interestingness Class Labeling Frequency

Correlation between the comparison methods. We studied how consistent is the output of the two comparison methods. First, we found that 68% of all recorded actions obtained exactly the same dominant measure as output, by both methods. We then performed a Chi-Square test for independence between the outputs of the comparison methods on all actions: The methods were found highly correlated with a negligible p-value $< 10^{-67}$. The latter result demonstrates the sensibility of our offline interestingness analysis, and that the methods may be used interchangeably.

Execution times. Applying the offline comparison methods includes three major parts:

(1) Calculating interestingness scores. Recall that the raw interestingness scores are precomputed for each measure in $\mathcal{I}$. While some measures are rather fast to compute (e.g. both Conciseness measures), others (such as the Outlier Score Function) are much more time consuming.

(2) Actions Execution. This part is relevant to the Reference-Based which compares the interestingness scores of a given action to a set of alternative actions. Thus, each such reference action needs to be executed on the same dataset (from the same parent display) by the IDA platform.

(3) Computing relative interestingness scores: Both methods calculate the relative interestingness score and return as output the dominant measure(s), yielding the highest relative interestingness.

We measured the running times required to apply the Reference-Based and the Normalized interestingness comparison methods, w.r.t. each of these computation parts. For the Normalized Comparison, running times include the corresponding segment in the preprocess routine for each action.

Table 3 depicts the average time required by the Reference-Based and the Normalized comparison methods in order to select

Component	Time (Seconds)	
	Reference Based	Normalized
Action Execution	2.218	–
Calc. Interstingness	4.84	0.106
Calc. Relative Scores	0.04	0.031
Total	**7.2**	**0.138**

Table 3: Offline Running Times

the dominant measure for a given action for each of the computation parts described above. First, observe that the Reference-Based requires an overall of 7.2 seconds compared to the Normalized which takes 0.138 seconds only. As for the computational parts, see that the Reference-Based requires a considerable amount of time for executing the alternative actions (Part 2) method (the average size of the reference actions set was 115) which is unneeded for the Normalized method. Consequently, the former requires computing all interestingness scores of the alternative actions, therefore its running time w.r.t. Part 1 is significantly longer than of the Normalized method. Part 3, in both methods is negligible.

4.2 Predictive Model Evaluation

We next describe our predictive kNN based model. We then explain how we evaluated its performance in comparison with several appropriate baselines.

Constructing the training set. **Extracting n-contexts.** We extracted *n*-contexts that belong to successful sessions from REACT-IDA repository using the DFS base method described in Section 3.2. We experimented with n-contexts of sizes 1 to 11 (we explain below how the default size was chosen). For each session state S_t, we stored the pair $\langle c_t, q_{t+1} \rangle$ comprising its context and the consecutive action.

Annotating n-contexts. We used the offline analysis results as described above, to label each pair $\langle c_t, q_{t+1} \rangle$ with its corresponding dominant interestingness measure $i^{\star}(q_{t+1})$. We then discarded all samples in which the maximal relative interestingness scores (obtained by $i^{\star}(q_{t+1})$) was smaller than the interestingness threshold θ_I (as described in Section 3.2).

In case that identical n-contexts obtained different labels[5] we unanimously labeled them by the most common label(s) associated with this n-context.

kNN Model implementation. As common for kNN based classification models, given a (non-labeled) n-context our model searches the training set for the top-k most similar labeled n-contexts, then selects the label by employing majority vote. To determine the similarity between n-contexts, we used the distance metric devised in [25] which was proven useful for IDA sessions. The metric is based on *tree edit-distance*, i.e. the minimum-cost sequence of *edit operations* (add, delete, and alter a node/edge) required to transform one n-context to the other. While a unit cost is given to delete/add operations, the cost of an *alter* operation (for a node/edge) is proportional to the similarity between the data displays and analysis actions. The latter is determined by two ground metrics for actions and displays: the first considers differences in the actions' syntax and the second measures the differences in the content of the compared displays.

As for the model's running times, we measured an average time of 6.04 milliseconds required to output a single prediction.

Parameter	Value Range	Default Configuration	
		Ref. Based	Norm.
n-Context Size (n)	$[1, 11]$	4	2
kNN Size (k)	$[1, 40]$	1	1
Dist. Thres. (θ_δ)	$[0, 0.5]$	0.2	0.1
Int. Thres. (θ_I)	$[0, 1]$ (RB) $[-2.5, 2.5]$ (N)	0.92	0.7

Table 4: Model Hyper-Parameters

We refrain from further discussing the computation costs and the scalability of the model and refer the reader to [25] for an in-depth performance evaluation of the nearest-neighbors search w.r.t. their distance metric.

Evaluation Methodology. We formed multiple test sets using the Leave-One-Out cross validation (LOOCV) method, i.e., in a single prediction task we take one sample (n-context) out of the training set to be used as a test set, then repeat the process for each and every sample.

We then used the following evaluation metrics: *(1) Accuracy*, which stands for the ratio between correctly predicted samples (true positives) and the number of all samples. Then, as is standard when evaluating multi-class classification models, we used the *(2) Macro-Averaged Precision* and *(3) Macro-Averaged Recall* measures, which takes the average of the precision (resp., recall) *w.r.t. each class* (i.e., each interestingness measure).[6] We also computed the *(4)Macro-Averaged F1* which is the harmonic mean of (2) and (3).

Last, since in some cases the kNN model does not output a prediction (recall from Section 3.2), we also measured the *(5) coverage rate*, namely the proportion of samples for which our model is able to produce a prediction.

Hyper-parameters tuning. Our model uses the following hyper-parameters: *(1) the size of n-contexts* used when constructing the training set to decide how many actions/displays are required to represent each session state S_t. *(2) The size of k*, i.e. the number of similar n-contexts used to perform a prediction. *(3) n-contexts distance threshold θ_δ*, representing the maximal distance allowed between each members of the kNN set and the given n-context. *(4) Interestingness Threshold θ_I*, i.e. the minimal relative interestingness score required for the sample to be considered as interesting and not to be discarded. Recall that relative interestingness is computed differently for each comparison method therefore this parameter has two sets of scales: For the Reference-Based method the threshold represents the minimal *percentile* rank of the actions in the reference set surpassed by the score of the given action (For example $\theta_I = 0.7$ means that we are interested in samples where the dominant measure ranks an action higher than at least 70% of the actions in the reference set). For the *Normalized* method, as the standardized scores largely falls between −2.5 and 2.5 standard deviations, θ_I represents the minimal number of standard deviations that the score should (positively) deviate from the mean.

To choose an optimal parameters configuration, we used a standard grid search consisting of more than $50K$ unique settings. Table 4 depicts the minimal, maximal and default value for each parameter, w.r.t. both interestingness comparison methods.

Since there is a tradeoff between the predictive performance and the coverage-rate (we explain this in the sequel), in order to choose an optimal configuration we calculated the skyline (also

[5] This can happen when users perform an identical subsequence of actions yet choose a different next action.

[6] In contrast, micro-averaged methods consider all true/false positives, regardless of the class.

	Reference-Based Comparison				Normalized Comparison			
	Accuracy	Macro-Precision	Macro-Recall	Macro-F1	Accuracy	Macro-Precision	Macro-Recall	Macro-F1
RANDOM	0.282	0.281	0.268	0.275	0.252	0.252	0.253	0.252
BestSM	0.397	0.397	0.250	0.306	0.329	0.329	0.250	0.284
I-SVM	0.632	0.636	0.482	0.549	0.655	0.674	0.617	0.643
I-kNN	**0.730**	**0.646**	**0.569**	**0.605**	**0.763**	**0.730**	**0.664**	**0.694**

Table 5: Interestingness Measure Prediction - Baseline Results

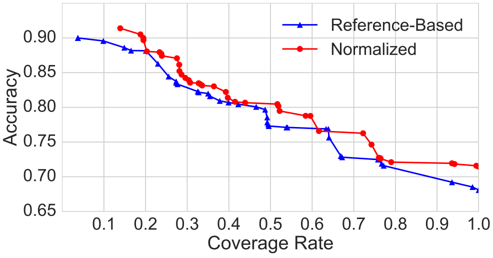


Figure 4: Configurations Skyline

called the Pareto frontier) for every comparison method. This resulted in a set of *dominant* configurations, w.r.t. the coverage and the accuracy/F1 score[7]. Figure 4 depicts the skyline plot for both comparison methods, between the coverage-rate (x-axis) and the accuracy (y-axis).

We chose default configurations from the skyline, as depicted in Table 4. Our default configurations yield accuracy scores of 0.730 and 0.763 (for the Reference-Based and Normalized methods, respectively), with coverage scores of 0.67 and 0.722 (resp.). In principle, any other configuration on the skyline can be chosen to ensure a different coverage-rate or predictive accuracy.

Baselines Comparison. We compared the performance of our predictive model, denoted *I-kNN*, with several other baselines.

(1) RANDOM, a naive baseline that selects a measure out of $\mathcal{I}$ uniformly at random. *(2) Best-SM*, this baseline chooses the *best single measure*, namely it always selects the one measure which is the most prevalent among the training set. This baseline corresponds to the common approach, taken in many analysis assistance tools (e.g. [10, 16, 31]), in which a single interestingness measure is chosen a-priori and used for all cases. Next, we experimented with two predictive models that are adequate to our setting in which the samples (namely, n-context) are compound objects rather than feature vectors: *(3) I-SVM*: a Support Vector Machine (SVM) model with a modified kernel [7] that can take an arbitrary distance matrix rather than using the Euclidean distance between vector-shaped samples. We used it with the dedicated distance metric for IDA sessions [25] described earlier in this section. We employed a standard grid search to tune the model's hyper parameters.

Table 5 lists the predictive evaluation scores for both the Reference-Based and Normalized comparison methods, averaged over all 16 measure combinations in $\mathcal{I}$.

First, we can see that the *Best-SM* baseline outperforms *RANDOM*, yet its accuracy is less than 40%. The latter reestablishes our first hypothesis that no single existing measure can adequately capture users' dynamic interestingness preferences, hence a-priori choosing a single interestingness measure in IDA may often lead to an erroneous outcome. Second, see that the *I-kNN* model outperforms the *I-SVM* demonstrating 14% higher accuracy and 10% higher F1 score. However, recall that in contrast to I-kNN (which uses the default configuration), the SVM base model obtains 100% coverage. Yet, as mentioned above, it is possible to choose a different configuration from the skyline (see Figure 4) to enforce full coverage. In such settings the improvement obtained by the kNN model over the SVM is less significant. Nevertheless, see that both predictive models *I-SVM* and *I-kNN* significantly outperforms *BEST-SM*, which corresponds to existing approaches to interestingness measures. This establishes our third hypothesis *that the right measure can be successfully predicted by examining the analysis n-context.*

In what comes next we examine the effect on the predictive performance (and coverage) of the model's hyper-parameters.

Hyper-parameters Effect. To study the effect of the model's hyper-parameters we repeated the predictive evaluation of the model while varying the values of each parameter and fixing the rest to their default values as depicted in Table 4.

In Figure 5 we present the Accuracy, Macro-F1 and the Coverage-Rate as a function of each of the system parameters (for both comparison method), where the other system parameters get their values from the default configuration (w.r.t. the comparison method used) in Table 4. We next examine the effect on the predictive performance and coverage when varying each of the model's parameters. Last, we present a summary of our findings.

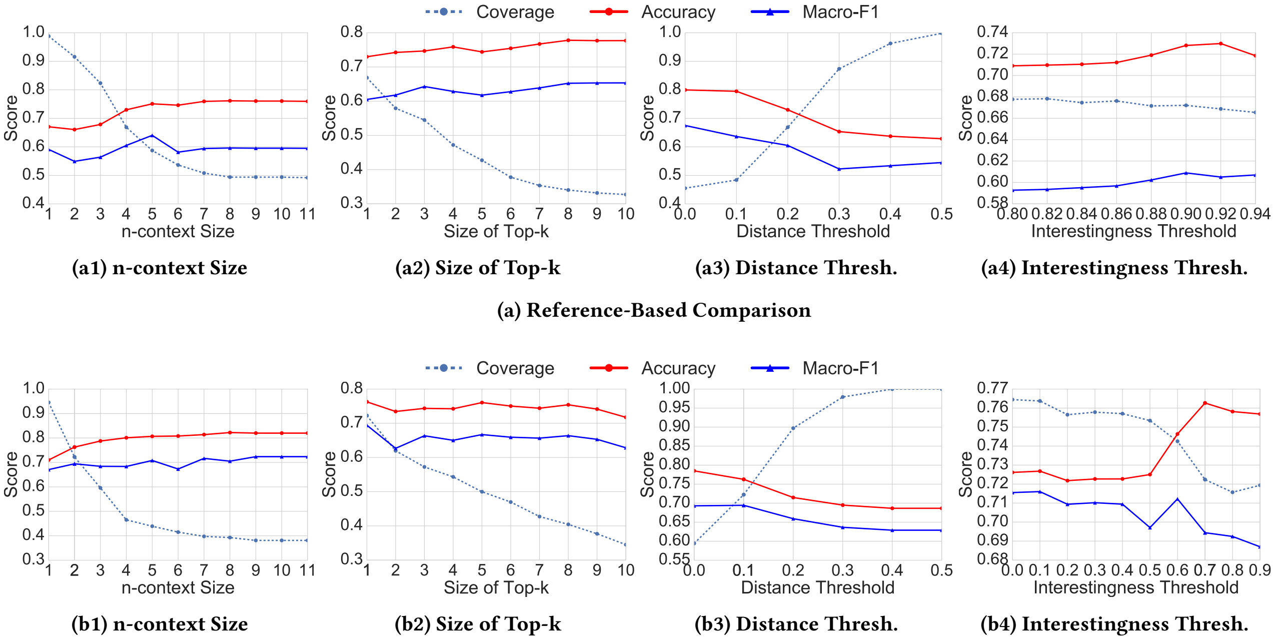
n-context Size. The n-context size, determined when preparing the training set, affects the amount of information the predictive models consider. Figure 5a1 and 5b1 depict the effect of n on the Accuracy, Macro-F1, and the Coverage-Rate of model for the Reference-Based and Normalized settings (resp.). As expected, increasing n (hence increasing the amount of information considered) positively affects the predictive performance. However, observe that the Coverage-Rate decreases. This is expected since when calculating the distance between larger, more compound, n-contexts the scores *increase* thus in more cases the k nearest-neighbors are not "similar enough" and the model does not output a prediction. When choosing n between 2-4, as in our default configurations, we obtained almost optimal predictive performance while retaining coverage of about 70% of the cases.

Size of k. The number of nearest neighbors considered by the model has a milder effect on performance, as can be seen in Figure 5a2 and 5b2 that demonstrate the effect of k on the Reference-Based and Normalized settings (resp.). In the Reference-Based method we can see a small, noticeable increase in performance, however it has a greater effect on the coverage of the model, since finding a larger set of nearest neighbors which are all similar enough to the given n-context is not always possible.

Distance Threshold θ_δ The distance threshold θ_δ is used by the model to enforce that the set of retrieved nearest neighbors are not too distant from the given n-context, hence avoiding erroneous predictions.

As expected, the lower (more tight) the distance threshold, the higher the predictive accuracy, as shown in Figures 5a3 and 5b3

[7] A configuration with x coverage and y accuracy is dominant if there is no other configuration with x' coverage and y' accuracy such that $x' \geq x \wedge y' > y$.



(a1) n-context Size **(a2) Size of Top-k** **(a3) Distance Thresh.** **(a4) Interestingness Thresh.**

(a) Reference-Based Comparison

(b1) n-context Size **(b2) Size of Top-k** **(b3) Distance Thresh.** **(b4) Interestingness Thresh.**

(b) Normalized Comparison

Figure 5: System Parameters Effect

displaying the effect of θ_δ when using the Reference-Based and the Normalized comparison methods. Naturally, the coverage decreases with θ_δ since there is not enough nearest neighbors with high similarity in many cases.

Interestingness Threshold θ_I. Recall that after we apply one of the offline comparison methods, we obtain the relative interestingness of each action. This allows us to filter out cases where the action executed by the user was not considered as interesting w.r.t. any of the measures in $\mathcal{I}$. For both comparison methods this indeed increases the predictive performance, as can be seen in Figures 5a4 and 5b4

Summary of Findings. We conclude this part by pointing out the trade-off between the predictive performance of the model and its coverage. When increasing the size of n-context and the kNN set size we increase the amount of information considered by the model, since a larger set of more comprehensive n-contexts are used as the basis for prediction. This naturally increases the model's predictive performance yet decreases the coverage since there are fewer cases where we can find, e.g., a large number of nearest neighbors highly similar to the given n-context.

A similar effect occurs when increasing the interestingness and similarity thresholds θ_I and θ_δ, which are used to ensure that the model uses only "high-quality" samples. Increasing theses thresholds improves the predictive performance yet decreases the Coverage-Rate, since there are fewer cases where such high-quality samples are relevant.

5 RELATED WORK

There is extensive literature that considers the interestingness of analysis actions on one hand, and interactive data analysis on the other.

Evaluating and comparing interestingness measures. As mentioned in the introduction to this work, interestingness measures are widely used in the field of data analysis, employed in tools e.g. for ranking and sorting association rules [17, 29], data patterns [4], generating useful visualizations [31], exploring OLAP data cubes [16] and many others.

Since dozens of measures were devised in the literature, each capturing different facets of the broad concept, several surveys and comparative studies have been performed to evaluate their usefulness [12, 14, 17, 18, 22, 29]: In these works the authors empirically evaluate many of the measures on real and synthetic datasets, and provide guidelines for choosing the right one for a given task and application domain. For example, [17] presents an empirical evaluation of more than 30 interestingness measures, applied for ranking buying patterns of customers. In [29], the authors examine 21 measures for association rules, concluding that neither one is consistently better than the others. [14] focuses on measures for data summary, empirically showing that the score distributions tend to be highly skewed.

While these notable studies contribute to the understanding of the usefulness of measures for different analysis tasks and scenarios, *they do not address the case of IDA - in which the interestingness criteria may dynamically change as the analysis session progresses.* To our knowledge, our work is the first to experimentally demonstrate this phenomenon and to provide a dynamic interestingness assessment (and selection of the appropriate measure), at each step of an ongoing analysis session.

Subjective facets of interestingness. For brevity, we considered in this work only *objective* facets of interestingness, such as Diversity, Peculiarity and Conciseness. However, several works suggest measures that capture *subjective* facets of interestingness [8, 20], i.e. that use prior information about the user and provide a more personalized interestingness assessment. For example, in [20] the authors devise measures for capturing interestingness facets such as *surprisingness* and *actionability* by considering user prior beliefs encoded as a set of classification rules (e.g. "has_job → loan_approved").

Incorporating such measures in our framework is possible yet requires the model to consider user information in addition to the n-contexts. This is an exciting direction for future research.

Learning Interestingness. Some works also suggest learning-based solutions for interestingness assessment. In [9] the authors present a system for guided data exploration based on active-learning. The system presents users with an initial set of tuples and asks then to annotate each tuple as interesting or not. Harvesting this feedback, the system can improve and personalize the tuples presented to the users. In [21], they present a visualization ranking system which is based on supervised binary classification of visualization into "interesting" or "non-interesting", based on students' annotations as ground truth.

In contrast to our work, these works tackle specific analysis tasks (i.e. filter/select queries and visualizations) hence can not be trivially generalized to the context of IDA which consists of sequences of actions of multiple types, and where (as we had shown in the experiments), interestingness, even for the same action on the same dataset, may vary through the process.

Second, both [9] and [21] require, and rely on users' feedback, which as explained in the introduction has limitations in our context. Different from these works, our solution provides a general-purpose system suitable for various types of analysis actions, and does not relay on particular user annotations. However, harnessing user feedback and learning-to-rank models in our system could be a promising direction for future work.

Mining IDA session-logs for action/query recommendation. Previous work [2, 11, 13, 25] suggests that mining IDA session logs can be used for predicting/recommending the next action in a session. These works utilize a collaborative-filtering approach, intuitively arguing that "if users are posing similar sequences of queries, they are likely interested in the same subpart of the dataset". However, in our previous work [25] we argue that in real life IDA scenarios analysts often examine different datasets from different purposes, hence recommending previous actions from the log to new users is generally impractical. To overcome this, the system described in [25] provides users with high-level "suggestions" that aggregate meaningful action-fragments mined from previous actions in the log. Combining our solution with [25] is an interesting direction for future work, where our solution can assist the system in [25] to better sort the output suggestions and further materialize them to concrete, executable actions.

6 CONCLUSION

This work examines interestingness measures in the context of interactive data analysis (IDA). We show that interestingness in IDA has unique characteristics: it dynamically changes even within a single session, and can not be holistically captured by just one measure. Using a real-life session log we demonstrated these characteristics and evaluated our interestingness predictive model, showing it can successfully select an appropriate interestingness measure for each step in an IDA session. Our model and framework may be employed in existing analysis recommender systems, allowing them to better fit the recommended next actions (e.g., visualizations, queries) to the current interestingness preferences of the user.

As for future work, we previously mentioned several ideas such as using our system to evaluate the effectiveness of analysis sessions. Also, incorporating user feedback and learning-to-rank models in our system is an exciting idea for future research.

ACKNOWLEDGEMENTS

This work has been partially funded by the Israel Innovation Authority, the Israel Science Foundation, Len Blavatnik and the Blavatnik Family foundation, and Intel®AI DevCloud.

REFERENCES

[1] REACT: Ida benchmark dataset. https://github.com/TAU-DB/REACT-IDA-Recommendation-benchmark.

[2] J. Aligon, E. Gallinucci, M. Golfarelli, P. Marcel, and S. Rizzi. A collaborative filtering approach for recommending olap sessions. *Decision Support Systems*, 69:20–30, 2015.

[3] J. Aligon, M. Golfarelli, P. Marcel, S. Rizzi, and E. Turricchia. Similarity measures for olap sessions. *KAIS*, 39, 2014.

[4] M. Boley, M. Mampaey, B. Kang, P. Tokmakov, and S. Wrobel. One click mining: Interactive local pattern discovery through implicit preference and performance learning. In *Proceedings of the ACM SIGKDD Workshop on Interactive Data Exploration and Analytics*, pages 27–35. ACM, 2013.

[5] G. E. Box and D. R. Cox. An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 211–252, 1964.

[6] V. Chandola and V. Kumar. Summarization–compressing data into an informative representation. *Knowledge and Information Systems*, 12(3):355–378, 2007.

[7] Y. Chen, E. K. Garcia, M. R. Gupta, A. Rahimi, and L. Cazzanti. Similarity-based classification: Concepts and algorithms. *Journal of Machine Learning Research*, 10(Mar):747–776, 2009.

[8] T. De Bie. Subjective interestingness in exploratory data mining. In *Advances in Intelligent Data Analysis XII*, pages 19–31. Springer, 2013.

[9] K. Dimitriadou, O. Papaemmanouil, and Y. Diao. Aide: an active learning-based approach for interactive data exploration. *IEEE Transactions on Knowledge and Data Engineering*, 28(11):2842–2856, 2016.

[10] M. Drosou and E. Pitoura. Ymaldb: exploring relational databases via result-driven recommendations. *The VLDB Journal*, 22(6), 2013.

[11] M. Eirinaki, S. Abraham, N. Polyzotis, and N. Shaikh. Querie: Collaborative database exploration. *TKDE*, 2014.

[12] L. Geng and H. J. Hamilton. Interestingness measures for data mining: A survey. *ACM Computing Surveys (CSUR)*, 38(3):9, 2006.

[13] A. Giacometti, P. Marcel, and E. Negre. *Recommending multidimensional queries*. Springer Berlin Heidelberg, 2009.

[14] R. J. Hilderman and H. J. Hamilton. Evaluation of interestingness measures for ranking discovered knowledge. In *PAKDD*. Springer, 2001.

[15] R. J. Hilderman and H. J. Hamilton. *Knowledge discovery and measures of interest*, volume 638. Springer Science & Business Media, 2013.

[16] M. Joglekar, H. Garcia-Molina, and A. Parameswaran. Smart drill-down. *Target*, 6000:0, 2014.

[17] M. Kirchgessner, V. Leroy, S. Amer-Yahia, and S. Mishra. Testing interestingness measures in practice: A large-scale analysis of buying patterns. In *DSAA*, 2016.

[18] T.-D. B. Le and D. Lo. Beyond support and confidence: Exploring interestingness measures for rule-based specification mining. In *SANER*, 2015.

[19] S. Lin and D. E. Brown. An outlier-based data association method for linking criminal incidents. *Decision Support Systems*, 41(3):604–615, 2006.

[20] B. Liu, W. Hsu, L.-F. Mun, and H.-Y. Lee. Finding interesting patterns using user expectations. *IEEE Transactions on Knowledge and Data Engineering*, 11(6):817–832, 1999.

[21] Y. Luo, X. Qin, N. Tang, and G. Li. Deepeye: Towards automatic data visualization. ICDE, 2018.

[22] K. McGarry. A survey of interestingness measures for knowledge discovery. *The knowledge engineering review*, 20(1):39–61, 2005.

[23] T. Milo and A. Somech. React: Context-sensitive recommendations for data analysis. In *SIGMOD*, 2016.

[24] T. Milo and A. Somech. Deep reinforcement-learning framework for exploratory data analysis. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, page 4. ACM, 2018.

[25] T. Milo and A. Somech. Next-step suggestions for modern interactive data analysis platforms. In *KDD*. ACM, 2018.

[26] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

[27] S. Sarawagi. User-adaptive exploration of multidimensional data. In *VLDB*, volume 2000, pages 307–316, 2000.

[28] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of olap data cubes. In *ICDT*, 1998.

[29] P.-N. Tan, V. Kumar, and J. Srivastava. Selecting the right interestingness measure for association patterns. In *SIGKDD*, 2002.

[30] T. C. Urdan. *Statistics in plain English*. Routledge, 2011.

[31] M. Vartak, S. Rahman, S. Madden, A. Parameswaran, and N. Polyzotis. Seedb: efficient data-driven visualization recommendations to support visual analytics. *VLDB*, 2015.

[32] Q. Yao, A. An, and X. Huang. Finding and analyzing database user sessions. In *International Conference on Database Systems for Advanced Applications*, pages 851–862. Springer, 2005.

Hidden Layer Models for Company Representations and Product Recommendations

[Application Paper]

Katsiaryna Mirylenka
IBM Research – Zurich
Switzerland
kmi@zurich.ibm.com

Paolo Scotton
IBM Research – Zurich
Switzerland
psc@zurich.ibm.com

Christoph Miksovic
IBM Research – Zurich
Switzerland
cmi@zurich.ibm.com

Jeff Dillon
IBM Canada
Markham, Canada
jdillon@ca.ibm.com

ABSTRACT

An increasing amount of marketing intelligence data is becoming available today. This includes data that describes information technology (IT) inventories, i.e. IT products purchased by companies. It is advantageous for hardware and software service providers to analyze this data and build recommender systems to propose new products for client companies. Real-time recommendations are usually done based on matrix factorization methods or association rules. In this work we study the applicability of generative models to the recommendation problem. We focus on models that are able to reveal latent connections between companies and deployed IT products and build discriminative features of the IT structure of a company. Additionally generative models of company-product data are of interest for service providers for efficient company comparison, application of similar marketing strategies towards the groups of similar companies. In this work, we first formalize the notion of a company and its IT install base. Then, we estimate various generative models that are able to reveal hidden structures in data. These are mainly topic and language modeling techniques emerging in natural language processing to the task of company-product modeling and sequential models that are widely used for product recommendations. More precisely, the analysis is done using (a) Latent Dirichlet Allocation (LDA) with the products in a company are treated as a set, (b) *n*-gram models or sequential association rules and (c) Recurrent Neural Networks (RNN), when the time of product appearance is taken into account. The techniques are used for a corpus consisting of 860k companies. The results of the study demonstrate that simpler generative models with lower number of parameters, such as LDA, fit company-product data better and are more beneficial for company IT install base modeling both in terms of goodness of fit of the model and recommendation quality.

1 INTRODUCTION

Information technology (IT) install base[1] data is provided by specialized companies that carefully maintain its quality in terms of confidence of IT products' presence and accuracy of timestamps of their appearance. This kind of data has already been exploited for industrial applications and proved to be extremely useful to get market insights in several contexts such as, for example, new market development or white space determination. Data usually refers to companies. For a specific company, different types of information are provided, for example, insights about the internal structure, what the company buys, etc. IT install base data is a specific type of marketing intelligence data that provides, for a set of companies, knowledge about the type of IT equipment they own and how this equipment is distributed across their physical locations.

Install base information is particularly useful when addressing white spaces. If a provider of hardware services tries to acquire new customers, install base information will illuminate the insights about the business potential for a particular product domain. More interestingly, when combined with data about established customers, install base information can be used to identify companies that are similar to existing clients and therefore have a high potential of becoming new customers by acquiring certain sets of products.

In this paper, we concentrate on the problem of modeling company-product relations in order to (1) identify similar companies and (2) build product recommendations. Besides the computational complexity of the similarity search problem due to the large number of companies and product types, another challenge is that a naïve comparison of the individual product types owned by companies turns out to be biased towards product types that are common to a large number of companies. Therefore, after reviewing the related work, we focus on two approaches that consider the hierarchical similarity of objects (companies) based on contextual proximity of their features (product types). This happens to be an assumption in the field of Natural Language Processing (NLP). In our case, the similarity on the lower level is defined via the co-occurrence of the product types in a company. Higher levels correspond to hidden structures in the install base and the representation of a company itself.

To get the best model representations of products and then of companies, we estimate and evaluate various types of generative models. We choose the best model in terms of goodness of fit and predictive power. This model is used to extract product and company representations which are then applied for similarity search queries and marketing recommendations. In particular, we compare the performance achieved by these two modeling techniques: Latent Dirichlet Allocation (LDA) and Recurrent Neural Networks (RNN). The products belonging to a company compose a multilevel structured representation of the IT install base. Such

[1] Install base refers to the IT inventory of a company.

products are used as the basis of company similarity evaluation. The LDA and RNN techniques are beneficial for the IT install base modeling as they are able to reveal hidden hierarchies. RNN is preferred over other neural network architectures as it is able to take into account time-correlations of product time series. Besides, both techniques are among the best performers in the corresponding domains. LDA produces interpretable parameters, while the embeddings obtained by RNN modeling lack the interpretability. Uninterpretable parameters is a significant drawback for business applications, which can be tolerated if these models would perform much better.

The main contributions of this work are the following:

(1) We formalize the problem of modeling the IT install base of companies for various types of input data, such that the state-of-the-art unsupervised techniques from NLP domain can be applied. First, we formalize company-product data treating products in a company as a set and use corresponding non-sequential models for this case. Second, we treat the product data as sequences according to the time of their appearance. The models are then compared using their data fitting quality.

(2) The applicability of topic modeling via LDA and language modeling via RNN is assessed for our data. We demonstrate that LDA with 2, 3 and 4 latent topics fits the data best and, additionally, provides the most discriminative company features for the task of company clustering.

(3) We assess the practical applicability of the LDA and RNN models to an industrial product recommender and compare them with sequential association-rule and matrix factorization approaches.

(4) After solving various data integration challenges, we apply the output of the best performing model, enriched with internal company-product data, to a recommendation tool that searches for similar companies and makes recommendations.

2 PRELIMINARIES AND PROBLEM FORMALIZATION

The goal of this paper is to develop a recommendation system that also allows companies to be compared based on their install base. This comparison will be used to generate sets of similar companies and will allow possible business developments to be identified in real time. As the system in development is to be used by offering managers, the interpretability of the results is a crucial requirement to be able to justify the outcome. In consequence, models with interpretable parameters bring an advantage in our settings.

As a source of marketing intelligence data, we use information provided by HG Data Company, Inc. [12]. HG Data Company is building and maintaining a comprehensive database with competitive intelligence about deployed technologies. Essentially, for each company assessed, this database provides the following information: the type of IT products available at each site of the company without specifying the quantities and product model details, some indication about the confidence of the information provided, and dates of the first as well as the most recent successful confirmation of product presence.

Product descriptions are organized in a hierarchical fashion. This hierarchy contains four levels. At the top, we find the *vendors*. For each vendor, there is a list of *category parents* giving a high-level grouping of the product types, for example, “Data Center Solution” or “Hardware (Basic)”. Each *category parent* contains a list of *categories*, which are finer-grain groups. Examples of categories are “Printers” or “Midrange Computers”. Finally, each category contains the *product types* available for the vendor considered.

Our aim is to develop a sales recommendation application. Given a customer, potential customers with similar IT install base are provided as input to our solution, which then combines this information with internal data. To align the product description between our proprietary data and the result of the company similarity evaluation, we have focused on the category layer. Unfortunately, the product types do not contain a certain product details, thus, do not allow us to link the products with the lowest level of product descriptions in our proprietary data.Therefore for each company we consider the product categories[2] associated with the company, independently of the vendor. In our version of the HG Data Company database, there are 91 distinct categories. Out of those categories, we decided, because of the nature of our application, to restrict our study to hardware and low-level hardware management software categories, thus, using 38 distinct categories.

To link data from the HG Data Company database to our internal data, we solved integration and cleaning issues. In the HG Data Company database, companies are identified by their D-U-N-S® numbers. This number is a unique 9-digit identifier, assigned by Dun & Bradstreet, Inc. [8] to each business location. Therefore, each company entity, such as branches, subsidiaries and headquarters, has an individual D-U-N-S® number, and the set of all D-U-N-S® numbers associated with a company is organized hierarchically.

After we had chosen the product categories or company attributes and aggregated the subsidiaries of a company, we created our corpus for model training which is a binary company-product matrix. As we do not have information about product quantities, we consider only binary values in the company-product matrix, where 1 means that the product belongs to the install base of the company and 0 means that it does not.

More formally, let us consider a set of N companies represented in the HG Data Company database $C = \{c_0, \ldots, c_{N-1}\}$. Each company c_i has a given set of products A_i in its install base belonging to k categories, which can also be called attributes. This set of attributes is included in the set of all possible attributes $A = \{a_0, \ldots, a_{M-1}\}$ containing M elements[3]. That is

$$\forall c_i \in C \; ; \quad c_i \longmapsto A_i = \{a_{i_0}, \ldots, a_{i_{k-1}}\} \subset A. \tag{1}$$

The attributes might be sorted by the time of their appearance in the IT install base of a company and treated as data series. We denote the sorted version of A_i as A_i^S. The information about the attributes or products from A_i can be re-written using vectors $\mathscr{A}_i$ instead of sets of products:

$$\forall c_i \in C \; ; \quad c_i \longmapsto \mathscr{A}_i \; , \quad \dim(\mathscr{A}_i) = M \; , \tag{2}$$

$$\mathscr{A}_i = \left[\mathbb{1}_{a_0 \in A_i}, \ldots, \mathbb{1}_{a_{M-1} \in A_i} \right] . \tag{3}$$

A naïve approach to compare companies is to calculate the distance between their initial attributes $\mathscr{A}$ or A^S. In Section 3, we discuss why such an approach leads to results that are not sufficiently meaningful. The initial attribute space does not represent companies in a discriminative manner because the distance

[2] In the remainder of the paper we use the terms products and product categories interchangeably, meaning product categories.

[3] In our application $M = 38$.

between companies is affected too strongly by the most popular attributes. To overcome this problem, we introduce a new space of company features that better represents the IT install base:

$$\forall c_i \in C\;;\;\; c_i \longmapsto \mathscr{B}_i \in \mathbb{R}^L\;,\;\; L < M. \tag{4}$$

Considering the new company features $\mathscr{B}$, it is possible to express the distance between two companies as a classical distance between two vectors:

$$\forall c_i\;,\;\; c_j \in C\;;\;\; dist(c_i, c_j) = d(\mathscr{B}_i, \mathscr{B}_j), \tag{5}$$

where $d(.,.)$ is any vector distance, e.g., euclidean or cosine distance.

One of the goals of this paper is to automatically discover the most representative features of a company $\mathscr{B}$, based on initial company attributes A^S or attribute vectors $\mathscr{A}$. The features should be representative in terms of goodness of fit of the generative model of company-product data and in terms of quality of similarity clusters of companies. The methods of feature learning for companies are discussed in Section 4. Learned company and product features are, then, used for product recommendations.

3 RELATED WORK

In this section, we consider the applicability of existing methods for building discriminative company representations, namely, the feature vectors $\mathscr{B}_i$ for each company c_i. $\mathscr{B}_i$ are built using product vectors of a company, A_i, or a sequence of products per company, A_i^S. This representations should capture the hidden structures in the company-product data.

3.1 Co-Clustering

Co-clustering or bi-clustering is a technique used for building two-dimensional groups of objects that are represented by a matrix. The matrix is clustered along the rows and the columns. The principle of co-clustering was first introduced by Hartigan in 1972 [10]. Since then, several algorithms have been proposed mainly aimed at product recommendations.

The PaCo algorithm [27] and the OCuLaR algorithm for co-clustering with overlapping [11] are most closely related to our problem. The main issue with these approaches is that they are built on initial company representations $\mathscr{A}_i$, which may be not the best features to distinguish IT install bases of companies. When we applied the PaCo algorithm as well as spectral co-clustering to a small sample of around 500 companies that belong to a healthcare industry, we could not generate meaningful co-clusters. The only co-cluster generated contained overall popular products. Our first attempt of using LDA for this small subset of companies and products [18] provided us with much better results and inspired us to continue our search of company features in this direction. We believe that modified product features could improve the quality of co-clusters. As we also show in Section 5, raw initial company-product representations $\mathscr{A}_i$ neither describe the generative model of our data well nor perform well for the task of company clustering. In addition, given the large number of companies to analyze (on the order of a million), co-clustering results are difficult to interpret because the intuitive visual way to consume co-cluster results is not possible in this case.

3.2 Pattern mining

Another possibility is to explore product install base modeling using the approaches of the frequent pattern search from the time series domain. One family of such approaches is Association Rule mining [2], which is partially time agnostic. It was demonstrated in [6], [7] that taking into account the sequential nature of data is beneficial for time series tasks, such as similarity matching and nearest neighbor search. This inspired another line of techniques that are based on estimating Markov Chain models via real-time algorithms for ‘conditional heavy hitters‘ discovery [20], [17]. However, the mined patterns are not able to represent the hidden structures of the IT install base of companies, though we use them to compare more advanced methods with.

3.3 Applicability of NLP Concepts

In the past decade, much attention in the literature was given to modeling techniques related to NLP. In this subsection, we discuss their applicability to the problem of modeling install bases of companies. One of the key tasks in NLP is language modeling. The goal is to learn representations for hierarchies of concepts, starting from words, phrases, and sentences, which are then organized into more sophisticated concepts, such as documents and topics. In recent years, a lot of research has been devoted to the advancement and improvement of topic modeling and language processing methods, including, among others, LDA and Deep Neural Network (DNN) approaches. We assume that the generative model of our company install bases is similar to the NLP models. The product-company world consists of the following layers: a layer of companies, a layer of product categories and a hierarchy of latent structures inside the install base. Given this assumption, these company layers can be mapped to NLP concepts for application in the LDA and DNN methods. Considering NLP terminology, we associate companies with documents and products with words. All companies that we consider in our analysis form the corpus of company documents. We further assume that products or product embeddings can be grouped into latent topics, which then construct specific and discriminative features $\mathscr{B}_i$ for each company c_i, $0 = 1, 2, ..., N - 1$.

3.4 Deep Neural Networks and Product Embeddings

Currently DNNs are the core of the-state-of-the-art techniques for the tasks related to NLP. Mikolov *et al.* [16] [15] use a simplified architecture of neural networks that allows them to use very large training datasets and to build accurate word embeddings in Euclidean space of high dimensionality very efficiently. The word embeddings can afterwards be used directly without any transformation or aggregation as features for clustering. They also can be aggregated to represent a document as a vector in a smaller space using, for example, the Fisher Kernel Framework (probabilistic modeling of the corpus of documents using a mixture of Gaussians [14]) similarly as described in the work [5].

The larger the training set (hundreds of millions) and the larger the dimensionality of the word embedding, the better is the representation, although after a certain point the improvement is no longer significant. This is partially due to the large size of the vocabulary that typically needs to be learned, namely, 600K. As in our case the number of products is much smaller, there is a chance that we will learn good embeddings from tens of thousands of companies.

State-of-the-art performance on language modeling task is achieved by RNN family of models and specifically by RNN with Long Short-Term Memory (LSTM) units with the dropout regularization method [29]. This technique applies a distinct view on the information flow, using information not about independent instances but taking into account the sequential nature of data. Therefore, recurrent networks are sensitive to the past inputs and can adapt to them. Other RNN realization, such as Gated Recurrent Unit (GRU) [4] which is a simpler version of LSTMs, has recently gathered popularity, but study [9] empirically demonstrates the the performance of GRUs and other RNN architectures can be better for some datasets, but do not outperform LSTM in general. We will apply LSTM architecture to our company-product modeling using a time series input, A_i, and analyze its performance both in terms of goodness of fit and as a recommender of future IT products of companies.

In work [19], we have demonstrated that LSTMs are applicable to the task of modeling company-product time series. We used RNNs with Long Short-Term Memory (LSTM) units with the dropout regularization method [29]. Prior to applying RNNs in our work [19] we demonstrated that company-product time series are of clear sequential nature. This was done using statistical hypothesis testing and was based on the fact that the frequency of the i.i.d. time series observations has binomial distribution. In this work, we compare accuracy of the RNN modeling, where product time series are taken into account, with the performance of LDA, where products in a company are treated independently, without their timestamps. We also assess both methods for the recommendation task, comparing their results with the association rule-based recommender.

3.5 Latent Dirichlet Allocation

Blei *et al.* [3] introduce LDA, which is a generative probabilistic model for collections of discrete data, such as corpora of documents (or company-product vectors in our case) that is also used for collaborative filtering. Each item, such as a document or a company, is modeled as a finite mixture of an underlying set of topics or hidden groups. The topic probabilities for an item provide an explicit representation of a document. In parallel, word embeddings are also trained. The embeddings represent relatedness of words in the space of document topics.

The number of latent topics is a user-defined parameter. It can be chosen using measures of the goodness of fit of the LDA model, such as, a total log-loss for a testing set of documents or as the average perplexity of how well each single word is modeled. In our case of company-product modeling, LDA learns both product embeddings and company representations $\mathscr{B}_i$ in the vector space that is the size of the number of latent topics. The main advantage of LDA over RNN and other topic modeling techniques such as Latent Semantic Indexing [13] is that LDA-learned features are easy to interpret. This fact is important for adopting those techniques in marketing environment.

4 MODELING APPROACH

In the current setting, we can model and learn the semantic information about companies in terms of their IT install base, which is based on the fact that similar products and, then, similar companies should be close in the L-dimensional space, where $L < M$. In this case, the recommendations are extracted using the notion of similarity between the companies. Recommendations are based on the similarity calculated given the dataset from HG Data Company. The gaps in possible product offerings are extracted from our internal databases for similar companies. The strength of the recommendation is in this case measured via the strength of the company similarity.

We compare two types of unsupervised models: non-sequential modeling (LDA-based), when products are considered independently ($\mathscr{A}$ company features are used as input), and sequential modeling (LSTM-based), when we take into account the order of product appearance in the HG Data Company database via A^S input. Both modeling techniques capture the hidden structures in the company-product data and produce features (representations) for products and companies. We assess the quality of the following company-product representations:

(1) Naïve representation: binary or Term Frequency-Inverse Document Frequency (TF-IDF) [22] vector of products. In our case, TF-IDF can be also reformulated as product frequency-inverse company frequency.
(2) RNN-based representation: embedding (vector) that shows the position of a company in an L-dimensional vector space. The position depends on the contextual similarity of products of a company.
(3) LDA-based representation: vector of real numbers that shows the probability that a company belongs to an LDA topic.

The modeling methods are evaluated using the measures of goodness of fit of a model, and, additionally, the representations are evaluated for the company clustering task. When the proper representation is chosen, we can find the top-k similar companies based on HG Data Company data. The models are also evaluated for the recommendation task.

4.1 Model adaptation and parameter estimation

We train LDA both on initial binary company-product representations[4] and TF-IDF representations. The type of data representation is considered as one of the parameters for LDA training. Although LDA intrinsically models data to give more weight to the most representative features, we verify whether the model improves if TF-IDF representation is given as an input. Another crucial parameter of LDA modeling is the number of latent topics; this number is chosen using goodness of fit measures.

For RNN, we used various architectures as modeling parameters. More details about LSTM modeling can be found in [19]. We select the parameters of LDA and LSTM by minimizing the perplexity level of a model. The average perplexity per product is calculated on a test set using $\mathscr{A}_i$ or A_i^S company representation, with the total number of products being n. Perplexity[5] shows how well the probability distribution defined by a model (for example, LDA or LSTM) predicts testing data and is calculated as follows:

$$Perplexity = \exp^{-\frac{1}{n}\sum_{i=1}^{n}\ln P(a_i)},$$

where $P(\cdot)$ is the probability distribution induced by a model. The lower the perplexity, the better the model. The best features of a company $\mathscr{B}$ are computed using the models with the lowest perplexity. Instead of the original binary vectors $\mathscr{A}$ or binary sets of products A^S, a company is represented via the vector of latent LDA topics or company embeddings trained on RNN.

[4] Initial binary representation can also be called 'Bag Of Words' (BOW) representation in terms of NLP theory.

[5] We use the terms perplexity and average perplexity per product interchangeably.

4.2 Validation of Company Representations using Clustering

To see how extracted features perform in comparison with initial binary features, or initial TF-IDF features, we assess the quality of learned representations for a clustering task.

As measure of clustering quality, we use silhouette scores[6]. The silhouette score is calculated as the ratio of intraclass and interclass distances. The higher the score, the better the clusters are separated from one another. We choose an appropriate model depending on the silhouette score value for the desired number of clusters. We expect that the model with the highest silhouette scores is also the best according to the perplexity results, as both validation measures favor the most descriptive and representative features.

4.3 Recommendation Capabilities of Generative Models

In addition to the perplexity evaluation, we also check the recommendation capabilities of LDA, LSTMs and n-gram based models. For the marketing recommendation scenario, it is essential to provide an outlook over a relatively long period of time; typically the span of interest r ranges from 6 to 24 months. To evaluate this capability, we assess the model recommender for a sliding window W_r that covers r months. This time window slides with a granularity of two month in order to accumulate significant changes in the install base of a company. At each iteration, the recommender provides a series of observations *i.e.* the probability of a given product appearing within a particular window W_r. In order to access the statistically significant accuracy, we gather the accuracy information for a number of sliding windows. The cardinality of the set of accuracy observations is equal to the number of sliding windows and is denoted as l. All the previous information that happened before the start of a sliding window is used for model training.

To obtain a recommendation for a particular product, we assess the conditional probability of seeing that product appearing given the time series of products that a company acquired so far. The probability is obtained as the output of a model. If for a product p_i the probability of the generative model M, $Pr(p_i|M, p_{i-1}, p_{i-2}, ...)$ exceeds a threshold ϕ we assume that the product p_i should be recommended to a given company. Products $p_{i-1}, p_{i-2}, ...$ represent previous products acquired by that company. As the optimal probability threshold ϕ is not known in advance we treat it as a parameter of the validation of the model recommender. For each ϕ and for each company time series we estimate the average precision and recall measures for the company time series and all the sliding windows l. Recall shows how many of the true products that company buys in the future are retrieved by the recommender. Precision represents the ratio of the retrieved products that are in the true future product set.

5 EXPERIMENTAL EVALUATION

The experiments are done for 860k aggregated companies and 38 product categories[7]. Company aggregation is performed using domestic D-U-N-S® numbers, that is, all company sites in one country are aggregated. Products are consequently aggregated into a set containing all products available in all sites of a company. The companies belong to 83 industries, such as “Health Services”, “Agricultural Services”, etc., which are encoded with the SIC2 codes.[8]

First, we estimate the perplexity of initial company representations $\mathscr{A}$. This is equivalent to the perplexity of the unigram ‘bag of words’ model. The perplexity is equal to 19.5. The perplexities of bi- and tri-gram models are also reported in [19]. Their value is not lower than 15.5, which will be the evaluation baseline.

LSTM. For LSTM, we used A^S company representations, where products are sorted according to the date of their first appearance in a company. LSTM is applicable for sequential data. The sequential nature of our data was checked in work [19], where we demonstrated that 69% of the bigrams and 43% of the trigrams have frequencies that are statistically significantly higher than in the case of independent identically distributed products. This demonstrates that the time dependencies among the products are indeed strong. The hypothesis testing was based on the binomial distribution of frequencies of n-grams.

We used 12 different architectures of LSTM model by varying the number of hidden layers ($N_{layers} = \{1, 2, 3\}$) and the number of nodes in the layers ($N_{nodes} = \{10, 100, 200, 300\}$). We used 70% of the initial corpus for training, 10% for parameter validation and 20% for model testing. We used the LSTM model implementation of the ‘tensorflow’ package [1]. Training was done for 14 epochs over the training data. The resulting perplexity values for each RNN architecture are given in Figure 1.

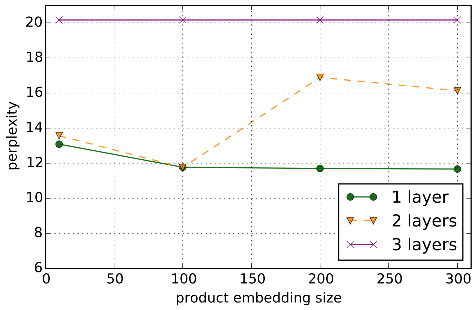


Figure 1: LSTM average perplexity per product for test data.

As can be seen from the results, the lowest (best) perplexity achieved by LSTM model is 11.6 for the test set, which corresponds to 1 hidden layer and 200 nodes per layer. The number of nodes per layer corresponds to the product embedding size. LSTM also learned meaningful representations of IT products, that are illustrated and discussed in [19].

LDA. In the case of LDA, we need to set the number of latent topics. Altough LDA takes into account the representativeness of words (products in our case), we consider both raw binary representations and TF-IDF representations as input for the model. The division of the corpus into training and test datasets is done in the same way as for RNN modeling. The LDA implementation used is from the *gensim* package [23]. The perplexity curves of LDA for both inputs are shown in Figure 2.

[6] For the experiments, we use the silhouette score implementation available in Python programming language (Python Software Foundation, https://www.python.org) package *sklearn*[21]

[7] The full list of product categories is available at: http://www.hgdata.com/Technologies-We-Track.

[8] Standard Industrial Classification (SIC) is a taxonomy established by the US Government to classify industries. The full list of SIC encoded industries is available at: http://siccode.com/.

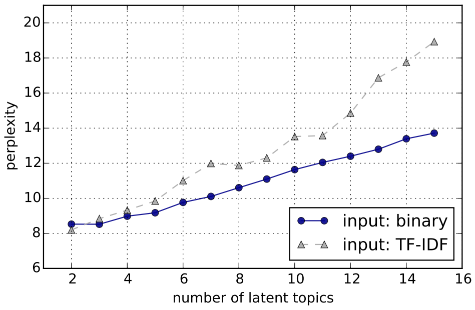


Figure 2: Average perplexity plots measured on test data for LDA models.

It appears that the perplexity of raw binary inputs is better than that of TF-IDF pre-processed input. The leads to the conclusion that LDA indeed is able to assign higher weights to the most representative products and that therefore, no additional pre-processing is needed. Moreover, lower numbers of hidden topics, namely. 2, 3 and 4 lead to lower perplexity values, which vary from 8.5 to 8.9.

The minimum achieved perplexities for each method with the ranking placing the best methods first are shown in Table 1.

Table 1: Minimum perplexities achieved by each method varying the parameter settings.

	Method Name	Min. Perplexity
1	LDA	8.5
2	LSTM	11.6
3	*N*-grams	15.5
4	Unigram 'bag of words'	19.5

Lessons learned. We conclude that, for our company-product data, LDA models perform better than LSTM models. Note that LDA models do not take into account the time component of the company-product space, whereas RNN is built over time ordered sequences of products. We assume that more sequential training data might be needed to build more accurate LSTM models as they have more parameters than LDA models. Indeed, the number of parameters to estimate in LDA models is equal to $n_t + n_t * M$ [3], where n_t is the number of latent topics in the LDA model and M is the vocabulary size of products. In the case of four latent topics, we have 156 parameters to estimate. The number of parameters to estimate for LSTM is dominated by $n_c * (4 * n_c + n_o)$ factor [24], where n_c is the total number of memory cells and n_o is the number of output units. In the case of one the best performing LSTM settings in our deployment with 100, the number of parameters is lower bounded by $100(4 * 100 + 100) = 50000$.

Another hypothesis is that the distribution and properties of hidden structures in company-product data correspond better to LDA modeling assumptions. The fact that LSTM with only one hidden layer fits the data the best out of other LSTM architectures supports this hypothesis.

5.1 Recommendation accuracy

To assess the recommendation accuracy we follow the methodology described in Section 4.3. The time span of the product time series in our deployment range from 1990 till the end of January, 2016. We use a trained model (LDA, LSTM or *n*-gram based) to predict products within a sliding window W_r of 12 month starting from January,1 2013, thus, $r = 12$. The window slides by two month. This way we obtain 13 accuracy observations. The first recommendation window starts on January 1, 2013 and finishes by January 1, 2014 and the last recommendation window starts on January 1, 2015 and finished on January 1, 2016.

The precision and the recall of the LDA3 is compared to the LSTM-based recommender and *n*-gram recommender based on exact Conditional Heavy Hitters [17], that is another data series technique capable of capturing time correlations in the data and predicting future states of the system. The exact Conditional Heavy Hitters are also exact time-dependent association rules. Based on the experiments to validate temporal correlations in product time series described earlier, the depth of the context for Conditional Heavy Hitters (CHH) is chosen to be 2. Thus, we study the dependencies on the previous products up to the second order.

The plot with the average Precision and Recall values, for different values ϕ of conditional probabilities that are used as a recommendation threshold is shown in Figure 3.

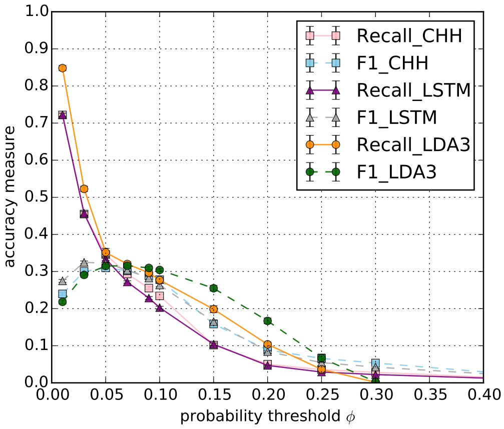


Figure 3: Recall and F1-score with the corresponding confidence intervals for the recommenders based on LDA, LSTMs and Conditional Heavy Hitters (CHHs).

The average numbers of retrieved and correctly retrieved products by each method with the confidence intervals are shown in Figure 4.

We can infer (Figure 4) that ϕ should be smaller or equal to 0.2, as for higher values the numbers of products retrieved by the methods are too low. The accuracy results demonstrate that the recall of a recommender based on LDA model is consistently higher than the recall of LSTM and CHH-based recommenders for the appropriate values of ϕ. The F1-score is also higher for large range of ϕ. For the values of $\phi \geq 0.25$ the recall values are not statistically significantly different as their confidence intervals intersect. Also we note that beyond the probability threshold $\phi = 0.5$ LDA and CHH did not produce any recommendations, thus, precision values are not defined for this points and recall values are equal to zero.

The plots on Figures 3 and 4 demonstrate also that the recall LSTM and CHH is similar as they retrieve similar number of true future products of a company, but CHH-based recommender

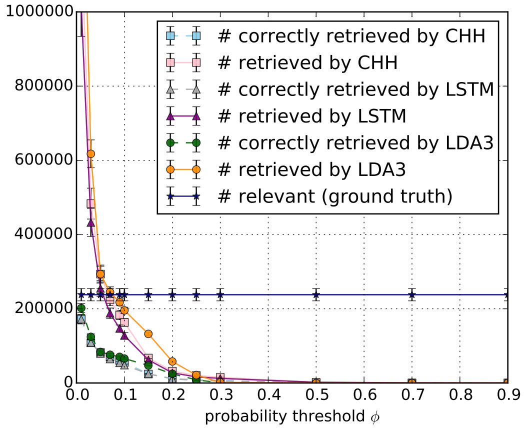


Figure 4: The average number of retrieved, correctly retrieved and relevant products for LDA, LSTM and CHH with the confidence intervals.

tends to produce more false positives, i.e. it recommends more products that should have not been recommended. This causes significant differences in precision.

The task of product recommendations is very hard as it is quite difficult to capture all the underlying processes that influence the choice of future models. This is witnessed by the fact that so far, the best models we tried are only able to reach the values of precision and recall around 0.25 - 0.43 for $\phi \in [0.05, 0.15]$. This means that they are able to capture a true generative model to some extend. The random generator that produced a product recommendation with a uniform probability = $1/38 \approx 0.026$ retrieved all the products for the thresholds ϕ <= 0.026 and on average close to zero of the correct products for higher thresholds. The differences between the accuracy measures for LDA and LSTM, LDA and CHH-based methods are statistically significant for most of ϕ values as the corresponding 95% confidence intervals do not intersect. We have also assessed the accuracies LDA-based recommender with two and four latent topics, their performance has been very similar to the results of LDA with the three latent topics which has been described above. As a future work we will study the influence of the sliding window size on the recommendation accuracy.

5.2 Comparison with Matrix Factorization

We also compared the hidden layer methods with one of the best-performing matrix factorization techniques, Bayesian Probabilistic Matrix Factorization (BPMF), which was introduced in work [25] and implemented in [28]. As the system requires rankings as an input, we use a ranking transformation of the training and test data described in the previous section. This means that if a company has product x, its ranking is equal to 1, otherwise, its ranking is equal to 0.

The BPMF recommendation results that we obtained for the product recommendations in our deployment were similar to the initial results for co-clustering that we reported in Section 3.1. In the majority of the cases, recommendations produced by BPMF for a given company include all possible products. The distribution of the BPMF recommendation scores can be seen in Figure 5. This is due to the fact that BPMF, and matrix factorization methods in general, was developed for sparse and imbalanced datasets. The data in our deployment is relatively dense, and cannot be reasonably approximated by a low-rank matrix, which is the basis of BPMF.

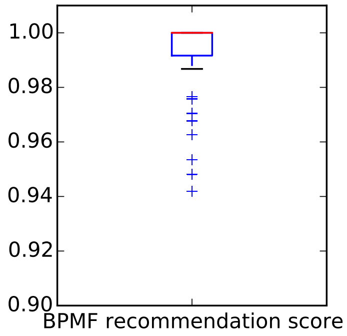


Figure 5: Boxplot of BPMF recommendation score values.

The precision, recall and F1-score values depending on varying BPMF recommendation score values (in the interval [0.9, 1.0]) are shown in Figure 6.

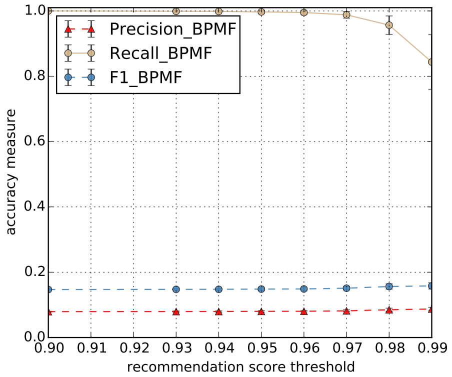


Figure 6: Accuracy values of Bayesian Probabilistic Matrix Factorization method.

All the scores for the thresholds of recommendation score that are lower than 0.94 are the same as for the values 0.91, 0.93 and 0.94. This means that almost for all of the threshold values the full set of available products is recommended regardless the previous products purchased by a company. Thus, BPFM does not produce meaningful recommendation results and does not provide us with effective features for company-product modeling. In contrast, additional feature search for products, as done in LDA and LSTM, leads to high-quality recommendations.

5.3 Company clustering.

As LDA is the best-performing model based on the analysis of perplexity and recommendation accuracy, we will now study the suitability of LDA-learned features for clustering. For this purpose, we build silhouette curves for the features obtained with the best-performing LDA configurations, that is, with the number of topics equal to 2, 3 and 4. We then compare these results with silhouette curves that are built on i) raw binary company-product representations, ii) raw TF-IDF company representations and iii)

LDA representations with TF-IDF input for 2 and 4 hidden topics. The results are shown in Figure 7.

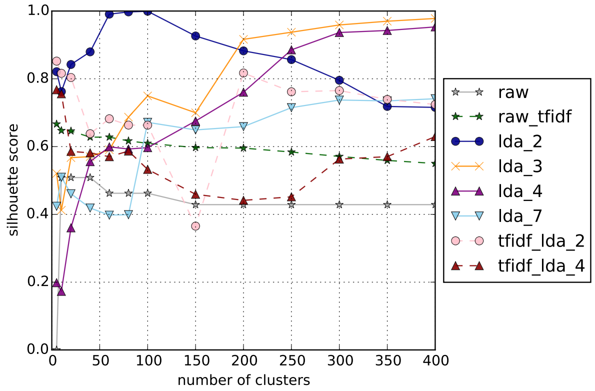


Figure 7: Silhouette curves.

Note that the higher silhouette score, the better the clusters of companies are separated. A higher score means that the distances between companies within one cluster is much lower than the distances between companies in different clusters. In Figure 7, we can see that the initial binary representations of companies are not very discriminative (blue line with stars) as the silhouette score is the lowest for almost all number of clusters. Clustering on the initial representation with TF-IDF transformation performs better then clustering on the initial binary representation, as the silhouette curve is higher and around 0.6 for a varying number of clusters. Clustering on LDA-generated representations with TF-IDF input (tfidf_lda_2 and tfidf_lda_4) performs better than clustering on raw TF-IDF, especially, when two latent topics are used. Company representation associated with the best silhouette curves are generated by LDA with raw binary inputs for the number of latent topics equal to 2, 3 and 4. This result is in accordance with the perplexity outcomes. This means that LDA with these numbers of latent topics represents the install base of the companies the best.

We notice that silhouette scores for lower numbers of clusters from 5 to 200 are higher for LDA representations with lower numbers of topics, i.e., 2 topics, whereas higher numbers of topics (3 or 4) discriminate larger numbers of clusters better.

The t-SNE [26] 2D projections for the product embeddings based on LDA3 and LDA4 are shown in Figures 8, 9. The original names of the product categories are shortened for better visualization, e.g., 'SW' and 'OS' stand for software and operating systems.

It appears that the main products that construct a topic produce clusters of products. It is also interesting to see that most of the hardware products are close to each other for both the LDA3 and the LDA4 representation. These are 'server_HW', 'storage_HW', 'HW_other'. Similarly, software products tend to appear together, for example, 'commerce', 'media', 'collaboration' 'product_lifecycle', 'electronics PCs SW' and 'retail'. Thus also the semantic proximity of the products is captured by LDA models.

Lessons learned. The good quality of company clusters and the meaningful representation of products discussed above mean that the LDA method is able to automatically infer representative features both for companies and products in our deployment.

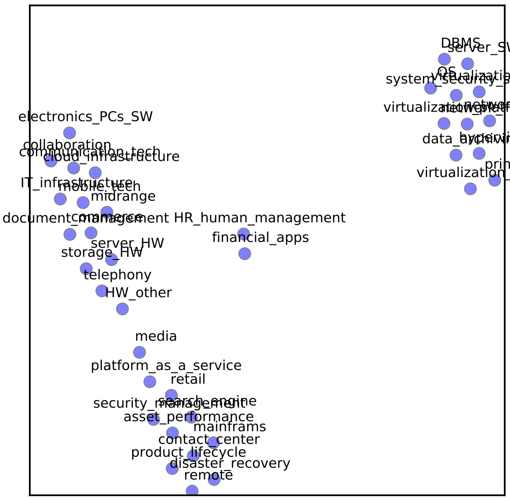


Figure 8: LDA3 product embeddings.

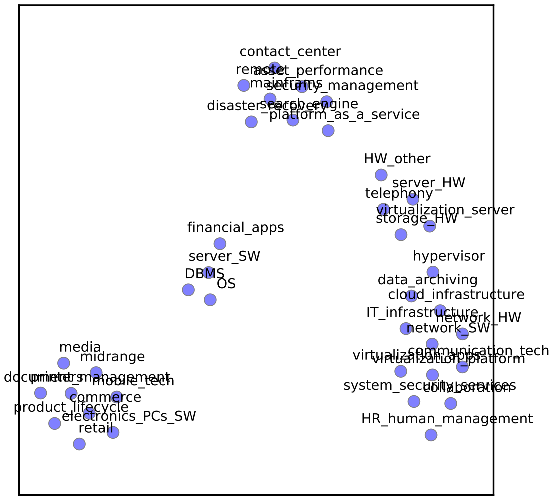


Figure 9: LDA4 product embeddings.

6 SALES APPLICATION

We have deployed LDA-based company representations in our recommendation tool. The company similarity search is based on LDA company representations with the HG Data Company dataset as input. Recommendations are built using our internal datasets. The tool searches for top-*k* similar companies that are calculated using their LDA representations based on HG input. As LDA training is not done in a streaming fashion, it is done offline and can be retrained on demand or when the concept shift is taken place. In addition to the global similarity search, the tool also provides the user with filtering capabilities based on industry, location, number of employees and revenue.

This tool is currently used for an internal recommendation system.

7 CONCLUSION AND DISCUSSIONS

In this work, we assessed several data modeling techniques for product-company modeling, company similarity matching and recommendation. Companies have been considered to be similar based on the closeness of the structure of their IT install base. Assuming intrinsic hierarchies between products, companies and possibly latent install base structures, we have compared several techniques from the NLP domain capable of learning this kind of hidden hierarchical structures. These are unsupervised modeling techniques, namely, Latent Dirichlet Allocation and Recurrent Neural Networks. Having evaluated different model architectures, we demonstrated that LDA with 2, 3 and 4 latent topics fits our data best. We applied the company features learned by LDA to determine the top-k similar companies, assessed the recommendation capabilities of the methods and deployed the best performers in a recommendation tool.

The results show that though there is clear sequential nature in the data, still the techniques that does not take time into account perform the best. The reason of this may be due to the higher number of parameters to learn in sequential techniques and the fact that our corpus is not enough to train all the parameters.

As future work, we will gather additional internal data about the IT structure of companies with proper timestamps and assess other deep neural network architectures starting from lower levels of product descriptions. We believe that, because of their hierarchical nature, deep neural networks should be able to discover hidden structures in IT install bases of companies if sufficient training data is provided.

8 ACKNOWLEDGMENTS

We would like to thank Viktor Kuropiatnyk for helping us make sense of D-U-N-S® data; Michel Speiser and Daniel Bauer for their algorithm of company name matching, which we used for record linkage. Katsiaryna Mirylenka also thanks Daniil Mirylenka for the fruitful discussions during the idea crystallization process.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.

[2] Charu C. Aggarwal. 2015. *Data Mining - The Textbook.* Springer.

[3] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *the Journal of Machine Learning Research* 3 (2003), 993–1022.

[4] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *CoRR* abs/1412.3555 (2014).

[5] Stéphane Clinchant and Florent Perronnin. 2013. Aggregating continuous word embeddings for information retrieval. *ACL 2013* (2013), 100.

[6] Michele Dallachiesa, Besmira Nushi, Katsiaryna Mirylenka, and Themis Palpanas. 2011. Similarity matching for uncertain time series: Analytical and experimental comparison. In *Proceedings of the 2nd ACM SIGSPATIAL QUeST.* 8–15.

[7] Michele Dallachiesa, Besmira Nushi, Katsiaryna Mirylenka, and Themis Palpanas. 2012. Uncertain Time-Series Similarity: Return to the Basics. *PVLDB* 5, 11 (2012), 1662–1673.

[8] Dun & Brddstreet Inc 2017. Dun & Bradstreet Company. http://www.dnb.com. (2017).

[9] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. 2016. LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems* (2016).

[10] J. A. Hartigan. 1972. Direct Clustering of a Data Matrix. *J. Amer. Statist. Assoc.* 67, 337 (1972), 123 – 129.

[11] R. Heckel and M. Vlachos. 2016. Interpretable recommendations via overlapping co-clusters. *ArXiv e-prints* (April 2016).

[12] HG Data Company 2017. HG Data Company. http://www.hgdata.com. (2017).

[13] Thomas Hofmann. 1999. Probabilistic latent semantic indexing. In *Proceedings of ACM SIGIR.* ACM, 50–57.

[14] Tommi S. Jaakkola and David Haussler. 1999. Exploiting Generative Models in Discriminative Classifiers. In *Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems II.* MIT Press, Cambridge, MA, USA, 487–493.

[15] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013a. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013).

[16] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013b. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013.* 3111–3119.

[17] Katsiaryna Mirylenka, Graham Cormode, Themis Palpanas, and Divesh Srivastava. 2015. Conditional heavy hitters: Detecting interesting correlations in data streams. *The VLDB Journal* 24, 3 (2015), 395–414.

[18] Katsiaryna Mirylenka, Christoph Miksovic, and Paolo Scotton. 2016a. Applicability of Latent Dirichlet Allocation for Company Modeling. In *Industrial Conference on Data Mining (ICDM'2016).* 55–60.

[19] Katsiaryna Mirylenka, Christoph Miksovic, and Paolo Scotton. 2016b. Recurrent neural networks for modeling company-product time series. In *2nd ECML/PKDD Workshop AALTD.*

[20] K. Mirylenka, T. Palpanas, G. Cormode, and D. Srivastava. 2013. Finding interesting correlations with conditional heavy hitters. In *IEEE 29th International Conference on Data Engineering (ICDE'2013).* 1069–1080.

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[22] Anand Rajaraman and Jeffrey David Ullman. 2011. Data Mining. In *Mining of Massive Datasets.* Cambridge University Press, 1–17.

[23] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks.* ELRA, Valletta, Malta, 45–50.

[24] Haşim Sak, Andrew Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth Annual Conference of the International Speech Communication Association.*

[25] Ruslan Salakhutdinov and Andriy Mnih. 2008. Bayesian probabilistic matrix factorization using Markov chain Monte Carlo. In *Proceedings of the 25th international conference on Machine learning.* ACM, 880–887.

[26] L. van der Maaten and G.E. Hinton. 2008. Visualizing High-Dimensional Data Using t-SNE. *Journal of Machine Learning Research* 9 (2008), 2579–2605.

[27] Michail Vlachos, Francesco Fusco, Charalambos Mavroforakis, Anastasios Kyrillidis, and Vassilios G Vassiliadis. 2014. Improving co-cluster quality with application to product recommendations. In *CIKM.* ACM, 679–688.

[28] Chyi-Kwei Yau. 2017. Recommend. https://github.com/chyikwei/recommend. (2017).

[29] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

Big POI Data Integration with Linked Data Technologies

Spiros Athanasiou[1], Giorgos Giannopoulos[1], Damien Graux[2], Nikos Karagiannakis[1], Jens Lehmann[2,3]
Axel-Cyrille Ngonga Ngomo[4,5], Kostas Patroumpas[1], Mohamed Ahmed Sherif[4,5], Dimitrios Skoutas[1]
[1]Information Management Systems Institute, Athena Research Center, Greece
[2]Enterprise Information Systems Department, Fraunhofer IAIS, Germany
[3]Smart Data Analytics Group, University of Bonn, Germany
[4]Computing Center, Universität Leipzig, Germany
[5]Data Science Group, Universität Paderborn, Germany
{spathan,giann,nkaragiannakis,kpatro,dskoutas}@imis.athena-innovation.gr,
{Damien.Graux,Jens.Lehmann}@iais.fraunhofer.de,{mohamed.sherif,axel.ngonga}@upb.de

ABSTRACT

Point of Interest (POI) data constitutes the cornerstone in many modern applications. From navigation to social networks, tourism, and logistics, we use POI data to search, communicate, decide and plan our actions. POIs are semantically diverse and spatio-temporally evolving entities, having geographical, temporal, and thematic relations. Currently, integrating POI datasets to increase their coverage, timeliness, accuracy and value is a resource-intensive and mostly manual process, with no specialized software available to address the specific challenges of this task. In this paper, we present an integrated toolkit for transforming, linking, fusing and enriching POI data, and extracting additional value from them. In particular, we demonstrate how Linked Data technologies can address the limitations, gaps and challenges of the current landscape in Big POI data integration. We have built a prototype application that enables users to define, manage and execute scalable POI data integration workflows built on top of state-of-the-art software for geospatial Linked Data. This application abstracts and hides away the underlying complexity, automates quality-assured integration, scales efficiently for world-scale integration tasks, and lowers the entry barrier for end-users. Validated against real-world POI datasets in several application domains, our system has shown great potential to address the requirements and needs of cross-sector, cross-border and cross-lingual integration of Big POI data.

1 INTRODUCTION

Our daily lives evolve around locations. From navigation applications, to social networks, tourism, and logistics, we use information about locations to search, communicate, decide, and plan our actions. Selecting a location for a given activity has a varying complexity and significance, ranging from simple, everyday routines (e.g., where to have dinner), to more complex planning (e.g., where to open a shop), and to life-changing decisions (e.g., where to live, work or invest). Locations that exhibit a certain interest or serve a given purpose are commonly referred to as *Points of Interest* (POIs), covering anything from shops, restaurants or museums to ATMs or bus stops. POIs are complex entities that are characterized by their *geospatial shape* (points, lines, polygons) along with various *thematic attributes* indicating their name, type, functionality, services, etc., as well as their *relations* to each other (e.g., containment, part-of) with respect to spatial, temporal, and thematic contexts.

Creation, update, and provision of POI datasets is a multi-billion, cross-domain, and cross-border industry. Advances in the timely and accurate provision of POIs result into significant direct and indirect gains throughout our Digital Economy[1]. The value and impact of POIs is reflected in the complex, expensive and labor-intensive effort required for their production and maintenance, which inherently involves stakeholders and users throughout their value chain. Initial production involves field-work, constant monitoring for their evolution and accuracy, integration of user-feedback mechanisms for reporting errors, quality assurance of new data, and roll-out across a plethora of services and products. The greater the *size*, *timeliness*, *richness*, and *accuracy* of POI data, the better the product's *value*. Inversely, incomplete or inaccurate information has a profound effect on all types of end-users and applications.

The *value chain* of POI data has rapidly changed in the last few years. The advent of *open* data, *crowdsourcing*, and *social media* provides new data sources of even greater volume, heterogeneity, diversity, veracity, and timeliness. Such Big POI Data assets are harnessed by both startups and established commercial vendors alike to enrich their products, while also giving rise to new business models founded on domain-specific collection and provision of POIs (e.g., Foursquare[2], Yelp[3]). Enrichment, curation, and update of POI data is increasingly becoming collaborative, with stakeholders and end-users actively involved in all steps of the value chain. This intensifies the challenges relating to their quality-assured integration, enhancement, and sharing.

POI data is by nature *semantically diverse* and *spatiotemporally evolving*, representing different entities and associations depending on their geographical, temporal, and thematic context. Due to its use in various domains and contexts, POI information is typically fragmented across diverse, heterogeneous sources. These are common issues in Big Data integration [7], but combining and assembling POI information is further hindered in practice by the lack of common identifiers and data sharing formats. Even the way we typically identify and share information about POIs is inherently *ambiguous*. Addresses, coordinates, and place names are equally used throughout applications as pseudo-identifiers; but practice shows that they fail to effectively disambiguate POIs. Integrating POI data using current approaches remains labor-intensive and does not scale, thus limiting data coverage. The industry makes typically two compromises to reduce the complexity to feasible levels: focus on a specific domain (e.g. fuel stations), and/or restrict the spatial-, temporal-, and feature-space of data. In both cases, this leads to loss of information and thus lost value.

[1] https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=COM:2018:0232:FIN
[2] https://enterprise.foursquare.com/products/places
[3] https://www.yelp.com

Series ISSN: 2367-2005

10.5441/002/edbt.2019.44

To tackle these challenges of Big POI data integration in the context of the SLIPO project[4], we transfer knowledge and apply state-of-the-art techniques and tools from the domains of Linked Data, Big Data and GIS. We argue that *Linked Data* technologies [2] are ideally suited to handle the inherent geospatial, thematic, and semantic ambiguities of POIs, without resorting to general-purpose entity matching platforms [16]. Recent advances in spatially-aware Linked Data technologies[5] address the scalability challenges of integrating, enriching, and querying semantically diverse geospatial Big Data assets and can effectively maximize the value extracted from open, crowdsourced and proprietary data sources.

The scope and ambition of our work in terms of complexity and coverage is inherently defined by current practices and needs in the industry. Indicatively, HERE Places API[6] offers information about 55 million POIs with names and categories in 237 countries; Google Places API[7] advertises over 150 million POIs globally; from OpenStreetMap, we have extracted more than 18.5 million POIs regarding specific categories[8]. Hence, we are targeting data integration concerning millions of POIs. We provide a complete suite of integrated software and services for POI data integration, supporting all stages of the POI data lifecycle (transformation, linking, fusion, enrichment). Our prototype application employs mature, scalable, open-source software specialized in geospatial linked data integration. We have tested its operation against several use cases for POIs in different application domains (geomarketing, tourism, navigation) with very encouraging results concerning execution cost and accuracy. Our experience shows that stakeholders can orchestrate those tools in coordinated, iterative *workflows* to progressively increase both the size and the quality of the integrated POI data.

The remainder of this paper is organized as follows. Section 2 presents the main challenges concerning Big POI data integration. Section 3 outlines the POI data integration lifecycle applied in SLIPO. Section 4 describes the SLIPO data model for representing information about POIs throughout the executed workflows. In Section 5, we explain the specific processes involved in the POI data integration process. Section 6 outlines the provided functionalities for extracting value-added analytics from the integrated POI datasets. Section 7 presents the current status of our prototype application. In Section 8, we report our experience from POI data integration scenarios in two real-world use cases. Finally, Section 9 summarizes the paper and discusses future work.

2 CHALLENGES IN POI DATA INTEGRATION

POIs are complex entities, described and associated with multifaceted and multi-modal information. They also exhibit complex (spatial, temporal, thematic) relationships, and they often have a long and complex lifespan. Any effective and systematic approach towards POI data integration needs to rely on robust, flexible and semantically-rich modeling of POI profiles and handling of POI identifiers, especially for applications dealing with cross-sector, cross-border, and cross-lingual content.

Consider a simple example regarding an imaginary POI. Suppose that the "Acropole Palace Hotel" was registered in 2015 at the local *Yellow Pages* directory and was assigned an identifier, along with some basic information (name, address, telephone, fax), as illustrated at the upper side in Figure 1. Later on, its address was geocoded and the resulting location coordinates were also included in the record to be used in a mobile city guide. At some point in time, this hotel was acquired by another company, which renovated and rebranded it to "Xenia Hotel", and also opened a restaurant at the roofgarden. This hotel is since listed in a *hotel reservation application* under a new identifier and with updated, multi-lingual information, which also features its current offers and services, as well as customer ratings (shown at the bottom side of Figure 1). Such changes need to be detected and included in the mobile city guide application, by matching and integrating new POI representations with old ones to avoid duplicates and to update obsolete information. In this example, such a matching (denoted with an `owl:sameAs` link) can be based on the phone number and the address of this POI (properties marked with red color in Figure 1). The challenge arises from the inherent *heterogeneity* in POI data characteristics. As shown in this example, this involves diverse attribute schemata, varying formats in values (e.g., in phone numbers) or different representations of the same information (e.g., address represented in a structured or unstructured format), which need to be resolved to determine whether these two records actually refer to the same POI.

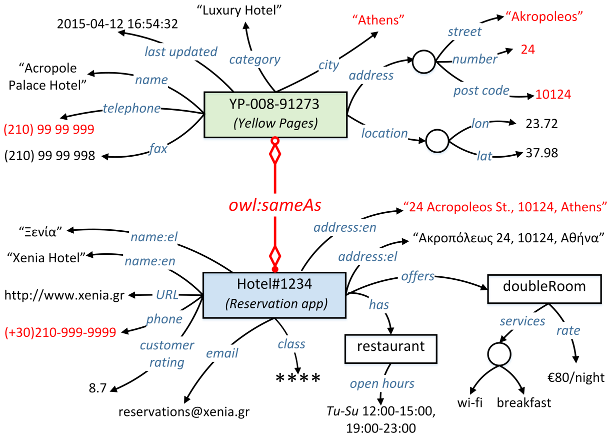


Figure 1: Matching different representations of the same POI coming from two data sources.

This example is only indicative of how different companies in different sectors and application domains collect, use, and extend information about the same POI. It also points out the importance of *volatile* data in the POI profile, e.g., its facilities, prices, events, etc. Moreover, it brings up several cases of *ambiguity* that arise and lead to data integration challenges that stakeholders have to face throughout this process. Next, we outline these challenging issues in POI data integration:

- *Lack of standardization.* Even though POI data is ubiquitous, there are no de jure standards yet for POI models, formats and identifiers. This is due to licensing and commercial competition, different hardware and software characteristics, and diverse requirements among mobile and web applications. Thus, POI datasets provided by different vendors are often not compatible with each other and require excessive effort and domain knowledge to be integrated and reused. At the most basic level, there are no (globally) *unique identifiers* assigned to POIs, making it difficult to identify duplicates among datasets and to link together different pieces of information for the same POI.

[4] Acronym for **S***calable* **L***inking and* **I***ntegration of big* **PO***I data*, http://slipo.eu/
[5] blog.geoknow.eu/the-linked-data-stack, http://aksw.org/Projects/LOD2.html
[6] https://developer.here.com/products/geocoding-and-search
[7] https://cloud.google.com/maps-platform/places/
[8] http://slipo.eu/?p=1397

- *Inherent ambiguity of POIs.* POIs are entities having a twofold nature, *geospatial* and *semantic*; moreover, their characteristics and associated information *evolves over time*. This results in multiple sources of ambiguity when dealing with POI data. In the spatial dimension, coordinates given for the same POI in different sources typically differ, while the spatial extent of a POI is often ignored, and its location is abstracted and approximated by a single point. Even if the shape is retained, it may have varying levels of accuracy. Hence, when encountering multiple POIs in different sources with slightly different coordinates or shapes, it is challenging to determine whether these refer to the same or different real-world entities. Similar issues arise when using POI names. The same POI may appear with slight naming variations in different sources, while different POIs may in fact have the same or similar names. Furthermore, different sources employ different classification or tagging schemes to categorize POIs and describe their type. This ambiguity is amplified when the temporal dimension is introduced, e.g., for determining whether two different representations refer to the same POI that evolved over time (as in Figure 1) or to two distinct POIs.
- *Long update cycles.* Due to the effort needed for maintaining and curating POI datasets, the contained information often remains relatively static. Further, it typically focuses on certain, mostly factual aspects of a POI, such as its title and a set of categories or tags. When and how this information is updated depends on the way it is collected and the available resources. Hence, for most providers, POI data is updated in yearly cycles. Moreover, if and when it is refreshed, typically the dataset will just be updated to the latest version, as it is not straightforward to apply a systematic and principled approach for recording and representing the *evolution* that has occurred, or more generally to track and record events that are related to that entity. Hence, even though such information may actually exist, there is often no historical profile of a POI that evolves over time and keeps track of associated events.
- *Fragmented POI profiles.* Based on how and for what purpose a POI dataset has been created, its contents typically cover only certain aspects of the POIs. For example, a navigation service and a city guide may have different priorities when deciding which POIs to include and what kind of information about them to collect. Although a wealth of information may exist for a POI, different parts and pieces may be found in different (types of) sources. Still, more complete POI profiles would allow more sophisticated and accurate analyses.
- *POIs treated independently and out of context.* POI datasets are typically treated as collections of individual entities. Each POI is modeled, stored and analyzed independently, without considering or establishing connections and links to other POIs. The reason is that *relationships* between POIs are more difficult to model, represent and analyze. However, this significantly limits the type of analyses that can be carried out over sets of POIs and creates gaps with the actual needs of users.

In SLIPO, our approach places particular emphasis on these issues related to POI models, identifiers, and resolving the aforementioned cases of ambiguity, as explained next.

3 THE POI DATA INTEGRATION LIFECYCLE

In this Section, we provide an overview of the POI data integration lifecycle supported in SLIPO. The underlying idea of our proposed system is to address the POI data integration challenges in the Linked Data domain [2]. Thanks to a simple, standardized, yet flexible model (*Resource Description Framework*[9]), Linked Data technologies can handle the inherent geospatial, thematic, and semantic ambiguities of POIs. Existing POI data assets need first to be transformed into RDF, so that individual POI profiles can be interlinked, fused, and enriched. This takes place in successive steps that progressively increase the size and/or the quality of POI data throughout a virtuous cycle, implementing an iterative workflow (Figure 2). Next, we outline the purpose of each stage, the processes that take place, and their output.

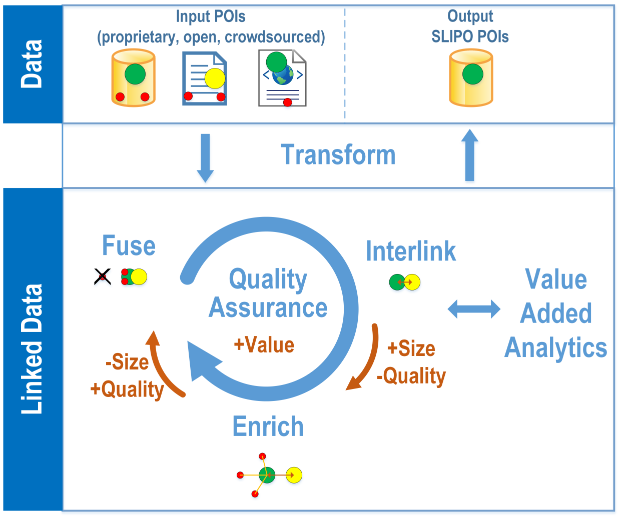


Figure 2: The POI data integration lifecycle.

The lifecycle begins with a *transformation* stage. This assumes as input POI data collected from heterogeneous and diverse data sources (proprietary, open, crowdsourced), having different attribute schemata and formats. The spatial, temporal, and thematic attributes in the input data are transformed into RDF triples conforming to a common, vendor-agnostic, well-defined, yet agile and extendable *POI ontology*. Hence, schema mappings from attributes of the original schemata to the classes and properties of this ontology are applied. After transformation, the resulting RDF triples can be stored in files or in an RDF store.

Subsequent stages are applied in the Linked Data domain against the previously transformed RDF data comprising an iterative, step-wise *workflow* that first increases the *size*, and then the *quality* of POIs. This forms a *virtuous cycle* that begins by expanding POI coverage, completeness, and richness, delivering data of greater size. Then, it focuses on increasing the quality of the POI data, fusing these intermediate results and enforcing appropriate quality assurance algorithms. This inherently reduces the size of data in absolute numbers, but increases their value. This process can be repeated in the same manner, iteratively increasing the size and then refining to increase quality, as many times as required. For example, an expert user can introduce additional data sources, apply different rules, focus on other types of metadata, etc. Such an iterative workflow involves the following stages:

- *Interlinking.* This is applied among transformed RDF datasets to link together individual RDF representations of the same real-world POI. It exploits structural properties, textual similarities, spatial proximity, etc., based on various user-specified

[9] https://www.w3.org/RDF/

metrics and thresholds. This deduplication process creates owl:sameAs links between matching POI entities, thus tackling the lack of common identifiers between POI entities across data sources, and enabling their management at later stages of the integration process.

- *Enrichment.* This step identifies and retrieves additional information from external sources that relates to the processed POIs, and creates extra properties for these POIs, increasing the richness and completeness of the data. It also discovers semantic relations between POI entities and other resources (e.g. areas, events, time), such as partOf relations (e.g., a school yard is part of a school etc.) or occursAt relations (e.g., events at a certain venue). Essentially, this phase enhances and contextualizes POI profiles with information extracted and assembled from other relevant sources, leading to a multi-faceted and more comprehensive view of POIs. For instance, it can enrich a POI profile with extra information concerning opening hours, price ranges, event timelines, etc.
- *Fusion.* This stage consolidates linked POIs and their properties. From two linked POI entities, it produces a unified and consolidated representation that is more complete, concise and accurate than the individual initial linked entities. For matching properties (e.g., similar names, nearby geometries) between a pair of linked entities, potentially different fusion actions can be applied. Fusion actions may include selection of properties from one of the entities, merging of properties from both entities, selection of those properties that satisfy a specific criterion (e.g., more complex, more timely), etc. Eventually, this leads to significant increase in completeness, coverage, timeliness and quality of POI data.
- *Value Added Analytics.* The POI data lifecycle includes an additional step, offering large-scale aggregation analytics in various dimensions, clustering, association rule mining, and predictive analytics. We can perform large-scale spatial, temporal, and thematic aggregation or extract *associations* between POIs and other entities. Results can be modelled as integrated sets or sequences of POIs. For instance, such a set can represent a thematic area of interest not explicitly defined in the original data, but implicitly discovered; this encapsulates a set of POIs that are spatially co-located and related thematically for direct exploitation in several use cases (e.g., geomarketing).

Quality assurance is performed throughout all steps of the lifecycle and ensures that each phase produces correct and accurate results, taking into account dataset-specific and use-case-specific quality indicators and rules, including also manual validation and authoring. Several indicators can be used, most of them already adopted by industrial vendors that manage and exploit POIs: size, timeliness, coverage, accuracy, etc.

Eventually, at the end of the POI data integration lifecycle, another transformation is also required. This involves the *reverse transformation* of the integrated POI data back to conventional formats (i.e., de facto POI formats), enabling their use in existing products, systems and services.

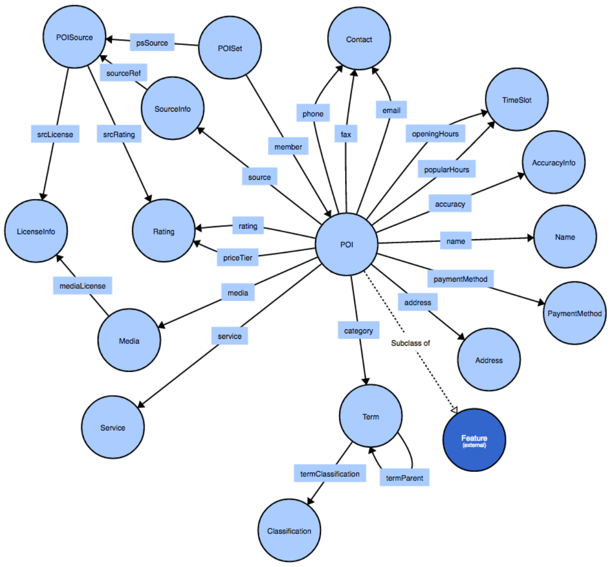


Figure 3: Main classes and properties in the POI ontology.

4 POI DATA MODEL

We have developed a comprehensive ontology for POI data to model and represent multi-faceted and enriched POI profiles and thus leverage Linked Data techniques and tools for POI data integration. This model accommodates and extends existing POI formats, providing a uniform and semantically rich model for assembling and managing POI data from heterogeneous sources, since a widely accepted, de facto model is missing. To design this ontology, we have taken into consideration POI data representations employed by several well-known and widely used POI data sources. These included both open data sources, like OpenStreetMap and Wikimapia[10], as well as proprietary data schemata like Foursquare or Google Places. We also studied the structure of POI data assets based on samples obtained by commercial vendors in several countries (Austria, Germany, Greece). Finally, we investigated other efforts, especially an ongoing work on POI modelling by W3C and OGC Working Groups[11], as well as models for representing places proposed by collaborative projects[12].

The main challenge in POI data modelling arises from the high degree of heterogeneity across sources, given that these have different scope, goals and purposes. Thus, they model POIs from different perspectives and with varying levels of detail. Among the main issues we addressed with our POI data model are:

- *Names*: Accommodate the presence of different types of names (e.g., official vs. commonly used ones), alternate or historical names, acronyms, multi-lingual names, etc., as well as phonetic transcriptions or transliterations useful for internationalized POI information.
- *Addresses*: Represent addresses both in structured formats (separate components for country, city, street, house number, etc.) and flat formats (concatenated strings), with potentially missing information in either case.
- *Geometry*: Allow both for point geometries (longitude and latitude coordinates) and more complex geometries (lines, polygons), potentially in different coordinate reference systems.
- *Classification*: Support both hierarchical categorizations (i.e., categories, subcategories, etc.) and free tags.
- *Source* of POI data (e.g., a commercial vendor) may be specified using its title, homepage, description, license, logo, etc.
- *Media*: Support for auxiliary assets such as photos, videos or audio associated with a POI.

[10] http://wikimapia.org/api/
[11] http://www.opengeospatial.org/projects/groups/poiswg
[12] https://schema.org/Place

- *Contact information*: Include various elements, such as phone, email, fax, Web page, social media accounts, etc., with different structures and formats.
- *Amenities*: Include extra information on a variety of attributes such as offered services, opening and/or popular hours, ratings, payment methods, etc.

Figure 3 illustrates a graph with the main classes and object properties of our OWL ontology[13] as used internally in the POI data integration lifecycle. POI is the main class for representing POI features and is modelled as subclass of a spatial Feature in GeoSPARQL [24], thus directly inheriting properties regarding geospatial location. It supports multiple geometric representations, and the type of each geometry (e.g., centroid, navigation point, map pin, boundary) may be specified as well. Extra attributes for specialized use cases not covered by the ontology can be represented in the form of key/value pairs.

Overall, this ontology adheres to well-established standards (RDF, GeoSPARQL) and is geared towards processing efficiency. In addition, it can be extended and enhanced with domain-specific POI profiles, which include additional properties and relationships. We are currently working on modelling the provenance of POIs and their metadata, their evolution across time leading to different versions, as well as changes occurring to their contents and representation (e.g., extra attributes).

5 POI DATA INTEGRATION STEPS

SLIPO offers a complete and integrated suite of software tools that support all steps of the POI data integration lifecycle. All tools are available as open source software and are the state of the art in geospatial linked data integration. Specifically, the suite includes: (a) TripleGeo, for POI data and metadata *transformation* into RDF; (b) Limes, for *interlinking* of POIs; (c) Fagi, for the *fusion* of linked POI data into unified, concise and complete descriptions; and (d) Deer, for *enrichment* of POIs with implicit metadata, third party datasets and thematic, temporal and spatial metadata. Next, we describe them in more detail.

5.1 Transformation

Transformation is the entry point for POI datasets, converting them from their original format to RDF, thus enabling their subsequent processing in the Linked Data domain. We have extended our open-source *Extract-Transform-Load* (ETL) software TripleGeo [25] to enable scalable and efficient transformation of POI datasets from a variety of de facto geospatial formats into RDF triples. In TripleGeo[14], we employ adaptable, configurable, and reusable mappings from existing attribute schemata into our POI ontology (Section 4). We also support classification hierarchies for assigning categories to POIs. Moreover, TripleGeo can handle all common geometry data types and established coordinate reference systems. Its main features include:

- *Native support for a multitude of geospatial data formats.* Currently, TripleGeo supports 9 common file formats (e.g., ESRI shapefiles, GML, KML, CSV, JSON), and JDBC-based access to 8 geospatially-enabled DBMSs (e.g., Oracle Spatial, PostGIS).
- *Improved geospatial support* not only of primitive geometry types (points, linestrings, polygons), but also more complex geometries (MultiPolygons, Geometry Collections), as well as on-the-fly reprojection to another coordinate reference system.
- *User-defined mappings* specify rules that dictate how to generate RDF triples from original thematic attributes in the input POI data according to a given ontology. Such mappings allow transformation of all available attributes per POI entity and can be specified in the generic RDF Mapping Language RML [5, 6]. We also provide an alternative, simplified mapping facility specifically tailored to our ontology, offering much faster transformation even for very large volumes of POI data.
- *Classification schemes.* POI data providers employ diverse classification or tagging schemes to categorize POIs and describe their type. TripleGeo accepts specification of (possibly hierarchical) classification schemes for POIs, produces RDF triples that fully describe this information along with especially assigned URIs, and introduces extra links between a POI and its respective category under this scheme.
- *Customized URIs.* We construct HTTP Universal Resource Identifiers for POIs based on automatically generated Universally Unique Identifiers (UUIDs). This follows recommended patterns and best practices for creating persistent, unique, vendor and technology independent URIs. Thus, POI data owners have enough flexibility and control over creating and managing their own POI identifiers, while still adhering to a uniform format.
- *Reverse transformation* from RDF to all common geographical file formats. POI data that have been interlinked, fused, and enriched in previous executions of the POI data integration lifecycle can become accessible and exploitable by existing software (e.g., DBMS, GIS) or services (e.g., web mapping, route planning) commonly utilized in the industry.
- *Comprehensive configuration.* Users can control various parameters of the transformation process at different levels of complexity based on their technical background and expertise.
- *Compliance to standards.* The produced RDF geometries are fully compliant with the OGC GeoSPARQL standard for RDF spatial entities [24]. Transformation of INSPIRE-aligned data and metadata [10] is also supported [26], thus abiding by the EU Directive for interoperable Spatial Data Infrastructures.
- *Scalability.* Our experiments with various data sources show that TripleGeo is currently orders of magnitude faster compared to its original release [25], as well as faster than GeoTriples [17]. It can efficiently transform millions of POIs even without any sophisticated data partitioning schemes. Indicatively, it can transform all 7.4 million POIs extracted from OpenStreetMap for Europe into RDF triples in less than 3 minutes, effectively generating more than 715,000 triples/sec.

Using TripleGeo, we have launched a free download service[15] that offers global POI data extracted from OpenStreetMap and transformed into RDF format, retaining all original tags and also creating resolvable, machine-readable URIs per POI.

We are currently working on extending TripleGeo towards *semi-automatic workflows* to assist and guide users in creating attribute mappings for new datasets. We have built a utility that employs Machine Learning to learn new mappings from a corpus of previously specified ones, available from the various use cases of POI datasets we have handled so far. This utility also analyzes the contents of each attribute in a new POI dataset, based on its data type (string, numeric, etc.), formatting (e.g., phone numbers, postal codes), as well as the presence of special characters. Users can then verify or modify the automatically suggested mappings through a graphical interface before applying them for transforming their POI data into RDF.

[13] OWL ontology is available at https://github.com/SLIPO-EU/poi-data-model

[14] https://github.com/SLIPO-EU/TripleGeo

[15] http://download.slipo.eu/results/osm-to-rdf/

5.2 Interlinking

The aim of interlinking POI datasets is to develop scalable approaches for integrating massive heterogeneous, and incomplete POI data at a world-scale. Limes[16] is integrated in SLIPO and incorporates many algorithms for performing efficient interlinking among POI resources. In the context of SLIPO, Limes receives as input two RDF POI datasets conforming to the SLIPO ontology. Thus, Limes's input POI data are first transformed by TripleGeo, into the proper RDF format and schema. Further, apart from the two input POI datasets, Limes requires as input a configuration file containing the Limes configuration parameters. Limes's output consists in a single file, which contains the links between corresponding POI entities from both input POI datasets. The output of Limes is essential for running other SLIPO tools in a POI data integration cycle (Figure 2).

Limes v1.0.0 is the first version of Limes that has been developed in the context of the SLIPO project and focuses on POI-specific interlinking. One of the major goals was to abstract as much complexity as possible from the end users. So, in order to keep user interaction at a minimum and requiring no knowledge of Linked Data technologies and concepts, we aimed at adapting and fine-tuning Limes's functionality specifically for POI data, as well as at automating the interlinking process as much as possible. To this end, we emphasized on the development of the backend of the platform, aiming to enrich and specialize the core interlinking functionality of the framework. Next, we outline the new features and functionality of Limes:

- *POI-specific point-set distances.* New point-set distances based on the vector representations of the POI resources (e.g. *Hausdorff, mean, surjection* and *sumOfMin*). Altogether, we implemented a set of 10 point-set distance functions based on our survey published on [30].
- *Topological relation discovery* based on the vector representations of the POI resources (e.g. a POI resource contains, crosses or touches another POI resource). For instance, find all parking locations within shopping malls. Therefore, we develop Radon [8, 29], an efficient algorithm for rapid discovery of topological relations among POI resources with 2D geometries.
- *Temporal relation discovery* based on timestamps associated with the POI resources (e.g., one POI takes place after, before or during another POI). For example, a specific area is used as a parking location only during a football match. To tackle this problem, we proposed Aegle [13], a novel approach for efficient computation of links between POIs' temporal representations according to *Allen*'s interval algebra.
- *Combining the new techniques with the ones already in Limes.* In Limes v.1.0.0 we integrated the novel algorithms for the 10 POI-point-set distances as well as Radon and Aegle into the Limes core. In particular, a new mapper is implemented for each of the new relation types. Such mappers are combined with the already existing mappers for efficient link discovery of the new types of relation specific for POI resources.
- *Novel machine learning approaches for POI interlinking.* In most cases, finding a good metric expression[17] (i.e. one that achieves high F-Measure in interlinking POI entities) is not a trivial task. Therefore, in Limes we implemented Wombat, a novel machine learning approach for auto-generation of mappings among POI resources. Wombat is inspired by the concept of generalisation in quasi-ordered spaces [28]. Wombat minimizes the Limes configuration task by providing unsupervised, supervised and active learning versions.
- *Integration with the SLIPO Workbench.* Limes v.1.0.0 realizes two deployment modes: (a) standalone, as an individual software that accepts as input linked POI datasets and provides as output a mapping file containing the links between the input POI datasets; (b) deployment within the SLIPO Workbench, where Limes serves as an integral component of the SLIPO Toolkit and is loosely integrated by the SLIPO Workbench with the other software components into forming POI integration workflows.
- *Scalability.* In addition to the original Limes parallelization algorithm [31] and optimized planners [12, 22], in Limes v1.0.0 we evaluated the scalability of our novel POI-specific approaches. Our evaluation proves that Radon [8, 29] is able to outperform state-of-the-art approaches up to 3 orders of magnitude while maintaining a precision and a recall of one. Also, our evaluations of the runtime of Aegle [13] show that Aegle outperforms the state of the art by up to 4 orders of magnitude while maintaining a precision and a recall of one. Recently, we have implemented a simple, yet efficient in our setting, distributed execution scheme, which functions independently of core interlinking in Limes. Specifically, we have two implementations based on Spark and Flink frameworks. Currently, we run an intensive evaluation for both frameworks to find the pros and cons of each. Finally, we studied the effect of geometry simplification on the scalability of POI interlinking [1]. We found that a suitable simplification setting can reduce interlinking cost with a minimum effect on quality.

5.3 Fusion

The fusion process in SLIPO follows the interlinking of different representations of the same POI across data sources. Fusion addresses the problem of assembling partial and incomplete POI profiles as well as resolving conflicting information in order to derive a more complete, consolidated profile per POI.

Fusion receives as input two POI datasets as well as a set of links between them. The output is a third merged dataset, which contains consolidated descriptions of the linked POIs. Each POI in the fused dataset is described by a set of richer, non-redundant, non-conflicting and complete properties, which have been derived by merging the initial descriptions of the linked POIs. The main challenge in this task is to efficiently apply the most appropriate fusion action in such way that the best elements of individual datasets are kept in the final composite dataset.

To support scalable and quality assured fusion of large POI datasets, we have extended our fusion framework Fagi [14]. Initially, Fagi was a map-based, user-interactive platform for manually performing property matching and fusion actions on individual properties of linked geospatial entities. In SLIPO, we adapted Fagi[18] to effectively handle the fusion of POI data, while we also minimized manual user effort. Specifically, the current version offers the following main features:

- *Advanced fusion facilities for POIs.* Fagi supports the graphical authoring of POI fusion specifications. These sets of rules examine the individual properties of pairs of linked POIs and decide, for each property, the most fitting fusion action. Fagi currently incorporates 25 *condition functions* for examining the properties of linked POIs, including string similarity, geometry comparison, etc. Further, it implements 15 *fusion actions*

[16] https://github.com/dice-group/LIMES

[17] A metric expression is a logical expression that describes when two resources should be linked.

[18] https://github.com/SLIPO-EU/FAGI

(regarding both thematic and geospatial properties) for deciding how to merge the values of matching properties. Fusion actions include: aligning geometries, maintaining the most complex value, maintaining both values for the same property, etc. Finally, combining several condition functions can be used to construct more elaborate fusion rules, while fusion results can also be marked as *ambiguous* for later inspection by the end user.
- *Link validation functionality.* An important aspect of quality assurance lies in validating the fusion input and deciding whether the linked entities should be either fused, further examined, or rejected as erroneous. To this end, in Fagi we define a set of validation actions, as well as a validation rule specification scheme. Similarly to POI fusion specifications, the user can define elaborate link validation specifications that jointly examine several properties of pairs of linked POIs in order to maintain or reject the specific linked POIs.
- *Quality indicators extraction.* Further emphasizing on quality assurance, Fagi supports the extraction of more than 25 quality indicators. The user is able to review several statistics on the input linked POI datasets before performing fusion on them (*pre-fusion statistics*), as well as on the output fused data (*post-fusion statistics*). The former provide an overview of the data at hand, which assists the integrator to properly define and configure the validation/fusion rules. The latter assist the user in the inspection of the fusion results, and potentially guide her into re-configuring and re-executing the fusion process.
- *Recommendation of link validation and fusion actions.* Fagi implements learning mechanisms for training on past user actions and recommending link validation and fusion actions for new POIs. It learns binary (for link validation) and multi-class (for fusion actions) classifiers on a series of extracted training features regarding the properties of the linked POIs. Then, it recommends actions for new pairs of linked POIs.

The aforementioned functionality of Fagi satisfies commercial-level data fusion needs. In a typical fusion scenario, the user can define configurable and re-usable fusion rule specifications of varying complexity, which collect names from different datasets in multiple languages or types (such as official, international, brand-names etc.) and complete other attributes, such as address information, websites, phone numbers, emails, ratings, reviews, opening hours, image links, etc. The resulting fused dataset then contains POIs with the most complete and accurate descriptions, as well as more precise and/or more complex geometries. Additionally, the user is able to examine a plethora of quality indicators and use them to assess and potentially improve the quality of the fusion process.

Fagi v2.0 focuses on satisfying the effectiveness and performance requirements of fusing Big RDF POI data. Thus, an important effort has been made to fine-tune the underlying algorithms in order to increase their efficiency and scalability. The performance of the current implementation of Fagi is tested against real-world commercial POI datasets, by applying a custom partitioning and distributed processing scheme that occupies 10 nodes and takes less than five minutes to fuse 1 million linked POIs, which corresponds to a country-level fusion process.

5.4 Enrichment

Enrichment is one of the main parts of the data integration process. In SLIPO, enrichment focuses on POI entities that are characterized by a set of major properties (e.g. name, coordinates and category) as well as potentially several additional properties (e.g. address, telephone, email, rating, etc.). Enrichment considers one or more input dataset(s) containing POIs. The goal of enrichment is to produce one or more enriched dataset(s), containing better descriptions of the input POIs based on information retrieved from external, third-party RDF data sources (e.g., SPARQL endpoints, DBpedia). That is, each POI entity in the final, enriched dataset must be described by a set of RDF triples that have been derived by merging the initial description for the POI with those generated via various enrichment operations. Note that some enrichment approaches can define a set of triples to be removed from the original POI descriptions. Those removed set of triples are either wrong or inaccurate. The enrichment process can also replace inaccurate triples with ones with correct values. Considering the big picture of the POI integration lifecycle (Figure 2), the enrichment process is tightly interconnected with validation and quality assurance. To this end, the enrichment process needs to incorporate several mechanisms to assess the quality of the proposed enrichment operations and their results.

The enrichment task is carried out via our generic enrichment component Deer[19] [27]. Deer incorporates many approaches for performing efficient enrichment among POI resources. In the context of SLIPO, Deer receives as input one or more RDF POI dataset(s) conforming to the SLIPO ontology. Thus, Deer's input POI data are first transformed by TripleGeo into the proper RDF format and schema. Moreover, Deer input datasets may be linked via Limes prior to be enriched by Deer. Further, Deer requires as input a configuration file containing its configuration parameters. Deer's output consists of one or more files that contain the enriched versions of the respective input datasets.

Deer offers a set of *enrichment operators*, i.e., artifacts in charge of enriching input POI dataset(s). The input for such an enrichment operator is a set of one or more datasets. The output is also a set of one or more enriched datasets. In the following, we highlight some of Deer's enrichment operators:

- *The Dereferencing enrichment operator.* For POI datasets which contain similarity proprieties links (e.g. `owl:sameAs` to DBpedia resources), we deference all links from our source dataset to other datasets (e.g., DBpedia) by using a content negotiation on HTTP. The returned set of triples needs to be filtered for relevant POI resources. Here, we use a predefined list of attributes of interest. Among others, we look for `geo:lat, geo:long, geo:lat_long, geo:line` and `geo:polygon`. This list can be reconfigured via Deer's configuration.
- *The NLP Enrichment Operator* enriches POI resources by extracting embedded POI information hidden within the datatype properties and making it explicit as new triples added to the original POI dataset. For example, find all POIs embedded within the description of all hotel POIs and add them as new triple to the respective hotel POI. The current version of Deer uses the Fox [33] framework for Named Entity Recognition (NER). By default, Deer extracts the POIs based on DBpedia as the background knowledge base. As Deer is a generic POI enrichment framework, the used NER framework can be configured as well as the used background knowledge base.
- *The Geo-Distance Enrichment Operator* aims to enrich a set of POI pairs (not necessarily of the same type) with the great elliptic distance between them. For example, the geo-distance operator can enrich all hotel POIs by adding the distance to the nearest POIs of bus stations/parking lots/hospitals.

[19] https://github.com/dice-group/DEER

5.5 Quality Assurance

Each SLIPO component provides a collection of several quality indicators and statistics. They all produce verbose *execution metadata* that can either be visualized or downloaded for further inspection by the end user. Particular effort has been put in Limes, Deer, and Fagi for POI linking, enrichment and fusion respectively. In particular, both Limes and Deer implement a series of quantitative quality indicators, such as *run-time*, *number of added triples* and *percentage of data increase after linking/enrichment*. Further, both Limes and Deer provide the qualitative quality indicators of *precision*, *recall*, and *F-measure* in cases where benchmark datasets are available. In case no benchmark dasasets are available, Limes is still able to provide the *pseudo-precision*, *pseudo-recall*, and *pseudo-F-measure* first introduced in [23]. The basic assumption behind these pseudo measures is that symmetrical one-to-one links exist between the resources in source and target datasets. Our pseudo-precision computes the fraction of links that stand for one-to-one links and is equivalent to the strength function presented in [15]. The pseudo-recall computes the fraction of the total number of resources (i.e. from both source and target datasets) that are involved in at least one link. Finally, the pseudo-F-measure is the harmonic mean of pseudo-precision and pseudo-recall.

Fagi implements a series of quality indicators (e.g., percentages of fused properties vs. initial POIs/initial links, average numbers of POI property completeness) that compare the linked POI datasets and the resulting fused POIs. In particular, *attribute gain* indicates the percentage of extra properties compared to the original (e.g., a gain of 0.4 on a given POI means it was complemented with 40% additional attribute values). *Confidence* indicates the degree of similarity (in names, geometry, phone number, etc.) between the original features that were fused into a unified one, with values close to 1 indicating almost perfect match. These indicators are utilized both internally in Fagi (similarity measures, learning mechanisms) and as output for the end user, for further inspection and manual validation of fused POIs.

6 VALUE-ADDED POI ANALYTICS

At the end of the POI data integration workflow, various services are provided to perform advanced analytics and extract added value from POIs. Currently available functionality includes *Best Region Search* and extraction of *Areas of Interest*, which can be further modeled using techniques such as LDA or semantic clustering. We describe these functionalities in more detail below.

6.1 Best Region Search

Given a set of POIs $\mathcal{D}$, an $\alpha \times \beta$ rectangle R, and a utility score function $f : \mathcal{P} \rightarrow \mathbb{R}$ that assigns an objective score to any subset $\mathcal{P} \subseteq \mathcal{D}$, the goal of the *Best Region Search* problem is to find the optimal placement of R over the space containing $\mathcal{D}$ such that the value of f over the enclosed subset of POIs $\mathcal{P}$ is maximized [11]. This problem has many applications in various domains, from geomarketing and tourism to real estate and urban planning, facilitating decisions about, e.g., selecting the best location to open a new store or to place an advertisement.

However, the existing state-of-the-art algorithm for this problem [11] only computes the best, i.e., the top-1 result. This is usually not sufficient in practice. For instance, it may not be possible to open a store at the identified best location (no available facilities to rent or purchase), or all hotels in the identified best area may be occupied or too expensive. Then, the user needs to examine alternative solutions in decreasing order of quality, until one is found that meets all desired criteria.

To address this shortcoming, we have introduced the k-Best Region Search (k-BRS) problem, which computes a ranked list of the top-k best regions according to the utility score function. The main challenge in doing so, is that by simply returning a top-k list of results ordered by their objective score, typically produces highly overlapping results. Instead, our proposed algorithm is able not only to compute top-k results progressively, but also to diversify the returned results by either minimizing or completely excluding overlap among them. This is achieved by progressively retrieving subsequent results beyond the top-1, and selecting the next best candidate based on their *marginal gain*, i.e., the added value of each new result in the context of those already selected. A detailed description of the algorithm can be found in [32].

6.2 Extracting and Modelling Areas of Interest

Another functionality for POI data analytics involves the application of spatial clustering to extract Areas of Interest (AOIs) from the integrated and enriched POI dataset and then the use of topic modelling techniques to characterize and compare those areas.

For the first step, we employ *density-based clustering* to identify areas with high concentration of POIs (e.g., shopping malls, transportation hubs, touristic attractions, etc.). We use the DBSCAN [9] or HDBSCAN [4] algorithms for this purpose. DBSCAN can identify clusters of arbitrary shapes. Moreover, it does not require the user to specify the desired number of clusters in advance; instead, the user specifies two parameters that control the density. HDBSCAN extends DBSCAN by offering a mechanism to adjust the density automatically based on the data distribution.

The resulting clusters comprise a set of AOIs but provide no additional information regarding their nature or characteristics. They merely point out that these areas have a higher concentration of POIs compared to the rest of the space. To provide further insights to the user as to what these AOIs are about and how they can be compared to each other, we employ *topic modelling*. Essentially, this draws inspiration from extracting a set of topics over a document collection and representing each document according to those topics.

In our case, we represent each AOI as a “document”, with the contents of the document being a bag-of-words representation of the union (multiset) of terms appearing in the POIs contained in that AOI. These terms may refer to POI categories or tags, or any other terms extracted from POI names, descriptions, reviews, etc. We then perform Latent Dirichlet Allocation (LDA) [3] over this “document” collection. The basic idea behind LDA is that each document can be described by a distribution of topics and each topic can be described by a distribution of words. The result of the process comprises two matrices. The first is a topic-terms matrix that defines each extracted topic as a distribution over POI terms. The second is an AOI-topics matrix that represents each AOI as a distribution over the identified topics.

This approach is quite flexible in practice because it allows to model each AOI as a mixture of topics, instead of assigning each AOI to a single category. This better reflects the fact that typically a region contains a mixture of different types of POIs and serves a mixture of purposes, rather than a single one.

The resulting AOI-topics matrix can be used to compare AOIs to each other, e.g., find AOIs with similar mixture of topics, providing a means to quantify these similarities and differences.

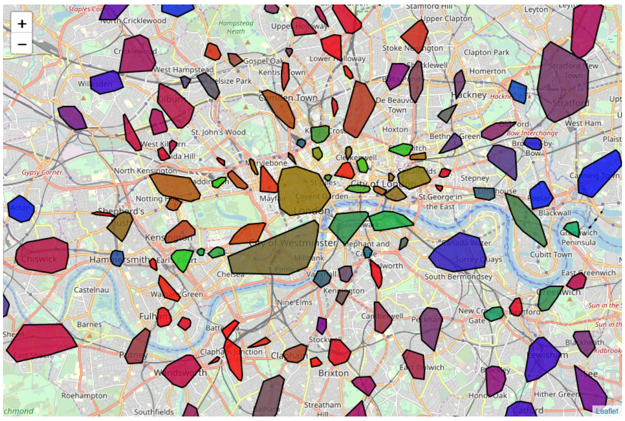


Figure 4: Example of extracted AOIs in London, with colors determined via topic modelling.

Moreover, it allows for intuitive visualizations of the extracted AOIs based on their topic distributions. An illustrative example is provided in Figure 4, where LDA was used to extract 3 topics for a given set of AOIs in London, and then each AOI was assigned a color by determining the RGB values based on its mixture of these 3 topics. Such visualizations allow not only to depict where AOIs are located, but also to intuitively identify similarities and differences between AOIs according to the underlying mixture of different types and attributes of POIs they contain.

6.3 Implicit POI Clustering

We have also developed strategies to group POIs together according to thematic, contextual and/or temporal considerations. For that purpose, we have integrated into SLIPO's toolkit the Power Iteration [19] and K-Means [20] clustering algorithms into Sansa[20] [18] since this semantic-web open-source stack provides these algorithms for RDF data out-of-the-box. We first process the input RDF data using Sansa and then apply the clustering methods. To facilitate the explanation of the algorithms, we use the following example data as input and give the corresponding results. Let's consider three POIs and their associated categories: $P_1(A, B, C)$, $P_2(A, C, D)$ and $P_3(A, B, D)$.

Power Iteration Clustering (PIC). In general, given a set of data points, we could represent the similarity between each pair of data points with a similarity matrix. For example, we could use Jaccard Similarity between two sets of categories that belong to corresponding POIs to represent the similarity between them. Therefore, a similarity matrix S, a diagonal matrix D and a Laplacian matrix $L = D - S$ are created. In Spectral Clustering [35], a subspace matrix consisting of eigenvectors corresponding to the smallest K eigenvalues was derived from the Laplacian matrix. The subspace matrix indicates the clustering results with K clusters. In PIC, the subspace matrix is an approximation to an eigenvalue-weighted linear combination of all the eigenvectors of a normalized similarity matrix [19]. The subspace matrix also indicates the clustering results. In order to prepare the correct input to the algorithm, we first collect the categories for each POI, and then we compute the Jaccard Similarity between category sets for each pair of POIs. Afterwards, we construct a similarity matrix which could be used by PIC algorithm.

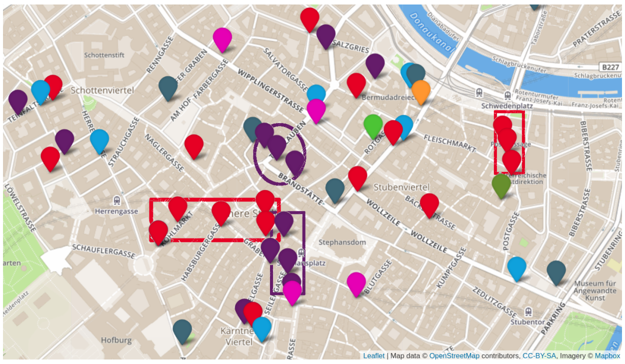


Figure 5: Example of AOIs in Vienna obtained after thematic and spatial clusterings.

K-means. This algorithm partitions POIs into different clusters such that POIs within each cluster have the smallest distance to the cluster centroid compared to placing them into any other cluster. K-means requires a distance metric to represent the distance between POIs. We use the following:

- One *hot encoding* converts categorical values into numerical vectors. Going back to our example, P_1 would be encoded into $(1, 1, 1, 0)$ since it belongs to categories A, B, C and not D.
- *Multidimensional scaling* [34] maps a distance matrix to some vectors in certain dimensions. The relative distances between different POIs are kept.
- *Word2Vec* [21] creates vector representations of words in a text corpus, which here concerns the set of all categories.

Since these methods do not consider the location of the POIs but only focus on implicit links, they produce thematic groups of POIs which are not necessarily geographically close. Indeed, since the groups are computed using categories, clusters might contain POIs sharing the exact same set of categories but actually located in different regions. For instance, Figure 5 represents the results when running PIC over POIs in Vienna. Pins having the same color are part of the same cluster. As expected, it appears that POIs of the same thematic cluster are distributed at various parts of the city center. Then, to get usable AOIs, we pipe together the two considered kinds of clustering to sub-group the thematic clusters according to their geographical locations. Thanks to that strategy, we ensure that the resulting AOIs are composed by POIs that are thematically related. This example demonstrates that the combination of these two approaches allows us to refine the thematic clusters and thus obtain the four AOIs depicted on map.

7 PROTOTYPE IMPLEMENTATION

We have been implementing a comprehensive open source software prototype aiming to support stakeholders in all stages of the POI data value chain. This prototype integrates all aforementioned tools for transforming, linking, fusing, enriching, and analyzing linked POI data. The SLIPO system (currently in beta version[21]) consists of the following main modules:

- *SLIPO Toolkit*: This is a collection of the individual software components (Section 5) applied in quality-assured POI data integration: transformation (TripleGeo), interlinking (Limes), fusion (Fagi), enrichment (Deer) and analytics (Sansa). These software components can be either installed locally or invoked

[20] https://github.com/SANSA-Stack

[21] All software is publicly available at https://github.com/SLIPO-EU

as part of the SLIPO Workbench and APIs functionality explained next.

- *SLIPO Workbench*: This is a web application, which integrates the Toolkit components to implement POI data integration workflows in a coherent, simple to use, and flexible manner. More specifically, it provides utilities for (a) uploading, searching and managing POI datasets in several formats, (b) designing, persisting and managing data integration workflows for POI datasets based on the features provided by the SLIPO Toolkit, (c) scheduling and monitoring the execution of the data integration workflows, and (d) visualizing the results of workflow executions.
- *SLIPO APIs*: This is a collection of RESTful HTTP programming interfaces for invoking SLIPO Toolkit component functionality and integrating it into third-party systems. APIs only support the invocation of simple atomic functions (e.g., POI transformation); otherwise the Workbench web application should be used. Both SLIPO Workbench and APIs are exposed through the same web application server.

Our prototype implements a *workflow engine* that executes data integration jobs and a *scheduler* for initializing workflow executions. A workflow consists of several loosely coupled tasks that together constitute a data integration process over POI datasets. A task may invoke an operation implemented by a Toolkit component (e.g., fusion), or perform secondary operations (e.g., prepare configuration files, update metadata, copy files).

The workflow engine and the SLIPO Toolkit components are deployed over a cloud infrastructure. Workbench and APIs exchange messages with the scheduler to execute workflows. The scheduler propagates requests to the workflow engine, which subsequently initiates the execution of one or more tasks. A task is executed either in-process locally on the scheduler host, or remotely using Docker containers. Each Toolkit component is responsible for providing a scalable implementation for the requested operation, inside the context of the running OS process. A Toolkit component that advertises itself as capable of partitioning its input (and, of course, merging its output) can also scale to multiple Docker containers. The scheduler only controls the total amount of resources allocated to a container, enforcing CPU/memory quotas derived from component-specific requirements and input data size.

Thanks to its modular, service-oriented architecture, SLIPO offers stakeholders the option to directly use the provided functionalities following a *Software-as-a-Service* paradigm. Alternatively, they are able to select specific tools to customize, extend and incorporate in their own POI data management workflows according to their specific needs and requirements. We expect that this will allow the rapid uptake of our innovations in a production setting without affecting any operations and processes already in place.

8 USE CASES

We have been extensively testing and evaluating SLIPO in real-world settings, against various POI data assets. These use cases cover diverse domains (geomarketing, tourism, navigation), ensuring that they reflect the requirements of cross-sector, cross-border and cross-lingual POI data integration. Next, we examine the ability of SLIPO to cope with two typical data integration scenarios against real-world POI datasets in two countries.

Table 1: POI datasets tested in the use cases.

	Dataset	*#POIs*	*Geometry*	*Thematic attributes (# in bold)*	*#triples*
Germany	D_1 (vendor)	35640	point	**(14)**: name(s), category, address, contact details	1114598
	D_2 (vendor)	24416	point	**(13)**: name, address, business data (turnover, #employees, etc.)	1098755
	OSM (open)	45750	point, line, polygon	> **25** *tags*: (multi-lingual) names, address, contact details, image, opening hours, operator, etc.	1130220
	GN (open)	7156	point	**(12):** name(s), city, zipcode, text description, last update	161473
Greece	D_3 (vendor)	72373	point	**(13):** bi-lingual names, address, category, contact details	2517030
	OSM (open)	102159	point, line, polygon	> **25** *tags*: (multi-lingual) names, address, contact details, image, opening hours, operator, etc.	2515476

8.1 Validation Settings

The first use case concerns *hotel POIs* in Germany, whereas the second deals with *general POIs* in Greece. Table 1 lists information concerning the datasets in each scenario. Note that data sources in each use case have different schemata, content and quality. Some datasets are crowdsourced such as OpenStreetMap (*OSM*) or GeoNames (*GN*), while others are offered by commercial vendors (D_1, D_2, D_3)[22]. Through SLIPO, we define integration workflows that deliver an output dataset having:

- *More POIs*, i.e., POIs missing from an original dataset are complemented from the other ones.
- Geometry representations get a *more detailed shape*, e.g., polygons obtained from *OSM* can replace (or complement) the original point (lat/lon) locations of certain POIs.
- *Extra thematic attributes* are derived by bringing together information (e.g., fax numbers, opening hours, links to photos, multi-lingual names) across all original data sources.
- Attribute values per POI are *more accurate* and complete, e.g., missing telephone numbers are filled or updated after checking against each available input.

In each use case, the original datasets are first transformed into RDF according to the SLIPO ontology (Section 4) with suitable attribute mappings. The last column in Table 1 indicates the number of resulting RDF triples. Next, each data *integration cycle* handles a pair of RDF datasets, either transformed from original data or intermediate ones derived from previous cycles. As mentioned in Section 3, a cycle involves these successive stages (cf. Figure 2):

- *Linking* two POI datasets to identify matching POI entities. The resulting RDF graph contains `owl:sameAs` links between their respective URIs.
- *Fusion* of the linked datasets into a new one according to several strategies for fusing spatial and thematic properties per POI.
- *Enrichment* of fused data with extra information from DBpedia.

Once data integration is complete, its RDF output passes through our *reverse transformation* service and delivers the integrated dataset in a traditional POI format (e.g., CSV, shapefile) readily expoitable by stakeholders.

Both scenarios were executed on a virtual machine deployed on top of a cloud stack. This VM offers an Intel® Xeon® E-52600 CPU with 16 virtual cores, 32GB RAM, 16GB swap space, and 200GB disk running Linux Ubuntu 16.04 LTS. In each case, we report data sizes, as well as the qualitative measures discussed in Section 5.5. All tests have been conducted with "cold" caches.

[22] Commercial vendors are anonymized for confidentiality.

8.2 Use Case A: Hotels in Germany

The first use case aims to integrate POIs in Germany concerning *hotels*. We assume that a stakeholder maintains a base POI dataset (D_1) and wishes to enrich it with information from other datasets (Table 1) in three successive integration cycles:

(#1) Integrate D_1 with *OSM*;
(#2) Integrate result of cycle #1 with D_2;
(#3) Integrate result of cycle #2 with *GN*.

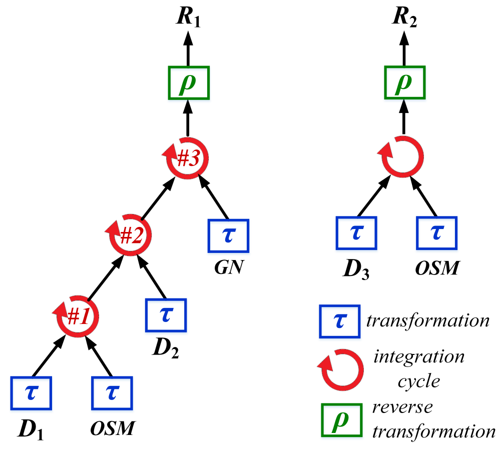
This workflow is illustrated in Figure 6(a). It includes transformation of each input dataset as well as the successive integration cycles (linking, fusion, enrichment). After reverse transformation of the integrated output, the resulting dataset R_1 is obtained. Under this scenario, the stakeholder does not wish to increase the number of POIs in its base data D_1. Instead, it only wants to enhance its content with extra thematic attributes, by filling in missing values and also obtain more detailed geometry representation where available from the other datasets.

Indicative quality and performance measurements for each cycle of Use Case A are listed in Table 2. In this use case, as all data belongs to a specific category (*hotels*), we specified that links between POI profiles should be based strictly on their spatial proximity. This explains the high linking confidence (0.98) in the resulting links. Although this choice can yield erroneous matches (e.g., two hotels may be close, but have different names), we sort them out later during fusion. As no ground truth is available, precision and recall concerning link detection cannot be estimated by Limes. In Table 2, we only provide the respective pseudo-measures (Section 5.5), but they yield rather poor estimates because the source/target datasets involved in linking at each cycle have different sizes (cf. Table 1).

Thanks to its validation rules, Fagi can filter out mismatches not only based on proximity, but also checking with POI names, phone numbers and addresses. The confidence of fused results remains consistently high across all three cycles and demonstrates the similarity among the original POIs that were fused together. At the end of the fusion process, we were able to achieve an increase in the amount of properties per POI, which at the final cycle exceeds 26% on average. Note that there are some POIs where the amount of their attributes increases by up to 47%, acquiring extra properties progressively in each cycle. The more the fused attributes, the faster the confidence stabilizes after each subsequent cycle close to 0.87. It is also important to mention that about a quarter of the POIs get more detailed geometry representations due to integration with the *OSM* data in Cycle #1. Since all other datasets include points only, no further geometric improvement occurs in subsequent cycles. Finally, we stress that the entire workflow concludes in about 6.5 minutes, delivering a unified, richer dataset that otherwise would require considerable human labor.

8.3 Use Case B: POIs in Greece

The second use case concerns general POIs in Greece of *various categories*, i.e., not only hotels as in the previous use case, but also restaurants, cinemas, schools, supermarkets, bus stops, ATMs, etc. As illustrated in Figure 6(b), the goal in this workflow (in one cycle only) is to create a single, integrated dataset R_2 that includes all available information from both input datasets (commercial D_3, open *OSM* data). In particular, apart from richer content (geometries, extra thematic attributes, and filled missing values), the resulting dataset R_2 will grow in size as well, containing many more POIs than any of the original ones.



(a) Hotels in Germany (b) POIs in Greece

Figure 6: Integration workflows for the two use cases.

Table 2: Execution Results.

Measurement	Use Case A			Use Case B
	Cycle #1	*Cycle #2*	*Cycle #3*	
# detected links	33645	15605	6448	29353
Pseudo-precision	0.69	0.81	0.61	0.65
Pseudo-recall	0.57	0.43	0.18	0.22
Pseudo-F-measure	0.62	0.56	0.28	0.33
Avg linking confidence	0.98	0.98	0.98	0.79
# fused pairs	19885	10903	1790	11036
Avg fusion confidence	0.91	0.88	0.87	0.86
Avg attribute gain	0.20	0.26	0.26	0.17
Max attribute gain	0.41	0.47	0.47	0.37
# resulting POIs	35640	35640	35640	159099
# non-point geometries	8728	8728	8728	28236
Execution cost (sec)	224.8	141.4	26.5	823.4

Statistics regarding this workflow are also listed in Table 2. In this use case, interlinking is not just based on spatial proximity but also on POI name similarity in order to avoid matching of possibly dissimilar entities that are next to each other in densely populated areas (e.g., city centers). Due to our relaxed criteria for interlinking, enough potential matches were detected (29353), although with less linking confidence (average: 0.79) compared to the previous use case.

However, we observed that a POI was often linked to multiple other POIs not always having a very similar name, but still close enough in space. During the fusion stage, most of those links were ignored based on our validation rules that also take into consideration more properties (phone, address) in similarity checks. Keeping only 11036 links that were deemed reliable, we derived fused POIs achieving strong confidence (0.86) that they actually concern the same entity. Regarding attribute gain, integrated results denote an average 17% increase in properties per POI. This result may seem poorer compared to Use Case A, but note that crowdsourced content in *OSM* for Greece is less rich than for Germany. Still, the most important outcome of this integration is that the final dataset R_2 includes more than 159 thousand POIs, which is over than double the size of commercial dataset D_3. As

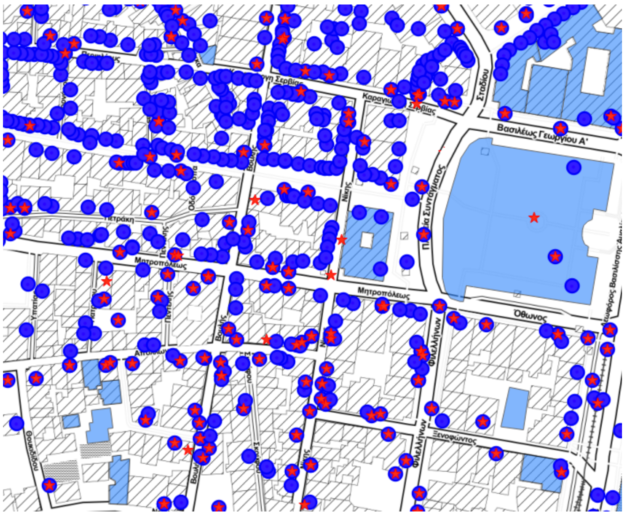


Figure 7: POIs in Athens city center before (red) and after integration (blue).

depicted in Figure 7, integration results (POIs in blue circles) supersede by far and drastically enhance the original information of dataset D_3 (shown with red stars). Furthermore, almost 18% of the resulting POIs now have more detailed geometries (shown as blue polygons) thanks to information extracted from *OSM*. Last, but not least, the fact that this workflow delivers its result in less than 14 minutes, clearly demonstrates the efficiency of SLIPO.

9 CONCLUSIONS AND FUTURE WORK

In this paper, we presented the SLIPO system, a cloud-based application encapsulating Linked Data technologies to efficiently address the challenges of large-scale integration of POI data assets. SLIPO transfers data integration to the Linked Data domain, thus allowing state-of-the-art software to be repurposed and focused to POI data, without requiring domain-specific knowledge from stakeholders or alterations in existing operational workflows. Our tests and evaluations in diverse application domains have shown that SLIPO offers clear advantages in terms of efficient, reliable, quality-assured POI data integration.

Our effort in SLIPO continues along several directions aiming to expand its relevance, efficiency, and value in an industrial setting. First, we will improve the individual software components with additional POI-specific rules and operations to increase performance and effectiveness. Further, we are working with our industrial partners to apply SLIPO in a plethora of domain-specific and cross-border commercial data integration tasks, directly comparing and documenting the gains in productivity, time-to-market, and value. In addition, we are creating periodic world-scale data integration workflows beyond the current reach of the industry, to enable low-cost and streamlined POI-based services. Finally, we are expanding the interoperability of the system to support third-party systems (e.g., signage recognition from street-level imagery) and its quality-assurance services, which will help embed SLIPO in the business workflows of most stakeholders in the value chain.

ACKNOWLEDGMENTS

This work was partially funded by the EU H2020 project SLIPO (#731581).

REFERENCES

[1] A. F. Ahmed, M. A. Sherif, and A.-C. N. Ngomo. On the effect of geometries simplification on geo-spatial link discovery. In *SEMANTiCS*, 2018.

[2] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data – the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[3] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[4] R. J. G. B. Campello, D. Moulavi, and J. Sander. Density-based clustering based on hierarchical density estimates. In *PAKDD*, pages 160–172, 2013.

[5] A. Dimou, D. Kontokostas, M. Freudenberg, R. Verborgh, J. Lehmann, E. Mannens, S. Hellmann, and R. Van de Walle. Assessing and Refining Mappings to RDF to Improve Dataset Quality. In *ISWC*, pages 133–149, 2015.

[6] A. Dimou, M. V. Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. V. de Walle. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In *LDOW*, 2014.

[7] X. L. Dong and D. Srivastava. *Big Data Integration.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2015.

[8] K. Dreßler, M. A. Sherif, and A.-C. Ngonga Ngomo. RADON results for OAEI 2017. In *Proceedings of Ontology Matching Workshop*, 2017.

[9] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[10] European Commission. INSPIRE Directive 2007/2/EC – Infrastructure for spatial information in Europe. https://inspire.ec.europa.eu/, 2007.

[11] K. Feng, G. Cong, S. S. Bhowmick, W. Peng, and C. Miao. Towards best region search for data exploration. In *SIGMOD*, pages 1055–1070, 2016.

[12] K. Georgala, D. Obraczka, and A.-C. Ngonga-Ngomo. Dynamic planning for link discovery. In *ESWC*, 2018.

[13] K. Georgala, M. A. Sherif, and A.-C. N. Ngomo. An efficient approach for the generation of allen relations. In *ECAI*, 2016.

[14] G. Giannopoulos, N. Vitsas, N. Karagiannakis, D. Skoutas, and S. Athanasiou. FAGI-gis: A tool for fusing geospatial RDF data. In *ESWC Satellite Events*, 2015.

[15] O. Hassanzadeh, K. Q. Pu, S. H. Yeganeh, R. J. Miller, L. Popa, M. A. Hernández, and H. Ho. Discovering linkage points over web data. *Proc. VLDB Endow.*, 6(6):445–456, 2013.

[16] P. Konda, S. Das, P. Suganthan G. C., A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. Naughton, S. Prasad, G. Krishnan, R. Deep, and V. Raghavendra. Magellan: Toward building entity matching management systems. *Proc. VLDB Endow.*, 9(12):1197–1208, 2016.

[17] K. Kyzirakos, D. Savva, I. Vlachopoulos, A. Vasileiou, N. Karalis, M. Koubarakis, and S. Manegold. GeoTriples: Transforming geospatial data into RDF graphs using R2RML and RML mappings. *J. Web Sem.*, 52-53:16 – 32, 2018.

[18] J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, I. Ermilov, S. Bin, N. Chakraborty, M. Saleem, A.-C. Ngonga Ngomo, and H. Jabeen. Distributed semantic analytics using the SANSA stack. In *ISWC Resources Track*, 2017.

[19] F. Lin and W. W. Cohen. Power iteration clustering. In *ICML*, pages 655–662, 2010.

[20] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[21] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.

[22] A.-C. Ngonga Ngomo. Helios – execution optimization for link discovery. In *ISWC*, 2014.

[23] A.-C. Ngonga Ngomo, M. A. Sherif, and K. Lyko. Unsupervised link discovery through knowledge base repair. In *ESWC*, 2014.

[24] Open Geospatial Consortium. OGC GeoSPARQL Standard – A Geographic Query Language for RDF Data. https://portal.opengeospatial.org/files/?artifact_id=47664, 2012.

[25] K. Patroumpas, M. Alexakis, G. Giannopoulos, and S. Athanasiou. TripleGeo: an ETL Tool for Transforming Geospatial Data into RDF Triples. In *EDBT/ICDT Workshops*, pages 275–278, 2014.

[26] K. Patroumpas, N. Georgomanolis, T. Stratiotis, M. Alexakis, and S. Athanasiou. Exposing INSPIRE on the Semantic Web. *J. Web Sem.*, 35:53–62, 2015.

[27] M. Sherif, A.-C. Ngonga Ngomo, and J. Lehmann. Automating RDF dataset transformation and enrichment. In *ESWC*, 2015.

[28] M. Sherif, A.-C. Ngonga Ngomo, and J. Lehmann. WOMBAT - A Generalization Approach for Automatic Link Discovery. In *ESWC*, 2017.

[29] M. A. Sherif, K. Dreßler, P. Smeros, and A.-C. Ngonga Ngomo. RADON - Rapid Discovery of Topological Relations. In *AAAI*, 2017.

[30] M. A. Sherif and A.-C. N. Ngomo. A Systematic Survey of Point Set Distance Measures for Link Discovery. *Semantic Web Journal*, 2017.

[31] M. A. Sherif and A.-C. Ngonga Ngomo. An optimization approach for load balancing in parallel link discovery. In *SEMANTiCS*, 2015.

[32] D. Skoutas, D. Sacharidis, and K. Patroumpas. Efficient progressive and diversified top-k best region search. In *SIGSPATIAL*, pages 299–308, 2018.

[33] R. Speck and A.-C. Ngonga Ngomo. Named entity recognition using FOX. In *ISWC*, 2014.

[34] W. S. Torgerson. Multidimensional scaling: I. theory and method. *Psychometrika*, 17(4):401–419, 1952.

[35] U. von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

Executing Entity Matching End to End: A Case Study

Pradap Konda[1], Sanjay Subramanian Seshadri[2], Elan Segarra[3], Brent Hueth[3], AnHai Doan[3]
[1]Facebook, [2]NetApp, [3]University of Wisconsin-Madison, U.S.A.

ABSTRACT

Entity matching (EM) identifies data instances that refer to the same real-world entity. Numerous EM works have covered a wide spectrum, from developing new EM algorithms to scaling them to building EM systems. But there has been very little if any published work on *how EM is carried out in practice, end to end.* In this paper we describe in detail a case study of applying EM to a particular domain end to end (i.e., going from the raw data all the way to the matches).

Specifically, we describe a real-world application for EM in the science policy research community. We describe how our team (the EM team) interact with the science policy team to carry out the EM process, using PyMatcher, a state-of-the-art EM system developed in the Magellan project at UW-Madison. We highlight the communication between the two teams and the zig-zag nature of the EM process. We identify a set of challenges that we believe arise in many real-world EM projects but that current EM systems have either ignored or are not even aware of. Finally, we provide all data underlying this case study, including labeled tuple pairs and documentation supplied by the science policy team, to serve as a good challenge problem for EM researchers.

1 INTRODUCTION

Entity matching (EM) identifies data instances that refer to the same real-world entity, such as (David Smith, UW-Madison) and (D. M. Smith, UWM). EM has been a long-standing challenge in data management [6, 11], and will become even more important in data science. This is because many data science projects need to integrate data from disparate sources before analysis can be carried out, and such integration often requires EM.

Consequently, EM has received enormous attention (see Section 2). Surprisingly, as far as we can tell, there has been very little if any published work on *how EM is carried out in practice, end to end.* The closest that we can find are works that perform EM in a particular domain, e.g., e-commerce, mobile data, patient records, drugs, etc. But these works focus on developing specialized EM algorithms that exploit the characteristics of the target domain, e.g., exploiting product taxonomies (to match e-commerce products) or the spatio-temporal nature of mobile data (to match phone calls).

In this paper we describe in detail a case study of applying EM to a particular domain, end to end (i.e., going from the raw data all the way to the matches). This case study is quite rich, *and it clearly demonstrates many novel challenges for current EM solutions and systems.* We will soon release all the data underlying this case study (to be available at [20]), so that our community can use it as a challenge problem for EM.

Specifically, in summer 2015 we started the Magellan project at the University of Wisconsin-Madison, to build EM systems. We believe that building practical EM systems is critical for advancing the EM field, the way systems such as System R, Ingres, Hadoop, and Spark are critical for advancing the fields of relational data management and Big Data. Subsequently, we built an EM system called PyMatcher [17], which is quite different from EM systems built so far (see Section 2). We then looked for real-world applications to "test drive" PyMatcher.

To do so, we talked with the science policy research community, which has been building a large "data lake" called UMETRICS. Currently UMETRICS collects data from 24 participating U.S. universities, on research grants and the researchers involved in these grants. They use the data to study questions such as "What are the results of investments in research?" and "How do universities affect the regional economy?", and indeed many studies have been published using UMETRICS data (e.g., [30]).

Building UMETRICS requires matching grants across different datasets. The UMETRICS team (at the University of Michigan) has developed and deployed a rule-based EM workflow, but its accuracy is not satisfactory. So we collaborated with a science policy team at UW-Madison to build a more accurate EM workflow, using PyMatcher.

This work was eye-opening to us. Before starting this work, we already felt that existing EM systems were not powerful enough because they failed to address many challenges that arise in real-world EM [17]. That was why we started the Magellan project. Working with UMETRICS suggested to us that we were probably on the right track, but that real-world EM does raise many more challenges that we have also failed to recognize.

These challenges apply equally well to Magellan and other existing EM systems. We discuss the challenges in detail in Section 13, covering the need to develop how-to guides (that provide detailed guidance to the users on how to carry out EM step by step, end to end), new pain points in EM (e.g., labeling, match definition), the need for different EM solutions for different parts of the data, support for easy collaboration among multiple team members (who are often in different locations), handling changes that inevitably arise during the EM process, managing machine learning "in the wild", the need to use both learning and hand-crafted rules, and the need to design a new type of EM architecture. In summary, we make the following contributions:

- We describe a real-world application for EM in the science policy community. We describe what the users want in terms of EM, how their goals change over time, and what end results they are aiming for.
- We describe in detail how our team (the EM team) interacts with the science policy team (the UMETRICS domain expert team) to carry out the EM process, using PyMatcher, a state-of-the-art EM system developed in our group. We highlight the communication between the two teams and the zig-zag nature of the EM process.
- We identify a set of challenges that we believe arise in many real-world EM projects but that current EM systems have either ignored, or are not even aware of, or have not addressed well. While in this paper we have focused on just one case study, in the past few years we have worked with many more real-world EM cases, and the challenges described here also commonly arise in those cases.

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.45

- Finally, we provide all data underlying this case study, including all the labeled tuple pairs and documentation supplied by the domain expert [20]. This dataset can serve as a good challenge problem for EM researchers.

Overall, we hope that the case study here can contribute to helping EM researchers understand better the challenges of the EM process and develop more effective EM solutions. The UMETRICS data and Jupyter notebooks will be available at [20], and a technical report with far more details will be available at [19].

2 BACKGROUND & RELATED WORK

We now discuss the EM problem, related work, then PyMatcher, which was used to perform EM for the case study.

Entity Matching: This problem, a.k.a. entity resolution, record linkage, etc., has received enormous attention (see [6, 11] for books and surveys). A common EM scenario finds all tuple pairs (a, b) that match, i.e., refer to the same real-world entity, between two tables A and B (see Figure 1). Other EM scenarios include matching tuples within a single table, matching into a knowledge base, matching XML data, etc. [6].

Most EM works develop matching algorithms, exploiting rules, learning, clustering, crowdsourcing, among others [6, 11]. They try to improve the matching accuracy and reduce costs (e.g., run time). Trying to match all pairs in $A \times B$ often takes very long. So users often employ heuristics to remove obviously non-matched pairs (e.g., products with different colors), in a step called *blocking*, before matching the remaining pairs. Several works have addressed scaling up blocking (e.g., [3, 8, 16, 29]), learning blockers [4, 9], and using crowdsourcing for blocking [13] (see [7] for a survey).

Many works have considered EM for a particular domain, such as e-commerce (e.g., [10, 15, 21, 22]), patient records [24], financial entities [12], medical sciences [5], drugs [26], and more. But they do not focus on the entire end-to-end process, the interaction between the EM team and the domain expert team, and on identifying the challenges for current EM systems, as we do in this paper.

In contrast to the extensive effort on EM algorithms, there has been relatively little work on building EM systems. As of 2016 we counted 18 major non-commercial systems (e.g., D-Dupe, DuDe, Febrl, Dedoop, Nadeef), and 15 major commercial ones (e.g., Tamr, Data Ladder, IBM InfoSphere) [6]. Our examination of these systems (see [18]) reveals the following four major problems: these systems do not cover the entire EM pipeline, they are stand-alone monolithic systems and hence they make it very difficult to exploit a wide range of techniques, it is very difficult to write code to "patch" these systems even though this is often necessary in practice, and these systems provide very little guidance to users on how to execute the EM process. These shortcomings motivated us to develop the Magellan project, which we briefly describe next.

The Magellan Project (PyMatcher and CloudMatcher): In the Magellan project we developed an on-premise Python-based EM system called PyMatcher and a cloud-based self-service system called CloudMatcher. In this paper we only used the PyMatcher system, which we briefly describe below.

Compared to current EM systems, PyMatcher is novel in four aspects. (1) It provides how-to guides that tell users what to do in each EM scenario, step by step. (2) It provides tools to help users do these steps; the tools seek to cover the entire EM pipeline, not just matching and blocking as current EM systems do. (3) Tools are built into the ecosystem of data science tools in Python, allowing PyMatcher to borrow powerful capabilities in data cleaning, visualization, learning, etc. (4) PyMatcher provides a powerful scripting environment to facilitate interactive experimentation and quick "patching" of the system. See [14, 17] for more, and see *sites.google.com/site/anhaidgroup/projects/magellan* for code.

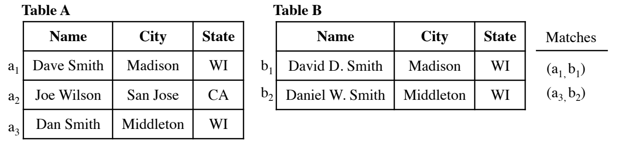


Table A

	Name	City	State
a_1	Dave Smith	Madison	WI
a_2	Joe Wilson	San Jose	CA
a_3	Dan Smith	Middleton	WI

Table B

	Name	City	State
b_1	David D. Smith	Madison	WI
b_2	Daniel W. Smith	Middleton	WI

Matches
(a_1, b_1)
(a_3, b_2)

Figure 1: An example of matching two tables.

3 PROBLEM DEFINITION

We now introduce the science policy research community, their effort to conduct data-driven research by creating a large data lake called UMETRICS, their need to perform EM to build UMETRICS, and the EM problem considered in this paper.

The Science Policy Research Community: A large country such as the U.S. spends hundreds of billions of dollars on science R&D (i.e., research & development) per year. It is important to be able to track the impact of this spending and develop effective science R&D policies. The science policy research community studies such issues.

UMETRICS: To do so effectively, in the past decade, this community has been building a data science infrastructure centering around a large data lake called UMETRICS, which stands for "Universities: Measuring the Impacts of Research and Innovation, Competitiveness and Science" [2].

Currently UMETRICS collects data from 24 participating universities, specifically data on (a) all people paid and all purchases from vendors and subcontracts for all federally funded grants, and (b) the specific job titles and the source of funding for research projects at the universities. As such, the data can be used to study questions such as "What are the results of investments in research?" and "How do universities affect the regional economy?", and indeed many studies have been published using UMETRICS data (e.g., [30]).

The Need for Entity Matching: While highly promising, building UMETRICS require a lot of work to integrate the data submitted by the 24 participating universities. For example, if UW-Madison submits a table that lists research grants from the National Science Foundation (NSF), then those grants must be matched into existing grants already in UMETRICS.

This matching is highly non-trivial. For example, the same research project can have different research titles recorded in UMETRICS and at universities. As another example, a grant given by a funding agency to a research project may be distributed to many smaller projects in the same university, and this information will be recorded in multiple entries in UMETRICS, so matching these entries is not trivial.

The Need to Improve the Current EM Solution: Currently the UMETRICS team (at the University of Michigan) performs such matching using hand-crafted rules. But the accuracy remains unsatisfactory. As a result, a team led by Professor Brent Hueth (in the Department of Agricultural and Applied Economics at UW-Madison and a co-author of this paper) set out to examine how to improve the accuracy of this EM solution.

Matching the USDA Dataset with the UMETRICS Dataset: As a concrete project, they first selected a dataset that contains grants awarded to UW-Madison from USDA (the U.S. Department of Agriculture) in a certain time period. We will refer to this dataset the USDA dataset.

Next, they wanted to match grants in this USDA dataset into UMETRICS. To do so, they performed a simple selection in UMETRICS to select all grants awarded to UW-Madison by USDA in the same time period (UMETRICS allows such selections). For simplicity, let us call this new dataset the UMETRICS dataset. *Next, the team wanted to match grants between the USDA dataset and the* UMETRICS *dataset. Their goal is to find a new EM workflow that is more accurate than the current rule-based EM workflow (deployed in* UMETRICS*).*

This team, which we will call the UMETRICS team, consisted of Professor Brent Hueth, a Ph.D. student in economics, and an hourly student in CS. Initially, we did not get closely involved in the EM process. Instead, we let the UMETRICS team try to match the two datasets themselves. We only provided advice for specific problems such as debugging the matches and evaluating the match results. However, after multiple meetings, we observed that the UMETRICS team simply did not know how to perform EM in a systematic fashion. They had no idea what to do first, what to do second, etc., even though they had a CS student in the team who has learned about EM in a data science class.

So we decided to get involved. Our team, which we will call the EM team, consisted of Professor AnHai Doan, a Ph.D. student in CS, and an hourly student in CS. Our team took over the matching work. We viewed the UMETRICS team as domain experts, and consulted with them in that capacity (by meeting for an hour per week and via emails). Our goal here is to simulate and understand the collaboration between an EM team and a domain expert team, as such settings commonly occur in real-world matching projects. Henceforth "we" refers to the EM team (in CS), when there is no ambiguity.

We decided to use PyMatcher to perform EM, by following its how-to guide (see the PyMatcher's homepage for this guide). In the following sections, we discuss the steps we followed. We intentionally discuss them in details, to show all the steps that human users must perform. We also discuss them in chronological order to show how zig-zag the process was. Finally, we discuss details such as where the files were stored, how the two teams communicated, to highlight the logistic aspects of executing such an EM project in a distributed fashion.

4 UNDERSTANDING THE DATA

We received the raw data from the UMETRICS team in a Google Drive folder. We started by exploring to understand the tables in the datasets, specifically to understand the "entities" in these tables and the relationships among the entities.

We first opened the raw data and found six CSV tables with the "UMETRICS" prefix and one CSV table with the "USDA" prefix. From the table names, we assumed that the names with the "UMETRICS" prefix correspond to UMETRICS-related tables and the name with the "USDA" prefix corresponds to the USDA table.

Next, we explored each table to obtain a brief understanding of the information included in it and the data values of its columns. Specifically, we browsed a few sample rows that were randomly selected from the table and examined general statistics such as the number of unique values, number of missing values, mean, median, etc., for each column.

UMETRICS

Table Name	Num. Rows	Num. Cols
UMETRICSAwardAggMatching	1336	13
UMETRICSEmployeesMatching	1454070	13
UMETRICSObjectCodesMatching	4574	3
UMETRICSOrgUnitMatching	264	5
UMETRICSSubAwardMatching	21470	23
UMETRICSVendorMatching	377746	21

USDA

Table Name	Num. Rows	Num. Cols
USDAAwardMatching	1915	78

Figure 2: Summary of the original UMETRICS and USDA tables given by the UMETRICS team.

Figure 2 lists the number of rows and columns for each table. Figure 3 shows a few rows from UMETRICS tables, and Figure 4 shows a few rows from the USDA table.

To explore the smaller tables (which have fewer than 300K tuples), we used pandas [25] and MS Excel. To explore the larger tables, we used SQLite. To profile the tables, we used the pandas-profiling tool [1] and custom Python scripts.

The UMETRICS Tables: The schemas of the six UMETRICS tables are as follows:

```
UMETRICSAwardAggMatching(UniqueAwardNumber,AwardTitle,
FundingSource,FirstTransDate, LastTransDate,
RecipientAccountNumber,TotalOverheadCharged,
TotalExpenditures,NumberOfTransactions,
DataFileYearEarliest, DataFileYearLatest,
SubOrgUnit, CampusID)

UMETRICSSubAwardMatching(UniqueAwardNumber, Address,
BldgName, City, Country, DUNS, DomesticZipCode, EIN,
ForeignZipCode, ObjectCode, OrgName, OrganizationID,
POBox, PeriodEndDate, PeriodStartDate, RecipientAccountNumber,
SrtName, SrtNumber, State, StrName, StrNumber,
SubAwardPaymentAmount, DataFileYear)

UMETRICSOrgUnitsMatching(CampusId, SubOrgUnit, CampusName,
SubOrgUnitName, DataFileYear)

UMETRICSEmployeeMatching(UniqueAwardNumber, PeriodStartDate,
PeriodEndDate, RecipientAccountNumber,
DeidentifiedEmployeeIdNumber, FullName,
OccupationalClassification, JobTitle, ObjectCode,
SOCCode, FteStatus, ProportionOfEarningsAllocated,
DataFileYear)

UMETRICSVendorMatching(UniqueAwardNumber, PeriodStartDate,
PeriodEndDate, RecipientAccountNumber, ObjectCode,
OrganizationID, EIN, DUNS, VendorPaymentAmount,
OrgName, POBox, BldgNum, StrNumber, StrName, Address,
City, State, DomesticZipCode, ForeignZipCode, Country,
DataFileYear)

UMETRICSObjectCodesMatching(ObjectCode, ObjectCodeText,
DataFileYear)
```

Entities of the UMETRICS Tables: Based on our exploration and profiling of the UMETRICS tables, we inferred the entities of each table as much as we could. "UMETRICSAwardAggMatching" included information about awards, i.e., research grants. It

UMETRICSSubAwardMatching

Unique Award Number	Address	BldgNum	City	Country	DUNS	Domestic ZipCode	EIN	Foreign Zip Code	Object Code	Org Name	Organization ID	POBox	Period End Date	Period Start Date	Recipient Account Number	Srt Name	Srt Number	State	Str Name	Str Number	SubAward Payment Amount	Data File Year
93.395 MSN100683	8701 Watertown Plank Road	NaN	Milwaukee	US	937639060	53226-0509	390806261	NaN	3845	MEDICAL COLLEGE OF WISCONSIN	007H055	NaN	2008-02-21	2008-02-21	MSN100683	NaN	NaN	WI	NaN	NaN	4114	2008
47.078 MSN102120	110 TECHNOLOGY CENTER BLDG	NaN	UNIVERSITY PARK	US	3403953	16802-7000	246000376	NaN	3845	PENNSYLVANIA STATE UNIVERSITY	G067944	NaN	2007-11-05	2007-11-05	MSN102120	NaN	NaN	PA	NaN	NaN	2843.97	2008

UMETRICSAwardAggMatching

UniqueAward Number	AwardTitle	Funding Source	FirstTrans Date	LastTrans DateDate	Recipient Account Number	Total Overhead Charged	Total Expenditures	NumberOf Transactions	DataFile YearEarliest	DataFile YearLatest	SubOrg Unit	Campus ID
00.070 03CS-11231300-031	MAPPING THE 1990 WUI AND 1990-2000 WUI CHNAGE AT THE CENSUS BLOCK LEVEL ACROSS THE UNITED STATES	USDA, FOREST SERVICE	2007-07-01	2008-07-31	MSN106198	0	1296.43	5	2008	2009	7	UWMSN
10.203 WIS01717	Pathogen and host variability in alfalfa with regard to Aphanomyces root rot	HATCH ACT FORMULA FUND	2014-01-01	2016-06-30	MSN158789	0	81171.5	30	2014	2016	7	UWMSN

UMETRICSObjectCodesMatching

ObjectCode	ObjectCode Text	DatFile Year
1000	Salary Default	2008
4880	Domest Govt's Documents	2008

UMETRICSEmployeeMatching

UniqueAward Number	PeriodStart Date	PeriodEnd Date	Recipient Account Number	Deidentified EmployeeId Number	FullName	Occupaitional Classification	JobTitle	Object Code	SOC Code	Fte Status	ProportionOf Earnings Allocated	Data File Year
81.049 DE-FG02-95ER40896	2007-07-01	2007-07-31	MSN103534	B6D6B9B8C09CC8993C7A7EEDFA70970523B2A8BC	CARLSMITH, DUNCAN L	Faculty	Professor	1003	25-1000.0	1	1	2008
00.600 MSN108110	2007-10-01	2007-10-31	MSN108110	37CCA6327EEC785343CE2AF84C02B725534F2127	FIORE, MICHAEL C	Faculty	Professor	1001	25-1000.0	1	0.05	2008

UMETRICSOrgUnitMatching

CampusID	SubOrgUnit	CampusName	SubOrg Unit Name	DatFile Year
UWMSN	37	University of Wisconsin - Madison	Wisconsin Collaboratory for Enhanced Learning	2011
UWMSN	85	University of Wisconsin - Madison	University Housing	2008

UMETRICSVendorMatching

Unique Award Number	PeriodStart Date	PeriodEnd Date	Recipient Account Number	Object Code	Organization ID	EIN	DUNS	Vendor Payment Amount	Org Name	POBox	Bldg Num	Str Number	Str Name	Address	City	State	Domestic ZipCode	Foreign ZipCode	Country	Data File Year
00.500 MSN102793	2008-02-20	2008-02-20	MSN102793	3740	0000000066-1	NaN	NaN	698.8	CAPITAL NEWSPAPERS INC	NaN	NaN	NaN	NaN	PO BOX 9069	MADISON	WI	53708	NaN	US	2008
81.049 DE-FG02-95ER40896	2007-07-13	2007-07-13	MSN103534	3105	0000000066-1	NaN	NaN	49.58	ACE HARDWARE CENTER	NaN	NaN	NaN	NaN	1398 WILLIAMSON ST	MADISON	WI	53703	NaN	US	2008

Figure 3: Example rows from the UMETRICS tables.

Accession Number	Project Title	Sponsoring Agency	Funding Mechanism	Award Number	Initial Award Fiscal Year	Recipient Organization	Recipient DUNS	Project Director	Multistate Project Number	Project Number	Project Start Date	Project End Date	Project Start Fiscal Year	…	…	Financial: USDA Contracts, Grants, Coop Agmt
175763	GENETIC ORGANIZATION AND EPIGENETIC SILENCING OF MAIZE R GENES	State Agricultural Experiment Station	State Funding	NaN	NaN	SAES – UNIVERSITY OF WISCONSIN	NaN	Kermicle, J.L	NaN	WIS04059	1997-07-01	2010-09-30	1997	…	…	NaN
190977	The Changing Location and Extent of the Wildland-Urban Interface During the 1990's	State Agricultural Experiment Station	State Funding	NaN	NaN	SAES – UNIVERSITY OF WISCONSIN	NaN	Hammer, R	NaN	WIS04593	2001-10-01	2011-09-30	2002	…	…	NaN

Figure 4: Example rows from the USDA table.

included funding information for the research projects in the university. "UMETRICSSubAwardMatching" included information about how the awards are split into multiple sub-awards to fund the research projects.

"UMETRICSOrgUnitsMatching" included information about different organization units to which the awards were given. "UMETRICSEmployeeMatching" included information about the employees in the university, and "UMETRICSVendorMatching" included information about the vendors interacting with the university. We were not clear about the information included in the "UMETRICSObjectCodesMatching" table.

Relationships among the UMETRICS Tables: We observed that "UMETRICSAwardAggMatching" was the central table to which most other tables (except "UMETRICSObjectCodesMatching") could be joined using a key-foreign key relationship. For this table (i.e., "UMETRICSAwardAggMatching"), the attribute "UniqueAwardNumber" was the primary key.

We found that the tables "UMETRICSEmployeeMatching," "UMETRICSVendorMatching," and "UMETRICSSubAwardMatching" included the "UniqueAwardNumber" column, which could be joined with the "UniqueAwardNumber" of the "UMETRICSAwardAggMatching" table. The table "UMETRICSOrgUnitsMatching" included a "CampusId" column that could be joined with "CampusId" in the "UMETRICSAwardAggMatching" table. The table "UMETRICSObjectCodesMatching" included an "ObjectCode" column that could be joined with the "ObjectCode" column in the "UMETRICSEmployeeMatching" table.

The USDA Table: Only one table contained USDA information (see Figure 4). This table has 78 columns. Because of space constraints, a partial schema of this table including a few columns is shown below.

```
USDAAwardMatching(AccessionNumber, ProjectTitle,
SponsoringAgency, FundingMechanism, AwardNumber, ...,
ProjectNumber, ProjectStartDate, ProjectEndDate, ...,
ProjectDirector,  ...,
Financial: USDAContracts, Grants, Coop Agmt)
```

5 UNDERSTANDING MATCH DEFINITION

After exploring the tables, we have obtained an understanding of the entities and their relationships. However, we did not know how to use these tables to match the awards (i.e., grants), e.g., which tables are relevant for matching? What does it mean to be a match between the UMETRICS and USDA datasets?

In response, the UMETRICS team sent us a matching document, which discusses the most relevant tables, provides a matching definition, and provides a few sample matching and non-matching record pairs. This document stated that only three tables (among seven) were most relevant for matching and provided the following matching definition.

- (M_1) If a part of "UniqueAwardNumber" in UMETRICS matches "Award Number" in USDA, then the record pair can be considered a match. Specifically, "UniqueAwardNumber" can take the form "XX.XXX YYYY-YYYY-YYYYY-YYYYY". Thus if "YYYY-YYYY-YYYYY-YYYYY" matches the "Award Number," then the record pair is a match. An example of such a match is shown in Figure 5.
- (M_2) A number of records in USDA do not have values for "Award Number." In such cases, the records may be matched by checking whether the "AwardTitle" in UMETRICS and the "Project Title" in USDA are similar. An example of such a match is shown in Figure 6.
- (M_3) A record pair from UMETRICS and USDA can also be matched by comparing the individuals involved in the project.

From the above matching definition, we could infer one positive matching rule based on M_1. Specifically, for a record pair from UMETRICS and USDA tables, if the second part of "UniqueAwardNumber" in the UMETRICS record matches exactly the "Award Number" in the USDA record, then the record pair could be declared a match.

It is possible to use this positive rule to filter out all the positive matches, then proceed with a smaller set for matching. However, we did not do that because we were not sure if the match definition had been stabilized as yet. Instead, we decided to incorporate this rule as a part of the blocking step (as we will see soon).

Apart from M_1, the other two instructions (M_2 and M_3) in the above matching definition are not precise. For example, in M_2, what does it mean to say "similar"? Further, if the award and project titles match exactly, but are very generic (e.g., "Lab Supplies"), then we still cannot conclude that the records match. So we cannot capture these matching instructions as rules, to be applied to the tables to obtain the matches.

As this example suggests, matching definitions are often written in English, in a verbose and imprecise fashion. Business owners often train analysts to perform matching. These analysts gain experience over time and tune their understanding of a "match." They do this by exploring a wide variety of examples and checking with the business owners when in doubt. *This suggests that understanding a matching definition is an iterative process that involves continuous interaction with the business owners.*

6 PRE-PROCESSING THE DATA

Recall that we were given six UMETRICS tables (describing how the funding from different agencies was used for the research projects at UW-Madison) and one USDA table (describing how the funding from USDA was used for research projects at UW-Madison).

UMETRICS		USDA	
field	*value*	*field*	*value*
UniqueAwardNumber	10.200 2008-34103-19449	Accession Number	214335
AwardTitle	DEVELOPMENT OF IPM-BASED CORN FUNGICIDE GUIDELINES FOR THE NORTH CENTRAL STATES	Project Title	Development of IPM-Based Corn Fungicide Guidelines for the North Central States
FirstTransDate	10/1/08	Award Number	2008-34103-19449
FirstTransDate	10/1/08	Project Start Date	8/15/08
		Project End Date	8/14/11
		Project Director	ESKER, PAUL

Figure 5: A matching pair based on Award Number.

UMETRICS		USDA	
field	*value*	*field*	*value*
UniqueAwardNumber	10.203 WIS01040	Accession Number	206746
AwardTitle	SWAMP DODDER (CUSCUTA GRONOVII) APPLIED ECOLOGY AND MANAGEMENT IN CARROT PRODUCTION	Project Title	Swamp Dodder (Cuscuta gronovii) Applied Ecology and Management in Carrot Production
FirstTransDate	10/1/07	Award Number	-
LastTransDate	12/31/08	Project Start Date	10/1/06
		Project End Date	9/30/08
		Project Director	Colquhoun, J.

Figure 6: A matching pair based on Award Title.

Even with this moderate number of tables, the matching process will be difficult if we have to consider all of them. So we decided to *subset the tables* (i.e., selecting only a portion of information from the tables, perhaps even using just a subset of the tables) and *apply transformations* to obtain only two tables that can be used for matching. To do so, we used the matching document provided by the UMETRICS team.

Specifically, we created two tables "UMERICSProjected" and "USDAProjected" (later we will match these two tables), using the following steps.

(1) First, from the six UMETRICS tables, we selected two tables judged most relevant for matching by the UMETRICS team (as discussed in the matching document): "UMETRICSAwardAggMatching" and "UMETRICSEmployeeMatching". (Naturally we also keep the table "USDAAwardMatching".)

(2) Next, we checked that "UniqueAwardNumber" and "Accession Number" were indeed the key columns in the "UMETRICSAwardAggMatching" and "USDAAwardMatching" tables, respectively. We also checked that "UniqueAwardNumber" was indeed a foreign key in the "UMETRICSEmployeesMatching" table with valid values, and could be joined with the "UMETRICSAggAwardMatching" table. We used PyMatcher and pandas to perform these validations.

(3) Then we checked the four remaining UMETRICS tables to see if they contained any information useful for matching. To do this, we manually examined the names of the attributes of these four UMETRICS tables and compared these names to the names of the attributes of the USDA table, to find attribute pairs with similar names. We found that "Recipient Organization" and

"Recipient DUNS" from the USDA table were similar to "OrgName" and "DUNS" in the "UMETRICSVendorMatching" table[1].

Next, we checked if the attributes with similar names have similar values. Specifically, we checked for any overlap of values and compared the distributions of values using mean, median, etc. We did this using pandas and custom Python scripts. We found that the values of "OrgName"and "DUNS" from the "UMETRICSVendorMatching" table did not overlap with the values of "Recipient Organization" and "Recipient DUNS".

We concluded that the four remaining UMETRICS tables do not share any information with the USDA table, and thus are not useful for matching. Thus, we ignore them in the subsequent pre-processing steps.

(4) Finally, we applied transformations to the selected tables. Specifically, we (a) projected out the UMETRICS and USDA tables to create two tables with relevant columns for matching, (b) matched the columns between the tables and renamed them with the same names, and (c) added an ID column to each table. We now elaborate on these steps.

(4.a) First, we projected and kept from the two tables "UMETRICSAwardAggMatching" and "USDAAwardMatching" only attributes that are relevant for matching (we consulted the matching document and the UMETRICS team about which attributes to keep). This produced the following two tables:

```
UMETRICSProjected(UniqueAwardNumber, AwardTitle,
FirstTransDate, LastTransDate)

USDAProjected(AwardNumber, ProjectTitle, ProjectStartDate,
ProjectEndDate, AccessionNumber, ProjectDirector)
```

The "AccessionNumber" was included in the "USDAProjected" table because the UMETRICS team required the output matches to be listed as pairs of "UniqueAwardNumber" and "AccessionNumber."

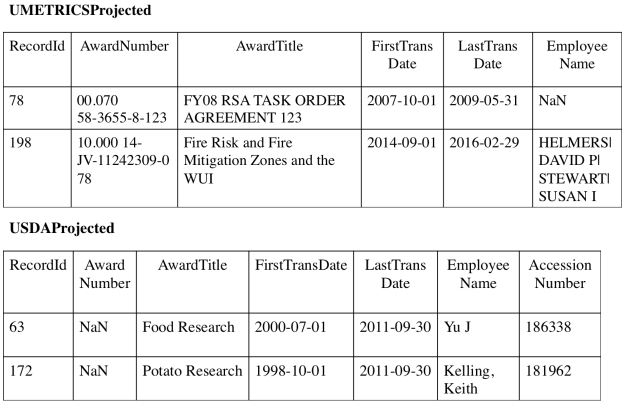
(4.b) Next, we matched the column names between the two tables and renamed them with the same name. Specifically, we matched "UniqueAwardNumber," "AwardTitle," "FirstTransDate," "LastTransDate" from the "UMETRICSProjected" table to "AwardNumber," "ProjectTitle," "ProjectStartDate," "ProjectEndDate," respectively. We named them "AwardNumber," "AwardTitle," "FirstTransDate," "LastTransDate." We renamed "ProjectDirector" in the "USDAProjected" table as "EmployeeName." The updated schemas are shown below.

```
UMETRICSProjected(AwardNumber, AwardTitle,
FirstTransDate, LastTransDate)

USDAProjected(AwardNumber, AwardTitle, FirstTransDate,
LastTransDate, AccessionNumber, EmployeeName)
```

Then we added a new column, "EmployeeName," to the "UMETRICSProjected" table. To do so, we joined this table with the "UMETRICSEmployeesMatching" table on the "AwardNumber" and "UniqueAwardNumber" columns. There were multiple employee names for the same award in the "UMETRICSEmployeesMatching" table. Therefore, for each award, these employee names were concatenated, and each employee name was separated by the | character.

(4.c) Finally, we added an ID column ("RecordId") to both the "UMETRICSProjected" and "USDAProjected" tables, to uniquely identify each record in each table. The final schema of the two tables are (see Figure 7):



UMETRICSProjected

RecordId	AwardNumber	AwardTitle	FirstTrans Date	LastTrans Date	Employee Name
78	00.070 58-3655-8-123	FY08 RSA TASK ORDER AGREEMENT 123	2007-10-01	2009-05-31	NaN
198	10.000 14-JV-11242309-078	Fire Risk and Fire Mitigation Zones and the WUI	2014-09-01	2016-02-29	HELMERS| DAVID P| STEWART| SUSAN I

USDAProjected

RecordId	Award Number	AwardTitle	FirstTransDate	LastTrans Date	Employee Name	Accession Number
63	NaN	Food Research	2000-07-01	2011-09-30	Yu J	186338
172	NaN	Potato Research	1998-10-01	2011-09-30	Kelling, Keith	181962

Figure 7: Sample rows of the UMETRICSProjected and USDAProjected tables.

```
UMETRICSProjected(RecordId, AwardNumber, AwardTitle,
FirstTransDate, LastTransDate, EmployeeName)

USDAProjected(RecordId, AwardNumber, AwardTitle,
FirstTransDate, LastTransDate, AccessionNumber,
EmployeeName)
```

We used pandas, PyMatcher, and custom Python scripts to perform the join and other transformations.

7 BLOCKING

After applying the transformations, the resulting tables, "UMETRICSProjected" and "USDAProjected", have just 1336 and 1915 records, respectively.

Since they are small, can we just match all pairs of records in their Catersian product? In other words, is blocking necessary? It turns out that blocking is still required even in this case. This is because to perform learning-based matching and to evaluate the match results, we must first take a sample from this set (of record pairs in the Cartesian product) and label them.

In our case, however, the Cartesian product of the input tables has 2.5M record pairs, and most of them would be non-matches. Random sampling from this set will result in very few matches. Therefore, we still have to perform blocking to remove obvious non-matching record pairs, so that later when we sample we can obtain more matches in the samples.

We used the matching definition provided by the UMETRICS team to guide the blocking step. We proceeded as follows:

(1) First, we applied a blocking scheme to include all record pairs that satisfy M_1. This is because if M_1 is indeed a positive matching rule, then all record pairs satisfying M_1 must be included in the candidate set (to be fed into the matching step).

Recall that M_1 declares a record pair a match if the second half of the "AwardNumber" attribute of the "UMETRICSProjected" table matches exactly the "AwardNumber" attribute of the "USDAProjected" table. To find all record pairs that satisfy M_1, we applied an attribute equivalence (AE) blocker to these tables. This blocker includes a record pair (in the candidate set) only if the blocking attributes of both input tables agree.

In our case, the AE blocker cannot be applied directly because the "AwardNumber" from "UMETRICSProjected" and "USDAProjected" cannot be compared for an exact match. So we first used a regular expression to extract the suffix of "AwardNumber" of the "UMETRICSProjected" table and stored the result as a temporary column, "TempAwardNumber," in the same table. Then we applied the AE blocker using "TempAwardNumber" from the

[1] What we did here is essentially schema matching [27]. But we did it manually due to the relatively small sizes of the tables.

"UMETRICSProjected" table and "AwardNumber" from the "USDAProjected" table as blocking attributes. Finally, we removed the temporary column ("TempAwardNumber") from the "UMETRICSProjected" table. This blocking scheme produced a candidate set of record pairs C_1.

(2) Next, based on the matching definition M_2, we decided to include the record pairs that have similar award titles. We examined a sample of the award titles in the "USDAProjected" and "UMETRICSProjected" tables, and observed that the titles often have multiple tokens (i.e., words in this case).

Intuitively, two similar award titles should share at least a few tokens. So we applied an overlap blocker to the input tables using "AwardTitle" as the blocking attribute. This blocker discards a record pair if the number of shared tokens (in the blocking attribute) is less than an overlap threshold K.

Specifically, we normalized all the strings in the "AwardTitle" column by lower casing and removing special characters (e.g., single/double quotation marks, hash symbols, exclamation marks, round/curly braces, etc.). Then we performed overlap blocking using a word-level tokenizer, using the overlap threshold 3 after trying a few other thresholds (e.g., the threshold of 1 resulted in 200K record pairs, and a threshold of 7 resulted in a few hundred record pairs). This produced a candidate set C_2.

(3) The overlap blocker drops a record pair if the number of tokens in the blocking attribute was less than the overlap threshold K (in our case, K was 3). So we examined the award titles between the two tables to see if similar titles with fewer than 3 tokens exist, and we found quite a few such title pairs[2].

To include these record pairs, we applied an overlap-coefficient blocker, using "AwardTitle" as the blocking attribute. For any two strings X and Y we have $overlap\_coefficient(X, Y) = |X \cap Y| / \min(|X|, |Y|)$ (assuming X and Y have been tokenized into two sets). This blocker is similar semantics-wise to the overlap blocker. However, it returns a score between 0 and 1, regardless of the title length, and thus can handle the case where a title has fewer than 3 tokens. To apply this blocker, we first lower cased all the strings in the "AwardTitle" column, removed special characters, then performed overlap-coefficient blocking using a word-level tokenizer and a threshold of 0.7 (after trying a few other thresholds). This produced a candidate set C_3.

(4) Next, we unioned C_1, C_2, and C_3 to obtain a consolidated candidate set C, which has 3177 record pairs[3].

We then checked for any potentially missing matches in C using the blocking debugger of PyMatcher [23]. Briefly, this debugger takes the two input tables ("UMETRICSProjected" and "USDAProjected") and the candidate set C, and returns the list of record pairs that (a) are in the Cartesian product of the two tables but not in C, and (b) are judged to be potential matches by the debugger. These pairs are ranked in decreasing likelihood of being matches. If the user does not see many true matches in this list (e.g., by manually examining the top 100 pairs), then he/she can conclude that the blocking process probably has not killed off many true matches. See [23] for details, including how the debugger performs the above process fast.

In our case, we observed that the top record pairs returned by the blocking debugger were not matches, and hence we decided to stop modifying the blocking pipeline. We used PyMatcher for blocking, and used custom Python scripts and pandas commands to preprocess the columns before applying blocking[4].

8 SAMPLING AND LABELING

After blocking, we wanted to obtain and label a sample of record pairs from the candidate set C. (Later we need this labeled sample to select the best learning-based matcher and then train this matcher to predict matches in the candidate set C.)

Sampling and labeling is not straightforward because the candidate set C has relatively few matches and the match definition is still evolving. Further, since we had to ask the UMETRICS team to label, we had to manage the logistics for labeling.

Setting Up: We first emailed the UMETRICS team, then followed up with a face-to-face meeting to discuss the logistics. Initially, we proposed to email the record pairs as a CSV file that they could download and use tools such as MS Excel to label pairs as a match or a non-match. The UMETRICS team wanted a tool with better UI that multiple team members could use. So we developed a simple cloud-based labeling tool with a good UI, but the tool was limited in that only one person could label at any time. The UMETRICS team accepted this and said they would discuss the matter internally and schedule the labeling.

Sampling and Labeling: Next, we performed sampling and labeling iteratively. Our goal was to obtain a sufficient number of positive matches in the labeled set. We first took a random sample of 100 record pairs from the candidate set C, uploaded the sampled pairs into the cloud-based labeling tool, then asked the UMETRICS team to label the record pairs as "Yes", "No", or "Unsure"[5].

The UMETRICS team trained a student to label using the tool. Meanwhile, we labeled the same set of record pairs, using our own understanding of the match definition. Afterward we cross-checked their labels against ours and observed 22 mismatched labels. Specifically, one record pair was labeled a non-match despite satisfying the matching rule M_1. The other 21 pairs had similar award titles, but labeled as a mix of match, non-match, and primarily unsures.

In a face-to-face meeting, the UMETRICS team confirmed that the record pair satisfying M_1 must be declared as match (they also confirmed that M_1 is indeed a positive matching rule). For others, they mentioned that, though the award titles were similar, some of them were not unique enough to be declared matches.

They said that they would have a closer look at the mismatches. After the discussion, we shared the record pairs with mismatched

[2] Specifically, we first used PyMatcher to find all title pairs whose Jaccard score (over a 3-gram tokenization) exceeds a threshold. Next, we kept only those pairs where at least one title has fewer than 3 words. Finally, we took a random sample of these pairs and manually examined the sample.

[3] It turned out that we want both C_2 and C_3 in the consolidated candidate set. C_2 and C_3 have 2,937 and 1,375 record pairs, respectively. $|C_2 \cap C_3|$ = 1,140, $|C_2 - C_3|$ = 1,797, and $|C_3 - C_2|$ = 235. Our examination revealed that if two titles are similar but share few tokens, the overlap blocker will include the pair, but the coefficient blocker will discard it. So we cannot just use C_3 and discard C_2.

[4] PyMatcher's blocking methods use string filtering techniques where appropriate to speed up the blocking process.

[5] It turned out that for many real-world datasets that we have seen, even domain experts had troubles labeling certain pairs, due to dirty, incomplete, or cryptic data. Hence the option "Unsure". As we will see, we ignore "Unsure" pairs in training and evaluating learning-based matchers. Our reasoning is that if even domain experts cannot label these pairs, it is unfair to ask learning algorithms to handle them. And if we can ask learning algorithms to label these pairs, how can we evaluate their accuracy, given that even the domain experts cannot label them?

We also believe that if the number of "Unsure" pairs is too high, that indicates that the dataset is too dirty, incomplete, or cryptic, and the domain experts should clean the dataset before EM is attempted. In this case, as we will see, 300 pairs were labeled for selecting/training matchers, out of which 32 were labeled "Unsure". Later 400 pairs were labeled for evaluating matchers, out of which 16 were labeled "Unsure". The UMETRICS team said they were okay with the quality of this dataset (i.e., judging the EM result from this dataset to be still useful for their domain science research).

labels using Google Sheets. The UMETRICS team updated 4 labels to "Yes" (keeping the labels of the remaining pairs). After the updated labels were submitted, 15 pairs were labeled as "Yes", 66 labeled "No" and 19 labeled "Unsure".

Next, because we had only 15 positive matches, we decided to obtain more labeled pairs. We discussed (via email) and asked the UMETRICS team to label at least 100-200 more record pairs to obtain a sufficient number of positive matches. Following this, we internally decided to obtain labeled record pairs in two iterations, with 100 record pairs per iteration[6]. The first iteration produced 29 "Yes" pairs, 64 "No", and 7 "Unsure". The second iteration produced 24 "Yes", 72 "No", and 4 "Unsure".

We now have 300 labeled pairs, consisting of 68 "Yes", 202 "No", and 30 "Unsure". Because we have obtained a sufficient number of positive matches, we stopped seeking more labeled pairs.

Debugging the Labeled Sample: Next, since the labeled pairs will be used to select and train a matcher, any labeling errors can impact the predicted matches. To minimize this impact, we decided to debug the labels.

To do so, we used leave-one-out cross-validation: we trained an ML matcher on all labeled record pairs except one, applied the matcher to predict the label for the left-out record pair, compared this predicted label with the label given by the UMETRICS team, and repeated the process. We used random forest as the ML matcher, and removed the unsure and sure matches (record pairs that satisfy M_1) from the labeled data before debugging[7]. We observed the following discrepancies:

(D_1) Record pairs were predicted matches, but labeled non-matches if the award titles were very similar but the award title in the "USDAProjected" table included the suffix "NC/NRSP."

(D_2) Record pairs were predicted matches but labeled non-matches if the award numbers were different but the award titles were the same or very similar.

(D_3) Record pairs were predicted matches but labeled a mix of matches and non-matches if the award number was missing from "USDAProjected" but the award titles were very similar.

We shared example record pairs (using Google Sheets) for each of the above discrepancies, and discussed via email and face-to-face meetings with the UMETRICS team. This team (based on their domain knowledge) decided that all labels in D_1 should be updated as unsure (even they did not know if these were matches). All labels in D_2 should be retained. For D_3, the labels must be updated as matches if the transaction dates for the awards are within a difference of few years (e.g., two years). This produced a final set of 300 labeled pairs, consisting of 68 "Yes", 200 "No", and 32 "Unsure."

9 MATCHING

We now use the above set of 300 labeled record pairs to find matches between the "UMETRICSProjected" and "USDAProjected" tables. Specfically, we used this labeled set to select a good learning-based matcher, then applied this matcher to the pairs in the candidate set C (obtained after blocking) to predict matches.

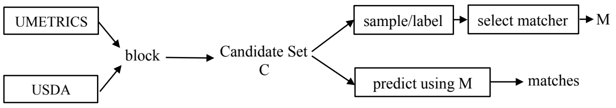


Figure 8: The initial EM workflow for matching the UMETRICS and USDA datasets.

Selecting a Good Matcher: To do this, we used PyMatcher. Let G be the set of 300 labeled pairs. First, we removed the pairs labeled "Unsure" and sure matches (i.e., pairs satisfying M_1) from G, then converted G into a set of feature vectors H (each pair was converted into a feature vector; we used PyMatcher to automatically generate a set of features F, then used F to generate feature vectors).

PyMatcher uses the scikit-learn package, and learning methods in this package cannot work with missing values in the feature vectors. So in the next step we filled in the missing values in the feature vectors in H (with the mean values of the respective columns).

Next, we selected the best (i.e., the most accurate) matcher using five-fold cross validation on H. Among decision tree, SVM, random forest, logistic regression, naive Bayes, and linear regression matchers (supplied by PyMatcher; these matchers in turn use corresponding learning methods in scikit-learn), we found the random forest (RF) matcher had the highest F_1 accuracy (averaged over the five folds).

Next, we debugged the RF matcher to try to improve its accuracy. Again, we used the debugging method described in PyMatcher. This method tried to find the mismatches, i.e., those pairs where the RF matcher predicted incorrectly, then examine these mismatches and take actions to fix them (see [17]). Toward this goal, we randomly split H into two sets I and J, trained the RF matcher on I, then applied it to J and identified mismatches in J, i.e., pairs in J where the label given by the RF matcher differs from the label given in the sampling and labeling process described earlier. We then trained the RF matcher on J and applied it to I, to identify the mismatches in I.

It turned out many mismatches occurred due to award titles having different letter cases. So we added more features to handle this problem[8]. We then performed cross validation to select the best matcher again. Now the decision tree performed the best with 97% precision, 95% recall, and 94.7% F_1, on average. We debugged this matcher using the decision tree matcher debugger of PyMatcher, but were not able to improve accuracy. So we stopped and selected this matcher as the best matcher.

Applying the Selected Matcher: We first trained the decision tree matcher M on the set of feature vectors H (i.e., the set of 300 labeled record pairs). Next, we removed the record pairs satisfying the positive matching rule M_1 from the candidate set C (which was obtained after blocking). There were 210 record pairs that satisfy M_1. Next, we converted the revised set C into a set of feature vectors C' (each pair becomes a feature vector) and imputed the missing values with the mean of the respective columns. Finally, we applied the trained matcher M to C'. This produced 807 matches. Figure 8 shows the overall EM workflow.

Counting also the record pairs that satisfy the positive matching rule, we obtained a total of 1,017 matches. We shared these matches in a CSV file with the UMETRICS team and discussed them in a face-to-face meeting.

[6] Since labeling is time consuming, we wanted to do this in iterations (rather than asking the UMETRICS team to label all 200 pairs in one shot) to make sure we can catch and handle any possible labeling glitches early.

[7] For cross validation, we need to convert each record pair into a feature vector. To do this, we applied PyMatcher to the schemas of the two tables "UMETRICSProjected" and "USDAProjected" to automatically generate a large set of features, which include both string related features (e.g., Jaccard over 3grams, edit distance, etc.) and numeric features (e.g., absolute difference, exact match, etc.). See the PyMatcher homepage for a description of how these features are automatically generated. We then used these features to convert each record pair into a feature vector.

[8] We did not not lowercase all words in the pre-processing step, because our experience with many EM projects suggests that that often resulted in a loss of information. So we only lowercased where necessary.

10 HANDLING MANY COMPLICATIONS

What happened next were many complications regarding whether the current match definition (i.e., what it means to be a match) is incorrect and should be revised, and how to acommodate more data (for the tables), which was accidentally omitted earlier.

Should We Match at the Cluster Level? During the discussion with the UMETRICS team, we observed a gap between their definition of a match between the UMETRICS and USDA awards and our understanding of the same. Specifically, in our set of 1,017 matches, there are many one-to-many matches, e.g., a record in the "UMETRICSProjected" table matches many records in the "USDAProjected" table. The UMETRICS team insisted that matches should be one-to-one, i.e., a record in "UMETRICSProjected" should match at most one record in "USDAProjected".

We were confused by this requirement. Recall from Section 3 that a grant given to a project may be distributed to many smaller projects (e.g., one per an academic year, or one per CS and one per biology) in the same university, and this information will be recorded in multiple entries. For instance, multiple records in "USDAProjected" can contain information about the exact same award, but for different academic years (e.g., a research group may receive a 3-year award; it can send 3 annual reports to USDA, resulting in 3 records in USDA about this award). Given this, a record in "UMETRICSProjected" can match many records in "USDAProjected" and vice versa.

Upon further discussion, it turned out that what the UMETRICS team really wanted is this. As mentioned earlier, a project may last for many years, and may be associated with many sub-awards. Each sub-award is captured by a record in a table. So the UMETRICS team wanted to cluster the records in each table, such that all sub-awards go into the same cluster. They then wanted to match these clusters. At this level, insisting that the matches be one-to-one, i.e., a cluster in "UMETRICSProjected" match at most one cluster in "USDAProjected" makes sense.

Unfortunately, since this desire was not conveyed to us earlier (they were not even aware of this sub-award problem originally), we already performed matching at the record level (not at the cluster level), and at this level one-to-many and many-to-one matches make sense.

We then analyzed the one-to-one, one-to-many, and many-to-one match predictions and shared our analysis with the UMETRICS team. Our goal was to show examples of these and their frequency in the set of matches. Tthe reasoning is that if a problem affects only a small number of matches, then it is not worth spending a lot of effort to solve that problem.

We had another discussion with the UMETRICS team. There they decided that the problem does not affect many matches and so probably would have an insignificant effect on their domain science (which relies on these matches). So they opted to keep matching at the record level, and kept the definition that one record in the "UMETRICSProjected" table can match many records in the "USDAProjected" table, and vice versa. This would avoid a redo of the project, i.e., starting by clustering the records, then trying to match the clusters.

Revising the Match Definition: Recall that the current UMETRICS repository already has a rule-based matching system, and that our goal is to beat that system. To do so, the UMETRICS team logged into UMETRICS and applied the rule-based matching system to the same data (i.e., the "UMETRICSProjected" and "USDAProjected" tables) to obtain matches. The idea was to compute and compare the precision and recall of the rule-based system to those of our system.

During this process, they discovered that another positive matching rule existed: *"If the award number in UMETRICS matches the project number in USDA, then the record pair is considered a match".* This rule could be used to pull more sure matches directly from the "UMETRICSProjected" and "USDAProjected" tables[9].

Thus, the match definition was revised to include this rule. The question is how would this complicate the EM process. The how-to guide of PyMatcher spells out a well-defined sequence of steps for EM (e.g., blocking, sampling, etc.). We trained the students to follow this sequence. By default, a revised match definition would trigger a re-do of this sequence, which is time consuming. For example, blocking can be easily revised by just using the new positive matching rule to add more record pairs to the candidate set C. But sampling and labeling would be time consuming. We need to take a *random* sample of the candidate set C, and since C has changed, we need to re-do the random sampling and label any new pair in the sample (that has not been labeled before). *While labeling a small number of pairs seems trivial, in practice it can take days, as we need to upload the pairs to the cloud-based labeling tool, then wait for the* UMETRICS *team, which often cannot label right away, due to other obligations.*

Thus, we did not want to redo the EM process from scratch. To avoid this, first we checked whether the ML matcher was already learning the above positive rule from the labeled data. Specifically, we checked and found that there were 411 pairs in C that satisfy that rule, and out of those 397 were predicted as matches. Thus, our matcher correctly predicted most of the pairs that satisfy the rule as matches. Next, we checked if blocking discarded any pairs that satisfy the new rule. We found that the Cartesian product of the "UMETRICSProjected" and "USDAProjected" tables contain 473 such pairs, but C included only 411. Thus, blocking discarded too many pairs. So the new positive matching rule does have an effect on the EM process, and something must be done.

Our solution was to leave the current EM workflow alone and create a new EM workflow (which will be used together with the current EM workflow). In this new workflow, we wrote a Python script to apply the new matching rule directly to the input tables "UMETRICSProjected" and "USDAProjected" to obtain the sure matches. *This new EM workflow can be viewed as a "patch" of the current EM workflow.* If a pair is predicted by both the old and new workflows, then we take the prediction of the new workflow. *An advantage of this approach is that we did not have to label any new pairs.*

Handling More Data: When the UMETRICS team inspected the matches produced by the rule-based matcher at UMETRICS, they found new award numbers that they had not seen before. Further inspection revealed that the original table "UMETRICSAwardAggMatching" was incomplete, missing 496 records.

We received these extra records in a CSV file. Revising the table "UMETRICSAwardAggMatching" (to add these records) and redoing the EM process following the PyMatcher's how-to guide would be time consuming (all the steps, blocking, sampling, labeling, matching, etc. must have been redone). So we followed the same strategy used to handle a change in the match definition. *We keep the current EM workflow "as is" or make only minimal changes to it, then "patch" it with new EM workflows.* This way

[9] "AwardNumber" is already in table "UMETRICSProjected". "ProjectNumber" is not in table "USDAProjected". However, it is in table "USDAAwardMatching" and thus can be easily added to "USDAProjected".

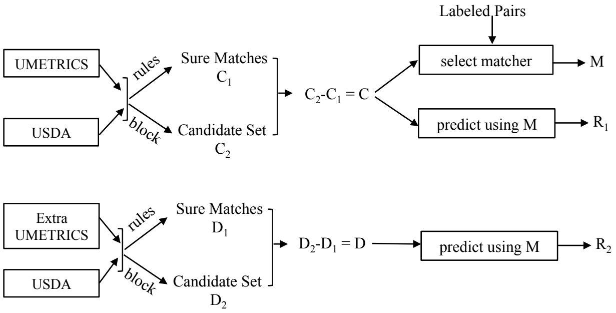


Figure 9: The updated EM workflow to accommodate extra data and positive matching rules.

we minimized the changes we had to make to our existing workflow, and minimized the total human effort. This resulted in the following procedure (see Figure 9):

1. Apply the sure-match rules to the original input tables. Specifically, apply rule M_1 (from the match definition) and the positive matching rule involving award number and project number to obtain a set C_1.
2. Apply blocking as before to obtain a candidate set C_2.
3. Remove the sure matches (C_1) from C_2 to obtain a set C; this set C is what will be predicted as matches and non-matches.
4. Use the labeled set without the sure matches to train the best matcher and predict on C, and call the resulting matches R_1.
5. Repeat Steps 1-3 for the extra records in UMETRICS and the whole USDA table until a set of sure matches D_1 and a candidate set D are obtained.
6. Apply the best matcher obtained using labeled data (in Step 4) to D to get a set of match predictions R_2.
7. Return the union of C_1, D_1, R_1, and R_2 as the set of matches.

The above procedure produced 1,137 matches. Specifically, we first applied the sure-matches rule to obtain 683 sure matches from the original input tables and 55 sure matches from the additional records. Then we applied blocking and removed the sure matches. This resulted in a candidate set from the original input tables having 2,556 record pairs and a candidate set from the extra records having 1,220 record pairs. Next, we removed the sure matches from the labeled set and selected the best matcher, which was a decision tree matcher. Finally, we applied this matcher to the candidate sets and obtained 399 matches from the original tables and no matches from the additional records.

11 ESTIMATING ACCURACY

Now that we had finally obtained the matches in our approach (and praying that no more complications arise), we were ready to estimate the accuracies of our solution and the rule-based matcher deployed at UMETRICS, which we will call the IRIS matcher (IRIS is the organization that manages UMETRICS).

Ideally, if we have the true labels for the record pairs in the Cartesian product of the input tables, then we can compute the accuracy, i.e., precision and recall. However, having the true labels would mean that there was no need to do EM in the first place. To address this, we met with the UMETRICS team and decided to follow the accuracy estimation approach described in the Corleone paper [13]: We proceeded as follows:

(1) To use the Corleone approach, both the IRIS matches and our predicted matches *must be from the same candidate set of record pairs*. So we looked for any award number-accession number pairs from the IRIS matches that were not included in our consolidated candidate set (from the original and extra records), which is $E = C_1 \cup C_2 \cup D_1 \cup D_2$ (see Figure 9). We found only one such pair. The UMETRICS team said that the award number in question was a terminated award (no longer valid) and could be discarded safely.

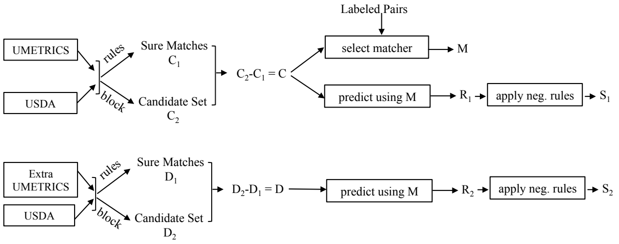


Figure 10: The final EM workflow with negative rules applied to the results of the learning-based matcher.

(2) We took a random sample of 200 record pairs from the consolidated candidate set E, uploaded it to the cloud-based labeling tool, and asked the UMETRICS team to label. Then we used E to estimate precision and recall, using Formulas 2-3 in Section 6.1 of the Corleone paper [13]. We estimated that our matcher had precision in the range (79.6%, 86.01%) and recall in (96.8%, 99.42%). The IRIS matcher had precision in the range (100%, 100%) and recall in (52.7%, 62.07%). Thus, the IRIS matcher had higher precision, but lower recall, compared to our matcher.

(3) To reduce the large interval sizes of the estimated precision and recall, we asked the UMETRICS team to label another 200 randomly sampled record pairs. Applying the same estimation procedure to all 400 labeled pairs[10], we estimated that our matcher had precision in (75.2%, 80.3%) and recall in (98.1%, 99.6%). The IRIS matcher had precision in (100%, 100%) and recall in (65.1%, 71.8%). The UMETRICS team liked the fact that our matcher was able to find more matches than the IRIS matcher.

12 IMPROVING ACCURACY USING RULES

Though we received positive feedback from the UMETRICS team, we had an internal discussion on how to improve the precision of our matcher. We decided to apply hand-crafted rules to the output of our learning-based matcher. The rules would allow us to make "localized changes". This hopefully would improve precision without reducing recall a lot.

Specifically, we wanted to solicit domain-specific rules from the domain experts (the UMETRICS team) to reduce the number of false positives, then apply these rules to the predictions of our learning-based matcher. To do so, we had an email conversation with the UMETRICS team to understand how to reduce the number of false positives. The UMETRICS team examined the predicted matches, then defined a negative rule (i.e., the rule will flip matches to non-matches), which we discuss next.

The Negative Matching Rule: This rule states that a record pair is considered a non-match if one of the following conditions is satisfied:

- Award numbers from UMETRICS and USDA are "comparable," (defined below), and they are not the same.
- Award number from UMETRICS and project number from USDA are "comparable," and they are not the same.

[10] The set of 400 labeled pairs consists of 92 "Yes", 292 "No", and 16 "Unsure". The estimation procedure ignores the "Unsure" pairs.

Here "comparable" means that the award numbers are considered (for this rule) only if they have the same pattern. For example, the UMETRICS award number "03-CS-112313000-031" and the USDA award number "2001-34101-10526" are not comparable because they follow different patterns (i.e., "##-XX-########-###," and "YYYY-#####-#####", respectively, where "#" is any number, "X" is any character, and "YYYY" is a four-digit year).

The UMETRICS award number "WIS01560" and the USDA project number "WIS04509" are comparable because they follow the same pattern "WIS#####" (but because the values are different this pair will be considered a non-match). The UMETRICS team gave us the list of possible patterns for the award numbers from UMETRICS and USDA as well as the project numbers from USDA (not shown for space reasons).

Applying the Negative Matching Rule: We applied the negative rule (provided by the UMETRICS team) to the matches and estimated the accuracy again. Conceptually, the learning-based matcher followed by rules could be considered just another matcher like the IRIS matcher. The updated EM procedure is shown in Figure 10. Here, we applied the negative rules to the sets of matches R_1 and R_2 (obtained from the learning-based matcher). The final set of matches is the union of C_1, D_1, S_1, and S_2.

We consider this new workflow a new matcher. Because the candidate set of this new matcher is the same as that of the learning-based matcher from our previous iteration, we can reuse the labeled set (of 400 pairs). Again, we used the Corleone approach to estimate the new precision and recall.

We found that our new matcher (decision tree followed by negative rules) had precision (96.7%, 98.8%) and recall (94.2%, 97.05%). In contrast, the learning-based matcher (without negative rules) had precision (75.2%, 80.3%) and recall (98.1%, 99.6%). The IRIS matcher had the precision (100%, 100%) and recall (65.1%, 71.8%). Thus, compared to the IRIS matcher, our learning-based matcher followed by rules has slightly lower precision, but much higher recall. The final result set has 845 matches, and was shared with the UMETRICS team in a CSV file that uses "UniqueAwardNumber"and "Accession Number" pairs to capture the matches.

The UMETRICS team was delighted with the result, and their director sent the following email:

> *That is really stupendous news! I'm surprised to see how much you were able to raise the precision and recall ... Thanks for all your brilliant work on this.*

The Next Steps: As the next immediate step, the UMETRICS team wanted us to package the matcher so that they could move it into the UMETRICS repository to do matching for other data slices. It is similar to moving a workflow that was developed in the development stage into production.

This step raises three challenges. First, the EM workflow is rather complex. It has rules at multiple places (to find sure matches and to update the predictions from the learning-based matcher) and a machine learning-based matcher. So we need to find out how to represent it effectively. Second, the new data may be dirty, so we need to monitor the accuracy of the match results[11]. Finally, if the accuracy is not good enough, we need a way to move back to the development stage and update the EM workflow. Currently, we are working with the UMETRICS team to address these challenges.

13 CHALLENGES FOR CURRENT ENTITY MATCHING SOLUTIONS

The end-to-end EM case study that we have just described raises many challenges for the current EM solutions and systems. In what follows we discuss the main challenges[12].

The Need for How-To Guides: It should be clear from the case study that it is extremely hard, if not impossible, to fully automate the EM process, end to end. The fundamental reason is because at the start, the user does not even fully understand the data, the match defition, and even what he or she wants.

Here, for instance, initially the users were not aware that some records were missing, or that the match definition was incomplete, or that they actually wanted to match at cluster level, and more. *As a result, most EM projects are really a "conversation" between the EM team and the domain expert team, and this conversation moves forward as new results were produced and discussed.*

If this is the case, then it follows that it is critical to have some how-to guides that tell both teams how to conduct this conversation, what to do first, what to do second, and so on.

Such guides are completely missing from most current EM solutions and systems. While PyMatcher does have an initial how-to guide, as this case study makes clear, that guide is still quite preliminary. It does not provide guidance to many steps such as how to converge on a match definition, and how to collaboratively label effectively, among others. In practice, guides are likely to be complex, as users want to do so many different things, and complications will arise (as seen in this study).

Many New Pain Points: Current EM work has largely focused on blocking and matching. This study makes clear that there are many pain points, i.e., *steps that require a lot of work from users*, that current EM research has ignored or not been aware of. For example, how to effectively explore, understand, and clean the tables? While data exploration and cleaning have received significant attention, most of this work has been carried out "in isolation", independently of work in EM, based on the implicit argument that these problems are orthogonal to EM. We believe, however, that *these problems should be solved in an EM-centric way*, i.e., we should not just try to understand the tables for understanding's sake, but rather to understand *only things that are important for subsequent EM.*

Other examples of pain points, which require a lot of work from users, include how to quickly converge to a match definition, how to label collaboratively, and how to update an EM workflow if something (e.g., data, match definition) has changed. We argue that more effort should be devoted to addressing these real pain points in practice.

Different Solutions for Different Parts of the Data: The vast majority of current EM works treat the input data as of uniform quality, but in practice, this is rarely the case. Instead, the data commonly contains dirty data of varying degree, incorrect

[11] This is typically done by taking a random sample of the predicted matches at regular intervals, manually labeling it, then using the labeled sample to estimate the accuracy. See [28] for an example of monitoring the accuracy of an e-commerce product classification system in production.

[12] In addition to this case study, in the past 3.5 years we have also worked on more than 20 other EM cases for 12 companies and domain science groups [14]. The challenges that we discuss here are not specific to the case stufy of this paper. They arise in many of these other EM cases as well. Further, even though we used PyMatcher in this case study, given the detailed examination of 33 other free and paid EM tools conducted in [18], we believe many of the challenges discussed here would also arise if we were to use these other EM tools.

data, and incomplete data that even domain experts cannot match. It makes no sense trying to debug the system, then spending more time and money to match incorrect and incomplete data. As a result, it is important to have tools that help the user explore and understand the data, then ways to help the user "split" the data into different parts and develop different EM strategies for different parts of the data, as illustrated in this case study (e.g., see Figure 10).

Support for Easy Collaboration: We found that in many EM settings there is actually a team of people wanting to work on the problem. Most often they collaborate to label a data set, debug, clean the data, etc. For example, in this case study the UMETRICS team collaboratively labeled and helped debug the labeled data. However, most current EM tools are rudimentary in helping users collaborate easily and effectively. Since users often sit in different locations, it is important that such tools are cloud-based, to enable easy collaboration.

Handling Changes Along the Way: No matter how careful we are at the start, changes will likely arise along the way, as the users gain more knowledge and may change their mind. So any effective EM solution need to have good ways to handle such changes. This case study suggested a way to do so, by minimally modifying existing EM workflows and patching them by adding more EM workflows. More research is necessary to evaluate and develop solutions for this problem.

Managing Machine Learning "in the Wild": It is clear that ML can be quite effective. This case study suggests that it can help significantly improve recall while retaining high precision, compared to rule-based EM solutions. But the study also shows that deploying even the simplest ML method "in the wild" raises all kinds of challenges, such as labeling, coping with new data, etc. Further, the study also suggests that *the best EM solutions are likely to involve a combination of ML and rules (such as the negative matching rule in this study).*

Designing EM System Architectures: Finally, this case study raises fundamental questions about what should be the "right" EM system architecture. It implies that a stand-alone monolithic EM system is not likely to work well, because the users often want to try so many different things and many complications often arise.

Observe that in this study, every time there was a complication, we had to devise a new EM workflow and then wrote Python script (that uses PyMatcher's commands) to implement the new workflow. This is more difficult to do with a stand-alone systems. Instead, the study suggests that an "open-world" EM architecture, such as the one PyMatcher has adopted, where the system is a set of tools that interoperate with one another and also with data science tool in PyData, is more promising. But far more studies and research are necessary to settle this question. For a more detailed discussion on this topic, see our recent work [14], which discusses the system aspects of Magellan.

14 CONCLUSIONS

In this paper we have described in detail a case study of entity matching, from raw data to the matches. We have highlighted aspects that previous EM work has ignored. Our case study clearly demonstrates many challenges for current EM work and systems. We hope that this case study on how "sausage is made" can help EM researchers understand better the challenges of the EM process, and thus develop more effective EM solutions.

Acknowledgments: We thank the anonymous reviewers for invaluable suggestions that significantly improve this paper. This work is supported by UW-Madison UW2020 grant and NSF grant IIS-1564282, and by Google, Informatica, Johnson Controls, American Family Insurance, and WalmartLabs.

REFERENCES

[1] [n. d.]. pandas-profiling. https://github.com/pandas-profiling/pandas-profiling.
[2] [n. d.]. Universities: Measuring the Impacts of Research on Innovation, Competitiveness, and Science. https://www.btaa.org/research/umetrics.
[3] Foto N Afrati, Anish Das Sarma, David Menestrina, Aditya Parameswaran, and Jeffrey D Ullman. 2012. Fuzzy joins using MapReduce *(ICDE)*.
[4] Mikhail Bilenko, Beena Kamath, and Raymond J Mooney. 2006. Adaptive blocking: Learning to scale up record linkage *(ICDM)*.
[5] Luiz Fernando Carvalho, Alberto Laender, and Wagner Meira Jr. 2015. Entity Matching: A Case Study in the Medical Domain. *CEUR Workshop Proceedings* 1378 (05 2015).
[6] Peter Christen. 2012. *Data Matching*. Springer.
[7] Peter Christen. 2012. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE TKDE* 24, 9 (2012), 1537–1555. https://doi.org/10.1109/TKDE.2011.127
[8] Xu Chu, Ihab F. Ilyas, and Paraschos Koutris. 2016. Distributed Data Deduplication. *PVLDB* 9, 11 (2016), 864–875.
[9] Anish Das Sarma, Ankur Jain, Ashwin Machanavajjhala, and Philip Bohannon. 2012. An Automatic Blocking Mechanism for Large-scale De-duplication Tasks *(CIKM)*. 10. https://doi.org/10.1145/2396761.2398403
[10] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. 2017. DeepER–Deep Entity Resolution. *arXiv preprint arXiv:1710.00597* (2017).
[11] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate Record Detection: A Survey. *IEEE TKDE* 19, 1 (2007), 1–16.
[12] Mark Flood, John Grant, Haiping Luo, Louiqa Raschid, Ian Soboroff, and Kyungjin Yoo. 2016. Financial entity identification and information integration (feiii) challenge: the report of the organizing committee. In *Proceedings of the Second International Workshop on Data Science for Macro-Modeling*. ACM, 1.
[13] Chaitanya Gokhale, Sanjib Das, AnHai Doan, Jeffrey F Naughton, Narasimhan Rampalli, Jude Shavlik, and Xiaojin Zhu. 2014. Corleone: Hands-off crowdsourcing for entity matching *(SIGMOD)*.
[14] Yash Govind et al. 2019. Entity Matching Meets Data Science: A Progress Report from the Magellan Project. UW-Madison Technical Report.
[15] Anitha Kannan, Inmar E. Givoni, Rakesh Agrawal, and Ariel Fuxman. 2011. Matching Unstructured Product Offers to Structured Product Specifications. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM.
[16] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Dedoop: efficient deduplication with Hadoop. *PVLDB* 5, 12 (2012), 1878–1881.
[17] Pradap Konda et al. 2016. Magellan: Toward Building Entity Matching Management Systems. *PVLDB* 9, 12 (2016), 1197–1208.
[18] Pradap Konda et al. 2016. Magellan: Toward Building Entity Matching Management Systems. Technical Report, http://www.cs.wisc.edu/~anhai/papers/magellan-tr.pdf.
[19] Pradap Konda et al. 2019. Executing Entity Matching End to End: A Case Study. Technical report pages.cs.wisc.edu/~anhai/papers/umetrics-tr.pdf.
[20] Pradap Konda et al. 2019. The UMETRICS Entity Matching Problem: Data, Documentation, and Matches. Available from Magellan's homepage: sites.google.com/site/anhaidgroup/projects/magellan.
[21] Hanna Köpcke, Andreas Thor, and Erhard Rahm. 2010. Evaluation of Entity Resolution Approaches on Real-world Match Problems. *Proc. VLDB Endow.* (Sept. 2010).
[22] Hanna Köpcke, Andreas Thor, Stefan Thomas, and Erhard Rahm. 2012. Tailoring Entity Resolution for Matching Product Offers. In *Proceedings of the 15th International Conference on Extending Database Technology*. ACM.
[23] Han Li et al. 2018. MatchCatcher: A Debugger for Blocking in Entity Matching. In *EDBT*.
[24] Xiaochun Li and Changyu Shen. 2013. Linkage of patient records from disparate sources. *Statistical Methods in Medical Research* 22, 1 (2013), 31–38.
[25] Wes McKinney. 2011. pandas: a Foundational Python Library for Data Analysis and Statistics. In *PyHPC*.
[26] Lee Peters, Joan E Kapusnik-Uner, Thang Nguyen, and Olivier Bodenreider. 2011. An approximate matching method for clinical drug names. In *AMIA Annual Symposium Proceedings*, Vol. 2011. American Medical Informatics Association, 1117.
[27] Erhard Rahm and Philip A. Bernstein. 2001. A survey of approaches to automatic schema matching. *VLDB J.* 10, 4 (2001), 334–350.
[28] Chong Sun et al. 2014. Chimera: Large-Scale Classification using Machine Learning, Rules, and Crowdsourcing. *PVLDB* 7, 13 (2014), 1529–1540.
[29] Rares Vernica, Michael J. Carey, and Chen Li. 2010. Efficient Parallel Set-similarity Joins Using MapReduce *(SIGMOD)*. 12. https://doi.org/10.1145/1807167.1807222
[30] B. Weinberg, J Owen-Smith, R. Rosen, L. Schwarz, B. Allen, R. Weiss, and J. Lane. 2014. Science Funding and Short-Term Economic Activity. *Science* 344, 6179 (2014), 41–43.

The Copernicus App Lab project: Easy Access to Copernicus Data

Konstantina Bereta
National and Kapodistrian University of Athens
Greece
konstantina.bereta@di.uoa.gr

Hervé Caumont
Terradue Srl
Italy
herve.caumont@terradue.com

Ulrike Daniels
AZO Anwendungszentrum GmbH
Germany
Ulrike.Daniels@azo-space.com

Daems Dirk
VITO
Belgium
dirk.daems@vito.be

Erwin Goor*
European Commission
Belgium
Erwin.GOOR@ec.europa.eu

Manolis Koubarakis
National and Kapodistrian University of Athens
Greece
koubarak@di.uoa.gr

Despina-Athanasia Pantazi
National and Kapodistrian University of Athens
Greece
dpantazi@di.uoa.gr

George Stamoulis
National and Kapodistrian University of Athens
Greece
gstam@di.uoa.gr

Sam Ubels
RAMANI B.V.
The Netherlands
sam.ubels@ujuizi.com

Valentijn Venus
RAMANI B.V.
The Netherlands
valentijn.venus@ujuizi.com

Firman Wahyudi
RAMANI B.V.
The Netherlands
firman.wahyudi@ujuizi.com

ABSTRACT

Copernicus is the European programme for monitoring the Earth. It consists of a set of complex systems that collect data from satellites and in-situ sensors, process this data, and provide users with reliable and up-to-date information on a range of environmental and security issues. Information extracted from Copernicus data is made available to users through Copernicus services addressing six thematic areas: land, marine, atmosphere, climate, emergency and security. The data and information processed and disseminated puts Copernicus at the forefront of the big data paradigm and gives rise to all relevant challenges: volume, velocity, variety, veracity and value. In this paper we discuss the challenges of big Copernicus data and how the Copernicus programme is attempting to deal with them. We also present lessons learned from our project Copernicus App Lab, which takes Copernicus services information and makes it available on the Web using semantic technologies to aid its take up by mobile developers. We also discuss open problems for information retrieval, database and knowledge management researchers in the context of Copernicus.

1 INTRODUCTION

Earth observation (EO) is the gathering of data about our planet's physical, chemical and biological systems via satellite remote sensing technologies supplemented by Earth surveying techniques. The Landsat program of the US was the first international program that made large amounts of EO data open and freely available. *Copernicus*, the European programme for monitoring the Earth, is currently the world's biggest EO programme. It consists of a set of complex systems that collect data from satellites and in-situ sensors, process this data and provide users with reliable and up-to-date information on a range of environmental and security issues. The data of the Copernicus programme is provided by a group of missions created by ESA, which is called *Sentinels*, and the *contributing missions*, which are operated by national, European or international organizations. Copernicus data is made available under a free, full and open data policy. Information extracted from this data is also made freely available to users through the *Copernicus services* which address six thematic areas: land, marine, atmosphere, climate, emergency and security.

The Copernicus programme offers myriad forms of data that enable citizens, businesses, public authorities, policy makers, scientists, and entrepreneurs to gain insights into our planet on a free, open, and comprehensive basis. By making the vast majority of its data, analyses, forecasts, and maps freely available and accessible, Copernicus contributes to the development of innovative applications and services that seek to make our world safer, healthier, and economically stronger. However, the potential (in both societal and economic terms) of these huge amounts of data can only be fully exploited if using them is made as simple as possible. Therefore, the straightforward data access every downstream service developer requires must also be combined with in-depth knowledge of EO data processing. The Copernicus App Lab[1] aims to address these specific challenges by bridging the digital divide between the established, science-driven EO community and the young, innovative, entrepreneurial world of mobile development.

*Work performed while at VITO

[1] http://www.app-lab.eu/

Copernicus App Lab is a two year project (November 2016 to October 2018) funded by the European Commission under the H2020 programme. The consortium consists of the company AZO[2] (project coordinator), the National and Kapodistrian University of Athens[3], the companies Terradue[4] and RAMANI[5] and the Flemish research institute VITO[6]. The main objective of Copernicus App Lab is to make Earth observation data produced by the Copernicus programme available on the Web as *linked data* to aid its use by users that might not be Earth observation experts.

Copernicus App Lab targets the *volume* and *variety* challenges of Copernicus data, and it follows the path of previous research projects TELEIOS[7], LEO[8] and MELODIES[9]. Under the lead of the National and Kapodistrian University of Athens, these three projects pioneered the use of linked geospatial data in the EO domain, and demonstrated the potential of linked data and semantic web technologies in the Copernicus setting by developing prototype environmental and business applications (e.g., wild-fire monitoring and burn scar mapping [18, 20], precision farming [9], maritime security [8] etc.).

Copernicus App Lab goes beyond these projects in the following important ways:

- It develops a software architecture that enables on demand access to Copernicus data using the well-known OPeNDAP framework and the geospatial ontology-based data access system Ontop-spatial [5]. Now users and application developers do not need to worry about having to download data or having to learn the details of sophisticated data formats for EO data.
- It brings computing resources close to the data by making the Copernicus App lab tools available as Docker images that are deployed in the Terradue cloud platform as cloud services. The platform allows application developers to access Copernicus data and carry out massively parallel processing without the need to download the data in their own servers and carry out the processing locally.
- It enables search engines like Google to treat datasets produced by Copernicus as "entities" in their own right and store knowledge about them in their internal knowledge graph. In this way, search engines will be able to answer sophisticated users questions involving datasets such as the following: "Is there a land cover dataset produced by the European Environmental Agency covering the area of Torino, Italy?"

A demo paper describing the application scenario presented in this paper won the "best demo award" prize at CIKM 2018 [3]. Shorter presentations of the Copernicus App Lab project also appears in [2]. Compared with the present paper, these papers emphasize only the semantic technologies developed and only the big data dimensions respectively.

The above innovations of Copernicus App Lab are discussed in detail in the rest of the paper, which is organized as follows. Section 2 presents the related work. Section 3 presents the conceptual architecture of the Copernicus integrated ground segment and the software architecture of Copernicus App Lab. Section 4 presents a simple case study which demonstrates the technologies of Copernicus App Lab. Section 5 discusses lessons learned by the use of these technologies and discusses open problems that need to be tackled by future research. Finally, Section 6 concludes the paper.

2 RELATED WORK

Open EO data that are currently made available by the Copernicus and Landsat programs are not following the linked data paradigm. Therefore, from the perspective of a user, the EO data and other kinds of geospatial data necessary to satisfy his or her information need can only be found in different data silos, where each silo may contain only part of the needed data. Opening up these silos by publishing their contents as RDF and interlinking them with semantic connections will allow the development of data analytics applications with great environmental and financial value.

Previous projects TELEIOS, LEO and Melodies funded by FP7 ICT, have demonstrated the use of linked data in Earth Observation. The European project TELEIOS was the first project internationally that has introduced the linked data paradigm to the EO domain, and developed prototype applications that are based on transforming EO products into RDF, and combining them with linked geospatial data. TELEIOS concentrated on developing data models, query languages, scalable query evaluation techniques, and efficient data-management systems that can be used to prototype the applications of linked EO data [18]. The European project LEO was to go beyond TELEIOS by designing and implementing software supporting the complete life cycle of linked open EO data and its combination with linked geospatial data and by developing a precision farming application that heavily utilizes such data [9]. The MELODIES project developed new data-intensive environmental services based on data from Earth Observation satellites, government databases, national and European agencies and more [7]. We focused on the capabilities and benefits of the project's "technical platform", which applied cloud computing and Linked Data technologies to enable the development of services, providing flexibility and scalability.

One of the most important objectives achieved by the aforementioned projects, was capturing the life cycle of open EO data and the associated entities, roles and processes of public bodies that make this data available, and bring the linked data paradigm to EO data centers by re-engineering a complete data science pipeline for EO data [16, 19].

3 THE COPERNICUS APP LAB ARCHITECTURE

Figure 1 presents the conceptual architecture of the *Copernicus integrated ground segment* and the Copernicus App Lab software architecture. A *ground segment* is the hardware and software infrastructure where raw data, often from multiple satellite missions, is ingested, processed, cataloged, and archived. The processing results in the creation of various *standard products* (level 1, 2, and so forth in EO jargon; raw data is level 0) together with extensive metadata describing them.

In the lower part of the figure, the Copernicus *data sources* are shown. These are Sentinel data from ESA, Sentinel data from the European Organisation for the Exploitation of Meteorological Satellites (EUMETSAT), satellite data from contributing missions

[2] http://www.anwendungszentrum.de/
[3] http://kr.di.uoa.gr/
[4] https://www.terradue.com/
[5] https://ramani.ujuizi.com/
[6] https://remotesensing.vito.be/
[7] http://www.earthobservatory.eu/
[8] http://www.linkedeodata.eu/
[9] https://www.melodiesproject.eu/

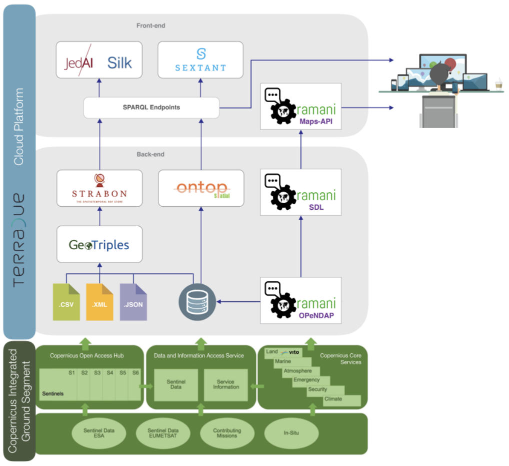


Figure 1: The Copernicus integrated ground segment and the Copernicus App Lab software architecture

(e.g., the RADARSAT-2 mission of Canada or the PROBA-V mission of Belgium) and in-situ data (e.g., data from sensors measuring major air pollutants). The next layer makes Copernicus data and information available to interested parties in three ways: via the Copernicus Open Access Hub, via the Copernicus Core Services and via the Data and Information Access Service (most often known by its acronym DIAS).

The Copernicus Open Access Hub[10] is currently the primary means of accessing Sentinel data. It offers a simple graphical interface that enables users to specify the extent of the geographical area one is interested in (by drawing a bounding box or a polygon). The user may also complete a search interface with information regarding the sensing period, the satellite, the platform, the sensor mode, the polarization etc. If a relevant product is found, the product can be downloaded to the user's computer.

The six core Copernicus services (land, marine, atmosphere, climate, emergency and security) are administered by European entrusted entities (e.g., the Copernicus App Lab partner VITO administers the global land service), whose job is to process Copernicus data and produce higher-level products (*information* in the Copernicus jargon) that are of importance in the corresponding thematic area (e.g., leaf-area index data in the case of the global land service).

The DIAS is not yet fully developed. In December 2017, the European Commission has awarded four contracts to industrial consortia for the development of four cloud-based DIAS platforms. A fifth DIAS is developed by EUMETSAT in collaboration with the French company Mercator Ocean and the European Center for Medium-Range Weather Forecasts (ECWMF). The five DIAS will bring computing resources close to the data and enable an even greater commercial exploitation of Copernicus data. The first versions of the five DIAS are now open to demo-users.

As we have already mentioned in the introduction, the goal of Copernicus App Lab is to make earth observation data produced by the Copernicus programme available as linked data to aid its use by developers that might not be experts in Earth Observation. The software architecture presented in the top two layers of Figure 1 has been designed for achieving this goal. Since none of the DIAS platforms was available when Copernicus App Lab started, all the software components of the project run in the Terradue cloud platform[11]. Terradue Cloud Platform is built as a Hybrid Cloud platform. The primary purpose of the Hybrid Cloud Platform is to facilitate the management of elastic compute resources, with low cost scale-out capabilities. It relies on the concept of an application integration environment (PaaS, or Platform-as-a-Service) and a production environment. Terradue Cloud Platform builds on three major outcomes of the recent developments in Computer Science and Web technology - Cloud Computing, Open Data repositories, and Web Services interoperability. The platform allows cloud orchestration, storage virtualisation, and virtual machine provisioning, as well as application burst-loading and scaling on third-party cloud infrastructures. Within the Terradue cloud platform, the developer cloud sandbox service provides a platform-as-a-service (PaaS) environment to prepare data and processors. It has been designed with the goal to automate the deployment of the resulting EO applications to any cloud computing facility that can offer storage and computing resources (e.g., Amazon Web services). In this manner, the AppLab Cloud Architecture provides the infrastructure (cloud environment) to bring together all the elements of the Copernicus App Lab, and ensure operations.

In Copernicus App Lab, access to Copernicus data and information can be achieved in two ways: (i) by downloading the data via the Copernicus Open Access Hub or the Web sites of individual Copernicus services, and (ii) via the popular OPeNDAP framework[12] for accessing scientific data. In the first case (workflow on the left part of the two top layers of Figure 1), the downloaded data should then transformed into RDF using the tool GeoTriples [23] or scripts written especially for this task. GeoTriples enables the transformation of geospatial data stored in raw files (shapefiles, CSV, KML, XML, GML and GeoJSON) and spatially-enabled RDBMS (PostGIS and MonetDB) into RDF graphs using well-known geospatial vocabularies e.g., the vocabulary of the Open Geospatial Consortium (OGC) standard GeoSPARQL [27]. The performance of GeoTriples has been studied experimentally in [22] using large publicly available geospatial datasets. It has been shown that GeoTriples is very efficient especially when its mapping processor is implemented using Apache Hadoop.

After Copernicus data has been transformed into RDF, it can be stored in the spatiotemporal RDF store Strabon [6, 21]. Strabon can store and query linked geospatial data that changes over time. It has been shown to be the most efficient spatiotemporal RDF store available today using the benchmark Geographica in [6, 15].

Copernicus data stored in Strabon may also be interlinked with other relevant data (e.g., a dataset that gives the land cover of certain areas might be interlinked with OpenStreetMap data for the same areas). To do this in Copernicus App Lab, we use the interlinking tools JedAI and Silk. JedAI is a toolkit for entity resolution and its multi-core version has been shown to be scalable to very large datasets [25]. Silk is a well-known framework for interlinking RDF datasets which we have extended to deal with geospatial and temporal relations [28].

The above way of accessing and using Copernicus data as linked data, has been introduced in previous projects TELEIOS, LEO and MELODIES discussed in the introduction, and it is not the focus of this paper. The *novel way of accessing Copernicus data and information* in Copernicus App Lab is captured by the

[10] https://scihub.copernicus.eu/

[11] https://www.terradue.com/portal/

[12] https://www.opendap.org/

workflow on the right part of the two top layers of Figure 1, and it is based on the popular *OPeNDAP framework* for accessing scientific data. OPeNDAP provides a powerful data translation facility so that users do not need to know the detailed formats of data stored in servers, and can simply use a client that supports a model they are comfortable with. The *streaming data library (SDL)* implemented by RAMANI communicates with the OPeNDAP server and receives Copernicus services data as *streams*. In the rest of this section, we describe how we can access Copernicus data on-the-fly using this workflow.

3.1 Enhance Cataloguing of Copernicus data streams

Copernicus Service Providers (CSP) publish their data holdings in a variety of data formats and access protocols (ftp, http, dap), with various metadata co-existing, and data and metadata either separate or combined in one container. In order to enrich the metadata coupled with the CSP's offerings, we set a minimum metadata standard which should be followed by interested parties in order to streamline the classification, mapping and RDF linkage. We present a mediation approach that facilitates multiple Metadata Standards to co-exist but are semantically harmonized through SPARQL Query. A command-line tool was build and published, entitled "DRS-validator", that validates a CSP's datasets exposed through the OPeNDAP interface by checking for compliance with the Data Reference Syntax (DRS) metadata. Given the proliferation of various metadata standards, a tool was developed that can translate between metadata conventions. In order to harvest the metadata, a Content Management System (CMS) was developed and published as a service allowing the CSP's to manage the metadata of their datasets, which allows them to mutate as and when they choose to expose them through the DAP. Completeness of metadata can be checked globally at SDL level or at an individual dataset level. Since we also use the netCDF variable attributes and global attributes to perform machine-to-machine communication of metadata, the publishing and then harvesting of metadata from CSPs is recurrent by design. For communicating metadata, we use the NetCDF Markup Language (NcML) interface service. This extends a dataset's OPeNDAP Dataset Attribute Structure (DAS) and Dataset Descriptor Structure (DDS) into a single XML-formatted document. The DDS describes the dataset's structure and the relationships between its variables, and the DAS provides information about the variables themselves. The returned document may include information about both the data server itself (such as server functions implemented), and the metadata and dataset referenced in the URL. To ensure that metadata contributes to the discoverability of a datasets, a tool was implemented that provides recommendations for metadata attributes that can be added to datasets exposed through the DAP to facilitate discovery of those using standard metadata searches.

Depending on the requirements imposed upon us by the other open-data, additional analytical functionality is created to ensure the geospatial data streams are commensurate with these requirements. We added a software layer to the SDL, entitled RAMANI Cloud Analytics, allowing on-the-fly spatial and temporal aggregations such that downstream services may request for derived variables to be returned, such as a long-term (moving) average (summer-time) or spatial central tendency (city-average), ensuring that the return is fully commensurate with the intrinsic requirements of the other data to be linked. The technology being used in this stage consists of background IP (BIPR) of RAMANI B.V., with the development of an additional layer for supporting Open Linked Data and Web Coverage Service (WCS). We designed a backend solution capable of providing analytics on data via linked relations. Basic analytical modelling processes can be represented straightforwardly by starting with data and do calculations/analysis that result in more data, which we eventually synthesize into a pithy result, like part of an App or a presentation. Also, the analysis can easily be rerun, if, for example, the data is extended in time, otherwise modified, or is replaced by a different data source providing similar variables based on semantically provided heuristics (e.g. based on "hasName" or "hasUnit"). To allow fast processing we developed a set of containers of the server-side analytics package. Then we used Kubernetes[13] for managing the containerized applications across multiple hosts, that provides the mechanisms for deployment, maintenance, and scaling of the RAMANI Cloud Analytics backend services. Kubernetes builds upon a decade and a half of experience at Google running production workloads at scale using a system called Borg, combined with best-of-breed ideas and practices from the community.

The SDL is also extended with Semantic Web standards providing a framework for explicitly describing the data models implicit in EO and other GRIDded data streams to enhance downstream display and manipulation of data. This provides a framework where multiple metadata standards can be described. Most importantly, these data models and metadata standards can be interrelated, a key step in creating interoperability, and an important step in being able to map various metadata formats in compliance with other standards, e.g. as put forward by the INPIRE Spatial Data Services Working Group. First we developed an Abstract Model capable of providing the basis of the semantics for the linked RDFS. Based on a proof-of-concept use case, an example of a RDF/XML expression for a remote OPeNDAP dataset[14] sourced from the Copernicus Land Monitoring programme was created. This dataset is published using a local deployment of the DAP at VITO. We then implemented a vetted RDF crawler that handles non-standard metadata and supports reasoners, query languages, parsers and serializers. The query languages can create new triples based on query matches (CONSTRUCT) and reasoners create virtual triples based on the stated interrelationships, so we have a framework for creating crosswalks between metadata standards, as well as creating code that is independent of the metadata standards. Finally, in case metadata at the source cannot be made compliant with ACDD, the CMS will allow for post-hoc augmentation using NcML blending metadata provided by the source and those required as-per the DRS validator.

In the current version of the Copernicus App Lab software, VITO -as prime contractor of the Copernicus Global Land Service-has engaged with RAMANI and now provides access to the Copernicus Global Land Service using a remote data access protocol facilitating discovery, viewing, and access of their unique Land Monitoring data-products. OPeNDAP and SDL are installed and configured by VITO on a virtual machine running on the VITO hosted PROBA-V mission exploitation platform[15], which has direct access to the data archives of the Copernicus global land service. The original data sources (i.e. Leaf Area Index, from PROBA-V) have been added to the Streaming Data Library (SDL) so their temporal and spatial characteristics are exposed in a queryable manner. Three Copernicus datasets as well as one

[13] http://kubernetes.io

[14] http://land.copernicus.eu/global/products/lai

[15] https://proba-v-mep.esa.int

Proba-V dataset were configured in the initial setup. These are BioPar BA300 (Burnt Area), LAI (Leaf Area Index) and NDVI (Normalised Difference Vegetation Index). The S5 (five-daily composite) TOC (Top of Canopy, i.e. after atmospheric correction) NDVI 100M was supposed to be implemented as a Proba-V dataset. Each dataset also contains a netCDF NCML aggregation, which is automatically updated when new data (a new date) becomes available. Three different services are exposed for each dataset: the OPeNDAP service, the NetcdfSubset service and the NCML service. The installation of OPeNDAP was done using Docker and access to the Copernicus global land and PROBA-V datasets via OPeNDAP is realised by mounting the necessary disks on the virtual machine.

3.2 Querying Copernicus data on-the-fly using GeoSPARQL

The geospatial ontology-based data access (OBDA) system Ontop-spatial [4] is used to make Copernicus data available via OPeNDAP as linked geospatial data, without the need for downloading files and transforming them into RDF. Ontop-spatial is the geospatial extension of the OBDA system Ontop[16] [10]. OBDA systems are able to connect to existing, relational data sources and create virtual semantic graphs on top of them using ontologies and mappings. An ontology provides a conceptual view of the data. Mappings encode how relational data can be mapped into the terms described in the ontologies. The mapping language R2RML is a W3C standard and is commonly used to encode mappings, but a lot of OBDA/RDB2RDF systems also offer a native mapping language. Since Ontop-spatial follows the OBDA paradigm in the geospatial domain, it can be used to create virtual semantic RDF graphs on top of geospatial relational data sources using ontologies and mappings.

Then, the Open Geospatial Consortium standard GeoSPARQL can be used to pose queries to the data using the ontology. As documented in [4], Ontop-spatial also achieves significantly better performance than state-of-the-art RDF stores. An extension of this work described in [5] shows how the system was extended to support raster data sources as well (support for raster data sources has not been foreseen in the GeoSPARQL standard) and it also describes how, using the OBDA approach, one can query both vector and raster data sources combined in a purely transparent way, without the need to extend the GeoSPARQL query language further.

In the context of the work described in this paper, we extend the OBDA paradigm even more, by enabling an OBDA system not only to connect to a non-relational data source, but also to query data sources that are available remotely, without accessing or storing the data locally (e.g., import the datasets into database); the data can be available through a REST API, for example, and can be accessed by the system only after a GeoSPARQL query is fired. Ontop-spatial has been extended with an adapter that enables it to retrieve data from an OPeNDAP server, create a table view on-the-fly, populate it with this data and create virtual semantic geospatial graphs on top of them. In order to use OPeNDAP as a new kind of data source, Ontop-spatial utilizes the system MadIS[17] as a back-end. MadIS is an extensible relational database system built on top of the APSW SQLite wrapper [11]. It provides a Python interface so that users can easily implement user-defined functions (UDFs) as rows, aggregate functions, or virtual tables. We used MadIS to create a new UDF, named `Opendap`, that is able to create and populate a virtual table on-the-fly with data retrieved from an OPeNDAP server. In this way, Ontop-spatial enables users to pose GeoSPARQL queries on top of OPeNDAP data sources without materializing any triples or tables. We stress that the relational view that is created is not materialized. The intermediate SQL layer facilitates the data manipulation process, in the sense that we can manipulate the data before they get RDF-ized. In this way we can perform “data cleaning” without (i) changing the data that arrive from the server (ii) changing any intermediate code, such as the opendap function and (ii) without requiring any extra pre-processing steps. In order to be able to integrate MadIS as back-end system of Ontop-spatial, apart from implementing the OPeNDAP virtual table function, we also had to extend its jdbc connector and make several modifications in the core of Ontop code so that it allows connections to non-relational data. The design choice behind our approach to implement the OPeNDAP adapter as a virtual table is that it improves the extensibility of the system, as more adapters could be added in the same way by implementing more UDFs in the MadIS system. The integration of the MadIS system to Ontop-spatial is a gateway to support any API in the future that could serve as a new data source to the system.

To improve performance, the OPeNDAP adapter also implements a caching mechanism that stores results of an OPeNDAP call in a cache for a time window w, so that if another, identical OPeNDAP call needs to be performed within this time window, the cached results can be used directly. The length of the time window w is configured by the user in the mappings and it is optional. In the context of the application scenario described in Section 4, we provide more details about the implementation and use of this new version of Ontop-spatial.

3.3 Visualization

Data can be visualized using the tools Sextant [24] or Maps-API.[18] Sextant is a web-based and mobile ready application for exploring, interacting and visualizing time-evolving linked geospatial data. What we wanted to achieve is develop an application that is flexible, portable and interoperable with other GIS tools. The core feature of Sextant is the ability to create thematic maps by combining geospatial and temporal information that exists in a number of heterogeneous data sources ranging from standard SPARQL endpoints, to SPARQL endpoints following the standard GeoSPARQL defined by the Open Geospatial Consortium (OGC), or well-adopted geospatial file formats, like KML, GML and GeoTIFF. In this manner we provide functionality to domain experts from different fields in creating thematic maps, which emphasize spatial variation of one or a small number of geographic distributions. Each thematic map is represented using a map ontology that assists on modelling these maps in RDF and allow for easy sharing, editing and search mechanisms over existing maps.

The Maps-API is similar to Sextant in terms of visualization functionality, but it takes its data from SDL and it cannot deal with linked geospatial data sources accessed by SPARQL or GeoSPARQL. Once data has been discovered, it can be consumed in the VISual Maps-API using any of the following data request-methods: getMetadata, getDerivedData, getMap, getAnimation, getTransect, getPoint, getArea, getVerticalProfile, getSpectralProfile (in case of multi-spectral EO-data), getMapSwipe, and getTimeseriesProfile. This is mainly intended for App Developers who wish to

[16] http://ontop.inf.unibz.it/

[17] https://github.com/madgik/madis

[18] https://ramani.ujuizi.com/maps/index.html

integrate and consume the Copernicus services' products in their favourite mobile platform(s) using straightforward visualization, e.g. as layers on a map-view or as independent graphics (w/o associated geometries on the map).

A more detailed discussion of the linked data tools introduced above and their use in the life cycle of linked Earth observation data is given in the survey papers [14, 17]. All tools are open source and they are available on the following Web page: http://kr.di.uoa.gr/#systems

4 A COPERNICUS APP LAB CASE STUDY

We will now present a simple case study which demonstrates the functionality of the Copernicus App Lab software presented in the previous section. The case study is the same as the scenario presented in the demo paper [3]. However, the presentation here is much more detailed and concentrates on the technical challenges and contributions of the case study.

The case study involves studying the "greenness" of Paris. This can be done by relating "greenness" features of Paris using geospatial data sources such as OpenStreetMap (e.g., for features like parks and forests) and relevant Copernicus datasets. The most important source of such data in Copernicus is the *land monitoring service.*[19] The data provided by this service belongs to the following categories:

- *Global*: The Copernicus Global Land Service (CGLS) is a component of the Land Monitoring Core Service (LMCS) of Copernicus, the European flagship programme on Earth Observation. The Global Land Service systematically produces a series of qualified bio-geophysical products on the status and evolution of the land surface, at global scale and at mid to low spatial resolution, complemented by the constitution of long term time series. The products are used to monitor the vegetation, the water cycle, the energy budget and the terrestrial cryosphere. These datasets are provided by the Copernicus App Lab partner VITO via its satellite PROBA-V. They include a series of bio-geophysical products on the status and evolution of the Earth's land surface at global scale at mid and low spatial resolution.
- *Pan-European*: The pan-European component is coordinated by the European Environment Agency (EEA) and produces satellite image mosaics, land cover / land use (LC/LU) information in the CORINE Land Cover data, and the High Resolution Layers. The CORINE Land Cover is provided for 1990, 2000, 2006 and 2012. This vector-based dataset includes 44 land cover and land use classes. The time-series also includes a land-change layer, highlighting changes in land cover and land-use. The high-resolution layers (HRL) are raster-based datasets which provides information about different land cover characteristics and is complementary to land-cover mapping (e.g. CORINE) datasets. Five HRLs describe some of the main land cover characteristics: impervious (sealed) surfaces (e.g. roads and built up areas), forest areas, (semi-) natural grasslands, wetlands, and permanent water bodies. The High-Resolution Image Mosaic is a seamless pan-European ortho-rectified raster mosaic based on satellite imagery covering 39 countries.
- *Local*: The local component is coordinated by the European Environment Agency and aims to provide specific and more detailed information that is complementary to the information obtained through the Pan-European component. The local component focuses on different hotspots, i.e. areas that are prone to specific environmental challenges and problems. It will be based on very high resolution imagery (2,5 x 2,5 m pixels) in combination with other available datasets (high and medium resolution images) over the pan-European area. The three local components are Urban Atlas, Riparian Zones and Natura 2000.
- *Reference data*: Copernicus land services need both satellite images and in-situ data in order to create reliable products and services. Satellite imagery forms the input for the creation of many information products and services, such as land cover maps or high resolution layers on land cover characteristics. Having all the satellite imagery available to cover 39 countries of EEA (EEA39), the individual image scenes have been processed into a seamless pan-European ortho-rectified mosaics. The Copernicus Land Monitoring Service also provides access to Sentinel-2 Global Mosaic service. A lot of in-situ data is managed and made accessible at national level. However, due to issues such as data access and use restrictions, data quality and availability across EEA39 countries, Copernicus services and particularly Copernicus Land Monitoring Service also relies on pan-European in-situ datasets created and/or coordinated at European level. These datasets are needed for the verification and validation of satellite data in the land monitoring service portfolio.

For our case study, the most relevant datasets from the land monitoring service of Copernicus are the leaf-area index dataset (global), the CORINE land cover dataset (pan-European) and the Urban Atlas dataset (local).

Leaf area index (LAI) is a dimensionless quantity that characterizes plant canopies and it is defined as the one-sided green leaf area per unit ground surface area in broadleaf canopies[20]. LAI may range from 0 (bare ground) to 10 (dense coniferous forests). LAI information from the global land service of Copernicus is made available as a NetCDF file giving LAI values for points expressed by their lat/long co-ordinates.

The *CORINE land cover dataset* in its most recent version (2012) covers 39 European countries[21]. Land cover is characterized using a 3-level hierarchy of classes (e.g., olive groves or vineyards) with 44 classes in total at the 3rd level. The minimum mapping unit is 25 hectares for areal phenomena and 100 meters for linear phenomena. It is made available in raster (GeoTIFF) and vector (ESRI/SQLite geodatabase) formats.

The *Urban Atlas dataset* in its most recent version (2012) provides land use and land cover data for European urban areas with more than 100.000 inhabitants[22]. It covers 800 urban areas in 28 European Union countries, the 4 European Union Free Trade Association countries (Switzerland, Iceland, Norway and Lichtenstein), Turkey and the West Balkans. Land cover/land use is characterized by 17 urban classes (e.g., discontinuous very low density urban fabric) with minimum mapping unit 0.25 hectares, and 10 rural classes (e.g., orchards) with minimum mapping unit 1 hectare. It is made available in vector format as ESRI shapefiles.

In addition to the above datasets, our case study utilizes data from OpenStreetMap and the global administrative divisions dataset GADM. OpenStreetMap is an open and free map of the

[19] https://land.copernicus.eu/

[20] https://en.wikipedia.org/wiki/Leaf_area_index

[21] https://land.copernicus.eu/pan-european/corine-land-cover/view

[22] https://land.copernicus.eu/local/urban-atlas/view

whole world constructed by volunteers. It is available in vector format as shapefiles from the German company Geofabrik[23]. For our case study, information about parks in Paris has been taken from this dataset.

GADM is an open and free dataset giving us the geometries of administrative divisions of various countries[24]. It is available in vector format as a shapefile, a geopackage (for SQLlite3), a format for use with the programming language R, and KMZ (compressed KML).

The first task of any case study using the Copernicus App Lab software is to develop INSPIRE-compliant ontologies for the selected Copernicus data. The *INSPIRE directive* aims to create an interoperable spatial data infrastructure for the European Union, to enable the sharing of spatial information among public sector organizations and better facilitate public access to spatial information across Europe[25]. *INSPIRE-compliant ontologies* are ontologies which conform to the INSPIRE requirements and recommendations. The INSPIRE directives provide a set of data specifications for a wide variety of themes. Our purpose is to categorize our datasets into INSPIRE themes, construct an ontology that follows the respective data specification and then extend this generic ontology to create a specialized version in order to model our datasets. Our initial approach was to reuse existing inspire-compliant ontologies , such as the ones described in [26], but since these efforts are not as close to the INSPIRE specifications as we would like to, we decided to create our own INSPIRE-compliant versions, following the data specifications as closely as possible. Our aim is to reuse these ontologies for other datasets that belong to the same INSPIRE themes and also publish them so that others can reuse these ontologies for their geospatial datasets as well.

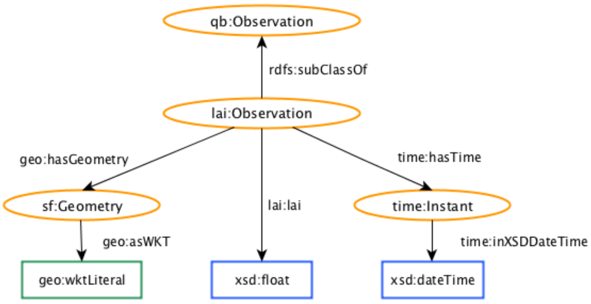


Figure 2: The LAI ontology

A simple ontology for the LAI dataset is shown in Figure 2. We have re-used classes and properties from the Data Cube ontology[26] (namespace `qb`) specializing them when appropriate. We also used classes and properties from the GeoSPARQL ontology [27] (namespaces `sf` and `geo`), from the Time Ontology[27] (namespace `time`), and datatypes from XML-Schema. The class and properties introduced by us use the prefix `lai`.

Once the LAI ontology is created, a user following the workflow depicted on the left, in the Copernicus App Lab software architecture of Figure 1, can use it to transform into RDF the most recent LAI dataset that is made available by the Copernicus global land service[28]. Since GeoTriples does not support NetCDF files as input, the translation was done by writing a custom Python script.

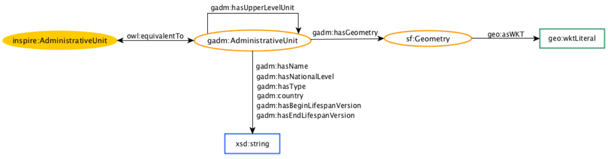


Figure 3: The GADM ontology

Figure 3 shows the ontology that we created for the GADM dataset by extending the GeoSPARQL ontology [27] (namespaces `sf` and `geo`). For the class and properties that we introduced we use the prefix `gadm`[29].The GADM ontology can be used so that a GADM dataset[30] can be either converted into RDF or queried on-the-fly.

A similar process can be followed for datasets CORINE land cover, Urban Atlas, and OpenStreetMap. The ontology which we constructed for the CORINE land cover dataset is not shown here due to space considerations but it is available online [31]. Among other entities (i.e., a class hierarchy corresponding to CORINE land cover categories), it includes the following elements:

- `clc:CorineArea`. This class is a subclass of the class `inspire:LandCoverUnit` of the INSPIRE theme for land cover.
- `clc:hasCorineValue`. This property associates a `clc:CorineArea` with its land cover, as characterised by CORINE.
- `clc:CorineValue`. This class is the range of the property `clc:hasCorineValue` and it is superclass of all the land cover classes of the CORINE hierarchy e.g., `clc:Forests`.

The CORINE land cover ontology can be used to model CORINE land cover data, either materialised as RDF dumps, or virtual RDF graphs created by an OBDA system.

Similarly, we have defined ontologies for Urban Atlas[32], and OpenStreetMap[33]. In the past OpenStreetMap data have been made available in RDF by project LinkedGeoData[34] [29] in the context of which a SPARQL endpoint for OpenStreetMap data was also created. However, the data in this endpoint is not up-to-date (the current version is from 2015), and also this endpoint does not support GeoSPARQL queries. Therefore, we constructed a new ontology for OpenStreetMap by following closely the description of OpenStreetMap data provided by Geofabrik[35] and made it available also in OWL[36]. Using this ontology, we have transformed OpenStreetMap data in shapefiles format into RDF using the tool GeoTriples.

Once all the above datasets are available in RDF, they can be stored in Strabon enabling users to pose interesting, rich queries against the combined dataset. For example, assuming appropriate PREFIX definitions, the GeoSPARQL query shown in Listing 1

[23] http://download.geofabrik.de/
[24] https://gadm.org/
[25] https://inspire.ec.europa.eu/
[26] https://www.w3.org/TR/vocab-data-cube/
[27] https://www.w3.org/TR/owl-time/

[28] https://land.copernicus.eu/global/products/lai
[29] The corresponding namespace is: http://www.app-lab.eu/gadm/
[30] https://gadm.org/data.html
[31] http://pyravlos-vm5.di.uoa.gr/corineLandCover.svg
[32] http://pyravlos-vm5.di.uoa.gr/urbanOntology.svg
[33] http://sites.pyravlos.di.uoa.gr/dragonOSM.svg
[34] http://linkedgeodata.org/About
[35] http://download.geofabrik.de/osm-data-in-gis-formats-free.pdf
[36] http://pyravlos-vm5.di.uoa.gr/osm.owl

retrieves the LAI values of the area occupied by the Bois de Boulogne park in Paris.

Listing 1: LAI in Bois de Boulogne

```
SELECT DISTINCT ?geoA ?geoB ?lai WHERE
{  ?areaA osm:poiType osm:park .
   ?areaA geo:hasGeometry ?geomA .
   ?geomA geo:asWKT ?geoA .
   ?areaA osm:hasName
   "Bois de Boulogne"^^xsd:string> .
   ?areaB lai:lai ?lai .
   ?areaB geo:hasGeometry ?geomB .
   ?geomB geo:asWKT ?geoB .
    FILTER(geof:sfIntersects(?geoA, ?geoB))
}
```

Similarly, in Figure 4, we have used Sextant to build a temporal map that shows the "greenness" of Paris, using the datasets LAI, GADM, CORINE land cover, Urban Atlas and OpenStreetMap. We show how the LAI values (small circles) change over time in each administrative area of Paris (administrative areas are delineated by magenta lines) and correlate these readings with the land cover of each area (taken from the CORINE land cover dataset or Urban Atlas). This allows us to explain the differences in LAI values over different areas. For example, Paris areas belonging to the CORINE land cover class `clc:greenUrbanAreas` overlap with parks in OpenStreetMap and show higher LAI values over time than industrial areas. Paris enthusiasts are invited to locate the Bois de Boulogne park in the figure.

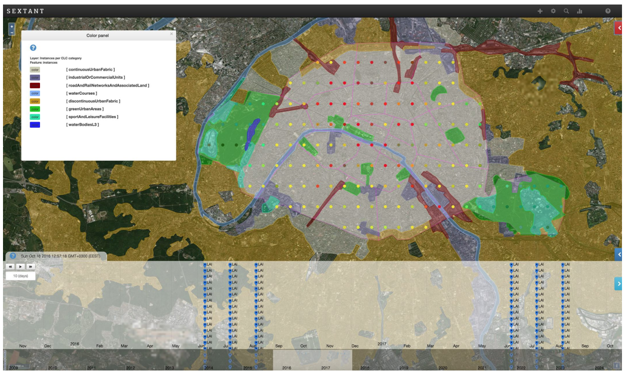

SEXTANT
Color panel
Layer: Instances per CLC category
Feature: instances
[continuousUrbanFabric]
[industrialOrCommercialUnits]
[roadAndRailNetworksAndAssociatedLand]
[waterCourses]
[discontinuousUrbanFabric]
[greenUrbanAreas]
[sportAndLeisureFacilities]
[waterBodiesL3]

Figure 4: The "greenness" of Paris

All RDF datasets that have been discussed above are freely available at the following Web page: http://kr.di.uoa.gr/#datasets

The "greenness of Paris" case study can also be developed using the workflow on the right in the Copernicus App Lab software architecture of Figure 1. In this case, the datasets of interest can be queried using Ontop-spatial and visualized in Sextant without having to transform any datasets into RDF. In this case, the developer has to write R2RML mappings expressing the correspondence between a data source and classes/properties in the corresponding ontology. An example of such a mapping is provided in Listing 2 (in the native mapping language of Ontop-spatial which is less verbose than R2RML).

Listing 2: Example of mappings

```
mappingId opendap_mapping
target lai:{id} rdf:type lai:Observation .
       lai:{id} lai:lai {LAI}^^xsd:float;
                time:hasTime {ts}^^xsd:dateTime .
       lai:{id} geo:hasGeometry _:g .
       _:g geo:asWKT {loc}^^geo:wktLiteral .
source SELECT id, LAI, ts, loc
       FROM (ordered opendap
      url:https://analytics.ramani.ujuizi.com/
      thredds/dodsC/Copernicus-Land-timeseries-
      global-LAI%29/readdods/LAI/, 10)
       WHERE LAI > 0
```

In the example mappings provided in 2, the `source` is the LAI dataset discussed above which provided through the RAMANI OPeNDAP server of the Copernicus App Lab software stack. The dataset contains observations that are LAI values as well as the time and location for each observation. The MadIS operator `Opendap` retrieves this data and populates a virtual SQL table with schema `(id,LAI,ts,loc)`. The column `id` was not originally in the dataset but it is constructed from the location and the time of observation. The LAI column stores LAI values of an observation as `float` values. The attribute `ts` represents the timestamp of an observation in date-time format. In the original dataset times are given as numeric values and their meaning is explained in the metadata. For example, it can be days or months after a certain time origin. The `Opendap` virtual table operator converts these values to a standard format. Because of the fact that the `Opendap` operator is implemented as an SQL user-defined operator, it can be embedded into any SQL query. In the above mapping, we also refine the data that we want to be translated into virtual RDF terms by adding an filter to the query to eliminate (noisy) negative or zero LAI values. The value `10` that is passed as argument to the `Opendap` virtual table operator is the length of the time window w of the cache that is used (in minutes). In this case, if $|w|$ is the length of the time window w, then $|w| = 10$ minutes. This means that results of a every OPeNDAP call get cached every 10 minutes. If a query arrives resulting in an OPeNDAP in time t, where $t < 10$ minutes later than a previous *identical* OPeNDAP call (resulting from a same or similar query that involves the same OPeNDAP call), then the cached results can be used directly, eliminating the cost of performing another call to the OPeNDAP server.

The `target` part of the mapping encodes how the relational data is mapped into RDF terms. Every row in the virtual table describes an instance of the class `lai:Observation` of the LAI ontology in Figure 2. The values of the LAI column populate the triples that describe the LAI values of the observation, and the values of the columns `ts` and `loc` populate the triples that describe the time and location of the observations accordingly.

Given the mapping provided above, we can pose the GeoSPARQL query provided in Listing 3 to retrieve the LAI values and the geometries of the corresponding areas.

Listing 3: Query retrieving LAI values and locations

```
SELECT DISTINCT ?s ?wkt ?lai
WHERE { ?s lai:hasLai ?lai .
        ?s geo:hasGeometry ?g .
        ?g geo:asWKT ?wkt }
```

Using queries like the one described in Listing 3, Sextant can again visualize the various datasets and build layered maps like the one in Figure 4. The visualization of the case study in Sextant is available on line at the following URL: http://test.strabon.di.uoa.gr/SextantOL3/?mapid=m8s4kilcarub1mun_

5 LESSONS LEARNED, FUTURE PLANS AND OPEN PROBLEMS

In this section, we discuss lessons learned, future plans and open problems in the following areas of research and development of the project: the cloud platform, the use of OPeNDAP for accessing global land service and PROBA-V data at VITO, the linked geospatial data technologies and the use of our technologies by users that are not experts in Earth Observation.

The Terradue cloud platform. The use of this platform has provided the Copernicus App Lab project with the ability to manage all software components as cloud appliances, manage releases of the project software stack, deploy on demand this software stack on target infrastructures (e.g., at VITO), monitor operations (by each partner for its part), provide a development and integration environment, and manage solution updates and transfer to operations via cloud bursting. In this way, when the five DIAS will be operational, the Copernicus App Lab software will also be able to run on them. To demonstrate this, we will be working closely with EUMETSAT and Mercator Ocean to make the Copernicus App Lab software run on their DIAS.

The AppLab Cloud solution is operated through both the AppLab Front-end service for Mobile App developers, and the AppLab Back-end service for partnerships with data providers. The concept of operation of the AppLab Cloud solution supports mobile App developers and provides an AppLab capacity "as-a-service" to them. Considering the strength of having several technology providers contributing to the AppLab architecture, the challenge is about how to streamline a deployment scenario that can be replicated on different Copernicus Collaborative Ground Segment (CCGS) partner environments. Each individual deployment must be configured for the target environment, in order to enable some specific data management functions. All together, these deployments also have to be configured on top of the CCGS data repositories, in order to deliver a "value adding" service, aimed at Mobile App developers, over the Copernicus Services data products. Terradue Cloud Platform supports this "AppLab service" integration work, based on the contributed Docker services and on standard protocols, and deploys it "as-a-Service" on a selected target environment. A pre-operational capacity, is providing managed services on Terradue Cloud Platform. For the back-end services, Terradue Cloud Platform supports the deployment (Cloud bursting) of tailored data access services onto the Cloud Computing layer of a selected CCGS provider. The deployment scenario relies on using Terradue Cloud Platform as a reference platform for evolutive maintenance and versioning of the whole AppLab solution, and its updates as new deployments on the selected data providers' environments (e.g. a Copernicus DIAS).

Deploying OPeNDAP at VITO. The use of OPeNDAP offers better data access capabilities specifically for application developers that are not experts in Earth Observation, and thus it a clear benefit. OPeNDAP and SDL provide streaming data to the end user and have some significant advantages over the OGC Web Coverage Service standard which is already offered by VITO. First of all, from a data provider perspective, OPeNDAP is easier to use, as it is able to deal with a wider variety of grid types. Furthermore, OPeNDAP can be easily extended with different conventions, allowing for easier integration of different dataset and without overhead like file conversion. Also, OPeNDAP enables the loose coupling of different Copernicus data sources into one data model, providing the user easy access through a single access point that uses this data model. Finally, when using the Web Coverage Service, there is limited possibility to obtain client-specific parts of the datasets (one is limited to, for example, a bounding-box). In contrast, OPeNDAP allows for the caching of datasets by serialization based on internal array indices. This increases cache-hits for recurrent requests of a specific subpart of the dataset which can be very useful, e.g., in a mobile application scenario, where the viewport of the application could be defaulting to a specific, user-configurable area of interest with only modest panning and zooming interaction. Also, OPeNDAP ensures that metadata is intrinsically embedded in the TCP/IP response, regardless of container type (GeoTIFF, NetCDF, HDF, grib, etc.), which is beneficial for the semantic enrichment process that may happen at higher layers of the architecture. For the remaining duration of the project, the OPeNDAP deployment at VITO will be improved by offering a more sophisticated access control facility.

In order to deploy DAP at VITO we created several Virtual Machines (VMs) to build a private cloud in the VITO data center and used Docker images for the configuration. To provide access to the Copernicus and Proba-V datasets via the DAP, these datasets are mounted on the virtual machine. The data available on these storage volumes will be exposed by the DAP. Copernicus data uses the netCDF extension while tiff extension is used for Proba-V data. During the implementation of the DAP at VITO, it became clear that the directory structure used for the Copernicus Global Land datasets is not supported by the DAP, so we had to create a virtual directory structure to be compliant with the DAP. The reason why the Copernicus dataset could not be supported is that this directory structure contains multiple versions of data for the same day: the production centre reprocesses data at several days when more accurate meteorological data becomes available. The DAP could not handle this deviation, so VITO made a script to create a directory structure that uses symbolic links to point at the most recent version of the data, as that is the only one that needs to be exposed. Also, to ensure security we used tokens that allow accessing the datasets throught the RAMANI API. Every user has to register an account on the RAMANI platform. Without proper registration users will not have any access to the datasets to ensure map uptake monitoring capabilities and to avoid abuse. Furthermore, this will allow the tracking of which users access which datasets.

The linked geospatial data tools. The linked geospatial data tools presented in Section 3 have been used to develop environmental applications not only in Copernicus App Lab but also in previous projects TELEIOS, LEO and MELODIES [14, 17]. The linked geospatial data tools have been welcome by users who could see the value of developing applications using semantic technologies. The most popular tools have been Sextant and Ontop-spatial. Sextant has been irreplaceable as there is currently no other tool for visualizing linked geospatial data. Ontop-spatial has been attractive for users given that most of them were not in favour of transforming their data into RDF and storing it in Strabon. The fact that Ontop-spatial is also faster than Strabon on most of the queries of the benchmark Geographica [4] has been another reason users preferred it over Strabon.

The most innovative, but also challenging, aspect of using Ontop-spatial in Copernicus App Lab has been its ability to give access to Copernicus data (e.g., LAI data) through the OPeNDAP framework. When the data gets downloaded at query-time, query execution typically takes two orders of magnitude more time than in the case where the data is materialized in a database or an RDF store. When data is stored in a database connected with Ontop-spatial, DBMS optimizations and database constraints are taken into account and query plans are optimized. This does not happen in the case where Ontop-spatial retrieves data on-the-fly from OPeNDAP, especially since data is preprocessed before it gets translated into virtual triples using Ontop-spatial. However, in the cases when we want to access Copernicus data that gets frequently updated, the virtual RDF graphs approach is useful as it avoids the repeated translation steps that have to be done by the data provider. For more costly operations (e.g., spatial joins of complex geometries), it is better to materialize the data. In our current work, we are developing further optimizations to improve the performance of this mechanism using techniques such as caching. We are also working on enabling Ontop-spatial to query other kinds of data e.g., HTML tables and social media data, since this functionality has been requested by users.

The following open problems will also be considered in the future:

- Although Strabon has been shown to be the most efficient spatiotemporal RDF store available today[37], much remains to be done for Strabon to scale to the *petabytes* of Copernicus data or the linked geospatial data typically managed by a national cartographic agency (e.g., Ordnance Survey in the United Kingdom or Kadaster in the Netherlands; they both use linked geospatial data). We plan to extend a scalable RDF store like Apache Rya[38] with GeoSPARQL support taking into account the lessons learned by similar projects in the relational world [13].
- It will usually be the case that different geospatial RDF datasets (e.g., GADM and OpenStreetMap) will be offered by different GeoSPARQL endpoints that can be considered a federation. There is currently no query engine that can answer GeoSPARQL queries over such a federation. The only system that comes close is SemaGrow which has been shown to federate a single Virtuoso endpoint and a single Strabon endpoint in [12]; more research is needed in this area for developing a state-of-the-art federation engine for GeoSPARQL.
- It is important to extend GeoTriples to be able to transform data in SQLite and ESRI geodatabases into RDF. The same should be done for scientific data formats such as NetCDF. The problem of representing array/gridded data in RDF has recently received attention by the W3C/OGC Spatial Data on the Web Working Group and some interesting working group notes were produced[39]. In a similar spirit an extension of SPARQL, called SciSPARQL, for querying scientific data has been proposed in the Ph.D. thesis of [1].

Using the Copernicus App Lab tools. Participants of the ESA Space App Camps that were organised in September 2017 and 2018 had the opportunity to use the Copernicus App Lab technologies to implement demo applications. ESA Space App Camps[40] are yearly events that bring together programmers to develop innovative applications based on satellite data. The objective is to make EO data, particularly from Copernicus, accessible to a wide range of businesses and citizens. Twenty-four developers from 14 countries attended the 2017 ESA App Camp in Frascati, Italy. It is important to note that these were the first two ESA Space App Camps that introduced the Copernicus App Lab tools and in both competitions he winning applications utilized the Copernicus App Lab technologies. The winning app of 2017 named AiR, displays an interactive projection of the Earth's surface to airplane travelers using Copernicus satellite imagery, letting them see information about the cities and landmarks they pass over during their flight, without the disruptions of clouds or parts of the plane getting in the way. The developers of AiR used Copernicus App Lab tools to access and integrate data from different sources (Copernicus land monitoring service data, OpenStreetMap data and DBpedia data about landmarks). In 2018 the winning app named Urbansat aims to guide greener, more ecological urban planning. It provides a range of data for planners, including information on green spaces, terrain and biodiversity and more. The app′s map interface has a drag and drop feature, which would allow users to compare scenarios pre and post build for their construction projects, through the generation of relevant data, largely derived from Sentinel satellites. This would allow them to see the anticipated impact of constructing a building on local air quality, for example. The developers of Urbansat used Copernicus App Lab tools to process data from different sources (Copernicus land monitoring service data, Urban Atlas data, Natura 2000 data and data provided from GADM). Another interesting application that was developed was Track Champ. Track Champ combines Earth observation data with data about points of interest from OpenStreetMap to find the perfect time and place to exercise while tracking personal performance over time.[41]

One of the lessons that we learned from the 2017 and 2018 ESA Space App Camps, after compiling user feedback, was that the capability of integrating Earth observation data together with linked open data (e.g., landmarks, points of interest) was very important in some applications. In all of the applications that were based on combining Copernicus data with open data, developers chose to use the Copernicus App Lab tools. Given the limited time they had available for coding, they found it easier to access the available GeoSPARQL endpoints rather than the original heterogeneous data sources. On the other hand, some developers also reported that getting familiarized with the Copernicus App Lab technologies required some effort, given that they did not have background knowledge in Semantic Web technologies or geospatial data management. These users suggested that this issue could be addressed by improving documentation material.

Google has recently activated the beta version of its dataset search (https://toolbox.google.com/datasetsearch), where the datasets that are indexed using schema.org (https://schema.org/), as proposed by Google, show up. Schema.org is a vocabulary created from the collaboration of four major search engines: Bing, Google, Yahoo, and Yandex. It aims to provide a unique structured data markup schema which would include a great amount of topics, including people, organizations, events, creative works such as books and movies, etc. The on-page markup allows search engines to understand information included in web pages, while it provides rich search features for users. We have followed these guidelines and annotated all the datasets used in the use case of Section 3, and made them available at the

[37] We have shown this in papers [6, 15] in 2013. The situation has not changed since then, according to recent experiments of our team.

[38] https://rya.apache.org/

[39] https://www.w3.org/2015/spatial/wiki/Main_Page

[40] http://www.app-camp.eu/

[41] All the applications developed are presented in more detail at the following Web page: https://www.app-camp.eu/winner-frascati-2017/

following link: http://kr.di.uoa.gr/#datasets. We have also recommended that the same practice is followed by the Copernicus services we have worked with (land monitoring, global land and atmosphere services).

Currently, EO datasets are hidden in the archives of big EO organizations, such as ESA, NASA, etc. These datasets are only available through specialized search interfaces provided by the organizations. It is important to make major search engines like Google able to discover EO datasets and not only general datasets, in the same way they can discover information about movies, concerts, etc. To achieve this goal, we designed an extension to the community vocabulary schema.org, appropriate for annotating EO data in general and Copernicus data in particular, by extending the class Dataset with subclasses and properties, which cover the EO dataset metadata defined in the specification OGC 17-003 for annotating EO product metadata using GeoJSON(-LD) (http://geo.spacebel.be/opensearch/myDocumentation/doc/index-en.html). The OGC 17-003 is based on the specifications OCC 10-157r4 (http://docs.opengeospatial.org/is/10-157r4/10-157r4.html) and UMM-G (https://wiki.earthdata.nasa.gov/display/CMR/CMR+Documents), and it is expected to be standardized by OGC during 2018. The OGC 10-157r4 - Earth Observation Profile of Observations and Measurements (O&M) provides a standard schema for encoding EO product metadata in order to describe and catalogue products from the sensors of EO satellites. The Unified Metadata Model for Grabules (UMM-G) is an extensible metadata model for Granules, that provides the mappings between NASA's Common Metadata Repository (CMR) supported metadata standards.

The schema.org extension for encoding EO metadata can be used by EO organizations like ESA for the encoding of the metadata of their EO datasets. In this way, EO datasets will be discoverable by search engines. In addition, this schema.org extension for EO products can be used by webmasters who want to annotate their webpages, so that search engines can find the EO datasets they provide. It is important to make this EO data available on the Web as linked data in order to increase their use by developers that might not be experts in EO. In this way, great amounts of data that are generated fast, can be made "interoperable" and more valuable when they are linked together.

6 SUMMARY

In this paper we argued that Copernicus data is a paradigmatic source of big data giving rise to all relevant challenges: volume, velocity, variety, veracity and value. The Copernicus App Lab project targets all these challenges with special focus in variety and volume, and has developed a novel software stack that can be used to develop applications using Copernicus data even by developers that are not experts in Earth observation. We presented a case study developed using the Copernicus App Lab software stack and discussed lessons learned, future plans and open problems.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825258 and Greek national funds through the Operational Program "Competitiveness, Entrepreneurship and Innovation", under the call "RESEARCH-CREATE-INNOVATE" (project code: T1EDK01000).

REFERENCES

[1] A. Andrejev. 2016. *Semantic Web Queries over Scientific Data.* Ph.D. Dissertation. Dept. of Information Technology, Uppsala University, Sweden.

[2] K. Bereta, H. Caumont, U. Daniels, D. Dirk, M. Koubarakis, D. Pantazi, G. Stamoulis, S. Ubels, V. Venus, and F. Wahyudi. 2019. From Big Copernicus Data to Big Information and Big Knowledge: the Copernicus App Lab project. In *BiDS*.

[3] K. Bereta, H. Caumont, E. Goor, M. Koubarakis, D.-A. Pantazi, G. Stamoulis, S. Ubels, V. Venus, and F. Wahyudi. 2018. From Copernicus Big Data to Big Information and Big Knowledge: A Demo from the Copernicus App Lab Project. In *CIKM*.

[4] K. Bereta and M. Koubarakis. 2016. Ontop of Geospatial Databases. In *ISWC*.

[5] K. Bereta and M. Koubarakis. 2017. Creating Virtual Semantic Graphs ontop of Big Data from Space. In *BiDS*.

[6] K. Bereta, P. Smeros, and M. Koubarakis. 2013. Representation and Querying of Valid Time of Triples in Linked Geospatial Data. In *ESWC*.

[7] J. Blower, M. Riechert, N. Pace, and M. Koubarakis. 2016. Big Data meets Linked Data: What are the Opportunities?. In *Conference on Big Data from Space (BiDS), Tenerife, Spain, March 2016.*

[8] S. Brüggemann, K. Bereta, G. Xiao, and M. Koubarakis. 2016. Ontology-Based Data Access for Maritime Security. In *ESWC*.

[9] S. Burgstaller, W. Angermair, F. Niggemann, S. Migdall, H. Bach, I. Vlahopoulos, D. Savva, P. Smeros, G. Stamoulis, K. Bereta, and M. Koubarakis. 2017. LEOpatra: A Mobile Application for Smart Fertilization Based on Linked Data. In *Proceedings of the 8th International Conference on Information and Communication Technologies in Agriculture, Food & Environment.*

[10] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro, and G. Xiao. 2017. Ontop: Answering SPARQL queries over relational databases. *Semantic Web* (2017).

[11] Y. Chronis, Y. Foufoulas, V. Nikolopoulos, A. Papadopoulos, L. Stamatogiannakis, C. Svingos, and Y. E. Ioannidis. 2016. A Relational Approach to Complex Dataflows. In *EDBT/ICDT*.

[12] A. Davvetas, I. A. Klampanos, S. Andronopoulos, G. Mouchakis, S. Konstantopoulos, A. Ikonomopoulos, and V. Karkaletsis. 2017. Big Data Processing and Semantic Web Technologies for Decision Making in Hazardous Substance Dispersion Emergencies. In *ISWC*.

[13] A. Eldawy and M. F. Mokbel. 2017. The Era of Big Spatial Data. VLDB.

[14] M. Koubarakis et al. 2016. Managing Big, Linked, and Open Earth-Observation Data: Using the TELEIOS/LEO software stack. *IEEE Geoscience and Remote Sensing Magazine* (2016).

[15] G. Garbis, K. Kyzirakos, and M. Koubarakis. 2013. Geographica: A Benchmark for Geospatial RDF Stores (Long Version). In *ISWC*.

[16] Manolis Koubarakis, Konstantina Bereta, George Papadakis, Dimitrianos Savva, and George Stamoulis. 2017. Big, Linked Geospatial Data and Its Applications in Earth Observation. *IEEE Internet Computing* 21, 4 (2017), 87–91. https://doi.org/10.1109/MIC.2017.2911438

[17] M. Koubarakis, K. Bereta, G. Papadakis, D. Savva, and G. Stamoulis. 2017. Big, Linked Geospatial Data and Its Applications in Earth Observation. *IEEE Internet Computing* (2017).

[18] M. Koubarakis, C. Kontoes, and S. Manegold. 2013. Real-time wildfire monitoring using scientific database and linked data technologies. In *EDBT*.

[19] M. Koubarakis, K. Kyzirakos, C. Nikolaou, G. Garbis, K. Bereta, R. Dogani, S. Giannakopoulou, P. Smeros, D. Savva, G. Stamoulis, G. Vlachopoulos, S. Manegold, C. Kontoes, T. Herekakis, I. Papoutsis, and D. Michail. 2016. Managing Big, Linked, and Open Earth-Observation Data Using the TELEIOS/LEO software stack. *IEEE Geoscience and Remote Sensing Magazine* (2016).

[20] K. Kyzirakos, M. Karpathiotakis, G. Garbis, C. Nikolaou, K. Bereta, I. Papoutsis, T. Herekakis, D. Michail, M. Koubarakis, and C. Kontoes. 2014. Wildfire monitoring using satellite images, ontologies and linked geospatial data. *J. Web Sem.* (2014).

[21] K. Kyzirakos, M. Karpathiotakis, and M. Koubarakis. 2012. Strabon: A Semantic Geospatial DBMS. In *ISWC*.

[22] K. Kyzirakos, D. Savva, I. Vlachopoulos, A. Vasileiou, N. Karalis, M. Koubarakis, and S. Manegold. 2018. GeoTriples: Transforming Geospatial Data into RDF Graphs Using R2RML and RML Mappings. *Journal of Web Semantics* (2018).

[23] K. Kyzirakos, I. Vlachopoulos, D. Savva, S. Manegold, and M. Koubarakis. 2014. GeoTriples: a Tool for Publishing Geospatial Data as RDF Graphs Using R2RML Mappings. In *Proc. of TerraCognita*.

[24] C. Nikolaou, K. Dogani, K. Bereta, G. Garbis, M. Karpathiotakis, K. Kyzirakos, and M. Koubarakis. 2015. Sextant: Visualizing time-evolving linked geospatial data. *Journal of Web Semantics* (2015).

[25] G. Papadakis, K. Bereta, T. Palpanas, and M. Koubarakis. 2017. Multi-core Meta-blocking for Big Linked Data. In *SEMANTICS*.

[26] K. Patroumpas, N. Georgomanolis, T. Stratiotis, M. Alexakis, and S. Athanasiou. 2015. Exposing INSPIRE on the Semantic Web. *Web Semant.* (2015).

[27] M. Perry and J. Herring. 2012. GeoSPARQL - A geographic query language for RDF data. OGC.

[28] P. Smeros and M. Koubarakis. 2016. Discovering Spatial and Temporal Links among RDF Data. In *LDOW*.

[29] C. Stadler, J. Lehmann, K. Höffner, and S. Auer. 2012. LinkedGeoData: A core for a web of spatial open data. *Semantic Web* (2012).

Modeling and Building IoT Data Platforms with Actor-Oriented Databases

Yiwen Wang
University of Copenhagen
Copenhagen, Denmark
y.wang@di.ku.dk

Julio Cesar Dos Reis
University of Campinas
Campinas, SP, Brazil
jreis@ic.unicamp.br

Kasper Myrtue Borggren
SenMoS
Copenhagen, Denmark
kmyrtue@gmail.com

Marcos Antonio Vaz Salles
University of Copenhagen
Copenhagen, Denmark
vmarcos@di.ku.dk

Claudia Bauzer Medeiros
University of Campinas
Campinas, SP, Brazil
cmbm@ic.unicamp.br

Yongluan Zhou
University of Copenhagen
Copenhagen, Denmark
zhou@di.ku.dk

ABSTRACT

Vast amounts of data are being generated daily with the adoption of Internet-of-Things (IoT) solutions in an ever-increasing number of application domains. There are problems associated with all stages of the lifecycle of these data (e.g., capture, curation and preservation). Moreover, the volume, variety, dynamicity and ubiquity of IoT data present additional challenges to their usability, prompting the need for constructing scalable data-intensive IoT data management and processing platforms. This paper presents a novel approach to model and build IoT data platforms based on the characteristics of an Actor-Oriented Database (AODB). We take advantage of two complementary case studies – in structural health monitoring and beef cattle tracking and tracing – to describe novel software requirements introduced by IoT data processing. Our investigation illustrates the challenges and benefits provided by AODB to meet these requirements in terms of modeling and IoT-based systems implementation. Obtained results reveal the advantages of using AODB in IoT scenarios and lead to principles on how to effectively use an actor model to design and implement IoT data platforms.

1 INTRODUCTION

Internet-of-Things (IoT) systems enable data interactions through machine-to-machine communication stemming from supporting devices connected to the Internet [13]. IoT systems generate a potentially huge amount of data from devices that dynamically enter and leave the IoT environment, with very high-speed data flow and processing. Data, in turn, are generated by a wide variety of devices, thus giving rise to highly heterogeneous data streams. In this paper, we distinguish between *IoT systems* (i.e., the entire software ecosystem involved in an IoT scenario) and *IoT data platforms* (i.e., the data and data management software modules that are part of an IoT system). Our work focuses on the latter.

Enormous challenges need to be addressed in order to realize the full potential of IoT. First, there is a tension between effective data management and fulfillment of performance requirements in IoT data platforms. Indeed, many IoT systems are processor-intensive and require processing a massive amount of highly concurrently generated data. The management of these interactions among data with low latency remains an open research problem. Second, being able to deal with dynamic scaling while guaranteeing protection of data from different entities is another significant challenge. Therefore, we focus our investigation on how to manage the data from large volumes of devices and, at the same time, ensure the dynamic and flexible development of applications. This dual aim must be achieved while respecting application constraints for low latency in interactive functionality as well as data protection and access control.

Given these characteristics, we propose that actor-oriented databases (AODBs) are ideally suited to manage the data of real-world IoT systems. Actors comprise a model of computation specifically aimed at high concurrency and distribution [1]. To that effect, actors keep their private states and can modify states by communicating with each other via immutable asynchronous messages [19]. As such, actors are natively applicable to support the management of an arbitrary number of independent and heterogeneous streaming data sources. AODBs, in turn, enrich actors with classic RDBMS functionality by integrating data management features, such as indexing, transactions, and query interfaces, into actor runtimes [17]. These features make AODBs attractive for building an IoT data platform. In more detail, AODBs stand out for several reasons. First, IoT systems comprise many different devices with distinct functionality. This requirement is directly met by the actor model, through the principle of assigning different logic and tasks to actors. Second, in IoT, data changes frequently; actors provide a natural alternative to conventional concurrency models that rely on synchronization of shared mutable state using locks. Third, the characteristics of non-blocking interactions via immutable messages between actors match well with the demands of IoT systems. Fourth, the number of actors can scale out quickly without consuming excessive resources. Dynamic scaling is a common situation in IoT in which all kinds of sensing devices (including humans!) can quickly enter – but also leave – a system.

There are several examples of the use of actors in IoT scenarios [4, 38, 41, 43]. However, these previous studies concentrate on implementation aspects, neither providing guidance on how to model IoT data platforms with actors nor analyzing the fit of AODB to the requirements and challenges brought about by IoT. By contrast, to the best of our knowledge, this paper is the first that builds an end-to-end case for the suitability of AODBs to manage IoT data, going from requirements and modeling to implementation and performance evaluation. Our work covers a wide gamut of issues to justify and showcase the adoption of AODBs as an appropriate solution to meet the main challenges of data management in IoT systems. Our main contributions are therefore:

(1) We discuss core requirements of IoT data platforms, and challenges to be met in their implementation. We illustrate

this discussion through the analysis of two real world IoT case studies.
(2) We present a methodology and guidelines to model an AODB for such platforms.
(3) We develop a prototype of one of the case studies and present its evaluation to show the effectiveness of adopting AODBs for IoT data platforms.

The remaining of this paper is organized as follows. Section 2 presents two case studies of IoT systems, which we use to extract functional and non-functional requirements, and to present some of the major challenges to be faced. Section 3 justifies our choice of AODBs as an appropriate technology to meet such requirements and challenges. In Section 4, we provide a detailed discussion of the challenges of modeling such platforms, and show how these challenges can be overcome for the running cases. Sections 5 and 6 respectively present our prototype for one of these cases and its evaluation. Section 7 revisits the paper by contrasting it with related work. Finally, Section 8 presents conclusions and ongoing work.

2 IOT DATA PLATFORM CASE STUDIES

In this section, we discuss the requirements for an IoT data platform and analyze two specific case studies. There are several scenarios in IoT data platforms, such as healthcare, personal security, traffic control, environmental monitoring, and disaster response. The two IoT data platform cases that we focus on are drawn from our experience with a structural health monitoring system (cf. Subsection 2.1) and a beef cattle tracking and tracing system (cf. Subsection 2.2). We have worked directly with these case studies, helping us validate common non-functional requirements for IoT data platforms as well as collect illustrative functional requirements for these applications.

For the first study, we have cooperated with SenMoS [45] in the area of structural health monitoring for large constructions, e.g., bridges. SenMoS is a Danish company that provides users with entire monitoring solutions, including requirement elicitation and cloud data management. The developers at SenMoS have participated in the design and implementation of the IoT data platform for the Great Belt Bridge [50]. The second case focuses on the management of cattle produce (and in particular the beef supply chain) from the perspective of traceability. This study is based on previous work with domain experts from the Brazilian agricultural research corporation Embrapa [28] that studied traceability in food for supply chains [35], and on interactions within the Danish Future Cropping partnership [30], particularly with experts from the agriculture solution provider SEGES [44]. Both of these organizations have substantial experience in the agricultural sector and are key players in agricultural extension systems of the respective countries. Both case studies concern the development of a scalable data platform that collects and stores data from IoT devices, processes operations, and provides information services to different users. Although these two IoT platforms target different scenarios, they present several common aspects and non-functional requirements. In particular, the systems should operate as Software-as-a-Service (SaaS) solutions and thus manage the data from several different tenants. Moreover, it is desired that scalability to large data volumes or users be achieved without a high burden on the data platform developers.
Non-Functional Requirements for IoT Data Platforms. We elicited the following common non-functional requirements shared by different IoT data platforms:

(1) **Data ingestion from endpoints**. The IoT data platform must have the capability to receive and store data from IoT devices, e.g., GPS collars on cows.
(2) **Multi-tenancy**. The IoT data platform must provide varied information services to different users.
(3) **Support for heterogeneous data**. The IoT data platform must be modular in its support for data ingested from IoT devices and allow for communication employing different data formats.
(4) **Cloud-based deployment**. The IoT data platform can be distributed in the cloud for ease of operation, management, and maintenance.
(5) **Scalable data platform**. The IoT data platform must not degrade in functionality or performance while expanding. This must occur without modifying existing software components.
(6) **High efficiency**. The IoT data platform must process massive amounts of concurrently generated data effectively.
(7) **Access control and data protection**. The IoT data platform should support data protection, enforcing authentication and access control over different users and profiles.

In addition to the requirements above, it is often the case that IoT data platforms must serve queries over historical data accumulated from devices over long time periods. In this paper, we focus, however, on online data ingestion and querying in SaaS scenarios. We note that at present, there is only limited support for declarative multi-actor querying in AODBs [17], and thus complex historical analyses could still be served by a data warehouse.

2.1 Case 1: Structural Health Monitoring

Structural Health Monitoring (SHM) systems aim to identify damaged sections on parts of large constructions that can cause safety concerns. SHM systems can help organizations save time on inspections by gathering and processing data so that the system can generate alerts when problems arise or suggest actions that can prevent faults. SHM systems are equipped with a set of sensors, e.g., to measure a bridge's extension, inclination, temperature, wind speed, and wind direction. Each sensor is connected to a data logger that converts the sensors analog signal into a digital one. The platform must collect, process and store data from the sensors. Figure 1 presents a context diagram of the Structural Health Monitoring Data Platform. This design is based on a real case study. Sensors provide data to different stakeholders, e.g., engineering experts monitoring the structure, data analysts, or the

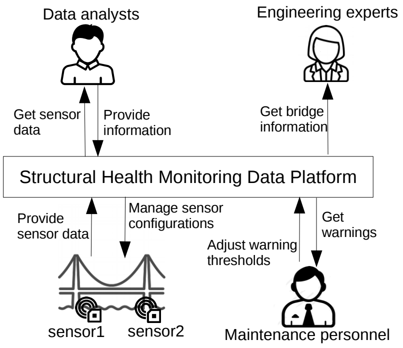


Figure 1: Context Diagram for Structural Health Monitoring Case Study.

maintenance personnel who manages the monitoring projects. The SHM system must meet the following functional requirements:

(1) The system must control several construction structures (e.g., bridges) using the same data platform.
(2) The system must be structured to support data storage, i.e., data must be saved in a way that allows for further data manipulation and analysis.
(3) The data platform must be able to maintain data from multiple sensors, users, projects, and organizations.
(4) The data platform must calculate the accumulated change for each data stream from a sensor, e.g., to gauge how far elements have moved when using extension sensors.
(5) The data platform must send customized alerts to users when thresholds are met, depending on individual sensors or sensor types. Thresholds can be used for determining the need for maintenance, or to call attention to ongoing events.
(6) The data platform must support plots providing statistical aggregates to help users spot meaningful events in time series. Besides, online plotting of recent raw sensor data is required to let personnel explore events interactively.
(7) The data platform must allow for browsing of live data from sensors, along with continuously derived equations, to provide a view of the current state of the structure.

2.2 Case Study 2: Beef Cattle Tracking and Tracing

Agricultural supply chains involve a complex network of producers, retailers, distributors, transporters, storage facilities, and consumers in the sale, delivery, and production of a particular product. Trackability and traceability are essential requirements in food marketing [24]. Tracking refers to following the path of an entity from the source to destination. Tracing refers to identifying original information regarding an entity and tracing it back in the system [37]. Systems for tracking and tracing agricultural products increase consumer confidence on provenance and quality of the food they buy, while at the same time helping retailers and certification authorities to monitor products.

The ability of IoT to collect data from sensors as well as trace entities is a crucial enabler for monitoring such chains. Systems that automate tracking and tracing in an agricultural supply chain should not only collect data, but also connect users and objects at any place and time. Data integration, processing, analysis and service support present many challenges in this context. For the sake of feasibility, we assume for this case study, similarly to other food tracing systems [33], that a global standard for supply chain messages, GS1 [32], is adopted by participants that connect with the IoT data platform. As such, we do not discuss the data integration problem in this paper.

Our case study refers to a part of the beef cattle supply chain, concentrating on cow tracking and meat product tracing, providing tracking information and helping consumers trace meat products. Figure 2 presents the entities interacting within the data platform. The system must provide multi-tenancy services to host the data of different participants and supply chains. From a high-level point of view, there are five kinds of tenants in our system: farmers, slaughterhouses, distributors, retailers, and consumers. Each involved part is the source of different types of data in the system to enable the tracing of the whole life-cycle of a given meat product.

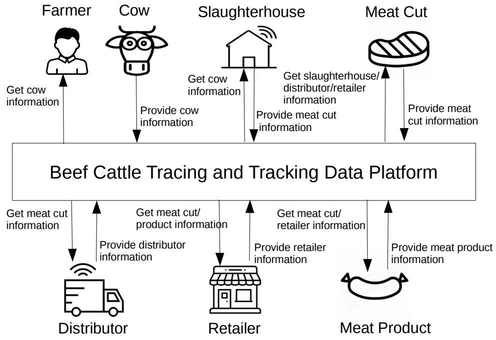

Farmer
Cow
Slaughterhouse
Meat Cut
Get cow information
Provide cow information
Get cow information
Provide meat cut information
Get slaughterhouse/ distributor/retailer information
Provide meat cut information
Beef Cattle Tracing and Tracking Data Platform
Get meat cut information
Provide distributor information
Get meat cut/ product information
Provide retailer information
Get meat cut/ retailer information
Provide meat product information
Distributor
Retailer
Meat Product

Figure 2: Context Diagram for Beef Cattle Tracking and Tracing Case Study.

Full-fledged cattle sensor-based systems involve the deployment of very many kinds of sensors – both in individual animals and in their environment. For instance, each animal has external sensors (e.g., collars, earrings) to measure movement, speed, location. Cattle often also have sensors inside their digestive tract (usually swallowed, sometimes implanted), to measure factors such as temperature, metabolic variables, or digestive characteristics. Environmental sensors may monitor factors such as cattle weight, or soil humidity. Additional sensors along the supply chain include devices that provide trackability (e.g., in transportation), but also traceability and quality (e.g., monitoring temperature inside warehouses). Without loss of generality, we have simplified this scenario to consider only a few of these sensing sources, keeping only enough distinct sensors to illustrate actual data and sampling rate heterogeneity. This simplified scenario must fulfill the following functional requirements:

(1) The data platform must store the data from animal and environment sensors, such as collars bound to each individual cow, to enable retrieval of location, motion, and other facts regarding traceable entities.
(2) Farmers need to track each cow's trajectory and behavior, and thus the data platform must record the locations of each cow over time. Geo-fencing can help identify whether a cow is in an appropriate area (e.g., when rotating pasture grounds) [20].
(3) Slaughterhouses wish to access services that provide information about cows that will be slaughtered. For instance, it must be possible to access tracing information such as the provenance of the cows and tracking information about where the meat cuts produced after slaughter are transferred to.
(4) Distributors wish to get tracing information of a meat cut and tracking information of where those cuts are to be sent to.
(5) Retailers aim to know the source of the meat cuts and manage their transformation into meat products for consumers.
(6) Consumers wish to get tracing information about meat products over the whole supply chain.

2.3 Challenges for IoT Data Platforms

The functional and non-functional requirements discussed in this section render the modeling and building of IoT data platforms a non-trivial undertaking. The construction of such a platform involves technical issues related to capturing, identifying and storing relevant events, managing associated constraints, processing varied types of queries, etc. Further complexity arises from taking into account the necessities of different stakeholders, and issues related to data precision, synchronization and availability.

Choosing the right database architecture is therefore a key decision for the success of an IoT data platform project. Virtualized deployment for efficiency and ease of scaling to large request volumes are significant obstacles [46]. Moreover, support for multi-tenancy needs to be carefully designed. While physical sharing of tenant data lowers overhead and increases efficiency [7], this strategy opens up security risks, which are related to the lack of modularity at the database level [47].

Thus, several questions need to be addressed when building IoT data platforms. First, a vast amount of data is concurrently generated from IoT devices. How can this data be managed and processed in the data platform? Second, how can data protection and access control across different entities be enforced while sharing data effectively? Third, our case studies suggest that a variety of queries, including analyses of time series from bridge sensors, spatial queries for cow locations, or graph navigation for tracing, need to be efficiently supported over IoT data. How can applications be modeled and built to support different types of queries? Fourth, it is necessary that the system be easily scaled without affecting functionality and performance. How can the platform be architected to easily scale out when it becomes necessary to manage more users and data? In this investigation, we observe that the issues regarding modularity and scalability pointed out in this section can be simultaneously addressed by AODBs [17]. We design cloud-centric actor-oriented database backends for the two IoT case studies introduced in this section. As a first step, we explain the motivation of taking an approach based on AODBs in the next section.

3 WHY ACTOR-ORIENTED DATABASES?

We argue that an actor-oriented database is the ideal organization for an IoT data platform, enabling fulfillment of all common non-functional requirements identified in Section 2. Moreover, AODBs ease the achievement of functional requirements by providing a modular, stateful, and scalable substrate for the modeling, design and implementation of an IoT data platform. The following characteristics of an AODB illustrate its suitability to address the challenges of IoT data platforms.

AODBs facilitate the management of distribution and the encapsulation of data. Actors are logically distributed, and can thus naturally map to dispersed entities such as sensors. The latter promotes the expression of parallelism in the application logic responsible for data ingestion into the platform. Moreover, an AODB-centric design functionally decomposes the data platform into different actors. State is encapsulated within each actor, and can only be communicated by asynchronous messages. As such, actors provide a mechanism for isolation of different functions and data, enabling efficient support for multi-tenancy.

Actor modularity in AODBs supports representation and sharing of heterogeneous data. Actors are the unit of modularity in an AODB. By encapsulating state and supporting specification of user-defined APIs, actors abstract heterogeneous data representations. Moreover, arbitrary data transformations can be coded in actor methods, enabling asynchronous exchange of data across heterogeneous actors. As such, actors offer an attractive model to capture heterogeneity of data formats and representations originating at multiple IoT devices.

AODBs employ multiple actor types and concurrent execution among actors to achieve scalability. The support for multiple actor types enables the representation of different kinds of entities in the IoT data platform. When a new entity type is added to the system, it is represented by a new actor type added to the data platform. AODBs thus support a gradual extension of the platform through new actors and actor types with minimal impact on existing components. The use of actors makes scaling out easier, since new actors can be deployed over additional hardware components to avoid violation of performance constraints. The resulting concurrent and distributed execution facilitates efficient use of computational resources to bolster scalability.

Parallelism across actors in AODBs allows for processing of massive amounts of concurrently generated data. By identifying tasks and associated logic among entities, we can model entities as independent actors, so that they can perform tasks concurrently. When independent tasks are then run in parallel across actors deployed on separate hardware components, data platform performance can be improved. Since sensors are naturally modeled as different actors, parallel execution can be leveraged in processing data from a large number of data streams.

Encapsulation and modularity in AODBs support data protection and access control. As data in one actor are invisible to others, access permissions can be checked when data are exchanged by asynchronous messaging [40]. In other words, data are protected inside an actor, and mechanisms for access control can ensure data are only shared with authorized users.

4 MODELING THE CASE STUDIES WITH ACTOR-ORIENTED DATABASES

AODBs provide scalable data processing, management, and storage over a set of application-defined actors [17]. However, to the best of our knowledge, there is scant guidance on how to model applications to reap the benefits of AODBs. In this section, we fill this gap by an in-depth modeling discussion of the two IoT case studies described in Section 2. We contribute to the construction of IoT data platforms in two ways: (1) we provide guidelines for modeling IoT data platforms with AODBs; and (2) we explain how an actor model affects the system both in design and implementation.

It is believed that application modeling helps to preserve and reuse information in other projects, as well as facilitates the automated generation of a system from models [51]. To support the latter aim, we leverage UML notation [53] to create models of actors, their encapsulated state and their operations. These data-centric models harken back to conceptual modeling approaches in databases, enabling both specifications of data requirements and, in the future, code generation for AODB platforms. To support the former aim, we focus on documenting database actors and their asynchronous interactions. In addition to database actors, the classic architecture of an IoT data platform contains a stateless tier that mediates the interaction with users or devices. The analysis of this tier is outside the scope of our work; we abstract its functionality as stateless actors operating as proxies and omit this tier from our models.

In the presented models, we represent the minimal necessary information to emphasize techniques for actor-oriented database design. Application details that would make the presentation unnecessarily complex are thus omitted, and simplifications are made where appropriate. In the sections that follow, we identify each core modeling question encountered in the two IoT data platform case studies and discuss lessons learned.

4.1 How can Actors be Identified?

A variety of entities exist in any system, and these entities either perform or collaborate to achieve different tasks. Moreover, these entities have different life cycles, distinct types, and varied needs regarding heavy computation or communication [14]. To take advantage of the actor programming model as well as achieve high availability and performance, it is an essential question to decide which entities or entity sets should be modeled as actors.

To appreciate a concrete example of this challenge, consider the beef cattle system introduced in Section 2, where many entities perform different tasks to satisfy multiple requirements. For instance, a collar sensor that is bound to a cow continuously collects real-time geo-data for this cow and sends it to the data platform. A historical trajectory for each individual cow is thus updated based on this sensor data. Besides, additional information may be needed, such as the cow's identifier or health-related data. In this sense, we can observe both collar and cow as separate entities engaged in cooperation to provide real-time and historical geo-information to farmers and slaughterhouses. The question is whether these two entities, the collar sensor, and the cow, should be one or two independent actors.

Actors comprise a model of computation for concurrency and distribution [1]. So not only should actors encapsulate state, as it is the case with non-actor objects, but they should also abstract concurrent tasks that need to be processed by the system. In our experience, we found it useful to answer the following questions when attempting to identify actors: (a) What services are provided by the system being modeled? (b) Who should provide these services and who are their users? (c) What is the output and input of every single task performed and can these tasks be executed concurrently?

Take our beef cattle tracking and tracing system as an example. Typically one actor is designed to carry out one specific real-world task with associated logic, such as slaughter or distribute. Different actors then capture simultaneous tasks. For instance, farmers would like to obtain information on cows and manage the herds that they own. Slaughterhouses would like to obtain information about cows that will be slaughtered and record how these cows get transformed into meat cuts. Farmers and slaughterhouses can thus be conceptualized as users of cow information services, which a cow ought to provide. Moreover, the interactions between farmers, slaughterhouses, and cows are concurrently executed by independent entities in the real world. In particular, farmers manage cows and their respective information, and slaughterhouses slaughter cows and record their transformation into meat cuts. Cows are associated with their sensor data, which are continuously updated by their collars. As such, we model farmers, slaughterhouses, and cows as independent actor types. Since each collar is bound to a cow, we encapsulate this sensor information inside cow actors.

Figure 3 shows the data platform model for the beef cattle system. Every actor encapsulates its state and communicates with other actors via asynchronous messages. Therefore, simple accesses to data in the state of an actor are rendered as asynchronous communication events across actors. As we can see from the figure, we model one cow as a *Cow* actor. A Cow actor has an aggregation relationship with many collar sensor readings indicating the GPS locations of the cow, and each such sensor reading is bound to exactly one cow. In other words, we use aggregation relationships to indicate that the objects of a non-actor class are encapsulated in the referred actors. Since cows are modeled as actors, real-time locations are reported to Cow actors, which serve this information to all interested readers along with other associated cow state data such as the cow identifier.

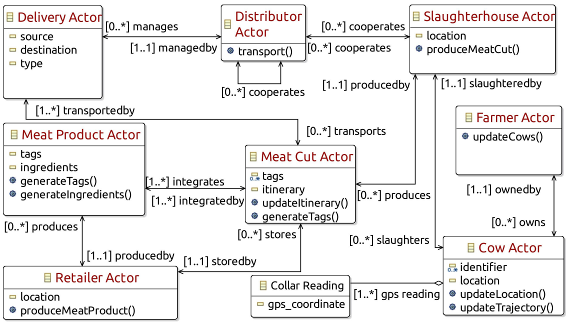


Figure 3: Actor Model of Beef Cattle Tracking and Tracing Data Platform.

We model one farmer or several farmers who work together (e.g., a cooperative) as one single *Farmer* actor because the state of this farmer or these farmers is organized as a unit.[1] One cow is owned by one farm unit, but one farm unit can own many cows. The Farmer actor can read the properties of any Cow actor that is associated with it through message passing. If such messages are exchanged under a security model with authentication, then we can enforce that cow information is only visible to its owner farmer tenant or properly authorized slaughterhouse.

A physical slaughterhouse is modeled as a *Slaughterhouse* actor. A cow can only be slaughtered once in exactly one slaughterhouse, but a slaughterhouse is responsible for slaughtering many cows. This constraint is reflected in the association between Cow and Slaughterhouse actors, and as above a Slaughterhouse actor can read data from any Cow actor via asynchronous messaging. The Slaughterhouse actor processes such data to derive *Meat Cut* actors, which represent units of beef to be distributed as a whole.

A Meat Cut actor processes updates to its itinerary property generated by *Delivery* actors as meat cuts are transported. In our model, a *Distributor* actor manages multiple Delivery actors, which themselves manage a transportation process with different source and destination locations. For example, a logistics company is modeled as a Distributor actor, and transportation processes in this company are modeled as various Delivery actors managed by the Distributor actor. A Delivery actor tracks a meat cut delivery from a source to a destination location using a given vehicle at a well-defined time. A meat cut can be delivered many times by one or several distributors during the whole itinerary, and a distributor is responsible for delivering many meat cuts.

We model the final destination of a meat cut to be a retailer, e.g., a supermarket chain, whose information is managed by a *Retailer* actor. Retailer actors can create *Meat Product* actors by disaggregating or combining meat cuts. Thus, Meat Product actors have a many-to-many association with Meat Cut actors.

Based on the above modeling process, we can summarize a general **principle of how to identify actors**:

> Typically, one actor is designed to carry out one specific real-world task with associated logic. Different actors capture different simultaneous tasks.

[1] Notice that this unit can be broken down into smaller units at will, depending on the focus of an application – e.g., individual farmers unite to provide beef, but a cooperative handles each farmer individually.

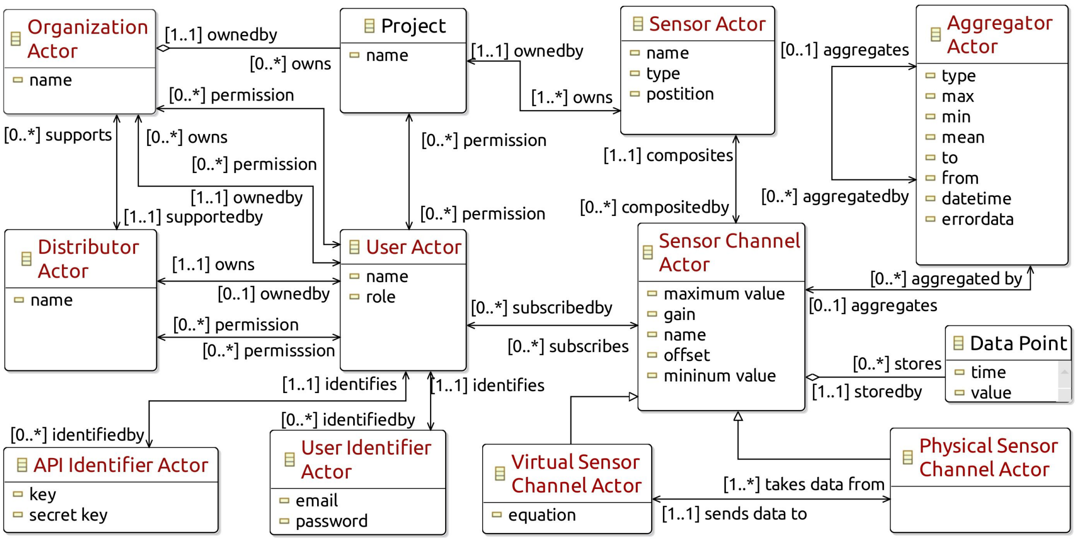


Figure 4: Actor Model of Structural Health Monitoring Data Platform.

4.2 What should the Granularity of Actor State be?

In an AODB, we allocate different tasks into separate actors, as this organization can help increase concurrency. However, if a single actor concentrates too much state or too much of the application logic, i.e., if an actor is *coarse-grained*, then it becomes increasingly difficult to reap the benefits of concurrency stemming from the application of the actor model. At the other extreme, since actors do not share state and communicate only through asynchronous messaging, an excessively *fine-grained* actor design can introduce unnecessary overheads in state manipulation as well as increased communication overhead. Moreover, a fine-grained design may cause the actors in the system to explode in number, e.g., one actor per data item or record in the system, which can challenge efficiency in an AODB platform. So deciding the granularity of actors is an essential problem when modeling any application with actor-oriented databases.

Previously, we have formulated a principle to identify actors out of the entities in an application scenario. However, we should balance this principle against the potential effect of actor granularity on application performance. In particular, we wish to keep concurrency high, but at the same time avoid unnecessary overheads and reduce the complexity of application modeling. To balance these goals, our experience has been that it is natural to make actors more fine-grained when they represent *active entities* for which detailed tracking is required by the application.

Figure 4 presents the data platform model for the structural health monitoring system. We observed during modeling that organization entities own project entities representing different constructions and that each such construction project is associated with some installed sensors. Note that only organizations are active, as they initiate and manage construction activities, while projects are passive structural schemes used by organizations. As such, we create *Organization* actors that encapsulate project information, as displayed by an aggregation relationship with a non-actor Project class, instead of utilizing separate actors. This modeling decision minimizes message exchange when there is no clear advantage in having the two entities run concurrently.

This notwithstanding, sensors are themselves active entities in that they may be relocated, leading to change of position, and also may generate multiple data streams originating from different physical sensor channels (e.g., if we consider a regular smartphone as a sensor, then the accelerometer and microphone would be sensor channels). Moreover, messaging is minimal between sensors and sensor channels, as data streams arriving at the platform can be disaggregated by proxies directly by sensor channel instead of being relayed through sensor entities. As such, we model separate *Sensor* and *Sensor Channel* actors. Sensor Channel actors hold a window of data points originating in the respective data stream. The data points are captured as non-actor objects since these entities are not active.

To help structure information about data points, additional actors are included. First, Sensor Channel is specialized into *Physical Sensor Channel* and *Virtual Sensor Channel* actor classes. Whereas the former represents a channel in a physical sensor, the latter represents a computation over potentially multiple physical channels (e.g., in our smartphone example, an equation merging the data from accelerometer and microphone sensor channels). While a virtual sensor channel provides data at the finest level of detail, it is necessary to provide statistical aggregates for online queries posed by data analysts at various levels of detail (e.g., per hour, day, or month). Since there can be parallelism in computing these aggregations across levels of detail (e.g., hourly aggregates serving as input to daily aggregates), it is useful to conceptualize them as active entities. We thus introduce *Aggregator* actors in the model.

Based on the above modeling process in the context of our case studies, we summarize a general **principle of how to decide on actor granularity**:

> An actor should represent the functionality of one active entity for which detailed tracking is required.

4.3 What is the Trade-Off between Employing Actors or Non-actor Objects for Frequently Accessed Entities?

We discussed the issue of actor granularity, which may result in decisions where entities from the domain are modeled as actors or alternatively as non-actor objects. The modeling principle for actor granularity calls our attention to active entities. By contrast, there are a number of entities that store data but do not proactively perform tasks. We call them *inanimate entities*, and they are exemplified in the beef cattle tracking and tracing case study by meat cuts and meat products. In Figure 3, we model these inanimate entities as actors. However, these actors only encapsulate state and manage corresponding queries and updates originating from active entities such as slaughterhouse, distributor or retailer, e.g., when meat cuts and products are created or transported. As such, a natural question is whether these inanimate entities could have been modeled as non-actor objects instead of actors.

For example, suppose a distributor wishes to obtain information about a meat cut that it transports. The corresponding Distributor actor would have to send a message to the respective Meat Cut actor to fetch this information. Furthermore, when a meat cut is transported, the Meat Cut actor has to communicate with a number of other source or destination actors, such as Slaughterhouse, Distributor, or Retailer actors. As such, a Meat Cut actor frequently interacts with other actors in the system. Since all information on meat cuts needs to be exchanged across actors through asynchronous messaging, casting meat cuts as actors can generate a considerable communication overhead.

To explore this question, we have created an alternative model for the beef cattle tracking and tracing case study (cf. Figure 5). In this alternative model, we capture inanimate, but frequently updated entities, such as meat cuts and meat products, as non-actor objects instead of actors. Actors are marked in red in Figures 3, 4 and 5, while the non-actor objects are marked in black. The non-actor objects represent a state and thus cannot exist in an AODB independently of some actor. To capture state mutation as meat cuts and products move across the supply chain, we create object versions that are always associated with a responsible actor at every stage. Consider how a meat cut is transferred from a slaughterhouse to a distributor. The meat cut is the same real-world entity, but the slaughterhouse and distributor may identify the meat cut differently. Upon transfer, the object representing the meat cut will be copied from the Slaughterhouse actor to the Distributor actor, where this new object version can be updated. Since each actor keeps a separate object version of the meat cut throughout the supply chain, communication to obtain meat cut information is obviated. All the actor logic that reads this information can now access the encapsulated entities in the respective actor state. For frequently accessed entities, this reduction in communication may pay off with respect to the overhead of copying non-actor objects. Furthermore, potentially more concurrency can be exploited in reading local object versions across several actors independently. However, some degree of data redundancy may be introduced in the model.

Based on the above modeling process, we can summarize a general **principle of when to model frequently accessed entities as non-actor objects instead of actors**:

> Frequently accessed entities can be modeled as actors or non-actor objects, and the latter representation should be preferred when reductions in communication overheads and gains from concurrency offset the disadvantages of copying overhead and data redundancy.

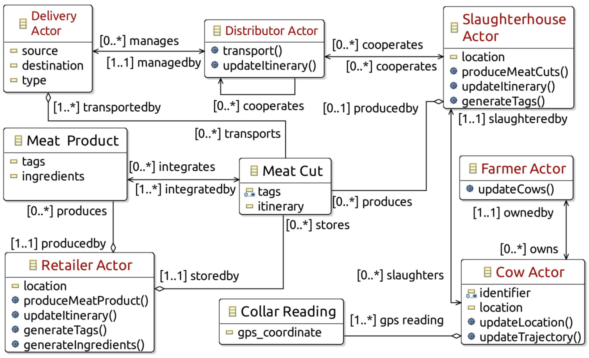


Figure 5: Alternative Actor Model of Beef Cattle Tracking and Tracing Data Platform.

4.4 How can Relationship Constraints be Enforced across Actors?

Because actors encapsulate state and only communicate through asynchronous messaging, relationships between actors are conceptually distributed. For instance, in the model of Figure 3, a farmer may own many cows, but a cow belongs to at most one farmer. In a typical implementation, each direction of this relationship would be represented as properties in Cow and Farmer actors. When performing updates to this relationship, we need to update both sides and make sure these two properties in different actors remain consistent. In particular, when a farmer sells a cow, the Cow actor should have its *ownership* relationship changed to the next owner, and the properties in the two affected Farmer actors ought to reflect that only one farmer retains the ownership of that cow. Since communication between actors is asynchronous, it is a challenge to keep consistency across actors in the presence of updates.

The consistency problem can be addressed by a transaction facility in the AODB, when available, or alternatively by a workflow that ensures that all actors in a relationship change are eventually updated to a consistent state. These options are similar in spirit to the proposal for indexing support in AODBs [17]. Since some actor systems, such as Akka, no longer support transactions [52], and update workflows operate under relaxed consistency, a final alternative is to keep all data related to a relationship, or more generally constraint, encapsulated in a single actor. This discussion leads us to our final **principle of how to enforce constraints when using actors**:

> Employ transactions to update data across actors consistently; however, in the absence of transactions, keep data related to a constraint in a single actor or design a multi-actor workflow for updates.

5 IMPLEMENTATION

In this section, we discuss the implementation of the IoT data platform for the first case study of Section 2 with an AODB. We choose the Structural Health Monitoring Data Platform (SHMDP) since the resulting implementation has been transitioned to the company SenMoS. However, the lessons learned and discussion extend more broadly to the applicability of AODBs to IoT data platforms in other domains, e.g., in beef cattle supply chains, among others.

Choice of AODB. Our implementation of the structural health monitoring data platform was based on the model of Figure 4 [18].

The first implementation challenge to be overcome was to find an appropriate platform supporting actor and non-actor object constructs, as well as the AODB approach. The vision for AODBs [17] was proposed in the context of the Orleans project [16], and we thus elect this actor runtime for the SHMDP. Orleans has also been used successfully in the context of other scalable applications [38], and can thus support real-world deployments. Unlike many other actor programming languages or frameworks such as Erlang [29] or Akka [3], Orleans employs the concept of *virtual actors*, i.e., named actors that are logically in perpetual existence. The Orleans actor runtime automatically creates activations of these virtual actors for processing whenever functions are asynchronously invoked on them, and eliminates activations when there is pressure on resources. As such, virtual actors simplify actor lifecycle management for an application built on Orleans.

In addition to virtual actors, Orleans provides an explicit storage model for actor state. In particular, actors run in a stateful middle-tier that can be conceptualized as an in-memory cache of actor state enriched with application code expressed as actor functions. Whenever persistence of actor state is required, a cloud storage system is employed by Orleans. The concrete storage system is specified through annotations in actor code. To meet the vision for AODBs, additional features are currently being implemented in Orleans to close the gap between actor runtime and DBMS functionality, e.g., indexing [17] and ACID transactions across actors [27].

Data Platform Architecture. A second implementation challenge was to architect an IoT data platform based on AODBs that fulfills all of the non-functional requirements of Section 2. Ideally, an AODB should handle online data ingestion and querying as well as analyses of historical data. However, as pointed out in Section 2, declarative querying functionality is still incomplete in AODBs currently [17]. Thus, we identify three core components for the SHMDP: actor runtime, cloud storage system, and analytical database system. The actor runtime was implemented by Orleans and provides the virtual actor abstraction. It also keeps any necessary in-memory data structures for online data processing and analysis as expressed in the model of Figure 4. The storage system provides durability of actor state, and allows large amounts of historical data to be archived. A key-value database system with efficient data ingestion [36] is useful for this purpose. Finally, data recorded in the storage system can be exported into a classic star schema implemented in the analytical database [34]. The latter component is targeted at analytical queries over historical data, and its description is outside the scope of this paper. The former two components comprised the online data ingestion, processing, and analysis functions of the SHMDP.

Support for Non-Functional Requirements. The AODB architecture supports the non-functional requirements listed in Section 2 as follows:

(1) **Data ingestion from endpoints**. Data from different endpoints was managed by distinct actors in Orleans, and recorded in the cloud storage system for durability.
(2) **Multi-tenancy**. Modularity, data encapsulation, and asynchronous communication were provided by virtual actors in Orleans, allowing isolation of functions and data sensitive to different users.
(3) **Support for heterogeneous data**. Orleans virtual actors support a number of data types and structures, e.g., representing simple alerts or real-time derived data for virtual sensors. In addition, Orleans was used to query time ranges of raw data, and to build aggregates for low latency requests over time periods. The problem of using Orleans for these functionalities was that declarative queries cannot access data across actors, and thus needed to be decomposed by the developer.
(4) **Cloud-based deployment**. Orleans was built to scale out on servers, and extend over multiple geographical locations. It is, moreover, open-source and designed with cloud deployment as a primary target.
(5) **Scalable data platform**. Modularization allows scalability in the number of actors, thus easily enabling the addition of more endpoints or users to the data platform.
(6) **High efficiency**. All processing in virtual actors occurs in-memory. Orleans employs multi-core and multi-server architectures to execute application logic in different actors in parallel.
(7) **Access control and data protection**. Authentication and access control were implemented at the application level by building on actor modularity features.

Virtual actor durability and deployment. Further implementation challenges arise from ensuring that the IoT data platform can effectively ingest and process the large number of concurrent update streams originating from devices. Two issues may impact performance substantially: enforcing durability and deploying actors over multiple machines in a cloud infrastructure.

Orleans virtual actors are called grains, and managed automatically by the Orleans runtime. When a grain has work to do, the grain is activated; when the grain has been standing idle for too long, the grain's resources are reclaimed by the system, removing it from memory. To provide durability, grains in Orleans may have a state storage class. This class defines all variables the developer wishes to store persistently. The developer can force the current state to persistent storage by invoking the WriteStateAsync grain method or configure the grain class to store state persistently when Orleans deactivates a grain. Whenever the Orleans runtime re-activates a grain, the runtime retrieves by default the latest grain state from cloud storage, if available. As such, Orleans lets the developer decide when state is written to persistent state storage.

In the SHMDP, durability requirements may vary depending on the task being implemented. Certain tasks require that the state of actors be immediately made durable, e.g., for creating structural entities such as organizations, sensors, projects, and sensor channels. Other tasks, such as gathering sensor data, can collect a window of updates before forcing them to storage. For example, in the Great Belt Bridge [50], the structural health monitoring project consisted of more than 200 sensor channels, with a typical requirement for live data being a reporting rate of one packet of readings per sensor per second. So if we wrote state to persistent storage after each request, we would need 200 write requests every second to the cloud storage system.

Activated grains in Orleans get distributed across a set of silos, where each silo is typically deployed in a server in a cluster of machines. The distribution of grain activations to silos is by default random, which is adequate for most use cases since it will spread load. However, this actor deployment can increase the cost of communication when certain actors interact frequently. Orleans suggests using prefer-local activation in these cases. For our data platform, we have had to change the activation placement strategy away from random placement for our sensor channels and aggregators. The prefer-local placement in these instances minimizes the need to perform remote procedure calls when processing incoming requests.

6 EXPERIMENTAL EVALUATION

Our goal in the experiments is to assess if the AODB-based implementation of our model from Figure 4 yields an IoT data platform that can scale in the number of sensors simulated and at the same time support low-latency online query functionality. In the following, we present our setup and the obtained results.

6.1 Setup

Benchmarking Tool. To stress-test the SHMDP, we created a command line tool in .NET that uses the Orleans framework client directly. This tool simulates data requests from sensors and users in order to generate variable load for the data platform. Sensors are simulated by tasks that each call a sensor grain and insert 10 data points. This procedure is repeated each second if all sensors have finished their calls, so as to adhere to the behavior expected in the real scenario based on our experience.

Even though we simulate sensors for experimentation with the benchmarking tool above, we envision that ingestion of sensor data points will be based on a REST interface in a production deployment. This way, sensors can send HTTP calls to the data platform. As part of data ingestion, message queues can be employed to accommodate for bursty behavior in sensor measurements [6]. To limit the scope of our evaluation, however, we focus on stressing only the virtual actor implementation of the IoT data platform, and not other layers related to communication with sensor devices.

The benchmarking tool stores data from each request sent to the data platform in a log. Each log entry includes the latency for the request, which request was sent (data insertion, live user data, or user data request), the sampling rate, and a timestamp. With this information, we can derive detailed throughput and latency statistics for the experiments.

Summary of Software. We needed the execution of several components for the experiments. The first one was the *Orleans silo*, typically with one instance deployed per server, where virtual actors are activated and run all application logic. We also employed *Amazon RDS* [12] for Orleans system storage, which keeps track of silo instances, reminders, and general system state. *Amazon DynamoDB* [9] was used for Orleans grain state storage. Besides, the C# benchmarking tool described above is invoked to generate load to silos.

Cloud Service and Deployment. To characterize the SHMDP's data ingestion and processing capabilities, we set up our benchmarking environment on Amazon AWS [8], employing the Amazon DynamoDB and RDS services [9, 12] as stated above and EC2 on-demand instances [10] for all remaining components. Given our budget, two types of instances were employed: T2 for low cost and burst performance features as well as M5 for more stable performance. All instances were running Windows Server 2012 R2 and Orleans 1.5.0. The configuration, unless otherwise mentioned, was designed to simulate a possible future production deployment of the data platform based on our previous experience with the project for the Great Belt Bridge [50]: m5.xlarge instances were employed for the Orleans silos, RDS db.t2.small for Orleans system storage, DynamoDB with 200 writes and 200 reads per second for Orleans grain storage, and an m5.2xlarge instance for the benchmarking tool.

Environment Configuration. For the experiments, we simulate sensors with two sensor channels each; every tenth sensor has a virtual sensor channel that is a summation of the two other sensor channels on the corresponding sensor. The latter choice reflects that only a subset of sensor data require additional processing to create a derived virtual sensor stream, which is close to the real life scenario from the Great Belt Bridge. We populated our actor-oriented database with synthetic data for users, organizations, projects, sensors, and sensor channels simulating a realistic scenario. For every 100 sensors, a new organization was constructed with a single user and a single project. Following the sensor configuration, these 100 sensors represent 210 sensor channels in total, out of which 200 are physical sensor channels and 10 are virtual sensor channels. This structure was used for all experiments, so that we can calculate exactly how many organizations, projects, users, and sensor channels are created given a number of sensors. Employing 100 sensors with 210 sensor channels in total is a configuration similar in size to the one in our previous experience with the Great Belt Bridge.

To achieve our experimental goal related to low latency queries, the upload of data points to the grain state storage has been configured to only happen when the Orleans silo service is shut down. This configuration ensures that we are not benchmarking DynamoDB storage, but rather the execution of in-memory actors. When using the system in production, the grains have to be configured to store data points to grain storage at an acceptable rate as explained in the considerations for durability in Section 5.

Load was offered to the SHMDP by sending requests with 20 data points for each sensor currently being simulated (i.e., 10 data points were generated for each physical sensor channel in each sensor). The requests were sent at a rate of 1 request per second. This frequency simulates sensors sampling data at 10 Hz, as specified in the Great Belt Bridge project for most sensors. As an example, consider that we wish to simulate 500 sensors: this number of sensors would correspond to 1,000 physical sensor channels and 50 virtual sensor channels. Thus, the resulting load would be of 500 requests per second being used to transmit 10,000 data points per second, and leading to the calculation of 500 virtual data points each second.

For each experiment, Figures 6, 7, 8 and 9 present the results. A single point on the figures aggregates 10 minutes of the whole service configuration running. The data was split into windows of 1 minute, and the first minute was removed to make sure the platform had started up correctly before measurement. In addition, the last minute was removed to ensure that only whole minutes were used. The average latency or throughput was then calculated as a measurement, and depicted along with standard deviation as error bars where appropriate.

6.2 Experimental Results

How many sensor readings can the SHMDP ingest using a single cloud server? In our first experiment, we aimed at establishing a relationship between the number of simulated sensors and the hardware utilization at the data platform, so that we can create a baseline load for the other experiments. In particular for these measurements, we employed the smallest VM size in the M5 series, the m5.large instance, and observed when the instance cannot process any more data insertion requests. We have chosen the smallest server size so that the experiment can be used for both scale up and scale out baselines.

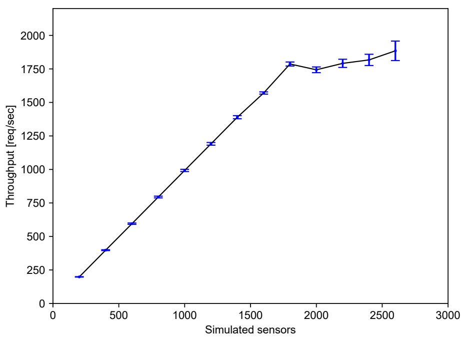


Figure 6: Single-server throughput experiment.

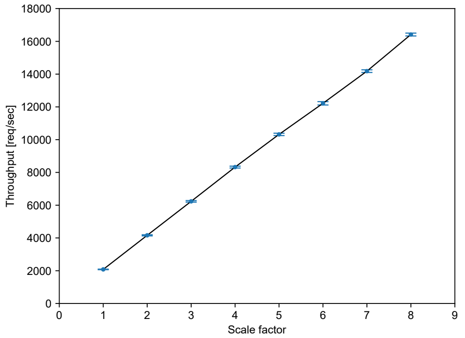


Figure 7: Scale out experiment over multiple servers.

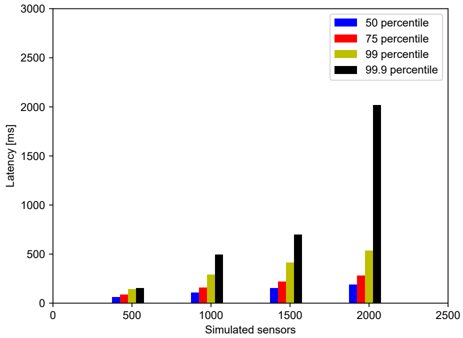


Figure 8: Latency percentiles for raw sensor channel data point time range requests.

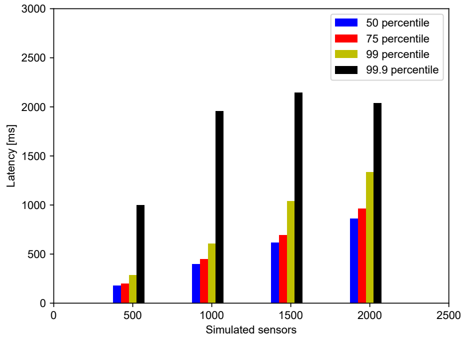


Figure 9: Latency percentile for organization live data requests.

Figure 6 shows the results from this single-server throughput experiment. Because each simulated sensor in the experiment was configured to send one request every second, note that the SHMDP deployment was processing all requests as long as the throughput is equal to the amount of simulated sensors. We observe that the ratio between simulated sensors and throughput is close to one until the number of simulated sensors reaches 2,000. At that point, throughput ceases to increase even if more load is offered. By monitoring the VM instance when performing the experiment, we have remarked that CPU Usage in Windows Task Manager was at 100% when the number of simulated sensors was above 1,800.

Does the SHMDP scale simultaneously on the number of sensors and servers? Our second aim was to verify whether the data platform can scale out in the number of data requests that it can ingest from simulated sensors by utilizing the computing power of more servers. To simulate a production environment, we employed larger m5.xlarge VMs as described in the experimental setup. From our single-server throughput experiments, we can estimate a baseline load to be offered per server. Based on this baseline, we can proportionally scale the load, number of servers, and organization structure in the experiment.

To estimate baseline load, we note that in a production environment, we wish to leave some CPU resources for user interaction. We chose to leave roughly 20% utilization for handling user online query requests and creating statistical aggregates. From the single-server throughput experiment of Figure 6, we know that roughly 1,800 requests per second can be processed by a m5.large instance. By removing 20% and rounding to the nearest 100 requests per second, we obtain 1,400 requests per second. Now, we can scale that number by the difference in computing power between the m5.large and m5.xlarge instances, which is estimated by their EC2 Compute Unit (ECU) values to be of a factor 1.5x. So the baseline for a single server corresponds to the load offered by 2,100 simulated sensors. This configuration is employed for a scale factor of one. As the scale factor is increased, we proportionally increase the number of simulated sensors and the number of servers used for Orleans silos.

Figure 7 shows that the throughput sustained by the data platform scales close to linearly with the scale factor. To illustrate this observation, consider that at a scale factor of five, we have five server instances and 10,500 simulated sensors. We observe as expected a throughput above 10,000 requests per second. Similarly, for a scale factor of eight, we have eight server instances and 16,800 simulated sensors, and a throughput above 16,000 requests per second is observed.

The results indicate that the data platform can potentially scale out even further than the 8 servers used in this experiment, since we did not hit any bottlenecks. We expect that the behavior can be maintained as we add a larger number of servers, since there are no dependencies across organizations and there is enough processing slack left to support eventual online user queries and calculation of statistical aggregates.

Does the SHMDP deliver low latency on online query functions concurrently with data ingestion? We have simplified the previous experiments by removing any user interactions, and made all sensors sample data at 10 Hz sending 1 request each second to the data platform. This scenario is close enough to our experience with a real deployment that we can observe how the data platform scales as we increase the number of sensor insertion requests. However, we still need to show that the 80% utilization rate chosen earlier will indeed leave enough room for the processing of online user queries. Furthermore, we aimed at better characterizing user request latencies under this target utilization level to make informed decisions when creating a production environment in the future.

To simulate user requests to the data platform, we estimated a relationship between simulated sensor requests and user interaction requests. We know from the requirements for the SHMDP that requests for live data as well as raw data kept in the sensor channel actors need to be supported. Requests for live data retrieved the most recent values from all sensor channels of a given organization, while requests for raw data retrieved the time series in a given sensor channel actor in an organization. From actual user interactions observed at the Great Belt Bridge project, we expect these online queries to be generated by at most one person looking at live data for each organization requesting data once every second, and at most one request for raw data a second for each organization. Since a deployment in that project would have around 100 sensors, we thus generate roughly 1% of the requests for live data from all sensor channels in a organization, 1% for raw data, and the remaining 98% as sensor data insertions.

Figures 8 and 9 show that the latency of online query requests increase, as expected, for higher percentiles of the latency distribution. This growth is especially pronounced for 99.9th percentile latency; however, even these extreme tail latencies can be ameliorated if utilization is reduced in the machine by offering load from less sensors. For example, for 500 simulated sensors, 99.9th percentile latency is minimal for raw data requests, and under 1 sec for live data requests. It is expected that latency of user interactions on the website be kept within a few seconds. This requirement can be fulfilled by the data platform even with the targeted 80% utilization load offered by 2,000 simulated sensors and at extreme latency percentiles. Moreover, the latency of raw data requests is often substantially below 0.5 sec, while that of live data requests is often below 1 sec at 2,000 simulated sensors.

7 RELATED WORK AND DISCUSSION

This section discusses research efforts related to our work. To the best of our knowledge, the literature lacks contributions explicitly justifying why and discussing how AODBs meet the challenges of IoT data platforms. However, earlier approaches have explored how to support different aspects of IoT data management employing a variety of data-centric system abstractions.

Approaches based on data stream management systems (DSMSs), in particular, are a commonly used solution in the context of IoT systems [11, 21, 31, 48]. DSMSs are apt at transforming multiple input streams, through a topology of data flow operators, into output streams containing, e.g., alerts and notifications for further processing. One challenge in these systems has been flexibility in responding to dynamically changing conditions as typical in IoT, e.g., through the addition or removal of input sources [49]. Actor-based streaming implementations have been proposed to address these concerns [5, 39], as adaptability is a built-in feature of the actor model [1]. However, a problem with data streaming approaches has been to additionally provide for data storage and online queries [22, 26]. In the context of IoT, AllJoyn Lambda explored a lambda architecture for IoT data storage analytics. Adnan et al. combined streaming and historical data to perform predictions in IoT systems based on machine learning models [2]. In contrast to AODBs, which abstract storage management with virtual actors and storage annotations, these approaches require developers to master complex APIs, often spanning across data stream and database systems. Moreover, while these systems provide for low-latency alerts, online queries are non-trivial to support efficiently. By contrast, an AODB acts as an in-memory, programmable cache where complex analyses can be executed in parallel over the encapsulated state of multiple actors employing user-defined methods.

Another class of solutions explored by previous work is that of cloud-centric actor-based IoT middleware, such as Ptolemy Accessor [42] and Calvin [41]. In these systems, every IoT device is modeled as an actor so that those multiple IoT components can be easily integrated into a potentially complex edge-cloud system. However, these middleware platforms lack integration with data management features that are central to an IoT data platform, such as efficient data storage with support for multi-tenancy and data protection. In addition to middleware, specific IoT applications have also been directly built over actor runtimes [4, 38]. For example, Pegasus is a cloud-based project aimed at gathering data with high-altitude balloons [23]. The system employs the Orleans actor runtime so as to simplify the development process of building a parallel, interactive and dynamic cloud service [15]. In contrast to our work, these previous implementation efforts do not provide any insights on data modeling decisions, nor do they analyze case studies to connect requirements for IoT data platforms with the necessary support from an actor-based solution. Even though there have been explorations of how to employ actors as a modeling construct for cyber-physical systems [25], none of these investigations fully satisfy our data platform requirements, namely storing, managing and processing large-scale data as well as providing for high scalability, real-time computation, data protection, and access control.

In line with the vision of Bernstein et al. [17], we argue that the integration of data management features into actor runtimes can help meet the increasing demand for scalable, low-latency data platforms. Recently, a relational actor programming model has been proposed for in-memory databases and realized in ReactDB [46]. Even though ReactDB shows that the actor model can be used to provide for low latency in databases, we did not consider it as a possible option for our data platform because it is a research prototype and currently not available for production use. Furthermore, in previous work combining actors and databases, there is no systematic review of how to model and structure IoT data platforms, nor discussion of the implementation of such IoT platforms employing an AODB approach. Our work matches the characteristics of an AODB with the requirements and challenges of IoT data platforms, showing how recent research on AODBs can be the basis for a new methodology to model and build IoT data platforms.

8 CONCLUSIONS AND FUTURE WORK

IoT systems require adequate data platforms for handling data storage, management, query and preservation. The modeling and deployment of these platforms remain an open research challenge. In this paper, we presented a generic actor-oriented data platform modeling approach for IoT data platforms, showing how actor-oriented databases can address challenges in the management of IoT data. Our discussion of challenges and their solution was showcased via two distinct case studies, specifically systems for structural health monitoring and beef cattle tracking and tracing. Our contribution covered the detailed modeling of these two real-world case studies and presented the entities and the patterns used to represent their dynamic behavior. This was accompanied by a discussion of modeling challenges, together with our recommendation of technologies and methodologies to address these challenges. As part of this work, we developed a prototype of a structural health monitoring system, which was transitioned to SenMoS. This prototype was validated through experiments demonstrating scalability as more simulated sensors are added as well as low latency in interactive query functions.

We believe that adopting AODBs for IoT systems can help attain the full potential of IoT by extending the reach, scalability, and maintainability of IoT data platforms. As future work, we plan to explore data integration issues in IoT data platforms modeled with the AODB approach, and devise approaches to enforce constraints in AODBs.

ACKNOWLEDGMENTS

Work partially conducted in the context of the Future Cropping partnership [30], supported by Innovation Fund Denmark. Experimental evaluation supported by the AWS Cloud Credits for Research program. In addition, this work was partly supported by the International Network Programme project "Modeling and Developing Actor Database Applications", funded by the Danish Agency for Science and Higher Education (number 7059-000528) and by FAPESP CEPID CCES 13/08293- 7. Additional funding provided by FAPESP project 17/02325-5 and by CNPq-Brazil.

REFERENCES

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

[2] Adnan Akbar, Abdullah Khan, Francois Carrez, and Klaus Moessner. 2017. Predictive analytics for complex IoT data streams. *IEEE Internet of Things Journal* 4, 5 (2017), 1571–1582.

[3] Akka 2018. Akka Documentation. https://doc.akka.io/docs/akka/1.3.1/Akka.pdf.

[4] Akka IoT cases 2018. Akka Documentation, Version 2.5.17, IoT example use case. https://doc.akka.io/docs/akka/2.5/guide/tutorial.html.

[5] Akka Streams 2018. Akka Streams version 2.5.18. https://doc.akka.io/docs/akka/2.5/stream/.

[6] AQStreamProvider 2019. Azure Queue (AQ) Stream Provider. https://dotnet.github.io/orleans/Documentation/streaming/stream_providers.html?q%3Dqueue%23azure-queue-aq-stream-provider%0A.

[7] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. 2008. Multi-tenant Databases for Software As a Service: Schema-mapping Techniques. In *Proc. ACM International Conference on Management of Data (SIGMOD).* 1195–1206.

[8] AWS 2017. Amazon Web Services. https://aws.amazon.com/.

[9] AWS DynamoDB 2018. Amazon DynamoDB. https://aws.amazon.com/dynamodb/.

[10] AWS EC2 2018. Amazon Web Services EC2 Instances. https://aws.amazon.com/ec2/instance-types/.

[11] AWS IoT 2018. AWS IoT Core. https://aws.amazon.com/iot-core/.

[12] AWS RDS 2018. Amazon Relational Database Service. https://aws.amazon.com/rds/.

[13] Debasis Bandyopadhyay and Jaydip Sen. 2011. Internet of things: Applications and challenges in technology and standardization. *Wireless Personal Communications* 58, 1 (2011), 49–69.

[14] Philip A. Bernstein. 2018. Actor-Oriented Database Systems. In *Proc. IEEE International Conference on Data Engineering (ICDE).* 13–14.

[15] Philip A Bernstein and Sergey Bykov. 2016. Developing cloud services using the orleans virtual actor model. *IEEE Internet Computing* 5 (2016), 71–75.

[16] Philip A Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. Orleans: Distributed virtual actors for programmability and scalability. *MSR-TR-2014–41* (2014).

[17] Philip A Bernstein, Mohammad Dashti, Tim Kiefer, and David Maier. 2017. Indexing in an Actor-Oriented Database.. In *Proc. Biennial Conference on Innovative Data Systems Research (CIDR).*

[18] Kasper Myrtue Borggren. 2018. *Scalable Structural Health Monitoring Data Platform using Actors as a Database.* Master's thesis. University of Copenhagen, Copenhagen Denmark.

[19] Shawn Bowers and Bertram Ludäscher. 2005. Actor-oriented design of scientific workflows. In *Proc. International Conference on Conceptual Modeling (ER).* Springer, 369–384.

[20] Thomas B Breen. 2009. System and method for updating geo-fencing information on mobile devices. US Patent 7,493,211.

[21] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair JG Gray. 2010. Enabling ontology-based access to streaming data sources. In *Proc. International Semantic Web Conference (ISWC).* 96–111.

[22] Sirish Chandrasekaran and Michael J. Franklin. 2004. Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams. In *Proc. International Conference on Very Large Data Bases (VLDB).* 348–359.

[23] Athima Chansanchai. 2018. Pegasus II mission sends balloon high above Earth and invites you along for an Internet of Things ride. https://news.microsoft.com/features/pegasus-ii-mission-sends-balloon-high-above-earth-and-invites-you-along-for-an-internet-of-things-ride/.

[24] Mario GCA Cimino, Beatrice Lazzerini, Francesco Marcelloni, and Andrea Tomasi. 2005. Cerere: an information system supporting traceability in the food supply chain. In *Proc. IEEE International Conference on E-Commerce Technology Workshops.* 90–98.

[25] Patricia Derler, Edward A. Lee, and Alberto L. Sangiovanni-Vincentelli. 2012. Modeling Cyber-Physical Systems. *Proc. IEEE* 100, 1 (2012), 13–28.

[26] Nihal Dindar, Peter M. Fischer, Merve Soner, and Nesime Tatbul. 2011. Efficiently correlating complex events over live and archived data streams. In *Proc. ACM International Conference on Distributed Event-Based Systems (DEBS).* 243–254.

[27] Tamer Eldeeb and Phil Bernstein. 2016. *Transactions for Distributed Actors in the Cloud.* Technical Report MSR-TR-2016-1001. Microsoft Research. https://www.microsoft.com/en-us/research/publication/transactions-distributed-actors-cloud-2/

[28] Embrapa 2018. Brazilian Agricultural Research Corporation - A Embrapa. https://www.embrapa.br/en/international.

[29] Erlang 2017. Build massively scalable soft real-time systems. https://www.erlang.org/.

[30] Future Cropping 2017. Future Cropping partnership website. https://futurecropping.dk/en/.

[31] Google 2018. Google IoT Core. https://cloud.google.com/iot-core/.

[32] GS1 2018. Global Standards One. https://www.gs1.org/.

[33] IBM 2018. IBM Food Trust: trust and transparency in our food. https://www.ibm.com/blockchain/solutions/food-trust.

[34] Ralph Kimball and Margy Ross. 2013. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling* (3rd ed.). Wiley Publishing.

[35] Andréia Akemi Kondo, Claudia Bauzer Medeiros, Evandro Bacarin, and Edmundo Roberto Mauro Madeira. 2007. Traceability in Food for Supply Chains.. In *Proc. International Conference on Web Information Systems and Technologies (WEBIST).* 121–127.

[36] Chen Luo and Michael J. Carey. 2018. Efficient Data Ingestion and Query Processing for LSM-Based Storage Systems. *CoRR* abs/1808.08896 (2018). arXiv:1808.08896

[37] A Mousavi, M Sarhadi, A Lenk, and S Fawcett. 2002. Tracking and traceability in the meat processing industry: a solution. *British Food Journal* 104, 1 (2002), 7–19.

[38] Orleans 2018. Who Is Using Orleans? http://dotnet.github.io/orleans/Community/Who-Is-Using-Orleans.html.

[39] Orleans Streams 2018. Orleans Streams. https://dotnet.github.io/orleans/Documentation/streaming/index.html.

[40] OrleansGrainCallFilters 2018. Grain Call Filters. https://dotnet.github.io/orleans/Documentation/grains/interceptors.html.

[41] Per Persson and Ola Angelsmark. 2015. Calvin–merging cloud and iot. *Procedia Computer Science* 52 (2015), 210–217.

[42] Ptolemy 2018. The Ptolemy Project: Accessors. https://ptolemy.berkeley.edu/accessors/.

[43] Daniel Diaz Sanchez, R Simon Sherratt, Patricia Arias, Florina Almenarez, and Andres Marin. 2015. Enabling actor model for crowd sensing and IoT. In *Proc.IEEE International Symposium on Consumer Electronics (ISCE).* 1–2.

[44] SEGES 2018. SEGES Landbrug & Fødevarer F.m.b.A. website. https://www.seges.dk/en.

[45] SenMos 2018. SenMoS: your sensor monitoring system. https://senmos.dk/.

[46] Vivek Shah and Marcos Antonio Vaz Salles. 2018. Reactors: A Case for Predictable, Virtualized Actor Database Systems. In *Proc. ACM International Conference on Management of Data (SIGMOD).* 259–274.

[47] Vivek Shah and Marcos Vaz Salles. 2018. Actor-Relational Database Systems: A Manifesto. *CoRR* abs/1707.06507 (2018).

[48] Zhitao Shen, Vikram Kumaran, Michael J Franklin, Sailesh Krishnamurthy, Amit Bhat, Madhu Kumar, Robert Lerche, and Kim Macpherson. 2015. CSA: Streaming Engine for Internet of Things. *IEEE Data Eng. Bull.* 38, 4 (2015), 39–50.

[49] Ayush Singhal, Rakesh Pant, and Pradeep Sinha. 2018. AlertMix: A Big Data platform for multi-source streaming data. *arXiv preprint arXiv:1806.10037* (2018).

[50] Storebælt 2018. Facts and History. https://www.storebaelt.dk/english/bridge.

[51] Toby J Teorey. 1999. *Database modeling & design.* Morgan Kaufmann.

[52] Transactors dropped 2018. Akka Migration Guide 2.3.x to 2.4.x. https://doc.akka.io/docs/akka/2.4/project/migration-guide-2.3.x-2.4.x.html.

[53] UML 2018. OMG Unified Modeling Language (OMG UML) Version 2.5.1. https://www.omg.org/spec/UML/2.5.1/PDF.

KSQL: Streaming SQL Engine for Apache Kafka

Hojjat Jafarpour
Confluent Inc.
Palo Alto, CA
hojjat@confluent.io

Rohan Desai
Confluent Inc.
Palo Alto, CA
rohan@confluent.io

Damian Guy
Confluent Inc.
London, UK
damian@confluent.io

ABSTRACT

Demand for real-time stream processing has been increasing and Apache Kafka has become the de-facto streaming data platform in many organizations. Kafka Streams API along with several other open source stream processing systems can be used to process the streaming data in Kafka, however, these systems have very high barrier of entry and require programming in languages such as Java or Scala.

In this paper, we present KSQL, a streaming SQL engine for Apache Kafka. KSQL provides a simple and completely interactive SQL interface for stream processing on Apache Kafka; no need to write code in a programming language such as Java or Python. KSQL is open-source, distributed, scalable, reliable, and real-time. It supports a wide range of powerful stream processing operations including aggregations, joins, windowing, sessionization, and much more. It is extensible using User Defined Functions (UDFs) and User Defined Aggregate Functions (UDAFs). KSQL is implemented on Kafka Streams API which means it provides exactly once delivery guarantee, linear scalability, fault tolerance and can run as a library without requiring a separate cluster.

1 INTRODUCTION

In recent years, the volume of data that is generated in organizations has been growing rapidly. From transaction log data in e-commerce platforms to sensor generated events in IoT systems to network monitoring events in IT infrastructures, capturing large volumes of data reliably and processing them in a timely fashion has become an essential part of every organization. This has resulted in an emerging paradigm where organizations have been moving from batch oriented data processing platforms towards realtime stream processing platforms.

Initially developed at LinkedIn, Apache Kafka is a battle hardened streaming platform that has been used to capture trillions of events per day [2] [16]. Apache Kafka has become the de-facto streaming platform in many organizations where it provides a scalable and reliable platform to capture and store all the produced data from different systems. It also efficiently provides the captured data to all the systems that want to consume it. While capturing and storing streams of generated data is essential, processing and extracting insight from this data in timely fashion has become even more valuable. Kafka Streams API along with other open source stream processing systems have been used to perform such real time stream processing. Such real time stream processing systems have been used to develop applications such as Streaming ETL, anomaly detection, real time monitoring and many more. Many of these stream processing systems require users to write code in complex languages such as Java or Scala and can only be used by users who are fluent in such languages. This is a high barrier of entry that limits the usability of such systems.

Motivated by this challenge, in this paper we present KSQL, a streaming SQL engine for Apache Kafka that offers an easy way to express stream processing transformations[8]. While the existing open source stream processing systems require expression of stream processing in programming languages such as Java, Scala or Python or offer limited SQL support where SQL statements should be embedded in the Java or Scala code, KSQL offers an interactive environment where SQL is the only language that is needed. KSQL also provides powerful stream processing capabilities such as joins, aggregations, event-time windowing, and many more.

KSQL is implemented on top of the Kafka Streams API which means you can run continuous queries with no additional cluster; streams and tables are first-class constructs; and you have access to the rich Kafka ecosystem. Similar to addition of SQL to other systems such as Apache Hive[17] and Apache Phoenix[4], we believe that introduction of SQL for stream processing in Kafka will significantly broaden the users base for stream processing and bring the stream processing to the masses.

The rest of the paper is organized as the following. In the next section we provide a brief overview on Apache Kafka and the Kafka Streams API. Section 3 presents our contribution in design and development of KSQL. We describe data model, basic concepts, query language and the internals of our SQL engine. In Section 4, we present how KSQL can be extended using UDFs and UDAFs. We describe different execution modes for KSQL in Section 5. We present our experimental evaluation results for KSQL in Section 6. Section 7 describes the related work. We present the future work directions and conclude the paper in Section 8.

2 BACKGROUND

KSQL is implemented on top of the Kafka Streams API. In this section we will provide a brief overview on Apache Kafka and the Kafka Streams API.

2.1 Apache Kafka

Apache Kafka is a large-scale distributed publish/subscribe messaging system where data is produced to and consumed from topics [2] [15] [16]. Messages in Kafka include a key and a value. Figure 1 depicts the anatomy of a topic in Kafka. Each topic consists of several partitions where messages are assigned to based on their key. Each partition is an ordered, immutable sequence of records that is continually appended to a structured commit log. To achieve fault tolerance, partitions are replicated across a configurable number of servers, called brokers, in a Kafka cluster. One broker for each partition acts as the leader and zero or more brokers act as followers.

Producers publish data to their desired topics by assigning the messages to the specific partition in the topic based on the message key. To consume the published data to a topic, consumers

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

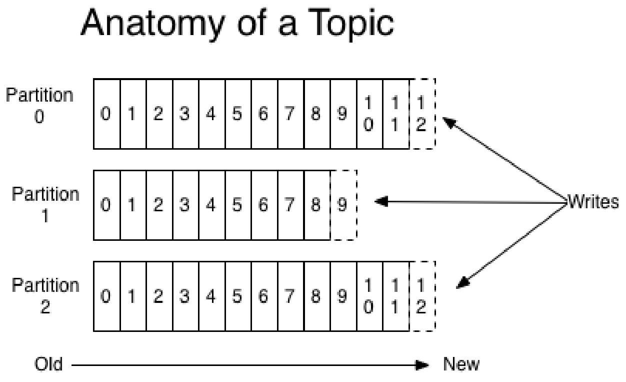


Figure 1: Anatomy of a Kafka topic

form consumer groups where each published message will be delivered to one instance in the consumer group. Figure 2 shows two consumer groups that consume messages from a topic with four partitions in a Kafka cluster with two brokers.

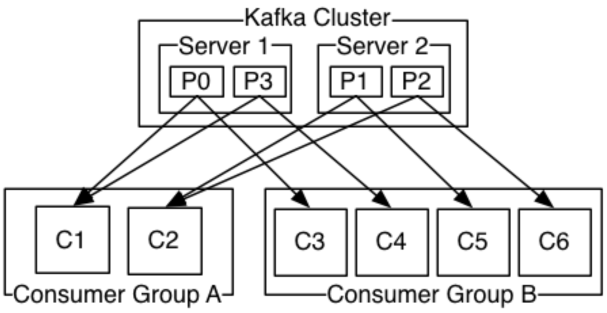


Figure 2: Two consumer groups reading from a topic with four partitions.

A consumer groups can expand by adding more members to it. It can also shrink when group members fail or are removed explicitly. Whenever, a consumer group changes, Kafka cluster will go through a rebalancing process for the consumer group to guarantee every partition in the topic will be consumed by one instance in the consumer group. This is done by Kafka group management protocol which is one of the fundamental building blocks of the Kafka streams API as we describe below.

2.2 Kafka Streams API

Kafka Streams API is a Java library that enables users to write highly scalable, elastic, distributed and fault-tolerant stream processing applications on top of Apache Kafka [2]. Unlike other stream processing frameworks that need a separate compute cluster to run stream processing jobs, Kafka streams runs as an application. You can write your stream processing application and package it in your desired way, such as an executable jar file, and run instances of it independently. If you need to scale out your application, you just need to bring up more instances of the app and Kafka streams along with Kafka cluster will take care of distributing the load among the instances. The distribution of load in Kafka streams is done with the help of Kafka group management protocol. Figure 3 depicts the architecture of a Kafka streams app.

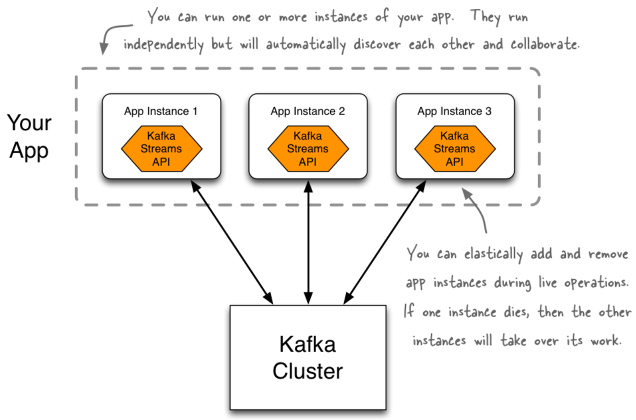

You can run one or more instances of your app. They run independently but will automatically discover each other and collaborate.
Your App
App Instance 1
Kafka Streams API
App Instance 2
Kafka Streams API
App Instance 3
Kafka Streams API
You can elastically add and remove app instances during live operations. If one instance dies, then the other instances will take over its work.
Kafka Cluster

Figure 3: Two consumer groups reading from a topic with four partitions.

A typical Kafka streams application will read from one or more Kafka topic and process the data and writes the results into one or more Kafka topics. Kafka streams app uses the same data model as Kafka where messages include a key and a value along with a timestamp and offset of the message in its corresponding partition. The processing logic in a Kafka streams app is defined as a processing topology that include source, stream processor and sink nodes. The processing model is one record at a time where an input record from the source is processed by passing through the whole topology before the next recored is processed. Kafka streams provides powerful stream processing capabilities such as joins, aggregations, event-time windowing, sessionization and more. Operations such as join and aggregation are done based on the message key. Kafka streams uses intermediate Kafka topics to perform shuffle for operations such as aggregation and join that need to colocate data based on a key. For instance, if two streams are being joined and the join key is not the same as the message key for both streams, Kafka streams repartitions both stream topics based on the join key and produces new intermediate topics where message key is the same as the join. This will ensure the colocation of the records with the same key that can be joined at the same node.

Kafka streams provides *stateful* stream processing through the so-called *state stores*. The state stores exist in every instance of the streaming application and are used to store the state in operations such as join and aggregation in a distributed fashion. By default Kafka streams uses RocksDB[9] to store application state, however, any in-memory hash map or other data structures can be plugged in.

3 KSQL

In this section we present KSQL, streaming SQL engine for Kafka. As mentioned KSQL uses Kafka streams to run the user queries, therefore, it inherits many properties of Kafka streams.

3.1 Data Model

As mentioned a message in a Kafka topic consists of a key and a value. To keep the messages generic, Kafka does not assume any specific format for the messages and both key and value are treated as array of bytes. In addition to the key and value, a message also includes a timestamp, a partition number that it belongs to and the offset value in the corresponding partition. In order to use SQL on top of Kafka topics we need to impose the relational data model, *schema*, on the value part of messages in Kafka topics. All of the message values in a topic should conform to the associated schema to the topic. The schema defines a message value as a set of columns where each column conforms to the defined data type. Currently, we support the primitive types of BOOLEAN, INTEGER, BIGINT, DOUBLE and VARCHAR along with the complex types of ARRAY, MAP and STRUCT. We plan to add DECIMAL, DATE and TIME types in future. KSQL supports nested column type using the STRUCT type. The fields in a ARRAY, MAP and STRUCT types themselves can be any of the supported types including complex types. As you can see, there is no limit in the level of nesting and users can have as many levels of nesting as they desire. The schema of message values for Kafka topic is used for serialization and deserialization of message values.

3.2 Basic Concepts

KSQL provides streaming SQL for Kafka topics meaning that you can write continuous queries that run indefinitely querying future data. There are two basic concepts in KSQL that users can use in their queries, stream and table. Depending on how we interpret the messages in a Kafka topic we can define streams or tables over Kafka topics in KSQL.

If we consider the messages arriving into a topic as independent and unbounded sequence of structured values, we interpret the topic as a ***stream***. Messages in a stream do not have any relation with each other and will be processed independently.

On the other hand, if we consider the messages arriving into a topic as an evolving set of messages where a new message either updates the previous message in the set with the same key, or adds a new message when there is no message with the same key, then we interpret the topic as a ***table***. Note that a table is an state-full entity since we need to keep track of the latest values for each key. In other words, if we interpret the messages in a topic as a **change log** with a state store that represent the latest state, then we interpret the topic as a table.

As an example consider we store the page view events for a website in a Kafka topic. In this case, we should interpret the topic as a stream since each view is an independent message. On the other hand, consider we are storing user information is a Kafka topic where each message either adds a new user if it is not stored already or updates the user information if we already have stored the user. In this case, we should interpret the topic as a table. Note that at any moment, the table should have the most up to date information for every user.

KSQL also provides *windowed* stream processing where you can group records with the same key to perform stateful processing operations such as aggregations and joins. Currently, KSQL supports three types of windows:

- **Tumbling window** which are time-based, fixed-sized, non-overlapping and gap-less windows
- **Hopping window** which are time-based, fixed-sized and overlapping windows
- **Session window** which are session-based, dynamically-sized, non-overlapping and data-driven windows

Note that the results of windowed aggregations are tables in KSQL where we need to keep the state for each window and aggregation group and update them upon receiving new values.

KSQL also support join operation between two streams, a stream and a table or two tables. The stream-stream join always requires a sliding window since we should prevent the size of the state store growing indefinitely. Every time a new message arrives to either of the streams, the join operation will be triggered and the new message for each matching message from the other stream within the join window will be produced. KSQL supports INNER, LEFT OUTER and FULL OUTER join operations for two streams. The RIGHT OUTER join can be implemented via the LEFT OUTER join by simply change the left and right sides of the join. Joining a stream with a table is a stateless operation where each new message in the stream will be matches with the table resulting in emission of zero or one message. Finally, joining two tables in KSQL is consistent with joining them in relational databases if we materialize both of them. KSQL supports INNER, LEFT OUTER and FULL OUTER joins for two tables.

3.3 Query Language

KSQL query language is a SQL-like language with extensions to support stream processing concepts. Similar to the standard SQL language, we have DDL and DML statements. DDL statements are used to create or drop streams or tables on top of existing Kafka topic. The followings are two DDL statements to create a pageviews stream and a users table:

```
CREATE STREAM pageviews (viewtime BIGINT,
 userid VARCHAR, pageid VARCHAR) WITH
 (KAFKA_TOPIC='pageviews_topic',
 VALUE_FORMAT='JSON');

CREATE TABLE users (registertime BIGINT,
  gender VARCHAR,
  regionid VARCHAR,
  userid VARCHAR,
address STRUCT<
  street VARCHAR, zip INTEGER
>
 ) WITH (
  KAFKA_TOPIC='user_topic',
  VALUE_FORMAT='JSON',
KEY='userid'
 );
```

Note that in addition to defining the schema for the stream or table we need to provide information on the Kafka topic and the data format in the WITH clause. After declaring streams and tables, we can write continuous queries on them.

Unlike standard SQL statements where the queries return finite set of records as the result, in streaming systems we have continuous queries and therefore the results also will be continuous while the query runs. To address this, KSQL provides two types of query statements. If the results of the query are stored as a new stream or table into a new Kafka topic we use **CSAS(CREATE STREAM AS SELECT)** , **CTAS(CREATE TABLE AS SELECT)** or **INSERT INTO** statements depending on the type of the query results. For instance, the following statement enriches the pageviews stream with extra user information by joining it with the users table and only passes the records with regionid = 'region 10'. The result is a new stream that we call enrichedpageviews:

```
CREATE STREAM enrichedpageviews AS
 SELECT * FROM pageviews LEFT JOIN
 users ON pageviews.userid = users.userid
 WHERE regionid = 'region 10';
```

On the other hand, the following CTAS statement creates a table that contains the pageviews for each user in every 1 hour. Here we use an aggregate query with tumbling window with size of 1 hour.

```
CREATE TABLE userviewcount AS
 SELECT userid, count(*)
 FROM pageviews
 WINDOW TUMBLING (SIZE 1 HOUR)
 GROUP BY useid;
```

Note that the results of the above query will be continuous count values for each user and window that will be stored in a Kafka topic. Every time we receive a new record the current count value for the corresponding userid and window will be updated and the new updated value will be written to the topic. As it can be seen the result will be a change log topic.

Depending on the execution model that we will discuss in the later section, KSQL also provides query manipulation statements where user can submit continuous queries, list the currently running continuous queries and terminate the desired ones.

3.4 KSLQ Engine

As mentioned, KSQL uses Kafka streams to run the streaming queries. The main responsibility of the KSQL engine is to compile the KSQL statements into Kafka streams apps that can continuously run and process data streams in Kafka topics. To achieve this KSQL has a metastore component that acts as a system catalog storing information about all the available streams and tables in the system. Currently metastore is an internal component in the KSQL engine. Depending on the execution mode the metastore can be backed by a Kafka topic to provide fault tolerance.

Figure 4 depicts the steps that are taken in the engine to compile KSQL statements into Kafka streams applications to run. As it can be seen, the first step is to parse the statements where the KSQL parser generates an Abstract Syntax Tree (AST). Using the metastore, the generated AST will be analyzed and the unresolved columns references will be resolved. This include detecting the column types along with resolving expression types in the queries along with extracting different components of a query including source, output, projection, filters, join and aggregation. After analyzing each query, we will build a logical plan for it. Logical plan is a tree structure where the nodes are instances of PlanNode class. Currently, we can have the following node types in KSQL logical plan: SourceNode, JoinNode, FilterNode, ProjectNode, AggregateNode and OutputNode. The leaf node(s) are of SourceNode type and the root node is OutputNode type. As it is indicated in Figure 4. rule-based optimization techniques can be applied to the generated logical plan, however, at the moment we do not apply any rule to the logical plan other than pushing down the filters.

The final step is to generate a physical execution plan from the logical plan. The physical plan in KSQL is a Kafka streams topology that runs the stream processing logic. We use the higher level topology structure that is called Kafka streams DSL[2]. Kafka streams defines two fundamental building blocks, **KStream** and **KTable** which are synonymous to stream and table in KSQL. Indeed, A KSQL stream represents a KStream along with a schema. Similarly, a KSQL table represents a KTable along with the associated schema. Kafka streams DSL also provides operations such as map, join, filter, aggregate, etc that convert KStreams/KTables into new KStream/KTables. These operations work on key and value of Kafka messages where there is no assumption on the schema of message value. KSQL defines similar operations on streams and tables, however, in KSQL we impose the proper schema on the message value.

KSQL engine also is responsible for keeping the metastore and queries in the correct state. This includes rejecting statements when they result in incorrect state in the engine. Dropping streams or tables while there are queries that are reading from or writing into them is one of the cases that would result the system to go into an incorrect state. A stream or table can only be dropped if there is no query reading from or writing into it. KSQL engine keeps track of queries that read from or write into a stream or table in the metastore and if it receives a **DROP** statement for s stream or table that is still being used, it will reject the **DROP** statement. Users should make sure that all of the queries that use a stream or table are terminated before they can drop the stream or table.

4 UDFS AND UDAFS

Although standard SQL statements provide a good set of capabilities for data processing, many use cases need to perform more complex and custom operation on data. By using functions in queries SQL systems enhance their data processing capabilities. KSQL also provide an extensive set of built in scalar and aggregate functions that can be use in queries. However, to even make KSQL more extensible, we have added capability of adding custom User Defined Functions (UDFs) and User Defined Aggregate Functions (UDTFs).

4.1 User Defined Functions

UDF functions are scalar functions that take one input row and return one output value. These functions are stateless, meaning there is no state is maintained between different function calls. Currently, KSQL supports UDFs written in Java. Implementing a new UDF is very straightforward using only two annotations. A Java class annotated by *@UdfDescription* annotation will be considered as a UDF containing class. User provide the name of the UDF by setting the *name* parameter of the *@UdfDescription* annotation. Any function in this class that is annotated by *@Udf* annotation will be considered as a UDF that can be used in any query similar to any other built in function. The following is an example UDF that implements multiplication.

```
@UdfDescription(name = "multiply", description =
     "multiplies 2 numbers")
public class Multiply {

  @Udf(description = "multiply two non-nullable INTs.")
  public long multiply(final int v1, final int v2) {
    return v1 * v2;
  }
}
```

After implementing the UDFs, users can package them in a JAR file and upload it to the designated directory in the KSQL engine where the functions will be loaded from when the KSQL engine starts up. The following query shows how the above UDF can be used in a query:

```
CREATE STREAM test AS
 SELECT multiply(col1, 25)
```

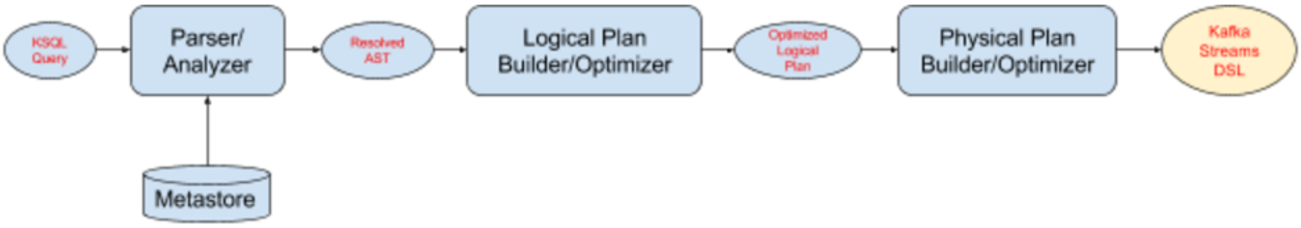


Figure 4: Steps to convert KSQL statements into Kafka streams apps.

```
 FROM inputStream;
```

4.2 User Defined Aggregate Functions

Aggregate functions are applied to a set of rows and compute a single value for them. Similar to the UDFs, KSQL UDAFs are implemented in Java using annotations. To create a new UDAF, users need to create a class that is annotated with *@UdafDescription*. Methods in the class that are used as a factory for creating an aggregation must be *public* and *static*, be annotated with *@UdafFactory*, and must return an instance of *Udaf* class. The instances of a textitUdaf class should implement *initialize*, *aggregate* and *merge* methods. The following is an example UDAF that will perform *sum* operation over double values:

```
@UdafDescription(name = "my_sum", description = "sums")
public class SumUdaf {
@UdafFactory(description = "sums double")
  public static Udaf<Double, Double> createSumDouble() {
    return new Udaf<Double, Double>() {
      @Override
      public Double initialize() {
        return 0.0;
      }

      @Override
      public Double aggregate(final Double aggregate,
           final Double val) {
        return aggregate + val;
      }

      @Override
      public Double merge(final Double aggOne, final
           Double aggTwo) {
        return aggOne + aggTwo;
      }
    };
  }
}
```

Similar to UDFs, UDAFs should be packaged in a JAR file and uploaded to the designated folder in the KSQL engine so the functions can be loaded at the engine start up.

We plan to add support for User Defined Table Functions (UDTFs) in near future.

5 EXECUTION MODES

As discussed above, KSQL engine creates Kafka streams topologies that execute the desired processing logic for Kafka topics. Therefore, running KSQL queries is the same as running Kafka streams topologies. Currently, KSQL provides three different execution modes that we describe here.

5.1 Application Mode

The application mode is very similar to running a Kafka streams app as described earlier. To deploy and run your queries, you need to put them in a query file and pass it as an input parameter to the KSQL executable jar. Depending on the required resources, you determine the number of instances that your application needs and similar to a Kafka streams execution model you will instantiate the instances by running the KSQL jar with the query file as input parameter. The deployment process can be done manually or through third party resource managers such as Mesos[3] or Kubernetes[10]. Note that you don't need any extra processing cluster and the only thing you need is to run your KSQL app by bringing up desired number of instances independently. Everything else will be handled by KSQL and Kafka streams.

5.2 Interactive Mode

KSQL also provides an interactive execution mode where users can interact with a distributed service through a REST API. One way of using the provided REST API is to use KSQL CLI which includes a REST client that sends user requests to the service and receives the response. The building block of the interactive mode is KSQL server that provides a REST end point for users to interact with the service and also KSQL engine to execute the user queries. Figure 5 depicts the architecture of the KSQL service with three servers in the interactive client-server execution mode.

Each KSQL server instance includes two components, the KSQL engine and the REST server. The KSQL service uses a special Kafka topic, **KSQL command topic**, to coordinate among the service instances. When a service instance is started, it first checks the Kafka cluster for the command topic. If the topic does not exists it creates a new command topic with a single partition and a configurable number of replications. All of the service instances then subscribe to the command topic. User interacts with the service by connecting to REST endpoint on one of the instances. The **KSQL command topic** has only one partition to ensure the order of KSQL statements for all server is exactly the same. The **KSQL command topic** can have more than one replicas to prevent loss of KSQL statements in presence of failure. The figure shows the KSQL CLI that connects to either of the instances.

When user submits a new KSQL statement through the REST endpoint, the instance that receives the request will append it to the KSQL command topic. The KSQL engine components in all of the instances will pull the new statement from the command topic and execute the statement concurrently. For instance, consider the following statement is submitted to the service through one of the instances and appended to the command topic.

```
CREATE TABLE userviewcount AS
 SELECT userid, count(*)
 FROM pageviews
```

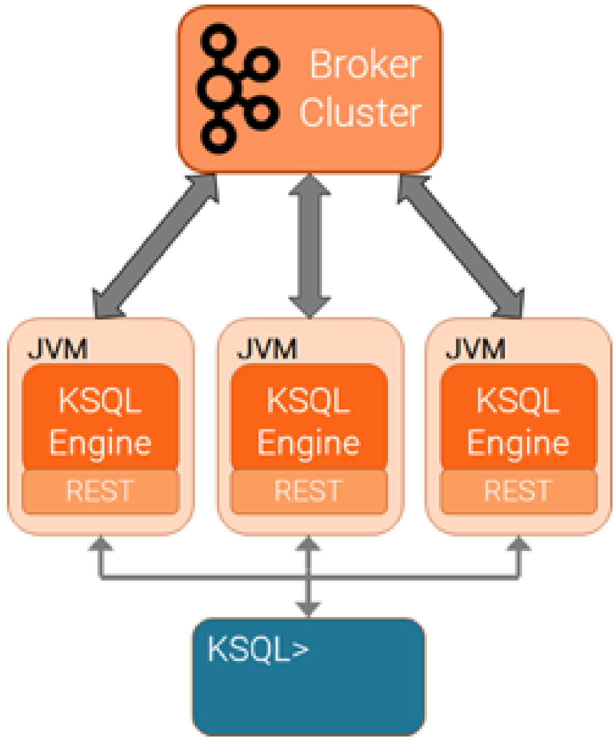


Figure 5: A KSQL interactive service deployment with three server instances.

```
WINDOW TUMBLING (SIZE 1 HOUR)
GROUP BY useid;
```

All of the instances will use the KSQL engine to start the processing of the query and as mentioned above each instance will process portion of the input data from the *pageviews* topic. The service instances continue running the queries until they are explicitly terminated using *TERMINATE* statement.

The KSQL service provides both elasticity and fault tolerance. Service instances can be added or removed independently, depending on the load and performance requirements. When a new instance is started it subscribes to the command topic and fetches all the existing KSQL statements from the command topic and starts executing them. When the new instance starts executing an existing query a rebalance process is triggered and the execution load will be redistributed among the existing instances. Note that the rebalance protocol is handled by the underlying Kafka consumers in the Kafka streams and is transparent for the KSQL service instance. After all of the existing continuous queries start running on the new instance too, it starts listening to the command topic for new queries. Similarly when an instance fails or terminated a rebalance process among the remaining instances of the service happens and the load of the removed instance will be distributed among the remaining instances. Even if all the instances fail and the whole system is restarted, the instances will pick up all the existing KSQL queries from the command topic when they come back online and the whole system will continue processing the queries from the point they were left.

5.3 Embedded Mode

The third execution mode available in KSQL is the embedded mode where we can embed KSQL statements in a Kafka streams application. As mentioned Kafka streams apps are Java programs that use the Kafka streams as library. KSQL provides a **KSQLContext** class with a **sql()** method where KSQL statements can be passed to run in the embedded engine. The following code snippet show a very simple example of using the embedded mode.

```
KSQLContext ksqlContext = new KSQLContext();
ksqlContext.sql("CREATE STREAM pageviews
 (viewtime BIGINT, userid VARCHAR,
  pageid VARCHAR) WITH
  (KAFKA_TOPIC='pageviews_topic',
  VALUE_FORMAT='JSON');");
ksqlContext.sql("CREATE STREAM pageviewfilter
 AS SELECT * FROM pageviews
  WHERE userid LIKE '\%10';");
```

Similar to the Kafka streams apps, in order to execute the queries in the embedded mode, you need to package the application and run instances of it. The deployment can be done through a range of available options such as manually bringing up instances or using more sophisticated tools such as Mesos or Kubernetes.

6 EXPERIMENTAL EVALUATION

6.1 Methodology

We want to evaluate KSQL's ability to handle different types of workloads. To do this, we ran a series of tests with different query types and measured the throughput that a single KSQL server can process. Each test case runs for ten minutes and periodically measures throughput in messages / second and bytes / second.

In practice, users will likely run multiple queries that feed into each other to form a streaming pipeline. We've included a multi-query test to measure performance for this scenario.

Finally, we run multiple queries on a pool of KSQL nodes to see how KSQL scales as servers are added.

6.1.1 Load Generation. To generate load against KSQL we ran a modified version of the "'ksql-datagen"' tool that has had some changes made to improve its performance and to allow it to produce different types of workloads. "'ksql-datagen"' is a tool for generating data into a kafka cluster. The tool generates Kafka records conforming to a specified schema. The schema can be one of a set of pre-defined schemas, or the user can specify their own avro schema for the tool to use. Records are written to Kafka using the Java Kafka producer client. For our tests, we extended "'ksql-datagen"' to support multiple threads, and to support a rate-limit parameter (specified in messages produced per second) implemented using a token-bucket algorithm.

6.1.2 Measuring Throughput. We measured throughput using a counter exported by KSQL that provides the number of messages consumed by a given topic. Throughput for a given time period is computed by sampling the counter at two points in time and dividing the difference in consumed messages by the difference in time. For tests that run multiple queries in a pipeline, consumption is measured against the topic that is the source for the final query in the pipeline. This gives us the throughput of the pipeline as a whole.

6.1.3 Environment. We ran our test in AWS EC2. The test environment consists of a Kafka cluster, a KSQL cluster, and a set of nodes that run the load-generator.

The Kafka cluster consists of 5 Kafka nodes, and 1 node for running ZooKeeper. To run Kafka we chose the i3.xlarge instance type, which has 4 2.3GHz VCPUs, 30.5GB of memory, "up-to-10Gbit" of network, and a 950GB NVME local disk which we'll use to store the Kafka topics. We used fio to benchmark the instance storage, and observed 380MBps write throughput, 950MBps read throughput, 2000016K random write IOPS, and 61000 16K random read IOPS. Finally, we ran some network benchmarks using iperf to get an idea of what network performance to expect, and observed 1.24Gbit of network throughput.

Zookeeper runs on an m3.medium.

The KSQL cluster consists of 1-4 nodes, depending on the test case. We chose i3.xlarge instance for KSQL as well, and use the NVME local disk to store the local state stores.

6.2 Test Data

The test data and workload are derived from a use case for analyzing metrics. Metrics are published by services that comprise a distributed application. The queries consume, transform, aggregate and enrich this metric data. The source stream for the metrics has the following schema:

```
ID VARCHAR,
SOURCE VARCHAR,
INSTANCE LONG,
CLUSTERID VARCHAR,
METRIC STRUCT<
    NAME STRING,
    VALUE INTEGER,
    METADATA1 STRING,
    METADATA4 STRING,
    METADATA5 STRING,
    METADATA6 STRING,
    METADATA7 STRING,
    METADATA8 STRING,
    METADATA9 STRING,
    METADATA10 STRING
>
```

For our tests, we produced test data serialized using AVRO. The average size for test records was 220 bytes. The source topic containing the metrics data has 32 partitions.

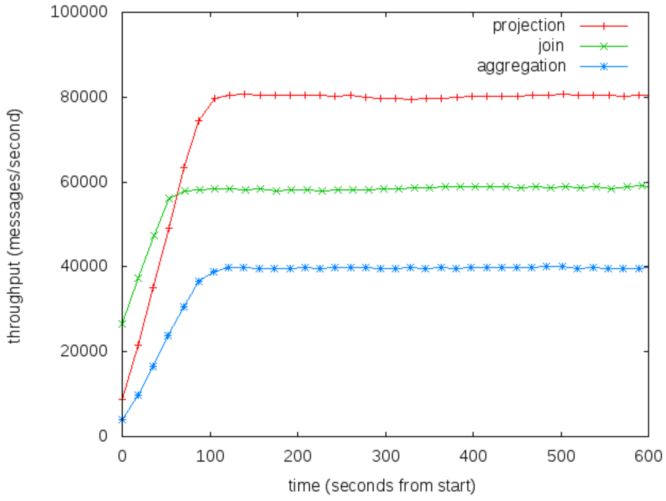


Figure 6: Throughput for Basic Query Types

6.3 Projection

The first type of query we'll evaluate is a basic projection. Users can use similar queries to apply basic stateless transformations and filters to their data. For the test, we'll run the following query:

```
CREATE STREAM SINK
 AS SELECT *
 FROM METRICS_STREAM;
```

Figure 6 depicts the measured throughput over the test run. Throughput starts low, increases as the JVM warms up, and stabilizes just under 90000 messages per second. The main bottleneck for this query type is CPU utilization, which is nearly 100% on the KSQL server during the test run. To discover why, we re-ran the test case while profiling using the YourKit JVM profiler and found the main hot-spots to be deserialization and serialization of the records after reading from, and before writing back to Kafka.

6.4 Aggregation

Next lets look at an aggregation. Aggregations allow the user to group records in a stream or table, and then apply an aggregation function on the grouped records. For this test, we'll run an aggregation to compute the average value of a metric:

```
CREATE TABLE SINK
 AS SELECT
  INSTANCE,
  SUM(METRIC->VALUE) / COUNT(*)
 FROM METRICS_STREAM
 GROUP BY INSTANCE
```

Figure 6 depicts the measured throughput over the test run. Again, the bottleneck is CPU utilization on the KSQL server. In this case, profiling revealed additional overhead from reading from / writing to RocksDB. Each read from RocksDB has to potentially traverse multiple SSTs within the database to find the latest update for the grouping key. The state for aggregations is quite small and fits comfortably within the block cache, so there is no added cost for reading from the kernel cache or decompressing. Writing also adds some cost to write to the memtable and append to the log. The source data for this aggregation is keyed on the grouping column, so there is no additional cost for serializing/deserializing to/from the repartition topic. Aggregations that require a repartition would incur this cost as well.

Space usage on the local disk is quite small, just a few hundred MBs. This makes sense since the state required to compute each aggregation result is small - just the key, sum, and current count, which are all 32-bit or 64-bit integers. Utilization is also quite low - around 50 IOPS and 1MBps. Interestingly, a good portion of the disk usage comes from storing checkpoints for per-task topic offsets rather than from RocksDB itself.

6.5 Windowing

Usually, users want to compute aggregations that correspond to a specific time interval - for example, the average value of a metric over a day or hour. KSQL provides tumbling and hopping windows to allow you to window your aggregation computation using a fixed-size window. A tumbling window computes results for one time interval without any overlap with the previous interval. A hopping window has two arguments: the time interval, and the hop size. Results are returned for windows that are as wide as the interval and advance by the hop size. This lets you approximate a sliding window. In this experiment, we'll compute tumbling and sliding windows and observe the effect on performance:

```
CREATE TABLE SINK
 AS SELECT
  INSTANCE,
  SUM(METRIC->VALUE) / COUNT(*)
 FROM METRICS_STREAM
 WINDOW TUMBLING (
  SIZE 60 SECONDS
 )
 GROUP BY INSTANCE

CREATE TABLE SINK AS SELECT
 INSTANCE,
 SUM(METRIC->VALUE) / COUNT(*)
 FROM METRICS_STREAM
 WINDOW HOPPING(
  SIZE 60 SECONDS,
  ADVANCE BY ? SECONDS
 )
 GROUP BY INSTANCE
```

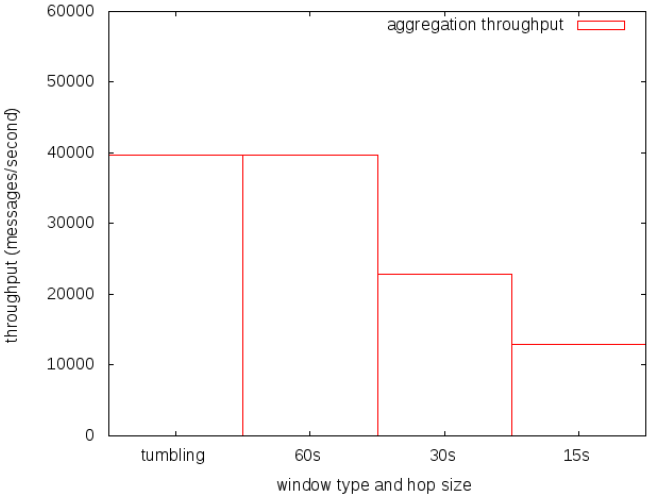


Figure 7: Throughput for Varying Window Sizes

Figure 7 shows that the realized throughput decreases as the number of windows that must be maintained increases. This makes sense - under the hood the aggregation result for each window that a given record maps to is maintained independently. If record arriving in the stream must update 4 windows (as in the ADVANCE BY 15 SECONDS case), then KSQL will do 4 separate reads and write from/to RocksDB. This adds to the CPU cost to process each record.

6.6 Stream Enrichment

Another common use case for KSQL is to enrich a stream of data with a dimension table. In this test, we'll enrich our metric stream by adding information from an instance table to each metric record:

```
CREATE STREAM SINK
 AS SELECT S.*, T.*
 FROM
  METRIC_STREAM S
  LEFT JOIN
  INSTANCE_TABLE S
 ON S.INSTANCE = T.INSTANCE;
```

Figure 6 shows the throughput over time. Again, the primary bottleneck is CPU utilization, which is near 100%. The realized throughput is around 55000 records per second - greater than the aggregation but lower than the projection. This is because to process each record in the stream we must read from RocksDB, but nothing is written back to the state store. Therefore, the CPU utilization per record is lower than that for aggregations, and the realized throughput is higher.

6.7 Multi-Query Application

Usually, KSQL users run multiple queries in a streaming pipeline to form a KSQL "application". To evaluate KSQL in this scenario, we'll build the following test app:

```
CREATE STREAM METRICS_WITH_INSTANCE_INFO
 AS SELECT S.*, T.*
 FROM
  METRIC_STREAM S
  LEFT JOIN
  INSTANCE_TABLE S
 ON S.INSTANCE = T.INSTANCE;

CREATE TABLE AVERAGE_VALUE_BY_PHYSICAL_INSTANCE
 AS SELECT
  T_PHYSICAL_INSTANCE,
  SUM(METRIC->VALUE) / COUNT(*)
 FROM METRICS_WITH_INSTANCE_INFO
 GROUP BY PHYSICAL_INSTANCE;
```

In the app we first enrich the metrics stream with the instance table (as in the previous section), and then compute average metrics grouped by the "physical instance" each instance resides on (which is another field of the instance table schema).

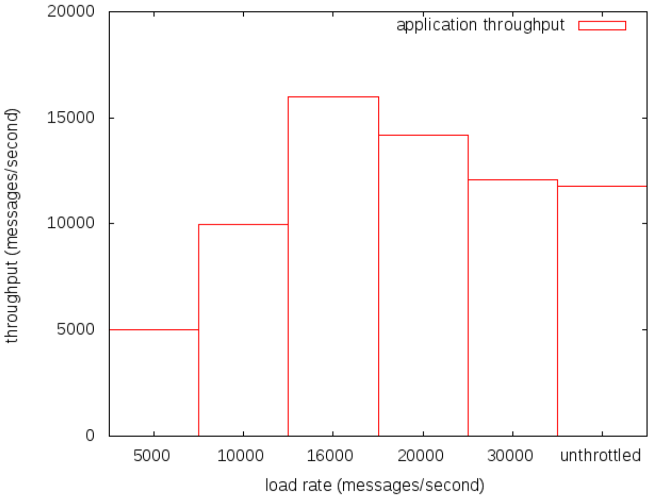


Figure 8: Throughput for Varying Load Rates

To evaluate the possible throughput a KSQL server can process against the app, we generate load at varying rates and measure the consumption rate from METRICS_WITH_INSTANCE_INFO. To understand why its important to rate limit the load generator to determine the maximum possible throughput, lets consider the behavior if load generation was run at a rate that would produce records faster than KSQL could consume them. The "application" presented in Figure 8 consists of 2 queries. The 2 queries have different performance characteristics. Specifically, the join consumes significantly fewer CPU cycles than the aggregation to process the same set of input records. Currently, KSQL does not coordinate execution between separate queries. So, each query would be afforded around half the available CPU time. Ideally, the aggregation should be afforded more CPU than the join. Throttling the input rate gives us a knob to do this indirectly. In practice, a user could accomplish the same thing by scaling out the capacity of their KSQL cluster, or allocating varying numbers of threads to different queries. Figure 8 shows that the maximum throughput a single KSQL server can process running this application is around 16000 records/second.

6.8 Scale-Out

KSQL is designed to scale record throughput as KSQL servers are added. In this section, we'll show that KSQL can process the application we evaluated in the previous section at higher volumes as the cluster grows. We'll evaluate a 2-node and 4-node KSQL cluster. For each configuration, we'll run at 2 times and 4 times the ideal load observed in the previous section, and measure the realized throughput.

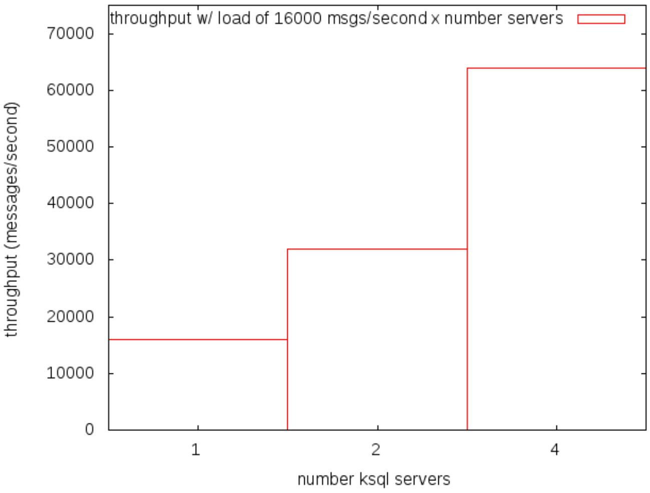


Figure 9: Throughput For Varying Cluster Sizes

Figure 9 shows the results of this experiment. Each bar represents the throughput processed by a KSQL cluster of a given size. Load was generated proportional to the cluster size - 16000 messages per second for the single node cluster, and 32000 messages per second and 64000 messages per second for the 2 and 4 node clusters, respectively. We can see that KSQL is able to process proportionally higher volumes of data as the cluster is grown.

6.9 Future Experiments

The previous sections detailed some initial results of our evaluation of KSQL performance. In the future, we plan to do additional analysis into performance as well as fix some of the issues identified. Latency is a very important metric when evaluating streaming systems that was not measured by our study. After all, one of the primary advantages of stream processing is to be able react to events in real-time. Therefore we plan to measure percentile latency as we invest in evaluating and improving KSQL performance. We also plan to evaluate additional query types. KSQL is a very powerful tool that can perform a wide variety of computations not covered by the evaluation presented here. Some candidate query types for the next round of testing include stream-stream joins and aggregations over session windows.

7 RELATED WORK

Over the past couple of decades stream processing has been one of the main subjects of interest in the data management community. Systems like STREAM [12], Aurora [11] and TelegraphCQ [14] are some of the pioneers in this area.

Increasing demand for scalable streaming engine resulted in development of many systems including open source systems such as Storm [7], Spark [6], Flink [1] and Samza [5]. Although these systems provide a scalable stream processing engine, however, all of them need deployment and management of a complex processing cluster in order to process the streaming jobs. On the other hand, KSQL uses Kafka streams and KSQL queries can be deployed as an application without requiring another complex processing cluster. This significantly simplifies deployment and management of long running streaming jobs in KSQL and provides a wide range of possible deployment options.

SQL language has been the de-facto data management language and many systems support SQL or SQL-like interface. Apache Hive [17] and Apache Spark SQL [13] have shown the effectiveness of using SQL to express computations in batch data processing. They are also great examples of how providing SQL interface can expand the access to scalable data processing systems by eliminating the need to write code in complex programming languages. Some of the open source stream processing frameworks including Apache Spark [6], Apache Flink [1] take this idea to scalable stream processing by providing SQL support. However, the level of SQL support in these system is not at the same level as in KSQL and in order to use SQL for stream processing in these systems users still need to write code in languages such as Java or Scala and embed their SQL statements in their code. On the other hand, in KSQL users describe their processing in KSQL statements and there is no need to write any code in Java or Scala or any other language. This approach significantly simplifies scalable stream processing and makes it available for greater audience.

8 CONCLUSIONS AND FUTURE WORK

In this paper we introduced KSQL, a streaming SQL engine for Apache Kafka. KSQL uses Kafka streams API to run continuous queries and is tightly integrated with Apache Kafka. We showed how a KSQL query is translated into a Kafka streams app in KSQL engine and how we can run KSQL queries in different execution modes. One of the main advantages of KSQL compared to other open source stream processing systems is the elimination of need to code in any other language. KSQL users can use the KSQL CLI and run their streaming queries by expressing them in only KSQL statements. The other main differentiator of KSQL compared to other open source streaming platforms is the query execution model. Unlike other systems that need deployment of a separate processing cluster to handle streaming queries, KSQL queries can run as applications independently without requiring deployment of additional complex cluster. This significantly simplifies deployment and management of streaming queries in KSQL.

We just announced availability of KSQL as an open source streaming SQL engine and released it under Apache license in 2017 Kafka Summit in San Francisco and since then we have witnessed significant interest from the community[8]. We plan to invest heavily in development and expansion of KSQL. Currently we are looking into expanding supported data formats in KSQL along with providing custom functionality through UDF and UDAF in addition to the ones we already have in KSQL. Improving the performance through optimized query planning along with even more simplified deployment and maintenance of KSQL service in the production environment are part of our near future work on KSQL. We believe KSQL along with Apache Kafka provide a full stack stream processing environment that can satisfy the all the real-time stream processing needs in enterprises and will be working towards this goal.

REFERENCES

[1] 2018. Apache Flink. (2018). http://flink.apache.org

[2] 2018. Apache Kafka. (2018). http://kafka.apache.org

[3] 2018. Apache Mesos. (2018). http://mesos.apache.org

[4] 2018. Apache Phoenix. (2018). http://phoenix.apache.org

[5] 2018. Apache Samza. (2018). http://samza.apache.org

[6] 2018. Apache Spark. (2018). http://spark.apache.org

[7] 2018. Apache Storm. (2018). http://storm.apache.org

[8] 2018. KSQL. (2018). http://github.com/confluentinc/ksql

[9] 2018. RocksDB. (2018). http://rocksdb.org

[10] 2018. RocksDB. (2018). http://kubernetes.io

[11] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. 2003. Aurora: a new model and architecture for data stream management. *VLDB Journal* 12 (2003), 120–139.

[12] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *VLDB Journal* 15 (2006), 121–142.

[13] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*.

[14] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Frederick Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*.

[15] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka a Distributed Messaging System for Log Processing. In *Proc. NetDB 11*.

[16] M.s Sax, G. Wang, M. Weidlich, and J. Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *Proc. BRITE 18*.

[17] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th International Conference on Data Engineering*.

The Power of SQL Lambda Functions

Maximilian E. Schüle
schuele@in.tum.de

Dimitri Vorona
vorona@in.tum.de

Linnea Passing
passing@in.tum.de

Harald Lang
harald.lang@in.tum.de

Alfons Kemper
kemper@in.tum.de

Stephan Günnemann
guennemann@in.tum.de

Thomas Neumann
neumann@in.tum.de

Technical University of Munich

ABSTRACT

This work demonstrates a wide range of applications that use lambda expressions in SQL. Such injected code snippets form a useful technique required by data mining algorithms to overcome the inflexibility of the SQL language, as the language is limited to predefined aggregations only. Following the ’move computation to the data’ paradigm, we extend SQL lambda functions—also known from common programming languages—for machine-learning tasks.

As machine-learning relies mostly on gradient descent and tensor data types, we use lambda expressions for clustering and graph-mining algorithms as well as to formulate loss functions and label data. To underline the flexibility gained in SQL, this work demonstrates a main memory database system with integrated lambda expressions accessible through table functions in SQL. By reusing SQL and performing data mining and machine-learning tasks faster than can dedicated tools, this demonstration aims at convincing data scientists of the capabilities of database systems for computational tasks.

1 INTRODUCTION

Database systems are commonly used for the initial analysis of data, whereas data analysis happens in specialised systems, covering the expensive Export, Transform, Load (ETL) process. In consequence, database systems are often underused as a storage system only, ignoring the advantages coming along with SQL as the declarative query language and the benefits of database systems as providing index structures for fast data retrieval.

Actually, data mining blends well with SQL: Using the same data types, we need only minor adjustments as iterations in SQL or injected code snippets. MADlib [3] proposes a data mining library for extending database systems such as PostgreSQL and Greenplum. The extension uses algorithms that are programmed in Python and can be used as table functions in SQL. Modern main memory database systems try to integrate data mining algorithms in the code generation phase. EmptyHeaded [1] generates and combines code out of special algorithms and relational algebra, whereas HyPer [5] provides specialised operators for data mining tasks. The latter provides flexibility for data mining by so-called SQL lambda functions, which allow for modifications to existing database operators by injecting user-defined code.

Although modern database systems provide data mining algorithms, machine-learning functionalities are still not covered. As machine-learning tasks often rely on tensors and gradient descent, providing an additional tensor data type and flexible loss functions for gradient descent contribute to in-database machine-learning.

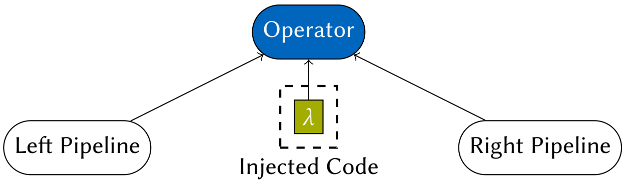


Figure 1: Lambda expression for code injection into an existing operator: The expression can be used inside unary operators to define functions on different tuples of the same relation and inside binary operators to combine tuples of different relations.

Let us shift the boundary between database systems and the specialised tools to save ETL costs and to enjoy the benefits of database systems a bit more throughout the process. We argue that lambda functions can be adapted for machine-learning tasks and extend SQL to a powerful query language for machine-learning. This results in a computational engine, which merits the full benefits of database systems and can be addressed in SQL.

This work demonstrates the first database system to incorporate lambda expressions for minimisation problems. An extended HyPer is presented in detail. The main memory database system developed at the chair of database systems at TU Munich is already familiar with flexible clustering algorithms. It is now extended by flexible PageRank, gradient descent and labelling algorithms for machine-learning. A gradient descent operator and the integrated tensor data type allow supervised machine-learning tasks. As lambda expressions are part of the database system’s core, the query optimiser implicitly performs optimisations, such as predicate push-down, and reduces unnecessary overhead.

The remainder of this paper is structured as follows. First, the paper recaps lambda expressions and shows how they extend database systems to a uniform tool for solving machine-learning tasks. In the evaluation section, we discuss the measurement of the performance of database systems with lambda functions in comparison to dedicated tools to show the competitiveness. Our demonstration scenario allows users to interact with the database system and to create self-defined lambda expressions, for example, to apply gradient descent, to label data or to measure distances in clustering algorithms. The impact of computational

Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.49

database systems—replacing dedicated machine-learning tools—is summarised in the section on benefits.

2 LAMBDA FUNCTIONS

Lambda functions originate from the lambda calculus invented in 1936 by Alonzo Church [2] and later adapted for programming languages to provide anonymous functions. For use in SQL, the concept of anonymous expressions is adapted to 'inject user-defined code' [4] into analytic operators. Lambda expressions allow a user-friendly way for data scientists—who are already familiar with SQL and lambda functions from other programming languages—to customise algorithms without any modifications of the operators in the database system's core. Another advantage of lambda functions is the implicit deduction of input and output data types from the input tables' attributes without requiring further specification.

Lambda expressions in HyPer have been used in clustering algorithms as distance metrics and in graph-mining algorithms, such as PageRank to specify the edges. As lambda functions allow 'variation points' [6] in inflexible data mining algorithms, their usage can be transferred to machine-learning algorithms to specify loss functions. Lambda expressions are composed as follows:

$$\lambda(< name1 >, < name2 >, \ldots)(< arithmeticexpression >).$$

In the lambda function's header, the arguments name the relations whose table attributes are used in the arithmetic expression itself. The number of expected arguments is hard-coded in the operators as well as in the relations referenced by the arguments. More precisely, in unary operators, all arguments reference one input pipeline, whereas, in binary operators, the references depend on a certain application. Internally, lambda expressions are treated as arithmetic expressions in HyPer like those in the projection operator. Therefore, all known functions supported by the referenced database types can be used inside the expression as long as the expression results in a single value. As HyPer compiles SQL queries, lambda expressions are precompiled to LLVM code and evaluated at runtime.

To demonstrate the use of lambda functions for data mining and machine-learning, we selected two tasks out of each domain where lambda expressions broaden the application area of database operators.

Clustering k-Means as a clustering algorithm assigns points to a predefined number of k centres, which are iteratively adjusted until the summed distance to each centre is minimised. Hereby, the distance function is given as a lambda expression to specify Manhattan (L^1) or Euclidean (L^2) distance.

Graph-mining PageRank is a graph-mining algorithm for gathering the importance of nodes by the number of incoming edges per node weighted by the score of the source node.

Optimisation Machine-learning algorithms rely on optimisation algorithms such as gradient descent. Given a model function parametrised by some weights, one aims to minimise the weights to obtain minimal loss in order to predict the targets/labels of the data and a loss function that measures the deviation from the true labels and their predictions. To allow user-defined loss functions, we adapt the concept of lambda expressions to work as a mathematical function for minimisation on the training set's attributes.

Labelling To label test data, an operator adds the result of a parametrised function to the input relation. Lambda expressions define the loss function to be evaluated on the test data set.

The algorithms for clustering and graph-mining are hard-coded as materialising pipeline breakers in the database system's core, as various iterations are needed to compute the clusters or the PageRank values. The operator for labelling—as part of the pipeline—simply evaluates the lambda expression and adds a new attribute to the input relation. The operator for gradient descent uses a self-developed framework for automatic differentiation based on *placeholders* for the input data and *variables* for the weights and can be designed as either materialising or non-materialising pipeline breaker. All operators define the usage of the lambda functions according to the number of input pipelines. As k-Means and PageRank only need one input pipeline, lambda expressions define the functions between the tuples of one input relation. The lambda expression for the distance metrics in k-Means takes two tuples S, T of the same relation $R\{[x, y]\}$ as argument (L^2 distance, for example):

$$\lambda(S, T)((S.x - T.x)^2 + (S.y - T.y)^2).$$

PageRank—contrary to the other operators—needs **two** lambda expressions to specify the edges: one for the source node and the second one for the destination node of the input relation $R\{[src, dst]\}$:

$$\lambda(R)(R.src), \lambda(R)(R.dst).$$

For gradient descent and labelling, we need data for the placeholders and weights for the variables. Therefore, the lambda expression takes one tuple from each of the two input relations $R\{[a, b]\}$ (initial weights) and $S\{[x, y]\}$ (training data) to formulate the loss function:

$$\lambda(R, S)(R.a * S.x + R.b - S.y)^2.$$

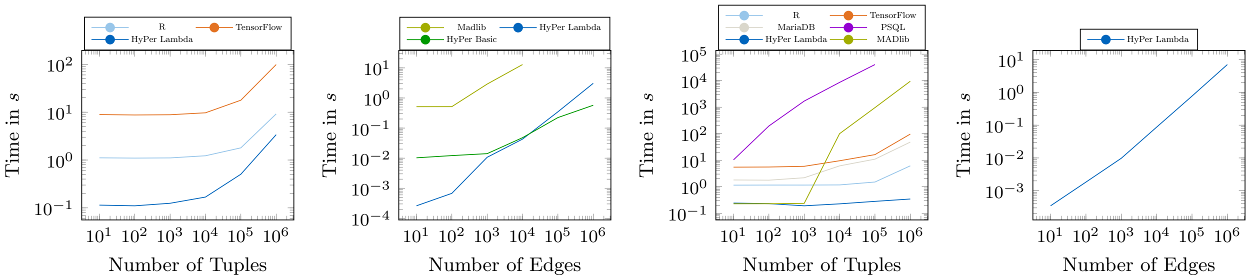
As lambda expressions are part of the database system, they are taken into account by the database's query optimiser. When the data is stored column-wise, only the relevant data for the lambda expression is loaded.

3 EVALUATION

The evaluation section presents the runtimes of the lambda-based operators. The experiments were run on an Intel Xeon E5-2660 v2 (20 cores of 2.20 GHz) server with 256 GB main DDR4 RAM running on Ubuntu 18.04 LTS. The test data was one-month excerpt of the Chicago taxi rides dataset[1] ($> 1.7 * 10^6$ tuples), on which we performed clustering with k-Means and logistic regression by gradient descent, and the LDBC data set with scale factor 10 for graph-mining with PageRank.

Using k-Means, we clustered the taxi dataset geographically by the trip's destination expecting 10 clusters. By logistic regression, we predicted the payment type depending on the distance of a trip and the ages of the customers. We reused the predicted weights to label the data. We take the *person-knows-person*-relationship from the LDBC benchmark to calculate the PageRank values of each person. Each test varied the input sizes to compare the runtime of our extended operators with the lambda functions in HyPer to the runtime of PostgreSQL 9.6.8 with its MADlib v1.13 extension

[1] https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew



(a) k-Means (2 dims, 10 clusters). **(b) PageRank (100 iter., no damping).** **(c) Gradient Descent.** **(d) Labeling.**

Figure 2: Runtimes of the operators using lambda functions and of the competitors varying the number of input tuples.

(database system with a plugin), TensorFlow 1.3.0 without GPU support (dedicated machine-learning tool) and R 3.4.2 (statistical tool). The tests were run five times and the average runtimes were taken.

As expected, all the operators scaled linearly in the size of the input dataset and no overhead caused by lambda expressions in SQL was measurable. Figure 2a shows k-Means in comparison to R and TensorFlow (both use predefined library functions). Our database operator with the included lambda function outperformed all dedicated tools by at least a factor of three.

The in-database PageRank function provided by the MADlib library scaled linearly as well but was still four times slower than the integrated operator (see Figure 2b). The lambda expressions did not slow down the runtime of the database system as both operators performed at quite the same time.

The currently introduced gradient descent operator (see Figure 2c) reduced the overhead caused by data extraction and transfer as the data could be processed directly. The results were performance gains of at least five times in comparison to R, the fastest dedicated tool. The hard-coded gradient descent function in PostgreSQL and the equivalent procedure in MariaDB ran out of scope and the MADlib extension for logistic regression performed as fast as our lambda-based operator only for small input relations. Figure 2d shows the labelling of the attributes that performed linearly in the size of the input tuples.

In summary, the lambda expressions broadened the application area of the database systems without performance losses and eliminated the need for dedicated tools.

4 DEMONSTRATION

The demonstration convinces by the simplicity of using SQL for machine-learning: the central part is an extended web interface for HyPer with SQL for the input queries and a tabular output representation (see Figure 3). To facilitate the use of database systems for data scientists, data visualisation—comparable to those of business intelligence tools—enriches the output by simple dragging and dropping the table's attributes on the axes of different sorts of diagrams (pie/bar/line charts). As an additional feature, a box for specifying lambda functions with their input weights allows the plotting of lambda expressions as mathematical functions inside the diagrams.

The database system is fed with an excerpt of the Chicago taxi dataset and the current OpenFlights dataset[2]. The web interface lets users choose between predefined example queries with sample lambda expressions or lets them create their own queries. Our predefined examples cover supervised machine-learning tasks based on gradient descent (see Listing 3), labelling and data mining tasks such as clustering (see Listing 1) and PageRank (see Listing 2).

In our demonstration scenario, the user is encouraged to perform machine-learning tasks, such as the predefined regression to predict, for example, the number of tips given or the kind of payment in dependency of the length of a trip. In addition, the users can specify any kind of loss functions to perform predictions or other methods for the type of gradient descent.

To show the performance of the database systems, two charts—one for the runtime and one for the operator tree—provide informations about optimisations. This feature helps the user to understand how operator reordering and predicate push-down of integrated lambda expressions increase the performance.

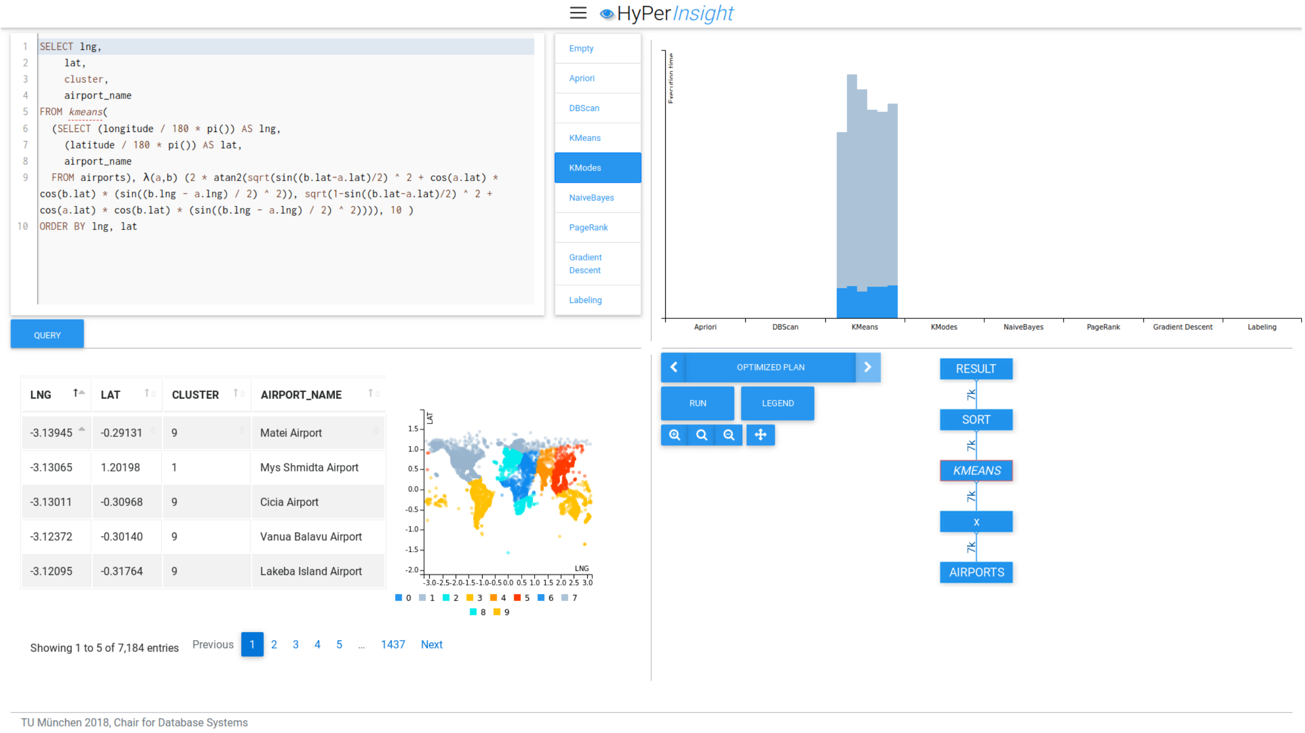
5 CONCLUSION

In this demonstration, we presented SQL lambda functions for in-database machine-learning and data mining with an interactive web interface devoted to its use by data scientists.

We proposed extending already known lambda expressions for use in machine-learning, especially to specify loss functions for gradient descent. By this extension, we used database systems with SQL as a universal computational engine, eliminated the need for dedicated machine-learning tools and reduced the time needed for data communication. To tackle the acceptance of SQL with lambda functions, we created a web interface that interactively combines SQL with data visualisation and provides informations about the optimised query plans.

This work aimed at adding lambda functions in standardised SQL to allow changing the database system as an underlying machine-learning tool. Lambda functions are essential for providing a higher-order machine-learning language to be used by data scientists. For that purpose, a declarative language is needed to further increase the acceptance of database systems and should be designed to be compiled to SQL or an executable to call the application interfaces of current dedicated tools.

[2] https://openflights.org/data.html



```
SELECT lng,
    lat,
    cluster,
    airport_name
FROM kmeans(
  (SELECT (longitude / 180 * pi()) AS lng,
    (latitude / 180 * pi()) AS lat,
    airport_name
  FROM airports), λ(a,b) (2 * atan2(sqrt(sin((b.lat-a.lat)/2) ^ 2 + cos(a.lat) *
cos(b.lat) * (sin((b.lng - a.lng) / 2) ^ 2)), sqrt(1-sin((b.lat-a.lat)/2) ^ 2 +
cos(a.lat) * cos(b.lat) * (sin((b.lng - a.lng) / 2) ^ 2)))), 10 )
ORDER BY lng, lat
```

LNG	LAT	CLUSTER	AIRPORT_NAME
-3.13945	-0.29131	9	Matei Airport
-3.13065	1.20198	1	Mys Shmidta Airport
-3.13011	-0.30968	9	Cicia Airport
-3.12372	-0.30140	9	Vanua Balavu Airport
-3.12095	-0.31764	9	Lakeba Island Airport

Showing 1 to 5 of 7,184 entries Previous 1 2 3 4 5 ... 1437 Next

Figure 3: We adapted our HyPerInsight web interface for demonstrating lambda functions. On the left side, we see the SQL interface with an exemplary lambda expression as a distance metric (top) with tabular output and visualisation (bottom). The right side shows the runtimes of the different algorithms (top) and the operator optimisations (bottom).

```
CREATE TABLE data(x FLOAT, y INTEGER);
CREATE TABLE center(x FLOAT, y INTEGER);
INSERT INTO ...

SELECT * FROM kmeans(
  (SELECT x,y FROM data),
  (SELECT x,y FROM center),
  -- the distance function
  λ(a, b) (a.x-b.x)^2+(a.y-b.y)^2,
  3 -- max. number of iterations
);
```

Listing 1: k-Means.

```
CREATE TABLE edges (a BIGINT, b BIGINT);
INSERT INTO ...

SELECT * FROM pagerank(
   (SELECT * FROM edges),
   λ(src) src.a, -- source
   λ(dst) dst.b, -- destination
   0.85,      -- damping factor
   0.00001,   -- threshold
   100        -- iterations
);
```

Listing 2: PageRank.

```
CREATE TABLE data (x FLOAT, y FLOAT);
CREATE TABLE weights(a FLOAT, b FLOAT);
INSERT INTO ...

SELECT * FROM gradientdescent(
  -- the loss function
  λ(d, w) (w.a*d.x+w.b-d.y)^2,
  -- training data set
  (SELECT x,y FROM data d),
  -- initial weights
  (SELECT a,b FROM weights w),
  0.05, -- learning rate
  100 -- max. number of iteration
);
```

Listing 3: Gradient descent.

Figure 4: Examples of using lambda functions in SQL.

ACKNOWLEDGEMENTS

This work is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Centre Digitisation.Bavaria, an initiative of the Bavarian State Government. Linnea Passing and Dimitri Vorona have been sponsored in part by the German Federal Ministry of Education and Research (BMBF), grant TUM: 01IS12057. This research was supported by the German Research Foundation (DFG), grant NE 1677/1-1 and has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286).

REFERENCES

[1] C. R. Aberger, A. Lamb, K. Olukotun, and C. Ré. Mind the gap: Bridging multi-domain query workloads with emptyheaded. *PVLDB*, 10(12):1849–1852, 2017.

[2] A. Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[3] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The madlib analytics library or MAD skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[4] N. Hubig, L. Passing, M. E. Schüle, D. Vorona, A. Kemper, and T. Neumann. Hyperinsight: Data exploration deep inside hyper. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, pages 2467–2470, 2017.

[5] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206, 2011.

[6] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann. SQL- and operator-centric data analytics in relational main-memory databases. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.*, pages 84–95, 2017.

MINARET: A Recommendation Framework for Scientific Reviewers

Mohamed R. Moawad, Mohamed Maher, Ahmed Awad, Sherif Sakr
University of Tartu, Estonia
{mohamed.ragab,mohamed.abdelrahman,ahmed.awad,sherif.sakr}@ut.ee

ABSTRACT

We are witnessing a continuous growth in the size of scientific communities and the number of scientific publications. This phenomenon requires a continuous effort for ensuring the quality of publications and a healthy scientific evaluation process. Peer reviewing is the de facto mechanism to assess the quality of scientific work. For journal editors, managing an efficient and effective manuscript peer review process is not a straightforward task. In particular, a main component in the journal editors' role is, for each submitted manuscript, to ensure selecting adequate reviewers who need to be: 1) *Matching* on their research interests with the topic of the submission, 2) *Fair* in their evaluation of the submission, i.e., no conflict of interest with the authors, 3) *Qualified* in terms of various aspects including scientific impact, previous review/authorship experience for the journal, quality of the reviews, etc. Thus, manually selecting and assessing the adequate reviewers is becoming tedious and time consuming task.

We demonstrate MINARET, a recommendation framework for selecting scientific reviewers. The framework facilitates the job of journal editors for conducting an efficient and effective scientific review process. The framework exploits the valuable information available on the modern scholarly Websites (e.g., Google Scholar, ACM DL, DBLP, Publons) for identifying candidate reviewers relevant to the topic of the manuscript, filtering them (e.g. excluding those with potential conflict of interest), and ranking them based on several metrics configured by the editor (user). The framework extracts the required information for the recommendation process from the online resources *on-the-fly* which ensures the output recommendations to be dynamic and based on up-to-date information.

1 INTRODUCTION

The world is witnessing a continuous growth in the size of scientific communities and the number of scientific publications. With the current rates, it is expected that the global scientific output doubles every nine years[1]. For example, Figure 1 shows the statistics of the popular DBLP indexing services for computer science publications[2]. In particular, the DBLP library is currently indexing over 3.8M publications. Out of these publications, the number of journal articles published in 2018 is about 120K articles. In 2017, Elsevier journals received 3919 submissions, out of which 530 were accepted with acceptance rate of 14%[3]. This situation raises a crucial need for continuous efforts to ensure and improve the quality of the scientific publication process.

In general, peer reviewing is a widely accepted practice to assess the quality of scientific publications [5, 9]. In this process, the *selection* of appropriate reviewers for evaluating the submitted manuscript is a significant step. For example, selecting reviewers without adequate knowledge in the topic of the submissions or selecting inexperienced reviewers would lead to poor reviews that harm the quality of the publishing venue, the scientific community, and the authors of the manuscript [3]. In addition, it is crucial to avoid any reviewers with potential conflict of interest [10]. In practice, assigning reviewers for a conference submission is less challenging than assigning a reviewer for a journal submission. In particular, the universe of reviewers is *closed* where it is limited to the program committee (PC) members who are commonly selected based on their experience and reputation in the scientific field of the conference. In addition, accepting the membership for the PC of a scientific conference indicates explicit commitment for the assigned review workload of the conference within the defined review deadline. Moreover, conference management systems provide a bidding process where each PC member should select the submissions that he/she would like to review. Thus, with this setup and conditions, it is possible to automate the paper-reviewer assignment task [2, 3, 8].

The manuscript review process for a journal submission is different. In particular, the universe of reviewers is *open* with various aspects of *uncertainty*. Thus, it depends much on the editor's experience, effort and professional network to select the adequate reviewers for a submitted manuscript. For example, there is no pre-defined agreement or arrangement between the journal and a set of committed reviewers. In particular, the manuscript review is a totally voluntary work with the only incentive of having a mutual benefit when the volunteering reviewer is an author of another manuscript submission that needs to be reviewed in the same or other journals. In addition, reviews' deadlines are soft constraints that are not obligatory for the reviewer. Thus, in order to achieve efficient and effective review process, it is the role of the editor to choose the reviewers that should at least cover the following main criteria: 1) Have matching research interests with the topic of the submitted manuscript, 2) Fair in their evaluation of the manuscript with no potential conflict of interest, 3) Have a good rank according to the editor's preferences in various criteria [4]. In practice, the first point can be managed by the editors' experience with the scholars of the scientific community of the journal and by browsing the profiles for candidate

[1] http://blogs.nature.com/news/2014/05/global-scientific-output-doubles-every-nine-years.html

[2] source: https://dblp.uni-trier.de/statistics/newrecordsperyear

[3] source:https://journalinsights.elsevier.com/journals/0142-9612/acceptance_rate

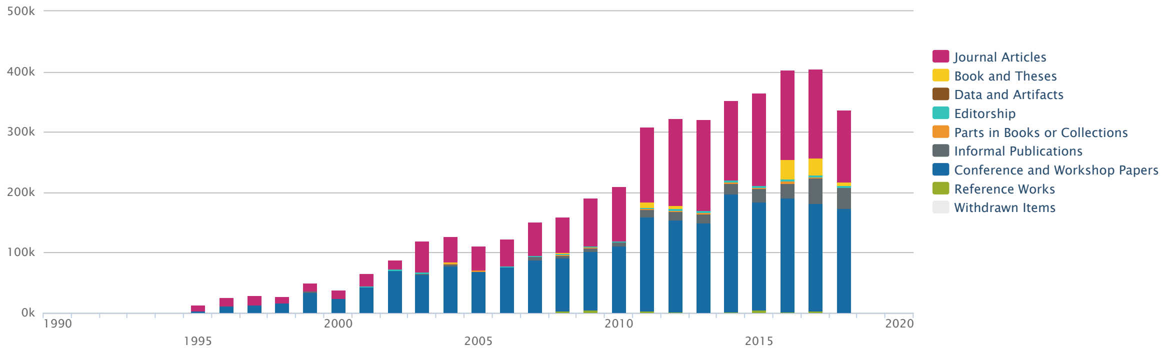


Figure 1: Statistics of DBLP library content

reviewers. However, the second point would require investigating the track record for both the authors and reviewers for discovering any potential for conflict of interest (e.g., co-authorship, having current/previous similar affiliations, ..., etc). Manually exploring and investigating this information would be a tedious and time-consuming task for the editors. The third point involves various aspects that need to be considered. For example, inviting a high-profile reviewer who happens to be quite busy will do nothing but delaying the review process as she might not reply to the invitation in a timely manner, simply reject it or accepting the invite and sending the review very late. Such selections may increase the turnaround time for making the decision on the submitted manuscript. Another aspect that can be considered is the history of review activities of the candidate reviewers and the quality of their reviews. Thus, it is crucially required that the reviewer selection process strikes a balance between these criteria and aspects.

Recently, we have been witnessing a continuous increase in the number of Websites and services that provide comprehensive scholary information. For example, DBLP provides the list of publications for a given author, Google scholar provides information about important metrics for the scientific impact (e.g., citation count, H-index, i10-index), Publons[4] provides information about the reviewing activities that have been conducted by a scholar. In this demonstration, we present MINARET, a recommendation framework for choosing scientific reviewers of a given manuscript information. The framework facilitates the job of journal editors for conducting an efficient and effective scientific review process by dynamically exploiting and integrating the available information on scholarly Websites *on-the-fly*. Within the search process, keywords representing the submission are semantically expanded to provide a wider range of related reviewers as candidates. Extracted information about the candidate reviewers are used to automatically exclude those with potential conflict of interests with the authors of the manuscript. Finally, the list of reviewers is ranked based on various criteria including the experience of the reviewer, recency of his familiarity with the topic of the submission, likelihood to accept and timely return his review, h-index, etc. The weight of these criteria is flexible to be configured by the users of the framework (editors).

2 REVIEWER RECOMMENDATION

Figure 2 shows the workflow of our recommendation framework. Using the basic information of the submitted manuscript (e.g., keywords, authors list and their current affiliation), the recommendation workflow goes through three main phases: *information extraction*, *filtration*, and *ranking* of the candidate reviewer list.

2.1 Information Extraction

The information extraction phase consists of the following main steps:

- *Verification of authors' identities*: This step is concerned with the disambiguation of authors' names [1, 6, 7]. For example, in the far east, many scholars may share one of the popular names[5]. The identification of the correct author profile is crucial as it influences the accuracy of the collected information. We use various services (e.g., DBLP, Google Scholar, ACM) to gather the information about the author list. In case of multiple matches, the user has to manually identify the correct profiles for the author list among the returned matches.
- *Extracting the track records of the author list*: This step focuses on extracting information about the publications list and affiliation history of the author list using multiple services (e.g., DBLP, Google Scholar, ACM, Publons). Extracting the authors' track record is particulary important to allow discovering any potential for conflict of interest.
- *Retrieval of candidate reviewers' profiles*: The main driver for candidate reviewers search is the list of keywords supplied as part of the manuscript details. Usually, this list contains three to five keywords defined by the authors to describe the research topic of their submission. To widen the search space of candidate reviwers, we employ a semantic keyword expansion. For this purpose, as our demonstration is focusing on the computer science community, we rely on an ontology of computer science topics[6]. Each relevant expanded keyword is assigned a similarity score $sc \in \left[0, 1\right]$ that defines the relevance between the returned keyword from the ontology and the original keyword. For example, if one of the manuscript's keywords is "RDF", the expansion module

[4] https://publons.com/

[5] https://dblp.uni-trier.de/pers/hd/z/Zhou:Lei

[6] https://cso.kmi.open.ac.uk/downloads

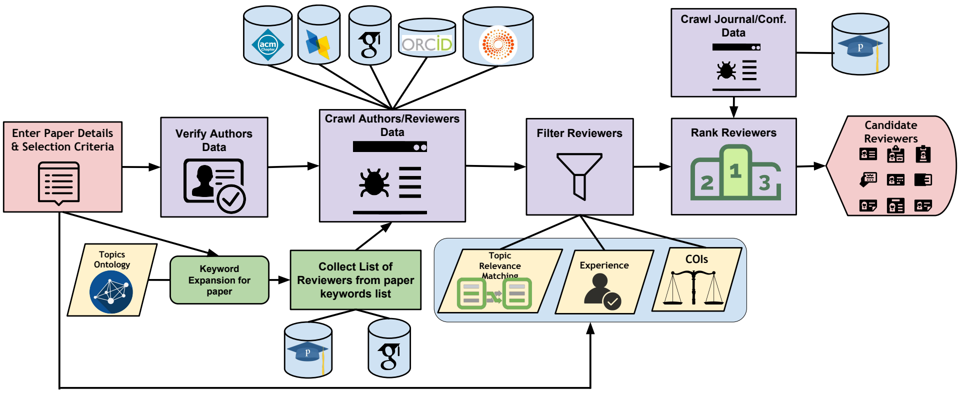


Figure 2: `MINARET` workflow

would return "Semantic Web", "Linked Open Data", and "SPARQL" as semantically related keywords among its results. Using the expanded keywords list, we retrieve the scholars who are registering these keywords as research interests by querying multiple services (e.g., Google Scholar and Publons).

`MINARET` is currently implemented to extract the information from six main sources: `Google Scholar`, `DBLP`, `Publons`, `ACM DL`, `ORCID` and `ResearcherID`. However, the framework is flexibly designed to include any further information from any additional scholary resource.

2.2 Filtering

In this phase, the list of candidate reviewers which is returned from the expanded keyword-based reviewer search gets filtered using the following conditions:

- *Conflict of interest*: COI is determined by checking the extracted profile information for both of the author list and candidate reviewers and based on the existence of a previous co-authorship between the candidate reviewer and one of author list or the existence of any shared affiliations on the level of the university or country, as configured by the editor.
- *Keyword matching score*: The editor can specify a threshold on the similarity score between the expanded keywords and those attached to the reviewers' profiles. Candidate reviewers with matching score below the defined threshold are filtered out.
- *A set of expertise constraints defined by the editor*: The user/editor can specify filtering out some of the candidate reviewers based on various user-defined filtering criteria (e.g., the range of number of citations / H-index, the number of previous review activities)

2.3 Ranking

The last phase of our workflow is ranking the candidate reviewers. `MINARET` ranks the list of reviewers by means of a score which is computed as a weighted sum that fuses the following components:

- *Topic coverage*: Represents the reviewer's coverage score for the keywords of the submitted manuscripts. For example, if the paper keywords were "Semantic Web" and "Big Data" and we have two recommended reviewers with fields of interest as "Semantic Web, Ontologies, RDF"and "Semantic Web, Big Data", respectively. `MINARET` gives the second reviewer a higher rank than the first, because the second reviewer covers more topics/keywords of the paper and therefore is more related to the paper.
- *Scientific impact*: This component is based on the number of citations/H-index of the reviewer, as configured by the user. Clearly, the higher the number of citations/H-index, the higher the rank.
- *Recency*: Reviewers who have recently authored papers in the topic of the reviewed manuscript are ranked higher than others with less recent publications in the topic [5].
- *Experience with manuscript reviewing*: This component is based on the total number of manuscript reviews that is previously conducted by the candidate reviewer. This information is obtained from the `Publons` profile of the candidate reviewer.
- *Familiarity/Activity with the target outlet*: The reviewer's familiarity score is calculated based on two sub-components. The first is the number of previous reviewers that are conducted by the candidate reviewer for the target outlet. The second subcomponent is how many times this reviewer has published papers in this journal.

`MINARET` allows the user to configure the weights of the different components for computing the final ranking score for the candidate reviewers.

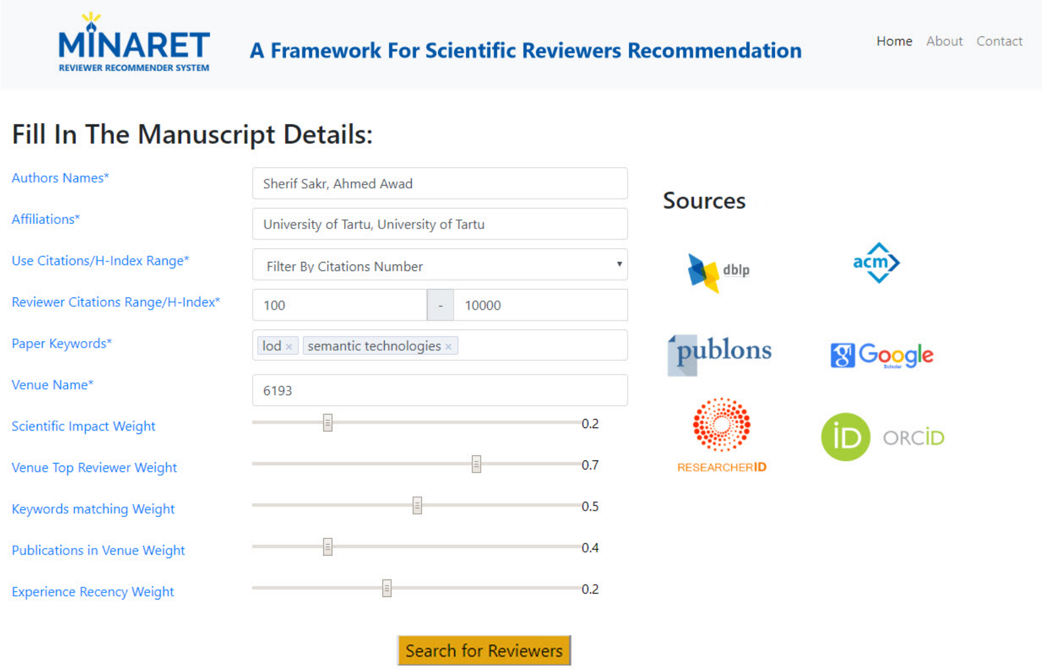
3 DEMO SCENARIO

`MINARET` is available both as a Web application[7] as well as RESTful APIs[8]. In this demonstration[9], we will present to the audience the workflow and the phases of the `MINARET` framewrok (Figure 2). We start by introducing to the audience the challenges we tackle, the main goal and the functionalities of our framework. Then, we take the audience through the reviewers recommendation process for sample manuscripts. We start by completing the manuscript details form (Figure 3) with the basic information including authors' names,

[7] https://bigdata.cs.ut.ee/minaret/

[8] The source code of the `MINARET` framework is available on https://github.com/DataSystemsGroupUT/Minaret

[9] A demonstration screencast is available on https://www.youtube.com/watch?time_continue=7&v=rNHTqdY6GuI



MINARET
REVIEWER RECOMMENDER SYSTEM

A Framework For Scientific Reviewers Recommendation

Home About Contact

Fill In The Manuscript Details:

Authors Names*: Sherif Sakr, Ahmed Awad

Affiliations*: University of Tartu, University of Tartu

Use Citations/H-Index Range*: Filter By Citations Number

Reviewer Citations Range/H-Index*: 100 - 10000

Paper Keywords*: lod × semantic technologies ×

Venue Name*: 6193

Scientific Impact Weight: 0.2

Venue Top Reviewer Weight: 0.7

Keywords matching Weight: 0.5

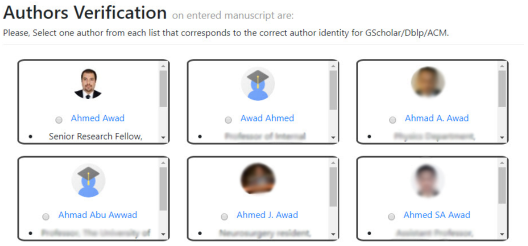
Publications in Venue Weight: 0.4

Experience Recency Weight: 0.2

Search for Reviewers

Sources

Figure 3: Screenshot of adding paper details page

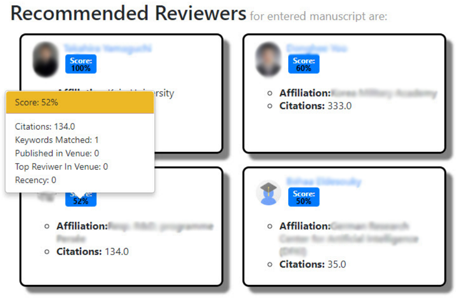


Authors Verification on entered manuscript are:

Please, Select one author from each list that corresponds to the correct author identity for GScholar/Dblp/ACM.

- ○ Ahmed Awad
  - Senior Research Fellow,
- ○ Awad Ahmed
- ○ Ahmad A. Awad
- ○ Ahmad Abu Awwad
- ○ Ahmed J. Awad
- ○ Ahmed SA Awad

Figure 4: Verification of authors identities



Recommended Reviewers for entered manuscript are:

Score: 100%

Score: 60%
- Affiliation:
- Citations: 333.0

Score: 52%

Citations: 134.0
Keywords Matched: 1
Published in Venue: 0
Top Reviwer In Venue: 0
Recency: 0

- Affiliation:
- Citations: 134.0

Score: 50%
- Affiliation:
- Citations: 35.0

Figure 5: Example recommended reviewers

authors' current affiliations, submission topics/keywords, target journal in addition to any user-defined filters for the target reviewers (e.g, citation range, H-index range). Next, we will show how MINARETchecks and verifies authors names (Figure 4). After this, we will show how MINARET queries the different scholary sites for extracting the required information (Section 2) for candidate reviewers. Next, we will continue with the stage of reviewers' filtration and ranking till returning the final results (Figure 5) where the computed score of each reviewer is shown. By clicking on the total score, score details for each ranking component will be displayed.

While MINARET is designed for tackling the more challenging case of recommending reviewer for journal submissions. It can be also integrated with conference management systems to automate the paper-reviewer assignment. In that case, the list of programme committee members can be used as a further filter. Thus, only candidate reviewers who belong to the programme committee are retained.

ACKNOWLEDGMENT

This work is funded by the European Regional Development Funds via the Mobilitas Plus programme (grant MOBTT75)

REFERENCES

[1] Jinseok Kim. 2018. Evaluating author name disambiguation for digital libraries: a case of DBLP. *Scientometrics* 116, 3 (2018).

[2] Ngai Meng Kou et al. 2015. A Topic-based Reviewer Assignment System. *PVLDB* 8, 12 (2015).

[3] C. Long, R. C. Wong, Y. Peng, and L. Ye. 2013. On Good and Fair Paper-Reviewer Assignment. In *ICDM*.

[4] Jennifer Nguyen et al. 2018. A decision support tool using Order Weighted Averaging for conference review assignment. *Pattern Recognition Letters* 105 (2018).

[5] Hongwei Peng et al. 2017. Time-Aware and Topic-Based Reviewer Assignment. In *DASFAA*.

[6] Jie Tang. 2016. AMiner: Mining Deep Knowledge from Big Scholar Data. In *WWW '16 Companion*.

[7] Jie Tang et al. 2008. ArnetMiner: Extraction and Mining of Academic Social Networks. In *KDD*.

[8] Kai-Hsiang Yang et al. 2009. A Reviewer Recommendation System Based on Collaborative Intelligence. In *WI-IAT*.

[9] Shu Zhao et al. 2018. A novel classification method for paper-reviewer recommendation. *Scientometrics* (2018).

[10] Indrė Žliobaitė and Mikael ForteliusUniversity. 2016. Revise rules on conflicts of interest. *Nature* 539 (2016), 10.

SCube: A Tool for Segregation Discovery

Alessandro Baroni
University of Pisa, Italy
baroni@di.unipi.it

Salvatore Ruggieri
University of Pisa, Italy
ruggieri@di.unipi.it

ABSTRACT

Segregation is the separation of social groups in the physical or in the online world. Segregation discovery consists of finding contexts of segregation. In the modern digital society, discovering segregation is challenging, due to the large amount and the variety of social data. We present a tool in support of segregation discovery from relational and graph data. The SCube system builds on attributed graph clustering and frequent itemset mining. It offers to the analyst a multi-dimensional segregation data cube for exploratory data analysis. The demonstration first guides the audience through the relevant social science concepts. Then, it focuses on scenarios around case studies of gender occupational segregation. Two real and large datasets about the boards of directors of Italian and Estonian companies will be explored in search of segregation contexts. The architecture of the SCube system and its computational efficiency challenges and solutions are discussed.

1 SOCIAL SEGREGATION

Ethical issues in data and knowledge management are gaining momentum in the last few years. In addition to the traditional field of privacy, techniques for data analysis are being designed or enhanced to take into account moral values such as fairness, transparency, accountability, and diversity[1]. We have recentely developed a novel data-driven technique for addressing segregation of social groups through multi-dimensional data analysis [4]. The approach is implemented in the SCube system, which we propose to demonstrate using real case studies.

Social segregation refers to the "separation of socially defined groups" [11]. People are partitioned into two or more groups on the grounds of personal or cultural traits that can foster discrimination, such as gender, age, ethnicity, income, skin color, language, religion, political opinion, membership to a national minority, etc. Contact, communication, or interaction among groups are limited by their physical, working or socio-economic distance. This can be observed when dissecting society in organizational units (neighborhoods, schools, job types). Due to the ubiquitous presence and pervasiveness of ICT, segregation is shifting from ancient forms of well explored spatial segregation[2] to novel forms of digital segregation. For instance, it has been warned that the filter bubble generated by personalization of online social networks may foster idelogical segregation [6], opinion polarization [10], and informational segregation. A data-driven technology that enables the assessment of the extent, nature, and trends of social segregation in the offline or online world, is of extreme interest for a wide audience: social scientistics, public policy makers, regulation and control authorities, professional associations, civil rights societies, and investigative journalists. Business decision makers should also care of business practices, particularly automated decision making, that segregate customers and products through stereotypes, because this limits diversity and reduces opportunities of cross-selling. Finally, data scientists and professionals should be aware of the unintended consequences of their models (recommender systems, link suggestion systems, classifiers) on the cohesion of society at large.

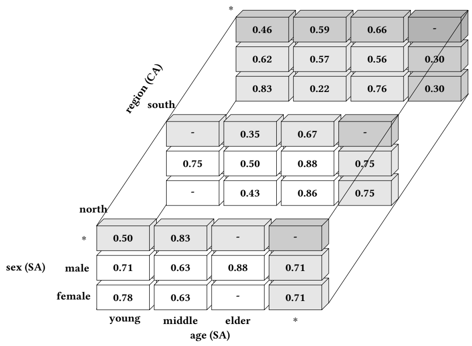


Figure 1: A segregation data cube with dissimilarity index.

2 SEGREGATION DISCOVERY

From a data analysis perspective, the key problem of assessing social segregation has been investigated so far by hypothesis testing, i.e., by formulating one or more possible contexts of segregation against a certain social group, and then in empirically testing such hypotheses. Such an approach is currently supported by statistical tools, such as the R packages *OasisR*[3] and *seg*[4] [9], or by GIS tools such as the *Geo-Segregation Analyzer*[5] [2]. The formulation of an hypothesis, however, is not straightforward, and it is potentially biased by the expectations of the data analyst of finding segregation in a certain context. In addition, exploration of multiple hypothesis can be time consuming, since data have to be processed multiple times. Finally, this approach is subject to erroneous conclusions if data is considered at wrong granularity – an instance of the Simpson's paradox.

Multi-dimensional segregation data cube. Our approach consists of providing the analysts with a multi-dimensional data cube that can be explored in search of candidate contexts of segregation. An example segregation data cube is shown in Fig.1. Dimensions of the data cube include two types of attributes:

- *segregation attributes* (SA), such as `sex`, `age`, and `ethnicity`, which denote (minority/protected) groups potentially exposed to segregation;
- *context attributes* (CA), such as `region` and `job type`, which denote contexts where segregation may appear.

Metrics of the data cube are chosen among the social science indexes proposed for measuring the degree of segregation of social groups within a society [12]. Here, we recall only one such index, but the SCube system is parametric to the indexes

[1] See e.g., the *Toronto declaration* at www.accessnow.org/toronto-declaration.

[2] See census stats, e.g., www.census.gov/topics/housing/housing-patterns/data.html

[3] cran.r-project.org/package=OasisR

[4] cran.r-project.org/package=seg

[5] geoseganalyzer.ucs.inrs.ca

and it computes 6 of them: dissimilarity, Gini, Information index, Isolation, Interaction, Atkinson. Also, we restrict to binary groups (minority/majority). Let T be the size of the total population under consideration, $0 < M < T$ be the size of a minority group, $T - M$ the size of the rest of society (or majority group) and $P = M/T$ be the overall fraction of the minority group. Assume that there are n organizational units (or simply, units – such as schools, neighboorhoods, job types, etc.), and that for $i \in [1, n]$, t_i is the size of the population in unit i, and m_i is the size of the minority group in unit i. The *dissimilarity index* D measures the absolute distance between the fractions of minority and majority groups over the units:

$$D = \frac{1}{2} \sum_{i=1}^{n} \left| \frac{m_i}{M} - \frac{t_i - m_i}{T - M} \right|$$

D ranges over $[0, 1]$, with higher values denoting higher segregation. Dissimilarity is minimum when for all $i \in [1, n]$, $m_i/t_i = M/T$, namely the distribution of the minority group is uniform over units. It is maximum when for all $i \in [1, n]$, either $m_i = t_i$ or $m_i = 0$, namely every unit includes members of only one group (complete segregation). Dissimilarity and other segregation indexes can be interpreted as metrics in a cell of a multidimensional cube as follows: set the total population as those individuals that satisfy the CA coordinates of the cell; and, set the minority population as those individuals that satisfy the SA coordinates. For instance, the cube cell in Fig.1 with SA coordinates `sex=female, age=young` and CA coordinates `region=north` contains the dissimilarity index for the population living in the north region and for the minority group of young women. Notice that the number n of organizational units here have to be determined *a-priori*, while the total population and minority groups in each unit depend on the values of cell coordinates. As in standard multi-dimensional modelling [7], the special value "$\star$" allows for considering different granularities of analysis.

Segregation analysis of tabular data. We assume in input a relational table with a tuple for every individual in the population, including SA and CA attributes, and with a further attribute `unitID` which denotes the unit an individual belongs to. Unfortunately, segregation indexes are not additive metrics (see [4]). This gives rise to the problem of efficiently computing a data cube for segregation analysis. Our approach is more specialized than generic holistic aggregate computation in datacubes [13]. We resort to frequent closed itemset mining [8]. Data cube coordinates are encoded into itemsets of the form **A**, **B**, where **A** denotes a minority subgroup and **B** denotes a context. Recalling the previous example, **A** =`sex=female, age=young` defines the SA coordinates, and **B** =`region=north` defines the CA coordinates. The *SegregationDataCubeBuilder* algorithm described in [4] fills data cube cells with the value of a segregation index by scanning frequent closed itemsets of the form above. Since relational data is transformed into transaction database for itemset mining, we obtain for free that CA or SA attributes can be multi-valued, e.g., to denote that an individual owns both a house and a car we admit a relation tuple σ such that $\sigma[\texttt{owns}] = \{\texttt{house}, \texttt{car}\}$.

Segregation analysis of graph data. While transaction databases are able to cover typical analysis from traditional social science, they are not enough powerful to deal with social network data. We formalize such a case using attributed graphs, where nodes are assigned values on a specified set of attributes. However, in this scenario, there is no *a-priori* defined notion of organizational unit, i.e., the `unitID` attribute assumed in input so far. Some forms of community discovery using graph clustering become necessary in order to determine the organizational units. Clustering attributed graphs consists of partitioning them into disjoint communities of nodes that are both well connected and similar with respect to their attributes [5]. In summary, attributed graph clustering can be used first to partition a social network into communities. At this stage, every node/individual in a community is described by its attributes and the community id, which will be our `unitID` attribute. We have thus reduced the problem to the analysis of relational data, for which the *SegregationDataCubeBuilder* algorithm can be applied.

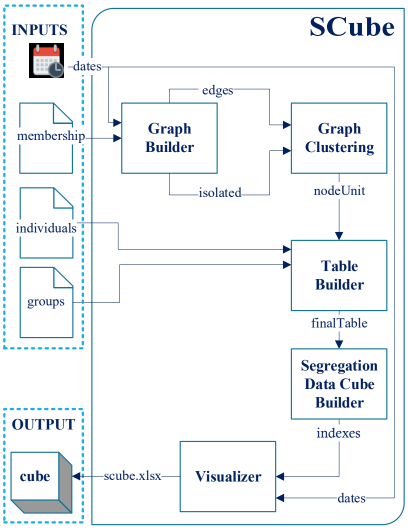


Figure 2: SCube architecture.

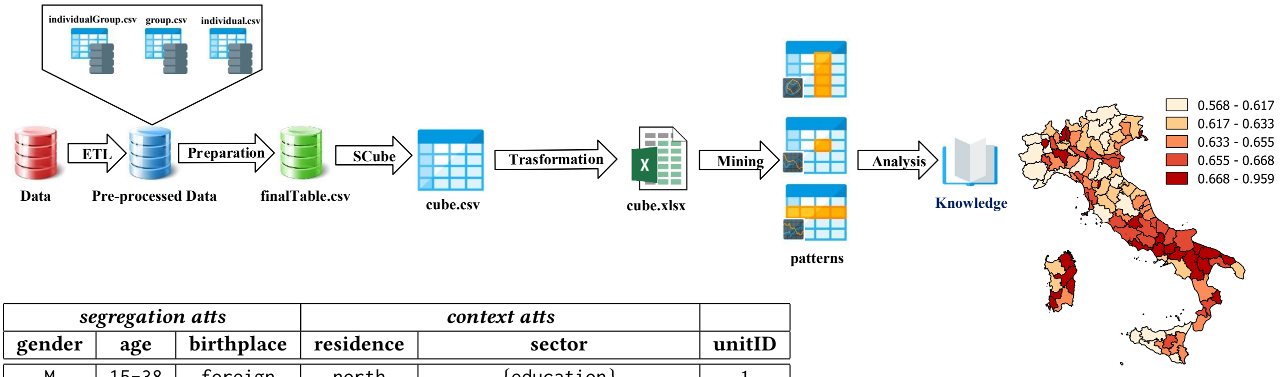
Segregation analysis of bipartite graphs. An even more complex scenario is when individuals are not connected among them, e.g., because they are friends, but through a connection with another entity, e.g., because they work in the same company. Here, a form of projection on unipartite graph is needed to reduce to the previous case. For instance, in [4], we adopt a bipartite projection of the bipartite graph of directors and companies to obtain a graph of companies connected by shared directors. Using projection, we have reduced the problem to the previous case, where attributed graph clustering can be adopted to find communities of companies, which then represent the organizational units for segregation analysis.

3 SCUBE ARCHITECTURE

The architecture of SCube is shown in Fig. 2. The system is developed in Java, and it relies on a few state-of-the-art libraries[6].

Inputs. The user has to provide features for two entities: *individuals* and *groups*. In the reference case studies, individuals are directors and groups are companies. The input `individuals` (a CSV file or a JDBC query) provides for each individual an ID and a number of attribute values, distinguished into segregation attributes (e.g., `gender`, `age`, `birthplace`) and context attributes (e.g., `residence`). A second input `groups` provides for each group an ID and a number of context attributes values (e.g., industrial `sector` of a company and its headquarter `location`). Notice that individuals are subjects to possible segregation, while groups are

[6] EWAH for compressed bitmaps (github.com/lemire/javaewah), Apache POI for OOXML docs (poi.apache.org), Borgelt's FPGrowth for frequent itemset mining (www.borgelt.net), FastUtil for graph storage (fastutil.di.unimi.it).



segregation atts			*context atts*		
gender	**age**	**birthplace**	**residence**	**sector**	**unitID**
M	15-38	foreign	north	{education}	1
F	39-46	south	south	{electricity, transports}	2
M	55-65	north	south	{agriculture}	1
...	...	...	...	...	...

Figure 3: The process of segregation discovery supported by SCube (left, top), input to SegregationDataCubeBuilder (left, bottom), and an output report on dissimilarity segregation index of the Italian provinces (right).

not. For this reasons, groups have no SA feature. A third input is `membership`, which includes the edges of the bipartite graph of individuals and groups, i.e., all pairs (`individualID`, `groupID`) for which the individual is related to the group. In our case studies, directors are related to companies they sit in the board of. We also admit that the pairs are labeled with a time interval of validity, thus allowing for temporal analysis of segregation. We have such an information for the Estonian dataset. A fourth input is a list of snapshot dates at which to consider snapshots of the membership relation.

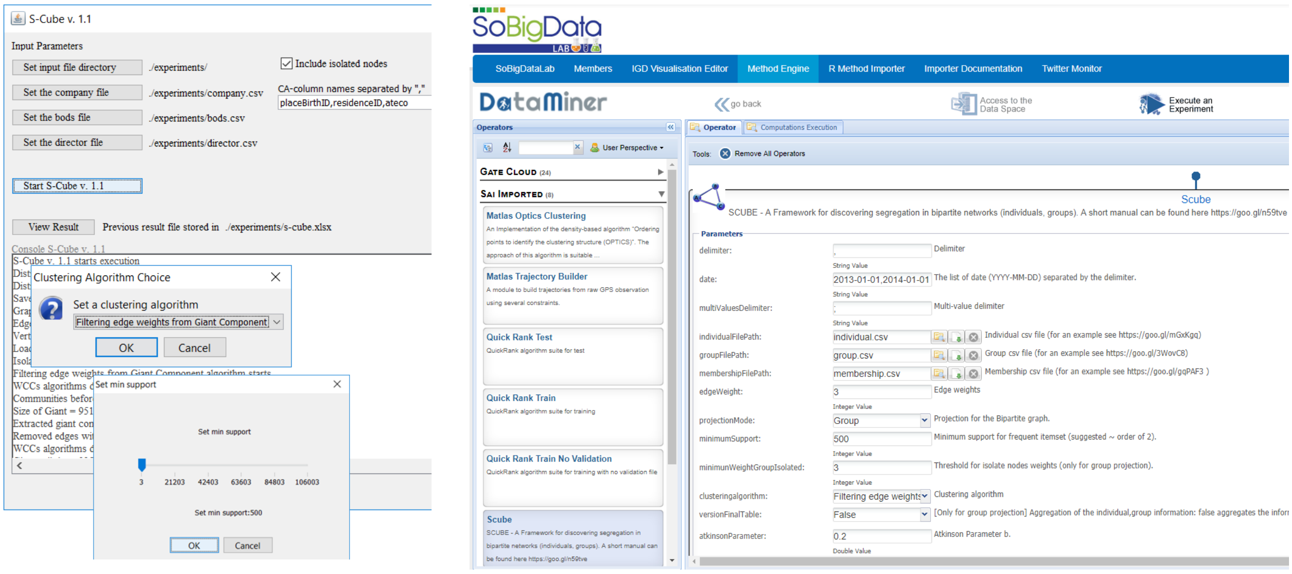
Modules. SCube consists of five software modules. *GraphBuilder* projects the bipartite graph of individuals and groups into an unipartite attributed graph, where nodes are groups and an edge connect two groups if they are related by at least one shared individual. In the case studies, nodes are companies, and edges connect companies that share at least one director in their boards. Edges are weighted by the number of shared directors. *GraphBuilder* outputs edges of the projection (`edges`), and nodes that have zero degree (`isolated`). The *GraphClustering* module computes then a clustering of nodes into organizational units (output file `nodeUnit`). Methods for clustering available in SCube include: extraction of connected components (Breadth-First Search), removal of edges from the giant component with weight below a threshold and then extraction of connected components (designed in [4]), and an attributed graph clustering method for very large graphs (SToC algorithm [3]). In our case studies, the result of *GraphClustering* is a partitioning of companies into clusters based on connections among companies determined by shared directors – which can be readily considered a signal of relationships (business, personal, or other) between companies. Clusters represent the organizational units needed for computing segregation indexes. *TableBuilder* joins features of individuals with features of the companies in an organizational unit. This yields a `finalTable` with a row per individual and organizational unit she belongs to. An example is shown in Fig. 3 (left, bottom). This is the input for the *SegregationDataCube* builder module, implementing the algorithm of [4]. Notice that if the data under analysis contains already the assignment of individuals to units, i.e., it is already in the form of `finalTable`, the pre-processing steps of bipartite projection and graph clustering do not need to be performed. The *Visualizer* module transforms the extended datacube in output of SegregationDataCube into a standard OOXML format that can be opened by Microsoft Excel, Libre Office, and other office productivity tools (see Fig. 5). Segregation data cube exploration can be easily interfaced with visualization tools, as in the map overlay in Fig. 3 (right).

Process, Wizard, and GUI. The whole process of segregation discovery supported by SCube is shown in Fig. 3 (left, top). To facilitate the adoption of SCube by non-technical users, we have developed two interfaces (see Fig. 4). The first one is a standalone wizard that guides the user throughout all the steps of the process, asking for inputs and parameters when appropriate, and finish launching Microsoft Excel or Libre Office on the output file. Using popular desktop tools as GUI's makes the learning curve of approaching and effectively using SCube more manageable. The second one is a cloud service offered by the SoBigDataLab freely accessible research infrastructure (www.sobigdata.eu/access/virtual), a web front-end comprising a catalogue of data, services, and virtual research environments for big data and social mining research.

4 DEMONSTRATION SCENARIO

The demonstration starts with a brief introduction on concepts and methods of segregation measurement [12] and segregation discovery [4]. This provides the audience with the basic definitions for understanding the SCube functionalities. The architecture of SCube is presented next. For interested participants, computational efficiency, algorithmic solutions, and source code internal aspects are discussed. Then, two running case studies in the context of occupational segregation in the boards of company directors [1] are introduced. They are based on a 2012 snaphost of Italian companies (3.6M directors, 2.15M companies), and on a 20-year long dataset of Estonian companies (440K directors, 340K companies). Such anonymized datasets are the largest ever considered in the literature of segregation analysis. We summarize the data pre-processing activities to produce the inputs for SCube.

The demonstration then proceeds by presenting three analysis scenarios based on input data of increasing complexity. In all scenarios, `gender`, `age`, and `birthplace` are used as segregation attributes. The first scenario considers tabular data, where company sector is used as organizational `unitID`, and it is intended



S-Cube v. 1.1

Input Parameters

Set input file directory ./experiments/

Set the company file ./experiments/company.csv

Set the bods file ./experiments/bods.csv

Set the director file ./experiments/director.csv

☑ Include isolated nodes

CA-column names separated by ","
placeBirthID,residenceID,ateco

Start S-Cube v. 1.1

View Result Previous result file stored in ./experiments/s-cube.xlsx

Console S-Cube v. 1.1

```
S-Cube v. 1.1 starts execution
Dist
Dist
Save
Grap
Edg
Vert
Load
Isola
Filtering edge weights from Giant Component algorithm starts
WCCs algorithms d
Communities befor
Size of Giant = 951
Extracted giant con
Removed edges wi
WCCs algorithms d
```

Clustering Algorithm Choice

Set a clustering algorithm
Filtering edge weights from Giant Component

OK Cancel

Set min support

Set min support

Set min support:500

OK Cancel

Gate Cloud (24)

Sai Imported (8)

Matlas Optics Clustering
An Implementation of the density-based algorithm "Ordering points to identify the clustering structure (OPTICS)". The approach of this algorithm is suitable ...

Matlas Trajectory Builder
A module to build trajectories from raw GPS observation using several constraints.

Quick Rank Test
QuickRank algorithm suite for test

Quick Rank Train
QuickRank algorithm suite for training

Quick Rank Train No Validation
QuickRank algorithm suite for training with no validation file

Scube
SCUBE - A Framework for discovering segregation in bipartite networks (individuals, groups). A short manual can be found here https://goo.gl/n59tve

Scube

SCUBE - A Framework for discovering segregation in bipartite networks (individuals, groups). A short manual can be found here https://goo.gl/n59tve

Parameters

delimiter: , Delimiter (String Value)

date: 2013-01-01,2014-01-01 The list of date (YYYY-MM-DD) separated by the delimiter. (String Value)

multiValuesDelimiter: ; Multi-value delimiter (String Value)

individualFilePath: individual.csv Individual csv file (for an example see https://goo.gl/mGxKgq)

groupFilePath: group.csv Group csv file (for an example see https://goo.gl/3WovC8)

membershipFilePath: membership.csv Membership csv file (for an example see https://goo.gl/gqPAF3)

edgeWeight: 3 Edge weights (Integer Value)

projectionMode: Group Projection for the Bipartite graph.

minimumSupport: 500 Minimum support for frequent itemset (suggested ~ order of 2). (Integer Value)

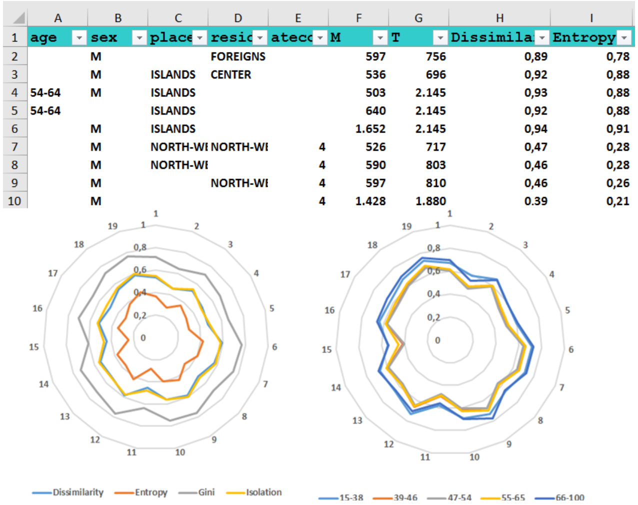
minimunWeightGroupIsolated: 3 Threshold for isolate nodes weights (only for group projection). (Integer Value)

clusteringalgorithm: Filtering edge weights Clustering algorithm

versionFinalTable: False [Only for group projection] Aggregation of the individual,group information: false aggregates the infor

atkinsonParameter: 0.2 Atkinson Parameter b. (Double Value)

Figure 4: SCube standalone wizard (left) and SCube method at the SoBigData research infrastructure (right).



	A	B	C	D	E	F	G	H	I
1	age	sex	place	resic	atecc	M	T	Dissimila	Entropy
2		M		FOREIGNS		597	756	0,89	0,78
3		M	ISLANDS	CENTER		536	696	0,92	0,88
4	54-64	M	ISLANDS			503	2.145	0,93	0,88
5	54-64		ISLANDS			640	2.145	0,92	0,88
6		M	ISLANDS			1.652	2.145	0,94	0,91
7		M	NORTH-WI	NORTH-WI	4	526	717	0,47	0,28
8		M	NORTH-WI		4	590	803	0,46	0,28
9		M		NORTH-WI	4	597	810	0,46	0,26
10		M			4	1.428	1.880	0.39	0,21

Figure 5: Top: sample multidimensional segregation cube. Bottom: radial plot of segregation indexes for directors in each of the 20 Italian company sectors.

to answer questions such as: how much are women segregated in company sectors? The second scenario considers attributed graph data, where nodes are directors, and edges connect two directors if they belong to a same company board. Here, the organizational units are determined through clustering over attributed graphs. This scenario can answer questions such as: how much are women segregated in communities of connected directors? Finally, the third scenario considers a bipartite attributed graph of directors and companies, as presented throughout the paper. An example of question it can answer is: how much are women segregated in communities of connected companies? For each scenario, the output of SCube is interactively explored using pivot tables and charts. The audience is guided to the discovery of a few actual cases of *a-priori* unknown segregation contexts and to the understanding of which attributes contribute the most to segregation. Moreover, a cross-comparison of the Italian vs Estonian segregation findings will be discussed.

5 CONCLUSION

This demonstration illustrates the SCube tool for interactive exploration of social segregation indexes in large and complex data. The audience is made aware of social exclusion issues that can be hidden in data and of the indexes that measure segregation. Real case studies on scenarios of increasing complexity are discussed and explored. Efficiency issues and algorithmic solutions adopted for scaling to large datasets and graphs are detailed.

Acknowledgements. This work is partially supported by the European H2020 Program under the funding scheme "INFRAIA-1-2014-2015: Research Infrastructures" grant agreement 654024 "*SoBigData*" (http://www.sobigdata.eu).

REFERENCES

[1] M. Aluchna and G. Aras, editors. *Women on Corporate Boards*. Routledge, 2018.

[2] P. Apparicio, J. C. Martori, A. L. Pearson, E. Fournier, and D. Apparicio. An open-source software for calculating indices of urban residential segregation. *Social Science Computer Review*, 32(1):117–128, 2014.

[3] A. Baroni, A. Conte, M. Patrignani, and S. Ruggieri. Efficiently clustering very large attributed graphs. In *ASONAM*, pages 369–376. ACM, 2017.

[4] A. Baroni and S. Ruggieri. Segregation discovery in a social network of companies. *J. Intell. Inf. Syst.*, 51(1):71–96, 2018.

[5] C. Bothorel, J. D. Cruz, M. Magnani, and B. Micenková. Clustering attributed graphs: models, measures and methods. *Network Science*, 3(03):408–444, 2015.

[6] S. Flaxman, S. Goel, and J. M. Rao. Filter bubbles, echo chambers, and online news consumption. *Public Opinion Quarterly*, 80:298–320, 2016. Available at SSRN: http://ssrn.com/abstract=2363701.

[7] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[8] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: Current status and future directions. *Data Min. Knowl. Discov.*, 15(1):55–86, 2007.

[9] S.-Y. Hong, D. O'Sullivan, and Y. Sadahiro. Implementing spatial segregation measures in R. *PLoS ONE*, 9(11):e113767, 2014.

[10] M. Maes and L. Bischofberger. Will the personalization of online social networks foster opinion polarization? *Available at SSRN: http://ssrn.com/abstract=2553436*, 2015.

[11] D. S. Massey. Segregation and the perpetuation of disadvantage. *The Oxford Handbook of the Social Science of Poverty*, pages 369–393, 2016.

[12] D. S. Massey and N. A. Denton. The dimensions of residential segregation. *Social Forces*, 67(2):281–315, 1988.

[13] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Data cube materialization and mining over mapreduce. *IEEE Trans. Knowl. Data Eng.*, 24(10):1747–1759, 2012.

SparkTune: tuning Spark SQL through query cost modeling

Enrico Gallinucci
DISI - University of Bologna
Bologna, Italy
enrico.gallinucci@unibo.it

Matteo Golfarelli
DISI - University of Bologna
Bologna, Italy
matteo.golfarelli@unibo.it

ABSTRACT

We demonstrate SparkTune, a tool that supports the evaluation and tuning of Spark SQL workloads from multiple perspectives. Unlike Spark SQL's optimizer, which mainly relies on a rule-based model, SparkTune adopts a cost-based model for SQL queries; this enables the accurate estimation of execution times and the identification of cost and complexity factors in a user-defined workload. The estimate is based on the cluster configuration, the database statistics (both automatically retrieved by the tool) and the resources allocated to the workload. Thus, for any given cluster, database and workload, SparkTune is able to identify the best cluster configuration to run the workload, to estimate the price to run it on a cloud platform while evaluating the performance/price trade-off, and more. SparkTune turns the cluster tuning efforts from manual and qualitative to automatic, optimized and quantitative.

1 INTRODUCTION

Over the past years, Apache Hadoop has become the most popular framework for Big Data handling and analysis. On top of it, *SQL-on-Hadoop* solutions have been introduced to provide a relational abstraction layer to the data. Among them, one of the most popular is Apache Spark, whose SQL-based sub-system (i.e., Spark SQL [1]) enables SQL queries to be rewritten in terms of Spark commands and to be executed in parallel on a cluster[1].

Although these systems are largely adopted and quickly becoming more solid and mature, they are still limited in terms of cost modeling features. For instance, the module in charge of translating SQL queries to Spark commands (i.e., Catalyst [1]) mainly relies on a rule-based optimizer. The cost model introduced to Catalyst in its 2.3.0 release is still very simple (e.g., as concerns join ordering, the cost function estimates a logical cost in terms of number of returned rows). The need for robust cost-based models is increasing, since the possibility to evaluate a priori the execution cost of a workload is still lacking. Very little work has been done on this aspect: some optimizers propose cost-based features, but they evaluate only portions of a query [8].

Our system, *SparkTune*, is an application that builds on a cost model [3] for Spark SQL to support the user understanding the cost of a workload and gaining the required knowledge to properly optimize the execution. From a high-level perspective, SparkTune allows to: i) evaluate the duration of a workload; ii) see the cluster's size and potential first-hand, so as to properly optimize the allocation of resources; iii) evaluate the trade-off between the execution time and the price to execute it on a cloud platform. The latter point is achieved by extending the cost model with the pricing information of major cloud providers, which enables the translation of resource consumption into actual money. On a more technical side, SparkTune also: iv) enables the prior identification of stragglers (i.e., tasks that performs more poorly than similar ones due to insufficient assigned resources); v) if adopted by Spark SQL, it would enable the creation of a full cost-based optimizer, both static and dynamic.

In Section 2 we summarize the core aspects of the cost model (full details have been published in [3]), while Section 3 discusses the features of SparkTune; the demonstration proposal is finally given in Section 4.

2 COST MODEL OVERVIEW

The Spark architecture consists of a *driver* and a set of *executors*. The driver negotiates resources with the cluster resource manager (e.g. YARN) and distributes the computation across the executors, which are in charge of carrying out the operations on data. Data are organized in *Resilient Distributed Datasets* (RDDs), i.e., collections of immutable and distributed elements *partitions* that can be processed in parallel. Partitions can either come from a storage (e.g. HDFS) or be the result of a previous operation (i.e., held in memory). At the highest level of abstraction, a Spark computation is organized in *jobs*, which are composed of simpler logical units of execution called *stages*. The physical unit of work used to carry out a stage on each RDD partition is called *task*. Tasks are distributed over the cluster and executed in parallel.

In Spark SQL, a declarative SQL query is translated in a set of jobs by its optimizer, Catalyst, which carries out the typical optimization steps: analysis and validation, logical optimization, physical optimization, and code generation. Physical optimization creates one or more *physical plans* and then it selects the best one. At the time of writing, this phase is mainly rule-based: it exploits a simple cost function only for choosing among the available join algorithms [1].

Our cost model [3] computes the query execution time given the physical plan provided by Catalyst. We remark that it is *not* a cost-based optimizer, as the latter implements query plan transformations to optimize the original plan, and it does so without necessarily computing the whole cost of the query.

The cost model covers a wide class of queries that composes three basic SQL operators: selection, join and generalized projection. The combination of these three operators determine GPSJ (Generalized Projection / Selection / Join) queries, which were first studied in [5] and which are the most common class of queries in OLAP applications. Each Spark physical plan modeling a GPSJ query can be represented as a tree whose nodes represent tasks; each task applies operations to one or more input tables, either physical or resulting from the operations carried out in its sub-tree. Our cost model relies on a limited number of *task types* (listed in Table 1) that, properly composed, form a GPSJ query. In particular, a feasible tree properly composes the following task types: *table scan* SC(), *table scan and broadcast* SB(), *shuffle join* SJ(), *broadcast join* BJ() and *group by* GB(). SC() and SB() are always leaf nodes of the execution tree since they deal with the

[1] Spark is not necessarily tied to Hadoop, although this is its most used architecture.

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

physical storage where the relational tables lie. SJ() and BJ() are inner nodes of the trees and can be composed to create left-deep execution trees; finally GB(), if present, is the latest task to be carried out.

The execution time is obtained by summing up the time needed to execute the tasks of the tree coding the physical plan of a query. In particular, the cost model is based on the disk access time and on the network time spent to transmit the data across the cluster nodes; CPU times for data serialization/deserialization and compression are implicitly counted by the disk throughput. This is consistent with [9], which clearly explains that *one-pass* workloads on Spark (e.g., SQL queries) are either network-bound or disk-bound, whereas CPU can become a bottleneck limitedly to serialization and compression costs. Also, the cost model assumes that data always fits the executors memory, so that data is never spilled to local disks.

Depending on its type, each task involves one or more basic operations (such as reading, writing, and shuffling) which we refer to as *basic bricks*; the cost of a task is calculated as the sum of the cost of the involved bricks (see 1 for the usage of bricks by the task types). In particular, each brick models the execution of an operation on a single RDD partition and considers the resource contentions given by parallel execution. Bricks are SQL-agnostic and require some parameters about Spark (e.g., network and disk read/write throughput) and about the cluster (e.g., number of executors per rack and number of cores per executor) to be known. As explained in Section 3, most of these parameters are automatically retrieved by SparkTune. In the following, we briefly discuss the nature of each basic brick.

- Read: consists in reading an RDD partition from disk. If the data does not reside on the executor's node, it must be read from another one, according to the locality principle. The cost model exploits the known cluster configuration to estimate the probability of such a case. When reading from another node, both the time to transfer the data over the network must be considered in addition to the time to read the data from disk. Since Spark enables in-memory pipelining of subsequent transformations that do not require shuffling (according to the Volcano model [4]), the overall time is computed as the maximum for disk reading and data transmission.
- Write: consists in writing an RDD partition to the local disk; no network data transfer is necessary.
- Shuffle read: consists in reading an RDD partition from disk and shuffling it through the network. Similarly to the Read brick, the overall time is computed as the maximum for disk reading and data transmission; in this case, however, disk reading only happens on the local disk.
- Broadcast: consists in loading the whole RDD on the application driver and in sending it to every executor. Since the two operations cannot be pipelined, the overall cost is determined as the sum of the two. The broadcast brick does not involve disk reading or writing, thus the cost only depends on network time.

Ultimately, we remark that – thanks to the known cluster configuration – the cost model is able to probabilistically estimate the amount of data to be read and/or transferred through the network for each brick.

Example 2.1. The following GPSJ query is taken from the TPC-H benchmark [7]; it computes the total income collected in a given period and for a specific market segment grouped by single orders and priority of shipping.

Table 1: Task types characterization

Task Type	Addittional params	Basic bricks
SC()	pred, cols, groups	Read, Write
SJ()	pred, cols, groups	Shuffle Read, Write
SB()	pred, cols	Read, Broadcast
BJ()	pred, cols, groups	Write
GB()	pred, cols, groups	Shuffle Read, Write

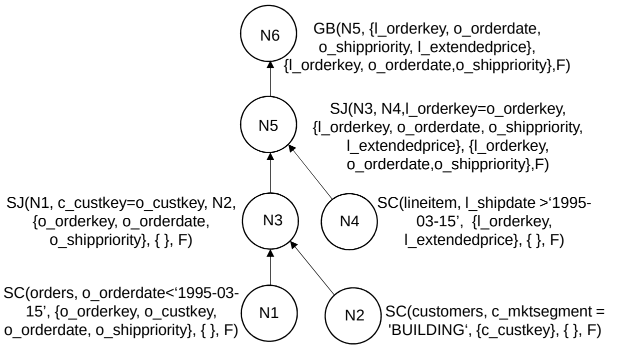

N6
GB(N5, {l_orderkey, o_orderdate, o_shippriority, l_extendedprice}, {l_orderkey, o_orderdate,o_shippriority},F)
N5
SJ(N3, N4,l_orderkey=o_orderkey, {l_orderkey, o_orderdate, o_shippriority, l_extendedprice}, {l_orderkey, o_orderdate,o_shippriority},F)
SJ(N1, c_custkey=o_custkey, N2, {o_orderkey, o_orderdate, o_shippriority}, { }, F)
N3
N4
SC(lineitem, l_shipdate >'1995-03-15', {l_orderkey, l_extendedprice}, { }, F)
SC(orders, o_orderdate<'1995-03-15', {o_orderkey, o_custkey, o_orderdate, o_shippriority}, { }, F)
N1
N2
SC(customers, c_mktsegment = 'BUILDING', {c_custkey}, { }, F)

Figure 1: GPSJ grammar derivation for the query in Example 2.1; node names substitute sub-expressions in the inner nodes

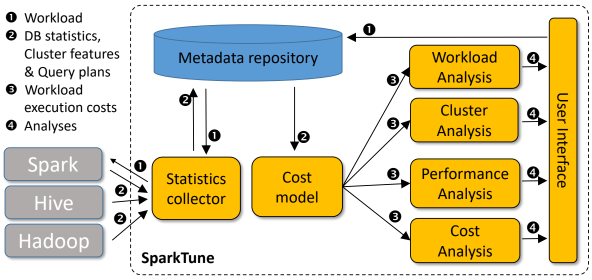


Figure 2: SparkTune's architecture and data flow

```
SELECT l_orderkey, o_orderdate, o_shippriority,sum(l_extprice)
FROM customer, orders, lineitem
WHERE c_mktsegment = 'BUILDING' AND
c_custkey = o_custkey AND l_orderkey = o_orderkey AND
o_orderdate < date '1995-03-15' AND
l_shipdate > date '1995-03-15'
GROUP BY l_orderkey, o_orderdate, o_shippriority
```

A graphical representation of the Spark physical plan chosen by Catalyst is reported in Figure 1.

3 THE SYSTEM

SparkTune is implemented as a web application on a classic WAMP stack (Windows, Apache, MySQL, PHP) and it is publicly available at http://semantic.csr.unibo.it/sparktune. We plan to release SparkTune as a standalone application in the near future. The architecture of the system is sketched in Figure 2, where the flow of data is also indicated.

3.1 Environment setup

The system enables registered user to setup their own environment in order to define custom scenarios and run user-specific analyses. The setup of the environment can be done either manually or in an automatic fashion. The information required by the system is the following:

- Cluster configuration, such as the number of nodes and racks, the number of cores per node, HDFS's replication factor, hardware statistics. Automatic retrieval of these data is enabled by providing credentials for an SSH connection to one of the cluster's nodes. Most parameters are obtained through calls to the Hadoop's APIs, while Spark jobs are run to infer the read and write throughput of the disk, as well as the intra-rack and inter-rack network throughput. Both throughput values are inferred as a function of the number of concurrent processes that require disk access and network data transfer.
- Database statistics. Automatic retrieval of these data is enabled by providing credentials to access the Hive Metastore. For any of the available database, the user can trigger the retrieval of the name, size, and statistics for every table and attribute.
- One or more workloads, meant as sequences SQL queries to be manually provided by the user; then, the system automatically retrieves from Spark the physical plan.

These data are stored in the Metadata repository. Each user can setup multiple clusters, databases, and workloads. The versions of Spark currently supported are 1.5 and 2.2.

3.2 Analyses

The system provides four kinds of analysis and simulation. Each of them requires to define a specific environment (i.e., to select a cluster, a database, and workload among the ones provided in the setup), possibly requests additional parameters (e.g., the number of executors to be allocated for each node), and outputs a detailed report depending on the kind of analysis. Figure 3 shows examples of such reports, which are fully discussed in the following.

Workload analysis. The goal is to provide an in-depth analysis of the complexity and cost of executing the workload on a cluster with a specific configuration (i.e., the number of executors and the number of cores per executor must be defined by the user). First of all, SparkTune runs each query's physical plan in the workload on the cost model to estimate the total execution time of the workload. Secondly, it provides full details about every task of every query (including the amount of data read/written, the time to read/write data from disk or to transfer it across the network). On the one hand, this greatly helps the identification of stragglers: Figure 3.1 shows the report of a single query, which presents the tasks in the execution tree colored on a red gradient (the harder the red, the higher the percentage of time required by the task w.r.t. to the query). On the other hand, it helps understanding which factors mainly impact on the cost of each task/query: below the execution plan in Figure 3.1, the total estimated time of the query is split among disk read time, disk write time, and network time; then, by clicking on a task in the tree, a pop-up as the one in Figure 3.2 shows the same data for each task — together with other detailed information (e.g., the amount of data read/written/transferred, the cardinality of the involved table(s), etc.).

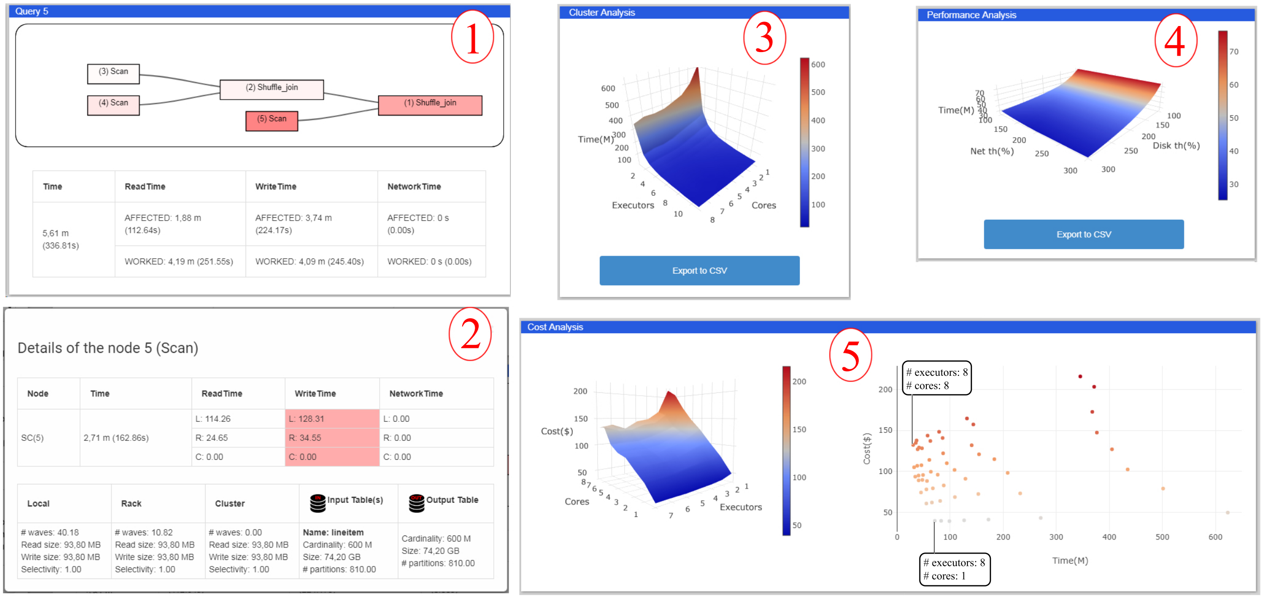
Cluster analysis. The goal is to optimize the allocation of resources by understanding the impact of the configuration parameters on the performance. The total execution time of the workload is calculated for every configuration potentially available within the boundaries set in the cluster setup[2]. The results are presented in a 3D surface graph (Figure 3.3), showing the execution time by the number of executors and the number of cores. This graph emphasizes how the increase of executors and cores progressively reduces the execution time; more specifically, it shows which of the two factors has the greatest impact in the time reduction for the given workload.

Performance analysis. The goal is to provide a what-if analysis that shows the potential impact of enhancing the performance of the cluster (in terms of disk throughput and network throughput) on the execution time of a workload. Given a specific cluster and its configuration, the system estimates the execution time by progressively improving the disk and/or network throughput in steps of 20% (i.e., 120% w.r.t. to the current throughput, 140%, etc.). The results are presented in a 3D surface graph (Figure 3.4), which helps understanding which factor (disk or network) is most critical in determining the execution time. Noticeably, the increase in disk throughput is assumed in the same measure for both reading and writing.

Cost analysis. The goal is to evaluate the price of executing a workload on a cloud infrastructure. To make such an estimate, we extended our cost model with the pricing information obtained from cloud providers Amazon AWS (http://aws.amazon.com) and Google Cloud (http://cloud.google.com); this enables the translation of the time usage estimates of every core and every executor to the amount of USD (United States Dollars) to be paid to the provider for the execution of the workload. Similarly to the Cluster analysis, the system exhaustively calculates the total execution time of the workload in every possible configuration (within the upper bounds set in the cluster setup), but it is eventually translated to USD. Ultimately, two results are presented. The first one (left-hand side of Figure 3.5) is a mere cost analysis, showing the price obtained with the different cluster configurations by means of a 3D surface graph; this enables the identification of the cheapest cluster configuration, as well as the understanding of which factor (i.e., number of executors or number of costs) is the most expensive. The second one (right-hand side of Figure 3.5) is an evaluation of the trade-off between execution time and price; it is shown as a 2D chart, pinpointing each cluster configuration in the 2D surface represented by the execution time and the price. This chart allows an immediate identification of the ideal cluster configurations, which minimize both the time spent and the money to be paid. For the sake of completeness, we note that the price estimate only considers the consumption of the cluster's resources for the estimated time; it does not consider the price of the disk, which is required to store the database.

Example 3.1. The reports in Figure 3 pertain to a workload of 8 queries from the TPC-H benchmark on our 11-node cluster. Workload analysis. Figure 3.1 represents the execution tree of the query in Example 2.1 and shows that the overall time (5.61 min) is mostly due to two tasks (nodes 1 and 5). Given Spark's in-memory pipelining, the "affected time" voice indicates the portion of the time spent reading/writing/transferring data (i.e.,

[2] We adopt this naive approach for the purpose of showing a full report to the user. A cost-based optimizer that only looks for an optimal configuration could apply some heuristics to avoid an extensive research [2].



Time	Read Time	Write Time	Network Time
5,61 m (336.81s)	AFFECTED: 1,88 m (112.64s)	AFFECTED: 3,74 m (224.17s)	AFFECTED: 0 s (0.00s)
	WORKED: 4,19 m (251.55s)	WORKED: 4,09 m (245.40s)	WORKED: 0 s (0.00s)

Details of the node 5 (Scan)

Node	Time	Read Time	Write Time	Network Time
SC(5)	2,71 m (162.86s)	L: 114.26	L: 128.31	L: 0.00
		R: 24.65	R: 34.55	R: 0.00
		C: 0.00	C: 0.00	C: 0.00

Local	Rack	Cluster	Input Table(s)	Output Table
# waves: 40.18 Read size: 93,80 MB Write size: 93,80 MB Selectivity: 1.00	# waves: 10.82 Read size: 93,80 MB Write size: 93,80 MB Selectivity: 1.00	# waves: 0.00 Read size: 93,80 MB Write size: 93,80 MB Selectivity: 1.00	**Name: lineitem** Cardinality: 600 M Size: 74,20 GB # partitions: 810.00	Cardinality: 600 M Size: 74,20 GB # partitions: 810.00

Figure 3: Screenshots of the various functionalities of SparkTune, discussed in Section 3

"worked time") that actually contributed to determining the overall query time. Figure 3.2 shows the details of node 5 in Figure 3.1; it is the scan of the Lineitem table and it shows that the task is mostly disk-bound. Cluster analysis. Figure 3.3 clearly shows that, for the given workload, the performance gain due to increasing the number of executors is superior to the one due to increasing the number of cores per executor. Performance analysis. Figure 3.4 further confirms that the workload is disk-bound; indeed, improving the network throughput would have little to no impact on the execution time, as opposed to the adoption of more performing disks. Cost analysis. The prices shown in Figure 3.5 refer to to Amazon AWS; interestingly, the left chart shows that the cheapest configurations are those with a low number of cores. This indicates that the higher core-power would not be adequately put to use, as it does not correspond to a significant reduction in time (as the Cluster analysis also anticipated). Ultimately, this is confirmed by the right chart, which shows a high disproportion between the price required to reduce the execution time of the cheapest configured and the time actually saved.

4 DEMO PROPOSAL

In the demonstration we will show how cluster tuning can be automatized and optimized adopting a cost-based approach. First, we will demonstrate the automatic retrieval of data for the environment setup, either on our own cluster or on a cluster owned by someone in the audience (provided their willingness to grant its access). Then, we will develop with the audience an experience to showcase the various functionalities of the system and to understand their value from the perspective of different professional figures. In particular, two distinct scenarios will be simulated for data scientists (which will be interested in analyzing workloads from a more technical angle) and data architects (which will be interested in an analyzing the same workloads from a different, higher-level perspective). With data scientists we will mainly focus on SparkTune's Workload and Cluster analyses to answers several questions, such as: *"What are the reasons for the poor performance of this workload?"*, or *"How can I reallocate the resources to speed up the execution of the workload without exceeding my budget?"*. Differently, with data architects we will mainly focus on SparkTune's Cost and Performance analyses to address issues related to the design and deployment of the infrastructure; in particular, we will answer questions such as *"Which cloud provider will let me run this workload at the lowest price and at a reasonable performance?"*, or *"What kind of scale-up or scale-out improvements can I bring to the cluster to improve its performance?"*. The demo will put the audience in the shoes of one of these figures and we will develop a simulation aimed at answering the aforementioned questions.

Our real cluster that we use as a reference consists of 11 nodes with 8 cores per node. SparkTune will be configured with different benchmark databases, including the well-known TPC-H [7] and the Big Data Benchmark [6]. We will define different workloads on each database, in order to present scenarios that require the exploitation of the different features of SparkTune.

REFERENCES

[1] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *SIGMOD*. 1383–1394.
[2] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. 2012. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems* 28, 5 (2012), 755–768.
[3] M. Golfarelli and L. Baldacci. 2018. A Cost Model for SPARK SQL. *IEEE Transactions on Knowledge & Data Engineering* (2018), 14. https://doi.org/10.1109/TKDE.2018.2850339
[4] Goetz Graefe and William J McKenna. 1993. The Volcano optimizer generator: Extensibility and efficient search. In *Data Engineering*. 209–218.
[5] Ashish Gupta, Venky Harinarayan, and Dallan Quass. 1995. Aggregate-Query Processing in Data Warehousing Environments. In *VLDB*. 358–369.
[6] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A comparison of approaches to large-scale data analysis. In *SIGMOD*. 165–178.
[7] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *ACM Sigmod Record* 29, 4 (2000), 64–71.
[8] John Pullokkaran. [n. d.]. Introducing cost based optimizer to apache hive. https://cwiki.apache.org/confluence/download/attachments/ 27362075/CBO-2.pdf.. Online; accessed 18 dEC. 2017.
[9] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. 2015. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *PVLDB* 8, 13 (2015), 2110–2121.

HOTMapper: Historical Open Data Table Mapper

Henrique Varella Ehrenfried
C3SL Labs, UFPR - Brazil
hvhrernfreid@inf.ufpr.br

Rudolf Copi Eckelberg
C3SL Labs, UFPR - Brazil
rc16@inf.ufpr.br

Hamer Iboshi
C3SL Labs, UFPR - Brazil
hi15@inf.ufpr.br

Eduardo Todt
C3SL Labs, UFPR - Brazil
todt@inf.ufpr.br

Daniel Weingaertner
C3SL Labs, UFPR - Brazil
danielw@inf.ufpr.br

Marcos Didonet Del Fabro
C3SL Labs, UFPR - Brazil
marcos.ddf@inf.ufpr.br

ABSTRACT

We present HOTMapper, a tool that maps tables of Open Data with historical information into unified data sources. The tool couples data exchange and integration techniques implemented into two main components: 1) a CLI script with commands to create and update tables, and to insert and update the data, using 2) a simple mapping definition file, interpreted by the CLI script, to store the schema and data mappings throughout different years. The tool is implemented using Python and MonetDb. This demo will show the creation of the mapping definition and the execution flow of the CLI script for creating a unified data source from scratch and then updating an existing one. It will unify real world data sources, with millions of records, containing information about the Brazilian educational system.

1 INTRODUCTION

The availability of large Open Data sources raises several opportunities for researchers from different domains to extract and process the data. Governments from different countries are releasing vast amounts of governmental data, i.e. regarding public services and expenditures. However, in order to effectively use the data, some difficult problems must be addressed: open data sources are often heterogeneous, with different representation formats, data models, schemas (if any), data quality, and others. Such problems can be categorized into Data Exchange, Data Integration and Table Stitching issues. Data Exchange is the problem of transforming data from one source, which has a source schema, into a target schema [10]. Data Integration is defined as the problem of uniformly accessing different source schemes through an integrated source [10]. Finally, Table Stitching is defined as the problem of unifying many tables with identical schemes into a single table identifying extra attributes for it [7].

Open Data is often de-normalized (e.g. in large CSV (Comma Separated Values) files) and represents a small period of time (i.e. a semester or a year). New data is periodically released, with several files that need to be integrated. In addition, from one release to another, data and schema changes often occur. While each periodic instance could be handled separately, having a unified view of the data allows to do historical analysis and comparisons. Thus, it is important to develop methods and tools that are able to create a unified view of the data, to translate each periodic source into the unified view, to perform transformations on the data sources and to update them periodically. This means it is necessary to provide simple ways to maintain schema and data mappings from several input schemes and data sources into a unified one, keeping data compatibility.

There are different solutions/tools partially covering these issues. The Clio tool [5] is one of the best known, being developed before the advent of Open Data and a precursor to many others. More recently the *Polystore*-like approaches, such as *BigDawg*[3], ESTOCADA [1] or MISO[6] focus on the data integration issues, as does the Data Civilizer system[2]. *Ling at al.* [7] handled the problem of table stitching: new columns in the data source need to be identified and stitched into unified tables. The approach from [8] presents a brief history on data integration solutions and the current issues when having multiple Open Data sources. Their solution for creating union-able tables [9] could be valuable for a mapping framework, though they do not focus on maintenance and updates of the data. As a drawback, the solutions completeness and adaptability to several kinds of queries and data sources makes it hard to be used in specialized scenarios, such as the historical Open Data mapping presented in this paper.

The **HOTMapper tool** [1] (Historical Open Data Table Mapper) was developed as a domain specific data mapping/integration solution, targeted to de-normalized Open data sources, spread into several input files, where the mappings are simple and need to be stored and modified periodically. The implementation couples different aspects of data integration, data exchange and table stitching. It consists of a Command Line Interface (CLI) tool for historical data and schema mapping, translated from CSVs into the *MonetDb*[2] column store.

The tool receives as input the CSV files and a mapping definition file with information on how to unify the data. The mappings are defined in CSV as well, relating all desired input data with the corresponding target columns. The mapping definition file has information to guide the following actions: 1) creation of the target unified schema; 2) source-to-target data transformations, which are maintained for each given period; 3) creation of new derived data 4) update of an existing source with new columns and mappings; and 5) full reconstruction of the unified source based on the input sources and mappings. More actions could be added if needed.

This demonstration will present the execution flow of the tool:

- development of the periodical data and schema mappings;
- creation of a new data source;
- insertion of the corresponding data into the unified model;
- update with a new data source;
- update in the columns and data transformations.

We will execute the tool in a real world scenario, processing information about the Brazilian educational system. The data includes the enrollments and related information (schools, courses or teachers) in schools and Universities, with hundreds of columns. The mappings and commands can be modified on

Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

[1] https://gitlab.c3sl.ufpr.br/tools/hotmapper
[2] https://www.monetdb.org/

10.5441/002/edbt.2019.53

the fly following the audience interactions, showing how fast and simple the tool is.

The unified data produced by the tool is currently being used and can be visualized in an open web portal [4] [3]. The amount of data already processed by the tool is summarized in Table 1. It was extracted from Open Data sources produced by INEP (Instituto Nacional de Pesquisa Educacionais Anísio Teixeira)[4].

Table 1: Summary of data from the INEP Open Data Source processed by HOTMapper

Year	Tables	Records
2017	13	102.176.661
2016	20	116.009.013
2015	18	116.946.948
2014	17	121.115.913
2013	18	112.645.020
2012	11	36.029.271
2011	7	12.025.035
2010	7	8.768.490

The paper is organized as follows: section 2 presents a motivating example of HOTMapper; section 3 explains HOTMapper's design and working principle and Section 4 describe the steps of the tool demonstration.

2 MOTIVATING EXAMPLE

Consider the necessity of extracting a metric (or indicator) regarding the number of enrolled students in all Brazilian schools from 2013 to 2017. The information about enrollments is available at an Open Data source produced by INEP. While this is a particular metric, the idea can be generalized as "extracting indicators from public Open Data sources". The input data is de-normalized in a set of large CSV files, with approx. 100 different fields each. The choice for publishing de-normalized data is common in open data sources, because it decreases the number of files that need to be managed in the long run.

The metric/indicator must be correctly extracted from this source and can be aggregated in different dimensions, i.e, location dimensions such as country, state or city; enrollments on public vs. private schools, enrollments on scientific courses, among others. The metric also needs to support yearly updates, because a new set of files is released every year, raising three main issues:

1. **Integrated source creation**: a first set of tables needs to be created to be able to execute queries considering the historical data. In this case, the input data has 87 columns for the 2013 and 2014 years, and 96 columns in 2015-2017.
2. **Schema evolution**: every year, the data sources definition changes, so it is necessary to provide schema mappings:
   (a) direct mappings:
   ```
   NATIONALITY <- [2013-2017] NATIONALITY
   SPECIAL_NECESSITY <- [2013-2014] HAS_NECESSITY
   SPECIAL_NECESSITY <- [2015-2017] SPECIAL_NECESSITY
   ```
   (b) new mapping (when a column is added):
   ```
   REGION <- [2013-2015] not-available
   REGION <- [2016-2017] REGION
   ```
3. **Data evolution**: data has to be transformed and kept compatible along all years, requiring instance mappings:
   (a) mapping creation: a new dimension is generated from existing data
   ```
   PROFESSIONAL_STUDIES<-[2013-2014]
   WHEN STUDIES_KIND between 30 and 40 THEN 1
   WHEN STUDIES_KIND between 41 and 50 THEN 2
   PROFESSIONAL_STUDIES<-PROFESSIONAL_STUDIES[2015-2017]
   ```
   (b) mapping update: the data mapping from the data sources may change (dimension or metric), which means the previous mappings, for all previous years, has to be updated
   ```
   GENDER <- [2013-2014]
             WHEN M THEN 1
             WHEN F THEN 2
   GENDER <- [2015-2017] GENDER
   ```

This motivating example considered just one indicator and illustrative examples, with intersections of data exchange, integration and table stitching. This kind of mapping is commonly applied to more or less 90 columns of annual data. In addition, INEP publishes at least four main raw data sets: Students, Teachers, Schools and Sessions, containing more than one hundred columns. There are other sources released on a similar basis, replicating this scenario for every new indicator produced, presenting a challenging setting for the extraction and maintenance of Open Data sources.

3 HOTMAPPER

In this section we present the HOTMapper tool, addressing the maintenance of data and mappings of Open Data sets.

3.1 Tool description

HOTMapper's architecture is shown in Figure 1. It consists of a CLI (Command Line Interface) manager interacting with three kinds of input and/or output, on which six predefined actions can be performed. With a flexible implementation using Python and SQL programming languages, the tool allows for an easy extension by adding more actions.

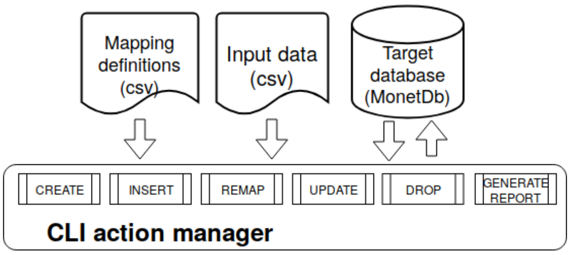


Figure 1: HOTMapper overview

The **mapping definition** files are in CSV format. There is one file for each table in the target RDBMS. It describes the columns that are created and the mappings, per year, of each input column.

The **input data** is the set of CSV files that are included in the target database, one file per metric (or set of metrics), and per year. For instance, the *enrollments* metric has five input files, from 2013 to 2017. The files are de-normalized, which means they have a high number of columns.

The **target database** has the set of tables that contain the integrated information produced by the tool. We use the *MonetDb* column store and keep the included data as similar as possible to the input data, i.e., we do not normalize the target data. This has some advantages: the definition and maintenance of mappings remains relatively simple; the data model does not need to be

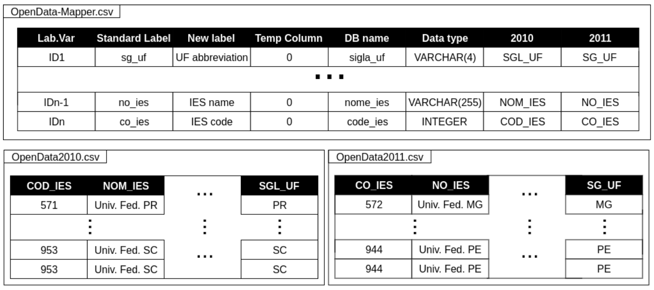
[3] http://dadoseducacionais.c3sl.ufpr.br
[4] http://portal.inep.gov.br/microdados

constantly adapted for every new release; the queries are fast, because joins are minimized; and bulk insertion operations are fast. It is important to emphasize that the target database is only updated by new releases of the input data, without OLTP operations.

The actions currently handled by HOTMapper are: **table maintenance** actions (1) `CREATE` and (2) `DROP`; **data and table manipulation** actions (3) `INSERT`, (4) `REMAP` and (5) `UPDATE`; and a **reporting** action (6) `GENERATE REPORT`:

1. `CREATE` action takes as input the table definition and executes the DML (Data Manipulation Language) commands;
2. `DROP` action deletes the table passed as parameter and any related data. The tool executes standard DML commands, so that it can be used in different RDBMS's;
3. `INSERT` action executes a bulk insert of the input CSV files into a temporary table in the target database. Then, it reads the mapping definition to transfer the data into the target table. The creation of a temporary table is important to facilitate the manipulation of the input data, avoiding direct operations on the CSV file;
4. `REMAP` action modifies the initial table definition;
5. `UPDATE` action updates a table after a change in the mappings; and
6. `GENERATE REPORT` action produces a report with column equivalences between the input table and the current database. It is not necessary for inserting or updating data, but eases the creation of the mapping definition file.

Figure 2 illustrates a mapping definition and two input CSV files, with information about the universities in Brazil.



OpenData-Mapper.csv

Lab.Var	Standard Label	New label	Temp Column	DB name	Data type	2010	2011
ID1	sg_uf	UF abbreviation	0	sigla_uf	VARCHAR(4)	SGL_UF	SG_UF
...							
IDn-1	no_ies	IES name	0	nome_ies	VARCHAR(255)	NOM_IES	NO_IES
IDn	co_ies	IES code	0	code_ies	INTEGER	COD_IES	CO_IES

OpenData2010.csv

COD_IES	NOM_IES	...	SGL_UF
571	Univ. Fed. PR		PR
⋮		⋮	⋮
953	Univ. Fed. SC	...	SC
953	Univ. Fed. SC		SC

OpenData2011.csv

CO_IES	NO_IES	...	SG_UF
572	Univ. Fed. MG		MG
⋮		⋮	⋮
944	Univ. Fed. PE	...	PE
944	Univ. Fed. PE		PE

Figure 2: Illustration of a mapping definition and two input CSV data files

File `OpenData-Mapper.csv` contains the mapping definition, composed of eight columns: `[Lab.Var]` is an identifier for each mapping; `[Standard Label]` has the name of the mapped columns; `[New Label]` has the description of the column; `[Temp Column]` defines whether the column is temporary (used when pre-calculations are necessary); `[DB name]` is the name of the column in the target database; `[Data type]` has the data type of the column defined by `[DB name]`; and columns `[2010]`,`[2011]` have the yearly mappings of each input column into the corresponding target column. Every time a new release is available, a new year column is added, with its corresponding mappings. The periodicity can be different, depending of the characteristics of the input data.

HOTMapper also supports source-to-target mappings. For a given line in the CSV file, the `[DB name]` column points to the target column in the database. Then, for each year, it is possible to define two kinds of mappings: 1) simple equivalence mappings, by indicating the name of the source column; 2) data transformation mappings, defined by `CASE` statements in SQL. The `CASE` statements are injected in the code responsible for the transformation. The context of each statement is the current cursor of the SQL execution.

Consider the first equivalence mapping from the motivating example, i.e., for `NATIONALITY`. Five columns, from 2013 to 2017, have the value of the input column `NATIONALITY`. The same is valid for `SPECIAL_NECESSITY`, though it will have different names depending on the year. Non-existent mappings are just left blank. All other data transformations are written using `CASE`, one per year, independently from each other.

Considering the `PROFESSIONAL_STUDIES` column, following code is written to process years 2013 and 2014:

```
CASE
  WHEN (STUDIES_KIND >= 30 AND STUDIES_KIND <= 40) THEN 1
  WHEN (STUDIES_KIND > 40 AND STUDIES_KIND <= 50) THEN 2
END
```

Simple join operations are also supported. For instance, the following expression (`~ SCHOOL.REGION_ID`) written in the mappings of the `ENROLLMENTS` table causes a join with the `SCHOOL` table and returns the `REGION_ID` column.

The mapping definition is kept as simple as possible, yet rich enough to express data translations. The choice of a sub-set of SQL enables fast SELECT + INSERT executions, which would demand handcrafted loops over millions of records with an imperative programming language. The complexity of the translation depends on the `CASE` expression written. They support 1-to-1 and N:1 mappings, and simple joins with other tables. This choice for simplicity is crucial for the continuous maintenance of the mappings.

4 HOTMAPPER DEMONSTRATION

The demonstration will show how to create a mapping definition file, how to use it to create and update tables and its corresponding data. The process will be interactive, so the mapping can be modified if necessary and the tool can be re-executed, to show its efficiency and ease of use.

4.1 Requirements

HOTMapper has following software requirements:

- A Python environment
- The MonetDB[5] database
- The Open Data files in CSV[6]
- A connection configuration file[6]
- The HOTMapper code[6]

After the installation, it is necessary to set up the configurations in the file `settings.py`. It contains the database configurations (login, host, database name) and the path to the CSV files. The utilization of the CLI interface is straightforward:

```
./manage.py create|insert|remap|drop|remap|generate_report
            INPUT_TABLE_NAME PARAMETERS
```

All the options process a mapping definition file with the same name of the input table name, except for the period indication. The development of this file is a central part of the tool usage.

4.2 Using the HOTMapper

This demonstration will show two scenarios, illustrating two workflows: 1) the creation and insertion of an open data source from scratch, and 2) the update of mappings and data of an existing table. In both scenarios the audience will be able to

[5] https://www.monetdb.org/

[6] https://gitlab.c3sl.ufpr.br/tools/hotmapper

propose modifications in the mapping definitions and check the results on the fly.

In the **first scenario**, we will map a table containing information about the undergraduate institutions in Brazil, from 2010 to 2016 (`localoferta_ens_superior`). It is a relatively small table, with approximately 200K records per year, and for its creation the mapping definition file must contain the necessary columns. We execute the command:

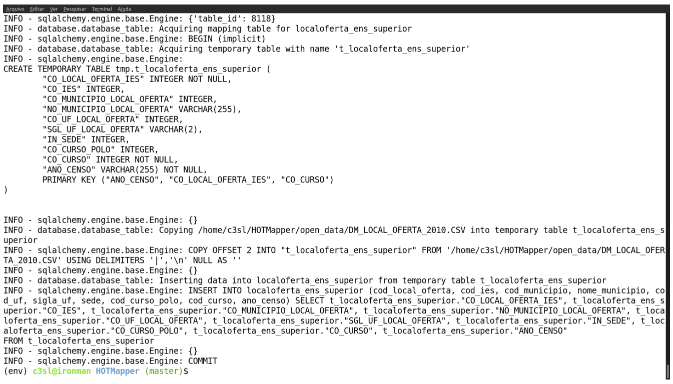
```
./manage.py create localoferta_ens_superior
```

In addition, HOTMapper creates an auxiliary table to store the mapping definitions, called `mapping_localoferta_ens_superior`. These mappings can be used for any subsequent modifications, enabling a faster processing and generation of SQL commands. A final commit is done after the tables are created. Then, we execute the command below to insert the data into the target table.

```
./manage.py insert /FILEPATH/DM_LOCAL_OFERTA_2010.CSV
            localoferta_ens_superior 2010 --sep="|"
```

The process is repeated for data files from each year, i.e., from 2010 to 2016, using the corresponding mapping definitions for each year. An excerpt of the mapping is shown in the three items below: (1) the labels of the mapping file; (2) a complete 1-to-1 mapping with the institution code; (3) a mapping with the institution start date. In this case, we do not have data for every year, leaving the corresponding column blank.

(1) `Label,Std.Label,New Label,DB Name,Type, 2010,2011,2012,2013,2014,2015,2016`
(2) `LOCAL-OFERTA,CO_IES,Institution code,cod_ies,INTEGER, CO_IES,CO_IES,CO_IES,CO_IES,CO_IES,CO_IES,CO_IES`
(3) `LOCAL-OFERTA,DT_INICIO_FUNCIONAMENTO,Start date, data_incio_funcionamento,VARCHAR(255),,,, DT_INICIO_FUNCIONAMENTO,DT_INICIO_FUNCIONAMENTO, DT_INICIO_FUNCIONAMENTO,DT_INICIO_FUNCIONAMENTO`



```
INFO - sqlalchemy.engine.base.Engine: {'table_id': 8118}
INFO - database.database_table: Acquiring mapping table for localoferta_ens_superior
INFO - sqlalchemy.engine.base.Engine: BEGIN (implicit)
INFO - database.database_table: Acquiring temporary table with name 't_localoferta_ens_superior'
INFO - sqlalchemy.engine.base.Engine:
CREATE TEMPORARY TABLE tmp.t_localoferta_ens_superior (
        "CO_LOCAL_OFERTA_IES" INTEGER NOT NULL,
        "CO_IES" INTEGER,
        "CO_MUNICIPIO_LOCAL_OFERTA" INTEGER,
        "NO_MUNICIPIO_LOCAL_OFERTA" VARCHAR(255),
        "CO_UF_LOCAL_OFERTA" INTEGER,
        "SGL_UF_LOCAL_OFERTA" VARCHAR(2),
        "IN_SEDE" INTEGER,
        "CO_CURSO_POLO" INTEGER,
        "CO_CURSO" INTEGER NOT NULL,
        "ANO_CENSO" VARCHAR(255) NOT NULL,
        PRIMARY KEY ("ANO_CENSO", "CO_LOCAL_OFERTA_IES", "CO_CURSO")
)


INFO - sqlalchemy.engine.base.Engine: {}
INFO - database.database_table: Copying /home/c3sl/HOTMapper/open_data/DM_LOCAL_OFERTA_2010.CSV into temporary table t_localoferta_ens_s
uperior
INFO - sqlalchemy.engine.base.Engine: COPY OFFSET 2 INTO "t_localoferta_ens_superior" FROM '/home/c3sl/HOTMapper/open_data/DM_LOCAL_OFER
TA_2010.CSV' USING DELIMITERS '|','\n' NULL AS ''
INFO - sqlalchemy.engine.base.Engine: {}
INFO - database.database_table: Inserting data into localoferta_ens_superior from temporary table t_localoferta_ens_superior
INFO - sqlalchemy.engine.base.Engine: INSERT INTO localoferta_ens_superior (cod_local_oferta, cod_ies, cod_municipio, nome_municipio, co
d_uf, sigla_uf, sede, cod_curso_polo, cod_curso, ano_censo) SELECT t_localoferta_ens_superior."CO_LOCAL_OFERTA_IES", t_localoferta_ens_s
uperior."CO_IES", t_localoferta_ens_superior."CO_MUNICIPIO_LOCAL_OFERTA", t_localoferta_ens_superior."NO_MUNICIPIO_LOCAL_OFERTA", t_loca
loferta_ens_superior."CO_UF_LOCAL_OFERTA", t_localoferta_ens_superior."SGL_UF_LOCAL_OFERTA", t_localoferta_ens_superior."IN_SEDE", t_loc
aloferta_ens_superior."CO_CURSO_POLO", t_localoferta_ens_superior."CO_CURSO", t_localoferta_ens_superior."ANO_CENSO"
FROM t_localoferta_ens_superior
INFO - sqlalchemy.engine.base.Engine: {}
INFO - sqlalchemy.engine.base.Engine: COMMIT
(env) c3sl@ironman HOTMapper (master)$
```

Figure 3: Screenshot of the insertion execution flow

The screen shot of the execution log is shown in Figure 3. The command first inserts the table into a temporary table, with the same structure as the input CSV, in a bulk insert action. Then, it inserts the data into a final table applying the mapping definitions. If this insertion is successful, the tool commits all changes to the MonetDb database. Once a first insertion is done, we will execute the drop command (`./manage.py drop localoferta_ens_superior`), and the audience can propose alterations to the mapping definition file to see different outcomes.

In the **second scenario**, we start from an already existing table, with more complex mappings and records. We use the table from the motivating example in Section 2, with the enrollments of all students from 2013 to 2017. As already stated, it has about 90 columns and millions of records per year. We execute the *remap* and *update* actions for 2013 as follows:

```
./manage.py remap matricula
./manage.py update_from_file /FILEPATH/MATRICULA_2013.csv
   matricula 2013 --columns="profissionalizante" --sep="|"
```

The actions check if there is a difference between the current table specification (`matricula`) and the new mapping definition provided. It updates the table and the data. The mappings have, in addition to simple definitions as the ones from the *scenario 1*, expressions requiring data conversions using `CASE` statements. We will edit the `PROFESSIONAL_STUDIES` mapping, whose excerpt is shown below:

```
IN_PROFISSIONALIZANTE,Educação Profissional,0,
                                          profissionalizante,BOOLEAN,
~CASE
WHEN ("FK_COD_MOD_ENSINO"=1 OR "FK_COD_MOD_ENSINO"=2
OR "FK_COD_MOD_ENSINO"=3) THEN CASE WHEN null THEN null
WHEN ("FK_COD_ETAPA_ENSINO">=30 AND "FK_COD_ETAPA\_ENSINO"<=40)
OR ("FK_COD_ETAPA_ENSINO">=59 AND "FK_COD_ETAPA_ENSINO"<=65)
OR ("FK_COD_ETAPA_ENSINO">=67 AND "FK_COD_ETAPA_ENSINO"<=68)
OR ("FK_COD_ETAPA_ENSINO">=73 AND "FK_COD_ETAPA_ENSINO"<=74)
OR "FK_COD_ETAPA_ENSINO"=57
THEN 1 ELSE 0 END
END
```

The last column is the mapping for year 2013, also repeated for 2014. The remaining years already have the desired information. During the demonstration, this and other mappings can be changed following audience interactions.

To summarize, the HOTMapper demo will show how to write, store and manage mappings and data, from Open Data sources with historical information into an integrated database. The simple definition format and rapid bulk insertions enables efficient management of several Open Data sources over time.

Acknowledgments: this work was partially funded by CAPES, Sistema de Monitoramento de Políticas de Promoção da Igualdade Racial (SNPPIR), Simulador de Custo-Aluno-Qualidade (SASE/MEC).

REFERENCES

[1] Francesca Bugiotti, Damian Bursztyn, Alin Deutsch, Ioana Ileana, and Ioana Manolescu. 2015. Invisible glue: scalable self-tuning multi-stores. In *Conference on Innovative Data Systems Research (CIDR)*.
[2] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K Elmagarmid, Ihab F Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer System.. In *CIDR*.
[3] Jennie Duggan, Aaron J Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, S. Madden, David Maier, Tim Mattson, and Stan Zdonik. 2015. The bigdawg polystore system. *ACM Sigmod Record* 44, 2 (2015), 11–16.
[4] Rudolf Eckelberg, Vytor Bezerra Calixto, Marina A. Hoshiba Pimentel, Marcos Didonet Del Fabro, Marcos Sfair Sunyé, Leticia Mara Peres, Eduardo Todt, Thiago Alves, Adriana Dragone, and Gabriela Schneider. 2018. Educational Open Government Data: From Requirements to End Users. In *Web Engineering - 18th ICWE, Cáceres, Spain, June 5-8, 2018*. 463–470.
[5] Ronald Fagin, Laura M. Haas, Mauricio Hernández, Renée J. Miller, Lucian Popa, and Yannis Velegrakis. 2009. *Clio: Schema Mapping Creation and Data Exchange*. Springer Berlin Heidelberg, Berlin, Heidelberg, 198–236.
[6] Jeff LeFevre, Jagan Sankaranarayanan, Hakan Hacigumus, Junichi Tatemura, Neoklis Polyzotis, and Michael J Carey. 2014. MISO: souping up big data query processing with a multistore system. In *Proc. of the 2014 SIGMOD*. 1591–1602.
[7] Xiao Ling, Alon Halevy, Fei Wu, and Cong Yu. 2013. Synthesizing union tables from the Web. In *IJCAI*.
[8] Renée J. Miller. 2018. Open Data Integration. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 2130–2139. https://doi.org/10.14778/3229863.3240491
[9] Fatemeh Nargesian, Erkang Zhu, Ken Q. Pu, and Renée J. Miller. 2018. Table union search on open data. *Proceedings of the VLDB Endowment* 11, 7 (2018), 813–825. https://doi.org/10.14778/3192965.3192973
[10] Cong Yu and Lucian Popa. 2004. Constraint-based XML query rewriting for data integration. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data - SIGMOD '04* (2004), 371. https://doi.org/10.1145/1007568.1007611

Demonstration

SmartML: A Meta Learning-Based Framework for Automated Selection and Hyperparameter Tuning for Machine Learning Algorithms

Mohamed Maher, Sherif Sakr
University of Tartu, Estonia
{mohamed.abdelrahman,sherif.sakr}@ut.ee

ABSTRACT

Due to the increasing success of machine learning techniques, nowadays, thay have been widely utilized in almost every domain such as financial applications, marketing, recommender systems and user behavior analytics, just to name a few. In practice, the machine learning model creation process is a highly iterative exploratory process. In particular, an effective machine learning modeling process requires solid knowledge and understanding of the different types of machine learning algorithms. In addition, all machine learning algorithms require user-defined inputs to achieve a balance between accuracy and generalizability. This task is referred to as *Hyperparameter Tuning*. Thus, in practice, data scientists work hard to find the best model or algorithm that meets the specifications of their problem. Such iterative and explorative nature of the modeling process is commonly tedious and time-consuming.

We demonstrate `SmartML`, a meta learning-based framework for automated selection and hyperparameter tuning for machine learning algorithms. Being meta learning-based, the framework is able to simulate the role of the machine learning expert. In particular, the framework is equipped with a continuously updated knowledge base that stores information about the meta-features of all processed datasets along with the associated performance of the different classifiers and their tuned parameters. Thus, for any new dataset, `SmartML` automatically extracts its meta features and searches its knowledge base for the best performing algorithm to start its optimization process. In addition, `SmartML` makes use of the new runs to continuously enrich its knowledge base to improve its performance and robustness for future runs. We will show how our approach outperforms the-state-of-the-art techniques in the domain of automated machine learning frameworks.

1 INTRODUCTION

Machine learning is the field of computer science that focuses on building algorithms that can automatically learn from data and automatically improve its performance without end-user instructions, influence or interference. In general, the effectiveness of machine learning techniques mainly rests on the availability of massive datasets, of that there can be no doubt. The more data that is available, the richer and the more robust the insights and the results that machine learning techniques can produce. Nowadays, we are witnessing a continuous growth in the size and availability of data in almost every aspects of our daily life. Thus, recently, we have been witnessing many leaps achieved by machine learning in wide range of fields [1, 9]. Consequently, there are growing demand to have increasing number of data scientists with strong knowledge and good experience with the various machine learning algorithms in order to be able to build models that can achieve the target performance and to keep up with exponential growing amounts of data which is produced daily.

In practice, the machine learning modeling process is a highly iterative exploratory process. In particular, there is no one-model-fits-all solution, i.e, there is no single model or algorithm which is well-known to achieve the highest accuracy for all data set varieties in a certain application domain. Hence, trying many machine learning algorithms with different parameter configurations is commonly considered an inefficient, tedious, and time consuming process. Therefore, there has been growing interest to automate the machine learning modeling process as it has been acknowledged that *data scientists do not scale*[1]. Therefore, recently, several frameworks have been designed to support automating the machine learning modeling process. For example `Auto-Weka` [8] is an automation framework for algorithm selection and hyper-parameter optimization which is based on Bayesian optimization using sequential model-based algorithm configuration (SMAC) and tree-structured parzen estimator (TPE). `Auto-Sklearn` [3] is a framework that has been implemented on top of the popular python `scikit-learn` machine learning package that automatically considers the past performance on similar datasets for its automation decision. Other tools include `Google Vizier` which is based on grid or random search [5] and `TPOT` which is based on genetic programming [10].

In this demonstration, we present `SmartML`, a meta learning-based framework for automated selection and hyperparameter tuning for machine learning algorithms (using 15 classifiers). In our framework, the meta-learning feature is emulating the role of the domain expert in the field of machine learning [4, 11]. In particular, we exploit the knowledge and experience from previous runs by storing a set of data meta-features along with their performance. In addition, our knowledge base is continuously updated after running each task over `SmartML` which contributes to improving framework performance over the time. Our meta-learning mechanism is mainly used for the algorithm selection process in order to reduce the parameter-tuning search space which is conducted using SMAC Bayesian optimization [7]. This is different from other tools [3, 8] which

[1] https://hbr.org/2015/05/data-scientists-dont-scale

	SmartML	**Auto-Weka**	**AutoSklearn**	**TPOT**
Language	R	Java	Python	Python
API	Yes	No	No	Yes
Optimization Procedure	Bayesian Optimization (SMAC)	Bayesian Optimization (SMAC and TPE)	Bayesian Optimization (SMAC)	Genetic Programming, and Pareto Optimization
Number of Algorithms	15 classifiers on top of R	27 classifiers on top of WEKA	15 classifiers on top of scikit learn	15 classifiers on top of scikit learn
Support Ensembling	Yes	Yes	Yes	No
Use Meta-Learning	Yes (incrementally updated KB)	No	Yes (Static)	No
Feature preprocessing	Yes	Yes	Yes	No
Model Interpretability	Yes	No	No	No

Table 1: Comparison between State-of-the-art Automated Machine Learning Frameworks

deal with algorithm selection as one of the parameters to be tuned.

`SmartML` can be used as a package in `R language`, one of the most popular languages in the data science domain, or as a Web application[2]. It is also designed to be programming language agnostic so that it can be embedded in any programming language using its available REST APIs. Table 1 shows a feature comparison between our framework and other state-of-the-art frameworks. In our demonstration, we will show that `SmartML` can outperform other tools especially at small running time budgets by reaching better parameter configurations faster. In addition, `SmartML` has the advantage that its performance can be continuously improved over time by running more tasks which makes `SmartML` *smarter* by getting more experience based on the growing knowledge base.

2 SMARTML ARCHITECTURE

Figure 1 illustrates the framework architecture of `SmartML`. In the *input definition* phase, the user uploads the dataset, choose the required options for features selection and preprocessing, specify which features of the dataset should be included in the modeling process, specify the target column which represents the classes of labels of the instances in the dataset and specify the `time budget` constraint for the framework for conducting the hyper-parameter tuning process. `SmartML` accepts `csv` and `arff` (attribute relation file format developed with the Weka machine learning software) file formats.

In the *preprocessing* phase, `SmartML` starts by performing the feature preprocessing operations specified by the selected features. Table 2 lists the feature preprocessing operations supported by the `SmartML` framework. In this phase, the dataset is randomly split into *training* and *validation* partitions where the former is used in algorithm selection and hyper-parameter tuning while the later is used for evaluating the selected configurations during parameter tuning. In addition, a list of 25 meta-features are extracted from the training split describing the dataset characteristics. Examples of these features include *number of instances*, *number of classes*, *skewness* and *kurtosis of numerical features*, and *symbols of categorical features.*

Currently, `SmartML` supports 15 different classifiers (Table 3). In the *algorithm selection* phase, the meta features of the input dataset at hand, which is extracted during the *preprocessing* phase, are compared with the meta features of the datasets that are stored in the knowledge base in order to identify the *similar* datasets, using a nearest neighbor approach. The dataset similarity detection process follows a weighted mechanism between two different factors. The first factors is the *Euclidean distance* between the meta-features of the dataset at hand and meta-features of all datasets stored in the knowledge base. The second factor is the magnitude of the best performing algorithms on the similar dataset. For example, it may be better to select the top n top performing algorithms on a single very similar dataset than selecting the first outperforming algorithm for n similar datasets. We use the retrieved results of the best performing algorithms on similar dataset(s) to nominate the candidate algorithms for the dataset at hand.

In the *hyper-parameter tuning* phase, `SmartML` attempts to tune the selected classifiers hyper-parameters for achieving the best performance. In particular, the knowledge base contains information about the best parameter configurations for each algorithm on each dataset. The configurations of the nominated best performing algorithms are used to initialize the hyper-parameter tuning process for the selected algorithms. The time budget constraint specified by the end user represents the time used in hyper parameter tuning of the selected classifiers. In particular, this budget is divided among all the selected algorithms according to the number of hyper-parameters to tune in each algorithm (Table 3). `SmartML` applies the SMAC technique for hyper-parameter optimization [7]. In particular, SMAC attempts to draw the relation between the algorithm performance and a given set of hyper-parameters by estimating the predictive mean and variance of their performance along the trees of the random forest model. The main advantage of using SMAC is its robustness by having the ability to discard low performance parameter configurations quickly after the evaluation on low number of folds of the dataset [7].

Finally, the results obtained from the hyper-parameter tuning process of the different nominated algorithms are compared with each other to recommend the best performing algorithm to the end user. In addition, a weighted ensembling [2] output of the top performing algorithms can be recommended to the end user based on their choice. In addition, we have integrated the `Interpretable Machine Learning (iml)` package[3] in order to explain for the user the most important features that have been used by the

[2] https://bigdata.cs.ut.ee/smartml/index.html

[3] https://cran.r-project.org/web/packages/iml/index.html

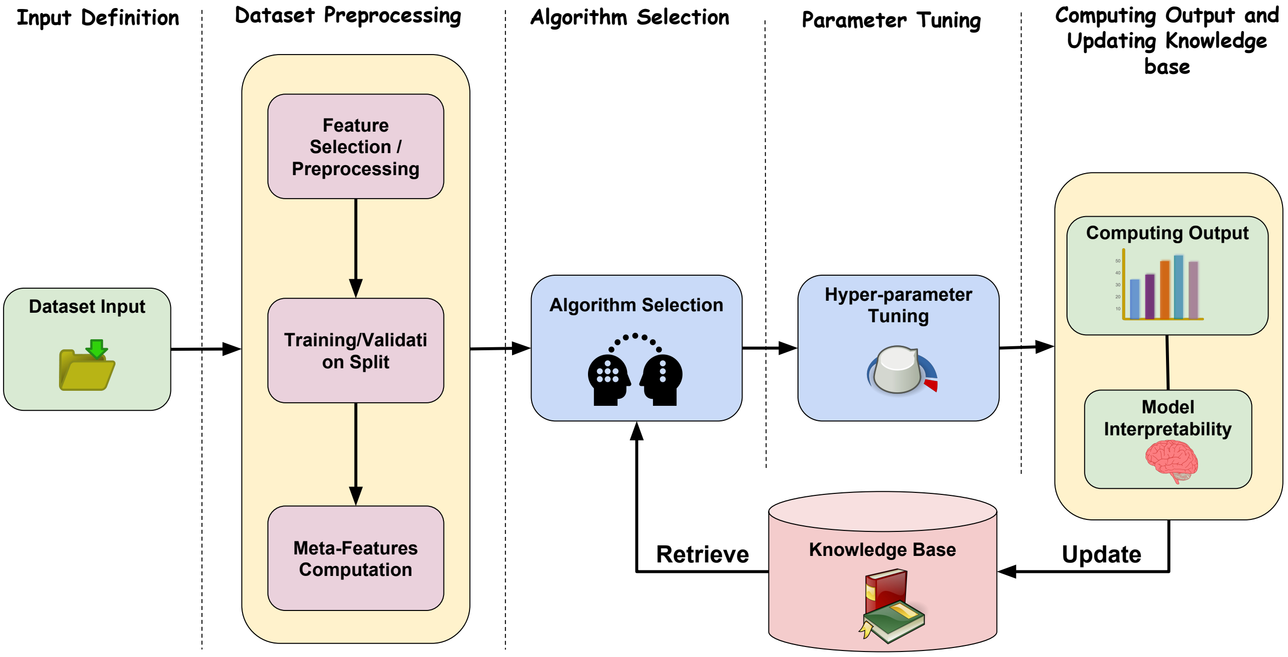


Figure 1: `SmartML` : Framework Architecture

center	subtract mean from values
scale	divide values by standard deviation
range	values normalization
zv	remove attributes with zero variance
boxcox	apply box-cox transform to non-zero positive values
yeojohnson	apply Yeo-Johnson transform to all values
pca	transform data to the principal components
ica	transform data to their independent components

Table 2: Integrated Feature Preprocessing Algorithms

Classification Algorithm	Categorical parameters	Numerical parameters	Package
SVM	1	4	e1071
NaiveBayes	0	2	klaR
KNN	0	1	FNN
Bagging	0	5	ipred
part	1	2	RWeka
J48	1	2	RWeka
RandomForest	0	3	randomForest
c50	3	2	C50
rpart	0	4	rpart
LDA	1	1	MASS
PLSDA	1	1	caret
LMT	0	1	RWeka
RDA	0	2	klaR
NeuralNet	0	1	nnet
DeepBoost	1	4	deepboost

Table 3: Integrated Classifier Algorithms

selected model for directing its prediction process [6]. The interactive interface of our system has been designed using the `Shiny` R Package[4].

Dataset	# Att.	# Classes	# Instances	Auto-Weka Accuracy	SmartML Accuracy
abalone	9	2	8192	25.14	**27.13**
amazon	10000	49	1500	57.56	**58.89**
cifar10small	3072	10	20000	30.25	**37.02**
gisette	5000	2	2800	93.71	**96.48**
madelon	500	2	2600	55.64	**73.84**
mnist Basic	784	10	62000	89.72	**94.91**
semeion	256	10	1593	89.32	**94.13**
yeast	8	10	1484	51.80	**66.23**
Occupancy	5	2	20560	93.99	**95.55**
kin8nm	8	2	8192	93.99	**96.42**

Table 4: Performance Comparison: `SmartML` VS `Auto-Weka`

3 DEMO SCENARIO

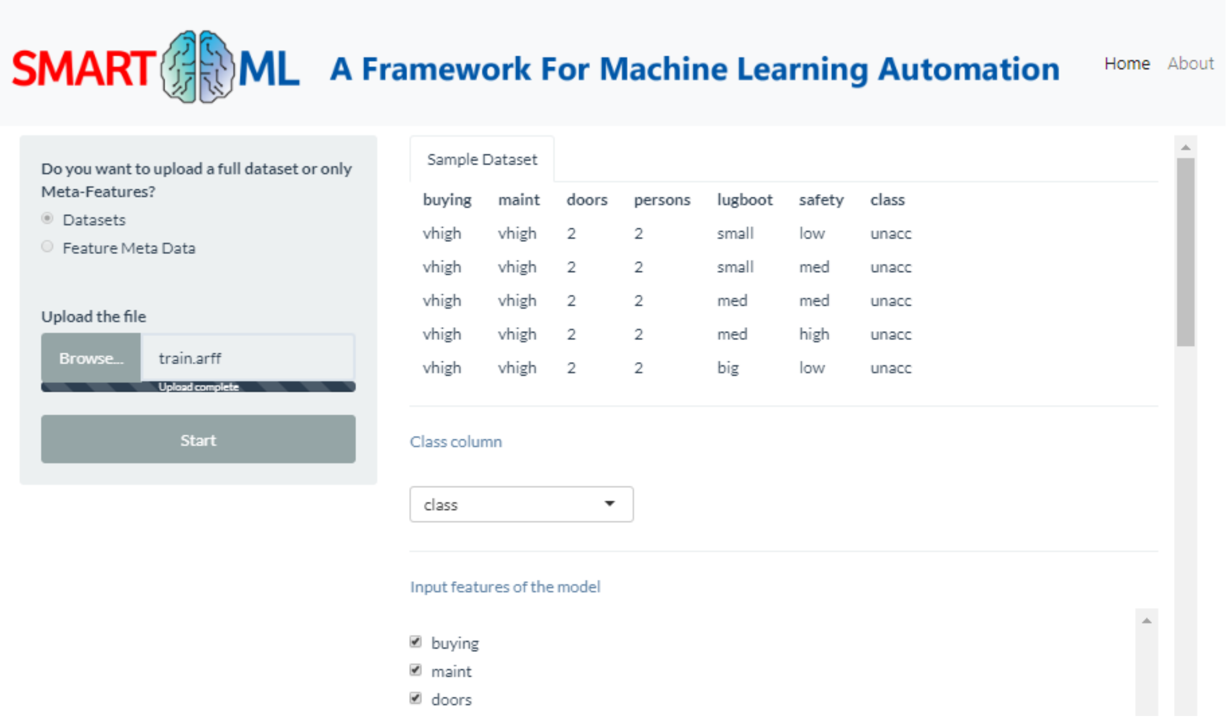
`SmartML` is available both as a Web application as well as RESTful APIs[5]. In this demonstration[6], we will present to the audience the workflow of the `SmartML` framework (Figure 1). In particular, we will show that how our approach can help non-expert machine learning users to effectively identify the machine learning algorithms and their associated hyperparameter settings that can achieve optimal or near-optimal accuracy for their datasets with little effort.

We start by introducing to the audience the challenges we tackle, the main goal and the functionalities of our framework. Then, we take the audience through the automated algorithm selection and hyper-parameter tuning process for sample datasets. We start by showing different features which is provided for the end-user (Figure 2). For example, the user can upload either a dataset file or a direct URL for the dataset. In addition, the user can choose either to perform both algorithm selection and hyper-parameter

[4] https://shiny.rstudio.com/

[5] The source code of the `SmartML` framework is available on https://github.com/DataSystemsGroupUT/Auto-Machine-Learning

[6] A demonstration screencast is available on https://www.youtube.com/watch?v=m5sbV1P8oqU&feature=youtu.be



Do you want to upload a full dataset or only Meta-Features?

● Datasets

○ Feature Meta Data

Upload the file

Browse... train.arff

Upload complete

Start

Sample Dataset

buying	maint	doors	persons	lugboot	safety	class
vhigh	vhigh	2	2	small	low	unacc
vhigh	vhigh	2	2	small	med	unacc
vhigh	vhigh	2	2	med	med	unacc
vhigh	vhigh	2	2	med	high	unacc
vhigh	vhigh	2	2	big	low	unacc

Class column

class

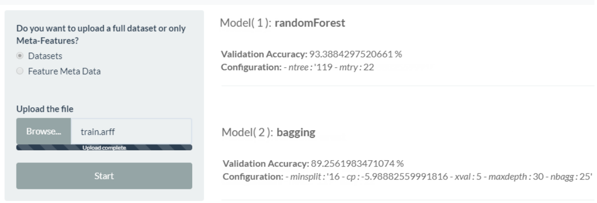
Input features of the model

☑ buying

☑ maint

☑ doors

Figure 2: Screenshot: Configuring an experiment for a dataset



Do you want to upload a full dataset or only Meta-Features?

● Datasets

○ Feature Meta Data

Upload the file

Browse... train.arff

Upload complete

Start

Model(1): randomForest

Validation Accuracy: 93.3884297520661 %
Configuration: - ntree : '119 - mtry : 22

Model(2): bagging

Validation Accuracy: 89.2561983471074 %
Configuration: - minsplit : '16 - cp : -5.98882559991816 - xval : 5 - maxdepth : 30 - nbagg : 25'

Figure 3: Screenshot: Sample experiment output from SmartML

tuning or only algorithm selection. In the later case, it is possible to upload only the dataset meta-features file instead of the whole dataset. The user will be also able to configure different options such as whether any kind of feature preprocessing is needed or not, whether model interpretability is needed or not and specify the time budget for hyper-parameter tuning. Then, we will take the audience through the different phases of the framework until returning the final results (Figure 1).

Table 4 shows the performance comparison between SmartML and Auto-Weka[7] using 10 datasets where a *time budget* of 10 minutes has been allocated for each dataset in each framework. In our experiments, we have bootstrapped the knowledge base of SmartML using 50 datasets from various sources including OpenMl[8], UCI repository[9] and Kaggle[10]. The results show that, using this relatively very small knowledge base, the accuracy results of SmartML outperform the results of Auto-Weka for all the datasets. As a part of our demonstration, we will provide the audience with the live chance to compare the performance of SmartML with Auto-Weka and other related frameworks using various datasets.

ACKNOWLEDGMENT

This work is funded by the European Regional Development Funds via the Mobilitas Plus programme (grant MOBTT75)

REFERENCES

[1] Rahul C Deo. 2015. Machine learning in medicine. *Circulation* 132, 20 (2015), 1920–1930.

[2] Thomas G Dietterich. 2000. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*.

[3] Matthias Feurer et al. 2015. Efficient and Robust Automated Machine Learning. In *NIPS*.

[4] Matthias Feurer et al. 2015. Initializing Bayesian Hyperparameter Optimization via Meta-learning. In *AAAI*.

[5] Daniel Golovin et al. 2017. Google Vizier: A Service for Black-Box Optimization. In *KDD*.

[6] Riccardo Guidotti et al. 2018. A survey of methods for explaining black box models. *ACM CSUR* 51, 5 (2018).

[7] Frank Hutter et al. 2011. Sequential Model-based Optimization for General Algorithm Configuration. In *LION*.

[8] Lars Kotthoff et al. 2017. Auto-WEKA 2.0: Automatic Model Selection and Hyperparameter Optimization in WEKA. *J. Mach. Learn. Res.* 18, 1 (2017).

[9] Sendhil Mullainathan and Jann Spiess. 2017. Machine learning: an applied econometric approach. *Journal of Economic Perspectives* 31, 2 (2017).

[10] Randal S. Olson and Jason H. Moore. 2016. TPOT: A Tree-based Pipeline Optimization Tool for Automating Machine Learning. In *Proceedings of the Workshop on Automatic Machine Learning*.

[11] Matthias Reif et al. 2012. Meta-learning for Evolutionary Parameter Optimization of Classifiers. *Mach. Learn.* 87, 3 (2012).

[7] https://www.cs.ubc.ca/labs/beta/Projects/autoweka/

[8] https://www.openml.org/

[9] http://archive.ics.uci.edu/ml/index.php

[10] Kaggle:https://www.kaggle.com/

devUDF: Increasing UDF development efficiency through IDE Integration. It works like a PyCharm!

Mark Raasveldt
CWI
Amsterdam
m.raasveldt@cwi.nl

Pedro Holanda
CWI
Amsterdam
holanda@cwi.nl

Stefan Manegold
CWI
Amsterdam
manegold@cwi.nl

ABSTRACT

User-defined functions (UDFs) facilitate the execution of analytics pipelines inside the database. They provide many advantages over traditional methods, such as close-to-data execution and automatic parallelization. However, the standard workflow for developing and debugging UDFs does not allow developers to use their regular toolchains and Integrated Development Environments (IDEs). As a result, writing functional UDFs is challenging. In this demo, we present the devUDF, a plugin to the PyCharm IDE that allows developers to develop and debug their MonetDB/Python UDFs directly from within the IDE.

1 INTRODUCTION

To perform data analysis, data scientists frequently use scripting languages, such as R and Python. These languages have a huge ecosystem of existing machine-learning and classification libraries (e.g., TensorFlow [1] or Sci-Kit Learn [6]). Using these languages in conjunction with a relational database management system (RDBMS) has many advantages, as the RDBMS can offer robust storage of data and handle common data wrangling operations. The traditional method of combining a RDBMS with these scripting languages is to connect to a RDBMS using a client protocol. The data is then transferred from the database to the analytical tool. However, this is not efficient when a large amount of data needs to be retrieved [8].

This issue can be solved by in-database analytics. By performing the analytics inside the database, the data transfer overhead is mitigated [7]. The primary way of performing in-database analytics is through the use of UDFs. To facilitate this, most RDBMS vendors support UDFs in at least one scripting language frequently used for analysis [3].

The development workflow for UDFs differs depending on the DBMS that is used. Certain databases provide their own custom tools for developing UDFs, such as pgAdmin [9] for Postgres and ODS [4] for DB2 and Oracle. However, these tools have a number of limitations. They are not database agnostic, only work for developing UDFs written in PL/SQL and require developers to learn how to use complex tools designed for DBAs.

The generic workflow for developing a UDF is to write a function using a simplistic text editor. The function can then be created inside the RDBMS through a SQL command, and used by calling it within a SQL query. If there are bugs or problems within the UDF, the function has to be recreated and the SQL query has to be rerun. This process has to be repeated until the problem is fixed.

This workflow is problematic when developing complex UDFs, as advanced IDE features and modern debugging techniques cannot be used. Using these IDE features is not easily doable because the developer has to manually perform code transformations to convert the Python code to a SQL command that creates the UDF. As seen in Table 1 [2], IDEs are heavily preferred for development over simplistic text editors due to their development features. Therefore, we argue that offering support for the usage of these features in the development workflow of UDFs will make developing UDFs more attractive, faster and easier for many developers.

Name	Market Share	Type
`Eclipse`	25.2%	**IDE**
`Visual Studio`	19.5%	**IDE**
`Android Studio`	9.5%	**IDE**
`Vim`	7.9%	Text Editor
`XCode`	5.2%	**IDE**
`IntelliJ`	4.8%	**IDE**
`NetBeans`	4.0%	**IDE**
`Xamarin`	3.8%	**IDE**
`Komodo`	3.4%	**IDE**
`Sublime Text`	3.3%	Text Editor
`Visual Studio Code`	3.3%	Text Editor
`PyCharm`	2.3%	**IDE**

Table 1: Most Popular Development Environments.

IDEs are also attractive because they facilitate the usage of sophisticated interactive debugging techniques, such as stepping through the code line by line and pausing code execution. However, these techniques cannot be used in conjunction with UDFs because the RDBMS must be in control of the code flow while the UDF is being executed. Instead, developers have to resort to inefficient debugging strategies (e.g., print debugging) to make their code work [3].

Another issue with the standard UDF workflow is that UDFs are stored within the database server. As a result, version control systems (VCSs) such as Git [5] cannot be easily integrated to keep track of changes to UDFs. Without a VCS, cooperative development is challenging and the development history is not stored.

In this demo we showcase devUDF, a plugin for the popular IDE PyCharm that facilitates developing and debugging MonetDB/Python UDFs [7] directly from within the IDE. Using our plugin, advanced debugging features can be used while refining and refactoring UDFs.

2 THE DEVUDF PLUGIN

The devUDF plugin is developed for the PyCharm IDE that facilitates the usage of advanced IDE features for development of MonetDB/Python UDFs. It allows developers to create, modify and test UDFs without leaving their IDE environment. All

Series ISSN: 2367-2005

10.5441/002/edbt.2019.55

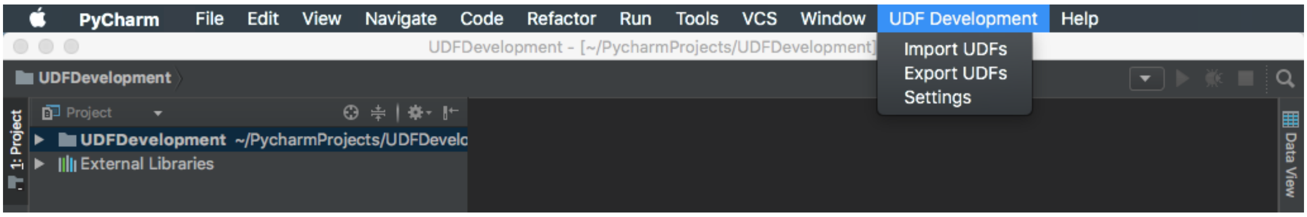
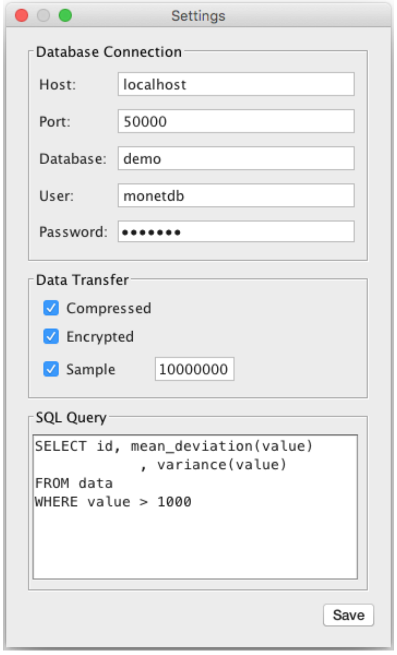


Figure 1: PyCharm Main Menu.

features of the IDE can be used to develop UDFs, including the sophisticated interactive debugger and VCS support.



Host: localhost

Port: 50000

Database: demo

User: monetdb

Password: •••••••

☑ Compressed

☑ Encrypted

☑ Sample 10000000

```
SELECT id, mean_deviation(value)
              , variance(value)
FROM data
WHERE value > 1000
```

Figure 2: Settings.

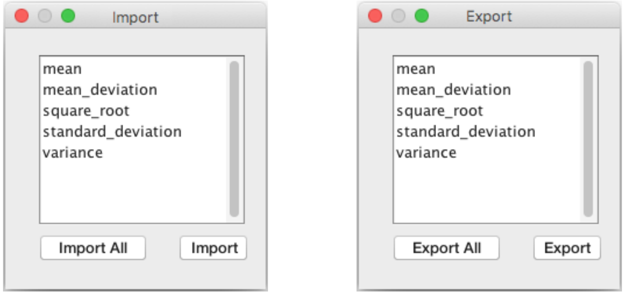
2.1 Usage

The devUDF plugin can be accessed through the main menu of the IDE (See Figure 1). In this menu, a submenu labeled "UDF Development" contains the three main aspects of the plugin.

Initially, devUDF must be configured so it can connect to an existing database server. This can be done through the settings window shown in Figure 2. The parameters required are the usual database client connection parameters (i.e., host, port, database, user and password).

After the devUDF plugin has been configured to connect to a running database server, the development process begins by importing the existing UDFs within the server into the development environment. This is done through the "Import UDFs" window, shown in Figure 3(a). The developer has the option to select the functions that he wishes to import, or he can choose to import all functions stored within the database server.

After the UDFs are imported, the code of the UDFs is exported from the database and imported into the IDE as a set of files in the current project. The developer can then modify the code of the UDFs in these files, use version control to keep track of changes to the UDFs and export the UDFs back to the database server for execution through the "Export UDFs" window (see Figure 3(b)).



(a) Import

(b) Export

Figure 3: Importing and Exporting UDFs from the Database.

The developer can also run any of the imported UDFs with the IDEs interactive debugger by running the project as they would run a normal PyCharm project (using the "Debug" command). Since a UDF is never executed in isolation, but always within the context of a SQL query, the user must provide a SQL query which executes the to-be-debugged UDF. This SQL query must be specified in the Settings menu (see Figure 2).

Running the UDF in the interactive debugger will execute the function locally on the developers' machine instead of remotely inside the database server. As the UDF requires data from the database (as its input parameters), the data must be transferred from the database server to the developers machine. For this data transfer, the developer can configure another set of options. As the data can be large, we offer a method of compressing the data during the transfer, leading to faster transfer times. In addition, the developer can choose to execute the UDF using a uniform random sample of the input data instead of the full set of input data. This will alleviate the data transfer overhead.

Since the data contained inside the database server might be sensitive, and it must be exported for debugging purposes, we also offer an optional encryption feature that can be used to safely transfer the sensitive data.

2.2 Implementation

The devUDF plugin works by connecting to the database using a JDBC connection. It then extracts the source code of the UDF together with its input parameters from the database by querying the databases' meta tables. An example of how MonetDB stores the source code of a Python function is shown in Listing 1. In order to be able to execute the UDF locally a set of code transformations has to be applied to this code, as the database only contains the function body. We need to create the header of the function using the function name and its parameters. To then run the created function, we need to obtain the input data from the database. In the generated code, we load the input data from a binary blob using the pickle library and pass it as a parameter to the function. The final transformed code is shown in Listing 2. When the user wants to export the UDF back to the database, these transformations are reversed and only the function body is committed.

When the user wants to debug the UDF locally using the interactive debugger, the input data of the function has to be extracted from the database. To obtain the input data, we take the user-submitted SQL query containing the call to the UDF, and we replace the call to the UDF with a predefined extract function that transfers the input data back to the client instead of executing the UDF inside the server. We then run the transformed SQL query inside the database server to obtain the input data, store it on the developers machine and run the code of the transformed UDF.

The extract function used changes depending on the data transfer options selected by the user. If encryption is requested, the data is encrypted by the extract function before being transferred using the password of the database user as a key. The client then reverses the encryption to obtain the actual input data. The compression option works in a similar fashion. If the sample option is enabled, a uniform random sample of a size specified by the user is taken before extracting the data from the database server.

```
+----------------+-----------------------------------+
| name           | func                              |
+================+===================================+
| train_rnforest | {                                 |
:                : import pickle                     :
:                : from sklearn.ensemble             :
:                :     import RandomForestClassifier :
:                :                                   :
:                : clf = RandomForestClassifier(n)   :
:                : clf.fit(data, classes)            :
:                : return {'clf': pickle.dumps(clf), :
:                :       'estimators':n }            :
:                : };                                :
+----------------+-----------------------------------+
```

Listing 1: MonetDB UDF Example.

2.3 Nested UDFs

Loopback queries inside UDFs are supported by MonetDB/Python. They allow users to query the database directly from within the UDF. The results of the query are converted to the host language of the UDFs. In MonetDB/Python UDFs, loopback queries can be can issued through the _conn object that is passed to every UDF. They are useful because they can bypass cardinality restrictions of the relational querying model.

The loopback queries can also contain UDFs themselves. An example of a nested UDF is shown in Listing 3. This UDF calls the function depicted in Listing 2 with a set of different parameters in order to find the best classifier and its parameters. In order to provide support for extracting and debugging these nested UDFs, we must execute the same transformation steps on the nested UDFs as we did for the main UDF being executed. With an additional transformation rule on the _conn object to the correct function call. After being transformed, we can execute the nested UDFs locally by transferring their input data in conjunction with the main UDF data, Finally they can be executed from within the IDE.

```
import pickle

def train_rnforest(data, classes, n_estimators):
    import pickle
    from sklearn.ensemble
        import RandomForestClassifier
    clf = RandomForestClassifier(n_estimators)
    clf.fit(data, classes)
    dict = {'classifier': pickle.dumps(clf),
            'estimators':n_estimators }
    return dict;

input_parameters =
        pickle.load(open('./input.bin','rb'))

train_rnforest(input_parameters['data'],
        input_parameters['classes'],
        input_parameters['n_estimators'])
```

Listing 2: Exported UDF Code Example.

```
CREATE FUNCTION find_best_classifier(esttest INT)
RETURNS DOUBLE LANGUAGE PYTHON {
import pickle
(tdata, tlabels) = _conn.execute("""SELECT data,
    labels FROM testingset""");
best_classifier = None
best_classifier_answers = -1
best_estimator = -1
for estimator in esttest:
    res = _conn.execute
        ("""
    SELECT *
    FROM train_rnforest(
        (SELECT data, labels
        FROM trainingset), %d);
    """ % estimator)
    classifier = pickle.loads(res['clf'])
    predictions = classifier.predict(tdata)
    correct_pred = predictions == tlabels
    correct_ans = numpy.sum(correct_predictions)
    if correct_ans > best_classifier_answers:
        best_classifier = classifier
        best_classifier_answers = correct_ans
        best_estimator = estimator
return {'clf': best_classifier,
        'n_estimators': best_estimator}
};
```

Listing 3: Nested UDF Example.

2.4 Extensions

Extending to Other Databases. Our solution is implemented for MonetDB. However, our plugin can be easily extended to work with other RDBMSes, as the same implementation strategy can be used. However, the processing model of the respective database needs to be taken into consideration. MonetDB uses the operator-at-a-time processing model, which means the UDFs are only called once with the entire columns as input. Row-store databases (e.g., Postgres or MySQL) use the tuple-at-a-time processing model, under which the UDFs are called many times with only individual rows as input. As this changes the way UDFs are called, the execution of the UDF must be adapted to these differing processing models. The tuple-at-a-time execution method can be simulated by issuing a loop over the input tuples.

Extending to Other UDF Languages. Our solution is implemented for Python/PyCharm. However the plugin is fully developed in Java and compatible with all the other JetBrains IDEs. In order to extend our plugin for other UDF languages, the code transformations for the new language must be added into the plugin. Additional care must be taken when dealing with compiled languages. Our model assumes that the RDBMS stores the source code of the UDF. If the database stores only a compiled blob of the UDF, the code transformations cannot be applied and an alternate solution must be used. In addition, when dealing with compiled languages some additional work must be performed on compiling and linking the code prior to execution.

2.5 Demo Outline

In the general outline for our interactive demo, we will introduce the typical setup for UDF Development. The general presentation for all the scenarios is as follows:

(1) We introduce the typical setup for UDF Development: Developers write code in their text editor of choice, insert the UDF into the database through a SQL command, repeating this process if the UDF has any bugs.
(2) We show common pitfalls in developing a UDF. Foremost, we focus on issues related to debugging.
(3) We show how bugs can be located using simplistic debugging strategies like print debugging.
(4) Finally, we repeat the same process but using devUDF to facilitate the development workflow. Showcasing how easy, fast and secure it is to use in the UDF development workflow.

In our demonstration we will ingest several CSV files, located in one directory, with one column of integers, our final goal is to create a UDF that calculates the mean deviation of said column, as a reference we compare the results with a correct version of the function. As common pitfalls, we will showcase the following scenarios:

Scenario A. In this scenario we present a UDF that calculates the median of a column with a bug, depicted in Listing 4. In line 9, the regular difference is calculated instead of the absolute difference which produces a semantic error, that is syntactically correct but logically incorrect.

Scenario B. Now, we use a correct version of the *mean_deviation* function. However, we introduce a bug in our data loader. Creating a data dependent error depicted in listing 5. In line 5 we introduce the bug that skips one of the CSV files in a given directory because it considers that *range* is right side inclusive.

3 SUMMARY

When it comes to assessing the potential impact of devUDF, we point out two current trends: First, the use of UDFs to perform In-Database Analytics is gaining popularity with support of many languages in major DBMSs. Especially in data science environments when the data is already stored inside a DBMS. Second, IDEs like Eclipse, IntelliJ and PyCharm have been gaining popularity over more simplistic text editors. Looking at both trends, we see a growing market for tools like devUDF, especially considering the void it fills in UDF development workflow.

```
1  CREATE FUNCTION mean_deviation(column INTEGER)
2  RETURNS DOUBLE LANGUAGE PYTHON {
3    mean = 0
4    for i in range (0, len(column)):
5        mean += column[i]
6    mean = mean / len(column)
7    distance = 0
8    for i in range (0, len(column)):
9        distance += column[i] - mean
10   deviation = distance/len(column)
11   return deviation;
12 };
```

Listing 4: Wrong mean deviation.

```
1  CREATE FUNCTION loadNumbers(path STRING)
2  RETURNS TABLE(i INTEGER)
3  LANGUAGE PYTHON {
4  files = os.listdir(path)
5  result = []
6  for i in range (0,len(files)-1):
7      file = open(files[i],"r")
8      for line in file:
9          result.append(int(line))
10 return result
11 };
```

Listing 5: Wrong data loader.

4 ACKNOWLEDGMENTS

This work was funded by the Netherlands Organisation for Scientific Research (NWO), projects "Data mining on high volume simulation output" (Holanda) and "Process mining for multi-objective online control" (Raasveldt).

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Pierre Carbonnelle. 2018. Top IDE index. https://pypl.github.io/IDE.html

[3] Pedro Holanda, Mark Raasveldt, and Martin Kersten. 2017. Don't Hold My UDFs Hostage - Exporting UDFs For Debugging Purposes. In *SSBD, Brazil.*

[4] IBM. 2018. Optim Development Studio. https://www.ibm.com/support/knowledgecenter/en/SSZ6WM_3.0.0/com.ibm.dbapp.doc/cloud-tools/cods_overview.html

[5] Jon Loeliger and Matthew McCullough. 2012. *Version Control with Git: Powerful tools and techniques for collaborative software development.* " O'Reilly Media, Inc.".

[6] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, Oct (2011), 2825–2830.

[7] Mark Raasveldt and Hannes Mühleisen. 2016. Vectorized UDFs in Column-Stores. In *SSDBM.*

[8] Mark Raasveldt and Hannes Mühleisen. 2017. Don't Hold My Data Hostage-A Case For Client Protocol Redesign. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1022–1033.

[9] The pgAdmin Development Team. 2018. pgAdmin 4. https://www.pgadmin.org/docs/pgadmin4/dev/l

ML2SQL

Compiling a Declarative Machine Learning Language to SQL and Python

Maximilian E. Schüle
schuele@in.tum.de

Matthias Bungeroth
bungeroth@in.tum.de

Dimitri Vorona
vorona@in.tum.de

Alfons Kemper
kemper@in.tum.de

Stephan Günnemann
guennemann@in.tum.de

Thomas Neumann
neumann@in.tum.de

Technical University of Munich

ABSTRACT

This demonstration presents a machine learning language *MLearn* that allows declarative programming of machine learning tasks similarly to SQL. Our demonstrated machine learning language is independent of the underlying platform and can be translated into SQL and Python as target platforms. As modern hardware allows database systems to perform more computational intense tasks than just retrieving data, we introduce the *ML2SQL* compiler to translate machine learning tasks into stored procedures intended to run inside database servers running PostgreSQL or HyPer. We therefore extend both database systems by a gradient descent optimiser and tensor algebra.

In our evaluation section, we illustrate the claim of running machine learning tasks independently of the target platform by comparing the run-time of three in *MLearn* specified tasks on two different database systems as well as in Python. We infer potentials for database systems on optimising tensor data types, whereas database systems show competitive performance when performing gradient descent.

1 INTRODUCTION

Database systems provide with SQL a declarative language that allows data manipulation and data retrieving without caring about optimisation details. With increasing hardware performance, database systems will not fully exploit the servers' hardware potentials as long as they are used for data retrieval only. To shift computation to the data stored in database systems, algorithms can be specified in SQL—as it has been Turing complete since providing recursive tables—or as user-defined functions. The latter allow injecting code as stored procedures to be executed inside the database system and make an additional data manipulation layer on top obsolete. Even though the run-time would decrease, user-defined functions are not fully established as they form a mixture of declarative and procedural language and are inconvenient to express for data scientists.

When dealing with data and minimisation problems, dedicated tools as TensorFlow [1] or Pytorch form the status quo for performing machine learning tasks with tensors and gradient descent. Another approach of formulating machine learning tasks is using a declarative language as MLog [7], that compiles to code using TensorFlow, but, so far, it lacks support for use together with database systems. For computations inside of database systems, the support of linear algebra together with matrices or tensors is essential. Different studies focus on the integration of linear algebra [8] and matrices inside of database systems [5]. Going one step further, so called array database systems replace relations by arrays as the native way of storing attributes. To support machine learning, TensorDB [6] aims at providing tensor calculus on top of array database systems. One study even provide an own declarative language (BUDS) [2] on top of a prototyped database system that supports matrix data types. Comparable domain specific languages are Weld[1] for data driven workloads and IBM SystemML[2] for creating flexible algorithms, but both cannot be used inside of database systems. The intermediate language Ferry [3] allows to translate from various code (i.e. Ruby or Haskell) into SQL but is not designed for use with array datatypes.

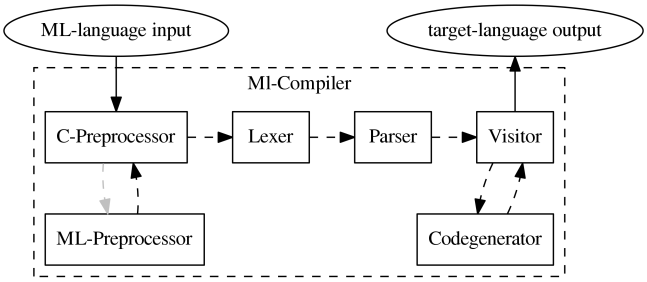


Figure 1: The compilation process: the *MLearn* language first gets preprocessed twice for handling includes, then the language gets tokenised and parsed. For each target platform, a generator allows to translate the abstract syntax tree into the target language.

However, while linear algebra in database systems have been integrated and declarative language concepts have been proposed, there is no successful study on bringing a declarative language, tensor calculus and gradient descent in database systems together. We therefore develop a declarative machine learning language aimed at optimising models for supervised machine learning and data analysis. Our *MLearn language* allows specifying tasks independently of the target language, changing the underlying engine and makes it easy to compare run-times and results of different underlying frameworks. This demonstration presents the *ML2SQL* compiler in particular, which compiles code written in *MLearn* to SQL (for PostgreSQL or HyPer [4]) or to Python using the frameworks NumPy and TensorFlow (see Fig. 1).

This demonstration paper is structured as follows: First, we introduce the *MLearn* language specifications and the details of the corresponding compiler. We evaluate the run-time of the generated code on the target platforms using the Chicaco taxi

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

[1] https://github.com/weld-project/weld
[2] https://systemml.apache.org

10.5441/002/edbt.2019.56

dataset as input data and linear regression as optimisation model (optimised by gradient descent or as closed form solution). At the end, we introduce the demonstration concept including our web interface for online testing and conclude by improvements that might increase database systems' computation performance.

2 MLEARN AND THE ML2SQL COMPILER

The *MLearn* language is designed to define machine learning tasks in a declarative manner to be compiled to SQL or Python. We begin by introducing the language specification needed for enjoying the demonstration as a visitor. We precede by listing the prerequisites of the target platforms in order to run the introduced tasks. Finally, we will give examples on how to use the *MLearn* language during the demonstration later on.

2.1 Language Specification

All operations work on integers, floating point numbers, Boolean values or strings as basic types, which can be composed to tensors. On these types, *MLearn* provides the following features (s. Lst. 5):

- **Reading CSV files** as the fundamental operation to store the data in variables or relations of the database system.
- **Mathematical expressions** as provided by NumPy or SQL (as part of the projection operator).
- **Tensors** form the main part in our computations. Beside mathematical operations we support accessing, slicing, concatenation and transposition.
- **Functions** allow to structure the code and to reduce code duplication. Also external functions imported from other files or of a target language are allowed.
- **Control blocks.** In addition to our declarative statements, we allow conditional expressions and loops.
- **Distributions** are used for data sampling when initializing tensors with random values.
- **Preprocessor statements** as known from C can be used to include files and to allow the abstraction of different functions to different files.
- **k-fold cross validation** as a predefined building block splits up a data set into training and test sets to find the best—so-called—hyper parameters.
- **Gradient descent** as a separate building block optimises the weights for a given loss function on input data.

2.2 Target Language

We designed our machine learning language to compile to Python code with the libraries that data scientists would use. To work with SQL we picked out the disk-based database system PostgreSQL and its main-memory counterpart, HyPer. We assume for both systems an underlying script language, PL/pgSQL in PostgreSQL and HyPerScript in HyPer, that combines declarative SQL statements with procedural control blocks. The tensor operations in Python are performed using NumPy library calls. HyPer has already implemented all basic tensor operations including addition, (scalar) multiplication, power (including inverse for matrices on negative exponents), transposition, initializing an identity matrix and filling a matrix by a predefined value. As those operations do not exist in PostgreSQL, we have implemented these operations as C library function calls, also supporting parallelism. Furthermore, we make use of predefined array operations as slicing and concatenation to divide the input dataset into training and testing one. Also, we wrap a PostgreSQL library extension around our already presented gradient descent [9] library to allow in-database gradient descent in PostgreSQL.

2.3 Example

Fig. 2 shows the exemplary usage of the *MLearn* language where we specify linear regression as closed form solution:

$$\vec{w} = (X'^T X')^{-1} X'^T \vec{y}.$$

The example code (see Lst. 1) splits a tensor (A) into features (X, the first three attributes) and labels (y). Then a tensor out of the value 1 as bias is prepended in front of the features. Afterwards, the optimal weights (w) are computed out of tensor algebra. The compiled code to Python can be seen in Lst. 2, the one for PostgreSQL in Lst. 4 and the one for HyPer in Lst. 3. We can see that the code written in our declarative language is much more compressed.

3 EVALUATION

For evaluation (s. Fig. 3), we specify linear regression as closed form or using gradient descent in our machine learning language and let the tasks run on the following target platforms: PostgreSQL version 10.5, Python 2.7.15 with NumPy 1.13.3 and TensorFlow 1.3.0 and the current HyPer system. We used an Ubuntu 18.04.01 LTS server with two sockets and twenty cores of Intel Xeon E5-2660 v2 processors in total (supporting hyper-threading). The server has 256 GiB of main memory and uses 1 TiB of SSD as background storage. As test data served 85 mio. tuples of the Chicaco taxi rides dataset[3].

We tested the run-time of loading data from CSV (s. Fig. 3a), the run-time of linear regression in closed form (s. Fig. 3b) and by using gradient descent (s. Fig. 3c). For gradient descent, we used a learning rate of 0.0000005 and varied the number of iterations from 1 to 10^4, but we used a constant input size (the whole dataset). The time measurements consider only the run-time needed for the specific operations, the time for data loading and array creation is measured separately.

The results show that data loading took in all three systems about the same time, no system seems to dominate one another. For the matrix operations used for closed form linear regression, Python using NumPy still dominates the other systems (even PostgreSQL does not support two dimensional array creation for more than 10^7 tuples). Hence, the integration of matrix calculus in database systems still has to be improved. Whereas using gradient descent, both database systems show competitive performance. Even some performance benefits originate from the used gradient descent optimiser, the results underline the possibility of analyzing data where it is stored.

In summary, the evaluation underlines the claim that the *ML2SQL* compiler makes it easy to compare different systems and that database systems show competitive performance on certain tasks.

4 DEMONSTRATION

For our demonstration scenario, we have created an interactive web interface (see Fig. 4) that allows formulating tasks in *MLearn*, compiling to the choosable target language and executing the tasks. Switching between the different target platforms (Python, HyPer, PostgreSQL) makes it possible to compare the results and the run-times of each target language. Behind the web interface,

[3] https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew/data

```
A = [[1.1,0.98,87.3,3],[0.1,3.15,42.05,3.3],
     [100.5,26.8,10.1,225.1],
     [1097.5,23000,10.1,24850.1]]
X = A[: , 0:2]
y = A[: , 3]
bias[1,len(X,0)] : 1
X = (bias::X.T).T
Xt = X.T
w = (Xt*X)^(-1) * Xt * y
print '%' , w
```

Listing 1: The specification in *MLearn.*

```
import numpy as np
def main():
    A = np.array([np.array([1.1,0.98,87.3,3]),np.array([0.1,3.15,42.05,3.3]),np.
          array([100.5,26.8,10.1,225.1]),np.array([1097.5,23000,10.1,24850.1])])
    X = DATA[ :,0:2 +1]
    y = DATA[ :,3:3 +1]
    bias = np.full( ( 1,np.size(X ,0 )), 1)
    X = (np.append(bias,X.T, axis=0)).T
    Xt = X.T
    w = np.dot( np.dot( np.linalg.matrix_power(( np.dot(Xt, X)), (-1)), Xt), y)
    print( '{}'.format(  w))
if __name__ == "__main__": main()
```

Listing 2: The translated code for Python using NumPy.

```
CREATE OR REPLACE FUNCTION ML_main() AS $$
  var A = array[array[1.1::float,0.98::float,87.3::float,3],array[
        0.1::float,3.15::float,42.05::float,3.3::float],array
       [100.5::float,26.8::float,10.1::float,225.1::float],array
       [1097.5::float,23000,10.1::float,24850.1::float]];
  var X = array_resetlower(array_slice(A,1,array_length(A,1),(0+1)
       ::int,(2+1)::int));
  var y = array_resetlower(array_slice(A,1,array_length(A,1),(3+1)
       ::int,(3+1)::int));
  var bias = array_fill(1::float, 1,array_length(X,0+1));
  X = array_transpose((array_cat(bias,array_transpose(X))));
  var Xt = array_transpose(X);
  var w = power((Xt*X), (-1)::int)*Xt*y; debug_print( '%',w);
$$ LANGUAGE 'hyperscript' strict;
select ML_main(); DROP FUNCTION ML_main();
```

Listing 3: As HyPerScript code for HyPer.

```
DO $$ declare
  A float[][]; X float[][]; Xt float[][];
  bias float[][]; w float[][]; y float[][];
begin
  A := array[array[1.1::float,0.98::float,87.3::float,3],array
       [0.1::float,3.15::float,42.05::float,3.3::float],array
       [100.5::float,26.8::float,10.1::float,225.1::float],array
       [1097.5::float,23000,10.1::float,24850.1::float]]
  X := A[:][0+1:2+1]; y := A[:][3+1:3+1];
  bias := array_fill(1::float, ARRAY[1,array_length(X,0+1)]);
  X := matrix_transpose((array_cat(bias,matrix_transpose(X))));
  Xt := matrix_transpose(X);
  w := matrix_power((Xt * X), (-1)::int ) * Xt * y;
  RAISE NOTICE '%',w;
END$$;
```

Listing 4: For PostgreSQL as PL/pgSQL procedure.

Figure 2: Closed form linear regression specified in *MLearn* and translated to Python and SQL: Lst. 1 shows the initial specification, a matrix of fixed values is created, then the optimal weights are computed by solving an equation system; Lst. 2 shows the translated code to Python using the NumPy matrix library calls. The other listings show the stored procedures in HyPerScript for HyPer (Lst. 3) and in PL/pgSQL for PostgreSQL (Lst. 4).

```
expression: INJECT | 'print' mathexplist | 'if' '(' mathexp ')' '{' explist '}' ['else' '{' explist '}'] | ('continue' | 'break')
 | 'while' '(' mathexp ')' '{' explist '}' | 'for' VARNAME ('from' mathexp 'to' mathexp | 'in' interval) '{' explist '}' | functions
 | 'create' 'tensor' VARNAME 'from' VARNAME '(' varlist ')'
 | 'save' 'tensor' VARNAME 'to' (VARNAME|STRING) [':' STRING] '(' varlist ')'
 | VARNAME '[' accessor (',' accessor)* ']' ('=' | ':') mathexp | VARNAME (',' VARNAME)* '=' VARNAME '(' [mathexp (',' mathexp)*] ')'
 | VARNAME '[' accessor (',' accessor)* ']' '~' VARNAME '(' [mathexp (',' mathexp)*] ')' | 'import' VARNAME
 | VARNAME '=' (mathexp | 'distribution' '(' VARNAME ',' VARNAME ')') | VARNAME '~' VARNAME '(' [mathexp (',' mathexp)*] ')'
 | returnType* 'function' VARNAME '(' [VARNAME (',' VARNAME)*] ')' '{' explist '}' | 'return' mathexp (',' mathexp)*
 | ('readcsv'|'writecsv') '{' (('name:' VARNAME) | ('file:' STRING) | ('columns:' varlist) | ('replace␣empty␣entries:' mathexp)
                             |('delimiter:' STRING) | ('replace:' '{' (STRING ':' STRING)+ '}') | ('delete␣empty␣entries'))+ '}'
 | 'gradientdescent' '{' (('function:' STRING) | ('data:' varlist) | ('optimize:' [nameshape (',' nameshape)*])
                        | ('learningrate:' mathexp) | ('maxsteps:' mathexp) | ('batchsize:' mathexp) | ('threshold:' mathexp))+ '}'
 | 'plot''{'(('xData:'mathexp) | ('yData:'mathexp) | ('xLabel:'STRING) | ('yLabel:'STRING) | ('type:'STRING) | ('filename:'STRING))+'}'
 | 'crossvalidate' '{' (('minfun' ':' VARNAME '=' fun) | ('kernel' ':' VARNAME ':' VARNAME ':' mathexp)
                       | ('data' ':' VARNAME (',' VARNAME)*) | ('n' ':' mathexp) | ('lossfun' ':' fun)
                       | ('folds' ':' mathexp) | ('test' '{' (VARNAME '=' interval)+ '}'))+ '}' ;
```

Listing 5: Grammar of the MLearn language: Predefined building blocks for gradient descent, cross validation, CSV file handling as well as procedural control blocks, function calls and the declaration of variables are allowed as expressions.

we run a PostgreSQL and an HyPer database server fed with an excerpt of the Chicago taxi dataset. The demonstration visitors are invited to try out the introduced types of tensor algebra as well as minimising arbitrary loss functions as, for example, linear or logistic regression.

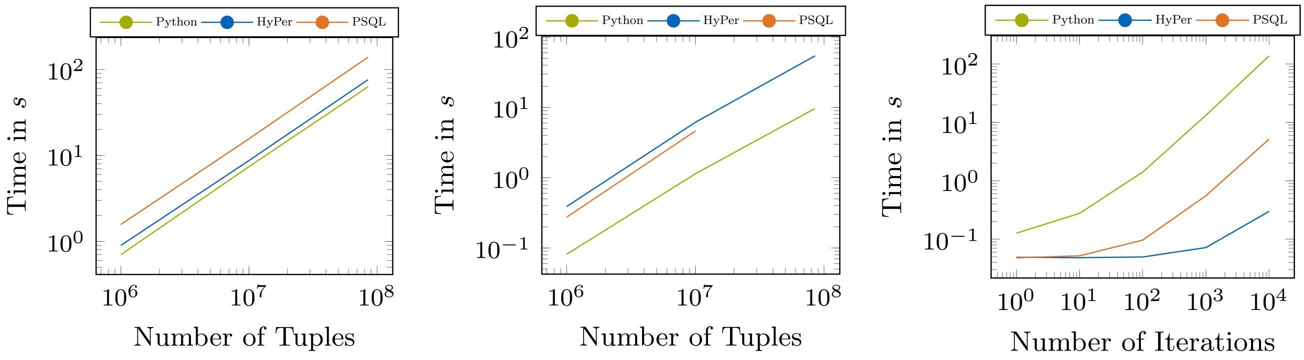
5 CONCLUSION

This demonstration presented the first declarative machine learning language *MLearn*, which allows describing machine learning tasks independently of the target engine and whose compiler allows running the code in the core of database systems. This paper first introduced comparable approaches before it presented the language specifications for performing linear regression and gradient descent using any possible loss function. Then, we evaluated the run-time of the tasks on the different target platforms PostgreSQL, HyPer and Python using NumPy and TensorFlow. The results showed, that it was indeed feasible to run the tasks as stored procedures inside of database systems showing comparable run-time especially during matrix creation.

Overall we have shown the potential of a declarative machine learning language of expressing tasks compactly and being independent of the underlying engine. As future work—to boost the capabilities of database systems for array data—remains the development of efficient array data types and the standardised integration of optimisation methods such as gradient descent inside of database systems.

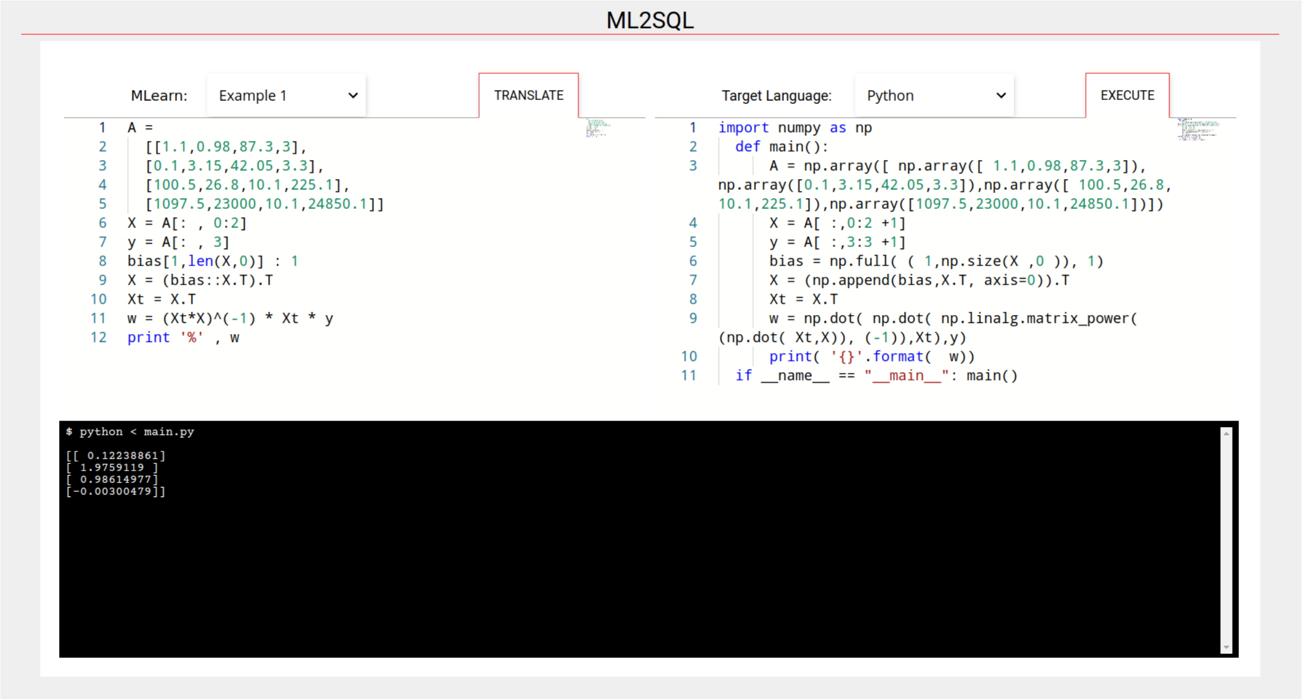
ACKNOWLEDGEMENTS

This work is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Centre Digitisation.Bavaria, an initiative of the Bavarian State Government. Dimitri Vorona has been sponsored in part by the German Federal Ministry of Education and Research



(a) Loading data from CSV. **(b) Closed form linear regression.** **(c) Linear regression with gradient descent.**

Figure 3: Run-time of (a) data loading from CSV, (b) solving linear regression using equation systems or (c) by gradient descent: For data loading and closed form linear regression, we varied the input size; for gradient descent we varied the number of iterations. PostgreSQL did not support the needed array operations for more than 10^7 tuples.



```
A =
  [[1.1,0.98,87.3,3],
  [0.1,3.15,42.05,3.3],
  [100.5,26.8,10.1,225.1],
  [1097.5,23000,10.1,24850.1]]
X = A[: , 0:2]
y = A[: , 3]
bias[1,len(X,0)] : 1
X = (bias::X.T).T
Xt = X.T
w = (Xt*X)^(-1) * Xt * y
print '%' , w
```

```
import numpy as np
  def main():
      A = np.array([ np.array([ 1.1,0.98,87.3,3]),
np.array([0.1,3.15,42.05,3.3]),np.array([ 100.5,26.8,
10.1,225.1]),np.array([1097.5,23000,10.1,24850.1])])
      X = A[ :,0:2 +1]
      y = A[ :,3:3 +1]
      bias = np.full( ( 1,np.size(X ,0 )), 1)
      X = (np.append(bias,X.T, axis=0)).T
      Xt = X.T
      w = np.dot( np.dot( np.linalg.matrix_power(
(np.dot( Xt,X)), (-1)),Xt),y)
      print( '{}'.format(  w))
  if __name__ == "__main__": main()
```

```
$ python < main.py

[[ 0.12238861]
[ 1.9759119 ]
[ 0.98614977]
[-0.00300479]]
```

Figure 4: Web interface for an interactive exploration of the *MLearn* language: Above left, the text editor allows to specify tasks, which are translated into the selected target language (above right). The code will be executed in the terminal.

(BMBF), grant TUM: 01IS12057. This research has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 725286).

REFERENCES

[1] M. Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] Z. J. Gao, S. Luo, L. L. Perez, and C. Jermaine. The BUDS language for distributed bayesian machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 961–976, 2017.

[3] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY: database-supported program execution. In *ACM SIGMOD, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 1063–1066, 2009.

[4] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206, 2011.

[5] D. Kernert, F. Köhler, and W. Lehner. Bringing linear algebra objects to life in a column-oriented in-memory database. In *Proceedings of the 1st International Workshop on In Memory Data Management and Analytics, IMDM 2013, Riva Del Garda, Italy, August 26, 2013.*, pages 37–49, 2013.

[6] M. Kim and K. S. Candan. Tensordb: In-database tensor manipulation with tensor-relational query plans. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*, pages 2039–2041, 2014.

[7] X. Li, B. Cui, Y. Chen, W. Wu, and C. Zhang. Mlog: Towards declarative in-database machine learning. *PVLDB*, 10(12):1933–1936, 2017.

[8] S. Luo, Z. J. Gao, M. N. Gubanov, L. L. Perez, and C. M. Jermaine. Scalable linear algebra on a relational database system. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 523–534, 2017.

[9] M. Schüle, F. Simonis, T. Heyenbrock, A. Kemper, S. Günneman, and T. Neumann. In-database machine learning: Gradient descent and tensor algebra for main memory database systems. In *18th symposium of "Database systems for Business, Technology and Web" (BTW), in Rostock, Germany. Proceedings*, 2019.

Incremental structural summarization of RDF graphs

François Goasdoué
Univ Rennes, Inria, CNRS, IRISA
France
fg@irisa.fr

Paweł Guzewicz
École polytechnique and Inria
France
pawel.guzewicz@inria.fr

Ioana Manolescu
Inria and École polytechnique
France
ioana.manolescu@inria.fr

ABSTRACT

Realizing the full potential of Linked Open Data sharing and reuse is currently limited by the difficulty users have when trying to understand the data modeled within an RDF graph, in order to determine whether or not it may be useful for their need.

We demonstrate our **RDFQuotient** tool, which builds compact summaries of heterogeneous RDF graphs for the purpose of first-sight visualizations. An RDFQuotient summary provides an overview of the complete structure of an RDF graph, while being typically many orders of magnitude smaller, thus can be easily grasped by new users. Our summarization algorithms are time linear in the size of the input graph and incremental: they incrementally update a summary upon addition of new data.

For the demo, we plan to show the visualizations of our summaries obtained from well-known synthetic and real data sets. Further, attendees will be able to add data to the summarized RDF graphs and visually witness the incurred changes.

1 INTRODUCTION

Semantic Web graphs are nowadays being published and shared at a massive scale, e.g., Linked Open Data (LOD) Cloud (https://lod-cloud.net) lists 1.200 graphs, while the LOD Atlas portal (http://lodatlas.lri.fr) references more than 20.000 graphs. Some of these graphs are *domain-oriented*, that is, they reflect a certain application domain, e.g., education, medical etc. On the contrary, a few RDF graphs are *encyclopedic*, e.g., DBpedia (https://wiki.dbpedia.org) and YAGO [7], covering many different topics; often, these are unions of many domain-specific ones, e.g., DBpedia is available for download as a set of domain-oriented "datasets". An overwhelming majority of the RDF graphs found in portals such as LOD Cloud or LODAtlas, https://data.gov.uk, https://data.gov etc. are domain-oriented.

Currently, a large obstacle toward exploiting this wealth of data is the difficulty for human users to make sense of a newly encountered RDF graph. The motivation for our work is to help users learn *at first sight*, without any prior knowledge about the graph and without having to set any parameter, the *(ideally complete) structure of a domain-specific RDF graph*. Given that RDF graphs can be very large, while the human information absorption capacity is relatively limited, *RDF graph summaries* have been used as intermediaries: from a given graph G a summary is extracted, then the summary is shown to the users in order to convey information about the structure and/or content of G.

We demonstrate **RDFQuotient**, a tool for constructing a complete summary of the structure of an RDF graph which does not require any user input. The particular advantage of RDFQuotient is its *tolerance to heterogeneity*, which enables it to build *compact, easy-to-visualize summaries* even from very large graphs, while preserving many of the important structural features of the graph. RDFQuotient summaries can be built efficiently, in linear time in the size of the graph. Further, they can be *incrementally updated*: upon addition of a triple t to a graph G, the summary of $G \cup \{t\}$ can be efficiently computed out of the current summary of G and t, without re-traversing the G triples.

Motivating example Figure 1 illustrates a possible visualization of an RDFQuotient summary of a BSBM [1] benchmark graph of $\mathbf{10^8}$ triples. This visualization reflects the *complete* structure of the graph, using only **5 nodes and 11 edges**, comparable to a simple small Entity-Relationship diagram. This summary reads as follows: (*i*) Non-leaf graph nodes belong to one of five disjoint classes, each represented by a summary node (boxes labeled $N1$ to $N5$ in Figure 1). The number of graph nodes in each class appears in parenthesis after the label Ni of their representative; (*ii*) Graph nodes from a class may have types. Each such type appears under the summary node label, together with its number of occurrences among graph nodes of that class, e.g., 5919 nodes represented by $N3$ are of type Producer, while 3050 are of type Vendor; (*iii*) Graph nodes from a class may have outgoing properties whose values are *leaf* nodes in the graph; the set of all such properties appears in the corresponding summary node, one property per line. For each property, e.g. country for $N3$, the summary node specifies how many graph nodes represented by this summary node have it (8969 in this case), and how many distinct leaf nodes are target of these edges (10 in this case); (*iv*) Graph nodes from a class may have outgoing properties whose values are *non-leaf* nodes in the graph. For each graph edge $n_1 \xrightarrow{a} n_2$, where n_1, n_2 are non-leaf graph nodes and a is the property (edge label), an a-labeled edge in the summary goes from the representative of n_1 to that of n_2. Next to a, this summary edge is also labeled with the number of graph edges to which it corresponds; (*v*) Properties from a small, fixed vocabulary are considered *metadata* (as opposed to *data*) and therefore they are not used to group nodes in classes, e.g., *rdf-schema#-comment* and *rdf-schema#-label* in Figure 1. More such visualizations can be found online[1]; below, we also work out an example leading from an RDF graph to its summary and then such a visualization.

We propose to *demonstrate the incremental construction of four* related (but different) *summaries*, i.e., show how summaries quickly adjust when triples are added to the summarized RDF graphs. Our summaries can be built from graphs where none, some or all nodes have one or more types; this is important because in many synthetic and real-life RDF graphs we studied, a large share of nodes is untyped [3]. Two of our summaries give preeminence to types (when available) to build the summary; nodes are first grouped by types and then by the relationships to other nodes. By contrast, the two other (including the one in Figure 1) give preeminence to node relationships; nodes are first grouped according to their relationships with others, then, each group is typed with the types of the graph nodes it represents within the summary (this is how each type has been attached to a summary node in our figure). In the total absence of types, each

[1] https://team.inria.fr/cedar/projects/rdfquotient/

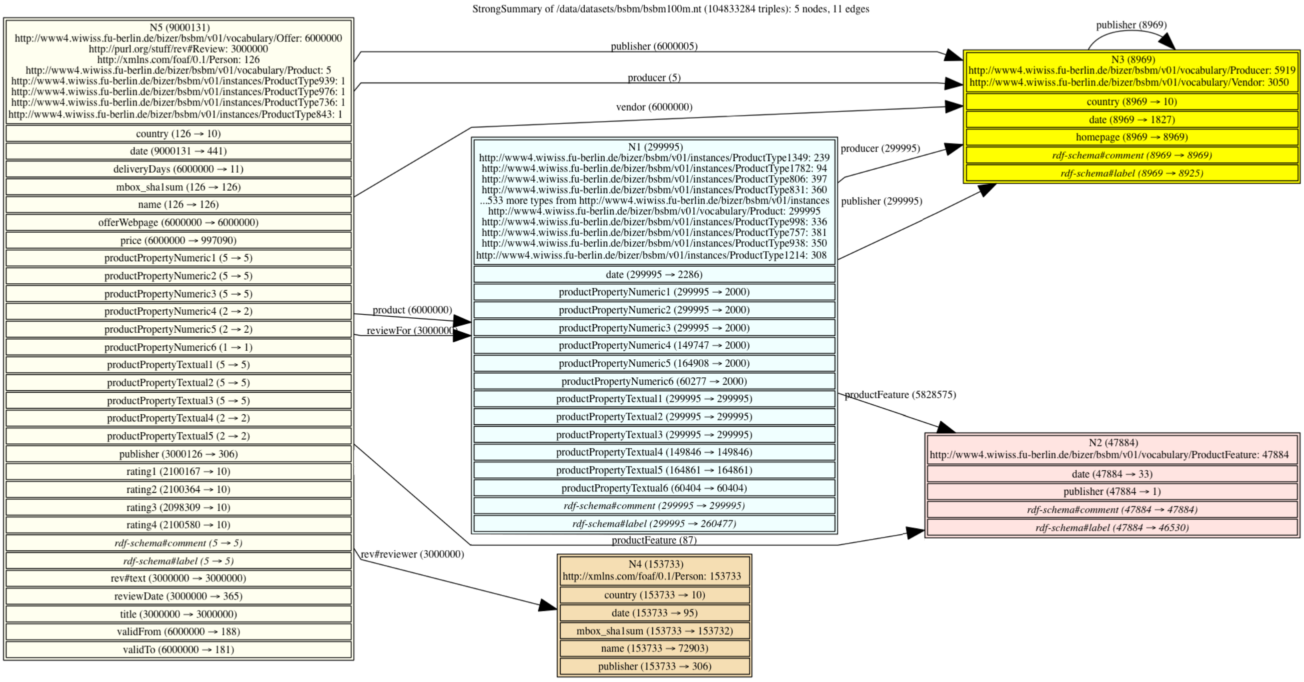

StrongSummary of /data/datasets/bsbm/bsbm100m.nt (104833284 triples): 5 nodes, 11 edges
N5 (9000131)
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/Offer: 6000000
http://purl.org/stuff/rev#Review: 3000000
http://xmlns.com/foaf/0.1/Person: 126
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/Product: 5
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType939: 1
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType976: 1
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType736: 1
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType843: 1
country (126 → 10)
date (9000131 → 441)
deliveryDays (6000000 → 11)
mbox_sha1sum (126 → 126)
name (126 → 126)
offerWebpage (6000000 → 6000000)
price (6000000 → 997090)
productPropertyNumeric1 (5 → 5)
productPropertyNumeric2 (5 → 5)
productPropertyNumeric3 (5 → 5)
productPropertyNumeric4 (2 → 2)
productPropertyNumeric5 (2 → 2)
productPropertyNumeric6 (1 → 1)
productPropertyTextual1 (5 → 5)
productPropertyTextual2 (5 → 5)
productPropertyTextual3 (5 → 5)
productPropertyTextual4 (2 → 2)
productPropertyTextual5 (2 → 2)
publisher (3000126 → 306)
rating1 (2100167 → 10)
rating2 (2100364 → 10)
rating3 (2098309 → 10)
rating4 (2100580 → 10)
rdf-schema#comment (5 → 5)
rdf-schema#label (5 → 5)
rev#text (3000000 → 3000000)
reviewDate (3000000 → 365)
title (3000000 → 3000000)
validFrom (6000000 → 188)
validTo (6000000 → 181)
publisher (6000005)
producer (5)
vendor (6000000)
product (6000000)
reviewFor (3000000)
rev#reviewer (3000000)
productFeature (87)
N1 (299995)
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType1349: 239
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType1782: 94
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType806: 397
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType831: 360
...533 more types from http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/Product: 299995
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType998: 336
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType757: 381
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType938: 350
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType1214: 308
date (299995 → 2286)
productPropertyNumeric1 (299995 → 2000)
productPropertyNumeric2 (299995 → 2000)
productPropertyNumeric3 (299995 → 2000)
productPropertyNumeric4 (149747 → 2000)
productPropertyNumeric5 (164908 → 2000)
productPropertyNumeric6 (60277 → 2000)
productPropertyTextual1 (299995 → 299995)
productPropertyTextual2 (299995 → 299995)
productPropertyTextual3 (299995 → 299995)
productPropertyTextual4 (149846 → 149846)
productPropertyTextual5 (164861 → 164861)
productPropertyTextual6 (60404 → 60404)
rdf-schema#comment (299995 → 299995)
rdf-schema#label (299995 → 260477)
producer (299995)
publisher (299995)
productFeature (5828575)
publisher (8969)
N3 (8969)
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/Producer: 5919
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/Vendor: 3050
country (8969 → 10)
date (8969 → 1827)
homepage (8969 → 8969)
rdf-schema#comment (8969 → 8969)
rdf-schema#label (8969 → 8925)
N2 (47884)
http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/ProductFeature: 47884
date (47884 → 33)
publisher (47884 → 1)
rdf-schema#comment (47884 → 47884)
rdf-schema#label (47884 → 46530)
N4 (153733)
http://xmlns.com/foaf/0.1/Person: 153733
country (153733 → 10)
date (153733 → 95)
mbox_sha1sum (153733 → 153732)
name (153733 → 72903)
publisher (153733 → 306)

Figure 1: Visualization built from an RDFQuotient summary.

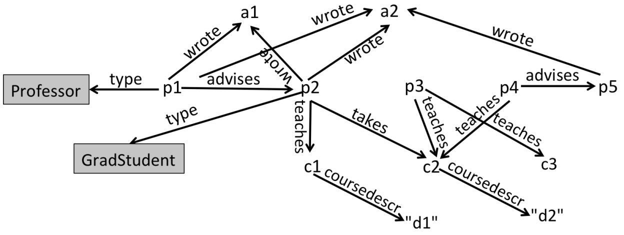


Figure 2: Sample RDF graph.

type-first summary coincides with a type-ignorant one (thus, our four summaries collapse into two).

Below, we define our summaries (Section 2) and summarization algorithms (Section 3). Then, we present the demonstration scenario based on summary visualizations (Section 5). Finally, we compare them with related work and we conclude (Section 6).

2 RDFQUOTIENT SUMMARIES

Let U be a set of URIs, L be a set of literals and B be a set of blank nodes as per the RDF specification. An RDF graph $\mathtt{G}$ is a set of *triples* of the form (s, p, o) where $s \in U \cup B \cup L$, $p \in U$ and $o \in U \cup B \cup L$. The special URI *type*, part of the RDF standard, is used to attach types to nodes. An RDF graph may contain *ontology (schema)* triples; while there are interesting interactions between summarization and ontologies [3], below we only focus on summarizing the non-schema triples, which make up the vast majority of all RDF graphs we encountered. Thus, we consider $\mathtt{G}$ consists exclusively of *type* triples and/or *data* triples (all those whose property is not *type*; we call these *data properties*).

An *RDF equivalence relation* denoted $\equiv$ is a binary relation over the nodes of an RDF graph that is reflexive, symmetric and transitive. Given an equivalence relation $\equiv$, an *RDF graph quotient* is an RDF graph having (i) one node for each equivalence class of nodes; (ii) for each edge $n_1 \xrightarrow{a} n_2$, a summary edge $n_1^{\equiv} \xrightarrow{a} n_2^{\equiv}$, where $n_i^{\equiv}$, $i \in \{1, 2\}$, is the summary node corresponding to the equivalence class of n_i, also called *representative of* n_i. The literature comprises many quotient graph summaries (see Section 6), which differ by their equivalence relations.

The equivalence relations we use are based on the concept of *property cliques*, which encodes a *transitive* relation of *edge co-occurrence on graph nodes.* Given an RDF graph $\mathtt{G}$, two data properties p_1, p_2 are in the same **source clique** iff: (i) there exists a $\mathtt{G}$ node n which is the source of p_1 and p_2 (i.e., $(n, p_1, x) \in \mathtt{G}$ and $(n, p_2, y) \in \mathtt{G}$ for some x and y), or (ii) there exists a data property p_3 such that p_3 is in the same source clique as p_1, and p_3 is in the same source clique as p_2. Symetrically, p_1 and p_2 are in the same **target clique** if there exists a G node which is the target of p_1 and p_2, or a data property p_3 which is in the same target clique as p_1 and p_2. In Figure 2, the properties *advises* and *teaches* are in the same source clique due to p_4. The same holds for *advises* and *wrote* due to p_1; consequently, *advises* and *wrote* are also in the same source clique. Further, the graduate student p_2 teaches a course and takes another, thus *teaches*, *advises*, *wrote* and *takes* are all part of the same source clique. In this example, p_1, p_2, p_3, p_4, p_5 have the source clique $SC_1 = \{advises, takes, teaches, wrote\}$, c_1, c_2, c_3 have the source clique $SC_2 = \{coursedescr\}$ and a_1, a_2 have the empty source clique $SC_3 = \emptyset$. Similarly, the target cliques are, respectively; $TC_1 = \{advises\}$ for p_2, p_5, $TC_2 = \{teaches, takes\}$ for c_1, c_2, c_3, $TC_3 = \{coursedescr\}$ for d_1, d_2, $TC_4 = \{wrote\}$ for a_1, a_2 and $TC_5 = \emptyset$ for p_1, p_3, p_4.

It is easy to see that the set of non-empty source (or target) cliques is a *partition over the data properties of an RDF graph* $\mathtt{G}$. Further, if a $\mathtt{G}$ node n is source of some data properties, they are all in the same source clique; similarly, all the properties of which n is a target are in the same target clique. Based on these cliques, for any nodes n_1, n_2 of $\mathtt{G}$, we define:

- n_1 is **weakly equivalent** to n_2, denoted $n_1 \equiv_{\mathtt{W}} n_2$, iff n_1, n_2 have the same source clique *or* the same target clique;

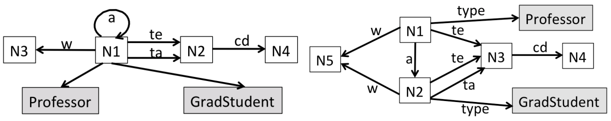


Figure 3: Weak (left) and strong (right) graph summary.

- n_1 is **strongly equivalent** to n_2, denoted $n_1 \equiv_{\mathtt{S}} n_2$, iff n_1, n_2 have the same source clique *and* the same target clique.

Further, we decide that in any RDF equivalence relation, *any class node*, i.e., a URI c appearing in a triple of the form $(n, type, c)$, is (*i*) only equivalent to itself and (*ii*) represented by itself in any RDFQuotient summary. This ensures that RDF types (classes), which (when present) denote an important information that data producers added to help understand their RDF graphs, are preserved in the summary.

The equivalence relations $\equiv_{\mathtt{W}}$ and $\equiv_{\mathtt{S}}$ lead to the **weak**, respectively **strong** summaries, defined as *quotients of* $\mathtt{G}$ *through* $\equiv_{\mathtt{W}}$, denoted $\mathtt{G}_{/\equiv_{\mathtt{W}}}$, respectively, *through* $\equiv_{\mathtt{S}}$, denoted $\mathtt{G}_{/\equiv_{\mathtt{S}}}$. Figure 3 illustrates these on the sample graph in Figure 2. For brevity, in Figure 3 and from now on, we use a, w, te, ta, cd to denote respectively the properties *advises*, *writes*, *teaches*, *takes*, and *coursedescr*.

In $\mathtt{G}_{/\equiv_{\mathtt{W}}}$, N_1 represents all the people (p_1 to p_5), N_2 represents the courses, N_3 the articles and N_4 the course descriptions. Note the self-loop from N_1 to itself; it denotes that *some* nodes represented by N_1 advise *some* nodes represented by N_1. This summary has only 4 nodes and 5 edges. It conveys the essential information that some nodes advise, write, also they teach and take something that has course descriptions. The Professor and GradStudent types of nodes p_1, respectively p_2, are attached to their common representative N_1.

$\mathtt{G}_{/\equiv_{\mathtt{S}}}$ differs from $\mathtt{G}_{/\equiv_{\mathtt{W}}}$ by representing the person nodes in two separate groups: those represented by N_1 advise those represented by N_2. This is because the target clique of p_1, p_3 and p_4 is empty, while the target clique of p_2 and p_5 is $\{advises\}$. This example illustrates the fact (visible from the summary definitions) that $\mathtt{G}_{/\equiv_{\mathtt{S}}}$ summarizes *at finer granularity* than $\mathtt{G}_{/\equiv_{\mathtt{W}}}$ (or, equivalently, $\equiv_{\mathtt{S}}$ entails $\equiv_{\mathtt{W}}$, but the opposite does not hold).

Clique-based structural summarization leads to **compact summaries** even in graphs with **heterogeneous structure**. This is because of the *transitive* aspect of the property cliques. For example, p_1 and p_3 have the same source clique, even though their property sets are disjoint: $\{a, w\}$ for p_1, $\{te\}$ for p_3; they are in the same source clique e.g. due to p_4, which has both a and te. In contrast, previously studied quotient summaries, in particular those aimed for indexing and query optimization, would not accept p_1 and p_3 as equivalent; in general, such summaries lead to more equivalence classes (summary nodes), thus also summary edges, making summaries hard to understand visually.

Type-first summarization The weak and strong summary group nodes according to their incoming/outgoing data triples and then just "carry" their types to the summary. A different choice is to group nodes first by their *set of types* (if any)[2], and use the data triples to group the nodes without types. We define:

- n_1 is **typed-weak equivalent** to n_2, noted $n_1 \equiv_{\mathtt{TW}} n_2$, iff (*i*) n_1, n_2 have the same non-empty set of types *or* (*ii*) both n_1, n_2 are untyped, and $n_1 \equiv_{\mathtt{W}} n_2$;

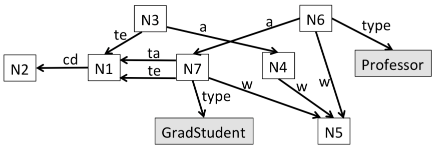


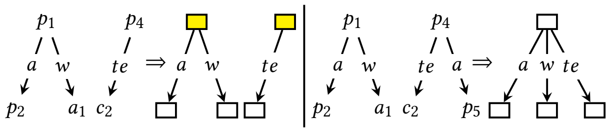
Figure 4: Typed weak graph summary.

- n_1 is **typed-strong equivalent** to n_2, noted $n_1 \equiv_{\mathtt{TS}} n_2$, iff (*i*) n_1, n_2 have the same non-empty set of types *or* (*ii*) both n_1, n_2 are untyped, and $n_1 \equiv_{\mathtt{S}} n_2$.

These relations lead to the **typed weak** ($\mathtt{G}_{/\equiv_{\mathtt{TW}}}$), respectively, to the **typed strong** ($\mathtt{G}_{/\equiv_{\mathtt{TS}}}$) RDFQuotient summaries. Figure 4 illustrates typed weak summarization on our sample graph; on this simple example, $\mathtt{G}_{/\equiv_{\mathtt{TS}}}$ is identical (in general, it may differ).

3 BUILDING RDFQUOTIENT SUMMARIES

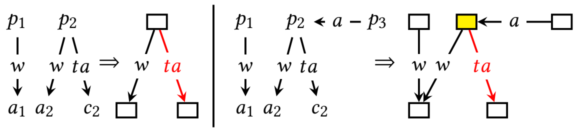
We have devised algorithms which build $\mathtt{G}_{/\equiv_{\mathtt{W}}}$, $\mathtt{G}_{/\equiv_{\mathtt{S}}}$, $\mathtt{G}_{/\equiv_{\mathtt{TW}}}$ and $\mathtt{G}_{/\equiv_{\mathtt{TS}}}$ through a single traversal of an RDF graph $\mathtt{G}$, in two flavors: (*i*) *global*: traverse $\mathtt{G}$, compute all the cliques, then traverse it again and represent nodes according to their cliques and/or types; (*ii*) *incremental*: in a single traversal of $\mathtt{G}$, *gradually build each source and target clique* based on the triples traversed *up to that point* and *simultaneously* represent $\mathtt{G}$ nodes in a summary that is continuously updated; after traversing the last triple of $\mathtt{G}$, each incremental summarization algorithm ends up with the respective summary of the full $\mathtt{G}$. The algorithms are detailed and their correctness is proved in [3]; below, we illustrate their interesting points on minimal examples.



First, let us see on an example how $\equiv_{\mathtt{W}}$ can grow during *incremental weak summarization*. Suppose the graph $\mathtt{G}$ in Figure 2 is traversed and summarized starting with: (p_1 *advises* p_2), then (p_1 *wrote* a_1), then (p_4 *teaches* c_2) (see the figure above). When we summarize this third triple, we do not know yet that p_1 is equivalent to p_4, because no common source of *teaches* and *advises* (e.g., p_3 or p_4) has been seen so far. Thus, p_4 is so far not equivalent to any other node, and represented separately from p_1. Now, assume the fourth triple traversed is (p_4 *advises* p_5): at this point, we know that *advises*, *wrote* and *teaches* are in the same source clique, thus $p_1 \equiv_{\mathtt{W}} p_4$, and their representatives (highlighted in yellow) must be **fused** in the summary. More generally, it can be shown that $\equiv_{\mathtt{W}}$ **only grows** as more triples are visited (i.e., is monotonic), in other words: if in a subset $\mathtt{G}'$ of $\mathtt{G}$'s triples, two nodes n_1, n_2 are weakly equivalent, then this holds in any $\mathtt{G}''$ with $\mathtt{G}' \subseteq \mathtt{G}'' \subseteq \mathtt{G}$.

Incremental strong summarization is even more complex because unlike $\equiv_{\mathtt{W}}$, $\equiv_{\mathtt{S}}$ **may grow and shrink** during summarization (i.e., is non-monotonic). For instance, assume the summarization of the graph in Figure 2 starts with (p_1 *wrote* a_1), (p_2 *wrote* a_2), (p_2 *takes* c_2) (see the figure below). After these, we know $p_1 \equiv_{\mathtt{S}} p_2$; their source clique is $\{wrote, takes\}$ and their target clique is $\emptyset$. Assume the next triple traversed is (p_3 *advises* p_2): at this point, p_1 is *not* $\equiv_{\mathtt{S}}$ *to* p_2 *any more*, because p_2's target clique is now $\{advises\}$ instead of $\emptyset$. Thus, p_2 **splits** from p_1, that is, it needs to be represented by a new summary node (shown in yellow below), distinct from the representative of p_1.

[2] We use *set of types* and not just "type" on purpose, because an RDF node may have more than one type. If we classified a node according to *each* of its types, as in e.g. [2], a node with many types would have more than one representative, which is incompatible with quotient summarization.



Further, note that the representative of p_1 and p_2 (at left above) had one *takes* edge (highlighted in red) which was *solely due to p_2's outgoing takes edge*. By definition of a quotient summary, that edge *moves* from the old to the new representative of p_2 (the yellow node). If, above at left, p_1 had also had an outgoing edge labeled *takes*, at right, both nodes in the top row would have had an outgoing *takes* edge. It can be shown that *splits only occur in such cases*, i.e., a node whose target clique becomes non-empty (respectively whose source clique becomes non-empty) and the node was previously represented together with other nodes; if it was represented alone, the respective clique of its representative is just updated.

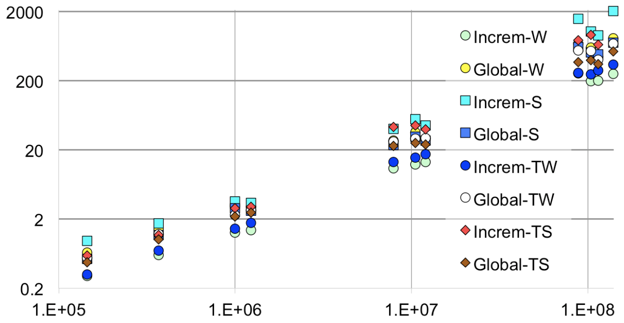


Figure 5: Summarization time (s) vs. graph sizes |G|.

The **amortized complexity** of our summarization algorithms is linear in the number of triples of G. Figure 5 illustrate this empirically on a variety of benchmark (LUBM and BSBM) and real-life (DBLP, Springer conference etc.) datasets ranging from a few hundred thousands to more than 100 million triples; note that both axes are in log-scale. The implementation is made in Java 1.8; RDF graphs are stored in Postgres 9.6 and traversed from there. Increm-W is the fastest overall; it traverses G only once, thus it is faster than global-W which performs one extra pass to compute the cliques. S, TW and TS, in this order, are more expensive, and finally incremental S, which pays an extra performance overhead for growing and shrinking the equivalence relation.

4 FROM SUMMARIES TO VIZUALIZATIONS

The core of our work is on defining and efficiently building summaries; here we present one possible way of rendering them through a vizualization like the one illustrated in Figure 1.

On our four summaries we apply **leaf and type inlining**, as follows. We remove type edges; instead, each type attached to a node in the summary is shown in the box corresponding to the node, after the node ID. Similarly, for each edge $n \xrightarrow{a} m$ where m is a leaf, we include a as an "attribute" of n, and do not render m (we say it has been "inlined" within n). A sizable part of an RDF graph's nodes are leaves; as we will show, inlining them into their parent nodes greatly simplifies the visualization.

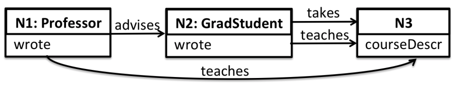


Figure 6: Leaf and type inlining on the sample strong summary from Figure 3 (right).

Figure 6 illustrates inlining for the S summary of our sample graph. This summary is extremely compact, yet rich with information; professors, students, and courses are visible at a glance. Articles have been inlined within their authors as they were leaves in $G_{/\equiv_{TS}}$ (Figure 3). This simplification can also be seen as a small loss of information: Figure 6 does not immediately suggest that Professors may have written articles together with GradStudents. However, (*i*) only leaf nodes are folded and (*ii*) after a first glance, users may pursue exploration by other means (e.g., queries to check for such joint articles).

5 DEMONSTRATION SCENARIO

Demonstration attendees will be able to pick an RDF graph from a list of well-known synthetic and real data sets, visualize their summaries, and compare with other close competitor summaries, such as those mentioned in Section 6; some of these vizualizations can be seen online (https://team.inria.fr/cedar/projects/rdfquotient/). Attendees will also be able to add new triples to an RDF graph, to figure out through our summary visualization how changes in the original data are rapidly reflected into the summary thanks to incremental summarization. To make it entertaining, we plan to use RDF data on the conference attendees, from DBLP, other public sources, and made-up triples to get interesting summary changes.

6 RELATED WORK & CONCLUSION

The literature comprises many RDF summarization techniques, more than a hundred of which we covered in a recent co-authored survey [4]. RDFQuotient summarizes *the structure of the data triples*, which form a vast majority of RDF graphs; complementary proposals summarize the ontology, the values, find the most frequent property groups etc. Closest to us are quotient summaries which group nodes by *the set of their outgoing data properties* ("characteristic sets" [6]) and possibly also by *by the set of their incoming data properties* (forward and backward bisimulation [5]). Our clique-based summarization differs from these by the *transitive* aspect of the cliques which leads to *heterogeneity-tolerant summarization*. Indeed, as we plan to show during our demonstrations, more strict summaries such as [5, 6] support query optimization and indexing well, but have too many nodes and edges, even after inlining, for a comfortable vizualization. In exchange, our summaries are not generally appropriate for indexing, as they do not give e.g. access to "all the resources having properties a and b": the graph nodes whose source clique is $\{a, b\}$ may have one or another or both.

REFERENCES

[1] Christian Bizer and Andreas Schultz. 2009. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.* 5, 2 (2009).

[2] Stéphane Campinas, Renaud Delbru, and Giovanni Tummarello. 2013. Efficiency and precision trade-offs in graph summary algorithms. In *IDEAS*.

[3] Šejla Čebirić, François Goasdoué, Paweł Guzewicz, and Ioana Manolescu. 2017. *Compact Summaries of Rich Heterogeneous Graphs*. Research Report RR-8920. INRIA Saclay ; Université Rennes 1. https://hal.inria.fr/hal-01325900

[4] Sejla Cebiric, François Goasdoué, Haridimos Kondylakis, Dimitris Kotzinos, Ioana Manolescu, Georgia Troullinou, and Mussab Zneika. 2018. Summarizing Semantic Graphs: A Survey. *The VLDB Journal* (2018). https://hal.inria.fr/hal-01925496

[5] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. 2002. Covering indexes for branching path queries. In *SIGMOD*.

[6] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*.

[7] Thomas Rebele, Fabian M. Suchanek, Johannes Hoffart, Joanna Biega, Erdal Kuzey, and Gerhard Weikum. 2016. YAGO: A Multilingual Knowledge Base from Wikipedia, Wordnet, and Geonames. In *ISWC*.

VISTA: A visual analytics platform for semantic annotation of trajectories

Amílcar Soares
Institute for Big Data Analytics
Halifax, Canada
amilcar.soares@dal.ca

Jordan Rose
Institute for Big Data Analytics
Halifax, Canada
jrose@dal.ca

Mohammad Etemad
Institute for Big Data Analytics
Halifax, Canada
etemad@dal.ca

Chiara Renso
ISTI-CNR
Pisa, Italy
chiara.renso@isti.cnr.it

Stan Matwin
Institute for Big Data Analytics
Halifax, Canada
stan@dal.ca

ABSTRACT

Most of the trajectory datasets only record the spatio-temporal position of the moving object, thus lacking semantics and this is due to the fact that this information mainly depends on the domain expert labeling, a time-consuming and complex process. This paper is a contribution in facilitating and supporting the manual annotation of trajectory data thanks to a visual-analytics-based platform named VISTA. VISTA is designed to assist the user in the trajectory annotation process in a multi-role user environment. A session manager creates a tagging session selecting the trajectory data and the semantic contextual information. The VISTA platform also supports the creation of several features that will assist the tagging users in identifying the trajectory segments that will be annotated. A distinctive feature of VISTA is the visual analytics functionalities that support the users in exploring and processing the trajectory data, the associated features and the semantic information for a proper comprehension of how to properly label trajectories.

1 INTRODUCTION

The increasing access to positioning devices technologies such as smart-phones, GPS-enabled cameras and sensors, resulted in vast volumes of mobility data collected, stored and available for analysis. Such mobility data are typically modeled as streams of spatio-temporal points, called trajectories. There is a growing research interest in analysis methods for semantically enriched trajectories [7, 8]. However, only a few datasets containing semantically labeled trajectory data are available. This is primarily because the semantic labels tend to indicate a specific behavior whose identification depends on the humans' interpretation: in fact, the understanding of the moving object behavior depends on multiple factors and, in most of the cases, it cannot be automatically inferred. The process of manual annotation may be, therefore, complex and time-consuming, even by domain experts who might be uncertain in the correct data interpretation or they might disagree. Indeed, the labeling process is complex due to a number of factors: (1) the annotator user needs to have an immediate view and understanding, not only of the spatio-temporal trajectory details and the numerical features deriving from them but also of the contextual semantic information; (2) different annotators might have different roles and interpretations on how to label trajectories (e.g., "is the ship fishing or not?"); (3) not only the whole trajectory might express a single behavior, very often the parts of a trajectory have to be individually labeled since different behavior may co-exist during the same journey (e.g., a vessel is first fishing then traveling to harbor). How to correctly identify the switch points from one behavior to the other is another critical issue to be solved. We believe that these issues might be alleviated by a proper trajectory annotation platform that effectively assists the user in coping with all these aspects during the semantic labeling process.

A promising approach is to build a platform supporting the annotation by exploiting concepts from Visual Analytics. The idea of Visual Analytics is to create systems that enable analytical reasoning about complex problems with the goal of making the data processing and the inferred information clear and evident for the analysis [5]. This is accomplished by integrating data processing methods with visualizations designed to assist users in making efficient decisions. We can observe that the human mind generally conveys information more effectively through visualization than only relying on textual and numerical data. Combining visualization with user interactions enables a system to explore the data from different perspectives, thus linking and combining distinct information pieces to derive new insights from the data.

Pioneering works on semantic trajectories annotations [6, 10] used a predefined set of rules established by domain experts to automatically assign the semantic labels to the whole trajectories or to segments of trajectories. Although these methods work well for many scenarios, they are not suitable in case there are no clear criteria for identifying the object behavior and/or the segment to be labeled. In this case, we need the human intervention to establish the most appropriate label for each trajectory (or trajectory segment) based on both numerical and semantic features. Recently, some works proposed methods to simplify the manual annotation task like in [9] where a web interface has been proposed to upload personal trajectories and annotate each segment with the activity performed by the user. However, in this case, all trajectory segments need to be annotated directly by the traveling user, and this becomes unfeasible when the number of trajectory segments is very high and/or the annotator does not represent the entity which performed the movement like in the case of vehicles, vessels or animals. To cope with this problem, other methods propose machine learning approaches (e.g. semi-supervised or active learning) to automatically classify trajectories into semantic labels by starting from a small set of manually annotated traces [3, 4]. All these approaches must rely on an accurately labeled dataset to reach a good classification accuracy. Therefore, there is a strong need for supporting the

Series ISSN: 2367-2005 10.5441/002/edbt.2019.58

manual annotation process which can lead to good quality annotated datasets and therefore reliable analysis findings. To the best of our knowledge, no approach combines visual analytics with trajectory processing functions to assist the user in the process of manually tagging trajectory data.

The challenge of a visual analytics trajectory annotation system is to provide efficient and effective support for the user interaction, helping the user in focussing on each specific annotation task, highlighting the features values for each trajectory point or segment and properly visually combine contextual knowledge. Specifically, the annotators need to use their domain knowledge for thinking, creating associations, and generating insights from the trajectory data. On the other hand, the system also needs capabilities for processing and aggregating trajectory data. Deciding where to set the segmentation point in a trajectory is very challenging because it will directly affect the values of the segments' features and therefore the overall labeling of the trajectory dataset. We propose VISTA as an interactive visual user interface integrating spatio-temporal processing capabilities to play an essential role in semantic trajectory annotation. With VISTA, we provide full support to the manual trajectory annotation by tailoring a visual analytics platform that guides the user in this process. We strongly believe that VISTA can significantly contribute to the scientific community in supporting the domain experts in the annotation process and produce more reliable semantically labeled trajectory datasets available for analysis.

2 SYSTEM ARCHITECTURE

The architecture of VISTA has been designed to provide solutions to the three issues introduced in the previous section, namely the immediate understanding, the different users' role and the support for the segment switch points identification.

A distinctive feature of VISTA is the possibility to handle a multi-user annotation process where the users might have different roles. VISTA supports both the creation of an annotation session (e.g., upload trajectories, contextual information, feature creation, etc) by the *session manager* and the annotation process itself by the *annotator* (e.g., user analyses a trajectory, provide partitioning positions, compare the segments, annotates a trajectory, etc).

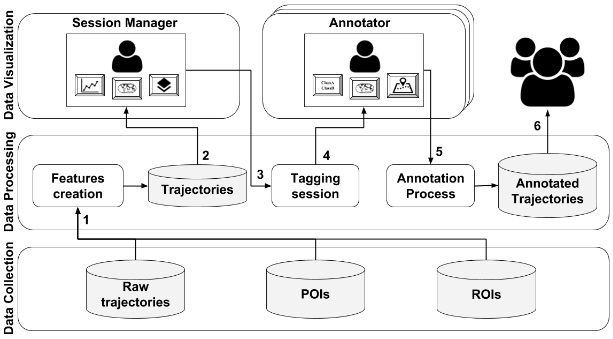


Figure 1: VISTA architecture and workflow.

The architecture and workflow overview is illustrated in Figure 1. As we can see, it is organized into three main components: (i) the data collection to handle raw trajectory and contextual geographical information like Point of Interests (POI) and Region Of Interests (ROI), (ii) the Data Processing that deal with trajectory and annotation data, and (iii) the Data Visualization to interact with the users in the processes of both creating a tagging session, detect the switch points and annotating trajectories. The workflow of VISTA to perform trajectory annotation includes six steps depicted as the numbers of the arrows of Figure 1. In the first step, the session manager is requested to set the stage for the annotation process, namely upload raw trajectories and the POIs and ROIs that are relevant to the studied domain. In the following step (Step 2), the data processing engine automatically creates numerical features related to each trajectory point, called *point features* like the distance traveled, the estimated speed, the bearing, the bearing rate, and the acceleration. The engine also calculates the shortest distances to a POIs, and detects when the trajectory points intersect with ROIs. The shapely library[1] was used to calculate the shortest distance and intersections. At this point, each trajectory uploaded by the user is stored as a document in a MongoDB[2] collection. In step 3, VISTA displays the trajectories associated to these new features to the session manager whose tasks are: (1) the data exploration for the selection of the features that are relevant for the tagging session, (2) the creation of the annotation classes (i.e. labels) that must be used in a tagging session, and (3) the invitation to the annotator users to participate in a tagging session. With a tagging session created, it is now possible for the invited annotators to start the tagging process of trajectory data. In step 4, the annotators explore the trajectory data and create the trajectory segments that must reflect the annotation classes available for tagging. Several visualization functionalities are available for the process of tagging and are detailed in Section 3. After going through all trajectories and tagging their segments with the labels (Step 5), a dataset with annotated trajectories will be available for all the users to download (Step 6).

As we discussed above, a crucial step in annotating trajectory data is to determine the parts of the trajectory where the moving object's behavior changes. Detecting such switches in the object movement behavior is challenging for an annotating user since there is a need to explore a possibly high number of features to precisely determine where and when the object behavior changed. This is done through a process usually called *segmentation*. Segmenting a trajectory means to find the partitioning points that are used to create trajectory segments, or sub-trajectories, characterized by the fact that each segment has uniform behavior respect to some criteria [2]. Once these segments have been identified, additional numerical features like average, median, standard deviations, and percentiles may be created to better characterize the behavior of the moving object in that segment (the so-called *segment features*).

3 DEMONSTRATION

The objective of our demonstration is to involve the user in the VISTA tagging experience by providing a trajectory dataset to be annotated together with semantic contextual information. We have selected two datasets: (1) 10 trajectories of vessels that should be annotated as "fishing" and "not "fishing" activity; (2) 20 trajectories of people that should be annotated with transportation means as "walking", "bike", "train", "bus", and "car". During the demo the authors will guide attendees through the whole process where the user will experience both roles, the *session manager* and the *annotator*, whose tasks and relative interfaces are detailed in the next sections.

[1] https://pypi.org/project/Shapely/

[2] https://www.mongodb.com/, version 3.6.2

3.1 Session Manager

The role of this user is setting the stage for the actual annotation process. This is done through four screens. In the first screen, the user is requested to create and give a name to a new tagging session or select a session that was previously created. In the second screen, the user is requested to upload raw trajectories in delimited separated file format (e.g., CSV, TSV, etc) and map the file fields to the columns representing the raw trajectory data: *trajectory_id, time, latitude, longitude*. In the third screen, the user is asked to upload POIs and/or ROIs that are relevant to the domain. Then, VISTA executes a process in the background to create the following point features for each raw trajectory point: time difference from the previous point, distance traveled from the previous point, speed, acceleration, bearing, jerk, bearing rate, and rate of bearing rate. The relative computation formulas and details can be found in [1]. The platform also will create other two point features using POIs and ROIs: (1) for every POIs layer, the platform will calculate the shortest distance to a POI on the particular layer; (2) for ROIs layers, the platform will verify if a trajectory point intersects a ROI in the specific layer. Finally, in the fourth screen, the user will create the labels that must be used by annotators and invite annotators in the trajectory tagging session by providing their emails. This sequence of actions represents steps 1, 2 and 3 of Figure 1. After this step, the annotators will receive a notification with the invitation to enter the tagging session.

3.2 Annotator

When a user receives an invitation for a tagging session, he/she can start to tag trajectories. The process of annotating trajectories is iterative and interactive, where a single trajectory is presented to the user and further explored in each iteration with the system. We recall that the two-fold objective of a tagging session is (1) to identify the segments of the trajectory with a uniform behavior, or, in other words, identify the change points and (2) actually tag the segment with the appropriate label. In VISTA, the annotator has access a dashboard providing tools to understand the behavior of the moving object, depicted in Figure 2. We observe that the main screen is divided into two interactive panels: (i) a map on the left and (ii) summary statistics on the right. The map panel visualization needs to be effective in showing the actual movement of the trajectory with point and trajectory features. For this reason, we implemented two visualization solutions to support the annotator in understanding the movement data: first, the actual moving object movement can be played, dynamically showing how the moving object performed its movement in a particular region. Second, the line colors are displayed in saturation grades of red, reflecting the value of the point feature selected on the right side of the screen; a low value is colored with a light intensity of red, and higher values are colored with a more intense red. There is an automatic interaction between the two panels: the user can select a segment and/or point features in the right panel and the relative parts in the map on the left are highlighted. Conversely, the annotator can select a trajectory point in the map, and the corresponding segment and point feature are highlighted in the right panel. On the bottom of the map, the red color legend is presented to the annotator user to have an immediate perception of what is happening between the points sequence. Inside the map and in the top left, VISTA provides some typical Geographical Information System (GIS) visualization options, such as zoom-in and zoom-out, display or hide POIs and ROIs, and change the colors of the annotation classes (e.g., fishing or not-fishing).

In more detail, the right panel provides the following options and statistics: (i) On the top, it displays a summary computed from the trajectory data with its total number of points, total distance travelled in meters and the average sampling rate between the trajectory points; (ii) at the bottom, the user may choose to visualize the data as a line chart that shows the values of the point features following the temporal order or a scatter plot where the user can try to find correlations between two point features.

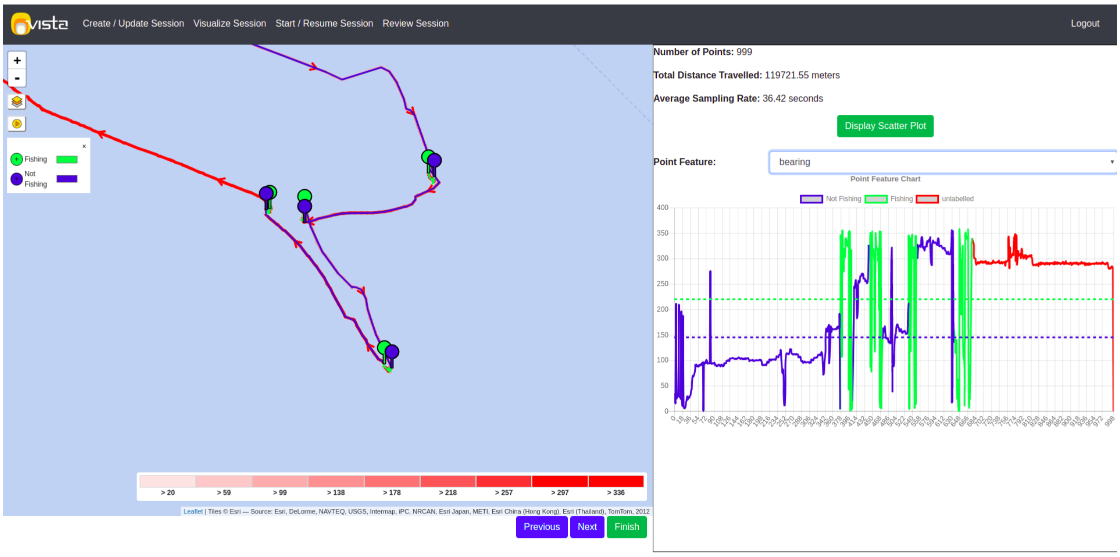
When a trajectory is shown on the screen, the annotator user is requested to provide the partitioning positions to split the trajectory into the segments representing uniform behavior to be assigned to the class labels. This is the most challenging functionality to develop since the correct segmentation is fundamental for a good quality annotation process. However, identifying the switch points is particularly difficult since many different aspects have to be considered at the same time. It is, therefore, challenging to support the user in this "multi-dimensional view" where the most crucial aspects have to be considered jointly. For this reason, we have created a drag and drop tool on the map where the user can add a new partitioning point to the screen, assign to a class label and pin exactly the switch point where to split the trajectory. By adding a new partitioning position, all the statistics regarding the classes of the trajectory have to be automatically recomputed since new labeled segments are provided. This is captured by the two panels described above. First, on the map panel, the colors of the trajectory segments change according to the class provided. Second, the colors of both the scatter plots and the line chart change to reflect the new information provided by the user. We also provide statistical measures regarding the trajectory segments and their classes. Trajectories are sent to user in a sequential way, one per interaction, which the user's objective is to segment it using the drag and drop pins provided in the left side of the panel. When a trajectory is completely annotated, the user can press the Next button on the bottom of the screen to receive a new trajectory to be annotated.

3.3 Summarizing annotations from users

When all the users conclude the annotation process, a summarizing screen (Figure 3) is created with the objective of exploring how the different users tagged the data. In VISTA, the users can confront their tagging with other annotators by exploring how they annotated their dataset and how the results of their tagging session are similar or not to the other users. The tagging session results can be compared for both point and segment features. Figure 3 shows the main screen with two panels: (1) on the left, the user is able to analyze statistics (e.g. minimum, maximum, average, standard deviation, and percentiles) of all point features available on the platform; (2) on the right, the user may want to analyze statistics regarding the segment features. For both charts, the average behavior per user and class are plotted with the objective of understanding if the users most likely agree or disagrees regarding some feature.

4 CONCLUSIONS AND FUTURE WORKS

We have proposed VISTA, an interactive tool based on visual analytics principles, supporting the users in semantically annotate trajectory data. A distinctive feature of VISTA is the support for the identification of trajectory segments and the assignment to the relative semantic label. We intend to expand this



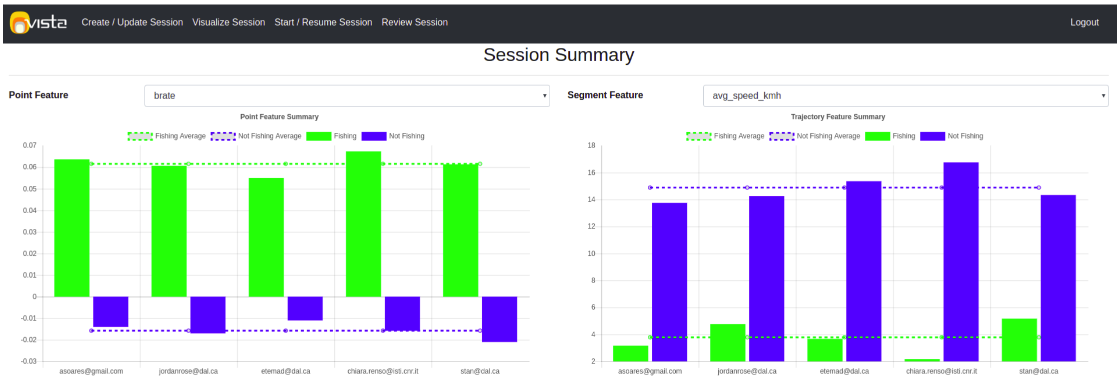
Number of Points: 999

Total Distance Travelled: 119721.55 meters

Average Sampling Rate: 36.42 seconds

Point Feature: bearing

Figure 2: Elements of the annotator user dashboard with the vessels trajectories dataset.



Point Feature brate

Segment Feature avg_speed_kmh

Figure 3: Summary results with the users' annotations.

platform into two directions. First, we want to create a module that automatically suggests how to segment the trajectory by learning from the previous interactions with the platform. Second, we intend to improve the results comparing how the labels have been assigned by the different users, highlighting when users agree or disagrees in identifying a specific behavior of a moving object. The tool is available for testing at the URL https://bigdata.cs.dal.ca/resources.

ACKNOWLEDGEMENTS

This work has been supported by the MASTER project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Slodowska-Curie grant agreement N. 777695. The authors would also like to thank NSERC (Natural Sciences and Engineering Research Council of Canada) for financial support.

REFERENCES

[1] Mohammad Etemad, Amílcar Soares Júnior, and Stan Matwin. 2018. Predicting Transportation Modes of GPS Trajectories using Feature Engineering and Noise Removal. In *31st Canadian Conference on Artificial Intelligence, Canadian AI 2018, Toronto, Canada, May 8–11*. Springer, 259–264.

[2] Amílcar Soares Júnior, Bruno Neiva Moreno, Valéria Cesário Times, Stan Matwin, and Lucídio dos Anjos Formiga Cabral. 2015. GRASP-UTS: an algorithm for unsupervised trajectory segmentation. *International Journal of Geographical Information Science* 29, 1 (2015), 46–68.

[3] Amílcar Soares Júnior, Chiara Renso, and Stan Matwin. 2017. ANALYTiC: An Active Learning System for Trajectory Classification. *IEEE computer graphics and applications* 37, 5 (2017), 28–39.

[4] Amilcar Soares Junior, Valeria Cesario Times, Chiara Renso, Stan Matwin, and Lucidio A. F. Cabral. 2018. A Semi-Supervised Approach for the Semantic Segmentation of Trajectories. In *2018 19th IEEE International Conference on Mobile Data Management (MDM)*. 145–154.

[5] Daniel Keim, Gennady Andrienko, Jean-Daniel Fekete, Carsten Görg, Jörn Kohlhammer, and Guy Melançon. 2008. Visual analytics: Definition, process, and challenges. In *Information visualization*. Springer, 154–175.

[6] Christine Parent, Stefano Spaccapietra, Chiara Renso, Gennady Andrienko, Natalia Andrienko, Vania Bogorny, Maria Luisa Damiani, Aris Gkoulalas-Divanis, Jose Macedo, Nikos Pelekis, et al. 2013. Semantic trajectories modeling and analysis. *ACM Computing Surveys (CSUR)* 45, 4 (2013), 42.

[7] Christine Parent, Stefano Spaccapietra, Chiara Renso, Gennady L. Andrienko, Natalia V. Andrienko, Vania Bogorny, Maria Luisa Damiani, Aris Gkoulalas-Divanis, José Antônio Fernandes de Macêdo, Nikos Pelekis, Yannis Theodoridis, and Zhixian Yan. 2013. Semantic trajectories modeling and analysis. *ACM Comput. Surv.* 45, 4 (2013), 42:1–42:32. https://doi.org/10.1145/2501654.2501656

[8] Chiara Renso, Stefano Spaccapietra, and Esteban Zimanyi (Eds.). 2013. *Mobility Data: Modeling, Management, and Understanding*. Cambridge Press.

[9] Salvatore Rinzivillo, Fernando de Lucca Siqueira, Lorenzo Gabrielli, Chiara Renso, and Vania Bogorny. 2013. Where Have You Been Today? Annotating Trajectories with DayTag. In *Advances in Spatial and Temporal Databases - 13th International Symposium, SSTD 2013, Munich, Germany, August 21-23, 2013. Proceedings*. 467–471. https://doi.org/10.1007/978-3-642-40235-7_30

[10] Zhixian Yan, Dipanjan Chakraborty, Christine Parent, Stefano Spaccapietra, and Karl Aberer. 2011. SeMiTri: a framework for semantic annotation of heterogeneous trajectories. In *Proceedings of EDBT*. ACM, 259–270.

SLIPO: Large-Scale Data Integration for Points of Interest

Spiros Athanasiou, Michalis Alexakis, Giorgos Giannopoulos, Nikos Karagiannakis,
Yannis Kouvaras, Pantelis Mitropoulos, Kostas Patroumpas, Dimitrios Skoutas
Information Management Systems Institute, Athena Research Center, Greece
{spathan,alexakis,giann,nkaragiannakis,jkouvar,pmitropoulos,kpatro,dskoutas}@imis.athena-innovation.gr

ABSTRACT

Points of Interest (POIs) are indispensable to many modern applications, services, and products. From navigation applications, to social networks, tourism, or logistics, we use POIs to search, communicate, decide, and plan our actions. In this demonstration, we showcase SLIPO, a system prototype that addresses the limitations, gaps and challenges in integrating, enriching, and sharing POI data. Leveraging the Linked Data paradigm to effectively extract the most out of open, crowdsourced or proprietary real-world data sources, SLIPO tackles their inherent spatial, temporal, or thematic ambiguities in POI data. Hiding all Linked Data complexities in the background, SLIPO orchestrates state-of-the-art software customized for POI data integration, enabling stakeholders to increase the value of their data and relieving them from labor-intensive, manual, error-prone, and costly updates.

1 INTRODUCTION

Points of Interest (POIs) refer to physical locations of some particular interest or utility, such as restaurants, shops, hotels, sport venues, etc. They are useful in our everyday lives (e.g., navigation, social networks, tourism) as well as in various commercial domains (such as logistics, advertising, or geomarketing). A POI is minimally characterized by its *name*, a *category*, and a *location*; however, POI profiles may often be quite complex, containing composite, multi-faceted and multi-modal information. This complexity may concern extra *thematic attributes* (address, contact details, opening hours, etc.) or their *relationship* to other entities (e.g., a shop within a mall).

Integrating POI data from multiple sources to create quality-assured, enriched, updated datasets is challenging. The advent of *open* data, *crowdsourcing*, and *social media* has provided new data sources of even greater volume, heterogeneity, diversity, veracity, and timeliness. Any consistent approach towards *POI data integration* also needs to rely on robust, flexible and semantically-rich modelling of POI profiles and handling of POI identifiers, especially when dealing with cross-sector, cross-border, and cross-lingual content. The greater the *size*, *timeliness*, *richness*, and *accuracy* of POI data, the better the end product's *value*.

Motivated by the highly fragmented landscape on POI data integration, curation and update of missing, out-of-date, or inaccurate information, we propose a pragmatic, yet highly effective approach. In the context of the SLIPO project[1], while maintaining interoperability with de facto POI standards, we opt to apply de jure *Linked Data* standards (RDF[2], OWL[3], GeoSPARQL[4]) for the inner workings of data integration, also offering capabilities to harness open POI data sources (e.g., OpenStreetMap). Linked Data technologies are ideal for handling the inherent geospatial, thematic, and semantic ambiguities of POIs. To this goal, we have built an open-source prototype system with a complete suite of software tools and services to orchestrate iterative POI *integration workflows* over multiple POI datasets, across all stages of the POI data lifecycle (transformation, linking, fusion, enrichment). Stakeholders should not adapt their current processes to collect, update, or roll-out POIs across services and products, since SLIPO hides all Linked Data complexities, and allows them to focus on their task: increase the value of their data.

The paper is structured as follows. In Section 2 we discuss the challenges in POI data integration. Section 3 overviews the data integration lifecycle as applied in SLIPO. Section 4 outlines the current status of our prototype. Finally, Section 5 showcases how a typical POI integration scenario can be handled in SLIPO.

2 ISSUES IN POI DATA INTEGRATION

POI data are by nature *semantically diverse* and *spatiotemporally evolving*, representing different entities depending on their geographical, temporal, and thematic context. Due to their use in various domains and contexts, POI-related information is typically found in diverse, heterogeneous sources. Assembling such pieces of information together is seriously hindered by the lack of common POI identifiers and data sharing formats. In addition, stakeholders also have to cope with volatile data in a POI profile, e.g., its facilities, opening hours, prices, events, etc.

Since integrating POI data with current approaches remains labor-intensive and does not scale, most stakeholders restrict their focus on domain-specific or small-sized datasets. But at a larger scale, all this complex process raises several cases of *ambiguity* that may severely hinder data integration of POIs. Addresses, coordinates, and place names are equally used throughout applications as pseudo-identifiers; but practice shows that they fail to effectively disambiguate POIs. Next, we outline these challenging issues in POI data integration.

i) Geospatial ambiguity:

- *Same POI, differing coordinates.* Locations of the same POI among different datasets almost never match exactly due to varying data collection procedures (e.g., field work, map digitization, GPS readings, crowdsourced markers).
- *Same POI, different shapes.* Although POIs are usually abstracted as point locations, they usually have a *shape* with a spatial extent (e.g., a building). But such detailed geometries are only an approximation, and their accuracy may vary significantly.
- *Different POIs, same location.* Multiple POIs may be co-located within a larger structure (e.g., multi-storey building) or a facility (e.g., shops in a mall). If abstracted as *points*, those distinct entities end up superimposed at the same location.
- *POI within another POI.* If POIs are represented by detailed *shapes* (e.g., polygons), they may exhibit topological (e.g., containment) relations. Sometimes, it is not clear whether a certain shape is a separate entity or merely a *part of* the larger one.

[1] Acronym for **S**calable **L**inking and **I**ntegration of big **PO**I data, http://slipo.eu/
[2] https://www.w3.org/RDF/
[3] https://www.w3.org/OWL/
[4] https://www.opengeospatial.org/standards/geosparql

Series ISSN: 2367-2005 10.5441/002/edbt.2019.59

ii) Temporal ambiguity:

– *Same POI, new location.* Location of a POI may change over time, e.g., a shop has moved to another (nearby?) place.
– *Defunct POI.* A POI may still be displayed on a map, a city guide, a navigation device, etc., but in the meantime may have stopped its operation or completely ceased to exist.
– *Same POI, change of type.* Often, the type of a POI or the operations, services and facilities it offers may change over time, e.g., a café turning to a restaurant or bar.

iii) Semantic ambiguity:

◇ *Same POI, different names.* POIs involving buildings, localities, etc. are often referred to by multiple names, in different contexts or time periods (e.g., "Acropolis/Parthenon", "Saint Petersburg/ Petrograd/ Leningrad"). The most typical case concerns *multi-lingual names* across datasets, possibly in different alphabets (e.g., "Acropolis" transcribed in Arabic, Cyrillic, or Chinese). Other textual characteristics can raise more concerns, especially *addresses* (e.g., renamed or renumbered streets).
◇ *Different POIs, same name.* It is relatively easy to disambiguate multiple locations or POIs with the same name when the spatial context is quite different (e.g., hotels with the same name in different cities). Instead, it can be quite challenging to infer what is the actual entity in the same spatial context (e.g., "Hyde Park" may refer to the park, to a nearby café, or to a hotel).
◇ *Same POI, different types.* Besides names, there is much heterogeneity in the use of *classification schemes*, category names and tags to semantically annotate and classify POIs. Each source typically employs its own vocabulary of categories and a hierarchy to classify POIs. Sometimes, user-defined *tags* may be assigned to POIs to describe them either instead of or in addition to predefined classification schemes.

All this makes it especially challenging and cumbersome to integrate and harmonize POI data from different sources. In SLIPO, our approach places particular emphasis to resolving ambiguity, as well as in coping with differing POI models, non-common identifiers, complex geometries, diverse attribute schemata, etc., by employing Linked Data technologies as explained next.

3 THE POI DATA INTEGRATION LIFECYCLE

In this Section, we provide an overview of the POI data integration *lifecycle*, as implemented in SLIPO. The underlying idea of our proposed system is to address the POI data integration challenges in the Linked Data domain, which is ideally suited to handle the inherent geospatial, thematic, and semantic ambiguities of POIs. Hence, POI data assets must first be transformed into RDF, so that POI profiles can be interlinked, fused, and enriched in successive steps. This is achieved through a virtuous cycle implementing iterative workflows (Figure 1) that progressively increase the size and/or the quality of the original POI data.. Next, we outline the processes and software tools involved in each step.

Transformation. In order to be handled in the Linked Data domain, POI assets from heterogeneous data sources must be transformed into RDF triples conforming to a common *OWL ontology*[5] for POI profiles. To provide a scalable and efficient transformation facility (shown as a thick red arrow in Figure 1), we extended our open-source software TripleGeo[6] to enable transformation of POI datasets from a variety of de facto geospatial formats into RDF triples with minimal overhead. Although TripleGeo is a general-purpose, spatially-aware ETL tool [3], we have included specific support for transforming POI data. This was possible through adaptable, configurable, and reusable mappings from existing attribute schemata into our POI ontology, and also supporting classification hierarchies in assigning categories to POIs. As TripleGeo inherently handles all geometry types and established coordinate reference systems, it can cope with the heterogeneity of POI formats and representations. Once data integration is complete, SLIPO introduces *reverse transformation* of the resulting integrated RDF data back to conventional POI formats (at the bottom in Figure 1), so that they can be exploited by existing products, systems, and services.

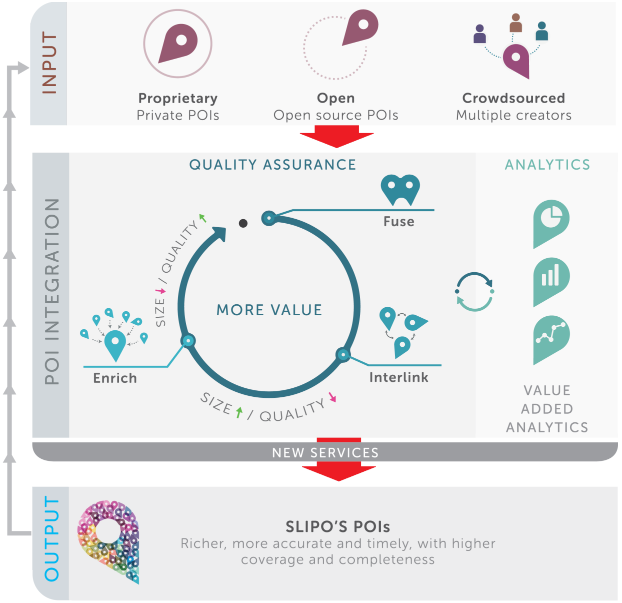


Figure 1: The POI data integration lifecycle.

All transformed RDF data are fed to a step-wise workflow abstracting a *virtuous circle*. This iterative cycle first increases the *size* (i.e., coverage, completeness, and richness) of POI data, and then refines them to increase *quality* of POIs by fusing intermediate results. For example, an expert user can repeatedly introduce additional data sources, apply different rules, etc. This iterative workflow involves the following stages:

Interlinking. This step is applied across the transformed RDF datasets coming from different sources in order to discover pairwise relations among real-world POI entities. We make use of Limes [5], a state-of-the-art interlinking software[7] that exploits the semantic structure of RDF data, textual similarities, proximity of geospatial representations, etc. In SLIPO, this actually concerns POI *deduplication*, as we wish to identify same real-world POIs based on user-specified metrics and thresholds. The output is `owl:sameAs` links between matching POI entities, which tackle the lack of common identifiers between POI entities across data sources, thus enabling their management at later stages of the integration process.

[5] Available at https://github.com/SLIPO-EU/poi-data-model
[6] Software available at https://github.com/SLIPO-EU/TripleGeo
[7] https://github.com/dice-group/LIMES

Enrichment. To produce enriched metadata and contextualize POI profiles based on information retrieved from external, third-party RDF data sources, we make use of Deer [4]. This software[8] identifies external (structured or unstructured) information related to POIs and creates extra properties. For instance, a POI profile can be enriched with opening hours, price ranges, event timelines, etc., available in SPARQL endpoints such as DBpedia[9]. It also discovers semantic interrelations between POI entities and other resources (e.g. areas, events, time), such as `partOf` relations (e.g., a shop is part of a shopping mall) or `occursAt` relations (e.g., events taking place at a certain venue).

Fusion. This stage consolidates linked POIs and their properties. From two linked POI entities it produces a unified representation, which is more complete, concise and accurate than the individual initial entities. In supporting scalable and quality-assured fusion of large POI datasets, we employ our fusion framework Fagi [1]. In SLIPO, we adapted and extended Fagi[10] with POI-specific similarity functions, learning mechanisms, and fusion actions. Such rules guide how POI properties will be fused (e.g., choose one, merge both) according to specific criteria (e.g. more complex, more timely) specified by stakeholders.

Value-Added Analytics. Having these integrated and enriched POI datasets it is then possible to provide added value services that involve *clustering* and *association discovery* among POIs. In SLIPO, we make use of Sansa [2], a software suite[11] that offers several large-scale aggregation strategies and predictive analytics, precious in geomarketing, tourism, logistics, etc.

As already mentioned, throughout this lifecycle we want to ensure that each phase produces correct and accurate results, taking into account dataset-specific and use-case-specific quality indicators and rules, including manual validation and authoring. Several indicators for such *quality assurance* can be used, most of them already adopted by industrial vendors that manage and exploit POIs: size, timeliness, coverage, accuracy, etc.

Last, but not least, we have implemented a service that allows users to track the integration and *evolution* of POI information across time and between different versions. This includes mechanisms for recording *provenance* by tracking the full history of changes per POI up to the current values of its various attributes. A graphical interface assists in visualizing and navigating through all available information, enabling users to intuitively explore where and how a POI actually changed across the various stages in the workflow.

4 THE SLIPO PROTOTYPE SYSTEM

We have been implementing a comprehensive, open-source software prototype that integrates tools for transforming, linking, fusing, enriching, and analyzing linked POI data aiming to support stakeholders in all stages of the POI data value chain. The SLIPO system[12] consists of the following main modules:

- *SLIPO Toolkit*: This is the collection of individual software components (Section 3) for transformation (TripleGeo), interlinking (Limes), fusion (Fagi), enrichment (Deer) and analytics (Sansa). Any tool can either be installed locally or invoked as part of the SLIPO workbench and APIs functionality.
- *SLIPO Workbench*: This web application allows users to orchestrate the Toolkit components and thus implement POI data integration workflows (like the one depicted in Figure 2) in a coherent, user-friendly, and flexible manner. It provides advanced capabilities for (a) uploading, searching and managing POI datasets in several formats, (b) designing, persisting and managing data integration workflows for POI datasets based on the features provided by the SLIPO Toolkit, (c) scheduling and monitoring the execution of data integration workflows, and (d) visualizing the results of such executions.
- *SLIPO APIs*: This is a collection of RESTful HTTP programming interfaces for invoking SLIPO Toolkit component functionality and integrating it into third-party systems. APIs only support the invocation of simple atomic functions (e.g., POI transformation). For composite operations, the Workbench web application must be used. Both SLIPO Workbench and APIs are exposed through the same web application server.

The SLIPO system is deployed within several virtual machines on top of the Synnefo cloud stack[13]. In the back-end, our prototype implements a workflow engine that executes data integration workflows and a scheduler for initializing workflow executions. The workflow engine and the SLIPO Toolkit components are deployed over a cloud infrastructure. Workbench and APIs exchange messages with the scheduler to execute workflows. A task is executed either in-process locally on the scheduler host, or remotely using Docker containers. Each component is deployed as a Docker Image and is responsible for providing a scalable instantiation for the requested operation (e.g., TripleGeo for transformation). A Toolkit component capable of partitioning its input and merging its output can also scale to multiple Docker containers. The scheduler only controls the total amount of resources allocated to a container, enforcing CPU/memory quotas derived from component-specific requirements and input size.

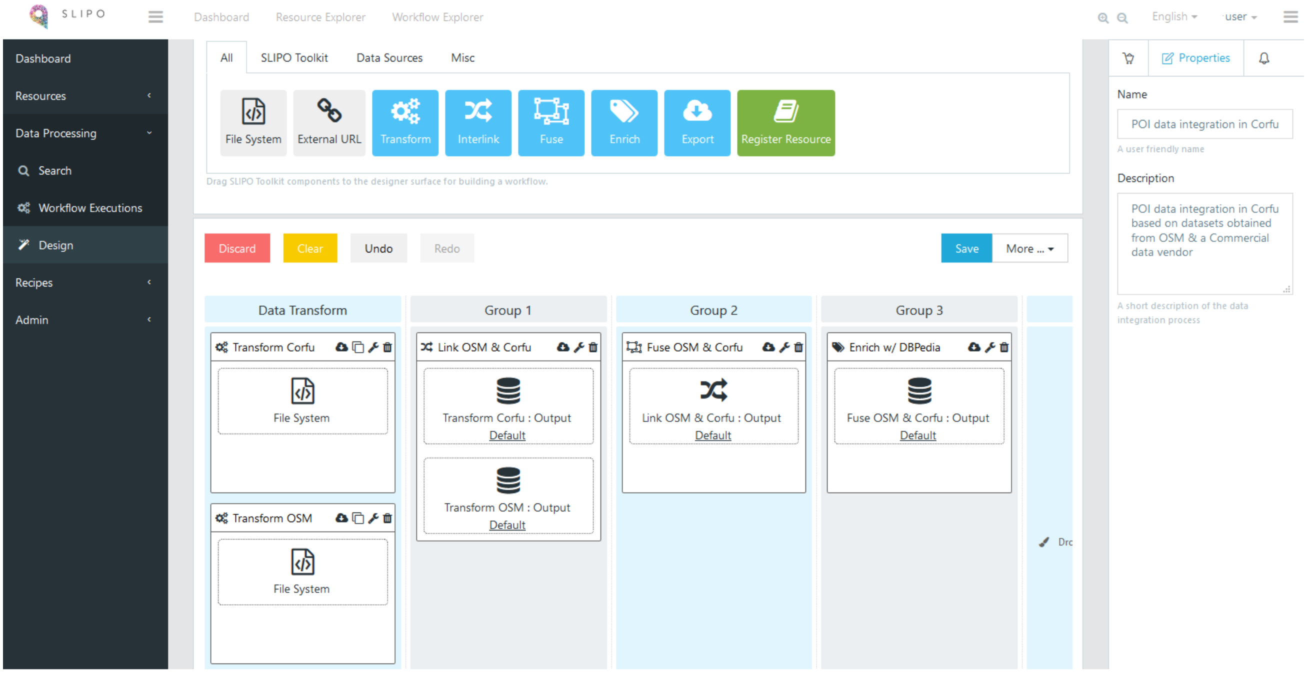
Thanks to its modular, service-oriented architecture, SLIPO offers stakeholders the option to directly use the provided functionalities following a Software-as-a-Service paradigm. Alternatively, they are able to select specific tools to customize, extend and incorporate in their own POI data management workflows using APIs according to their specific needs and requirements.

5 DEMONSTRATION

In this demonstration, we will showcase a data integration workflow using the SLIPO Workbench. This workflow accepts input POI datasets in a given geographical area (e.g., an island, a city, or a country). Data sources generally differ in schema, content, and quality; some concern *crowdsourced* information extracted from an open database (like OpenStreetMap[14] or GeoNames[15]), but others may be *proprietary* supplied by a commercial vendor.

Using the SLIPO Workbench, we will demonstrate how a user can define data integration workflows that deliver a single dataset in just a few minutes. Orchestrating the various tools into an executable workflow (like the one in Figure 2) can be carried out very quickly thanks to readily available profiles we have prepared for several common POI datasets. Such a workflow can be easily setup using drag and drop actions without the need to write any code, by only a basic parametrization per step (Section 3). This particular workflow first involves transformation of the input datasets. After discovering links between them, it fuses their

[8] https://github.com/dice-group/DEER
[9] https://wiki.dbpedia.org/
[10] https://github.com/SLIPO-EU/FAGI
[11] https://github.com/SANSA-Stack
[12] Current beta version is publicly available at https://github.com/SLIPO-EU
[13] https://www.synnefo.org/
[14] https://www.openstreetmap.org/
[15] https://www.geonames.org/



Name

POI data integration in Corfu

A user friendly name

Description

POI data integration in Corfu based on datasets obtained from OSM & a Commercial data vendor

A short description of the data integration process

Figure 2: Designing a POI data integration workflow in the SLIPO Workbench application.

properties according to user-specified rules and finally enriches the integrated result with external sources (e.g., DBpedia).

After executing such a workflow, the unified output dataset will be enhanced with information from all input datasets:

- The output dataset will contain *more POIs*. Starting with a base dataset (one of the input datasets), POIs missing from it will be complemented with information from the rest.
- Geometry representations can get a *more detailed shape*, e.g., polygons obtained from OpenStreetMap can replace (or complement) the original point (lat/lon) locations of certain POIs.
- *Extra thematic attributes* will be derived in the integrated dataset, by bringing together information (e.g., fax numbers, opening hours, links to photos, multi-lingual names) across the original data sources.
- Attribute values per POI will be *more accurate* and complete, e.g., missing telephone numbers can be filled or updated after checking against all available input.

We have prepared a short video[16] that demonstrates such a scenario on Corfu Island with a commercial POI dataset (*GET*)[17] enriched from OpenStreetMap (*OSM*). The map in Figure 3 shows how integration results (POIs depicted in blue circles or blue polygons) supersede by far and enhance the original information of the base dataset (*GET*) shown with red stars. Also, thematic properties per POI are substantially enriched with more attributes, while missing values in the base dataset are properly updated. Improvement in quality can be tracked graphically per individual POI by inspecting how it evolved along the integration progress, but also through statistics (attribute gain, confidence, etc.) estimated over the final dataset.

Overall, we believe that this demo will offer more insight not only about the challenges, but also regarding the benefits of POI data integration using SLIPO. As we continue our efforts to enhance and further develop our software, we expect rapid uptake of our innovations by stakeholders in a production setting without affecting any operations and processes already in place.

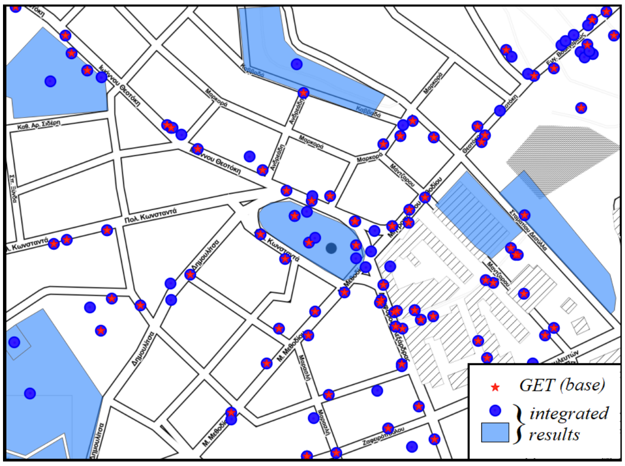


Figure 3: POIs before and after data integration in Corfu.

ACKNOWLEDGMENTS

This work was partially funded by the EU H2020 project SLIPO (#731581).

REFERENCES

[1] G. Giannopoulos, N. Vitsas, N. Karagiannakis, D. Skoutas, and S. Athanasiou. FAGI-gis: A Tool for Fusing Geospatial RDF Data. In *ESWC*, pages 51–57, 2015.

[2] J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, I. Ermilov, S. Bin, N. Chakraborty, M. Saleem, A.-C. N. Ngonga, and H. Jabeen. Distributed Semantic Analytics using the SANSA Stack. In *ISWC*, 2017.

[3] K. Patroumpas, M. Alexakis, G. Giannopoulos, and S. Athanasiou. TripleGeo: an ETL Tool for Transforming Geospatial Data into RDF Triples. In *EDBT/ICDT Workshops*, pages 275–278, 2014.

[4] M. Sherif, A.-C. Ngonga Ngomo, and J. Lehmann. Automating RDF dataset transformation and enrichment. In *ESWC*, 2015.

[5] M. A. Sherif and A.-C. N. Ngomo. A Systematic Survey of Point Set Distance Measures for Link Discovery. *Semantic Web Journal*, 2017.

[16] https://drive.google.com/file/d/1NPhl2mgbSdqH9A5KMZufQF3-7zihZ3zb/view

[17] Data sample courtesy of GET Ltd., http://www.getmap.eu/en/

Demonstration

A Map Search System based on a Spatial Query Language

Yuanyuan Wang
Yamaguchi University, Japan
y.wang@yamaguchi-u.ac.jp

Panote Siriaraya
Kyoto Sangyo University, Japan
spanote@gmail.com

Haruka Sakata
Kyoto Sangyo University, Japan
g1544647@cc.kyoto-su.ac.jp

Yukiko Kawai
Kyoto Sangyo University/Osaka University, Japan
kawai@cc.kyoto-su.ac.jp

Keishi Tajima
Kyoto University, Japan
tajima@i.kyoto-u.ac.jp

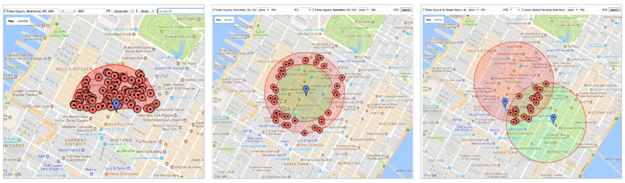
ABSTRACT

We propose a novel query language that can express complex spatial queries in a concise and intuitive way for map search. The proposed language can express conditions on the range, direction, and time distance within their spatial search queries. In this language, we introduce several spatial operators, such as “space character” operator which is used to represent the geographical distance between matching objects in a concise and intuitive way as well as arithmetic and directional operators which enables the combination and manipulation of spatial areas. We also show how a map search system supporting this query language can be implemented, and describe several applications created using this system to highlight how the query language can be put into practice. These applications include a web interface which allows developers to embed a function of spatial search by our query language into their systems. In addition, we developed a mobile Android application that allows nonprofessional users to easily search for nearby venues and routes by using the proposed query language. Finally, we outline the results of a study carried out to evaluate the potential usefulness of our proposed search system.

1 INTRODUCTION

In map search systems, keyword-based queries are widely adopted due to their ease of use. Such queries are however inadequate when more complex requests are needed. For example, consider the case where a software tool would need to be developed to help identify appropriate locations for real estate development. Such a software needs to deal with search requests which contain multiple constraints and spatial conditions. For instance, those interested in constructing a family-friendly apartment building would search for vacant locations that have many schools and playgrounds within traveling distance, but are far enough away from inappropriate locations or noisy public spaces.

In map search systems with simple keyword-based queries, such a search task requires multiple query transactions by user operations. For example, to find a good apartment for a family, the transactions need to (1) search for apartments by using a keyword query, (2) also identify schools on the map, (3) limit the query result to those within 4km from the found schools, and (4) also exclude those which are within 200m from some inappropriate locations. The same is true for various complex requests such as “Finding all the restaurants located between two famous tourist landmarks” (when developing an application for tourism) or “Finding all the shops located next to parks in a city” (for location-based recommender systems). As shown in these examples, it is difficult to process complex search requests using simple keyword-based queries. Systems only supporting such keyword-based queries require multiple steps including non-textual interactions to process them.



(a) ^ operation (b) - operation (c) * operation

Figure 1: Examples of map search using spatial operators.

On the other hand, there have been much research on spatial logic or algebra. In a spatial database of PostGIS[1], the spatial search task with location queries can be run in SQL, and they can represent complex spatial conditions in queries, but they require users to learn and understand the progrmming-language-like syntax, and as a result, they are too complicated for general users in many applications. They are, therefore, impractical for the use in such systems.

The goal of this research is to design a simple spatial query language which can represent such complex queries within a single query statement with a concise, and more intuitive syntax so that users can easily specify complex queries. Our language uses 10 spatial operators to represent conditions on directions, ranges, angles, and time distance, which allow users to incorporate spatial conditions and manipulation of spatial areas within their keyword-based-like queries. Fig. 1 shows examples of map search carried out by queries including spatial operators in our language, such as ([^] [−] [*]). For example, Fig. 1 (c) shows the result of a query (“my house”␣0.5km␣“pizza parlor”) * (“friends house”␣0.5km␣“pizza parlor”).

2 RELATED WORK

As explained in Section 1, there have been much research on spatial databases based on region algebra or region logic. However, most current commercial location-based services such as Google Maps or Bing Maps are designed mainly to help general users to execute two types of simple tasks: (1) find certain places and locations within a specified geographical area and (2) find the best route (e.g., shortest distance, most economical) between two given locations. Recent academic research in this domain also mainly focus on similar problems, such as locating comfortable, aesthetically pleasing or safe routes [2] and personalizing the search results by identifying locations which better match the latent interests of users [5].

[1] https://postgis.net/

Series ISSN: 2367-2005

10.5441/002/edbt.2019.60

Table 1: Spatial operators by using the space-key

Operator	␣*	␣^	␣␣	␣@	␣[x-y]	␣$	␣#
Processing	Surrounding	Direction (north/up)	Direction (south/down)	Angle	Range	Size	Time

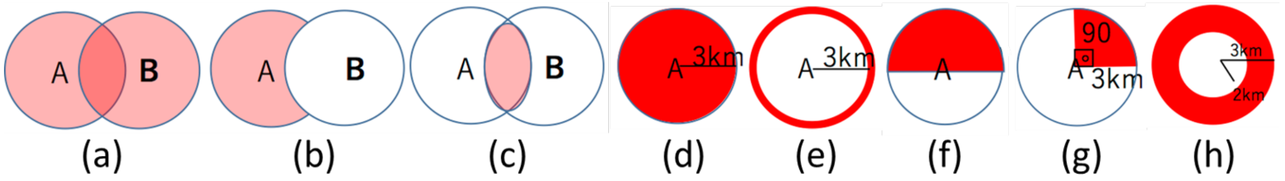


Figure 2: (a) Union (b) Difference (c) Intersection (d) Within (e) Distance (f) Direction (north) (g) Angle (90 degree) (h) Range

Because most of the current commercial systems are focusing on general users executing these types of tasks, they use simple keyword-based queries instead of complicated query languages. For example, if users need to search for restaurants in London, they would simply input a query such as "Restaurants in London". For users who need more complex and precise search requests, however, such search systems are inadequate. Although various API systems[2] [3] have been created to provide access to the more advanced features of the location search systems, they are often limited to single-process tasks (finding locations within a specific distance, geo-coding a specific location name, etc.). Therefore, in this paper, we propose a novel query language to enhance the existing API systems, which allows users to express conditions on distance, space and time distance towards objects matching the query keywords through the use of spatial operators.

While the use of operators in keyword-based queries for map search is not common, the use of operators in keyword-based queries can be found in other domains. In document search, proximity operators have been proposed as a way to limit search results to those that contain the keywords in a specific order or within a specific distance [3]. Also in video database domain, Pradhan et al. [4] proposed operators which allow users to represent constraints on pairs of matching objects to be joined. In graph database domain, Cypher[4] is a declarative query language that allows users to state what actions they want performed upon their graph data without requiring them to describe (or program) exactly how to do it.

3 THE SPATIAL QUERY LANGUAGE

A description of our proposed spatial query language is described in this section. Overall, there are two main rules which are used to define the syntax for a spatial search unit in our query language.

Rule 1: The syntax of the most primitive unit of spatial queries is defined as follows: "A␣spatial length␣α" A and α are keywords, with A denoting the location of the origin for the spatial search for the object with the property α. It is permissible to either denote the spatial length by using a unit distance (i.e., 200m, 3 minutes) or by using continuous spaces. When continuous space is used, the spatial length would represent the N nearest locations with the property α, where N is represented by the number of continuous spaces.

(Example) : A␣800m␣α denotes a query statement to identify the α objects which exists inside the region 800m from the origin point A.

(Example) : A␣␣␣α denotes a query statement to identify the nearest 3 α objects from the origin point A.

Rule 2: The keywords (e.g., A and α) used in the primitive unit of the spatial query would be encapsulated within a double quotation mark (e.g., "Tokyo tower" or "Grand Central Terminal, New York" for A or "pizza shop" or for α).

In addition, various spatial, directional and distance operators can be used to impose conditions when conducting a spatial search.

Rule 3: Each primitive spatial search Unit can be combined with other units through the use of spatial, directional and distance operators in a mathematical equation format.

(Example) : (A␣800m␣α) + (B ␣300m␣α)

3.1 Spatial Operators

The standard set operators can also be used as spatial operations for the query unit defined previously. These include the union [+], difference [-], and intersection [*]. Users can use such operations to manipulate the spatial region they wish to search into. Examples of queries including these operators are as follows:

Union calculation (Ex.1) : (A␣3km␣α)+(B␣3km␣α) denotes the union of the spatial region *within* 3km from point A AND the spatial region *within* 3km from point B (see Fig. 2 (a)).

Difference calculation (Ex.2) :(A␣3km␣α)-(B␣3km␣α) denotes the spatial region *within* 3km from point A which is NOT *within* 3km from point B (see Fig. 2 (b)).

Intersection calculation (Ex.3) :(A␣3km␣α)*(B␣3km␣α) denotes the spatial region that is *within* 3km from point A which is ALSO *within* 3km from point B (see Fig. 2 (c)).

All set operators (+, -, *) can be used to search for objects with the properties identified in the query unit. For example, the aforementioned (A␣3km␣α)*(B␣3km␣α) query would search for objects with the property α which is located within the spatial region that is the result of the intersection between 3km from points A and B.

3.2 Directional and Distance Operators

Table 1 shows the 7 spatial operators which can be used to further denote distance and direction within our spatial query. Examples of four expressions which use these operators are described below:

Distance operation (Ex.4) : A␣3km␣α

[2] https://www.microsoft.com/en-us/maps/choose-your-bing-maps-api
[3] https://cloud.google.com/maps-platform/
[4] https://neo4j.com/cypher-graph-query-language/

retrieves the objects α which are *within* 3km from point A (Fig. 2(d)).

Within operation (Ex.5) : A␣*3km␣α

retrieves the objects α that are 3km *away from* point A (Fig. 2 (e)).

Direction operation (Ex.6) : A␣^3km␣α

retrieves the objects α which are to *the north* of, and within 3km from, point A (see Fig. 2 (f)).

Angle operation (Ex.7) : A␣3km@90␣α

retrieves the objects α which exist *within* 3km in the 90 *degree* counterclockwise direction from point A (see Fig. 2 (g)).

3.3 Range, Size, and Time Operators

Our proposed spatial query language also includes a variety of range operators which allows users to more accurately specify the desired search range within their query (see Table 1). For example:

Range operation (Ex.8) : A␣[1km-3km]␣α

retrieves the objects α in the region from 1km to 3km from point A (see Fig. 2 (h)).

In addition, size [$] and time operators [#] can be used to formulate search queries. The size operator $ extracts the size of the corresponding property of object α and uses it as a unit of measure (i.e., 1 city block = 0.5km). The same is true for time operations depend on the context (i.e., using A␣#3min to find venues which are 3 minutes walking distance from point A, when a user selects a "walking" direction).

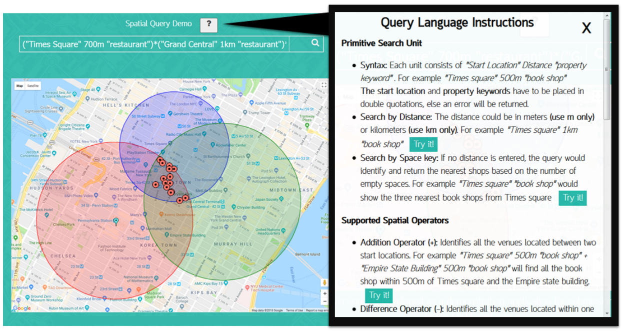
4 SPATIAL SEARCH SYSTEM

In this section, we explain the structure of our search system. The system consists of three main components: (1) an Input/Output component that processes user requests and outputs them to the appropriate format; (2) an interpreter component that parses and processes queries and (3) the data processing program component to link the search system to appropriate data sources.

Users of our system would send an HTTP request to the server with details of the spatial query as parameters. The query specified by the client is then passed to the interpreter and the data processing components. These components would parse the query, process the request and send the results back to the Web In/Output Processing component which would transmit the results back to the client as an HTTP response in a data format such as JSON or XML.

The role of the interpreter component is to process the spatial operators sent as the request. This component consists of a query parser, a spatial data converter, and a spatial data calculator. For the parser, the role is to analyze the user query and determine the appropriate operations and procedures to process it. For example, the following query: (A␣^3km␣α) + (B␣^3km␣α) would be processed by the parser into the following steps: Var1= A␣^3km␣α (step 1), Var2= B␣^3km␣α (step 2), and Result=Var1+Var2 (step 3).

These steps would then be processed by the interpreter. Each spatial variable is sent to the data converter to convert the elements (such as A␣^3km or B␣^3km) in the query to spatial regions which represents the correct distribution of those elements. The conversion program would access information provided by the data processing component to calculate the appropriate regions. For example, when processing the element "A␣^3km␣shops", the data processing component would calculate the geographical location of point A as well as the geographical locations of shops within a 3km radius. After the data has been converted, the spatial operators are then processed. For example, if the request query contains the intersection operator [*], it would calculate the region which is the overlap between the converted A␣3km and B␣3km regions. After all the calculations have been completed, the result is sent back to the client in the appropriate data type (JSON or XML etc.) as specified by the data processing component.



Spatial Query Demo ?

("Times Square" 700m "restaurant")*("Grand Central" 1km "restaurant")*

Query Language Instructions X

Primitive Search Unit

- **Syntax:** Each unit consists of *"Start Location" Distance "property keyword"*. For example *"Times square" 500m "book shop"* **The start location** and **property keywords** have to be placed in double quotations, else an error will be returned.
- **Search by Distance:** The distance could be in meters **(use m only)** or kilometers **(use km only)**. For example *"Times square" 1km "book shop"* Try it!
- **Search by Space key:** If no distance is entered, the query would identify and return the nearest shops based on the number of empty spaces. For example *"Times square" "book shop"* would show the three nearest book shops from Times square Try it!

Supported Spatial Operators

- **Addition Operator (+):** Identifies all the venues located between two start locations. For example *"Times square" 500m "book shop" + "Empire State Building" 500m "book shop"* will find all the book shop within 500m of Times square and the Empire state building. Try it!
- **Difference Operator (-):** Identifies all the venues located within one

Figure 3: Spatial query language demo application [5].

A prototype of the search system engine was implemented as a RESTFUL web service using nodeJS. The current system supports spatial map search, with the input being the requested as a spatial query (an HTTPS GET request) and the output is an array of locations which match the spatial query (returned using the JSON data format) together with details such as the name of the place and the address. The equation expression within the spatial query was parsed using the Shunting-yard algorithm. Google Maps API was used in the data processing proportion to identify the various locations specified in the primitive spatial query unit (i.e., "times square") and their geographical positions. The system could also later be easily adapted to utilize other data sources such as Open Street Map data or a customized SQL database as well.

5 DEMONSTRATION APPLICATIONS

To highlight how the system could be useful in practice, a number of web applications were created which utilized our proposed spatial search query language and would be shown during the demonstration. The first application was a web interface for our search system engine which users could use to test the query language or search for locations using spatial equations[3]. Users would be able to use the spatial, range and directional operations described in Section 3 as well as mathematical expressions such as brackets to compose their search queries. After clicking the search button, the system would send the user's query to the search system server and would then render the search results received from the server onto the map. For example, Fig. 3 shows the results of the query: (("Times Square"␣700m␣"restaurant")*("Grand Central"␣1km␣"restaurant"))*("Pennsylvania station"␣1km␣"restaurant"), which aims to identify all restaurants located within 700m of Times Square and 1km from Grand Central and Pennsylvania station. One potential use-case for such a query is for example to identify potential meeting places for three users based on their starting locations (For example, when one user works near Times Square and the other near Grand Central and the final near Pennsylvania station and the system

[5] http://yklab.kyoto-su.ac.jp/~sakata/spatialQueryDemo

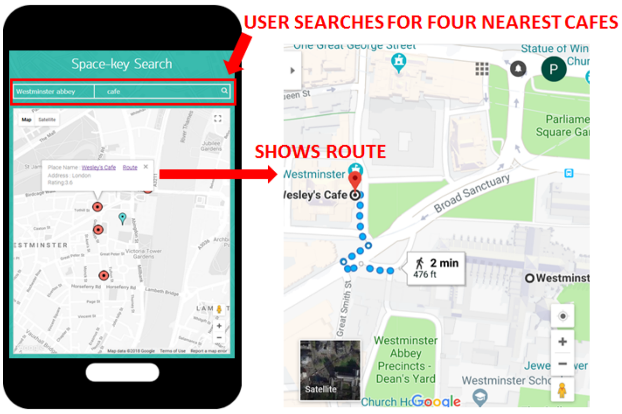


Figure 4: Space-key search application for novice users.

would need to find a restaurant that is equally near to all three of their workplaces for them to meet for lunch). The web interface system also provides an instruction page where the various operators in our query language are explained and a number of examples shown (see Fig. 3). Users could click on the "try it" button to examine the search results of the examples and could also freely modify the example equations.

Furthermore, another application which utilized our proposed query language (only using the "space-key" for a more simple and intuitive search) would also be shown during the demonstration. This application was conceptualized by looking at how non-professional common users generally used location-based mapping services. Although route navigation was a commonly used feature, users also generally used location-based services to quickly identify different types of nearby venues and then find out how they could travel to such locations. Therefore, we developed an android application ("space-key search" application) which utilized the primitive unit of our spatial query language to allow users to search for nearby venues (users are able to search for nearby locations using only the space-key). For example, the user would enter the query "Current Location␣␣␣␣␣Restaurants", to find the four nearest restaurants to them on auto adjust map zooming (see Fig. 3). Clicking on the markers would show details of the venue (the address, review scores etc.) as well as a link with the details of the route to the store. The application itself could be downloaded from Google play store[6]. A mobile web version of this application [7] was also developed for evaluation and demonstration purposes (Fig. 4 shows screen-shots of the applications). A demonstration video of the applications discussed in the paper which would be presented at the conference could be viewed from the following link [8].

6 USER STUDY

A user evaluation study was also carried out to evaluate the potential usefulness of our proposed query language. The main aim was to determine whether such an equation based query language would be feasible for developers to learn and use. 15 students from a university-level computer science course were recruited and asked to carry out a series of location search tasks. Each participant was asked to use both our proposed query language through the web interface system which was developed (spatial language condition) as well as through the Google Maps system (Google map system) to complete 5 search assignments. Each search assignment consisted of a task to search for places (e.g., pizza parlors) near a specific location (e.g., The White House). For example, one task consisted of trying to find the number of pizza parlors located within 500m of Times Square. In another task, participants were asked to find the number of pizza parlors located within 700m of Times Square which is also located 1000m from Empire State Building. Written instructions and examples were provided to help participants complete the tasks and introduce the various spatial operators. An objective measurement of performance was obtained by measuring the time users spent on each task. To measure subjective user experience, the System Usability Scale (SUS) was used [1], which involved the rating of perceived effectiveness, efficiency, and satisfaction.

Overall, participants rated a higher SUS score for the spatial language condition (Mean=70.17, SD=13.09) than the Google map condition (Mean=24.46, SD=10.61) (t(13)=7.24, p<0.001)). In addition, participants were able to complete the tasks using less time (seconds) in the spatial language condition (Mean=82.27, SD=28.18) then the Googlemaps condition(Mean=180.86,SD=49.25) (t(13)=-8.219, p<0.001). Therefore, it seems that at least for search tasks which involve the combination and manipulation of spatial regions, the proposed map search system could indeed be useful.

7 CONCLUSION

In this paper, we proposed a novel query language for spatial search which can be used to express complex queries in a text equation format. We implemented a prototype of our proposed system and developed two applications (a web based application and a mobile application) to showcase how the query language could be put into practice. In addition, we conducted a user study, The results of which highlights the potential usefulness of our query language.

In the future, we look to expand our query language to other search domains such as text and video search. Although we have shown how our language could be used in map search, our proposed query language could easily be applied to spatial search within documents and videos as well. For example, a query searching for text within a document that contains the word "B" and is within 5 sentences from the word "A" is expressed by "A␣5sentences␣B".

ACKNOWLEDGMENT

The work in this paper is partially supported by JSPS KAKENHI Grant Numbers 16H01722, 15K00162, 17K12686.

REFERENCES

[1] John Brooke. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

[2] Jaewoo Kim, Meeyoung Cha, and Thomas Sandholm. 2014. SocRoutes: Safe Routes Based on Tweet Sentiments. In *Proceedings of the 23rd International Conference on World Wide Web (WWW '14 Companion)*. ACM, New York, NY, USA, 179–182. https://doi.org/10.1145/2567948.2577023

[3] Elsevier Newsletter. 2015. How Can I Search Literature with Reduced Noise? Utilization of "Proximity Operator" in ScienceDirect & Scopus. (August 19 2015).

[4] Sujeet Pradhan, Keishi Tajima, and Katsumi Tanaka. 2001. A query model to synthesize answer intervals from indexed video units. *IEEE Transactions on knowledge and data engineering* 13, 5 (2001), 824–838.

[5] Hongzhi Yin, Yizhou Sun, Bin Cui, Zhiting Hu, and Ling Chen. 2013. LCARS: A Location-content-aware Recommender System. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '13)*. ACM, New York, NY, USA, 221–229. https://doi.org/10.1145/2487575.2487608

[6] https://play.google.com/store/apps/details?id=com.kawaiLab.spatialQuery

[7] https://yklab.kyoto-su.ac.jp/~sakata/simple/spatialQuary/

[8] http://yklab.kyoto-su.ac.jp/SpatialDEMO/Spatial_demo_movie.mp4

FaiRank: An Interactive System to Explore Fairness of Ranking in Online Job Marketplaces

Ahmad Ghizzawi[1], Julien Marinescu[2], Shady Elbassuoni[1], Sihem Amer-Yahia[2], Gilles Bisson[2]
[1]The American University of Beirut (Lebanon), [2]Univ. Grenoble Alpes, CNRS, LIG (France)
[1]{ahg05,se58}@aub.edu.lb, [2]firstname.lastname@univ-grenoble-alpes.fr

ABSTRACT

We demonstrate FaiRank, an interactive system to explore fairness of ranking in online job marketplaces. FaiRank takes as input a set of individuals and their attributes, some of which are *protected*, and a scoring function, through which those individuals are ranked for jobs. It finds a partitioning of individuals on their protected attributes over which fairness of the scoring function is quantified. FaiRank has several appealing features: (1) It can be used by different users: *the auditor* whose role is to monitor the fairness of ranking in a job marketplace, *the job owner* seeking to examine the influence of a scoring function and its variants on the ranking of candidates for a job, and *the end-user* who wants to assess the fairness of jobs on different marketplaces; (2) It is able to quantify fairness under different data and process transparency settings: when some attributes are anonymized and when only the ranking (and not the scoring function) is available; (3) It is interactive and lets its users explore different scoring functions and examine how fairness evolves; (4) It is generic and provides the ability to quantify different notions of fairness. Our demonstration will provide attendees with several scenarios for fairness of ranking in job marketplaces to experiment with and acquire an understanding of this important research question and its impact in practice.

1 INTRODUCTION

Freelancing marketplaces have become an online destination to find a temporary job. The ranking of individuals on platforms such as Qapa and MisterTemp' in France, and TaskRabbit and Fiverr in the USA, naturally poses the question of fairness. Fairness in ranking has recently received great attention from the data mining, information retrieval and machine learning communities (See for instance [1, 4, 6, 9, 10]). The most common definition of fairness in decision making was introduced in [2, 11] as *demographic parity*, and formalized in [3] as *group unfairness*. This definition captures the unequal treatment of a person based on *belonging to a certain group of people* defined using protected attributes such as gender and ethnicity. For instance, in the French Criminal Law (Article 225-1), 23 such attributes are listed as discriminatory.[1] The exact formulation of fairness varies and the purpose of FaiRank is to explore different formulations and unveil their impact on individuals.

User Roles. FaiRank appeals to different users. *The auditor*, whose role is to monitor the fairness of ranking in a marketplace, can use FaiRank to examine different jobs on that marketplace and quantify their fairness. *The job owner*, who wants to study the behavior of a scoring function and its variants, can use FaiRank to understand their impact on the ranking of individuals, and choose fairest one. Finally, *the end-user*, who is being ranked, can use FaiRank to assess the fairness of jobs on different marketplaces and make an informed decision.

Positioning. Most previous work on group-level fairness have either assumed that groups are pre-defined [9] or that they are defined using a single protected attribute (e.g., males vs females or whites vs blacks) [5]. FaiRank extends prior work to examine groups of people defined by *any combination of protected attributes* (the so-called *subgroup fairness* [6]). The scoring function yields one histogram per group as a score distribution. We use the Earth Mover's Distance (EMD) [8], a measure commonly used to compare histograms, to quantify the difference between score distributions across groups. The intuition is that if score distributions between groups differ significantly, the scoring function treats individuals in those groups unequally. This allows exploring different fairness formulations in FaiRank as any aggregation function over pairwise distances of score distributions in groups (highest average, lowest variance, etc.).

Since we do not want to focus only on pre-defined groups to quantify fairness, we must exhaust all possible ways of partitioning individuals into groups based on their protected attributes. This would capture cases where a scoring function treats males and females equally but is unfair to older African Americans compared to younger White Americans for instance. To examine all groups under different fairness definitions, we formulate an optimization problem as finding a partitioning of the ranking space, i.e., individuals and their scores, that exhibits some aggregation over pairwise partitions (e.g., the highest average EMD between partitions, the lowest average, the highest variance, etc.). Exhaustively enumerating all groups is exponential in the number of values of protected attributes. Therefore, to enable interactive response time, FaiRank relies on an efficient heuristic algorithm. At each step, the algorithm greedily splits individuals using the *most unfair* attribute according to the fairness definition. This local condition is akin to the one made in decision trees using gain functions [7]. The algorithm stops when there are no further attributes left to split on or when the current partitioning of individuals exhibits more unfairness than it would if its partitions were split further.

Data and Function Transparencies. In practice, data about individuals, i.e., their attributes, or the scoring function itself, may not be available. We integrate FaiRank with the k-anonymization ARX tool[2] and explore fairness for anonymized datasets. When the function is not available, FaiRank builds histograms using ranks of individuals rather than actual function scores.

Demonstration. Our demonstration combines the features of FaiRank to help attendees explore fairness of ranking in online job marketplaces and its impact in practice. It also sheds light

[1] https://www.legifrance.gouv.fr/affichCodeArticle.do?cidTexte=LEGITEXT000006070719&idArticle=LEGIARTI000006417828

[2] https://arx.deidentifier.org/

Series ISSN: 2367-2005

10.5441/002/edbt.2019.61

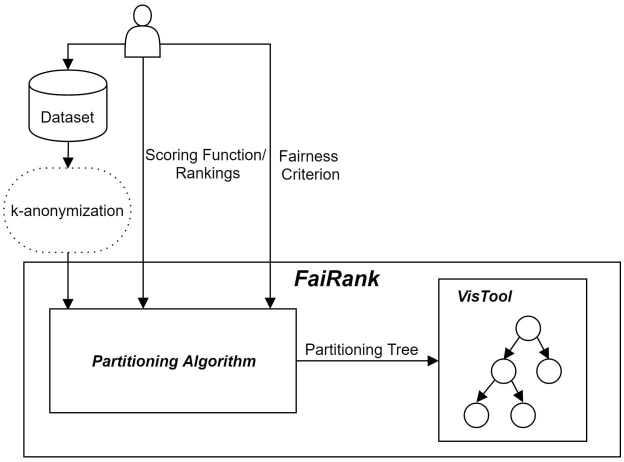


Figure 1: System architecture of FaiRank

on the interplay between data and function transparencies and the ability to quantify fairness. Additionally, FaiRank enables the exploration of different scoring functions, which can help choose the fairest for a given job. Finally, FaiRank can be used with standalone datasets and scoring functions, and since it can operate under various transparency settings, it can be used as a service to quantify fairness in existing blackbox job marketplaces.

2 SYSTEM OVERVIEW

Figure 1 depicts the system architecture of FaiRank. The user can select or upload a dataset which consists of a set of individuals and their attributes. The attributes can be protected such as gender, age, location, ethnicity, etc. or reflective of the performance or skills of the individuals such as reputation, knowledge in plumbing, writing skills, and mathematical abilities. Some of these attributes can also be anonymized. The user of the system can define or select a scoring function to rank individuals. The scoring function can be defined on a subset of the attributes of the individuals, for example a linear combination of an individual's reputation and plumbing skills, or of English writing skills and expertise in computer science. In addition, the user can filter the individuals based on protected attributes. This can be helpful in scenarios where the user is only interested in ranking a subset of individuals that satisfy certain criteria, say only individuals who speak Arabic or who are located in New York city. Instead of a scoring function, the user can also provide some ranking for the individuals (i.e., in the case that the scoring function is not transparent).

FaiRank solves an optimization problem that finds a partitioning of individuals over their protected attributes for which unfairness is subjected to an aggregation function (maximized, minimized, etc). The partitioning is displayed in a panel for the user. The user can interact with the returned partitions, view statistics such as the number of individuals in each partition, as well as a histogram of the scores of the individuals in each partition. The user can also choose to modify the scoring function or the fairness formulation, and obtain several panels to explore how that impacts fairness quantification. In the next section, we explain how we partition workers and quantify fairness of a scoring function given a set of individuals.

3 QUANTIFYING FAIRNESS

3.1 Model

To quantify fairness, we model the problem as aggregating a distance between the score distributions of all possible partitions of individuals. Unlike previous work where partitions were defined or known a priori (e.g., [5]), in FaiRank we explore the space of all possible groups defined by a combination of values of the individuals' protected attributes. The goal becomes finding an unfair partitioning of individuals under the scoring function. This can be formulated in many ways. For instance, the worst-case formulation would correspond to finding the *highest* distance between partitions. We cast this goal as an optimization problem as follows.

Definition 1 (Most Unfair Partitioning Problem). *We are given a set of individuals W, where each individual is associated with a set of protected attributes $A = \{a_1, a_2, ..., a_n\}$ and observed attributes $B = \{b_1, b_2, \ldots, b_m\}$. The protected attributes are inherent properties of the individuals such as gender, age, ethnicity, origin, etc. The observed attributes represent the skills of individuals for jobs and could include, for instance, the reputation and writing skills of an individual. We are also given a scoring function $f : W \longrightarrow [0,1]$, which is defined using observed attributes as follows: $f(w) = \sum_{i=1}^{m} \alpha_i b_i$, where α_i is a user-defined weight for observed attribute b_i. A weight of zero indicates that the corresponding attribute is not relevant for the user in ranking the individuals. Our goal is to* fully *partition the individuals in W into k* disjoint *partitions $P = \{p_1, p_2, \ldots, p_k\}$ based on their protected attributes in A using the following optimization objective:*

$$\begin{aligned} &\underset{P}{argmax} && unfairness(P, f) \\ &subject\ to && \forall i, j\ p_i \bigcap p_j = \phi \\ & && \bigcup_{i=1}^{k} p_i = W \end{aligned}$$

Another formulation, *Least Unfair Partitioning Problem*, would be to find the partitioning that results in the smallest unfairness (i.e., *argmin* instead of *argmax* in the formulation above).

We now define how to compute the amount of unfairness of a function f for a partitioning P, or *unfairness*(P, f) in the above optimization problem.

Definition 2 (Average Pairwise Unfairness). *For a set of individuals W, a full-disjoint partitioning of the individuals $P = \{p_1, p_2, \ldots, p_k\}$ and a scoring function f, unfairness of f for the partitioning P is quantified as the average pairwise Earth Mover's Distance (EMD) between the distribution of scores in the different partitions of P, which is computed as follows:*

$$unfairness(P, f) = \underset{i,j}{avg} \quad EMD(h(p_i, f), h(p_j, f))$$

where $h(p_i, f)$ is a histogram of the scores of individuals in p_i.

Another possible formulation is to compute unfairness as the maximum pairwise EMD, which would correspond to finding the partitioning with the highest maximum EMD between any pair of partitions.

Example. Consider the example dataset shown in Table 1 consisting of 10 individuals on a crowdsourcing platform who are ranked for some job using a scoring function f. Figure 2 shows one possible partitioning of those 10 individuals, that results

Table 1: An example dataset consisting of 10 individuals and a scoring function using Language Test and Rating

Individual	Gender	Country	Year of Birth	Language	Ethnicity	Experience	Language Test	Rating	f(w)
w1	Female	India	2004	English	Indian	0	0.50	0.20	0.29
w2	Male	America	1976	English	White	14	0.89	0.92	0.911
w3	Male	India	1976	Indian	White	6	0.65	0.65	0.65
w4	Male	Other	1963	Other	Indian	18	0.64	0.76	0.724
w5	Female	India	1963	Indian	Indian	21	0.85	0.90	0.885
w6	Male	America	1995	English	African-American	2	0.42	0.20	0.266
w7	Female	America	1982	English	African-American	16	0.95	0.98	0.971
w8	Male	Other	2008	English	Other	0	0.30	0.15	0.195
w9	Male	Other	1992	English	White	2	0.32	0.25	0.271
w10	Female	America	2000	English	White	5	0.76	0.56	0.62

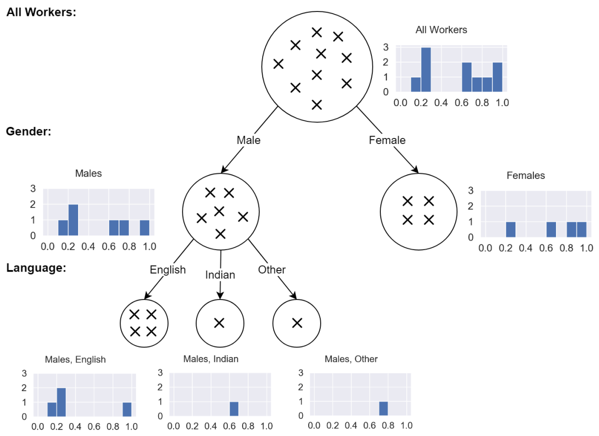


Figure 2: A partitioning of the example dataset

Algorithm 1 QUANTIFY(*current*: a partition, *siblings*: a set of partitions, f: a scoring function, A: a set of attributes)

```
if $A = \emptyset$ then
    Add $current$ to $output$
else
    $currentAvg = avg(EMD(current, siblings, f))$
    $a = mostUnfair(current, f, A)$
    $A = A - a$
    $children = split(current, a)$
    $childrenAvg = avg(EMD(children, siblings, f))$
    if $currentAvg \geq childrenAvg$ then
        Add $current$ to $output$
    else
        for each partition $p \in children$ do
            QUANTIFY($\{p\}, children - \{p\}, f, A$)
        end for
    end if
end if
```

from splitting them based on Gender first, and then splitting only the Male partition based on Language to get the following partitioning of individuals: *Male - English, Male - Indian, Male - Other,* and *Female*. We quantify the unfairness of partitioning P as the average EMD between the pairs of partitions in P. To identify the most unfair partitioning, one must exhaust all possible *full disjoint* partitionings of individuals based on their protected attributes To do that, we generate a histogram for each partition as indicated in Figure 2 based on the function scores by creating equal bins over the range of f and counting the number of individuals whose function scores fall in each bin. The most unfair partitioning is then the one with maximum average pairwise EMD between its partitions' histograms.

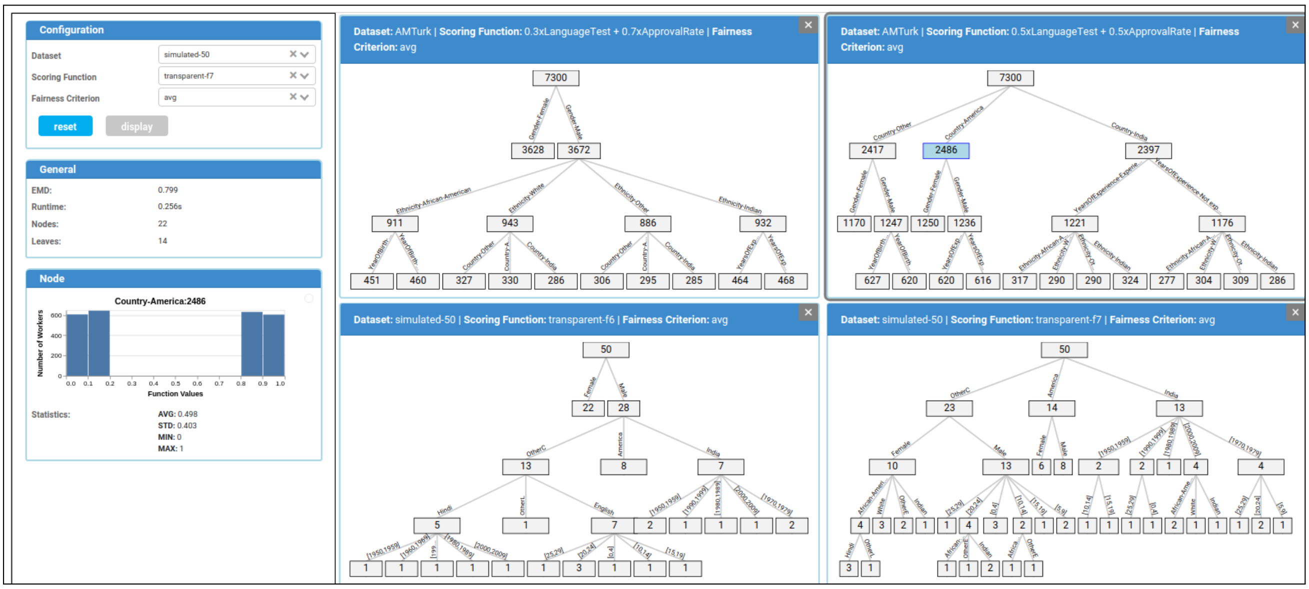
3.2 Algorithm

Our optimization problem for finding the most unfair partitioning is hard since there are many possible partitionings P (exponential in the number of protected attribute values). For this reason, we propose an efficient heuristic algorithm to identify a partitioning of individuals with respect to our optimization objective within reasonable time. Our algorithm (pseudocode given as Algorithm 1) is *recursive*. We describe it with one unfairness formulation (the worst-case one provided in Equation 1 and with one aggregation function, namely average). Our algorithm decides whether or not to split a given partition by comparing the average EMD of that partition with its siblings to that of its children with its siblings. The intuition behind this is that it assesses what would happen to unfairness as measured by the average EMD if the partition was replaced by its children. It only splits a partition if its average pairwise EMD with its siblings is less than the average pairwise EMD of its potential children with the partition's siblings (that is in the case of the worst-case formulation of unfairness - other formulations require to change this test only). To invoke the algorithm for the first time, we first split the given set of individuals using the most unfair attribute and then the algorithm is called once for each resulting partition. After all recursive calls of the algorithm terminate, the output is returned as the final partitioning of the individuals.

4 DEMONSTRATION SCENARIOS

FaiRank caters to different user roles. A screenshot of the interface is shown in Figure 3. **A video of the demonstration is available at https://youtu.be/MckMJColcDk.** We propose to demonstrate it with 3 scenarios, one per role. During the demonstration, we will rely on two types of datasets, simulated datasets mimicking crowdsourcing platforms and real-data crawled from online freelancing marketplaces. In each case, we will explore various scoring functions representing different jobs as well as variants for the same job. We will also allow the exploration of transparency settings and their effect on fairness quantification, by making use of the ARX tool to k-anonymize the datasets[3] and by considering cases where the scoring function is available and cases where it is not.

[3] https://arx.deidentifier.org/



Configuration

Dataset: simulated-50

Scoring Function: transparent-f7

Fairness Criterion: avg

General

EMD: 0.799

Runtime: 0.256s

Nodes: 22

Leaves: 14

Node

Statistics: AVG: 0.498 STD: 0.403 MIN: 0 MAX: 1

Dataset: AMTurk | **Scoring Function:** 0.3xLanguageTest + 0.7xApprovalRate | **Fairness Criterion:** avg

Dataset: AMTurk | **Scoring Function:** 0.5xLanguageTest + 0.5xApprovalRate | **Fairness Criterion:** avg

Dataset: simulated-50 | **Scoring Function:** transparent-f6 | **Fairness Criterion:** avg

Dataset: simulated-50 | **Scoring Function:** transparent-f7 | **Fairness Criterion:** avg

Figure 3: A snapshot of FaiRank's interface. The Configuration box on the left allows users to choose which dataset and which scoring functions they want to explore. It allows them to also choose a fairness criterion. The partitioning trees are displayed on the right in multiple panels, which allows the user to compare multiple scoring functions/datasets. General information about a partitioning tree can be found in the General box on the left, and the user can view statistics about a particular partition in the Node box by clicking on that partition in the tree.

AUDITOR Scenario. This scenario provides auditors with the ability to monitor a marketplace that offers multiple jobs, each with its own scoring function. It provides a big picture to auditors and lets them identify which jobs are most unfair to which individuals based on their rankings and under different notions of fairness. For instance, an auditor may be looking to draft a "fairness" report on a freelancing marketplace such as Qapa or TaskRabbit. The auditor would want to quantify the fairness for each job offered on the platform, and identify demographics groups that are least/most favored on the platform by each job. Additionally, the auditor might consider cases where the marketplace does not provide full transparency, either in terms of attributes of its users or in terms of the scoring functions used to rank those users, and we show the effect of this on quantifying fairness compared to the case when both attributes and the scoring function are available.

JOB OWNER Scenario. This scenario emphasizes the ability to define different scoring functions, and examine their impact on individuals. This exploration will help owners understand the behavior of their scoring functions and will guide them to choose the best function for their job, i.e., the one that satisfies some desired fairness. For instance, for an online job that requires people to write code, the owner can select those for whom the scoring function induces the least unfairness.

END-USER Scenario. This scenario offers end-users the ability to immerse themselves and simulate different cases in which they are to be ranked. For instance, an end-user wishing to find a job online, can examine how unfair some job is with respect to different groups of people. Given a group to which the end-user belongs (e.g., Young professionals in Grenoble) and a job of interest (e.g., installing wood panels), the end-user can see how well the marketplace is treating that group and make an informed decision of whether to target that job or not.

ACKNOWLEDGMENTS

This work is supported by the American University of Beirut Research Board (URB)

REFERENCES

[1] A. J. Biega, K. P. Gummadi, and G. Weikum. Equity of attention: Amortizing individual fairness in rankings. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval, SIGIR 2018*, pages 405–414, 2018.

[2] T. Calders and S. Verwer. Three naive bayes approaches for discrimination-free classification. *Data Mining and Knowledge Discovery*, 21(2):277–292, Sep 2010.

[3] S. A. Friedler, C. Scheidegger, and S. Venkatasubramanian. On the (im)possibility of fairness. *CoRR*, abs/1609.07236, 2016.

[4] S. Hajian, F. Bonchi, and C. Castillo. Algorithmic bias: From discrimination discovery to fairness-aware data mining. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 2125–2126, 2016.

[5] A. Hannak, C. Wagner, D. Garcia, A. Mislove, M. Strohmaier, and C. Wilson. Bias in online freelance marketplaces: Evidence from taskrabbit and fiverr. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing, CSCW 2017, Portland, OR, USA, February 25 - March 1, 2017*, pages 1914–1933, 2017.

[6] M. J. Kearns, S. Neel, A. Roth, and Z. S. Wu. Preventing fairness gerrymandering: Auditing and learning for subgroup fairness. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 2569–2577, 2018.

[7] S. K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data mining and knowledge discovery*, 2(4):345–389, 1998.

[8] O. Pele and M. Werman. Fast and robust earth mover's distances. In *2009 IEEE 12th International Conference on Computer Vision*, pages 460–467. IEEE, September 2009.

[9] A. Singh and T. Joachims. Fairness of exposure in rankings. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, pages 2219–2228, 2018.

[10] M. B. Zafar, I. Valera, M. Gomez-Rodriguez, and K. P. Gummadi. Fairness constraints: Mechanisms for fair classification. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*, pages 962–970, 2017.

[11] I. Zliobaite. A survey on measuring indirect discrimination in machine learning. *CoRR*, abs/1511.00148, 2015.

MM-evolver: A Multi-Model Evolution Management Tool

Demo Paper

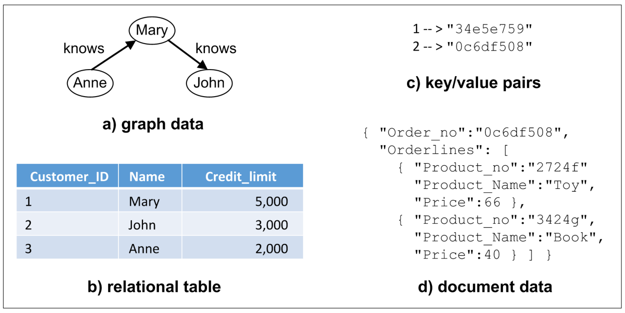
Michal Vavrek
Department of Software
Engineering, Charles University
Prague, Czech Republic
vavrekmichal@gmail.com

Irena Holubová
Department of Software
Engineering, Charles University
Prague, Czech Republic
holubova@ksi.mff.cuni.cz

Stefanie Scherzinger
OTH Regensburg
Regensburg, Germany
stefanie.scherzinger@
oth-regensburg.de

ABSTRACT

There is a new generation of databases specifically addressing *Big Data Variety*: *Multi-model* databases store and process structurally heterogeneous data, managing several data models in one integrated backend. Yet one of the many challenges these systems face is evolution management. In our demonstration, we present our prototype implementation of a tool called *MM-evolver*. *MM-evolver* carries out user-triggered schema modification operations over a multi-model database, and propagates them across all models. As a novelty, *MM-evolver* supports both inter- and intra-model schema modification operators. To the best of our knowledge, ours is the first tool addressing evolution management in the world of multi-model databases.



```
1 --> "34e5e759"
2 --> "0c6df508"
```

Customer_ID	Name	Credit_limit
1	Mary	5,000
2	John	3,000
3	Anne	2,000

```
{ "Order_no":"0c6df508",
  "Orderlines": [
    { "Product_no":"2724f"
      "Product_Name":"Toy",
      "Price":66 },
    { "Product_no":"3424g",
      "Product_Name":"Book",
      "Price":40 } ] }
```

Figure 1: A multi-model data scenario [6]

1 INTRODUCTION

In recent years, the Big Data movement has broken down the borders of many technologies and approaches that have so far been acknowledged as mature and robust. One of the most challenging issues is the "*Variety*" of Big Data. Data may be present in various types and formats – structured, semi-structured, and unstructured – and produced by different sources, and hence natively have various models.

The challenge of handling variety has inspired a new generation of dedicated *multi-model databases* (MMDs), capable of storing and processing structurally different data, by supporting several data models in a single DBMS having a unified query language and API. The MMD way of solving the polyglot persistence problem offers advantages in data modeling, allowing to represent data in its most native model. This can be considered as opposite to the "*One Size Does Not Fit All*" argument [10]. However, it can be also understood as a way of re-architecting traditional database models to address new requirements [4]. Nothing shows the picture better than the Gartner Magic quadrant for operational database management systems [3], which (correctly) assumed that, by 2017, the majority of leading DBMSs will offer multiple data models in a single DBMS platform.

To illustrate the challenge of multi-model data management, consider the simple example depicted in Figure 1. There we have data with four distinct data models. Customer data is stored in a relational table – their ID, name, and credit limit. Graph data bear information about mutual relationships between the customers, i.e., who knows whom. In JSON documents, each order has an ID and a sequence of ordered items, each of which includes product number, name, and price. The fourth type of data, key/value pairs, bears a relationship between customers (or rather, their IDs) and orders (or rather, their IDs).

One of the many challenges [6] these systems are facing is evolution management. As user requirements change, the data structures evolve, and, consequently, also the respective storage strategy, queries etc. This problem is challenging even in the world of single-model databases. In simpler applications, we can rely on a skilled DB administrator, but in more complex situations it is a difficult and error-prone task. In addition, we can observe contradictory approaches to this problem in different types of DBMSs. In the world of traditional relational or XML databases, the evolution of data structures requires immediate changes in the schema. With NoSQL systems, we can (to some extent) exploit the schemalessness and ability to store data with similar, but not necessarily the same structure. Considering the existence of a schema, another complication is existence of schema-full, schema-less and even schema-mixed MMDs. Consequently, the problem of evolution management in MMDs is much more complex.

Consider again Figure 1. We may want to add a new property to one of the models (e.g., an address to JSON documents representing orders), which does not affect the other models. But, later we may decide to move this property to another model (e.g., to represent addresses in the relational model instead). Hence, we need to change data in both models. In addition, there might already exist a reference to the modified property, which then needs to be updated accordingly.

To address the indicated problems, we extend our previous research results [7, 8] for single-model systems (XML or relational) or systems with closely related models (namely aggregate-oriented NoSQL). In this demonstration, we present a tool called *MM-evolver*, which carries out user-required changes over a multi-model schema and propagates them across all sub-models. To the best of our knowledge, this is the first solution addressing the problem of evolution management in the world of multi-model databases. We see it as the first step towards a unification of evolution management across multiple models.

In the remainder of this paper, we introduce the ideas implemented in *MM-evolver* and outline our demonstration.

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.62

2 MULTI-MODEL EVOLUTION MANAGEMENT

There are two main approaches to supporting different models:

- **Complex engine** (e.g., CouchBase [2]) – The DBMS transforms all supported data types to a single core model. Its engine has to pre-process and map all operations to the core model.
- **Layer-based architecture** (e.g., Oracle Database 12c [1]) – The DBMS supports different models via different layers on top of an engine. Data are stored in the relevant model. Each data model has its own component which communicates with the engine.

We focus on the layer-based architecture, which is used in a significant portion of existing MMDs, because there is no need to introduce a generic approach for specific complex engines, since they often internally map all supported models onto a single model. Figure 2 shows two main layers of the layer-based approach inspired by the principles of the Model Driven Architecture: *model-specific* and *model-independent*[1]. For the sake of simplicity we assume that the data in the individual models have a schema. However, such a schema does not have to be explicitly defined. It can be a kind of an agreed structure, as often used in practice. The engines in the model-specific layer can thus differ also with regards to this aspect.

The MMD engine is located in the model-independent layer. It is a facade for functions of the database, such as queries, and distributes queries and commands to the respective individual models. Also, it collects data from them and creates the final result for the user.

2.1 Database Schema Evolution Language

Our aim is a general solution for schema evolution in MMDs and the following models as the most common representatives: (1) relational, (2) column, (3) graph, (4) key/value, and (5) document (i.e., JSON or XML). By the generic term *kind*, we refer to an abstract label that describes or groups related records. In the relational model, this corresponds to a table. Some MMD vendors use the terms *class* (as in OrientDB[2]) or *collection* (as in ArangoDB[3]).

First, we settle on a common set of operations which can be supported by all models. For this purpose, we extend the work from [9], where the *NoSQL Schema Evolution Language* (NoSQLSEL) covers most of the representatives, namely the aggregate-oriented NoSQL databases, i.e., document, column and key/value models. It involves three basic operations that affect all entities of a given kind to (1) *add* (introducing a new property with a specified default value), (2) *delete* (removing a property), and (3) *rename* (changing the name of a property). It further involves operations to (4) *move* (removing a property from one kind of entity and adding it to another one), and (5) *copy* (copying a property from one kind of entity to another one).

In order to avoid complex extensions of NoSQLSEL towards the missing models, we use a strategy common to most of the existing MMDs [5], i.e., a kind of a unification of the models. For example, we can treat the graph model like ArangoDB, where special edge collections bear information about edges in a graph whose nodes correspond to documents. Similarly, we can assume that entities are represented as rows in a specific table and their properties are columns of the table, where each entity has a unique identification *id*. We call this extension covering the model-specific layer the *Database Schema Evolution Language*.[4]

2.2 Multi-Model Schema Evolution Language

Having a common interface supported by all models in the model-specific layer, we can introduce the *Multi-Model Schema Evolution Language* (MMSEL) which is executed in the model-independent layer. The multi-model engine has to distinguish which models are affected by a given operation and propagate the operations to them. We can divide the operations into two separate groups:

- *Intra-model* operations (i.e., *add*) affect just one model.
- *Inter-model* operations (i.e., *copy*, *move*, *delete* and *rename*) can affect multiple models[5]. The first two can also trigger data transfer between a *source* and a *target* model; the last three can trigger changes in other models, due to references that need to be updated accordingly.

Figure 3 shows the EBNF grammar for the MMSEL syntax and Figure 6 shows an example statement. Intra-model operations, as well as inter-model operations operating within a single model are propagated by the multi-model engine to the specific target model(s) which is/are already able to ensure correct data processing.[6] When entities should be transferred between models (i.e., copied or moved), the multi-model engine gets all entities of the given kind from the source model and inserts them into the target model. In case of operation *move*, it has to delete them from the source model. It is also able to track all cross-model references. When the engine detects a change in a referenced entity, it propagates these changes to the referencing entity.

2.3 Implementation of MMSEL

The core logic of the MMSEL happens in the model independent layer. Internally, MMSEL schema modification operations are translated into a lower-level language, which we introduce next. To distinguish between the models, we introduce the *data model set* $DMS = \{column, document, key/value, graph, relational\}$ and a *model key* δ: $\delta \in DMS$. To create an abstract model of the MMD, we follow the notation from [9] which uses the term *application state* for the current state of the application space. It is a non-persistent application memory. *Database state* is the current state of the database and it represents all stored data.

We need to call specific schema evolution functions in specific models. Consequently, we introduce a modified set of functions called Multi-Model Database Programming Language (MMDPL) which extends the NoSQL Database Programming Language [9] with DMS operations (see Figure 4). The main difference is in operations for getting entities from the database and to save them in the database. Function *empty* does not modify the application space or the database. Despite the original plan of having a common set of operations, we decided to use it for the key/value model, where there is no support for operation *move* since in this model, entities do not have several properties, just a single, opaque value. Otherwise it could be implemented as *rename*. The remaining operations are extended with the DMS, but their logic remains. Rule 7 extends function $put(\delta, \kappa)$ by parameter δ to distinguish where the entity with the key κ is stored. For that purpose we introduce function $model(\kappa)$ which returns a

[1] We borrow from the idea only two layers (levels) and call them slightly different to express our specific context.

[2] https://orientdb.com/

[3] https://www.arangodb.com/

[4] We refer interested readers to the extended version [11] for technical details.

[5] But they can be restricted to intra-model only, like they are in existing MMDs.

[6] We assume re-use of existing single-model change propagation approaches and therefore focus on the novel problem of multi-model change propagation.

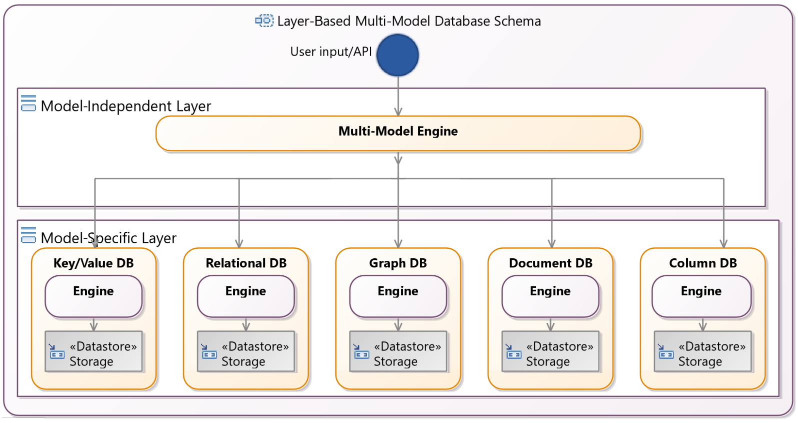


Figure 2: Architecture of a layer-based MMD

```
mmevolutionop ::= add | delete | rename | move | copy;

add    ::= "add" property "=" value [selection];
delete ::= "delete" property [selection];
rename ::= "rename" property "to" pname [selection];
move   ::= "move" property "to" kind [complexcond];
copy   ::= "copy" property "to" kind [complexcond];

selection   ::= "where" conds;
complexcond ::= "where" (joincond | conds |
                    (joincond "and" conds));

joincond ::= property "=" property;
conds    ::= cond {"and" cond};
cond     ::= property "=" value;

property ::= mname "." kname ["." pname];
kind     ::= mname "." kname;
mname    ::= identifier;
kname    ::= identifier;
pname    ::= identifier;
```

Figure 3: EBNF syntax of MMSEL

model where the entity occurs. We use this approach to detect the affected model in all modified functions. In Rule 8 we extend function $delete(\delta, \kappa)$ by key of the model δ which contains the entity with key κ. Rules 9, 10, and 11 add parameter δ to function get. All modified functions get load entity/entities from the specified model by key δ to the application space. Rule 12 is also extended by the model key δ to load the property from the specified model.

2.4 Reference Evolution in MMDs

We next discuss how referential integrity is maintained as schema modification operations are carried out.[7] In the first version of our solution, we consider the reference simply as a pointer from a property of a referencing entity to a property of a referenced entity. We describe the source or target of a reference by a triple *(model, entity, property)*. A reference is then represented as a pair $(source, target)$ and we assume that at least one model is able to store the pairs in a *reference space*.

Next we can divide operations into two groups: *Safe* operations (i.e., *add* and *copy*) do not trigger any reference updates, whereas *unsafe* operations (i.e., *delete*, *rename*, and *move*) can.

To avoid complex extensions and instead stay within the MMDPL framework, we internally represent a reference as a special type of entity. It has three properties: (1) the referenced property of the entity, (2) the key of the property in the MMD, and (3) an array of triples describing entities referencing it. Each triple in the array consists of three properties: a model, a kind, and a property.

During our analysis of reference migration, we discovered that WHERE conditions make the solution much more difficult. It is caused by the nature of MMDs which allow a user to move a subset of the properties. This behavior can split, delete or move completely an existing reference based on the affected set of the values. We introduce a solution for operations without WHERE conditions and keep it as an open challenge.

Another point to discuss is the behavior when a referenced property is removed. We have two options what can happen with the referencing property: (1) set to a default value, (2) delete the property. We decided to use the second approach, because it is a clear solution for the used models. (The first approach has to define what should be the behavior when a MMD contains an entity without a referencing property, as well as default values for all models. Also, the default value can be considered as a value of the property in an application so it can be confusing.)

The next step defines operations for creating and managing references in the MMDPL. We need to create a reference, store it, remove it and find it. Let *reference store model* (RSM) be a store which is able to persist the reference entities. We use the RSM to extend the MMDPL and define functions which help us to implement reference management in the MMSEL. Figure 5 shows the extension of the MMDPL which provides functions for the mentioned operations. We introduced a special type of entities for the references which is stored in the RSM, but we work with the type in the same way as with other database entities.

3 DEMONSTRATION OUTLINE

For the purpose of experimental evaluation of the above described ideas, we have implemented a first prototype called *MM-evolver*. The application is based on the .NET framework and is written in C#. To be able to experiment in the future with various models, not just those provided by a particular MMD, we created an *abstract* layer-based model, where each particular model can be represented by a separate DBMS. To test also this feature, in the first version we use MongoDB[8] and MariaDB[9] representing the document model and the relational model.

In our demo of *MM-evolver*, we build two use cases – one around real-world data, which is based on the Internet Movie Database (IMDb)[10], and the other on the multi-model benchmark UniBench[11]. As shown in Figure 6 for the first case, the data is stored both in the document model (such as `movies.Contributor` to the left), as well as in the relational model (`name_basics` to the right). To the top, we show an inter-model schema modification operation that we are about to execute. We further highlight the affected data (relational in red, document in blue). In interaction with our audience, we will gradually evolve the database state, simulating realistic demands. We intend to make the benefits of the declarative language evident, so that attendees get a clear picture how they would use it in practice. For each supported operation (both intra-model and inter-model), we demo:

(1) the state of the database before and after the change,

[7] Note that the earlier language NoSQLSEL does not consider references, because most NoSQL databases do not support them. Maintaining referential integrity is therefore another new contribution of *MM-evolver*.

[8] https://www.mongodb.com/

[9] https://mariadb.org/

[10] https://www.imdb.com/

[11] http://udbms.cs.helsinki.fi/?projects/ubench

Let dms be a DMS, ds be a database state, as be an application state, δ be a model key, and Ω be a data model. Let κ, κ' be entity keys. Let n, n' be property names, and let v be a property value. Symbol $\perp$ denotes an undefined value. Let π, π' be properties, i.e., mappings from property names to property values. *kind* : *Keys* $\mapsto$ *Kind* is a function that extracts the entity kind from a key. *model* : *Keys* $\mapsto$ *Data model* is a function that extracts the entity model from a key. Θ is a conjunctive query, and c is a string constant.

$$\llbracket empty() \rrbracket(dms, ds, as) = (dms, ds, as) \tag{1}$$

$$\llbracket new(\kappa) \rrbracket(dms, ds, as) = (dms, ds, as[\kappa \mapsto \emptyset]) \tag{2}$$

$$\llbracket new(\kappa, \pi) \rrbracket(dms, ds, as) = (dms, ds, as[\kappa \mapsto \pi]) \tag{3}$$

$$\llbracket setProperty(\kappa, n, v) \rrbracket(dms, ds, as \cup \{\kappa \mapsto \pi\}) = (dms, ds, as \cup \{\kappa \mapsto (\pi[n \mapsto v])\}) \tag{4}$$

$$\llbracket setProperty(\kappa, n, \kappa') \rrbracket(dms, ds, \quad as \cup \{\kappa \mapsto \pi\} \cup \ \{\kappa' \mapsto \pi'\}) = (dms, ds, as \cup \{\kappa \mapsto (\pi[n \mapsto \pi'])\} \cup \{\kappa' \mapsto \pi'\}) \tag{5}$$

$$\llbracket removeProperty(\kappa, n) \rrbracket(dms, ds, as \cup \{\kappa \mapsto \pi\}) = (dms, ds, as \cup \{\kappa \mapsto (\pi[n \mapsto \perp])\}) \tag{6}$$

$$\llbracket put(\delta, \kappa) \rrbracket(dms \cup \{\delta \mapsto \Omega\}, ds, as \cup \{\kappa \mapsto \pi\}) = (dms \cup \{\delta \mapsto \Omega\}, ds[\{\kappa \mapsto \pi \mid model(\kappa) = \delta\}], as \cup \{\kappa \mapsto \pi\}) \tag{7}$$

$$\llbracket delete(\delta, \kappa) \rrbracket(dms \cup \{\delta \mapsto \Omega\}, ds, as) = (dms \cup \{\delta \mapsto \Omega\}, ds[\{\kappa \mapsto \perp \mid model(\kappa) = \delta\}], as) \tag{8}$$

$$\llbracket get(\delta, \kappa) \rrbracket(dms, ds, as) = (dms \cup \{\delta \mapsto \Omega\}, ds, as \cup [\{\kappa \mapsto \pi \mid \kappa \mapsto \pi \in ds \wedge model(\kappa) = \delta\}]) \tag{9}$$

$$\llbracket get(\delta, kind = c) \rrbracket(dms, ds, as) = (dms \cup \{\delta \mapsto \Omega\}, ds, as[\{\kappa \mapsto \pi \mid \kappa \mapsto \pi \in ds \wedge kind(\kappa) = c \wedge model(\kappa) = \delta\}]) \tag{10}$$

$$\llbracket get(\delta, kind = c \wedge \Theta) \rrbracket(dms, ds, as) = (dms \cup \{\delta \mapsto \Omega\}, ds, as[\{\kappa \mapsto \pi \mid \kappa \mapsto \pi \in ds \wedge kind(\kappa) = c \wedge model(\kappa) = \delta \wedge \llbracket \Theta \rrbracket(\kappa \mapsto \pi)\}]) \tag{11}$$

$$\llbracket getProperty(\delta, \kappa, n) \rrbracket(dms \cup \{\delta \mapsto \Omega\}, ds, as \cup \{\kappa \mapsto (\{n \mapsto v\} \cup \pi) \mid \kappa \mapsto (\{n \mapsto v\} \cup \pi) \in ds \wedge model(\kappa) = \delta\}) = v \tag{12}$$

Figure 4: The commands for interfacing with the multi-model database

Let rs be a RSM, ρ_1,ρ_2 be keys in rs and η_1 its set of properties. Let dms be a DMS, ds be a database state, as be an application state. Let κ_1, κ_2 be entity keys. Let n_1 be property name. Let δ be a model key, c be a kind and v be an array of triples of *m*, *k*, and *p*. Symbol $\perp$ denotes an undefined value. Let π_1 be properties, i.e., mappings from property names to property values. key : *RSM keys* $\mapsto$ *model keys* is a function that extracts the entity key from a reference store model key.

$$\llbracket \mathit{newReference}(\kappa_1) \rrbracket(dms, ds, as, rs) = (dms, ds, as[\{\rho_1 \mapsto \perp \mid key(\rho_1) = \kappa_1\}], rs) \tag{13}$$

$$\llbracket \mathit{putReference}(\rho_1) \rrbracket(dms, ds, as \cup \{\rho_1 \mapsto \eta_1\}, rs) = (dms, ds, as \cup \{\rho_1 \mapsto \eta_1\}, rs[\rho_1 \mapsto \eta_1]) \tag{14}$$

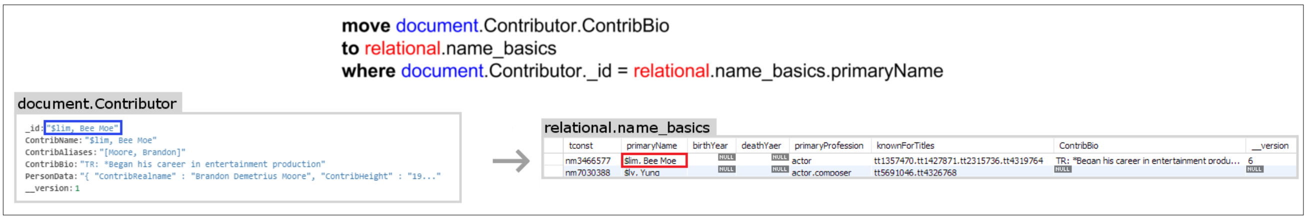
$$\llbracket \mathit{getReference}(\kappa_1) \rrbracket(dms, ds, as, rs) = (dms, ds, as[\{\rho_1 \mapsto \eta_1 \mid \rho_1 \mapsto \eta_1 \in rs \wedge key(\rho_1) = \kappa_1\}], rs) \tag{15}$$

$$\llbracket \mathit{getReferencedBy}(\delta, c_1, n_1) \rrbracket(dms, ds, as, rs) = (dms, ds, as[\{\rho_1 \mapsto \eta_1 \mid \rho_1 \mapsto \eta_1 \in rs \wedge \{''s'', v\} \in \eta_1 \wedge \{''m'' : \delta,'' k'' : c,'' p'' : n_1\} \in v\}], rs) \tag{16}$$

$$\llbracket \mathit{deleteReference}(\kappa_1) \rrbracket(dms, ds, as, rs) = (dms, ds, as, rs[\{\rho_1 \mapsto \perp \mid key(\rho_1) = \kappa_1\}]) \tag{17}$$

$$\llbracket \mathit{renameReference}(\kappa_1, \kappa_2) \rrbracket(dms, ds, as, \quad rs \cup \{\rho_1 \mapsto \eta_1 \mid \rho_1 \mapsto \eta_1 \wedge key(\rho_1) = \kappa_1\}) = (dms, ds, as, rs[\{\rho_2 \mapsto \eta_1 \mid key(\rho_2) = \kappa_2\}]) \tag{18}$$

Figure 5: Dedicated commands for manipulating references in the multi-model database



```
move document.Contributor.ContribBio
to relational.name_basics
where document.Contributor._id = relational.name_basics.primaryName
```

document.Contributor

```
_id:"$lim, Bee Moe"
ContribName:"$lim, Bee Moe"
ContribAliases:"[Moore, Brandon]"
ContribBio:"TR: *Began his career in entertainment production"
PersonData:"{ "ContribRealname" : "Brandon Demetrius Moore", "ContribHeight" : "19..."
__version:1
```

relational.name_basics

tconst	primaryName	birthYear	deathYaer	primaryProfession	knownForTitles	ContribBio	__version
nm3466577	$lim. Bee Moe	NULL	NULL	actor	tt1357470.tt1427871.tt2315736.tt4319764	TR: *Began his career in entertainment produ...	6
nm7030388	$lv. Yung	NULL	NULL	actor.composer	tt5691046.tt4326768	NULL	NULL

Figure 6: Carrying out an inter-model schema modification operation in *MM-evolver*

(2) the number of affected entities, i.e., those changed during the execution of an operation,
(3) the number of targeted entities, i.e., those that correspond to the change request, and
(4) the generated code which propagates these changes.

Interested attendees can experiment with *MM-evolver*, issuing their own operations and thus evaluating our approach.

Acknowledgements and Remarks

This paper is based on Michal Vavrek's Master thesis [11]. In part, this project was funded by the MŠMT ČR project SVV 260451 and by the *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation), grant #385808805.

REFERENCES

[1] Bob Bryla and Kevin Loney. 2013. *Oracle Database 12C The Complete Reference* (1st ed.). McGraw-Hill Osborne Media. 1472 pages.

[2] Couchbase 2018. https://www.couchbase.com/ [Online; Accessed 2-February-2018].

[3] Donald Feinberg, Merv Adrian, Nick Heudecker, Adam M. Ronthal, and Terilyn Palanca. [n. d.]. *Magic Quadrant for Operational Database Management Systems.* Gartner. 12 October 2015.

[4] Zhen Hua Liu and Dieter Gawlick. 2015. Management of Flexible Schema Data in RDBMSs – Opportunities and Limitations for NoSQL. In *CIDR '15*. Online Proceedings, Asilomar, California, USA.

[5] Jiaheng Lu and Irena Holubová. 2017. Multi-model Data Management: What's New and What's Next?. In *EDBT '17*. OpenProceedings, Venice, Italy, 602–605.

[6] Jiaheng Lu, Irena Holubová, and Bogdan Cautis. 2018. Multi-model Databases and Tightly Integrated Polystores: Current Practices, Comparisons, and Open Challenges. In *CIKM '18*. ACM, Torino, Italy, 2301–2302.

[7] Martin Nečaský, Jakub Klímek, Jakub Malý, and Irena Mlýnková. 2012. Evolution and Change Management of XML-based Systems. *J. Syst. Softw.* 85, 3 (March 2012), 683–707.

[8] Stefanie Scherzinger, Thomas Cerqueus, and Eduardo Cunha de Almeida. 2015. ControVol: A framework for controlled schema evolution in NoSQL application development. In *ICDE '15*. IEEE Computer Society, Seoul, South Korea, April 13-17, 1464–1467.

[9] Stefanie Scherzinger, Meike Klettke, and Uta Störl. 2013. Managing Schema Evolution in NoSQL Data Stores. In *DBPL'13*. arXiv, Riva del Garda, Trento, Italy.

[10] Michael Stonebraker and Ugur Cetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE '05*. IEEE Computer Society, Washington, DC, USA, 2–11.

[11] M. Vavrek. 2018. *Evolution Management in NoSQL Document Databases.* Master Thesis. Charles University in Prague, Czech Republic. http://www.ksi.mff.cuni.cz/~holubova/dp/Vavrek.pdf.

Resense: Transparent Record and Replay of Sensor Data in the Internet of Things

Dimitrios Giouroukis Julius Hülsmann Janis von Bleichert Morgan Geldenhuys
Tim Stullich Felipe Oliveira Gutierrez Jonas Traub Kaustubh Beedkar Volker Markl
Technische Universität Berlin & DFKI
firstname.lastname@tu-berlin.de

ABSTRACT

As the scientific interest in the Internet of Things (IoT) continues to grow, emulating IoT infrastructure involving a large number of heterogeneous sensors plays a crucial role. Existing research on emulating sensors is often tailored to specific hardware and/or software, which makes it difficult to reproduce and extend. In this paper we show how to emulate different kinds of sensors in a unified way that makes the downstream application agnostic as to whether the sensor data is acquired from real sensors or is read from memory using emulated sensors. We propose the Resense framework that allows for replaying sensor data using emulated sensors and provides an easy-to-use software for setting up and executing IoT experiments involving a large number of heterogeneous sensors. We demonstrate various aspects of Resense in the context of a sports analytics application using real-world sensor data and a set of Raspberry Pis.

1 INTRODUCTION

The growth of the Internet of Things (IoT) has led to many disruptive technologies and applications such as smart homes, autonomous vehicle fleets, health and well being, personal and home security, and natural disaster management. In such applications, the IoT data (i.e., sensor data generated by devices connected to the Internet) may get very large and may involve billions of sensors [6]. Therefore, efficient means to automatically collect, store, and analyze massive amounts of IoT data plays an important role in our modern information-based society.

IoT research connects two communities: The database community deals with sensor data acquisition [3, 12, 14] and distributed query processing [8, 15]. The infrastructure and networking community deals with network connections, application and resource management across sensor nodes, and cloud computing [4, 9, 10].

For database researchers, it is important to develop and test data management solutions on IoT testbeds which include large numbers of sensor nodes and provide real networking and data processing conditions. However, it is hard to conduct repeatable experiments on such testbeds for two main reasons: *(i)* One needs to fine tune and test different versions of algorithms (A/B or split testing), but sensors data varies over time which leads unequal test conditions. *(ii)* Applications need to be tested on rare events which leads to extremely long test durations. For example, consider a sports analytics application that predicts injuries in real time. Player injuries are rare and highly important, but one cannot repeat them in the real world. Being able to replay sensor data (e.g., data recorded from sensors on players' body) on real test beds solves the issues stated above and enables database researchers to run repeatable experiments.

In this paper we present Resense, a framework which transparently emulates sensors and provides an efficient way to replay and record sensor data. Resense emulates sensors at the operating system level such that it is transparent for applications whether they acquire values from real sensors or from memory using emulated sensors. One can install Resense on testbeds without changing any application and gains the flexibility to switch between live sensor data and replayed sensor data easily.

Resense further provides mechanisms to orchestrate IoT experiments involving a large number of heterogeneous sensors. Our framework allows users to easily configure many different physical and/or emulated sensors as well as the data to be replayed. We provide an intuitive user interface for loading experiment configurations, for deploying experiment data to a large number of sensor nodes, and for starting/stopping experiments on all (or some) sensor nodes. The automatic deployment of the required data, the centralized configuration of sensor nodes, and the centralized control and monitoring of experiments reduce the administrative overhead when running IoT experiments and developing IoT applications.

Our demonstration highlights how easy it can be to setup and run experiments on a real testbed. We show various aspects of Resense using a set of Raspberry Pis as sensor nodes and some physical sensors. In particular, we demonstrate the record and replay functionality where attendees can attach physical sensors to a Raspberry Pi and use the graphical user interface to inspect the (physical) sensor readings, record them, and replay the recorded data. We also demonstrate setting up and executing a full scale IoT experiment based on real-world data from the DEBS 2013 Grand Challenge dataset [13]. The data consists of about 15 000 position events per second which were recorded at a football match in which sensors were embedded on the shoes of the players as well as the ball. Attendees will be able to use the Resense UI to deploy, control, and monitor the experiment and configure the sensor nodes.

In summary, this paper makes the following contributions:

(1) We show how to transparently emulate sensors in order to record and replay sensor data.
(2) We provide an easy to use software for setting up and executing IoT experiments involving large numbers of heterogeneous sensors.
(3) We demonstrate our software by recording and replaying real world sensor data in the context of sports analytics on a set of Raspberry Pis.

Resense is available as open source project[1] and runs on any GNU/Linux system independent of the underlying hardware (e.g., x86-based servers and ARM single-board computers).

The rest of the paper is organized as follows: in Section 2, we give an overview of our approach for transparently emulating sensors and orchestrating IoT experiments. Section 3 gives a

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

[1] https://github.com/TU-Berlin-DIMA/resense

Series ISSN: 2367-2005

10.5441/002/edbt.2019.63

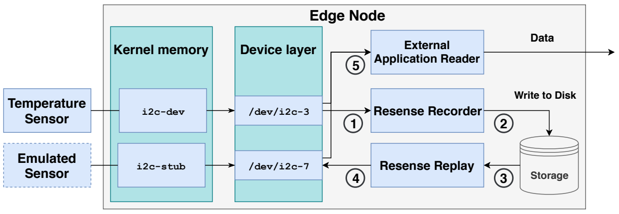


Figure 1: Recording and replaying data with Resense.

description of our demonstration. We discuss related work in Section 4 and conclusion in Section 5.

2 EMULATING SENSORS

Implementing the emulation of sensors while being capable of replaying scientific datasets is hard since the number of sensors that can be involved in a single experiment can vary. On top of this, the heterogeneity of the sensors makes it hard to include every possible sensor architecture in a single emulator. In this section we discuss the decisions behind the architecture of Resense and show how we overcome these obstacles.

2.1 Emulation Abstraction & Replaying Data

The first major contribution of this paper is to emulate a sensor transparently. Transparent emulation allows applications that request data from sensors to treat the emulated sensor as if it was a physical one. The applications cannot distinguish between physical and emulated sensors and expect the same behavior from them. The GNU/Linux kernel provides modules with a stable API and implementations suited for such a task. For example, it provides kernel drivers for commonly used sensor protocols like I2C, SPI, and UART among others. These kernel modules are fully featured drivers that emulate physical devices. We employ their capabilities in Resense in order to provide emulated sensors that are indistinguishable from physical sensors from the point of view of an application.

In more detail, the kernel modules allow the addition and removal of emulated sensors using file descriptors at the device layer of the host operating system. These file descriptors have a consistent naming scheme and are allocated in memory by the kernel module in order to be subsequently perceived as physical components by other parts of the system. Since different file descriptors provide the same API to external applications, it allows Resense to emulate different sensor types regardless of their underlying protocol (and communication bus) for which the host operating system includes its respective kernel module.

Figure 1 shows an end-to-end example from the perspective of a single edge node. A physical temperature sensor is attached to the edge node through an I2C bus. In addition, there is an emulated sensor provided by Resense. The `i2c-dev` kernel module[2] makes all I2C sensors accessible from userspace using character device files. Each device gets assigned a number, starting from 0. The device files follow the pattern of `/dev/i2c-{0,1,2,...}`. In our example, `i2c-dev` has allocated `/dev/i2c-3` for the temperature sensor. The `i2c-stub` kernel module[3], which is also depicted in Figure 1, is responsible for creating character device files for *emulated* devices. The emulated device files follow the same pattern as the ones created from `i2c-dev` and are indistinguishable from the point of view of a userspace application. In our example, the emulated sensor is allocated to `/dev/i2c-7`.

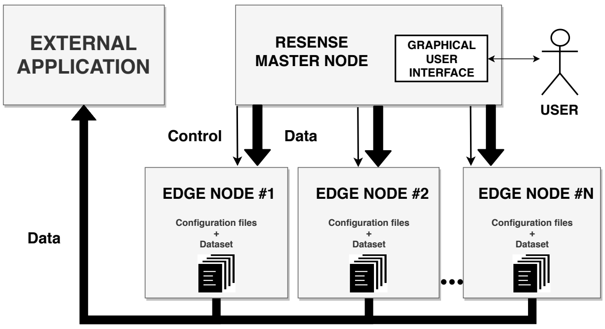


Figure 2: Resense architecture.

Three applications operate on top of physical and emulated sensors in our example: *(i)* The Resense Recorder, *(ii)* Resense Replay, and *(iii)* a reader of an external application. The applications act as follows: The `Resense Recorder` reads sensor values ① and stores them for later use ②. This allows for capturing events in order to replay them later on. The `Resense Replay` module can read the sensor values stored by the recorder ③ and replay them through the emulated sensor ④. We consider this process a *replay* of an experiment. The external application reader can access all sensors (the physical temperature sensor and the emulated sensor) in the same way ⑤.

The `Resense Recorder` and the `Resense Replay` module act as a bridge between the physical sensors of an edge node and the emulated sensors. Any application which consumes sensor data can consume values from both, physical and emulated sensors, since both types of sensors are exposed through device files. Our current approach replays data at the same rate at which they were captured. In future, we plan to support more fine grained control over the rate of recording and replaying. Moreover, our future work also includes time synchronization mechanisms across edge nodes in order to provide reliable and accurate replaying of sensor data. As a remark, the maximum number of physical and emulated sensors per edge node is limited by the host operating system as well as the node's hardware.

2.2 Experiment Orchestration

In order to administrate a huge number of edge nodes and sensors, Resense employs a Master/Slave approach. A visualization of the interaction between master and slave nodes can be seen in Figure 2. Users can control Resense through the graphical user interface of the master node. From there, users can deploy sensor data to sensor nodes, replay and record data, and monitor the progress of experiments. For external applications it is transparent whether they read data which originates from physical or emulated sensors.

Resense stores experiment setups pertaining to applications in configuration files in order to reload and rerun experiments as needed. An experiment consists of sensor data, edge nodes, emulated sensors, and physical sensors. Listing 1 shows an example configuration file. We first define the name of the experiment in Line 1. Starting from Line 2, we provide the list of edge nodes associated with the experiment. Each edge node has a unique id (Line 4), an IP address or domain name (Line 5), and the user name and password for remote access (Lines 6 and 7). Each edge node may host many sensors listed starting from Line 8. Each sensor has a unique id, an output type, and a sensor address. The sensor id refers to a sensor contained in the data file associated with the experiment. In our example, we connect two sensors from the DEBS 2013 grand challenge.

[2] https://www.kernel.org/doc/Documentation/i2c/dev-interface
[3] https://www.kernel.org/doc/Documentation/i2c/i2c-stub

```
    "experiment": "debs−2013",
    "nodes": [
        {
            "edgeId": "edge−1",
            "host": "192.168.1.3",
            "user": "pi",
            "password": "pi",
            "sensors": [
                {"sensorId": "ball", "type": ["GPS"], "address": 3},
                {"sensorId": "referee_left", "type": ["GPS"], "address": 4}
            ]
        },{. . .}
    ]
}
```

Listing 1: An excerpt of sample configuration file.

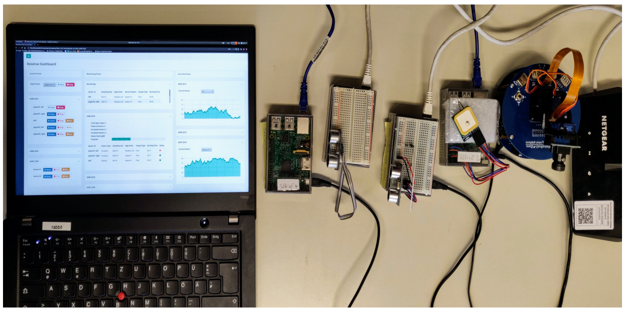


Figure 3: Resense demonstration setup.

In order to run an experiment with Resense, users first create a configuration file through the graphical user interface or load an existing experiment configuration. Resense automatically splits datasets associated with the experiment such that each result file contains the data of one individual sensor declared in the configuration file. The master node deploys the sensor data and the node configuration to each edge node as needed. Each edge node now creates the file descriptors for the emulated sensors according to the configuration provided by the master node. Once the deployment and setup are complete, the user can start the experiment from the dashboard. Edge nodes will then replay sensor data as described in Section 2.1 and shown in Figure 1. Users can also pause, resume, stop, and restart experiments through the dashboard. During execution, Resense monitors the progress of the experiment and displays the status of sensors and edge nodes as well as the currently replayed data.

3 DEMONSTRATION

We demonstrate Resense on a Raspberry Pi testbed as shown in Figure 3. In our setup, the laptop machine acts as the master node, five Raspberry Pis as edge nodes, and some of the edge nodes are equipped to read data from physical sensors (a GPS, an accelerometer, and couple of ultrasonic sensors).

In our demonstration, attendees can record sensor data, replay the recorded data, and set up experiments with data from real-world data sets. Thereby, users can combine physical and emulated sensors installed onto our demonstration platform. Through the Resense dashboard, users can monitor edge nodes and sensors. We further provide an example application which is independent of Resense, reads data from emulated and physical sensors, and streams the live-data to a visualization dashboard.

3.1 Real-World Testdata

As an example, we use the datasets of the DEBS 2013 Grand Challenge [13], which was recorded at a football match and contains the speed, acceleration, and position of the players, the referees, and the balls used during the match. We chose this dataset because it provides sensor values recorded at a high data rate. The tracking frequency is 200Hz for players and of 2000Hz for the ball. Players are tracked with two sensors located on their shoes. Goal keepers are equipped with two additional sensors located at their hands. Overall the dataset provides values from 37 sensors with a total data rate of 15000 position events per second.

3.2 Demonstration Platform

The Raspberry Pi single board computer is our platform of choice for this demo. Its cost effectiveness, its plethora of available connection protocols as well as the fact that it is popular for sensor-based research were the most important factors that lead to our decision. We use Raspberry Pi 3 Model B for our demonstration and the latest version of the operating system that is available, specifically version `2018-10-09` as of January 21, 2019.

For our demo, we choose to focus on the I2C protocol and the `i2c-stub` kernel module to emulate sensors from file descriptors. The protocol supports serial, 8-bit oriented, synchronous, bidirectional data transfer. Synchronization of devices on the bus uses a Master/Slave architecture. Moreover, I2C supports multi-master configurations with collision detection. We chose I2C over the other protocols that are available on a Raspberry Pi board because of the simplicity of the protocol and the maturity of the code of the kernel modules associated with it.

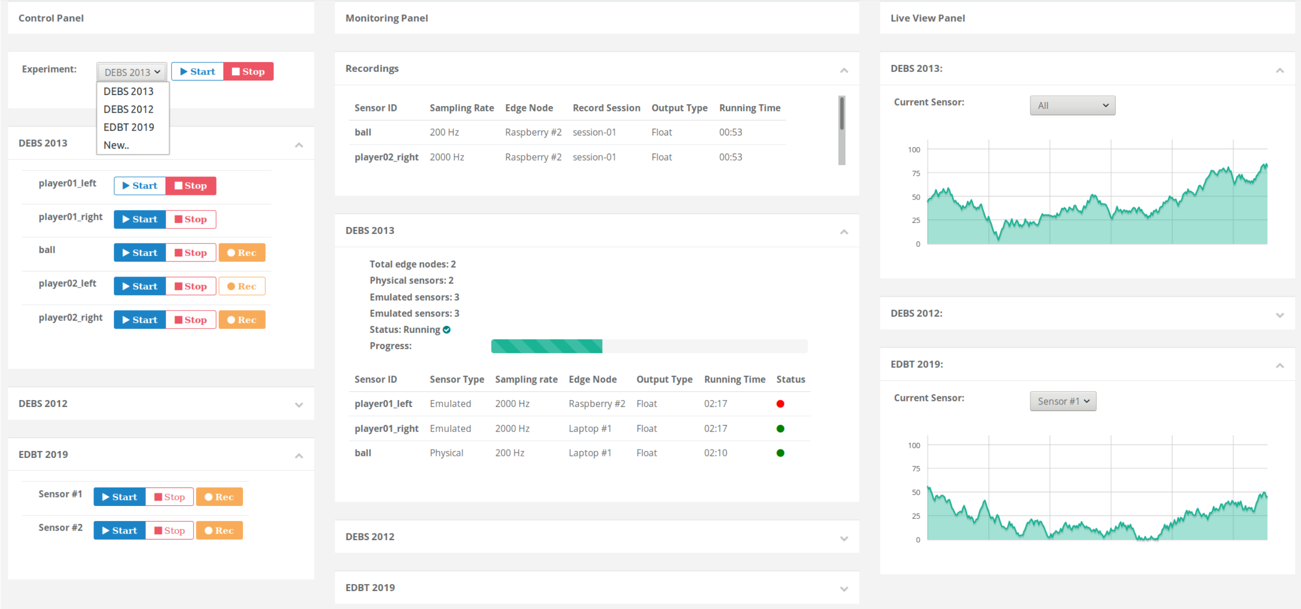
It is worthy of note that Resense is not tied to this demonstration platform. Resense runs on any GNU/Linux based operating system and is agnostic to the hardware architecture. This allows for running experiments with Resense on a large variety of hardware architectures and helps to emulate sensors that lack driver support for multiple architectures.

3.3 Demonstration Scenario

Figure 4 shows a screen shot of Resense's UI. The dashboard has three panels: a control panel on the left, a monitoring panel on the center and a live-view panel on the right.

The control panel allows either for selecting an existing experiment or creating a new one to be run. For each running experiment, the panel shows the list of sensors involved and the available controls. The controls depend on whether the sensor is physical or emulated. Readings from physical sensors can be stopped, resumed, or can be recorded (for replaying later). For emulated sensors, their readings can either be stopped or resumed. By default, all sensors are activated upon starting an experiment. The monitoring panel displays statuses of current recordings and experiments that are running. In particular, the box for recordings shows details about sensors and data that is being recorded. The box for each experiment shows progress, number and type of sensors, number of edge notes, and details about the sensors and the data being read. Finally, the live-view panel shows time series data for sensor readings for currently running experiments and allows to select specific sensor(s) from which data should be rendered.

During the demonstration, the attendees will run an experiment by selecting an existing configuration file (e.g., based on the DEBS 2013 dataset) or by creating a new one. Attendees can interactively control individual sensors and view readings in the live-view panel. Attendees will also be able to create an experiment involving available sensors, record sensor data, and replay recorded data.



Control Panel

Experiment: DEBS 2013

DEBS 2013

Sensor	
player01_left	▶ Start ■ Stop
player01_right	▶ Start ■ Stop
ball	▶ Start ■ Stop ● Rec
player02_left	▶ Start ■ Stop ● Rec
player02_right	▶ Start ■ Stop ● Rec

DEBS 2012

EDBT 2019

Sensor	
Sensor #1	▶ Start ■ Stop ● Rec
Sensor #2	▶ Start ■ Stop ● Rec

Monitoring Panel

Recordings

Sensor ID	Sampling Rate	Edge Node	Record Session	Output Type	Running Time
ball	200 Hz	Raspberry #2	session-01	Float	00:53
player02_right	2000 Hz	Raspberry #2	session-01	Float	00:53

DEBS 2013

Total edge nodes: 2
Physical sensors: 2
Emulated sensors: 3
Emulated sensors: 3
Status: Running ✓
Progress:

Sensor ID	Sensor Type	Sampling rate	Edge Node	Output Type	Running Time	Status
player01_left	Emulated	2000 Hz	Raspberry #2	Float	02:17	🔴
player01_right	Emulated	2000 Hz	Laptop #1	Float	02:17	🟢
ball	Physical	200 Hz	Laptop #1	Float	02:10	🟢

DEBS 2012

EDBT 2019

Live View Panel

DEBS 2013:

Current Sensor: All

DEBS 2012:

EDBT 2019:

Current Sensor: Sensor #1

Figure 4: Resense dashboard

4 RELATED WORK

In the existing literature, Chernysov et al. [1] give an overview on the topic of emulating and simulating IoT infrastructure. According to their work, IoT simulators are categorized into three different sets: *(i)* perceptual emulation, emulates all of the layers of an IoT infrastructure, *(ii)* network emulation, focuses on network properties and *(iii)* application level emulators, emulate workloads only on the application layer. To the best of our knowledge, our work cannot be constrained to only these three types because our implementation is capable of ingesting sensor data at the kernel level of the host operating system. Thus, it is fully transparent to external software that can itself be already in one of these aforementioned categories.

In contrast to the work on IoT simulators [4, 5], our implementation differentiates between the edge nodes and the sensors and focuses on emulating the sensors and reading from sensors. This allows for more granularity while replaying sensor data and can give a more detailed view of the experiment. While other solutions [7] require a fixed full-stack emulation, our work provides a mixed approach since it is able to integrate emulated as well as physical sensors. This makes it easier for testing new physical sensor architectures with existing ones.

Emulating sensors for conducting experiments have also been studied using FPGAs [2, 11]. Usually, FPGAs are first designed using a hardware description language (like VHDL/Verilog) and later implemented into the actual hardware. Our solution reduces the time and the cost involved in prototyping an experiment as it does not depend on any specialized hardware.

5 CONCLUSIONS

In this paper we showed how to emulate sensors to replay recorded sensor data independent of the underlying hardware or software. We presented the Resense framework that allows transparent record and replay of sensor data and provides an easy to use software for setting up and executing IoT experiments. We demonstrated various features of Resense using real world sensor data, a set of Raspberry Pis, and some physical sensors.

Acknowledgments: This work was supported by the German Ministry for Education and Research as Berlin Big Data Center (01IS14013A) and the European Union's Horizon 2020, under the Marie Skłodowska-Curie grant agreement No 765452.

REFERENCES

[1] M. Chernyshev, Z. Baig, O. Bello, and S. Zeadally. 2018. Internet of Things (IoT): Research, Simulators, and Testbeds. *IEEE Internet of Things Journal* 5, 3 (2018), 1637–1647.

[2] Antonio De La Piedra, An Braeken, and Abdellah Touhafi. 2012. Sensor systems based on FPGAs and their applications: A survey. *Sensors* 12, 9 (2012), 12235–12264.

[3] Amol Deshpande, Carlos Guestrin, Samuel R Madden, Joseph M Hellerstein, and Wei Hong. 2004. Model-driven data acquisition in sensor networks. In *VLDB*. 588–599.

[4] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya. 2017. iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. *Software: Practice and Experience* 47, 9 (2017), 1275–1296.

[5] Son N Han, Gyu Myoung Lee, Noel Crespi, Nguyen Van Luong, Kyoungwoo Heo, Mihaela Brut, and Patrick Gatellier. 2015. Dpwsim: A devices profile for web services (DPWS) simulator. *IEEE Internet of Things Journal* 2, 3 (2015), 221–229.

[6] Mark Hung. 2017. Leading the IoT, Gartner Insights on How to Lead in a Connected World. *Gartner Research* (2017), 1–29.

[7] Vilen Looga, Zhonghong Ou, Yang Deng, and Antti Yla-Jaaski. 2012. Mammoth: A massive-scale emulation platform for internet of things. In *CCIS*. 1235–1239.

[8] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst* 30, 1 (2005), 122–173.

[9] Alli Mäkinen, Jaime Jiménez, and Roberto Morabito. 2017. ELIoT: Design of an emulated IoT platform. In *PIMRC*. 1–7.

[10] Mirjana Maksimović, Vladimir Vujović, Nikola Davidović, Vladimir Milošević, and Branko Perišić. 2014. Raspberry Pi as Internet of things hardware: performances and constraints. *Design Issues* 3 (2014), 8.

[11] Eric Monmasson and Marcian N Cirstea. 2007. FPGA design methodology for industrial control systems—A review. *IEEE Trans. on Industrial Electronics* 54, 4 (2007), 1824–1842.

[12] Conor Muldoon, Niki Trigoni, and Greg MP O'Hare. 2011. Combining sensor selection with routing and scheduling in wireless sensor networks. In *VLDB*.

[13] Christopher Mutschler, Holger Ziekow, and Zbigniew Jerzak. 2013. The DEBS 2013 grand challenge. In *DEBS*. 289–294.

[14] Jonas Traub, Sebastian Breß, Tilmann Rabl, Asterios Katsifodimos, and Volker Markl. 2017. Optimized on-demand data streaming from sensor nodes. In *SoCC*. 586–597.

[15] Yong Yao and Johannes Gehrke. 2002. The cougar approach to in-network query processing in sensor networks. *ACM Sigmod Record* 31, 3 (2002), 9–18.

Improving Named Entity Recognition using Deep Learning with Human in the Loop

Ticiana L. Coelho da Silva
Insight Data Science Lab
Fortaleza, Ceara, Brazil
ticianalc@ufc.br

Regis Pires Magalhães
Insight Data Science Lab
Fortaleza, Ceara, Brazil
regismagalhaes@ufc.br

José Antônio F. de Macêdo
Insight Data Science Lab
Fortaleza, Ceara, Brazil
jose.macedo@dc.ufc.br

David Araújo Abreu
Insight Data Science Lab
Fortaleza, Ceara, Brazil
araujodavid@lia.ufc.br

Natanael da Silva Araújo
Insight Data Science Lab
Fortaleza, Ceara, Brazil
natanael_silva@alu.ufc.br

Vinicius Teixeira de Melo
Insight Data Science Lab
Fortaleza, Ceara, Brazil
viniciusteix@alu.ufc.br

Pedro Olímpio Pinheiro
Insight Data Science Lab
Fortaleza, Ceara, Brazil
pedro.olimpio@alu.ufc.br

Paulo A. L. Rego
Federal University of Ceara
Fortaleza, Ceara, Brazil
pauloalr@ufc.br

Aloisio Vieira Lira Neto
Brazilian Federal Highway Police
Fortaleza, Ceara, Brazil
aloisio.lira@prf.gov.br

ABSTRACT

Named Entity Recognition (NER) is a challenging problem in Natural Language Processing (NLP). Deep Learning techniques have been extensively applied in NER tasks because they require little feature engineering and are free from language-specific resources, learning important features from word or character embeddings trained on large amounts of data. However, these techniques are data-hungry and require a massive amount of training data. This work proposes Human NERD (stands for Human Named Entity Recognition with Deep learning) which addresses this problem by including humans in the loop. Human NERD is an interactive framework to assist the user in NER classification tasks from creating a massive dataset to building/maintaining a deep learning NER model. Human NERD framework allows the rapid verification of automatic named entity recognition and the correction of errors. It takes into account user corrections, and the deep learning model learns and builds upon these actions. The interface allows for rapid correction using drag and drop user actions. We present various demonstration scenarios using a real world data set.

1 INTRODUCTION

Named Entity Recognition (NER) is a challenging problem in Natural Language Processing (NLP). It corresponds to the ability to identify the named entities in documents, and label them with one of entity type labels such as person, location or organization. Given the sentence "Trump lives in Washington DC", traditional NER taggers would identify the mentions 'Trump' and 'Washington DC' to person and location labels, respectively. NER is an important task for different applications such as topic detection, speech recognition, to name a few.

However, there is a long tail of entity labels for different domains. It is relatively simple to come up with entity classes that do not fit the traditional four-class paradigm (PER, LOC, ORG, MISC), such as, in Police report documents, *weapon type* is none of the above. For these cases, labeled data may be impossible to find. Even, orthographic features and language-specific knowledge resources as gazetteers are widely used in NER, such approaches are costly to develop, especially for new languages and new domains, making NER a challenge to adapt to these scenarios[10].

Deep Learning models have been extensively used in NER tasks [3, 5, 9] because they require little feature engineering and they may learn important features from word or character embeddings trained on large amounts of data. These techniques from Deep Learning are data-hungry and require a massive amount of training data. While the models are getting deeper and the computational power is increasing, the size of the datasets for training and evaluating are not growing as much [14].

In this work, we address this problem by including humans in the loop. Several methods have been proposed to improve the efficiency of human annotations, for instance in computer vision applications [11, 14] and NER tasks via active learning [2, 7, 8, 12, 13]. Those methods are promising for NER but still leave much room for improvements by assuming the annotation cost for a document measured regarding its length, the number of entities or the number of user annotation actions, for instance. While these are important factors in determining the annotation and misclassification cost, none of them provide the ability to create and incrementally maintain deep learning models based on iterative annotation. Indeed, all of them expect NER tasks to have very few labels. Prodigy [1] is a promising annotation tool that works on entity recognition, intent detection, and image classification. It can help to train and evaluate models faster. However, we could not explore Prodigy, since it is not free. In this work, our goal is to provide an interactive framework called Human NERD (stands for Human Named Entity Recognition with Deep learning) to assist the user in NER classification tasks from creating a massive dataset to building/maintaining a deep learning NER model. Human NERD provides an interactive user interface that allows both the rapid verification of automatic named entity recognition (from a pre-trained deep learning NER model) and the correction of errors. In cases where there are multiple errors, Human NERD takes into account user corrections, and the deep learning model learns and builds upon these actions. The interface allows for rapid correction using drag and drop user actions. We need to point out that our framework consider two types of user: reviewer and data scientist. The reviewer is a domain

Series ISSN: 2367-2005

10.5441/002/edbt.2019.64

expert that can correct possible classification errors and enrich the labels while data scientist focuses on tuning the models. In the next Sections, we provide more details and a screencast is available at YouTube[1].

To the best of the authors' knowledge, this work is the first that simplifies the task of annotating datasets, minimizing supervision, and improving the deep learning NER model, which in turn will make the overall system more efficient. To achieve this end, we first propose the framework in Section 2. Then, we present the demonstration scenarios in Section 3. Section 4 draws the final conclusions.

2 HUMAN NERD

For a large document collection, Human NERD keeps the user engaged in the process of building a large named entity dataset. The input to the framework is a set of documents to annotate and a set of NER labels. The output is a set of annotated documents, a deep learning model for entity recognition and the evaluation metrics values that can estimate the *operational cost* during the annotation time and the *gain* regarding model accuracy.

We incorporate deep learning NER models as the Entity Recognizer models from Spacy [2] framework into Human NERD to reduce the cost of human time dedicated to the annotation process. Indeed, these models have led to a reduction in the number of hand-tuned features required for achieving state-of-the-art performance [6]. Human NERD can also incorporate models such as [3, 5].

Human NERD suggests a potential entity annotation in every interaction loop, and the user as a reviewer can accept or reject individual entities. He/she can also tag a new excerpt from the document text with an entity that was not suggested by the NER model. The feedback is then used to refine the model for the next iteration and enrich the dataset. Our framework simplifies the task of building large-scale datasets, minimizing supervision, and improving the deep learning NER model, which in turn will make the overall system more efficient.

The general workflow of Human NERD follows five main steps (overview in Figure 1): (1) collecting a large set of unlabeled documents; (2) the current NER model recognizes and annotates entities in the document according to labels drawn from a given set of entity classes L (i.e., person, location, among others); (3) user as a reviewer can accept or reject individual entities; he/she can also manually label the document according to L; (4) generating a deep learning NER model for each iteration; (5) estimating the *gain* over the iterations and the *loss*, for improving the model accuracy and the operational cost during annotation time, respectively.

First step. Starting from a large pool of unlabeled documents $T = \{t_1, ..., t_m\}$ collected from different and heterogeneous resources (as Twitter, Wikipedia, Police reports, among others), where t_i is a variable-length sequence of input symbols $t_i = \{w_1, ..., w_n\}$. A sequence of consecutive w_i with the same label λ_j are treated as one mention of such label. Input symbols are word tokens w_i drawn from a given vocabulary V. Let $L = \{\lambda_j : j = 1...q\}$ denotes the finite set of labels in a learning task. We aim at annotating t_i with a sequence of output symbols $Y = \{y_1, ..., y_p\}$. Output symbols are labels λ_j drawn from a given set of entity classes L.

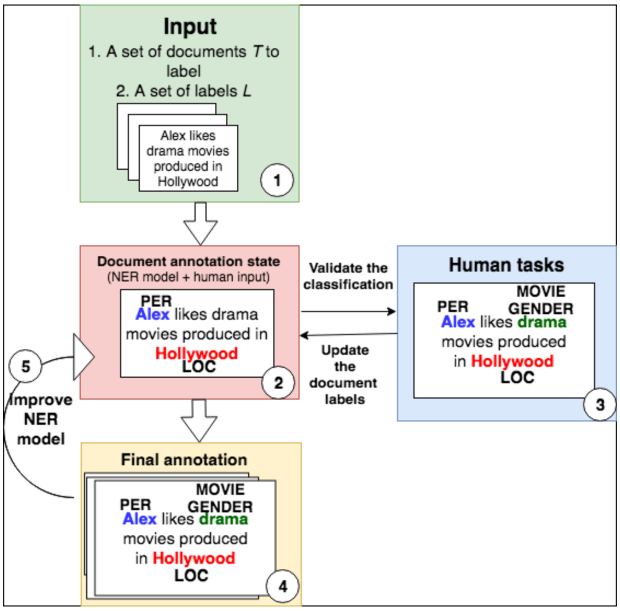

Input
1. A set of documents T to label
2. A set of labels L
Alex likes drama movies produced in Hollywood
1
Document annotation state
(NER model + human input)
PER
Alex likes drama movies produced in Hollywood
LOC
2
Validate the classification
Update the document labels
Human tasks
MOVIE
PER GENDER
Alex likes drama movies produced in Hollywood
LOC
3
5
Improve NER model
Final annotation
MOVIE
PER GENDER
Alex likes drama movies produced in Hollywood
LOC
4

Figure 1: Overview of Human NERD. Given a set of documents for annotating as input, the system alternates between NER model classification and requesting user feedback through human tasks. The outcomes are the documents annotated to improve the NER model.

Second step. Human NERD acquires entity classes (i.e., person, location, among others) for T from a deep learning NER model as [3, 5] or using Spacy's models (i.e., Entity Recognizer model - which is trained using multilinear perceptron and convolutional network). The deep learning model is initially trained with $D = \{(x_i, Y_i) : i = 1...m\}$, a set of labeled training examples x_i, where $Y_i \subseteq L$ the set of labels of the i-th example. At this step, the pre-trained model classifies the entity mentions on $T = \{t_1, ..., t_m\}$ using the labels described on L and outputs $O = (t_i, Y_i) : i = 1...m$, where $t_i \in T$ and $Y_i \subseteq L$ the set of labels of the i-th document.

Third step. Human NERD presents to the user an interactive web-based annotation interface used for adding entity annotations or editing automatic pre-annotations in O. As the entities are labeled in O, users (as reviewers) then accept or reject these to indicate which ones are true. Each document $t_i \in O$ is presented to one user. Thus no two users labeled the same document at the same instant of time. This step outputs O with its user corrections. Human NERD logs both the time elapsed during the labeling process, and the number of labeling *actions* taken for each document t_i, i.e., it keeps track of actions like labeling an entity or removing a label incorrectly assigned to an entity.

Fourth step. Based on the user corrections, the NER model can learn and improve from O. In the interactive interface, the user as a data scientist can demand Human NERD to incrementally update the pre-trained deep NER model or build a new one from O. At this step, the system logs the *accuracy* and *loss* over the iterations of the model construction. These data are useful in the next step.

Fifth step. By putting humans in the loop, Human NERD has a *gain*, since the users help to improve the NER model by validating a new massive training set over time. With such data, we expect to increase the NER model accuracy and decrease its

[1] https://youtu.be/KGJeWKO_3Xw
[2] https://spacy.io/

error in short-term. On the other hand, the *drawback* is the human effort by adding entity annotations or editing pre-annotations (third step). To estimate the framework *loss*, we measure the user efforts for annotating the document regarding its length, number of characters, number of entities or number of user annotation actions (editing or adding new entities). As much the deep NER model learns, Human NERD framework becomes more efficient and minimizes the user supervision.

To measure the agreement between the deep NER model (second step outputs) and the user (third step outputs), Human NERD computes the kappa coefficient (k). Cohen's kappa [4] measures the agreement between two raters who each classifies N items into L mutually exclusive categories.

$$k = \frac{p_0 - p_e}{1 - p_e} \tag{1}$$

such that

$$p_0 = \sum_{i=1}^{|L|} \frac{n_{ii}}{n}; p_e = \frac{1}{n^2} \sum_{i=1}^{|L|} n_{i.} \times n_{.i}; \tag{2}$$

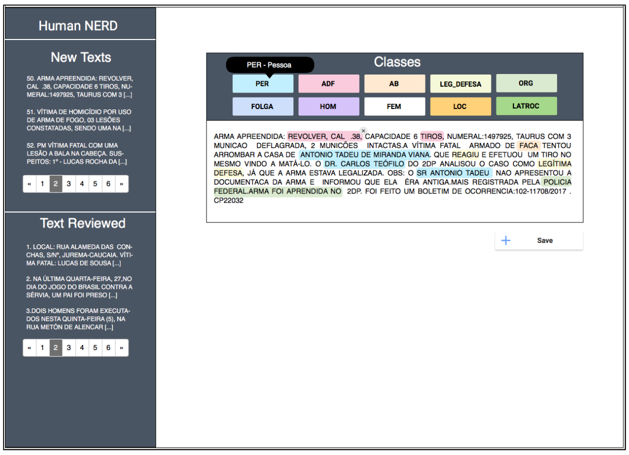
where n_{ii}, $n_{.i}$ and $n_{i.}$ are the number of entities: labelled by the NER model and the user in category i, labelled by the NER model in category i, labelled by the user in category i, respectively. Let n be the total number of entities in T. The closer k is to one, the greater the indication that there is an agreement between the model and the user (as a reviewer). On the other hand, if k is closer to zero, the greater the chance of the agreement be purely random. We expect to increase the NER model accuracy over time and to improve the kappa coefficient to a value as close to one as possible.

3 DEMONSTRATION DETAILS AND SCENARIOS

We cover various scenarios that demonstrate the usefulness of Human NERD. An interactive interface is the access point for the user to: (i) upload several unlabeled documents; (ii) validate each individual entity annotation from the output generated by a pre-computed model executed over those documents; The user can also manually label new entities; (iii) re-build the deep NER model to learn from the annotated documents after the user feedback; and (iv) estimate the *gain* and *loss* regarding the improved NER model and the human efforts.

Human NERD considers two types of user: reviewer and data scientist. The former can perform only the task (ii) described above. The later can perform (i), (iii) and (iv). Moreover, the data scientist can remove or add new labels and texts into L and T, respectively.

To examine the quality of our framework, we use a real dataset with unlabelled texts from Police reports. The dataset contains real-world stories in Portuguese language regarding homicides from Fortaleza city (Brazil). We started by using a pre-computed NER model called Wikiner from Spacy framework which includes only four labels: PER, LOC, ORG, and MISC. We removed MISC and added more than 20 label classes like firearm, melee weapon, wrongful death, among others. After that, the Wikiner model classified the reports according to the labels, and the expert reviewers by means of a web interface added and edited entity annotations. From those reviews, Human NERD created a new deep learning NER model for police domain. This data example confirms that Human NERD can be applied in different contexts and languages.



50. ARMA APREENDIDA: REVOLVER, CAL .38, CAPACIDADE 6 TIROS, NUMERAL:1497925, TAURUS COM 3 [...]

51. VÍTIMA DE HOMICÍDIO POR USO DE ARMA DE FOGO, 03 LESÕES CONSTATADAS, SENDO UMA NA [...]

52. PM VÍTIMA FATAL COM UMA LESÃO A BALA NA CABEÇA. SUSPEITOS: 1° - LUCAS ROCHA DA [...]

1. LOCAL: RUA ALAMEDA DAS CONCHAS, S/N°, JUREMA-CAUCAIA. VÍTIMA FATAL: LUCAS DE SOUSA [...]

2. NA ÚLTIMA QUARTA-FEIRA, 27,NO DIA DO JOGO DO BRASIL CONTRA A SÉRVIA, UM PAI FOI PRESO [...]

3.DOIS HOMENS FORAM EXECUTADOS NESTA QUINTA-FEIRA (5), NA RUA METÓN DE ALENCAR [...]

ARMA APREENDIDA: REVOLVER, CAL .38, CAPACIDADE 6 TIROS, NUMERAL:1497925, TAURUS COM 3 MUNICAO DEFLAGRADA, 2 MUNICÕES INTACTAS.A VÍTIMA FATAL ARMADO DE FACA TENTOU ARROMBAR A CASA DE ANTONIO TADEU DE MIRANDA VIANA. QUE REAGIU E EFETUOU UM TIRO NO MESMO VINDO A MATÁ-LO. O DR. CARLOS TEÓFILO DO 2DP ANALISOU O CASO COMO LEGÍTIMA DEFESA, JÁ QUE A ARMA ESTAVA LEGALIZADA. OBS: O SR ANTONIO TADEU NAO APRESENTOU A DOCUMENTACA DA ARMA E INFORMOU QUE ELA ÉRA ANTIGA.MAIS REGISTRADA PELA POLICIA FEDERAL.ARMA FOI APRENDIDA NO 2DP. FOI FEITO UM BOLETIM DE OCORRENCIA:102-11708/2017 . CP22032

Figure 2: User (as a reviewer) validates the documents annotated by the pre-trained NER model.

It is worth to mention that Human NERD improved Wikiner model with the human help, including its extension for covering new labels and a new data domain. The demonstrated scenarios are as follows.

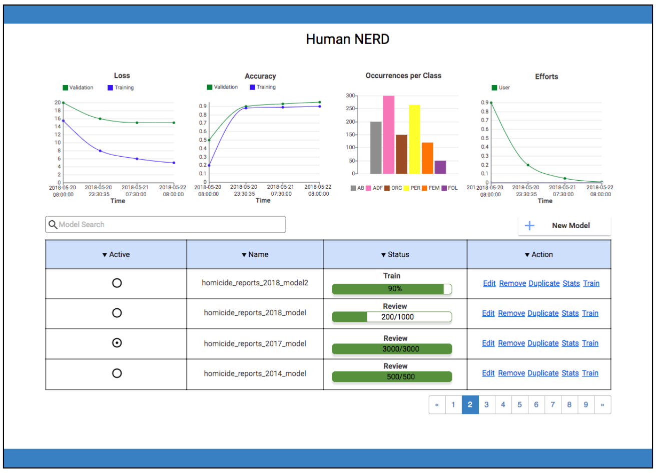
A. Reviewer in the Loop

Human NERD puts the pre-trained model and the human in the loop, so they can actively participate in the process of improving the NER model, using what both know. The model learns as the user goes, based on the labels the user assigns to text excerpts. As shown in Figure 2, the reviewer as a user interacts with the framework through a click-and-drag interface. The framework initially presents a set of documents annotated by the pre-trained model according to a set of label classes L. Each label receives a different color for better visualization.

The reviewer validates one document per time. If he/she agrees with the annotations of the current document (shown in Figure 2), he/she saves the document. Human NERD appends each validated document in a historical dataset, which will be used to improve the deep learning NER model. The reviewer can reject individual entities, in this case, he/she can click on the "x" button. If the reviewer identifies an entity not annotated by the model, he/she can manually label it. In this case, first, he/she should click on the class label (on top of the Figure 2), then the class will appear in evidence. After that, the reviewer selects the sequence of words in the document to annotate. Most annotation tools avoid making any suggestions to the user, to prevent biasing the annotations. Human NERD takes the opposite approach: demands the user to annotate but as little as possible considering that the NER model is continuously improving over time.

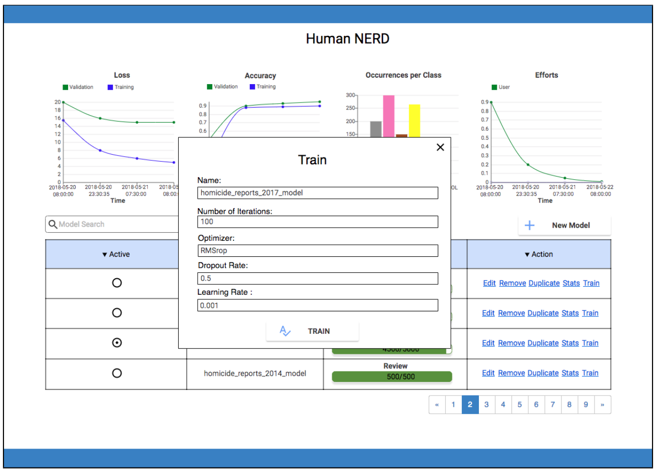
B. Data scientist in the Loop

Human NERD offers a full view of the imported or already trained NER models to the data scientist (Figure 3). A model is active if the framework is currently using it to annotate the documents during the user review (the previous scenario). The framework reports the status of review and train processing. The former corresponds to how many documents the reviewer already validated, and the later to how many epochs already finished during the model training. The data scientist can request the framework to train, duplicate, remove, edit the settings and visualize the statistics of NER models which were imported or trained by the framework.



▾ Active	▾ Name	▾ Status	▾ Action
○	homicide_reports_2018_model2	Train 90%	Edit Remove Duplicate Stats Train
○	homicide_reports_2018_model	Review 200/1000	Edit Remove Duplicate Stats Train
⊙	homicide_reports_2017_model	Review 3000/3000	Edit Remove Duplicate Stats Train
○	homicide_reports_2014_model	Review 500/500	Edit Remove Duplicate Stats Train

Figure 3: Data scientist visualization.



Train

Name: homicide_reports_2017_model

Number of Iterations: 100

Optimizer: RMSrop

Dropout Rate: 0.5

Learning Rate : 0.001

TRAIN

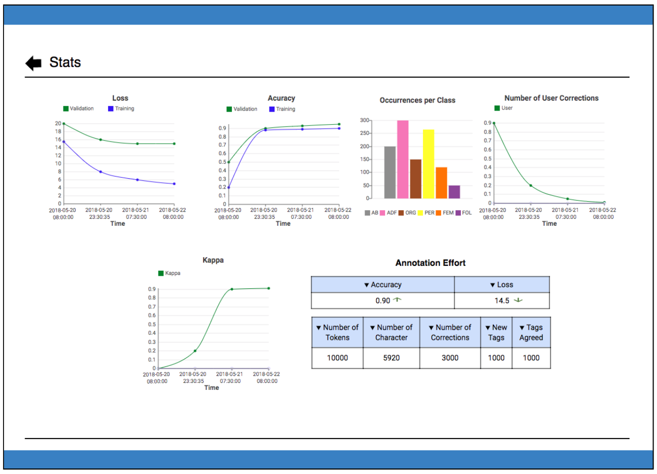
Figure 4: Human NERD enables the data scientist to train a model.

Figure 4 shows the interface presented to the data scientist which enables him/her to train a new model or update a previous one based on the texts validated by the reviewer. Human NERD requires for this functionality the inputs: number of iterations (epochs), optimizer, dropout rate, and learning rate. When the data scientist asks to edit the model settings, he/she can remove or add new label classes.

When the data scientist requests the visualization of statistics, Human NERD framework reports the *gain*, since we expect that the NER model improves over time, and the *user efforts* for annotating documents. Figure 5 summarizes those statistics collected over time when Human NERD trains a model or during documents annotation/validation performed by the reviewer.

4 CONCLUSION

In this demonstration, we proposed a framework called Human NERD to improve named entity recognition models by using a human in the loop. Human NERD relies on deep neural architectures for NER tasks that are free from language-specific resources (e.g., gazetteers, word clusters id, part-of-speech tags) or hand-crafted features (e.g., word spelling and capitalization patterns). We validate Human NERD framework with a real data set from Police reports in the Portuguese language, and we built upon Wikiner from Spacy a new deep learning NER model for this domain. A future work would be to improve the deep learning NER models by using ensemble techniques. Another direction is to provide a collaborative framework to allow multiple concurrent active models and reviewers.



Annotation Effort

▾ Accuracy	▾ Loss
0.90 ↑	14.5 ↓

▾ Number of Tokens	▾ Number of Character	▾ Number of Corrections	▾ New Tags	▾ Tags Agreed
10000	5920	3000	1000	1000

Figure 5: Statistics reports: the model *gain* and *loss*.

ACKNOWLEDGMENTS

This work has been supported by FUNCAP SPU 8789771/2017 research project.

REFERENCES

[1] 2017. Prodigy: A new tool for radically efficient machine teaching. https://explosion.ai/blog/prodigy-annotation-tool-active-learning. (2017). [Online; accessed 9-January-2019].

[2] Shilpa Arora, Eric Nyberg, and Carolyn P Rosé. 2009. Estimating annotation cost for active learning in a multi-annotator environment. In *Proceedings of the NAACL HLT 2009 Workshop on Active Learning for Natural Language Processing*. Association for Computational Linguistics, 18–26.

[3] Jason PC Chiu and Eric Nichols. 2015. Named entity recognition with bidirectional LSTM-CNNs. *arXiv preprint arXiv:1511.08308* (2015).

[4] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[5] Cıcero dos Santos, Victor Guimaraes, RJ Niterói, and Rio de Janeiro. 2015. Boosting Named Entity Recognition with Neural Character Embeddings. In *Proceedings of NEWS 2015 The Fifth Named Entities Workshop*. 25.

[6] John Foley, Sheikh Muhammad Sarwar, and James Allan. 2018. Named Entity Recognition with Extremely Limited Data. *arXiv preprint arXiv:1806.04411* (2018).

[7] Ashish Kapoor, Eric Horvitz, and Sumit Basu. 2007. Selective Supervision: Guiding Supervised Learning with Decision-Theoretic Active Learning.. In *IJCAI*, Vol. 7. 877–882.

[8] Trausti Kristjansson, Aron Culotta, Paul Viola, and Andrew McCallum. 2004. Interactive information extraction with constrained conditional random fields. In *AAAI*, Vol. 4. 412–418.

[9] Xuezhe Ma and Eduard Hovy. 2016. End-to-end sequence labeling via bi-directional lstm-cnns-crf. *arXiv preprint arXiv:1603.01354* (2016).

[10] Xuezhe Ma and Fei Xia. 2014. Unsupervised dependency parsing with transferring distribution via parallel guidance and entropy regularization. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 1337–1348.

[11] Olga Russakovsky, Li-Jia Li, and Li Fei-Fei. 2015. Best of both worlds: human-machine collaboration for object annotation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2121–2131.

[12] Burr Settles, Mark Craven, and Lewis Friedland. 2008. Active learning with real annotation costs. In *Proceedings of the NIPS workshop on cost-sensitive learning*. Vancouver, Canada, 1–10.

[13] Dan Shen, Jie Zhang, Jian Su, Guodong Zhou, and Chew-Lim Tan. 2004. Multi-criteria-based active learning for named entity recognition. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 589.

[14] Fisher Yu, Ari Seff, Yinda Zhang, Shuran Song, Thomas Funkhouser, and Jianxiong Xiao. 2015. Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop. *arXiv preprint arXiv:1506.03365* (2015).

Demonstrating Data Collections Curation and Exploration with CURARE

Extended Demonstration Abstract

Genoveva Vargas-Solar
Univ. Grenoble Alpes, CNRS,
Grenoble INP, LIG-LAFMIA
Grenoble, France
genoveva.vargas@imag.fr

Gavin Kemp
Univ Lyon, University of Lyon 1,
LIRIS UMR 5205 CNRS
Villeurbanne, France
gavin.kemp@liris.cnrs.fr

Irving Hernández-Gallegos
Universidad Autónoma de
Guadalajara
Zapopan, Mexico
irving.hernandez.g@gmail.com

Javier A. Espinosa-Oviedo
Delft University of Technology
2628BL Delft, Netherlands
javier.espinosa@tudelft.nl

Catarina Ferreira Da Silva
Univ Lyon, University of Lyon 1,
LIRIS UMR 5205 CNRS
Villeurbanne, France
catarina.ferreira-da-silva@liris.cnrs.fr

Parisa Ghodous
Univ Lyon, University of Lyon 1,
LIRIS UMR 5205 CNRS
Villeurbanne, France
parisa.ghodous@liris.cnrs.fr

ABSTRACT

This paper demonstrates CURARE, an environment for curating and assisting data scientists to explore *raw* data collections. CURARE implements a data curation model used to store structural and quantitative metadata semi-automatically extracted. It provides associated functions for exploring these metadata. The demonstration proposed in this paper is devoted to evaluate and compare the effort invested by a data scientist when exploring data collections with and without CURARE assistance.

1 INTRODUCTION

The emergence of new platforms that produce data at different rates, is adding to the increase in the amount of data collections available online. Open data initiatives frequently release data collections as sets of records with more or less information about their structure and content. For example, different governmental, academic and private initiatives release open data collections like Grand Lyon in France[1], Wikipedia [2], Stack Overflow [3] and those accessible through Google Dataset Search [4]. Releases often specify minimum "structural" metadata as the size of the release, the access and exploitation license, the date, and eventually the records structure (e.g., names and types of columns). This brings an unprecedented volume and diversity of data to be explored.

Data scientists invest a big percentage of their effort processing data collections to understand records structure and content and to generate descriptions. Useful descriptions should specify the values' types, their distribution, the percentage of null, absent or default values, and dependencies across attributes. Having this knowledge is crucial to decide whether it is possible to run analytics tasks on data collections directly or whether they should be pre-processed (e.g., cleansed). Therefore, several questions should be considered by a data scientist. For example, whether one or more data collections can be used for target analytics tasks; whether they are complementary or not, or whether they can be easily integrated into one data set to be analyzed; whether certain attributes have been cleaned, computed (e.g., normalized) or estimated. Keeping track of cleansing changes is important because such operations can bias certain statistics analysis and results.

Therefore, the data scientist must go through the records and the values exploring data collections content like the records structure, values distribution, presence of outliers, etc. This is a time consuming process that should be done for every data collection. The process and operations performed by the data scientist are often not described and preserved, neither considered as metadata. Thus, the data scientist effort cannot be capitalized for other cases and by other data scientists.

Curation tasks include extracting explicit, quantitative and semantic metadata, organizing and classifying metadata and providing tools for facilitating their exploration and maintenance (adding and adjusting metadata and computing metadata for new releases) [3, 4, 6]. The problem addressed by curation initiatives is to address the exploration of data collections by increasing their usefulness and reducing the burden of exploring records manually or by implementing ad-hoc processes. Curation environments should aid the user in understanding the data collections' content and provide guidance to explore them [1, 2, 5, 8].

Our work focuses on (semi)-automatic metadata extraction and data collections exploration which are key activities of the curation process. Therefore, we propose a data collections' curation and exploration environment named CURARE.

CURARE provides tools in an integrated environment for extracting metadata using descriptive statistics measures and data mining methods. Metadata are stored by CURARE according to its data model proposed for representing curation metadata (see Figure 1). The data model organizes metadata into four main concepts:

- Data Collection: models structural metadata concerning several releases like the number of releases it groups, their aggregated size, the provider, etc.
- Release: models structural metadata about the records of a release (file) including for example the columns' name in the case of tabular data, the size of the release, the date of production, access license, etc.

[1] https://www.metropolis.org/member/grand-lyon
[2] https://www.wikidata.org/wiki/Wikidata:Main$_{page}$
[3] https://www.kaggle.com/stackoverflow/datasets
[4] https://toolbox.google.com/datasetsearch

Series ISSN: 2367-2005 10.5441/002/edbt.2019.65

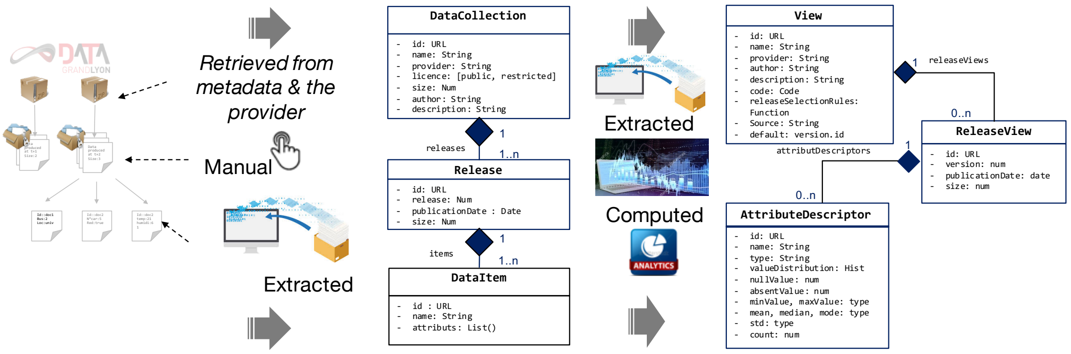


Figure 1: CURARE Model

- Release view: models quantitative metadata of the release records of a data collection. For example, the number of columns and the number of records in a release. A release view can store, for every column in all the records of a release, its associated descriptive statistics measures (mean, median, standard deviation, distribution, outlier values).
- View: models quantitative aggregated metadata about the releases of a data collection.

CURARE also provides functions for exploring metadata and assist data analysts determining which are the collections that can be used for achieving an analytics objective (comparing several collections, projecting and filtering their content). Rather than performing series of self-contained queries (like keyword search or relational ones when possible), a scientist can stepwise explore into data collections (i.e., computing statistical descriptions, discovering patterns, cleaning values) and stop when the content reaches a satisfaction point.

The demonstration proposed in this paper is devoted to evaluate and compare the effort put by a data scientist when exploring data collections with and without CURARE assistance. Through a treasure seeking metaphor we ask users to find cues within non curated Stack Overflow releases. Then, we compare the effort for finding cues by exploring metadata extracted by CURARE. The tasks are intended to show the added value of using metadata to explore data collections for determining whether they can be used for answering target questions.

The remainder of the paper is organized as follows. Section 2 describes the general architecture of CURARE and the extraction of structural and quantitative metadata. Section 3 describes the demonstration scenario for CURARE and the tasks used for measuring user experience during the demo. Section 4 concludes the paper and discusses lessons learned from the demonstration.

2 CURARE

The curation tasks implemented by CURARE are coordinated through a general iterative workflow consisting in three steps: (i) data collections' harvesting (manually or automatically) and preservation; (ii) structural and statistical meta-data extraction; (iii) exploration. The workflow is iterative because data collections can be updated with new releases and therefore releases are harvested recurrently. Similarly, as a result of curated data collections exploration, new metadata can be defined and the processing step can be repeated to include new metadata and store them.

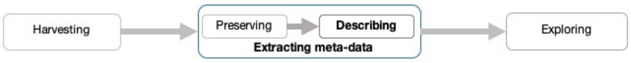


Figure 2: CURARE data curation workflow

2.1 General architecture

Figure 3 shows the architecture of CURARE consisting of services that can be deployed on the cloud. The current version of CURARE are Azure services running on top of a data science virtual machine. The services of CURARE are organized into three layers that correspond to the phases of the curation workflow. They are accessible through their Application Programming Interfaces (API's). Services can be accessed vertically as a whole environment or horizontally from every layer.

- The first layer is devoted to harvesting data collections and extracting structural metadata like the size, the provenance, time and location time-stamps.
- The second layer addresses distributed storage and access of curated data and metadata.
- The third layer provides tools for extracting quantitative metadata by processing raw collections.

The structural and quantitative metadata of CURARE are organized into four types of persistent data structures implementing CURARE data model (i.e., *data collection, release, view and release view*). According to the CURARE approach, a data collection consists of several sets of data released at a given moment. For example, Stack Overflow is a data collection consisting of packages of records (i.e., releases) produced periodically. A Stack Overflow release provides a kind of snapshot of the whole interactions happening in this social network. It is made available at some point in time. A release view, and a view, are data structures that provide an aggregated and quantitative perspectives of resp. a data collection and its several associated releases.

These data structures are then explored using ad-hoc operators within semi-declarative expressions or programs instead of directly exploring raw data collections. For example, is a release included in another? To which extent a release has new records with respect to others? Which is the percentage of missing values of the releases of a data collection?

2.2 Extracting structural metadata

The *data collection model* shown in the left hand side of Figure 1 provides concepts for representing data collections as sets of releases of raw data. Each release consists of a set of items (e.g.

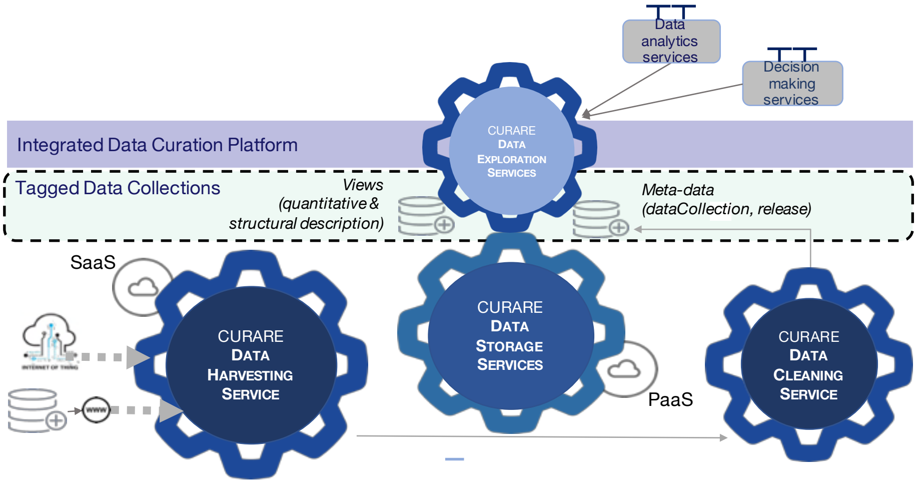

Data analytics services
Decision making services
Integrated Data Curation Platform
CURARE DATA EXPLORATION SERVICES
Tagged Data Collections
Views (quantitative & structural description)
Meta-data (dataCollection, release)
SaaS
CURARE DATA HARVESTING SERVICE
CURARE DATA STORAGE SERVICES
PaaS
CURARE DATA CLEANING SERVICE

Figure 3: General functional architecture of CURARE

records). The data collections model is used by the data harvesting and cleansing services for organizing structural and context metadata related to collections (provider, objective, URL, item structure).

The demonstrated version of CURARE provides a data collection upload facility for CSV files, which is an explicit data harvesting process. During this process, the structure, the size and release date of the file are extracted automatically. Then, other metadata for example the URI of the provider and the type of license can be provided by the data scientist by filling a formulary.

As said before, metadata are stored by CURARE. Thus, in the exploration phase a data scientist can search for releases of a given size, with a "open source" licence, or produced by a specific provider, expressing queries or executing explicit exported methods.

2.3 Extracting quantitative metadata

The *view model* of CURARE shown in the right hand side of Figure 1 provides concepts for representing quantitative and analytic aspects of the releases of a data collection. For instance, statistics of the content including the distribution of the values of each attribute across the items of a release, missing values, null values.

Depending on the attribute type (only atomic types are considered for statistics) the strategies for computing measures can change. CURARE first defines for every attribute in a tabular file, for example, its type and it approximately discovers possible null, absent and default values. Then, it computes description statistics measures (mean, mode, median, standard deviation, distribution) of the values of every attribute of the structure of the data collection in tabular format. It interacts with the data scientist to decide what to do with those attributes with a domain containing "dirty" values. Should these values be excluded from the computation? Or how can they be adjusted if they should be considered? If values are modified, should CURARE create a new release? Should it be preserved completely tagging the modified values as modified and indicating the applied formula or procedure. This is specified by the data scientist and managed by CURARE. For computing statistics of string values, CURARE processes the strings and creates frequency matrices and inverted indexes for the terms.

CURARE processes collections with many attributes and records that can be somehow costly depending on the tasks the data is used for, and the data volume. Thus, CURARE uses a Spark environment for computing statistics measures that sits on the data science virtual machine [5]. This strategy can be related to existing approaches like the one proposed in [7].

3 DEMONSTRATION OVERVIEW

The demonstration of CURARE uses a Jupyter notebook prepared for performing the tasks intended to test the use of metadata (i.e., entities of the CURARE data model) for exploring data collections. We use the Azure Notebooks environment [6] to run the demonstration. The back-end of the demonstrated version uses the Python data science libraries, nltk [7] for dealing with the processing of string typed attributes. We use ATLAS [8], the clustered public service of MongoDB for storing metadata associated to raw data collections.

The demonstration provides 3 releases of Stack Overflow [9] made available between the 1st. and 4th. January 2018 each consisting in five files "badges", "comments", "posts", "users" and "votes". These files range between ca. 300 KB to 20 MB.

CURARE extracts structural metadata and stores them as Data Collection documents according to its data model. It extracts quantitative metadata and stores them as Views documents. For the demonstration purposes we use a notebook that shows the process and the results. The releases used in the demonstration are processed and the results are stored in Atlas. The result weights 126,5 MB in Mongo and the data collection of structural metadata 9,5 KB.

Demonstration scenario. The demonstration of CURARE shows in which situations it can be useful to have an integrated curation and exploration environment. This can be evaluated through a set of tasks proposed by the demonstration scenario with a game called *The cure challenge*. The objective is to compare the effort when performing tasks to explore Stack Overflow releases manually and with CURARE.

[5] https://azure.microsoft.com/en-us/services/virtual-machines/data-science-virtual-machines/
[6] https://notebooks.azure.com
[7] https://www.nltk.org
[8] https://www.mongodb.com/cloud/atlas
[9] https://data.stackexchange.com/stackoverflow/query/new

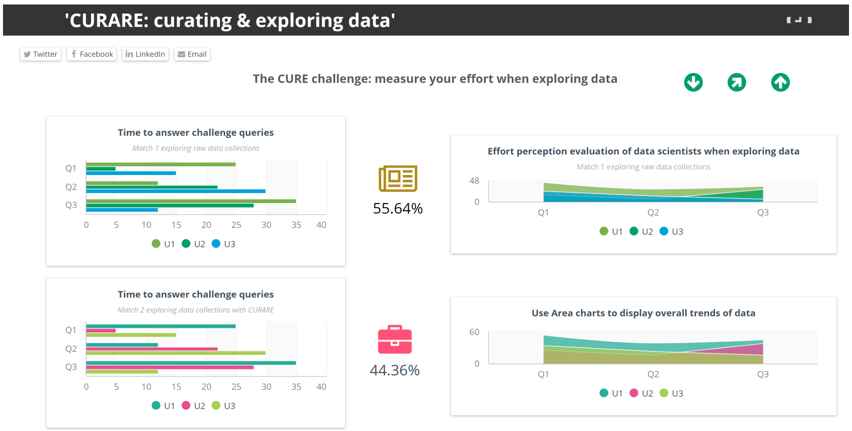

'CURARE: curating & exploring data'
Twitter
Facebook
LinkedIn
Email
The CURE challenge: measure your effort when exploring data
Time to answer challenge queries
Match 1 exploring raw data collections
Q1
Q2
Q3
0
5
10
15
20
25
30
35
40
U1
U2
U3
55.64%
Effort perception evaluation of data scientists when exploring data
Match 1 exploring raw data collections
48
0
Q1
Q2
Q3
U1
U2
U3
Time to answer challenge queries
Match 2 exploring data collections with CURARE
Q1
Q2
Q3
0
5
10
15
20
25
30
35
40
U1
U2
U3
44.36%
Use Area charts to display overall trends of data
60
0
Q1
Q2
Q3
U1
U2
U3

Figure 4: Assessment dashboard results of the demonstration scenario

When the game is activated it gives access to raw data collections and to CURARE metadata. The game consists of two curation matches where a data scientist has to perform the following tasks.

(1) Data collections exploration tasks:
   - Discover the release of the Stack Overflow data collection with the best quality. That is, the one with less missing, null and default values in the posts.
   - Compare the releases size in terms of number of records.
   - Compare releases in terms of values distribution for a given attribute.
   - Compare releases with respect to the topic by comparing terms of a given attribute.

(2) Discover the attribute(s) of the Stack Overflow posts that can be used to compute the popularity of the answers and the reputation of the author (stared answers and authors).

(3) Discover the attribute(s) that can be used to identify the most trendy topics addressed in the release. Will missing, null, default values bias the observation of the trends?

(4) Choose the attributes that can be used as sharding keys to fragment the release using a hash based and an interval based strategy.

In the first match the tasks are done by exploring directly the raw data collections. The second using CURARE's exploration facilities. For both matches a dashboard shown in Figure 4 automatically measures the degree of effort given by comparing with the accuracy/precision and correctness of the answers against time used for providing an answer. These measures are completed with an explicit score for every task given by the data scientist (one star low effort, three stars high effort).

Measuring data exploration effort. The experiment is assessed by comparing the evaluation results of the tasks performed in the two matches. We evaluate:

- Whether the metadata in views provide representative information of the raw content of a release (Q_1 and Q_2)
- How easy it is to see data collection quality in terms of consistency of the structure, the degree of missing, null and absent values of the attributes (Q_2)
- Usefulness of views for exploring the data collections to determine which kind of analytics questions they can answer. (Q_3, Q_4)

4 CONCLUSION AND RESULTS

We demonstrate CURARE that implements a data curation approach integrating metadata describing the structure, content and statistics of raw data collections. Thereby raw data collections can be comfortably explored and understood for designing data centric experiments through exploration operations. The demonstration shows the usefulness of CURARE by measuring the effort of a data scientist for performing exploration tasks in predefined Stack Overflow releases, used as demonstration examples.

5 ACKNOWLEDGEMENT

This work was partially funded by the Rhône-Alpes region through the project AMBED of the ARC 7 program[10] and the LDE Centre for BOLD Cities [11]. The implementation of CURARE was partially funded by the CONACYT "beca mixta" fellowship program of the Mexican government awarded to Irving Hernández Gallegos.

REFERENCES

[1] L. Battle, M. Stonebraker, and R. Chang. 2013. Dynamic reduction of query result sets for interactive visualizaton. In *2013 IEEE International Conference on Big Data*. 1–8. https://doi.org/10.1109/BigData.2013.6691708

[2] M. Bostock, V. Ogievetsky, and J. Heer. 2011. Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec 2011), 2301–2309. https://doi.org/10.1109/TVCG.2011.185

[3] André Freitas and Edward Curry. 2016. Big data curation. In *New Horizons for a Data-Driven Economy*. Springer, 87–118.

[4] Alon Halevy, Flip Korn, Natalya F Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. 2016. Goods: Organizing google's datasets. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 795–806.

[5] Martin L Kersten, Stratos Idreos, Stefan Manegold, Erietta Liarou, et al. 2011. The researcher as guide to the data deluge: Querying a scientific database in just a few seconds. *PVLDB Challenges and Visions* 3, 3 (2011).

[6] Michael Stonebraker, Daniel Bruckner, Ihab F Ilyas, George Beskales, Mitch Cherniack, Stanley B Zdonik, Alexander Pagan, and Shan Xu. 2013. Data Curation at Scale: The Data Tamer System.. In *CIDR*.

[7] Abdul Wasay, Xinding Wei, Niv Dayan, and Stratos Idreos. 2017. Data canopy: Accelerating exploratory statistical analysis. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 557–572.

[8] Sam Likun Xi, Oreoluwa Babarinsa, Manos Athanassoulis, and Stratos Idreos. 2015. Beyond the wall: Near-data processing for databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*. ACM, 2.

[10] http://www.arc7-territoires-mobilites.rhonealpes.fr/

[11] http://boldcities.nl/

SparkER: Scaling Entity Resolution in Spark

Luca Gagliardelli
University of Modena and Reggio Emilia
Modena, Italy
luca.gagliardelli@unimore.it

Giovanni Simonini
MIT CSAIL
Cambridge, MA, USA
giovanni@csail.mit.edu

Domenico Beneventano
University of Modena and Reggio Emilia
Modena, Italy
domenico.beneventano@unimore.it

Sonia Bergamaschi
University of Modena and Reggio Emilia
Modena, Italy
sonia.bergamaschi@unimore.it

ABSTRACT

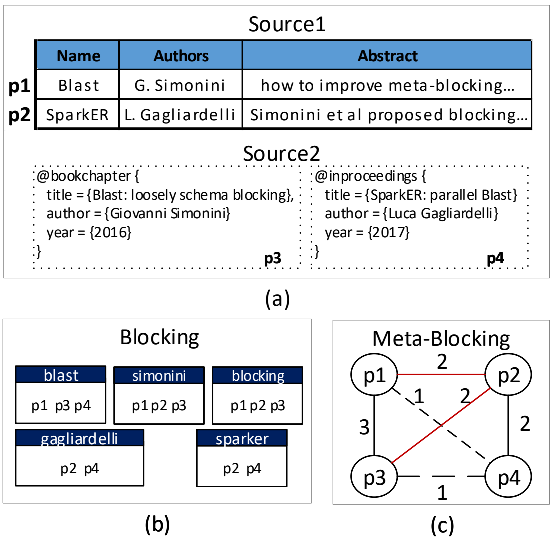
We present SparkER, an ER tool that can scale practitioners' favorite ER algorithms. SparkER has been devised to take full advantage of parallel and distributed computation as well (running on top of Apache Spark). The first SparkER version was focused on the *blocking* step and implements both *schema-agnostic* and *Blast meta-blocking* approaches (i.e. the state-of-the-art ones); a GUI for SparkER, to let non-expert users to use it in an unsupervised mode, was developed. The new version of SparkER to be shown in this demo, extends significantly the tool. *Entity matching* and *Entity Clustering* modules have been added. Moreover, in addition to the completely unsupervised mode of the first version, a supervised mode has been added. The user can be assisted in supervising the entire process and in injecting his knowledge in order to achieve the best result. During the demonstration, attendees will be shown how SparkER can significantly help in devising and debugging ER algorithms.

1 INTRODUCTION

Entity Resolution (ER) is the task of identifying different representations (*profiles*) that pertain to the same real-world entity. ER is a fundamental and expensive task for Data Integration [2]. The naïve solution of ER (i.e. comparing all profiles to each others) is impracticable when the data volume increases (e.g. Big Data), thus *blocking* techniques are employed to cluster similar records and to limit the number of comparisons only among the profiles contained in the same block.

In a real-world scenario, to identify a blocking strategy (i.e. the *blocking key*) yielding high recall and precision is a hard task [4]. In particular, in the Big Data context, *schema-aware* techniques have two main issues: (i) schema alignment, hardly achievable with a high heterogeneity of the data; (ii) labeled data to train classification algorithms, or human intervention to select which attributes to combine. To overcome these problems, the *schema-agnostic* approach was introduced [10]: each profile is treated as a *bag of words* and schema-information is ignored. For instance, *Schema-Agnostic Token Blocking* considers as blocking key each token that appear in profiles, regardless of the attribute in which it appears (Figure 1(b)). However, *schema-agnostic* methods produce a very low precision.

So, to mitigate this problem, they are typically coupled with *meta-blocking* [6, 10, 13]. The goal of *meta-blocking* is to restructure a blocking collection by removing least promising comparisons. This is achieved in the following way: profiles and comparisons are represented as nodes and edges of a graph, respectively; each node is connected to another one if the profiles co-occurs in at least one block. Then, the edges are weighted on the basis of the co-occurence of its adjacent profiles and for each profile a threshold is computed. Finally, the graph is pruned removing the edges which have a weight lower than the threshold. A toy example is shown in Figure 1(c): each edge is weighted counting the co-occurring blocks of its adjacent profiles, and is retained if its weight is above the average. The dashed lines are the removed comparisons.



Source1

	Name	Authors	Abstract
p1	Blast	G. Simonini	how to improve meta-blocking...
p2	SparkER	L. Gagliardelli	Simonini et al proposed blocking...

Source2

```
@bookchapter {
  title = {Blast: loosely schema blocking},
  author = {Giovanni Simonini}
  year = {2016}
}
```
p3

```
@inproceedings {
  title = {SparkER: parallel Blast}
  author = {Luca Gagliardelli}
  year = {2017}
}
```
p4

(a)

Blocking

blast	simonini	blocking
p1 p3 p4	p1 p2 p3	p1 p2 p3

gagliardelli	sparker
p2 p4	p2 p4

(b)

(c)

Figure 1: Schema-agnostic (meta-)blocking process.

In [13] we proposed *Blast*, which introduces the notion of *loose schema information* extracted from the data and composed of: (i) *attribute partitioning* and (ii) *attribute partition entropy* (Figure 2(a)). The idea beyond *attribute partitioning* is that more values two attributes share, more are similar, thus similar attributes are put together in the same partition. Then, the *meta-blocking* takes into account the generated attributes' partitions: the blocking key is composed by tokens concatenated to partition IDs; in this way, the token "Simonini" (Figure 2(b)) is split into two tokens, disambiguating "Simonini" as author ("Simonini_1"), and

Series ISSN: 2367-2005

10.5441/002/edbt.2019.66

"Simonini" as cited author; note that "Simonini_1" do not generate any block, since it appears only in p_2.

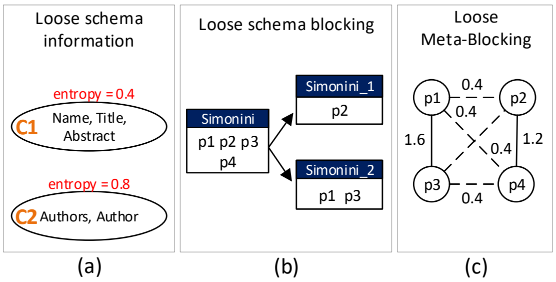


Figure 2: Meta-blocking with loose schema information.

Attribute partition entropy computes the entropy of each cluster and gives more importance to the profiles that co-occurs in blocks generated from clusters with high entropy. The idea is that finding equalities inside a cluster with a high variability of the values (i.e. high entropy) has more value that finding them in a cluster with low variability (i.e. low entropy). The *attribute partition entropy* is used in order to improve the edges weights: each edge of the *meta-blocking* graph is re-weighted according to the entropy associated to the block that generates it (i.e. the entropy of the partition from which the blocking key belongs), as shown in Figure 2(c). This affects the meta-blocking by helping to remove more superfluous comparisons than the ones removed by schema-agnostic blocking (the two retained red edges of Figure 1(c) are now removed).

At the end of the pruning step, the *meta-blocking* produces the *candidate pairs*, i.e. pairs of profiles related to the same entity. Then, these pairs have to be resolved, i.e. it is necessary to decide if a pair is a true match or not, this task is called *entity matching*. Several techniques can be applied to perform this task, e.g. resolution functions, classifiers, crowdsourcing, etc. Finally, , the retained matching pairs are clustered (*entity clustering*) in order to group together all the profiles associated to the same entity.

Several tools were proposed to cover the full Entity Resolution stack [9, 11]. In particular, JedAI [11] is more devoted to work with semi-structured data, a *schema-agnostic* approach, and the *entity matching* phase uses only unsupervised techniques (i.e. no labeled data are required). In contrast, Magellan [9] is meant to work with structured data and a supervised approach, so the user has to align the schema, to provide matches examples to perform *entity matching*, and supervise each step. Moreover, JedAI covers the *entity clustering* step, while Magellan not.

Nevertheless, none of these tools exploits the benefits of distributed computing. Works on the *meta-blocking* parallelization have been proposed [5], but they are implemented using *Hadoop MapReduce*, that is not the best paradigm to exploit modern cluster architectures [3, 12]. SparkER[1] is an Entity Resolution tool for Apache Spark[2] designed to cover the full Entity Resolution stack in a big data context.

Our approach. The first SparkER version [14] was focused on the *blocking* step and implements using *Apache Spark* both *schema-agnostic* [10] and *Blast* [13] *meta-blocking* approaches (i.e. the state-of-the-art ones). The description of the algorithms that we devised for Apache Spark (and any MapReduce-like system) can be found in our technical report [15]. Also, we developed a GUI for SparkER to let non-expert users to use it in an unsupervised mode.

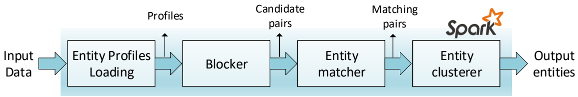


Figure 3: SparkER architecture.

The new version of SparkER that will be shown in this demo, extends significantly the tool. *Entity matching* and *entity clustering* modules have been added. Moreover, in addition to the completely unsupervised mode of the first version, a supervised mode has been added. The user can be assisted in supervising the entire process and in injecting his knowledge in order to achieve the best result.

In the following Section 2, we present the main modules that compose SparkER and in Section 3 the process debugging. Finally, in Section 4 we present the demonstration for the EDBT attendees.

2 SPARKER

SparkER is a distributed entity resolution tool, composed by different modules designed to be parallelizable on Apache Spark. Figure 3 shows the architecture of our system. There are 3 main modules: (1) **blocker**: takes the input profiles and performs the blocking phase, providing as output the candidate pairs; (2) **entity matcher** takes the candidate pairs generated by the blocker and label them as match or no match; (3) **entity clusterer** takes the matched pairs and groups them into clusters that represents the same entity. Each of these modules can be seen as black box: each one is independent from the other.

2.1 Blocker

Figure 4 shows the blocker' sub-modules implementing the *Loose-Schema Meta-Blocking* method described in the introduction.

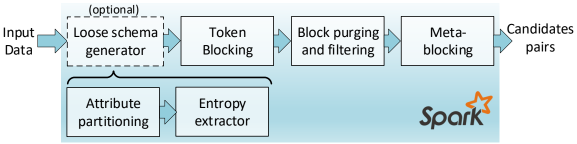


Figure 4: Blocker module

Loose Schema Generator-Attribute Partitioning: attributes are partitioned in cluster using a *Locality-sensitive Hashing* (LSH) based algorithm. Initially, LSH is applied to the attributes values, in order to group them according to their similarity. These groups are overlapping, i.e. each attribute can compare in multiple clusters. Then, for each attribute only the most similar one is kept, obtaining pairs of similar attributes. Finally, the transitive closure is applied to such attributes pairs and then attributes are partitioned into nonoverlapping clusters (Figure 2(a)). All the attributes that do not appear in any cluster are put in a *blob* partition.

[1] https://github.com/Gaglia88/sparker
[2] http://spark.apache.org

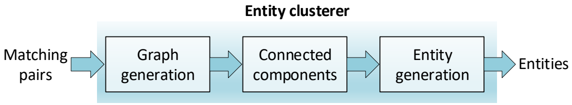


Figure 5: Entity clusterer.

Loose Schema Generator-Entropy Extractor: computes the Shannon entropy for each cluster.

Block Purging and Filtering : the block collection is processed to remove/shrink its largest blocks [10]. *Block Purging* discards all the blocks that contain more than half of the profiles in the collection, corresponding to highly frequent blocking keys (e.g. stop-words). *Block Filtering* removes each profile from the largest 20% blocks in which it appears, increasing the precision without affects the recall.

Meta-Blocking: Finally, the *meta-blocking* method [10, 13] introduced in the introduction is applied. The parallel *meta-blocking*, implemented on Apache Spark, is inspired by the broadcast join: it partitions the nodes of the blocking graph and sends in broadcast (i.e, to each partition) all the information needed to materialize the neighborhood of each node one at a time. Once the neighborhood of a node is materialized, the pruning function is applied.

The output of the `blocker` module are profile pairs connected by an edge, which represent candidate pairs that will be processed by the *entity matcher* module.

2.2 Entity Matcher and Clusterer

Regarding Entity Matching, any existing tool can be used. In the demo we will show the one implemented in Magellan [9]. The `Entity Matcher` produces *matching pairs* of similar profiles with their similarity score (*similarity graph*). The user can select from a wide range of similarity (or distance) scores, e.g.: Jaccard similarity, Edit Distance, CSA [1].

The `Entity Clusterer` receives as input the similarity graph, in which the profiles are the nodes and the matching pairs represent the edges, and partition its nodes into *equivalence clusters* such that every cluster contains all profiles that correspond to the same entity. Several entity clustering algorithms have been proposed in literature [8]; at the moment, we use the *connected component* algorithm[3], based on the the assumption of *transitivity*, i.e., if p_1 matches with p_2, p_2 matches with p_3, then p_1 matches with p_3. At the end of this step, the system produces clusters of profiles: the profiles in the same cluster refer to the same real-world entity.

3 PROCESS DEBUGGING

The tool can work in a completely unsupervised mode, i.e. the user can use a default configuration and performs the process on its data without taking care of the parameters tuning. Otherwise, the user can supervise the entire process, in order to determine which are the best parameters for its data, producing a custom configuration. Given the iterative nature of this process (e.g. the user try a configuration, if it is not satisfied changes it, and repeat the step again), it is not feasible to process the entire input data, as the user should waste too much time. Thus, it is necessary to sample the input data, reducing the size. The main problem is to take a sample that represents the original data, and also contains matching and non matching profiles. This problem was already addressed in [9], where the authors proposed to pick up some random K profiles P^K, then for each profile $p_i \in P^K$ pick up $k/2$ profiles that could be a match (i.e. shares a high number of token with p_i) and $k/2$ profiles randomly. K and k are two parameters that can be set by the user based on the time that she wants to spend (e.g. selecting more records requires a higher computation time).

Each step can be assessed using precision and recall, if a ground-truth is available; otherwise the system selects a sample of the generated profile pairs (e.g. pairs after blocking, matching pairs after matching, etc.) and shows them to the user who, on the basis of his experience evaluates whether the system is working well or not.

In the `blocker` each operation (blocking, purging, filtering, and meta-blocking) can be fine tuned in order to obtain better performances, e.g. the purging/filtering are controlled through parameters that can change the aggressiveness of filters, or the *meta-blocking* can use different types of pruning strategies, etc. Moreover, if the *Loose Schema Blocking* is used, it is possible to see how the attributes are clustered together, and how to change the clustering parameters in order to obtain better clusters.

In the entity matching phase, it is possible to try different similarity techniques (e.g. Jaccard, cosine, etc.) with different thresholds.

At present no tuning activity is possible in the *clustering* step since the connected component algorithm used does not have any parameters. At the end of the process, the system allows to explore the generated entities and to store the obtained configuration. Then, the optimized configuration can be applied to the whole data in a batch mode, in order to obtain the final result.

4 DEMONSTRATION OVERVIEW

During the demonstration, participants will explore the features of our system on the `Abt-Buy` dataset[4]. It contains 2,000 products extracted from *Abt.com* and *Buy.com* catalogs, denoted respectively in red and blue. The dataset comes with a ground-truth that allows to analyze the performances of each `SparkER` step. Also, different datasets can be used[5] during the demonstration.

In this demo we focus on showing the attribute partitioning unsupervised/supervised step, the use of Attribute Partition Entropy was illustrated in our previous paper [7] and the meta-blocking step including entropy.

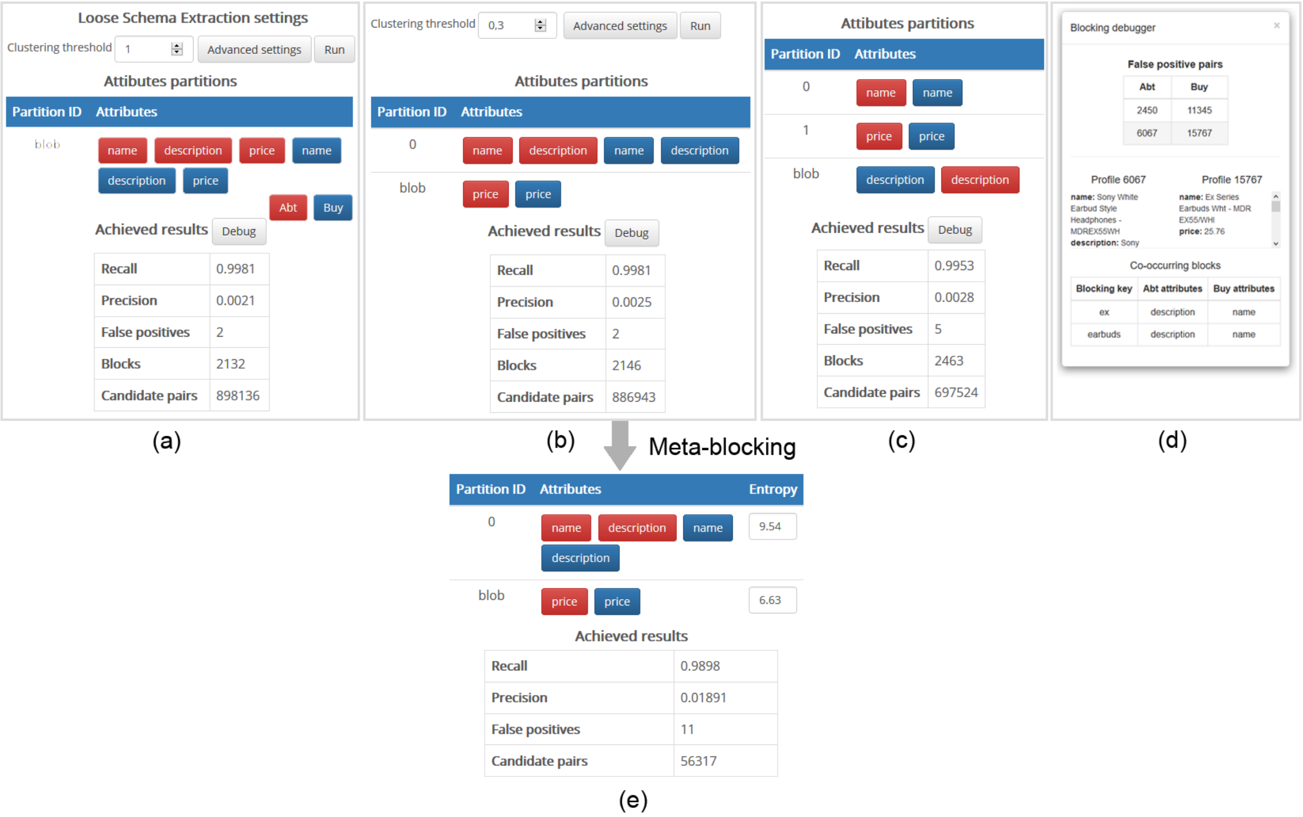
The tool displays the attributes partitions, recall/precision, the number of blocks (blocking keys) generated, the number of candidate pairs in the blocks, and the number of false positives (i.e. the pairs that are in the ground-truth but are lost during the blocking process) obtained after blocking. Through the interface it is possible to modify the clustering threshold and other parameters (*Advanced settings*) which influence the algorithm in a more marginal way.

We start setting the threshold to the maximum value (1) e.g a schema-agnostic token blocking is applied and all the attributes fall in the same *blob* cluster (Figure 6(a)). Then the user decreases the threshold (0.3) and looks at what happens (Figure 6(b)). Two clusters are created, representing, respectively the name with the

[3] This approach is implemented by using the GraphX library of Spark (https://spark.apache.org/graphx/) that natively implement the *connected component* approach.

[4] https://dbs.uni-leipzig.de/en/research/projects/object_matching/fever/benchmark_datasets_for_entity_resolution

[5] The datasets are available at: https://sourceforge.net/projects/sparker/files/datasets/



(a)

Loose Schema Extraction settings

Clustering threshold 1 Advanced settings Run

Attibutes partitions

Partition ID	Attributes
blob	name description price name description price
	Abt Buy

Achieved results Debug

Recall	0.9981
Precision	0.0021
False positives	2
Blocks	2132
Candidate pairs	898136

(b)

Clustering threshold 0,3 Advanced settings Run

Attibutes partitions

Partition ID	Attributes
0	name description name description
blob	price price

Achieved results Debug

Recall	0.9981
Precision	0.0025
False positives	2
Blocks	2146
Candidate pairs	886943

(c)

Attibutes partitions

Partition ID	Attributes
0	name name
1	price price
blob	description description

Achieved results Debug

Recall	0.9953
Precision	0.0028
False positives	5
Blocks	2463
Candidate pairs	697524

(d)

Blocking debugger

False positive pairs

Abt	Buy
2450	11345
6067	15767

Profile 6067

name: Sony White Earbud Style Headphones - MDREX55WH
description: Sony

Profile 15767

name: Ex Series Earbuds Wht - MDR EX55/WHI
price: 25.76

Co-occurring blocks

Blocking key	Abt attributes	Buy attributes
ex	description	name
earbuds	description	name

Meta-blocking

(e)

Partition ID	Attributes	Entropy
0	name description name description	9.54
blob	price price	6.63

Achieved results

Recall	0.9898
Precision	0.01891
False positives	11
Candidate pairs	56317

Figure 6: Process debugging. The figure shows how it is possible to debug the blocking phase.

description, and the prices of the products. However, we see how precision slightly increases but the number of candidate pairs has been reduced.

Now, the user try to modify the clusters, as apparently separating the attributes that refer to the name from those which refer to the description of products (Figure 6(c)) seems a good idea. He looks at the result and sees that unfortunately the number of false positive increases.

By the *Debug* button it is possible to understand where the false positives come from (Figure 6(d)). The tool shows the list of false positive pairs (i.e. pairs that are in the ground-truth but are not present after the blocking). By clicking on a pair, its profiles and shared *blocking key* are shown and the user can understand why this pair was lost. In the example we can see that the lost pairs match on blocking keys referring to the name and description attributes. So, partitioning descriptions and names was a wrong choice and the automatic solution proposed by the tool was better (Figure 6(b)). Moreover, it suggests that the choice of partitioning the attributes on the bases of their names (i.e. exploting schema information) can be wrong.

Finally, Figure 6(e) shows the debugging of the meta-blocking phase, with the Entropy's values obtained by the Entropy Extractor module. We can see a large decrease in the number of candidate pairs w.r.t. 6(b) thus proving the effectiveness of our technique.

REFERENCES

[1] Fabio Benedetti, Domenico Beneventano, Sonia Bergamaschi, and Giovanni Simonini. 2019. Computing inter-document similarity with context semantic analysis. *Information Systems* 80 (2019), 136–147.

[2] Sonia Bergamaschi, Domenico Beneventano, Francesco Guerra, and Mirko Orsini. 2011. Data Integration. In *Handbook of Conceptual Modeling - Theory, Practice, and Research Challenges*. 441–476.

[3] Sonia Bergamaschi, Luca Gagliardelli, Giovanni Simonini, and Song Zhu. 2017. BigBench workload executed by using Apache Flink. *Procedia Manufacturing* 11 (2017), 695–702.

[4] P. Christen. 2012. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE transactions on knowledge and data engineering* 24, 9 (2012), 1537–1555.

[5] V. Efthymiou, G. Papadakis, G. Papastefanatos, K. Stefanidis, and T. Palpanas. 2017. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Information Systems* 65 (2017), 137–157.

[6] Simonini G., Papadakis G., Palpanas T., and Bergamaschi S. 2018. Schema-Agnostic Progressive Entity Resolution. In *ICDE 2018*. 53–64.

[7] Simonini G., Gagliardelli L., Zhu S., and Bergamaschi S. 2018. Enhancing Loosely Schema-aware Entity Resolution with User Interaction. In *HPCS 2018, July 16-20, 2018*. 860–864.

[8] O. Hassanzadeh, F. Chiang, H. C. Lee, and R.ée J Miller. 2009. Framework for evaluating clustering algorithms in duplicate detection. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1282–1293.

[9] P. Konda, S. Das, P. Suganthan GC, A. Doan, A. Ardalan, J. R Ballard, H. Li, F. Panahi, H. Zhang, J. Naughton, et al. 2016. Magellan: Toward building entity matching management systems. *VLDB Endowment* 9, 12 (2016), 1197–1208.

[10] G. Papadakis, G. Papastefanatos, T. Palpanas, and M. Koubarakis. 2016. Scaling Entity Resolution to Large, Heterogeneous Data with Enhanced Meta-blocking.. In *EDBT*. 221–232.

[11] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, and M. Koubarakis. 2018. The return of jedAI: end-to-end entity resolution for structured and semi-structured data. *VLDB Endowment* 11, 12 (2018), 1950–1953.

[12] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. 2015. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment* 8, 13 (2015), 2110–2121.

[13] G. Simonini, S. Bergamaschi, and HV Jagadish. 2016. BLAST: a loosely schema-aware meta-blocking approach for entity resolution. *VLDB Endowment* 9, 12 (2016), 1173–1184.

[14] G. Simonini, L. Gagliardelli, S. Zhu, and S. Bergamaschi. 2018. Enhancing Loosely Schema-aware Entity Resolution with User Interaction. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 860–864.

[15] G. Simonini, Gagliardelli L., Bergamaschi S., and Jagadish H.V. 2019. Technical Report. (2019). http://dbgroup.unimo.it/paper/g/scaling_er_report.pdf

Demonstration

Exploring Interpretable Features for Large Time Series with SE4TeC

Jingwei Zuo
DAVID Lab, University of Versailles
University of Paris-Saclay
Versailles, France
jingwei.zuo@uvsq.fr

Karine Zeitouni
DAVID Lab, University of Versailles
University of Paris-Saclay
Versailles, France
karine.zeitouni@uvsq.fr

Yehia Taher
DAVID Lab, University of Versailles
University of Paris-Saclay
Versailles, France
yehia.taher@uvsq.fr

ABSTRACT

Time Series (TS) data are ubiquitous in enormous application fields, such as medicine, multimedia, and finance. In this paper, we present the demonstration with **SE4TeC**: A **S**calable **E**ngine for efficient and expressive **T**ime S**e**ries **C**lassification, which is applicable to certain fields in Big Data context, where the TS features and their extraction process should be interpretable. *SE4TeC* improves the state of the art solutions by proposing a scalable and highly efficient method to classify TS based on characteristic subsequences (i.e., shapelets). We explain the techniques we adopt, and show how to use *SE4TeC* for exploring the real-life datasets in medical diagnosis and in industrial troubleshooting.

1 INTRODUCTION

Tony is informed unhealthy heart status on the basis of his electrocardiogram (ECG) during a medical diagnosis. An experienced doctor can easily correlate the abnormal ECG with the diseases, then explain to Tony the symbolic abnormality in the ECG and the relevant treatment. Nowadays, Machine Learning technique can partly replace the role of an experienced doctor and do the diagnosis very accurately. In Tony's case, the electrical activity of the heart from physiological sensor is collected as Time Series (**TS**). From the perspective of the machine, the diagnosis can be considered as a Time Series classification (**TSC**) problem.

The classical approaches [1] on TSC problems are usually based on the statistical features extracted from time series, such as mean, standard deviation of subsequences, which are assumed to represent the global characteristics of time series. Intuitively, they get very superficial information with low noise tolerance. On the basis of solving these limitations, the **Shapelet** approach [12] has attracted great interest over the past years, owing to its high discriminative feature and good interpretability. However, extracting shapelets from data series has a large computation cost. Even for small sized datasets, the algorithm can take days. This is mainly due to the repeated similarity search between a sub-sequence (i.e., a candidate shapelet) and TS instances in the database. Some typical speed-up techniques (i.e., indexing [11], lower-bounding [6] and early abandoning [12]), introduce always extra parameters, which is difficult to operate without prior knowledge. Some low-dimensional representation methods have also been proposed, such as Piece-wise Aggregate Approximation (PAA) [4], which computes the mean of each subsequence of time series in a given length, and transforms the raw data in coarse-grained sub-components. Symbolic Aggregate approXimation (SAX) [5], transforms subsequences of raw time series into value-characterized symbols, which is eligible for a hierarchic indexing iSAX [2] to accelerate similarity search. Nevertheless, scalability remains the bottleneck.

Our work is biased towards raw time series processing which has a higher accuracy performance, but a relatively high time complexity [12]. Unlike some hardware-based implementations, such as using GPUs to accelerate the similarity calculation [3], we focus on the scalability of TSC based on shapelet extraction. Traditional TSC algorithms on raw TS data are not applicable for big data context, because of their low scalability. Here are our contributions in this paper:

(1) We propose a novel method to assess the importance of shapelets in batches
(2) We introduce a scalable engine to extract the shapelets
(3) Based on the scalable engine, we propose an optimization strategy to speed-up the shapelets extraction.

The rest of this paper is organized as follows. In Section 2, we review the background and state the research problems. We present our scalable engine for Time Series Classification in Section 3. Section 4 shows an empirical evaluation of our method, as well as a guidance for the demonstration. Finally, we give our conclusions and perspectives for future work in Section 5.

2 BACKGROUND

2.1 Definitions and Notations

We start with defining the notions used in the paper:

Definition 1: *A Time Series T* is a sequence of real-valued numbers T=$(t_1, t_2, ..., t_i, ..., t_n)$, where n is the length of T.

Definition 2: *A subsequence* $T_{i,m}$ of Time Series T is a continuous subset of values from T of length m starting from position i. $T_{i,m} = (t_i, t_{i+1}, ..., t_{i+m-1})$, where $i \in [0, n-m+1]$.

Definition 3: *Shapelet* $\hat{s}$ is a time series subsequence which is particularly representative of a class. As such, it shows a shape which can distinguish one class from the others.

Definition 4: *A Dataset D* is a collection of time series T_i, and its class label c_i. Formally, $D = <T_1, c_{j_1}>, <T_2, c_{j_2}>, ..., <T_N, c_{j_N}>$, where N is the number of instances in D. $C = c_1, c_2, ..., c_{|C|}$ is a collection of class labels, where $|C|$ denotes the number of labels.

Definition 5: *Z-Normalization Time Series* is a formal representation of Time Series, which is defined as $ZNormal(T) = \frac{T-\mu}{\sigma}$, where μ is the sample mean, σ is the standard deviation:

$$\mu = \frac{1}{m}\sum_{i=1}^{n} t_i, \quad \sigma^2 = \frac{1}{m}\sum_{i=1}^{n} t_i^2 - \mu \tag{1}$$

Z-Normalization allows us to focus on the structural feature of T, rather than its amplitude value. It addresses the problem of data stability. For instance, assume that the Euclidean Distance(ED) between two time series $T_{x,m}$, $T_{y,m}$ is expressed as follows:

$$ED_{x,y} = \sqrt{\sum_{i=1}^{m}(t_{x,i} - t_{y,i})^2} \tag{2}$$

10.5441/002/edbt.2019.67

Some little changes (e.g., the noise) will cause an evident bias for the result, Z-Normalization is a way of smoothing the bias value.

Definition 6: *Normalized Euclidean Distance(N-ED)*, is expressed by the formula $\sqrt{\frac{1}{m}\sum_{i=1}^{m}(t_{x,i} - t_{y,i})^2}$

Definition 7: *Distance Profile* DP_i is a vector which stores the Normalized Euclidean Distance between a given subsequence/query $T_{i,m}$ and every subsequences $T'_{j,m}$ of a target Time Series T'. Formally, $DP^m_{i,j} = dist(T_{i,m}, T'_{j,m}), \forall j \in [0, n' - m + 1]$

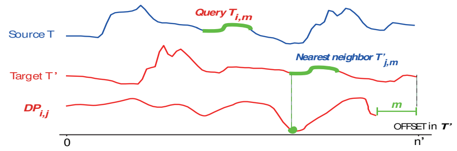


Figure 1: Distance Profile between Query $T_{i,m}$ and target time series T', where n' is the length of T'. Obviously, $DP_{i,j}$ can be considered as a meta TS annotating target T'

Definition 8: *MASS*, namely Mueen's ultra-fast Algorithm for Similarity Search, computes Distance Profile based on Fast Fourier Transform(FFT), which requires just $O(nlogn)$ time, other than $O(nm^2)$ time in classical *N-ED* similarity search.

Definition 9: *Matrix Profile MP* is a vector of distance between subsequence $T_{i,m}$ in source T and its nearest neighbor $T'_{j,m}$ in target T'. Formally, $MP^m_i = min(DP^m_i)$, where $i \in [0, n - m + 1]$.

Unlike the distance profile, the matrix profile is a meta TS annotating the source time series. The highest point on MP corresponds to the TS discord, the lowest points correspond to the position of a query which has a similar matching in target TS.

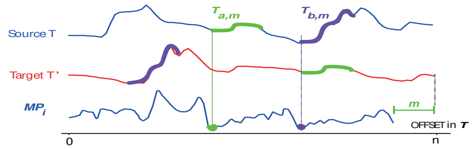


Figure 2: Matrix Profile between Source time series T and Target time series T', where n is the length of T. Intuitively, MP_i shares the same offset as source T

2.2 Evaluation of Candidate Shapelets

The quality of a candidate shapelet $\hat{s}$, can be assessed by its ability to separate the instances of different class in the dataset D. A prerequisite of the quality measure for $\hat{s}$, is that a set of distance $D_{\hat{s}}$ must be calculated, where $D_{\hat{s}} = D_{\hat{s},1}, D_{\hat{s},2}, ...D_{\hat{s},n}$, n is the number of T in dataset D.

Information Gain, an evaluation method based on Decision Tree, is widely adopted in previous works [12]. An iteration test of $D_{\hat{s}}$ is conducted to extract the split distance which brings the highest Information Gain. The distance instance will be applied as a property of candidate Shapelet, to check the inclusion between the candidate and time series. Another simple approach, is to use the F-Statistic [7], based on the difference of means in an Analysis of Variance A(NOVA). The main idea of this statistic method, is to assess the difference in distributions of $\hat{s}$ between the class distances.

2.3 Problem Statement

The high degree of coupling inside classical TSC algorithm [12] leads to the problem of not being able to parallelize. The speedup method such as Early Abandoning [12] is based on classical Euclidean Distance measure, which has a time complexity of $O(N^2n^4)$ with several orders of magnitude higher than *MASS* [8]: $O(N^2n^3logn)$. Another common trick played by previous work [12]: If we know $\hat{s}$ is a low-quality candidate, then any similar subsequence $\hat{s}'$ to $\hat{s}$ must also result in a low quality and therefore, a costly computation of the distance set $D_{\hat{s}'}$ (evaluation of $\hat{s}'$) can be skipped. However, a candidate shapelet is evaluated by its quality ranking among all candidates of the same length. Assume that the distributed nodes have generated from dataset a collection of candidates $\hat{s}_l$, an aggregation operation between nodes is required to extract the candidate with the best quality. Extra aggregations will be made along with the iteration of candidate length. Apparently, the acceleration from the classical pruning techniques can be easily offset by the communication cost caused by the aggregation.

3 SYSTEM OVERVIEW

The main idea of our system is that the calculation should be shared and executed independently, less communication between the nodes, more powerful the algorithm would be.

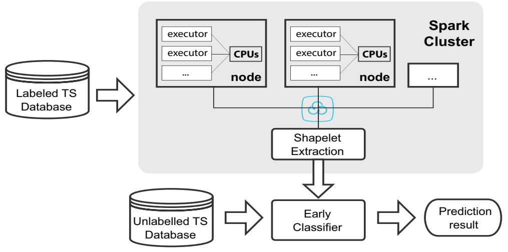


Figure 3: System overview

3.1 Main Structure

The conventional Time Series classification problems are tackled with nearest neighbor (kNN) algorithm [12] due to its easy-design feature. As shown in Figure 3, on the basis of kNN, an early classifier [10] adopted in the system allows to give the prediction result as earlier as possible without waiting for the entire sequence. The processing of labelled Time Series data requires to be flexibly arranged for nodes in the cluster, where the executors share the CPU/memory resource. To this end, a suitable algorithm is applied here allowing assignment of computing tasks which are relatively independent of each other.

3.2 SMAP: Shapelet Extraction on MAtrix Profile

Matrix Profile provides a meta-data which facilitates the representation of a complex correlation between two time series. As shown in *Algorithm 1*, *SMAP* takes the time series as the smallest processing unit between nodes, and utilizes the normalized quality to extract the most important parts in each processing unit and then merges them by an aggregation process. For this reason, the number of candidate shapelet could be greatly reduced. Moreover, a single aggregation is required to get the global shapelet result of different class. In *line 5*, dataset is broadcast to distributed nodes in order to reduce the communication cost caused by accessing the common data. Then, each cluster partition shares the computing tasks for a set of time series. The function *computeDiscrimP* aims at computing in batches the quality of candidate shapelets. The visualized process is shown in **Figure 4**.

The batch quality of instances in a TS is defined by *Discrimination Profile*, which refers to the concept *Representative Profile*:

$$RP(T_i^C, D) = avg(MP_{T_i^C, T_j}) \tag{3}$$

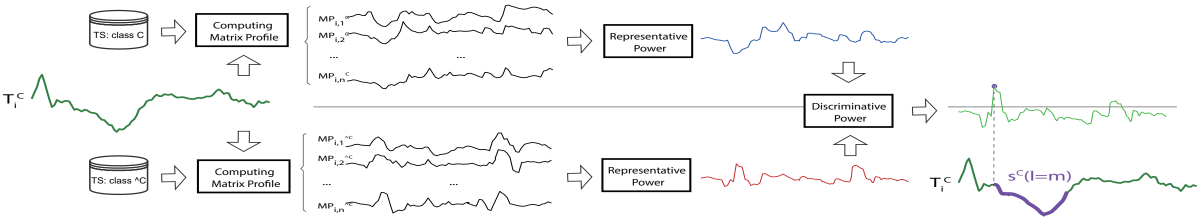


Figure 4: Discrimination Profile Extraction

Algorithm 1: SMAP(Shapelet on MAtrix Profile)

```
   Input: Dataset $D$, classSet $\hat{C}$, $k$
   Output: $\hat{S}$
1  $minLength \leftarrow 2$, $maxLength \leftarrow getMinLen(D)$
2  double[] $DiscmP \leftarrow []$, double[] $DistThresh \leftarrow []$, $\hat{S} \leftarrow \emptyset$
3  $D$.cache(); //cache all the dataset in the cluster, where each time series has an unique ID
4  MapPartition $(Set\ of < ID, T >: T_{set})$
5      for $< ID, T > \in T_{set}$ do
6          for $m \leftarrow minLength$ to $maxLength$ do
7              $DiscmP[m], DistThresh[m] \leftarrow computeDiscmP(T, D, m)$
8              $DiscmP[m] \leftarrow DiscmP[m] * \sqrt{1/l}$
9          $DiscmP \leftarrow pruning(DiscmP)$
10         emit(DiscmP, DistThresh)
11 MapAggregation $(class, (DiscmP, DistThresh))$
12     for $c \in \hat{C}$ do
13         $\hat{S}' \leftarrow getTopk(DiscmP[c], DistThresh[c], k)$
14         for $\hat{s} \in \hat{S}$ do
15             $\hat{s}.matchingIndices \leftarrow getMatchingIndices(\hat{s}, D)$
16         $\hat{S} \leftarrow \hat{S} \bigcup \hat{S}'$
17 return $\hat{S}$
```

where $T_j \in D^C$. Representative Profile targets thus on the minimal processing unit (i.e., time series) in SMAP, which shows a vector of Representative Power of each instance in the processing unit. To put it simply, the *Representative Power* of a subsequence in class C, is its normalized distance to the global instance cluster of class C. Intuitively, it represents the relevance between the subsequence (i.e., the candidate shapelet) and the class.

The fact that a subsequence is discriminative for its class towards others, can be expressed by the difference of Representative Power from class C to others *(OVA, one-vs-all). Discrimination Profile* is then defined as follows:

$$Discm_{Profile}(T_i^C, D) = -(RP(T_i^C, D^C) - RP(T_i^C, D^{!C})) \qquad (4)$$

A quality *Normalization* in *line 8* is made which allows to assess the Discrimination Power for shapelet of different length in an uniform way. Similar as the concept *Information Gain*, but Discrimination Profile is a technique more interpretable serving to assess the candidate shapelets. Moreover, in this manner, a split distance can be given directly, other than iterating every possible distance and deciding the best one with the highest Information Gain, in time $O(N^2n^2)$. A strategy to check if T contains a shapelet can be defined as the following:

$$sInT(T, \hat{s}^C) = \begin{cases} true, & if\ dist(T, \hat{s}^C) \le RP(\hat{s}^C, D^C) \\ false, & otherwise \end{cases} \qquad (5)$$

3.3 Optimization Strategy

The pruning function in *line* 9 is capable of eliminating the number of candidate shapelet, and then reducing the communication cost during the aggregation process. We can simply take the "TopK" strategy, which extracts the biggest K values of the *Discrimination Profile*. However, such a technique is far from lightening the computation during *MapPartition* process.

Since each processing unit should be independent from each other, a tenable technique for updating the profile of a long range query could be adopted. The *Lower Bounding distance* [6] is defined to estimate a minimal possible Z-Normalized Euclidean Distance between two subsequences $T_{i,l+k}$ and $T_{j,l+k}$, based on the distance already computed between $T_{i,l}$ and $T_{j,l}$. Compared to a linear time complexity of computing the exact distance, LB Distance Profile can be calculated in a constant time, which can accelerate greatly the computation of Matrix Profile in *Figure 4.* For example, from shapelet length $l = m$ to $m + 1$, the time complexity of computing the distance $d_{i,j}^{l+1}$ is $O(n\frac{m(m-1)}{2}m)$, where $j \in [0, n\frac{m(m-1)}{2}]$ which represents the number of subsequences in D, n is the number of instance in D, m is the length of the longest instance in D. Accordingly, LB distance takes $O(n\frac{m(m-1)}{2})$ which shows an apparent advantage when the query length is relatively long. Lower Bounding distance [6] is defined as:

$$LB(d_{i,j}^{l+k}) = \begin{cases} \sqrt{l}\frac{\sigma_{j,l}}{\sigma_{j,l+k}}, & if q_{i,j} \le 0 \\ \sqrt{l(1 - q_{i,j}^2)}\frac{\sigma_{j,l}}{\sigma_{j,l+k}}, & otherwise \end{cases} \qquad (6)$$

where $q_{i,j} = \frac{\sum_{p=1}^{l} \frac{(t_{j+p-1} t_{i+p-1})}{l} - \mu_{i,l}\mu_{j,l}}{\sigma_{i,l}\sigma_{j,l}}$

Empirically, the matching subsequence $T_{j,l}$ which is the nearest neighbor of $T_{i,l}$, can deduce a longer subsequence $T_{j,l+1}$, which is probably the nearest neighbor of $T_{i,l+1}$. Assume that the matching subsequence keeps in the same position in T_{target} when query $T_{i,l}$ length increases, then the time complexity for computing the minimal distance between $T_{i,l}$ and T_{target} is $O(l)$, other than $O(l(n - l + 1))$. As mentioned in **Definition 9**, $MP_i^m = min(DP_i^m)$, the main idea here is to utilize LB Distance to accelerate the computation of $min(DP_i^m)$, rather than computing the entire DP_i in a higher time complexity.

4 ABOUT THE DEMONSTRATION

This demonstration is intended to show a distributed approach to extract features from large-scale time series and to make the extraction process and extracted features very easy to understand and interpret, thanks to the visual power of shapelets. Through this demonstration, the attendees will have a general understanding of time series, as well as its application areas and the current challenges. With two real datasets, attendees will have the opportunity to experience and interact with *SE4TeC* from two aspects:

(1) *Practical operation for distributing computation tasks:* The attendees are invited to connect to our elastic cluster, and will further explore the distribution mechanism for feature extraction. For instance, the relationship between performance and adjustable parallelism, the progress monitoring of parallel tasks on each distributed node, etc.
(2) *Exploration of shapelet extraction process:* Based on a small-sized dataset, the attendees can interactively perform each intuitive step shown in *Figure 4,* and are invited to analyze the hidden meaning behind each intermediate features.

In the view of the attendees, the reliability of extracted shapelets can be proved by utilizing naked eyes or a smart classifier, on a test instance.

4.1 Demonstration Platform

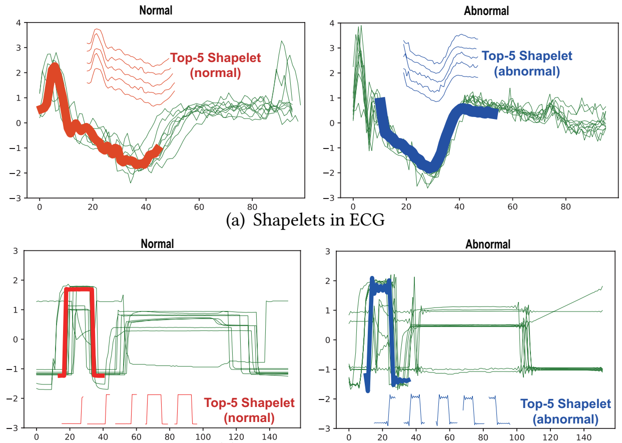
SE4TeC is implemented by Python3.6, powered by Apache Spark. The program is executed on AWS EMR cluster. We provide also an 1-click cluster based on Docker, to facilitate the attendees to replay the distributed test offline. The baseline of the evaluation is **USE** in [9], which utilizes the traditional method for shapelet extraction based on Information Gain. **1NN** classifier is applied for all accuracy tests, and **5** shapelets are extracted for each class.

4.2 Demonstration Scenario

Due to page limitations, here we show two examples, more details and videos can be found on the demonstration page[1].

4.2.1 ECG medical diagnosis. The dataset **ECG200**, from MIT-BIH Long-Term ECG Database (ltdb), is collected by two electrodes which record the brain activities in distinct body positions. Each heartbeat has an assigned label of normal or abnormal. All abnormal heartbeats are representative of a cardiac pathology known as supraventricular premature beat. ECG200 contains **100** labelled records with a fixed length of 96.

4.2.2 Wafer industrial troubleshooting. To put the evaluation into larger scale context, we choose the time series dataset **Wafer**, which contains **1000** training records with a fixed length of 152. The dataset, collected during the manufacture of semiconductor microelectronics, comprises a collection of sequences for the measurements recorded by one vacuum-chamber sensor during the etch process applied to one silicon wafer. The class of manufacture quality can be normal or abnormal.



(a) Shapelets in ECG

(b) Shapelets in Wafer

Figure 5: Interpretable Shapelet Feature Results

Reliability & Interpretability: Through the demonstration, the attendees are capable of extracting the shapelets in various manners, and comparing their difference. The shapelets extracted by SMAP are shown in *Figure 5*, which has a relatively higher prediction accuracy than USE: 84%|76% (ECG), 97%|92% (Wafer).

Scalability: The performance results are shown in *Figure 6*. $SMAP_{LB}$ shows a gain in performance: **24.5X** (ECG), **587.3X** (Wafer) faster. As the size of ECG (i.e., 100) is lower than the parallelism power when we expand the cluster to 10 nodes, therefore, according to the time $O(N^2 n^3 logn)$, the instance length n will be the decisive factor in execution time. We should know that the speed-up performance relies on the computing power of the cluster. We are more inclined to consider its parallel capability which can be assessed by the aggregation cost between distributed nodes. From 1 to 30 nodes on cluster mode, the aggregation cost for *Wafer* increases by **181%**, the total cost drops to **0.68%**. Obviously, considering the gain, the aggregation cost can be ignored when we expand the cluster to a larger scale.

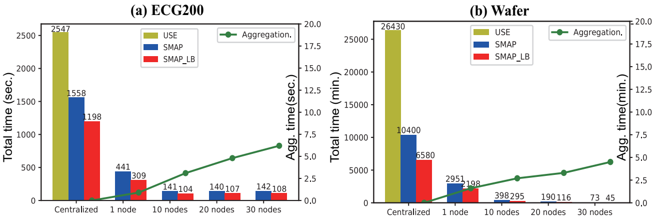


Figure 6: Scalability performance

5 CONCLUSION

In this paper, we have proposed a novel methodology, namely SMAP, for Time Series Classification. SMAP adopts the concept of Matrix Profile, and extracts the shapelet features in an interpretable and scalable manner. Within SMAP, the Discrimination Profile is defined to assess in batches the quality of candidate shapelets. On the basis of Lower Bounding, we have proposed an acceleration strategy that benefits from distributed environment. The satisfactory results proved the efficiency and competitiveness of our approach. Different optimizations are in our planning list. Specifically, we intend to expand SMAP for longer time series, while ensuring its high accuracy and scalability. This may lead us to combine SMAP with dimensionality reduction.

ACKNOWLEDGEMENT

This research was supported by DATAIA convergence institute as part of the « Programme d'Investissement d'Avenir », (ANR-17-CONV-0003) operated by DAVID Lab, University of Versailles Saint-Quentin, and MASTER project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie-Slodowska Curie grant agreement N. 777695.

REFERENCES

[1] Ling Bao and Stephen S. Intille. 2004. Activity recognition from user-annotated acceleration data. Springer, 1–17.
[2] Alessandro Camerra et al. 2010. iSAX 2.0: Indexing and Mining One Billion Time Series. In *Proc. ICDM 2010.* 58–67.
[3] Kai Wei Chang et al. 2012. Efficient pattern-based time series classification on GPU. In *Proc. ICDM 2012.* 131–140.
[4] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. 2001. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. *Knowledge and Information Systems* (01 Aug 2001), 263–286.
[5] Jessica Lin et al. 2003. A Symbolic Representation of Time Series, with Implications for Streaming Algorithms. In *Proc. 8th ACM SIGMOD (DMKD '03).*
[6] Michele Linardi and Yan et al. Zhu. 2018. VALMOD: A Suite for Easy and Exact Detection of Variable Length Motifs in Data Series. In *Proc. SIGMOD'18.*
[7] Jason Lines, Luke M Davis, Jon Hills, and Anthony Bagnall. 2012. *A Shapelet Transform for Time Series Classification.* Technical Report.
[8] Chin-Chia Michael Yeh et al. 2016. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View That Includes Motifs, Discords and Shapelets. In *Proc. ICDM '16.* 1317–1322. https://doi.org/10.1109/ICDM.2016.0179
[9] Raef Mousheimish, Yehia Taher, and Karine Zeitouni. 2017. Automatic Learning of Predictive CEP Rules: Bridging the Gap between Data Mining and Complex Event Processing. In *Proc. DEBS '17.* 158–169.
[10] Zhengzheng Xing, Jian Pei, and Philip S. Yu. 2009. Early Prediction on Time Series: a Nearest Neighbor Approach. *21st IJCAI* (2009), 1297–1302.
[11] Djamel-Edine Yagoubi et al. 2017. DPiSAX: Massively Distributed Partitioned iSAX. In *Proc. ICDM 2017.* 1–6.
[12] Lexiang Ye and Eamonn Keogh. 2009. Time series shapelets: A New Primitive for Data Mining. *Proc. 15th ACM SIGKDD* (2009).

[1] https://github.com/JingweiZuo/SE4TeC

HASQL: A Method of Masking System Failures

Mark Hannum, Adi Zaimi, Michael Ponomarenko, Dorin Hogea, Akshat Sikarwar, Mohit Khullar, Rivers Zhang, Lingzhi Deng, Nirbhay Choubey, Joe Mistachkin
Bloomberg L.P.
{mhannum,azaimi,mponomar,dhogea,asikarwar1,mkhullar1,hzhang320,ldeng33,nchoubey,jmistachkin}@bloomberg.net

ABSTRACT

We demonstrate a methodology of masking system failures in a way that doesn't require programmer or operational intervention, and that strives to be imperceptible to the client. High Availability SQL (HASQL) masks system failures in a clustered database-system by *seamlessly* restoring a transaction's state against a different machine in the cluster. We have implemented HASQL in Comdb2, an open source RDBMS developed by Bloomberg L.P.

To demonstrate, we allow participants to kill (via a button) database instances one at a time, and all instances simultaneously, as we execute an ongoing transaction against a Comdb2 cluster. Upon achieving this, viewers will see the command-line session and transaction pause briefly as the Comdb2 client-API connects to a different cluster node and re-establishes the transaction's state, allowing it to resume processing at the exact point of the disconnect. We repeat this demonstration, showing that a transaction's state can be correctly re-established even though it is midway through consuming a result set.

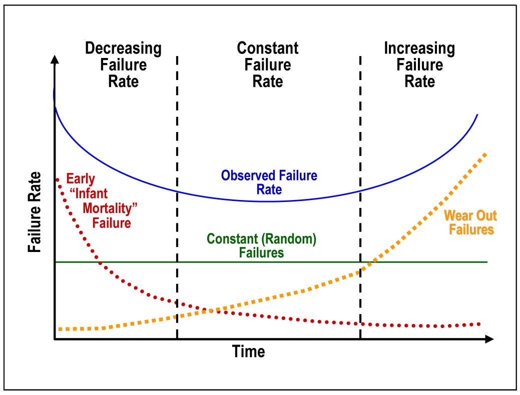


Figure 1: 'Bathtub curve' hazard function, adopted from [1], describes the failure rate of a single-component.

1 INTRODUCTION

Plummeting hardware prices have created increasing pressure on software developers to design redundant systems, as the chance for failure for any component of a system increases over time (see figure 1). For RDBMS systems, *High-Availability* solutions attempt to restore service quickly and minimize the effects of outages[2]; indeed, many commercial RDMBS systems[3, 5, 6] support automatic failover to provide uninterrupted service under the loss of part of a database cluster. In traditional failover strategies, a crashed server will return a `CONNECTION LOST` error to the client. Application writers address this by programming defensively, marrying database API calls to complex and often poorly tested retry logic[7]. Handling a `CONNECTION LOST` error in response to a `COMMIT` directive gives the programmer the additional burden of determining the fate of that transaction.

Our contribution is unique in that we show how to provide *seamless* continuation of in-flight transactions under an *optimistic concurrency control (OCC)* system: HASQL clients do not reissue SQL in the face of machine failure, and need not be aware that a machine failure has occurred, as every in-flight transaction will be automatically re-established and continued against another machine in the cluster, and any partially consumed result set will continue to be returned, as the system we describe guarantees that the client will never experience duplicate or missing data.

Oracle's *Application Continuity*[4] feature is a proprietary implementation which achieves the same goal. As we have implemented this as part of an open source system, we are able to describe our methodology explicitly, and we hope that by doing so, other systems can likewise provide this feature to their users. We note that any correct methodology will ensure that the replayed SQL modifies the same set of records that would have been modified on the original host, but that in an OCC system, write conflicts for replayed transactions are handled in the same manner as write conflicts for any two competing transactions: the winning transaction will be allowed to modify the rows in its write-set, while the losing transaction will return a verify error to the client. So even though HASQL provides the ability to perfectly replay a read-write transaction, as with every transaction in an OCC system, the replayed transaction will only be able to commit successfully if the rows in its write-set have not been modified.

In this paper, we describe HASQL as implemented in Comdb2, an open source RDBMS developed at Bloomberg L.P.[1], noting that a brief overview of HASQL was outlined in [8]. As this method relies on a limited number of architectural features, we describe it generally, trusting that it should be straightforward to implement HASQL in a similar system. Our motivation for this feature grew organically from Bloomberg's business need to have an *always available* database server and to provide an intuitive and reliable software infrastructure layer to its application developers. HASQL achieves this by shifting the responsibility for handling hardware failures from the client application to the database system.

2 SYSTEM OVERVIEW

We now describe the architecture and methodology for implementing HASQL. See [8] for a comprehensive review of Comdb2's architecture.

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

[1] https://github.com/bloomberg/comdb2

2.1 Architecture

A *database cluster* consists of N *database instances* running on N physically separate machines sharing no components and connected to each other by a network. Each instance contains a complete copy of the data, and is able to start and maintain arbitrary point-in-time snapshot transactions.

The cluster maintains a single *master* which may modify database state, and which synchronously replicates these modifications to every other instance in the cluster. We assume that transactions are immutable and are applied atomically on the master in a serial order defined by the global order of transaction `COMMIT`s.

For any two committed transactions, T1 and T2, if T1 is committed first, a point-in-time snapshot transaction started at T1's commit point will observe all the effects of T1's modifications, and none of the effects of T2's modifications, while new snapshot transactions will observe the effects of both T1 and T2. Modifications are applied to non-master instances (or *replicants*) atomically in the same serial order as they were applied on the master.

A *point-in-time token (or PIT-token)* identifies a specific point in the serial order of committed transactions. Given a PIT token, each instance is able to produce a snapshot view corresponding to that point in the serial order. The *transaction-id (tid)* is a unique identifier for a transaction; it is generated by the client API at the beginning of a transaction.

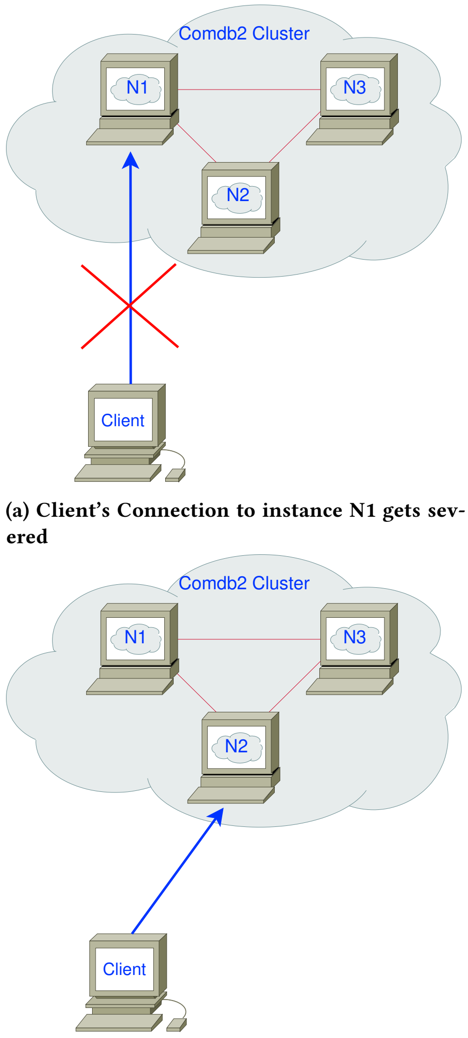
Writes performed against a replicant are not executed locally, but rather validated and executed on the master after the client issues a `COMMIT`. The client API is aware of the cluster's topology and maintains a single connection against an arbitrary replicant.

2.2 Methodology

At the beginning of a transaction, the client API generates a tid, establishes a connection against a replicant (if not already connected), and retrieves from that instance a PIT-token corresponding to the current state of that instance. As transactions are applied in the same order on replicants as they are on the master, the snapshot described by this PIT-token is guaranteed to be equivalent to the current or some former state of the data as it existed on the master, and may be used on any instance to recreate the same snapshot so long as that instance has applied transactions at least up to that point. During the course of the transaction, all write statements and the most recent read statement are cached by the client API. It's important to note that the results returned by the most recent read statement need not be retained: rather, the API retains only a count of the rows already retrieved from an in-flight `SELECT` statement.

Upon losing connection with the cluster, the client API reconnects to a different instance and begins a snapshot transaction at the time described by the original PIT-token, as doing so ensures that the client's view of the database on that instance will be identical to its view on the original. The client API then re-issues the transaction's write statements. If the connection was lost while the client was consuming results of a read statement, the client API re-issues the most recent read statement using the cached count to skip (or alternatively, ask the server to skip) records that have been previously returned to the user.

At commit time, the tid is used as a key to store the transaction's result in a replicated *global-transaction-table*. The pre-existence of a tid in the global-transaction-table indicates that the transaction has already executed, and that the current thread should roll back any work that it has done and return the original result to the client. The tid and the global-transaction-table ensure that a transaction will only be executed once should the client API replay a transaction after issuing a `COMMIT`.



(a) Client's Connection to instance N1 gets severed

(b) Client reconnects to instance N2

Figure 2: Connection to Cluster gets severed and Client subsequently reconnects to a different instance.

2.3 Illustrated Example

We describe a concrete example using the event diagram shown in Figure 3. Events in green represent SQL statements submitted by the user. Events in blue represent operations executed on behalf of the user by the client API code. Server responses are displayed in black.

As previously described, upon beginning a transaction, the API generates a unique tid, which is sent to the server along with the `BEGIN` statement. The server responds with the PIT-token. Each write statement of the transaction is cached locally as it is sent to the server. Figure 3 shows a failure occurring after reading the second record of a `SELECT` statement. The client API reconnects to Node 2, begins a transaction at the point-in-time described by the PIT-token, reissues the transaction's write statements, and the most recent incomplete `SELECT`. As the first two rows have been returned to the application, the client API skips them, returning the third result row to the caller. The client finally issues a COMMIT to complete the transaction.

2.4 Considerations

The HASQL scheme as described above works with SQL which behaves deterministically. It requires that each participating instance impose the same implicit order for results not sorted by an ORDER BY clause. We note also that the implicit ordering requirement does not exclude server-side parallelism, but concede that any ORDER BY clause, either explicit or implicit, may incur a performance penalty.

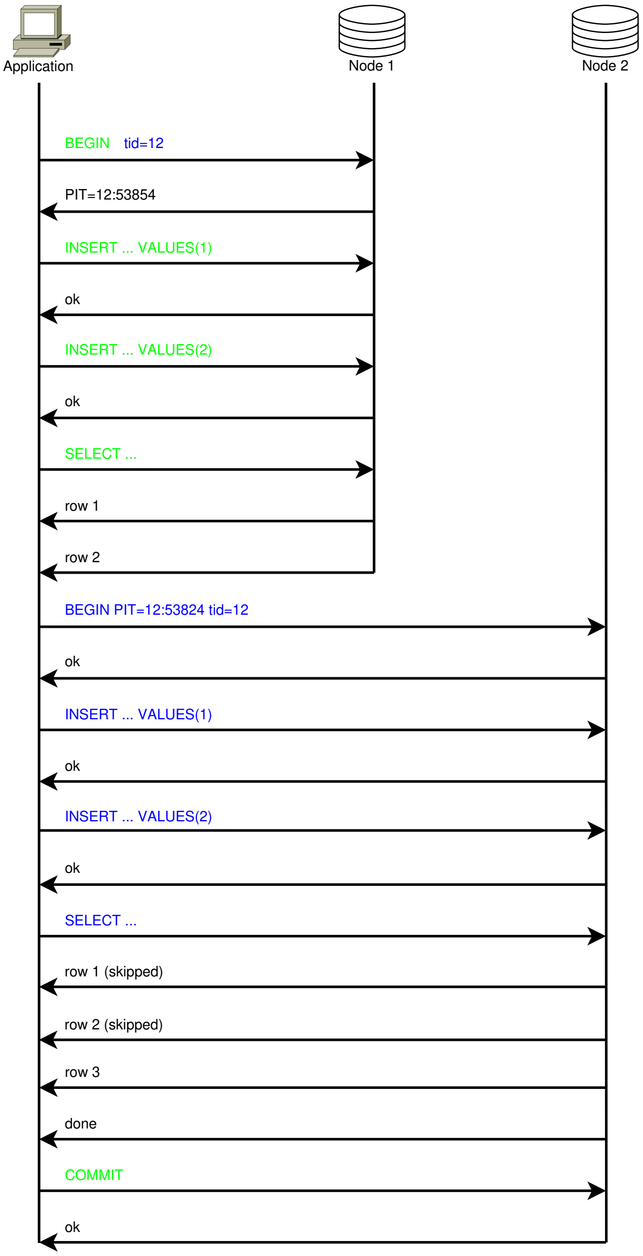


Figure 3: Example HASQL sequence: After connection to Node 1 is severed, Client Application reconnects to Database Cluster Node 2, and continues processing where it left off.

We now consider strategies for handling other non-deterministic SQL. Rather than attempting to address this exhaustively, we consider two situations which can employ a similar strategy; we offer this strategy as a blueprint for how non-determinism might be addressed.

```
BEGIN
DELETE FROM schedule WHERE updatetime <= NOW()
SELECT * FROM schedule
COMMIT
```

Listing 1: SQL using the NOW() function

If replayed, the NOW() function will execute at a different time on a second machine, and delete a different set of records. A replay which occurs during the SELECT could have already returned records which are deleted on the retry. The replay would subsequently skip over *non-deleted* records, as the client API's cached count pertains to the original result set. To address this, we propose that NOW() be frozen during the course of a snapshot transaction, always returning the wall clock time of the BEGIN issued on the original node. This time can be included as part of an extended PIT-token.

```
BEGIN
UPDATE contestants SET winner = 1 WHERE
    ticketnumber = (SELECT ticketnumber FROM
    contestants ORDER BY RANDOM() LIMIT 1)
SELECT * FROM contestants ORDER BY ticketnumber
COMMIT
```

Listing 2: SQL using the RANDOM() function

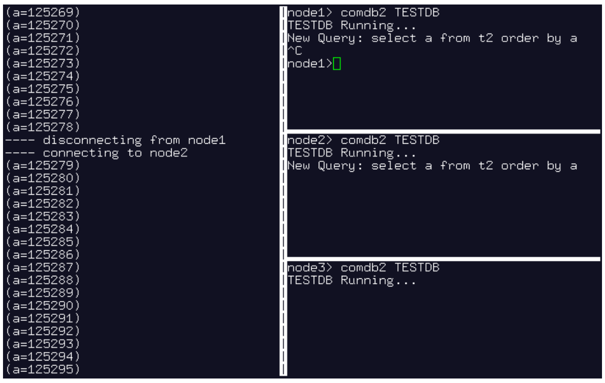
Because RANDOM() can return a different value on the replay machine, the re-issued UPDATE statement can update a different record. Should a replay event occur while retrieving results from the SELECT, the result set can show that two different contestants have winner set to 1. A system could work around this issue by making note of the RANDOM() number generator's current *seed* value at the beginning of a transaction. A replicant which seeds its RANDOM() number generator with this value should return the same sequence of random numbers as the original machine. As with NOW(), the seed value could be included as part of an extended PIT-token.

3 DEMO

We present a simple interactive demonstration which exhibits the HASQL scheme. We begin by starting a 3-node cluster and presenting a volunteer with three buttons, each programmed to stop and restart the database instance running on one of the cluster nodes. From a separate machine, we open a command-line session and begin a transaction against this cluster. Our volunteer will be instructed to press, at his or her discretion, the kill-and-restart-button corresponding to the machine that is currently executing the transaction. We present this as a simple game, allowing us to demonstrate the HASQL feature in a lighthearted and engaging way.

When the volunteer kills the correct instance, spectators will see the client-session pause briefly as the client API reconnects to a different cluster machine. We encourage our volunteer to kill and restart the active cluster node multiple times as we continue the transaction, being sure to demonstrate HASQL's ability to resume a transaction in the middle of retrieving a result set.

We then perform a second demonstration which is identical to the first, except that instead of killing a single cluster node, we ask



```
(a=125269)
(a=125270)
(a=125271)
(a=125272)
(a=125273)
(a=125274)
(a=125275)
(a=125276)
(a=125277)
(a=125278)
---- disconnecting from node1
---- connecting to node2
(a=125279)
(a=125280)
(a=125281)
(a=125282)
(a=125283)
(a=125284)
(a=125285)
(a=125286)
(a=125287)
(a=125288)
(a=125289)
(a=125290)
(a=125291)
(a=125292)
(a=125293)
(a=125294)
(a=125295)
```

```
node1> comdb2 TESTDB
TESTDB Running...
New Query: select a from t2 order by a
^C
node1>
```

```
node2> comdb2 TESTDB
TESTDB Running...
New Query: select a from t2 order by a
```

```
node3> comdb2 TESTDB
TESTDB Running...
```

Figure 4: Demonstration of HASQL with additional trace enabled. Left panel shows trace emitted from client application. Right panels show trace emitted by the three Comdb2 cluster nodes. Server instance on Node 1 (top-right) was manually terminated.

three volunteers to kill all three cluster nodes simultaneously. As the cluster restarts, spectators will see again that the transaction is resumed seamlessly.

We proceed to describe our implementation of HASQL, and repeat both demonstrations with additional trace enabled, which allows audience members to witness the sequence of events outlined in Figure 3. In contrast to seemlessly continuing a transaction, this demonstration seeks to exhibit HASQL's underlying mechanism.

We further explore HASQL's behavior by repeating both demonstrations, this time with an increased number of database instances, and with varying levels of background writes. Spectators will observe that HASQL's performance is unaffected by the increased cluster size, but that it is directly impacted by external writes to a table which an HASQL transaction reads. We use this as a starting point for a discussion of Comdb2's implementation of point-in-time snapshot isolation and other architectural features of Comdb2.

4 FUTURE WORK

A transaction which survives a machine crash naturally takes longer to complete. While this does not effect the correctness of a read-only query, the increased transaction time increases the likelihood that a write transaction, T1, will fail, as it allows greater opportunity for a competing transaction to write in the space of T1's write-sets. This is a small concession to make, as prior to HASQL, a machine crash would certainly cause T1 to fail, and there are a great number of non-intersecting write loads that would permit T1 to commit.

Future work includes finding ways to minimize the amount of time it takes to restore a partially completed transaction. We observe that the slowness is most pronounced when a machine crash occurs after a client has retrieved a substantial part of a large result set which must be skipped.

We could gain substantial improvement by maintaining simultaneous connections to multiple cluster machines, using the original PIT-token from the *primary* connection to establish one or more *secondary* connections. Each SQL statement would be issued to the primary and to the secondary handles in lock-step. In the normal case, the redundant sessions are essentially wasted computing power, but as hardware resources continue to become cheaper, this may eventually be a valid concession to make.

5 CONCLUSION

HASQL's contribution is one of resiliency: application developers need not know or care if the underlying system has experienced a critical error. We believe this is superior to other failover schemes, where a machine failure, in addition to failing all outstanding transactions, can stall clients for several minutes before a failover machine is available. An API return code which does not designate the success or failure of an operation places a disproportionate burden on the application programmer in answering a question which would be more appropriately addressed by the database system itself. Though we concede that this is unavoidable at times, HASQL addresses a significant subset of these errors. As hardware is guaranteed to fail, it is the responsibility of system designers to minimize the impact of failure. HASQL demonstrates an intelligent way to utilize increasingly less expensive hardware to create more robust service.

REFERENCES

[1] Bathtub curve. https://en.wikipedia.org/wiki/Bathtub_curve. Accessed: 2017-11-17.

[2] Gray, J. N., and Siewiorek, D. P. High-availability computer systems. *IEEE Computer 24* (1991), 39–48.

[3] MySQL. Mysql high availability. https://www.mysql.com/products/enterprise/high_availability.html. Accessed: 2017-11-17.

[4] Oracle. Application continuity. https://www.oracle.com/database/technologies/high-availability/app-continuity.html. Accessed: 2018-11-28.

[5] Oracle. Oracle database high availability. https://www.oracle.com/database/high-availability/index.html. Accessed: 2017-11-17.

[6] PostgreSQL. Postgresql high availability. https://www.postgresql.org/docs/8.3/static/high-availability.html. Accessed: 2017-11-17.

[7] Scotti, A. Adventures in building your own database. In *In All Your Bases Conference* (November 2015).

[8] Scotti, A., Hannum, M., Ponomarenko, M., Hogea, D., Sikarwar, A., Khullar, M., Zaimi, A., Leddy, J., Angius, F., Zhang, R., and Deng, L. Comdb2: Bloomberg's highly available relational database system. *PVLDB 9*, 13 (2016), 1377–1388.

Query-Driven Data Minimization with the DataEconomist

Peter K. Schwab
Friedrich-Alexander-Universität Erlangen-Nürnberg
peter.schwab@fau.de

Julian O. Matschinske
Friedrich-Alexander-Universität Erlangen-Nürnberg
julian.matschinske@fau.de

Andreas M. Wahl
Friedrich-Alexander-Universität Erlangen-Nürnberg
andreas.wahl@fau.de

Klaus Meyer-Wegener
Friedrich-Alexander-Universität Erlangen-Nürnberg
klaus.meyer-wegener@fau.de

ABSTRACT

In this paper, we explain the demonstration of the DataEconomist, a framework for query-driven data minimization in relational DBMS conformable to law. Our approach automatically minimizes user access rights based on the analysis of SQL query logs and thereby enables a parsimonious data processing. A minimization of data collection is reached by automatic detection and manual deletion of data belonging to unneeded schema elements.

In contrast to existing approaches, the DataEconomist supports privacy officers by focusing on the queries instead of making them trawl through a vast amount of collected personal data and specify legal use cases for their processing. SQL queries easily show who has processed when which data for which purpose and make it easy for the privacy officers to decide about the queries' compliance with data privacy regulations. Our framework does not require any knowledge of SQL by providing a graphical tool for searching and filtering queries and visualizations of query result sets.

1 INTRODUCTION

The new EU General Data Protection Regulation (GDPR) has an impact on every organization around the world [16], but most of them are prepared inadequately and are not aware of upcoming legal requirements [4]. They face the challenge of adapting their handling of personal data to become compliant to the contemporary requirements of data privacy. Art 5(1) GDPR postulates data minimization, which is one of the six general data-protection principles of the GDPR [1]. It requires that all collection and processing of personal data is only for a specific purpose, and that the quantity of these data is kept down as much as possible, so they are stored only as long as it is required to reach the intended purpose [15]. There are approaches like privacy by design [17] that include privacy protection in the overall conception of technical systems, but the vast majority of current systems does not consider these approaches. In order to ensure conformance to the principle of data minimization it is not sufficient to analyze the collected data sets only. This does not provide a possibility to verify that the data processing is exclusively for a specific purpose. But the responsible privacy officers often lack technical knowledge, which is usually required for the evaluation of data processing.

Problem Statement. In order to enable privacy officers to ensure data privacy in established systems, novel approaches are required, which provide mechanisms that do not premise profound technical knowledge to determine who collects personal data from where and who processes when which data for which purpose. The mechanisms must provide user-friendly interfaces and a data-processing description close to natural language [7].

Contribution. We demonstrate an extensible framework for query-driven data minimization in existing IT landscapes. We analyze SQL query logs and enrich queries with meta-information about the query structure, their environment, their execution, and their context. In contrast to currently trending approaches, queries are our first-class citizens for analysis of conformance to data privacy, instead of trawling through data stores that harbor vast amounts of personal data. Our framework aims to complement such approaches and supports privacy officers in minimizing user access to personal data, minimizing the storage of personal data to what is really necessary, and preventing future collection of unnecessary data.

Our framework is minimally-intrusive because it has no impact on productive operations. It can serve as a foundation of a system that automatically identifies unneeded schema relations and attributes in target databases (DBs), based on the queries running on these systems. Queries can be classified manually as legal or illegal regarding data-privacy regulations. User access rights can automatically be customized, and unnecessary data can be automatically deleted from the target DBs. Furthermore, we list the queries and the related users that have inserted unneeded data into the target DBs.

This paper describes the client-server architecture of our framework (cf. sec. 2.1), illustrates the conceptual schema for the query characteristics, i. e. the queries and their meta-information (cf. sec. 2.2), and outlines the reference implementation of the DataEconomist (cf. sec. 2.3). In [12], we have already described a user story from a healthcare scenario to emphasize our approach's benefits for ensuring data privacy, especially data minimization.

2 SYSTEM OVERVIEW

The software system is implemented with a client-server architecture using Web technologies. An end user can have three different roles: administrator, DB user, and privacy officer. Each of them needs to interact with the system, so it has to be accessible from different personal machines. Splitting the application into a client and server part allows for that and additionally simplifies initial setup for end users.

2.1 Client-Server Architecture

Internally, the DataEconomist server uses a PostgreSQL DB to persist query characteristics. It also connects with target DBs when needed to retrieve meta schemas required to understand the queries and execute them when requested. It supports all major relational DBMS and the Java JDBC API as well as standard SQL and several of its dialects.

Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.69

The multi-user Web client is platform independent and does not require any kind of installation for the end user. It communicates with the server over HTTP using a RESTful JSON API.

The backend is implemented in Java using Apache Calcite [6] and the Spring Framework[1]. Fig. 1 shows its architecture.

The *Query Manager* and the *Database Manager* take care of ensuring consistency between main-memory representations of Query and Database objects and their DB entries. We must store additional information about these objects to control the redundancy. For queries, this information includes syntactical correctness, the number of restrictions, compliance with user privileges, and other potentially interesting properties. For target DBs, we fetch and store the schema in Java objects on the server.

The *Query Controller* and the *Database Controller* perform actions requested by the API on the resp. objects. They can only cause updates on the internal DB by using a manager module.

The *Search Module* maps search requests from the visual search form onto the internal DB. On top of that, it makes use of the *Query Manager* to consider information only available on the server application.

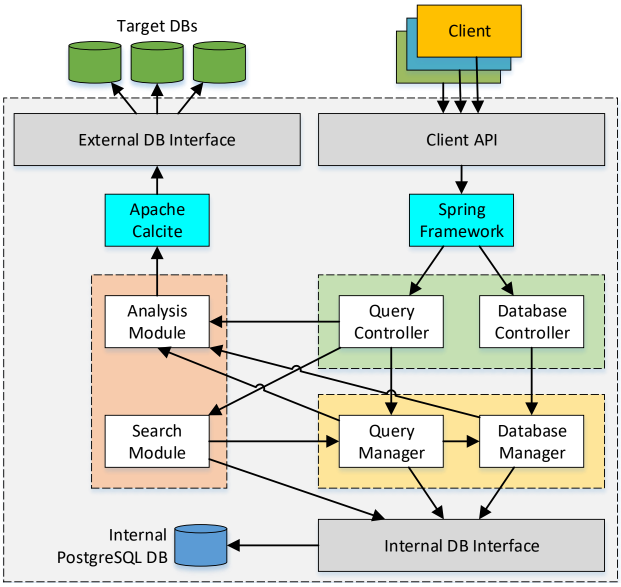


Figure 1: Architecture of the DataEconomist

2.2 Conceptual Schema

Fig. 2 shows an Entity-Relationship diagram containing the DataEconomist's conceptual schema to hold the queries and their meta-information. The Query entity is identified by the Text attribute, holding the query text. The complex, multi-valued attribute Relations stores the query's relations and attributes. It contains the two nested attributes RelName and Attributes. For each relation, we store its name in RelName and the names of all the relation's attributes in the multi-valued attribute Attributes.

The purpose of the Project entity is to group several target DBs in order to analyze their queries collectively. The Database entity holds connection information to target DBs.

A Query Execution reflects the run of a certain Query by a certain User on a certain Database at a certain Timestamp. The entity's further attributes are the RunTime in case of a successful query execution and the ErrorMessage if the query failed.

The Query Context points out a user's intention or the situation in which a Query Execution happens. Regarding this, privacy officers can specify the query's priority and its legitimacy concerning data privacy in this context. A query execution can only happen in one context, but a context may contain many query executions.

Fig. 2 highlights four groups of query characteristics in color. Some of them can be derived automatically; others must be entered manually. The Query Environment group is derived from the user and her initial connection to the target DB. The Query Execution group bundles when which query ran with what runtime successfully or not. Finally, the Query Context group cannot be derived automatically at all; it must be entered manually as it externalizes tacit user knowledge.

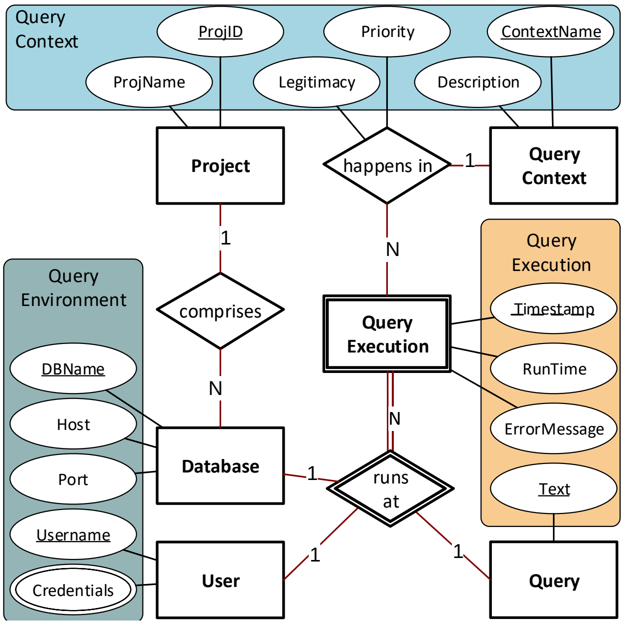


Figure 2: Conceptual schema of the DataEconomist

2.3 Reference Implementation

As first step, Apache Calcite performs basic syntax checking for each query. When checking which tables and columns are used by a query (a task that belongs to the core functionality of our DataEconomist), syntax alone is not sufficient. We additionally need the schema information. Asterisk selections are a trivial example to illustrate that. In order to detect used schema objects, Calcite's query planner transforms the query into a logical query plan. Calcite performs a query optimization and normalization. By applying the visitor pattern, the query-plan object is traversed and all used tables and columns are detected and collected for later review.

To provide the HTTP API, we use the model-view-controller module of the Spring framework. It maps HTTP requests onto the corresponding Java methods and serves as a bridge between Java objects and their JSON representation. The API mainly follows a CRUD approach, offering endpoints for entity creation, retrieval, update, and deletion.

[1] http://spring.io/projects/spring-framework

The frontend is implemented in Angular[2], a widely used Web framework written in TypeScript. It allows for the use of advanced GUI techniques such as reusable components and programming patterns such as reactive programming.

Fig. 3 shows the functionalities of the frontend interface. They are partitioned in three main views for projects, target DBs, and queries. The admin role can create projects and relate one or several target DBs. The admin needs to specify credentials for the target DBs so that the DataEconomist can connect to them and can fetch their schema information and import query logs. The admin role can also add and modify queries in our frontend, whereas the DB-user role can only choose a project and run the related queries.

Privacy officers can browse through all queries. We support them with a tool for extensive search-query formulation that can be utilized with purely visual means and does not require any knowledge of SQL. They can span a search-query tree by arbitrarily nesting search conditions concerning query structure or query characteristics. Any result set of queries can be manually filtered to be used in the next step for automatic detection of unneeded schema elements. Privacy officers can then check and adjust the DataEconomist's proposal of user-access-right minimization. The final configuration can be enforced to the target DBs with a single click.

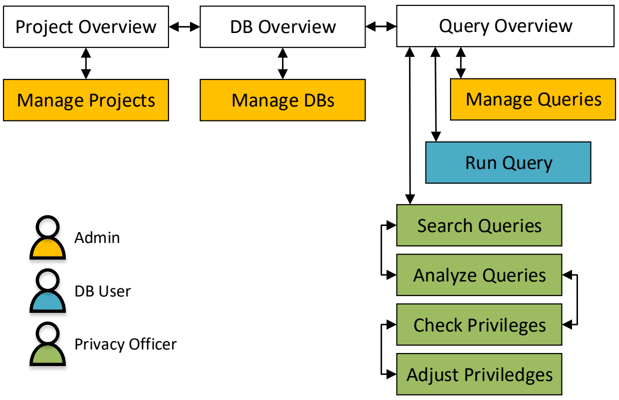


Figure 3: Frontend Use of the DataEconomist

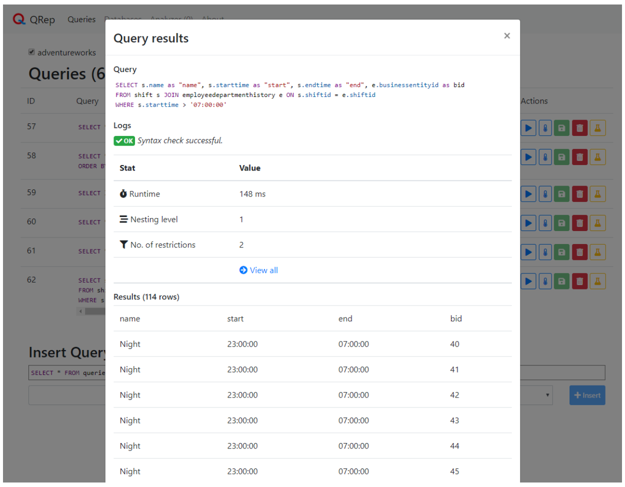
3 DEMONSTRATION WALKTHROUGH

For this demonstration, we will shortly present our approach's fundamental concepts to the conference attendees. Then they will be briefed to use our framework and learn about various aspects of the current version of our DataEconomist. Attendees may go through three different demonstration scenarios that build upon each other, but do not require the preceding steps for understanding. Most steps in the process include interaction with the attendees.

Setting up a project and analyzing queries. We have prepared a substantial SQL query log containing queries of different complexities from an enterprise scenario, stored in a MS-SQL DB. The attendees can choose the queries to be investigated. When a query is added to the DataEconomist, it will automatically start to find various properties of that query and to make them available for a potential search (see next scenario).

After the analyzing step is finished, attendees can see its results. The derived properties include the number of restrictions, the level of nesting, the number of different columns used by a query, and more.

Along with these properties, a sample of the query result is shown as well to help understand the query (see Fig. 4).



Query results

Query

```
SELECT s.name as "name", s.starttime as "start", s.endtime as "end", e.businessentityid as bid
FROM shift s JOIN employeedepartmenthistory e ON s.shiftid = e.shiftid
WHERE s.starttime > '07:00:00'
```

Logs

✓ OK *Syntax check successful.*

Stat	Value
Runtime	148 ms
Nesting level	1
No. of restrictions	2
	View all

Results (114 rows)

name	start	end	bid
Night	23:00:00	07:00:00	40
Night	23:00:00	07:00:00	41
Night	23:00:00	07:00:00	42
Night	23:00:00	07:00:00	43
Night	23:00:00	07:00:00	44
Night	23:00:00	07:00:00	45

Figure 4: Query stats and results dialog

Finding potentially illegal queries. For this scenario, we challenge the attendees to play the role of a privacy officer and to find queries violating data-privacy regulations. The attendees can use the visual-search interface and track down queries that retrieve for example data revealing racial or ethnic origin, political opinions, or religious or philosophical beliefs. To achieve a high level of flexibility, the visual-search queries can be nested and combined with Boolean operators (see fig. 5).

The attendees can directly consider accessible properties, such as the names of DB users, query-execution times, SQL strings, and all other query characteristics mentioned before.

We will examine two different cases that require different approaches. One requires the correct choice of properties to check. The other additionally requires a correct combination of properties. Both demonstrate the possibilities offered by the search-query formulation implemented in the DataEconomist.

Minimizing user privileges and deleting unnecessary data. When illegal queries have been found, we demonstrate how the DataEconomist displays which tables and columns are used by a query or a set thereof (see Fig. 6).

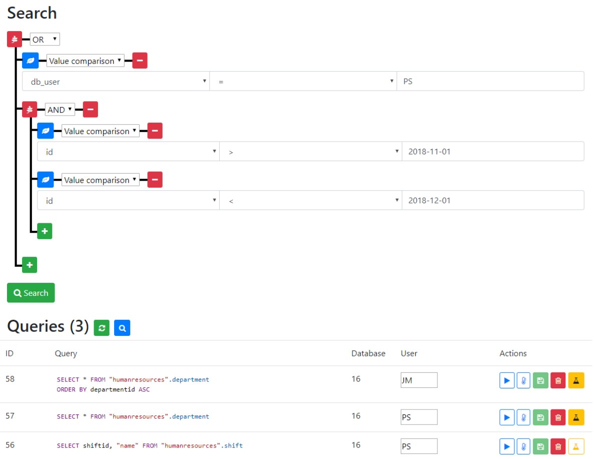
Attendees can then adapt the user privileges by unchecking tick boxes at all schema objects the user should no longer have access to. These user privileges will actually be enacted on the MS-SQL server through the interface. To show the effects, we will then try to execute an illegal query on the DB, which will fail, and inform the user about the illegal access attempt.

After that, attendees can use the DataEconomist to automatically determine unnecessary schema elements, i. e. data that have no user access to them. Again, by a single click, all these data will be deleted on the MS-SQL server.

4 STATE OF THE ART

Srivastava et al. provide a relational framework for managing queries [5]. We adopt their way of administrating queries together with their meta-information in a relational model. There are

[2] https://angular.io/



Search

db_user = PS

id > 2018-11-01

id < 2018-12-01

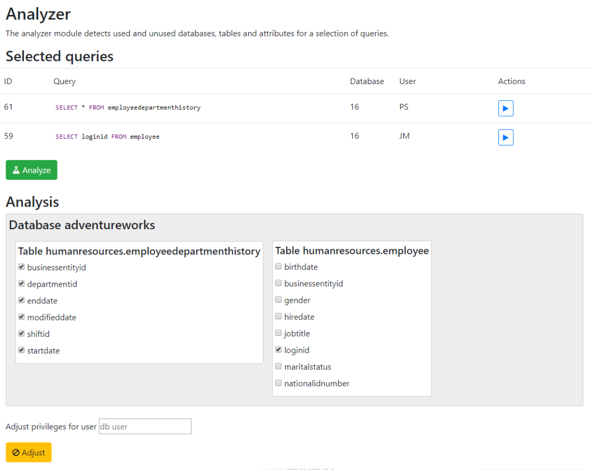
Queries (3)

ID	Query	Database	User	Actions

58	`SELECT * FROM "humanresources".department`
	`ORDER BY departmentid ASC`
		16	JM	

57	`SELECT * FROM "humanresources".department`	16	PS	
56	`SELECT shiftid, "name" FROM "humanresources".shift`	16	PS	

Figure 5: Visual query search form



Analyzer

The analyzer module detects used and unused databases, tables and attributes for a selection of queries.

Selected queries

ID	Query	Database	User	Actions
61	`SELECT * FROM employeedepartmenthistory`	16	PS	
59	`SELECT loginid FROM employee`	16	JM	

Analysis

Database adventureworks

Table humanresources.employeedepartmenthistory
- ☑ businessentityid
- ☑ departmentid
- ☑ enddate
- ☑ modifieddate
- ☑ shiftid
- ☑ startdate

Table humanresources.employee
- ☐ birthdate
- ☐ businessentityid
- ☐ gender
- ☐ hiredate
- ☐ jobtitle
- ☑ loginid
- ☐ maritalstatus
- ☐ nationalidnumber

Adjust privileges for user db user

Figure 6: Privilege adaption form

several systems dedicated to query management, e. g. [2, 10]. However, these tools are aimed at experts and lack usability for privacy officers. During the query-log analysis of [3], we considered the query structure and detected similarities among queries. However, we did not yet provide a way to report user-specific meta-information about queries.

QueRIE [9] performs a query-log analysis to give personalized query recommendations. Manta and SQLDep[3] visualize schema lineage of SQL queries. None of these approaches considers query context and other meta-information. This is also the case for several graph-based approaches to representing SQL queries, e. g. [8, 11]. Tools like the Query Patroller [14] or the DataLawyer [13] provide a wide range of mechanisms for query analysis. However, they aim at SQL experts and the latter is even limited to a particular DBMS.

5 CONCLUSION AND FUTURE WORK

The DataEconomist supports several aspects of data minimization: Based on the query-driven approach, user access to unneeded schema elements can be automatically customized. Completely unneeded data can be automatically detected and deleted. These features minimize data processing. In first measurements, we determined the average latency for analyzing the structure (117ms) and detecting unneeded schema elements (19ms) of a single query. The structure analysis includes the generation of the query execution plan, the detection works with this execution plan. We intend to present more detailed measurement results in a follow-up publication.

Our framework also points at queries inserting unneeded data and at these queries' authors. That allows privacy officers to take specific measures in minimizing data collection.

Up to now, privacy officers have to manually classify illegal queries regarding data privacy, which can be an outrageous effort. Therefore, we work on a semi-automatic classification of illegal queries to minimize user interaction. We are also enhancing our framework by an intuitive visualization of all data processed by a query to fully support non-expert SQL users in classification. Furthermore, we aim to additionally consider unneeded tuples for a more fine-grained data minimization.

Acknowledgement: The authors would like to thank the anonymous reviewers for their valuable remarks.

[3] https://getmanta.com/ and https://sqldep.com/

REFERENCES

[1] Council of European Union. 2016. Council regulation (EU) no 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *OJ* L 119 (4 5 2016), 1–88.

[2] Jan Van den Bussche et al. 2005. Towards practical meta-querying. *Inf. Syst.* 30, 4 (2005), 317–332.

[3] Andreas M. Wahl et al. 2018. A graph-based framework for analyzing SQL query logs. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Houston, TX, USA, June 10, 2018.* 11:1–11:5.

[4] Christina Tikkinen-Piri et al. 2018. EU General Data Protection Regulation: Changes and implications for personal data collecting companies. *Computer Law & Security Review* 34, 1 (feb 2018), 134–153.

[5] Divesh Srivastava et al. 2007. Intensional associations between data and metadata. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007.* 401–412.

[6] Edmon Begoli et al. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018.* 221–230.

[7] Fei Li et al. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *PVLDB* 8, 1 (2014), 73–84.

[8] George Papastefanatos et al. 2005. Hecataeus: A Framework for Representing SQL Constructs as Graphs. In *Proceedings of 10th International Workshop on Exploring Modeling Methods for Systems Analysis and Design-EMMSAD*, Vol. 5.

[9] Magdalini Eirinaki et al. 2014. QueRIE: Collaborative Database Exploration. *IEEE Trans. Knowl. Data Eng.* 26, 7 (2014), 1778–1790.

[10] Nodira Khoussainova et al. 2009. A Case for A Collaborative Query Management System. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings.*

[11] Nodira Khoussainova et al. 2010. SnipSuggest: Context-Aware Autocompletion for SQL. *PVLDB* 4, 1 (2010), 22–33.

[12] Peter K. Schwab et al. 2018. Towards Query-Driven Data Minimization. In *Proceedings of the Conference "Lernen, Wissen, Daten, Analysen", LWDA 2018, Mannheim, Germany, August 22-24, 2018.* 335–338.

[13] Prasang Upadhyaya et al. 2015. Automatic Enforcement of Data Use Policies with DataLawyer. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015.* 213–225.

[14] Sam Lightstone et al. 2002. Toward Autonomic Computing with DB2 Universal Database. *SIGMOD Record* 31, 3 (2002), 55–61.

[15] Thibaud Antignac et al. 2014. Privacy Architectures: Reasoning about Data Minimisation and Integrity. In *Security and Trust Management - 10th International Workshop, STM 2014, Wroclaw, Poland, September 10-11, 2014. Proceedings.* 17–32.

[16] Michelle Goddard. 2017. The EU General Data Protection Regulation (GDPR): European Regulation that has a Global Impact. *International Journal of Market Research* 59, 6 (nov 2017), 703–705.

[17] Marc Langheinrich. 2001. Privacy by Design - Principles of Privacy-Aware Ubiquitous Systems. In *Ubicomp 2001: Ubiquitous Computing, Third International Conference Atlanta, Georgia, USA, September 30 - October 2, 2001, Proceedings.* 273–291.

ITGC: Information-theoretic Grid-based Clustering

Sahar Behzadi
Faculty of Computer Science
University of Vienna
Vienna, Austria
sahar.behzadi@univie.ac.at

Hermann Hinterhauser
Faculty of Computer Science
University of Vienna
Vienna, Austria
a00946522@unet.univie.ac.at

Claudia Plant
Faculty of Computer Science and
ds:Univie
University of Vienna
Vienna, Austria
claudia.plant@univie.ac.at

ABSTRACT

Grid-based clustering algorithms are well-known due to their efficiency in terms of the fast processing time. On the other hand, when dealing with arbitrary shaped data sets, density-based methods are most of the time the best options. Accordingly, a combination of grid and density-based methods, where the advantages of both approaches are achievable, sounds interesting. However, most of the algorithms in these categories require a set of parameters to be specified while usually it is not trivial to appropriately set them. Thus, we propose an **I**nformation-**T**heoretic **G**rid-based **C**lustering (ITGC) algorithm by regarding the clustering as a data compression problem. That is, we merge the neighbour grid cells (clusters) when it pays off in terms of the compression cost. Our extensive synthetic and real-world experiments show the advantages of ITGC compared to the well-known clustering algorithms.

1 INTRODUCTION

Among various clustering approaches some of them attract more attentions because of their advantages. Partition-based clustering algorithms are popular due to their simplicity and the relative efficiency [7], [2]. K-means [7] is a well-know and well-studied representative for this approach where initially the data is partitioned into k non-empty sets and iteratively the data points are assigned to their nearest cluster. Despite the mentioned advantages, the clustering algorithms in this group suffer from some drawbacks. For instance, the number of clusters k should be specified in the beginning and the results are not deterministic because of their sensitivity to the initialization. Moreover, they are not suitable to discover clusters with non-convex shapes. As a subset of this group, model-based clustering algorithms consider a specific distribution model to represent the data sets. Among them, Expectation-Maximization (EM) algorithm interpret the data as a mixture of Gaussian distributions [5]. On the other hand, density-based clustering algorithms [6], [3] are appropriately designed to deal with clusters having an arbitrary shape. Unlike the partition-based algorithms, the algorithms in this approach are able to deal with noisy data sets. However, in order to find dense regions we need to specify two parameters representing the radius and the density of a neighborhood. Additionally, density-based algorithms are not designed to efficiently deal with clusters with various densities. Spectral clustering [9] is another approach which has become popular due to its simple implementation and its performance in many graph-based clustering. It can be solved efficiently by any standard linear algebra software. However, this approach is expensive for the large data sets since the Computing eigenvectors is the bottleneck.

Another well-known approach is grid-based clustering where any data set is partitioned using a set of grid-cells and data points are assigned to the appropriate grid cell. Grid-based methods [1], [14], [11] quantize the object space into a finite number of cells (hyper-rectangles) and then perform the required operations on the quantized space. The main advantage of grid-based methods is their fast processing time which depends on the number of cells in the grid. In the other word, no distance computation is required and the clustering is performed on summaries and not on the individual objects. Thus, the complexity of grid-based algorithms is usually *O(number of populated grid cells)* and not *O(number of objects)*. Beyond their ability to deal with noisy data sets, grid-based clustering algorithms are able to identity clusters irrespective of their shapes. Unlike most of the clustering algorithm which require an initialization phase, the algorithms in this category are insensitive to the order of input records and therefore are deterministic.

Despite the valuable advantages of grid-based clustering algorithms, to the best of our knowledge, all of them are parametric algorithms where a user is required to specify the parameters. However, most of the time it is not trivial to appropriately set them. Thus, utilizing the principle of Minimum Description Length (MDL) we propose a non-parametric **I**nformation-**T**heoretic **G**rid-based **C**lustering algorithm where we regard the clustering task as a data compression problem so that the best clustering is linked to the strongest data compression. First, an adaptive grid is constructed corresponding to the statistical characteristics of any data set and non-empty cells are considered as single clusters. Then, we combine the concept of density and grid-based methods and employing our compression-based objective function we start merging clusters with their neighbour grid cells only if it pays off in terms of the compression cost.

In this paper we propose an information-theoretic clustering algorithm offering the following contributions:

- **Adaptive partitioning:** We utilize the statistical characteristics of any data set, e.g. local and global dispersion, in order to introduce an adaptive partitioning of the data.
- **Non-parametric clustering:** Employing the MDL-based objective function, we iteratively merge clusters when it pays off in terms of the compression cost automatically. Thus, no parameter needs to be specified.
- **Insensitivity to the shape of clusters:** ITGC employs the concept of density-based methods in order to select the next merging candidate. Thus, it is insensitive to the shape of clusters whether they are Gaussian, arbitrary or even having various density regions.
- **Scalability:** Analogous to other grid-based clustering algorithm, the complexity of ITGC depends on the number of cells not on the number of objects which leads to a scalable algorithm in terms of the number of objects.

2 INFORMATION-THEORETIC GRID-BASED CLUSTERING

In order to introduce a grid-based clustering algorithm we need to address two fundamental questions: (1) how to find a specific appropriate partitioning (grid) corresponding to any data set; (2) how to efficiently merge the cells to discover the hidden clusters. Thus, our proposed algorithm ITGC consists of two main building blocks: (1) finding a suitable grid corresponding to specific characteristics of any data set and (2) employing MDL principle to effectively and efficiently merge the cells without any parameter to be specified.

2.1 Partitioning the Data

Finding a suitable partitioning with respect to the data is a crucial task in a grid-based clustering algorithm. Inspired by [8], we utilize the characteristics of any data set to introduce the best fitting partition. That is, we are looking for a partition which leads to high internal homogeneity in the cells and high external heterogeneity of each cell with respect to its neighbors for every single cell. Thus, for any cell C_j consisting of n_j data points the statistical indicators are defined as:

$$\bar{X_j} = \frac{\sum_{i=1}^{n_j} X_{ij}}{n_j} \quad and \quad S_j = \sqrt{\frac{\sum_{i=1}^{n_j} (X_{ij} - \bar{X_j})^2}{n_j - 1}} \tag{1}$$

Where X_{ij} is the distance of the $i - th$ data point in C_j to the center of this cell. Thus, $\bar{X_j}$ is the average distance of data points to the center of C_j and S_j is the standard deviation of the cell. These are statistical indicators on the local level (each individual cell), similar indicators are calculated on the global level (the entire grid) as:

$$\mu = \frac{\sum_j \bar{X_j}}{N} \quad and \quad \sigma = \sqrt{\frac{\sum_j (\bar{X_j} - \mu)^2}{N - 1}} \tag{2}$$

Where μ is the average center - distance of all cells, N is the total number of cells and σ is the standard deviation of all cells. Based on these indicators, we define $CV_{Local}(j)$ and CV_{Global} as the coefficient of variation (CV) corresponding to any cell C_j and the global variation, respectively. That is,

$$CV_{Local(j)} = \frac{S_j}{\bar{X_j}} \quad and \quad CV_{Global} = \frac{\sigma}{\mu} \tag{3}$$

In another point of view, the above scores show how widespread the data points are indicating the relative dispersion at the local (cell) and global (grid) levels. Finally the partitioning cost is defined as:

$$gridCost = \frac{CV_{Global}}{avg \quad CV_{Local}(j)} \tag{4}$$

Considering the cost corresponding to any grid size $k \times k$, we iteratively increase k starting from 1 until it pays off. However, it is not trivial to justify a terminal for this process without observing its trend. Initially, the grid cost increases sharply by increasing k then it slows down quickly and continues linearly. Observing this common trend, we conduct a simple linear regression on various costs with respect to various k values. The regression line is expected to fit more through the higher ks where the costs have lower deviations. Thus, the optimal partitioning can be set to the first k where the grid cost deviated from the fitted line lower than the average.

On the other side, by increasing the size of grid the area of non-empty cells decreases. Thus, it is reasonable to assume this trend to continue with even smaller cells, but the descending trend slows down while decreasing cell sizes. Visualizations of the collected area reveals a common trend which is reverse to the previous one. The area starts at a maximum value, decreases very sharply at lower k and keeps decreasing at a lower gradient. In order to find the optimum value for k, we analogously fit a linear regression through the data set rejecting the low k values which deviate larger than average from the fitted line. The following steps summarize this procedure.

- Step 1: The grid is divided into $k \times k$ cells where the initial size for k is 1.
- Step 2: The grid cost as well as the area of non-empty cells are determined and the values are stored.
- Step 3: We iteratively increase k ranging from 1 to a max_k and repeat the previous steps
- Step 4: Now the optimum partitioning is determined employing two different criteria.

2.2 MDL-based Objective Function

Utilizing the Minimum Description Length (MDL) principle [10] we regard the clustering task as a data compression problem so that the best clustering is linked to the strongest data compression. Given the appropriate model corresponding to any attribute, MDL leads to an intuitive clustering result employing the compression cost as a clustering criterion. The better the model matches major characteristics of the data, the better the result is. Following the MDL principle, we encode not only the data but also the model itself and minimize the overall description length. Simultaneously, we avoid over-fitting since the MDL tends to a natural trade-off between model complexity and the goodness-of-fit. That is, for a given cluster C_i the corresponding compression cost is defined as:

$$MDL(C_i) = CodingCost(C_i) + ParamCost(C_i) + IDCost(C_i) \tag{5}$$

where *CodingCost* shows the cost of coding the data points in cluster C_i by means of a coding scheme. The next two terms illustrate the model complexity where the model itself needs to be encoded. In this paper we employ the Huffman coding scheme to encode the data considering an appropriate model. That is, given the corresponding *Probability Distribution Function* (PDF) to any attribute, the coding cost of any object x is determined by $-log_2 PDF(x)$. Any PDF would be applicable and using a specific model is not a restriction [4] for our algorithm. In this paper, we consider Gaussian PDF for simplicity. In the following we elaborate our objective function more concretely.

- **Objective Function:** The overall MDL-based objective function is the summation of the all compression costs with respect to various clusters. That is,

$$MDL(\mathcal{D}) = \sum_{C_i \in C} MDL(C_i) \tag{6}$$

where $\mathcal{D}$ is the entire data set and $C = \{C_1, ..., C_k\}$ is the set of all clusters.

- **Data Coding Cost:** Let $\mathcal{X} = \{X_1, ..., X_d\}$ denote the set of all attributes. For any object $x = (x_1, ..., x_d)$ the corresponding coding cost is the sum of encoding any attribute value x_i. Putting all together, the coding cost corresponding to cluster C_i is given by:

$$CodingCost(C_i) = -\sum_{X_j \in \mathcal{X}} \sum_{x \in C_i} \log_2 PDF_j(x) \qquad (7)$$

where $PDF_j(.)$ is the Gaussian model with respect to $j-th$ attribute X_j.

- **Model Complexity:** Without taking the model complexity into account, the best result will be a clustering consisting of singleton clusters. This result is completely useless in terms of the interpretation. In order to specify the associated cluster with any data object, we need to encode the cluster IDs. Thus, the IDCosts are required to balance the size of clusters and defined as:

$$IDCost(C_i) = |C_i|.log_2 \frac{|C_i|}{|\mathcal{D}|} \qquad (8)$$

Following the fundamental results from the information theory [10], for any attribute X_j the parameters corresponding to model employed to encode the data need to be encoded as well. That is, concerning any Gaussian distribution PDF_j with respect to the attribute X_j, the mean value and the standard deviation need to be encoded, i.e.

$$ParamCost(C_i) = \frac{1}{2}.(2|\mathcal{X}|).log_2|C_i| \qquad (9)$$

2.3 Algorithm

As mentioned, ITGC consists of two main building blocks. Algorithm 1 summarizes our grid-based algorithm ITGC. First, an optimal grid is constructed following the steps mentioned in Section 2.1, i.e. the procedure *FindOptimumGrid(.)*. Then, we start merging the cells if it pays off in terms of our objective function (Section 2.2). Initially every cell is considered as a cluster while empty cells are ignored. The cluster with the most number of data points is chosen in the sense that at the end the results are deterministic. We compute the coding cost with respect to the selected cluster MDL_{before} and merge this cluster with one of its neighbors and compute the cost after merging two clusters MDL_{after}. If the cost after merging is smaller than the cost before, we merge two clusters and continue the merging process. Otherwise, the visited cluster is marked. Finally the algorithm terminates if no unmarked non-empty cell exists.

3 EXPERIMENTS

In this section we assess the performance of ITGC comparing to other clustering algorithms in terms of *Normalized Mutual Information* (NMI) which is a common evaluation measure for clustering results [13]. NMI numerically evaluates pairwise mutual information between ground truth and resulted clusters scaling between zero and one.

We conducted several experiments evaluating our algorithm on synthetic and real-world data sets. In order to investigate the effectiveness of ITGC we generated various data sets and compared to the base-line clustering algorithms, i.e. k-means [7] and DBSCAN [6]. While the insensitivity of ITGC to the shape of clusters as well as its effectiveness is illustrated by synthetic experiments, we extended the comparison to the wider range of well-known clustering algorithms. Our algorithm is implemented in Java and the source code as well as the data sets are publicly available [1].

Algorithm 1: Information-theoretic grid-based clustering

```
ITGC ($\mathcal{D}$)
$\mathcal{G}$ = FindOptimumGrid($\mathcal{D}$);
$C = \{C_1, ..., C_k\}$        // Non-empty cells in $\mathcal{G}$
seeds := non-visited clusters;
while (seeds != empty) do
    $C_i$ := the cluster with the most data points in $C$
    $C_i$ is visited
    while ($MDL_{before} > MDL_{after}$) do
        $MDL_{before} = MDL(C_i)$
        $C_j$ := a random non-visited neighbor cell w.r.t $C_i$
        $C_m$ := the cluster after merging $C_i$ and $C_j$
        $MDL_{after} = MDL(C_m)$
        if $MDL_{before} > MDL_{after}$ then
            remove $C_j$ and $C_i$ from $C$ .
            add $C_m$ to $C$
        end if
    end while
    seeds := non-visited clusters;
end while
return ($C$)
```

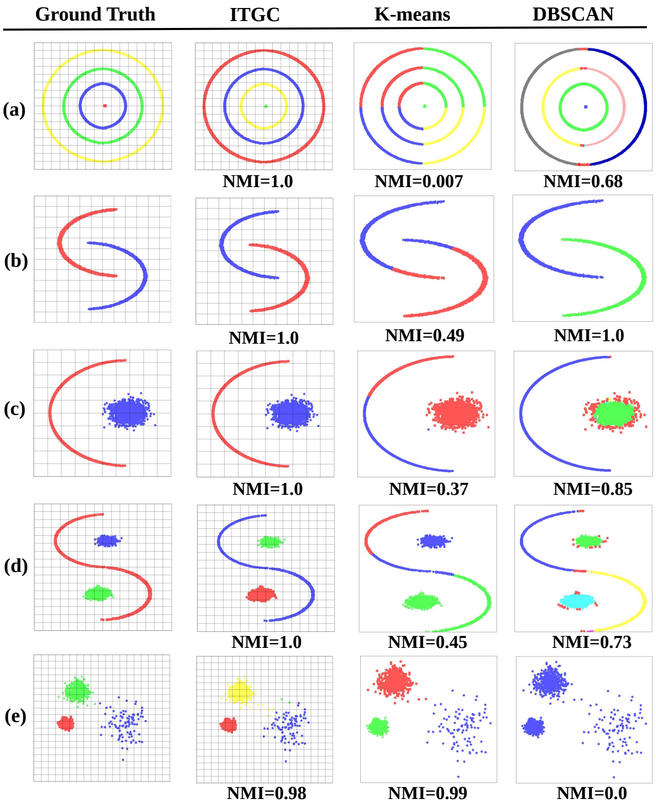


Figure 1: Comparison on various synthetic data sets.

3.1 Synthetic Experiments

In order to cover all aspects of ITGC, we investigate the performance of the algorithms considering various synthetic data sets including arbitrary shaped data sets as well as clusters with different densities. Then, we continue experiments by comparing all algorithms in terms of the scalability.

Performance: Most of the time any clustering algorithm is designed for a specific kind of data sets. For instance, k-means

[1] https://tinyurl.com/y85gglpx

Dataset	*Attr./Objects*	*ITGC*	*k-means*	*DBSCAN*	*EM*	*Spectral C.*	*CLIQUE*	*Single L.*
Iris	4/150	**0.66**	0.53	0.59	0.60	0.6	0.00	0.59
Occupancy Detection	7/20560	**0.61**	0.56	0.00	0.31	0.00	**0.61**	-
Breast Cancer	9/286	**0.47**	0.32	0.41	0.45	0.45	0.39	0.27
User Knowledge	5/403	0.24	**0.27**	0.01	**0.27**	**0.27**	0.00	0.01

Table 1: Comparison on real data sets.

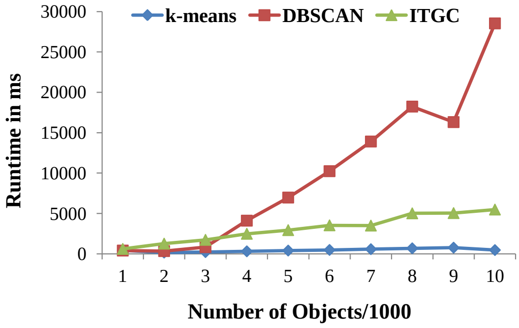


Figure 2: Scalability of the algorithms by increasing the number of objects.

appropriately deals with Gaussian shaped data sets and its performance dramatically decreases when the clusters have no specific shape. On the other hand density-based clustering algorithms are sensitive to different densities w.r.t. various clusters. In order to evaluate the performance of ITGC concerning various shapes of clusters, we synthetically generated arbitrary shaped clusters in a combination with some Gaussian clusters. Figure 1 shows the effectiveness and the insensitivity of ITGC considering various cases. As expected, k-means fails when the clusters are not Gaussian (Figure 1a,b,c,d). On the other hand, DBSCAN is not able to discover the clusters with various densities (Figure 1d,e)

Scalability: To evaluate the efficiency in terms of the runtime complexity we generated 5 dimensional synthetic data sets where we iteratively increased the number of data objects ranging from 1,000 to 10,000. Figure 2 shows the result of this experiment. As expected, k-means is the fastest algorithm while DBSCAN is the worse since its complexity highly depends on the number of objects. Although ITGC is not able to outperform k-means, its corresponding execution time is still reasonable and more efficient than DBSCAN.

3.2 Real Experiments

In this section we extend our experiments to the wider range of clustering algorithms including EM [5], Single link [12], spectral clustering [9] and CLIQUE [1] as the well-known representatives for any clustering approach. We evaluate clustering quality of ITGC on real-world data sets. We used *Iris*, *Occupancy Detection*, *User Knowledge* and *Breast Cancer* data sets from the UCI Repository [2]. Table 1 shows the characteristics of any data set and the results of applying various algorithms in terms of NMI. Concerning any data set the best NMI is high lighted and when getting "Out Of Memory" error we inserted "-" in the table. As Table 1 illustrates ITGC outperforms other algorithms considering the first 3 real-world data sets. Interestingly, in this experiment we outperform CLIQUE which is a well-known grid and density-based clustering algorithm (the results are similar on the *Occupancy* data set). Although some of the comparison methods perform slightly better than ITGC on *User Knowledge* data set, our result is still comparable and we outperform DBSCAN, CLIQUE and Single link.

4 CONCLUSION AND FUTURE WORKS

In this paper we propose an information-theoretic clustering algorithm, ITGC, utilizing the MDL-principle. Firstly, We employ the statistical characteristics of any data set to appropriately partition the data without any presumptions. Then, an MDL-based objective function is proposed to iteratively merge the neighbour clusters when it pays of in terms of the compression cost of the clusters. Our experiments on synthetic and real-world data sets show the advantages of our proposed algorithm compared to other well-known clustering algorithms. Similar to other grid-based clustering algorithms, our algorithm may lead to inefficiency when dealing with huge data sets in terms of the dimensionality. Thus, a possible future work would be to investigate the parallelization approaches in the sense that the required memory to store the grid information could be distributed. As another option for the further investigation could be to enhance the partitioning procedure in the sense that it results a sparse grid which is cheaper in terms of the memory.

[2] http://archive.ics.uci.edu/ml/index.php

REFERENCES

[1] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. 1998. Automatic subspace clustering of high dimensional data for data mining applications. In *SIGMOD Conference*. 94–105.

[2] Paul E. Green Anil Chaturvedi and J. Douglas Caroll. 2001. K-modes Clustering. *Journal of Classification* 18, 1 (2001).

[3] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points to Identify the Clustering Structure *(SIGMOD '99)*. New York, NY, USA.

[4] Christian Böhm, Christos Faloutsos, Jia Pan, and Claudia Plant. 2006. Robust information-theoretic clustering. In *KDD*.

[5] A. P. Dempster, N. M. Laird, and D. B. Rubin. 1977. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)* 39, 1 (1977), 1–38.

[6] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise.. In *KDD Conference*.

[7] J. Macqueen. 1967. Some methods for classification and analysis of multivariate observations. In *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*. 281–297.

[8] Joel D. Melo, Edgar M. Carreno, Aida Clavino, and Antonio Padilha-Feltrin. 2014. Determining spatial resolution in spatial load forecasting using a grid-based model. *Electric Power Systems Research* 111 (2014), 177–184.

[9] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. 2001. On Spectral Clustering: Analysis and an Algorithm *(NIPS'01)*.

[10] Jorma Rissanen. 2005. *An Introduction to the MDL Principle*. Technical Report. Helsinkin Institute for Information Technology.

[11] Gholamhosein Sheikholeslami, Surojit Chatterjee, and Aidong Zhang. 1998. WaveCluster: A Multi-Resolution Clustering Approach for Very Large Spatial Databases *(VLDB '98)*. San Francisco, CA, USA.

[12] R. Sibson. 1973. SLINK: An optimally efficient algorithm for the single-link cluster method. *Comput. J.* 16, 1 (1973), 30–34.

[13] Nguyen Xuan Vinh, Julien Epps, and James Bailey. 2009. Information theoretic measures for clusterings comparison: is a correction for chance necessary?. In *ICML Conference'09*.

[14] Wei Wang, Jiong Yang, and Richard R. Muntz. 1997. STING: A Statistical Information Grid Approach to Spatial Data Mining *(VLDB '97)*. San Francisco, CA, USA, 186–195.

Adaptive Watermarks: A Concept Drift-based Approach for Predicting Event-Time Progress in Data Streams

Ahmed Awad
University of Tartu, Estonia
ahmed.awad@ut.ee

Jonas Traub
Technische Universität Berlin
jonas.traub@tu-berlin.de

Sherif Sakr
University of Tartu, Estonia
sherif.sakr@ut.ee

ABSTRACT

Event-time based stream processing is concerned with analyzing data with respect to its generation time. In most of the cases, data gets delayed during its journey from the source(s) to the stream processing engine. This is known as *late* data arrival. Among the different approaches for out-of-order stream processing, low watermarks are proposed to inject *special* records within data streams, i.e., watermarks. A watermark is a timestamp which indicates that no data with a timestamp older than the watermark should be observed later on. Any element as such is considered a late arrival. Watermark generation is usually *periodic* and heuristic-based. The limitation of such watermark generation strategy is its rigidness regarding the frequency of data arrival as well as the delay that data may encounter. In this paper, we propose an adaptive watermark generation strategy. Our strategy decides adaptively when to generate watermarks and with what timestamp without a priori adjustment. We treat changes in data arrival frequency and changes in delays as concept drifts in stream data mining. We use an Adaptive Window (ADWIN) as our concept drift sensor for the change in the distribution of arrival rate and delay. We have implemented our approach on top of Apache Flink. We compare our approach with periodic watermark generation using two real-life data sets. Our results show that adaptive watermarks achieve a lower average latency by triggering windows earlier and a lower rate of dropped elements by delaying watermarks when out-of-order data is expected.

1 INTRODUCTION

Stream analytics is concerned with analyzing data on the move. Several stream processing engines (SPEs) have been developed that vary in their processing capabilities. One fundamental feature is the ability to process data with respect to their generation time rather than their arrival time at the SPE [2]. This is commonly known as event-time versus processing-time. For a data stream element e, we define $te(e)$ as a function that returns the timestamp of the event, i.e. the time it was created at the source, we also define $tp(e)$ to be the timestamp when it was first seen by the SPE. The event-time notion is more relevant in several applications. However, as the data sources are distributed and can be placed far away from SPEs, *delays* can occur until an element e reaches the SPE. This is known as *late arrival.* Moreover, a stream element can arrive *out-of-order.* For example, two stream elements e_1 and e_2 where e_1 was created before e_2, i.e., $te(e_1) < te(e_2)$ can arrive in the opposite order, i.e., $tp(e_2) < tp(e_1)$. This is known as out-of-order arrival.

Stream analytics jobs are usually defined by a sort of a window [4, 5, 7]. Windows are used to slice the infinite stream into finite chunks and apply the analytics computation, e.g. averaging, a.k.a window function, on the content of the window. Time windows are common types of windows that define stream element's membership to the window based on the elements timestamp w.r.t. window start and end times, when working in event time. The window function cannot be applied on the window content until the SPE decides that the window is closed. For system (processing) time mode, the decision is made when the internal clock (wall clock) of the SPE passes the window end. However, for event time processing, we need an external notion progress.

Low watermarks generation [1, 2] is a technique to account for the progress in event time. A watermark is a timestamp that *indicates* that there should be no future data older than the watermark to be seen. When a watermark is received by a window operator in a pipeline, it triggers executions of completed time windows whose closure completed at a time earlier than the watermark. A watermark is monotonic. That is, a new watermark cannot be less than the last generated watermark. Watermark progress affects two metrics in stream processing: latency and late arrival of data items and thus accuracy. Latency is hurt when too few watermarks are generated. Late arrival increases when too many watermarks are generated and thus accuracy decreases due to more elements being ignored (dropped).

Currently, watermark generation is either heuristic-based and periodic [1, 2] or punctuation-based [11]. The latter assumes knowledge about the content of the data. The former is inflexible with the change of the distribution of data arrival rate or the distribution of the delays in arrival. Those are unique properties of data streams. In this paper, we contribute an approach to watermark generation that is data-content-agnostic. We call our approach as adaptive watermarks. By treating the change in the skewness between stream elements event time and processing time as a concept drift, we employ a drift detection technique, the adaptive window (ADWIN) [3, 6], to decide when to generate a new watermark and with which value. Moreover, we provide another control, late arrival threshold, to decide about the watermark generation. We implement our approach on Apache Flink and compare it with a baseline periodic watermark generator using two real-life data sets. Our experiments empirically prove the superiority of adaptive watermarks with respect to a reduced latency and/or a reduced number of dropped elements.

2 ADAPTING TO DATA ARRIVAL RATES

2.1 Baseline: Periodic Watermark Generation

Periodic watermark generation is a heuristic-based approach. Heuristically, application developers determine a max allowed lateness m. With the arrival of a new event, the *maximum* timestamp seen is updated. Then, every s milliseconds, the SPE obtains the maximum timestamp t seen so far. The watermark value is $t - m$. New stream elements that arrive after the watermark are considered late. Depending on the configuration of the stream processing pipeline, such elements can still be included in the

Series ISSN: 2367-2005

10.5441/002/edbt.2019.71

Table 1: Parameters used for watermark generators

Parameter	Description
δ^*	Sensitivity to change $\in [0, 1]$. Default is 1.
l	Late arrival threshold, $l \in (0, 1]$. Default is 1.
m^*	Skewness between event time and ingestion time.
Δ_δ	Sensitivity change ratio, $\Delta_\delta \in (0, 1]$. Default is 1.
w	Warmup tuples used to initialize m.

Parameters with * are derived by the system

result or ignored. This approach is simple to implement. However, it is does not adapt to changes in 1) the arrival rate of data elements, 2) the lateness of elements.

2.2 Adaptive Watermark Generation

In practice, it would be desirable to learn both m, the lateness, and the period s for generating a new watermark and keep updating them as new elements arrive. Moreover, to account for high correctness, one would hold the generation of new watermarks if the ratio of late arrival elements to the total elements goes above a certain threshold l since the last generated watermark. In this situation, the period s at which a watermark is generated will change (adapt) according to the change in the distribution of data inter-arrival time. Similarly, m can be learned upon each change detection. The change of the inter-arrival time can be seen as a concept drift in data streams [10]. In our case, the drift to be detected is the change of events' inter-arrival time. To detect this drift, we employ the ADWIN algorithm [3].

ADWIN is an algorithm which detects concept drifts (e.g., changes in users opinions) and enables adapting machine learning models on data streams. It works by maintaining a window of data items over time. The size of the window changes over time based on the frequency of change detected in incoming data. The algorithm does not require a pre-determined period to trigger change detection. It checks for drifts on a per-tuple basis. The more change occurs, the smaller the size of the window as older items are considered irrelevant and dropped. The algorithm has a single parameter δ (Table 1) which controls the sensitivity to change. The higher the value of δ the more sensitive it will be to change. In our work, δ is by default set to 1 so that a change is detected as early as possible.

ADWIN works as follows: Upon the arrival of a tuple, it is added to the window. Then, the algorithm tries to detect a change by finding two sub-windows whose value distributions are *significantly different* with respect to δ. Here, ADWIN iterates over all possible combinations of sub-windows. As an optimization, ADWIN maintains histograms (buckets) of the sum and variance of the data rather than the data itself to have a better memory footprint. The histogram grows logarithmically to the number of data points. Grulich et al. [6] parallelize different parts of the original algorithm and reach two orders of magnitude enhancement of its throughput. Table 1 summarizes the parameters we use for the adaptive watermark generation technique.

Algorithm 1 describes how we employ change detection by ADWIN to adapt the generation of watermarks. At the arrival of a new element, that might be late, the *difference* between the new element's timestamp and its ingestion timestamp is inserted into ADWIN and a check is made for detection of a change (drift) (Line 12). Such a detection can be an indicator to generate a new watermark. It is not necessary that every change detection leads to the generation of a new watermark. We need to look at the second indicator which is the rate of late arrivals. A new watermark is generated only if this rate is less than the threshold l at the time of change detection (Line 15). This rate is reset each time a new watermark is generated.

Algorithm 1: Adaptive watermark generation

```
   Input: A data stream $S$
   Input: Sensitivity change rate $\Delta_\delta$
   Input: Late arrival threshold $l$
   Input: Warmup tuples count $w$
1  $warmup \leftarrow 0$; $m \leftarrow 0$; $watermark \leftarrow 0$;
2  $lateElements \leftarrow 0$; $totalElements \leftarrow 0$; $\delta \leftarrow 1$
3  $maxTimestamp = -\infty$
4  $adWin \leftarrow initializeAdWin(\delta)$
5  foreach $e \in S$ do
6      $maxTimestamp = max(te(e), maxTimestamp)$
7      if $warmup \leq w$ then
8          $m \leftarrow max(m, tp(e) - te(e))$
9          $warmup \leftarrow warmup + 1$
10     else
11         $totalElements \leftarrow totalElements + 1$
12         if $adWin.driftDetected((tp(e) - te(e))/m, \delta)$
            then
13             if $lateElements = 0$ then
14                 $\delta = increaseSensitivity(\Delta_\delta)$
15             if $lateElements/totalElements < l$ then
16                 $watermark = maxTimestamp - m$
17                 $emit(watermark)$
18                 $lateElements \leftarrow 0$
19                 $totalElements \leftarrow 0$
20             else
21                 $m \leftarrow updateSkewness()$
22                 $\delta = decreaseSensitivity(\Delta_\delta)$
23         else
24             if $te(e) < watermark$ then
25                 $lateElements \leftarrow lateElements + 1$
```

We call the elements collected between two successive watermarks a *chunk*. The value of m is the maximum difference between elements' ingestion and event time within the chunk (Line 21). The value of the new watermark is $maximum\ timestamp - m$ (Line 16). Upon the generation of a new watermark, all data about late arrival is reset (Lines 18 and 19). In case a chunk has no late arrivals, the sensitivity is increased (Line 14) to speedup change detection. However, upon crossing the late arrival threshold l, δ is adapted according to the δ_Δ parameter (Line 22).

As ADWIN is designed to work with values in the range $[0, 1]$, we need to normalize the deviation between element's event and ingestion times. For this, we use the first w tuples of the stream to learn about m (Line 7). That can be in the form of minimum, maximum, or average skewness observed between elements' event time and ingestion time.

For the adaptive watermark generation, the user needs to specify the late arrival threshold l, the sensitivity change ratio δ_Δ, and the warmup tuples w. The default values for these parameters are 1, 1, and 10000 tuples, respectively. The choice of a value of l is mainly driven by the accuracy expected by the application. The more accurate results expected, the lower the value of l should

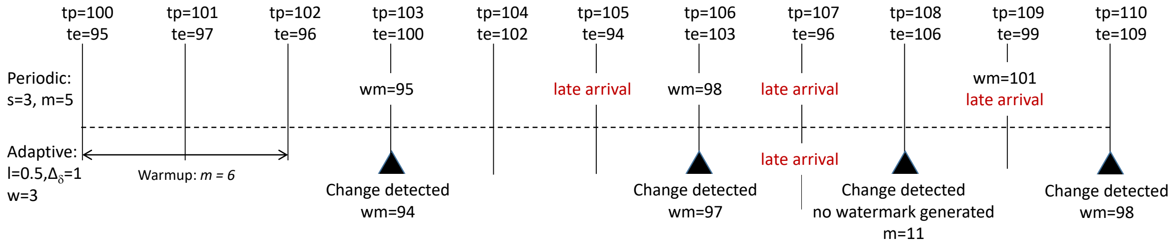


Figure 1: Example for adaptive watermark generation versus periodic generation.

be. If the source is stable and does not produce considerable late arrivals, the low value of l should not affect the latency of the results. Otherwise, lower values of l shall affect the latency, but this should be acceptable as the main concern is accuracy. Sensitivity to change δ_Δ is effective only in case that late arrivals rate is above l. In such case, δ is decreased and thus ADWIN will have to observe a larger number of elements before detecting a change. The value of w affects latency as for the first w tuples, no watermarks are generated and thus no window contents is processed. Larger value of w could be used with data sources with out-of-order arrivals. In Algorithm 1, we use the *maximum* skewness observed to update m. However, this can be changed to *average* or other measures.

Example: Figure 1 exemplifies how adaptive watermark generation works in comparison to the periodic one on a hypothetical stream. Vertical lines represent stream elements with their processing and event time respectively. The periodic generator is configured with $s = 3$ seconds and $m = 5$ seconds. The adaptive generator is configured with $l = 0.5$, $\Delta_\delta = 1$ and $w = 3$. The periodic generator will generate watermarks at $tp = 103$, $tp = 106$, and $tp = 109$ with values 95, 98, and 101, respectively. The elements arriving at $tp = 105$, $tp = 107$ and $tp = 109$ are late. For the adaptive generator, the first three tuples are used to initialize m; which is set to 6, as the max skewness observed between tp and te. At $tp = 103$, ADWIN detects a change. As there are no late arrivals, a watermark is generated with the value 94. At time $tp = 106$, another change is detected and a new watermark is generated with value 97. The next element at $tp = 107$ is late. ADWIN detects another change at time $tp = 108$. At this time, no watermark is generated as the late arrival ratio is equal to 0.5. Also, at this time, m is updated to 11, the skewness between $tp = 107$ and $te = 96$. At $tp = 110$, ADWIN detects another change. A watermark is generated as the late arrival ratio drops below 0.5.

3 EVALUATION

Setup: We have implemented adaptive watermark generation on top of Apache Flink v1.6.2, using the `Source` APIs to control the emission of watermarks. Our code base for the implementation and experimental evaluation can be found online[1]. We run our experiments on a standalone cluster with 6 *GB* of main memory and 4 cores at 2 *GHz*.

Data and query: For the comparison between adaptive and periodic watermark generation, we use two data sets from the *DEBS* grand challenges of 2012 [8] and 2015 [9] respectively. The *DEBS 2012* data set has 32,390,519 tuples and 1.5% of the elements arrive out-of-order with an average of 100 tuples per second. The *DEBS 2015* data set has 14,776,616 tuples and has 78.6% of its tuples arrive out-of-order. Moreover, the arrival rate of the data varies along the data set with an average of 6 tuples per second. For both of the data sets, we project the timestamps and create events with dummy content and the timestamps projected. We use a simple pipeline that applies a count function on the content of a time window. We report the window boundary, start and end, the number of elements in the window and the difference between window end and the watermark. The resulting tuples are of the form $\langle start, end, count, delay \rangle$. We use these values to construct the comparison metrics as we show next.

Table 2: Metrics for periodic watermark generation

Data-set	Allowed lateness	Period	Win. size	Dropped %	Avg. win. delay (ms)
DEBS 2012	1000	200	1000	1.24	9,452,454
			100	1.24	3,083,109
	100	10	1000	1.50	713,846
			100	1.50	175,497
DEBS 2015	1000	200	1000	98.72	644,046,759
			100	98.62	648,821,195
	100	10	1000	99.93	546,682,126
			100	99.97	392,904,397

Metrics: We measure two metrics: the percentage of dropped elements and the average delay between a window end and the watermark after which the window function was triggered. We call it the window delay and report it in milliseconds.

Parameters: For the periodic watermark generator, we set the value for the generation period $s \in \{10, 200\}$ *ms*. For the allowed lateness, $m \in \{100, 1000\}$ *ms*. For the adaptive watermark generator, we set w the number of tuples to initialize m to 10000, we set the late arrival threshold l and the sensitivity change rate Δ_δ to one of $\{1.0, 0.1, 0.01\}$, see Table 1 for descriptions of the parameters. For the size of the time window in the query, we vary it between 100 and 1000 milliseconds.

Results: For the data collected after running the pipeline for each unique combination of configuration parameters, to compute results we: 1) drop out result tuples for which the watermark is `Long.MAX_VALUE`. Flink generates a watermark with this value to flush any windows that were waiting for a watermark to trigger its computation, 2) count the number of tuples (*count* above) and subtract that from the total number of tuples we have in the respective data set to obtain the tuple drop percentage, and 3) average the delay over the computed windows remaining after step 1. Table 2 shows the results for the periodic generator and Table 3 shows the results for the adaptive one.

For the *DEBS 2012* data, the periodic generator provides lower delay for the shorter watermark generation period. But, with

[1] https://github.com/DataSystemsGroupUT/Adaptive-Watermarks

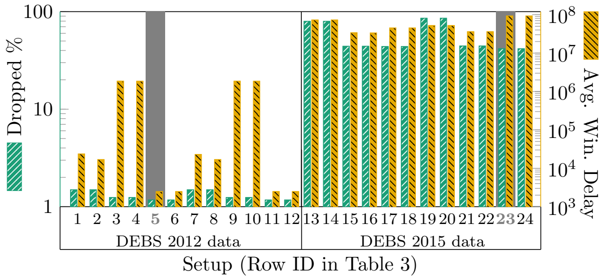


Figure 2: Evaluation of adaptive watermark generation

Table 3: Metrics for adaptive watermark generation

	Dataset	Δ_δ	l	Win. size	Dropped %	Avg. win. delay (ms)
1	DEBS	1	1	1000	1.49	23,633
2	2012	1	1	100	1.49	16,796
3		1	0.1	1000	1.24	1,853,667
4		1	0.1	100	1.24	1,847,622
5		1	0.01	1000	1.17	2,467
6		1	0.01	100	1.17	2,469
7		0.1	1	1000	1.49	22,905
8		0.1	1	100	1.49	16,796
9		0.1	0.1	1000	1.24	1,851,629
10		0.1	0.1	100	1.24	1,847,523
11		0.1	0.01	1000	1.17	2,472
12		0.1	0.01	100	1.17	2,471
13	DEBS	1	1	1000	79.60	72,863,504
14	2015	1	1	100	79.60	72,863,574
15		1	0.1	1000	44.14	33,826,576
16		1	0.1	100	44.14	33,827,281
17		1	0.01	1000	43.82	45,279,068
18		1	0.01	100	43.82	45,279,328
19		0.1	1	1000	85.52	51,965,257
20		0.1	1	100	85.52	51,965,876
21		0.1	0.1	1000	44.42	36,007,475
22		0.1	0.1	100	44.42	36,007,727
23		0.1	0.01	1000	41.64	92,672,393
24		0.1	0.01	100	41.64	92,673,050

higher drop rate. This is logical because of the trade off between tuple drop percentage and window delay. In the case of the adaptive watermark generator, with the default configuration (rows 1&2 in Table 3), it has almost the same tuple drop percentage as the periodic generator but with at least an order of magnitude improvement in window delay. Setting late arrival threshold $l = 0.1$, the tuple drop percentage is decreased but with two orders of magnitude higher window delay (rows 3&4). Yet, the delay is still below the delay of the periodic generator with the same tuple drop percentage. Setting $l = 0.01$ improves the tuple drop percentage by 0.07% and with very low window delay (row 5). This might look counterintuitive. But, by investigating the data for this configuration, we found that several windows were just fired by the termination of Flink's job. Thus, we had fewer windows that were processed due to the generated watermarks. This is logical as with a lower value of l, less number of watermarks is generated in general. Table 3 shows that changing the value of Δ_δ for the *DEBS 2012* data does not have much of an effect and the results are almost identical. The control is mainly driven by l.

For the *DEBS 2015* data set, we can see that the periodic generator drops almost all the tuples for the different configurations (Table 2). The window delay is very huge. Using the adaptive generator, with the default configuration (rows 13 and 14 in Table 3), we achieve a drop rate close to the percentage of the out-of-order tuples in the data set. However, the delay is an order of magnitude less than the periodic generator. Restricting the late arrival threshold to 0.1 reduces the tuple drop percentage and reduces the window delay by about 50% (rows 15 and 16). Pushing l to 0.01 reduces the tuple drop rate slightly but with an increase in window delay. The least tuple drop percentage is achieved when $\Delta_\delta = 0.1$ and $l = 0.01$ but with higher window delay (row 23). Figure 2 visualizes the tuple drop percentage and the average window delay for the different parameter values of the adaptive generator.

Our experimental results show the superiority of adaptive watermarks to baseline periodic ones. For both ordered and unordered streams, less tuple drop rate as well as several orders of magnitude saving in window delay.

4 CONCLUSION

In this paper, we have proposed an adaptive approach for generating low watermarks for event-time stream processing. It adapts to changes in data arrival frequency as well as to late arrival ratio. Compared to the baseline heuristic periodic watermark generation, and as indicated by application on data sets with different arrival frequency and delay characteristics, our approach strikes a balance between latency and accuracy of stream applications.

Currently, setting the late arrival threshold to very low percentage ceases the generation of the next watermark until the threshold is reached. This might be inconvenient and might cause high-latency. We intend to investigate an automated approach that automatically balances between latency and accuracy without having the user to specify threshold explicitly.

Acknowledgment: The work of Ahmed Awad and Sherif Sakr is funded by the European Regional Development Funds via the Mobilitas Plus programme (grant MOBTT75). The work of Jonas Traub is funded by the German Ministry for Education and Research as BBDC II (01IS18025A).

REFERENCES

[1] Tyler Akidau et al. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *PVLDB* 6, 11 (2013), 1033–1044.

[2] Tyler Akidau et al. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB* 8, 12 (2015), 1792–1803.

[3] Albert Bifet and Ricard Gavaldà. 2007. Learning from Time-Changing Data with Adaptive Windowing. In *SIAM.* 443–448.

[4] Nihal Dindar, Nesime Tatbul, Renée J Miller, Laura M Haas, and Irina Botan. 2013. Modeling the execution semantics of stream processing engines with SECRET. *VLDB Journal* 22, 4 (2013).

[5] Buğra Gedik. 2013. Generic windowing support for extensible stream processing systems. *Software: Practice and Experience* 44, 9 (2013), 1105–1128.

[6] Philipp Marian Grulich, René Saitenmacher, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2018. Scalable Detection of Concept Drifts on Data Streams with Parallel Adaptive Windowing. In *EDBT.*

[7] Martin Hirzel, Guillaume Baudart, Angela Bonifati, Emanuele Della Valle, Sherif Sakr, and Akrivi Vlachou. 2018. Stream Processing Languages in the Big Data Era. *SIGMOD Record* 47, 2 (2018), 29.

[8] Zbigniew Jerzak, Thomas Heinze, Matthias Fehr, Daniel Gröber, Raik Hartung, and Nenad Stojanovic. 2012. The DEBS 2012 Grand Challenge. In *DEBS.*

[9] Zbigniew Jerzak and Holger Ziekow. 2015. The DEBS 2015 Grand Challenge. In *DEBS.*

[10] Imen Khamassi et al. 2018. Discussion and Review on Evolving Data Streams and Concept Drift Adapting. *Evolving Systems* 9, 1 (2018), 1–23. https://doi.org/10.1007/s12530-016-9168-2

[11] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. on KDE* 15, 3 (2003), 555–568.

Identifying Bias in Name Matching Tasks

Alexandros Karakasidis
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
a.karakasidis@uom.edu.gr

Evaggelia Pitoura
Department of Computer Science & Engineering
University of Ioannina
Ioannina, Greece
pitoura@cs.uoi.gr

ABSTRACT

One of the applications of string similarity measures regards identifying the same entity within one or more corpora of data, also known as the problem of entity resolution. While a lot of methods have been introduced for assessing the similarity of two strings, there has not been yet a study to examine whether these methods are appropriate, in terms of effectiveness and fairness when applied to names of specific groups of individuals. In this paper, we provide extensive experimental results over a number of popular string measures which indicate that string comparison methods fall short when applied to specific groups, a fact leading to algorithmic bias against these groups. We also share some thoughts and guidelines on the way we envision, database practitioners should address such cases.

1 INTRODUCTION

Recently, the problem of algorithmic bias and fairness has drawn significant attention [8]. There is an increasing concern about the potential risks of data-driven approaches in decision making algorithms [2, 8, 15, 16, 18], raising a call for equal opportunities by design [9]. Biases can be introduced at different stages of the design, implementation, training and deployment of machine learning algorithms.

The problem has mainly been studied in the context of fairness in classification tasks, where individuals are classified in a positive or negative class. Example applications include, among others, hiring, school admission, crime risk factor estimation, and advertisement selection. There are two general approaches to defining fairness, namely *group* and *individual fairness* [6]. A common example of group fairness is *statistical parity*, where the proportion of members in a protected class that receive positive classification is asked to be identical to the proportion in the general population. Individual fairness requires that similar people are treated similarly.

In this paper, we draw attention to issues of bias in the task of entity resolution and, in particular, to string matching for name matching tasks. Approximate string matching forms the basis of a variety of algorithms in many applications where the ability to identify a word, despite of misspellings or typographical errors, is of crucial importance.

As such, there is a significant body of work on string comparison methods for name matching tasks [3, 7]. There has also been work on assessing the performance of these string matching algorithms [4, 17]. Amongst others, Lange and Naumann [11] consider different similarity measures for different frequencies of names in a database. Nevertheless, to the best of our knowledge, it is the first time that the problem of bias occurring in name matching is examined.

In this paper, we first provide a definition of bias in name matching tasks. Intuitively, we consider a matching algorithm to be negatively biased against a specific group of names, if the mismatches occurring for names in this group exceed those occurring on the average. Then, we examine a number of widely used string similarity measures in terms of bias.

We provide extensive experimental evidence that there is bias against certain race groups when applying string comparison algorithms for name matching tasks. This kind of biased behavior appears mainly against people with Asian origin, not intentionally, but as a consequence of the basic characteristic these names have, which is short length, compared to the names of people of other races. We then discuss approaches towards making methods for approximately matching names fair.

In brief, the contributions of this paper are outlined as follows:

- We identify the problem of bias in string comparison methods for name matching tasks.
- We propose a definition of bias for name matching tasks.
- We provide empirical evidence (in the form of extensive experimentation) of bias in a number of string similarity measures.
- We discuss how this problem may be addressed.

The rest of this paper is structured as follows. Section 2 introduces our measure for quantifying bias in name matching tasks, followed by a short description of the methods used in our assessment. Next, in Section 3, we present our experimental evaluation. Finally, we conclude in Section 4, summing up our findings, providing our vision on how bias cases for name matching tasks should be addressed by database practitioners and provide some thoughts for future work.

2 METHODOLOGY

In this section, we present our definition for bias in name matching tasks and a brief description of the string matching methods used in our evaluation.

2.1 Defining Bias

We need to quantitatively define the notion of bias in the context of record matching. In general, we want to capture the fact that for specific groups in the user population, there is unfair behavior. Let us use U to represent the general set of users. We assume that users are divided into m groups, G_i, $\cup_i G_i = U$, and $G_i \cap G_j = \emptyset$, for $i \neq j$, based on the value of some protected attribute, such as, for example, race, or gender.

Intuitively, we assume that there is bias against some specific user group G, if there are more errors in the results of matching when it comes to users in G than in the general population. There are two cases of errors, or *mismatches*. The first case is when two records are falsely reported as a match. This corresponds to a False Positive (FP). The second case is when two records that correspond to the same entity are not matched. This corresponds to a False Negative (FN). This is formalized in Definition 1.

Definition 1 (group mismatch). *The mismatch M for a user group G of size N is defined as $M(G) = \frac{\sum_{u \in G} FP(u)+FN(u)}{N}$, where $FP(u)$ and $FN(u)$ are the false positives and false negatives when matching names referring to user u in G.*

We determine whether a name matching method is biased towards a particular group by comparing its mismatch performance for this group with its mismatch performance for all groups of the general population U. To this end, we define

$$GM(U) = \frac{\sum_{G_i \in U} M(G_i)}{m} \tag{1}$$

Definition 2 (name matching bias). *The bias B of a string comparison method for a group population G is defined as:*

$$B(G) = \frac{GM(U) - M(G)}{GM(U)} \tag{2}$$

Let us now discuss the values of bias and their meanings. If G's mismatches are equal to the average number of mismatches of all groups, then its bias is equal to zero. On the other hand, if the mismatches of G are more than the average, then bias gets a negative value, and, in this case, we consider that the method examined is negatively biased against G. Finally, when the average number of mismatches is greater than that of G, then we assume that the method is biased in favor of G. Values close to 0 indicate low bias, while as the absolute value of bias increases, this is an indication of increased bias, either positive or negative.

In this paper, we consider string matching algorithms applied to names and race as the protected attribute.

2.2 Methods for Name Matching

Let us now briefly present some popular string comparison methods for name matching tasks that we use in our empirical evaluation. Let us consider two strings, which, in our case, represent names, S_1 and S_2.

Edit Distance. The edit distance between S_1 and S_2 is the minimum number of edit operations required to transform S_1 to S_2. We use the Levenshtein edit distance [12] where an edit operation is a character insertion, deletion or replacement, and each operation has cost equal to 1.

Jaro & Jaro-Winkler Similarities. The Jaro similarity [10] between two strings S_1 and S_2 is defined as:
$JS(S_1, S_2) = \begin{cases} 0, & \text{if m=0} \\ \frac{1}{3}\left(\frac{m}{|S_1|} + \frac{m}{|S_2|} + \frac{m-t}{m}\right), & \text{otherwise} \end{cases}$, where $|S_1|$, $|S_2|$ are respectively the lengths of S_1 and S_2, m is the number of matchings, and t is the number of transpositions defined as follows. Two characters are considered matching, if they are the same and within distance less than $\left\lfloor \frac{max(|S_1|, |S_2|)}{2} \right\rfloor - 1$. The number of transpositions is calculated as follows: The i-th common character in S_1 is compared with the i-th character in S_2. Each nonmatching character is a transposition.

Jaro - Winkler similarity [19] is based on Jaro similarity, favoring prefix matches, as they are considered of higher importance for name matching. It is defined as: $JWS(S_1, S_2) = JS(S_1, S_2) + l\,p(1 - JS(S_1, S_2))$, where l is the length of common prefixes of S_1, S_2, topping at four common characters and p a scaling parameter defaulting to 0.1.

Q-gram based methods. Q-grams are successive substrings of a string each of length Q. Each string is divided into a set of distinct Q-grams. Let N_1 and N_2 be the sets of Q-grams of strings S_1 and S_2 respectively. There are various set comparison measures [13]. In this paper, we use the Dice coefficient and Jaccard index, where the Dice coefficient is defined as $DC(S_1, S_2) = \frac{2|N_1 \cap N_2|}{|N_1|+|N_2|}$, and the Jaccard index as to $JI(S_1, S_2) = \frac{|N_1 \cap N_2|}{|N_1 \cup N_2|}$.

Table 1: Soundex mapping table.

a, e, h, i, o, u, w, y	→	0	*l*	→	4
b, f, p, v	→	1	*m, n*	→	5
c, g, j, k, q, s, x, z	→	2	*r*	→	6
d, t	→	3			

Soundex. Soundex, based on English language pronunciation, is the oldest (patented in 1918 [14]) and best known phonetic encoding algorithm. It keeps the first letter of a string and converts the rest into numbers according to Table 1. All zeros (vowels and 'h', 'w' and 'y') are then removed and sequences of the same number are merged to a single one. The final code is the original first letter and three numbers (longer codes are stripped off, and shorter codes are padded with zeros).

3 EXPERIMENTAL EVALUATION

In this section, we present the findings of our empirical assessment.

3.1 Experimental Setup

For our evaluation, we use data from the US Census Bureau[1], list of names. This list contains 162253 distinct names and the frequency of appearance of each of these names in several race groups in the agency's database. We selected three of the most popular of these race groups. These are the groups of names from people characterized as White, Afro-americans and those with Asian or Pacific origin. Since people of different races may have the same name, we keep only those names for which at least 90% of the people having these names belong to a single group. Then, we extracted the 50 most common names from each of these groups. The statistics of the dataset used in this study are depicted in Table 2.

Since our aim is to assess the effectiveness of string similarity methods for name matching tasks, we need alternative or erroneous versions of the names that we have extracted, for matching them with their original versions. For this purpose, we have employed the data corrupter from the German Record Linkage center [1]. This was configured to perform one random edit operation (insertion, deletion or replacement) in each of the names we have extracted, so as to produce 1 error for each of them, as it is rather unusual for a word to have more than one or two typographical errors [5].

Then, we perform record linkage between the original and the corrupted version of the datasets using each of the string comparison methods presented earlier. For each of these methods, we perform the same experiment 10 times. The matching thresholds used for each method are illustrated in Table 3.

For the Levenshtein distance, the minimum difference between two strings is equal to 1, meaning one modification. Jaro, Jaro-Winkler, Dice are Jaccard are similarity measures returning values between 0, for total dissimilarity, and 1 for absolute similarity. Finally, we desire codes produced by Soundex to totally coincide, since its representation contains approximation characteristics.

[1] Available at https://www.census.gov/topics/population/genealogy/data/2010_surnames.html

Table 2: Statistics of the dataset.

Race Group	Average occurrences per name in top-50	Average occurrences per name in group	Number of distinct names in group
Asian	68861.44	964.1	2897
Black	3027.06	272.52	2364
White	91959.86	681.72	100688

We built a testbed using Anaconda Python 3.5 and conducted our experiments on Ubuntu 18.04 LTS, powered by an Intel i5 processor with 8 GBs of RAM. The results of our assessment are illustrated in Figure 2. In all plots, the horizontal axis is used to display the various thresholds used in our evaluation, while the vertical axis represents bias. In each plot, the results for all races are illustrated using error bars, displaying the minimum, average and maximum measurement.

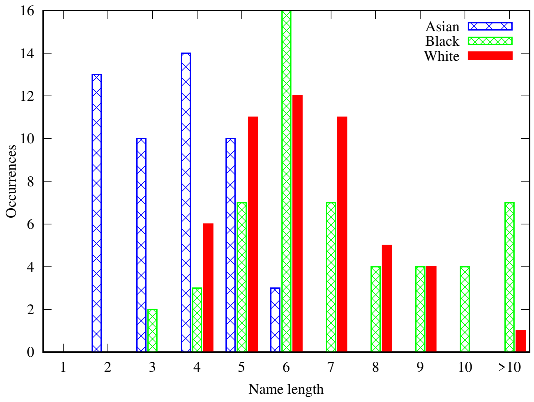


Figure 1: Length distributions for the top-50 names.

3.2 Discussion of the Results

Figure 2 indicates that all measures treat names from different groups differently, that is, all measures exhibit some form of bias. As a first step to interpret this bias, we seek to identify similar characteristics in each of the group of names. As such, we measure the lengths of the names in each group. These results are illustrated in Fig. 1. As we can see, the shortest names belong to people whose race is characterized as Asian.

We will now attempt to associate the length of these race groups with the behavior of the string comparison methods. As we can observe from Fig.2(a), the Levenshtein distance method performs worse for names with Asian origin than for names associated with people of White or African race. For the Asian group, the Levenshtein distance incurs negative bias lower than -1.67 when considering as matching names featuring an edit distance equal to 2, which is the lowest score in our evaluation. We speculate that, the reason for this behavior is the shorter length of the Asian names, as opposed to the other two groups, causing more ambiguous matches. The method seems to be slightly biased in favor of both the other two groups, with the White group receiving higher positive bias between the two.

As shown in Fig 2(b), Jaro also exhibits a continuous negative bias for the Asian group and a positive bias towards the African group. For the White names dataset, however, the situation is somewhat mixed. For matching threshold equal to 0.7, the bias is negative, then it becomes almost zero for threshold equal to 0.8 and it turns positive for a threshold equal to 0.9. This occurs because as we increase the threshold, the number of false positive matches drops, while the number of false negative matches remains merely unchanged. For Jaro-Winkler (Fig. 2(c)) the negative bias against Asian names is evident, again, as the positive bias in favor of African names. In this case, however, the method seems to also be negatively biased against White names as well.

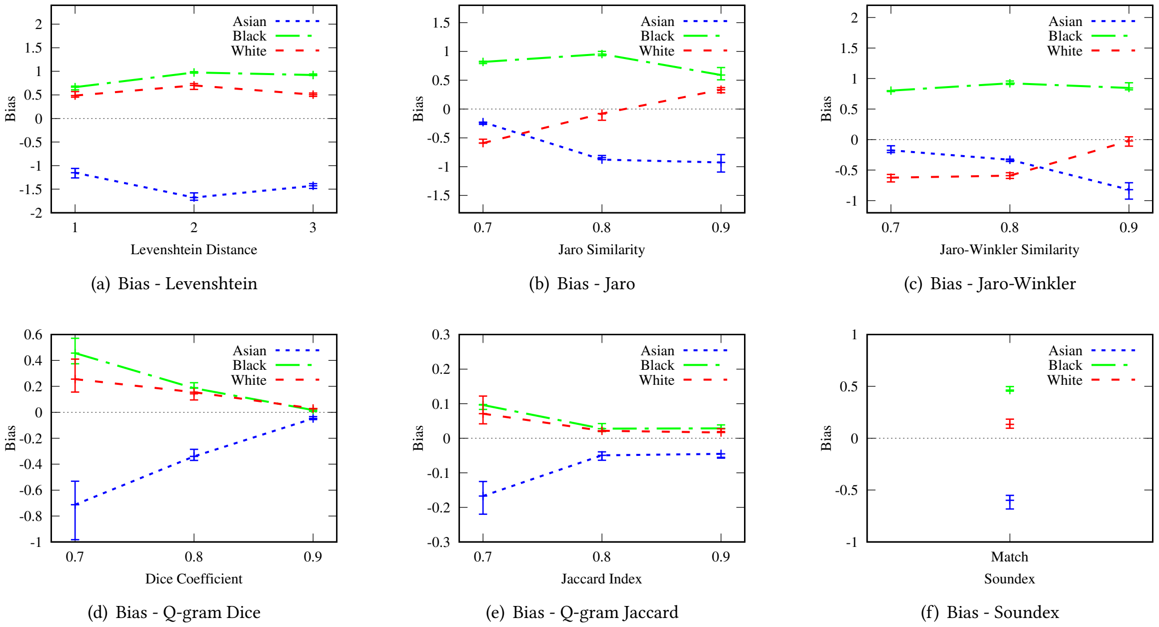
Let us now consider the case of breaking names into sets of Q-grams. For the Dice coefficient 2(d), we observe that the bias for all groups tends to converge to zero, when we use a high matching thresholds equal to 0.9. This is because, there are zero matches in most cases. As such, the number of false negatives is almost the same for each case and the number of false positives is the factor affecting the bias for each method. Examining each group individually now, it is evident that, again, there is negative bias towards the Asian group. The other two groups exhibit similar behavior, with the African one receiving more positive bias. The use of the Jaccard index (Fig. 2(e)) results in a similar behavior. Again, the Asian dataset is far below average, while the other two are very close and slightly positive. In both cases, however, the incurred bias is much lower than in the previous methods.

Table 3: Comparison methods and threshold values used.

Method	Threshold values
Levenshtein	1, 2, 3
Jaro, Jaro-Winkler, Dice, Jaccard	0.7, 0.8, 0.9
Soundex	1

Finally, we consider the use of Soundex, which has been specifically designed for name matching tasks. In this case, a threshold is not used for matching, and there is a match when two phonetic codes coincide. The results for this case are illustrated in Figure 2(f). As we may see, again, the results are similar with the previous cases. The Asian names exhibit negative bias, exceeding -0.5, while African names score very close to +0.5. White names are very close to the average case, slightly positively biased, nevertheless. A reason for this behavior is that, Soundex does not consider vowels. As such, since Asian names are usually short having similar vowels, it is very easy for the algorithm to be led to false positives.

Summarizing, regardless of the string similarity measure used, there is significant evidence that there is negative bias when matching names of people belonging to the Asian race, while there is positive bias when matching names of people belonging to the African race. Depending on the method used, White names may be either positively or negatively biased, falling, in most situations, within the average case. Besides this general behavior, some of the string measures are better than others in terms of the volume of bias, with the Q-gram Jaccard method outperforming the others by exhibiting the smaller bias.



(a) Bias - Levenshtein

(b) Bias - Jaro

(c) Bias - Jaro-Winkler

(d) Bias - Q-gram Dice

(e) Bias - Q-gram Jaccard

(f) Bias - Soundex

Figure 2: Bias of the string comparison methods.

4 CONCLUSIONS, THOUGHTS AND FUTURE WORK

Bias of this or some other form may be present in different steps of entity resolution and a lot of work is needed to formalize, detect and remedy it in the many different algorithms and techniques around this important task.

Our preliminary studies have revealed that, even in the basic task of name matching, there may be bias, in the sense that the quality of the results varies based on the group an individual belongs to. In this case, bias is not malicious, but rather a side effect of the data values (names) of the attributes of individuals belonging to specific groups. For name matching, we believe that we provide some useful insight to database practitioners for handling such situations. To this end, considering a single threshold for matching names from all origins, does not yield correct results for the Asian names, therefore this is a factor of concern. A first measure to alleviate this issue would be to consider the lengths of the names, or race information, if available, in cases of name matching tasks and maybe perform this task separately for this class of names.

For the future, there are many additional steps to be taken, as our study focuses on a very specific task, namely string matching based on names. We aim at performing further experiments considering data from more distinct groups which are going to be addressed as hidden parameters and examining deeper the structures of the names of each case, in order to reveal more factors that cause algorithmic bias in name matching tasks.

REFERENCES

[1] Tobias Bachteler and Jarg Reiher. 2012. *A Test Data Generator for Evaluating Record Linkage Methods.* Technical Report. German RLC Work. WP-GRLC-2012-01.

[2] Solon Barocas and Andrew D. Selbst. 2014. Big Data's Disparate Impact. *SSRN eLibrary* (2014).

[3] Peter Christen. 2012. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer Publishing Company, Incorporated.

[4] William Cohen, Pradeep Ravikumar, and Stephen Fienberg. 2003. A comparison of string metrics for matching names and records. In *Kdd workshop on data cleaning and object consolidation*, Vol. 3. 73–78.

[5] Fred J. Damerau. 1964. A Technique for Computer Detection and Correction of Spelling Errors. *CACM* 7, 3 (1964).

[6] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. 2012. Fairness through awareness. In *Proceedings of the 3rd innovations in theoretical computer science conference.* ACM, 214–226.

[7] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 19, 1 (Jan. 2007), 1–16.

[8] Sara Hajian, Francesco Bonchi, and Carlos Castillo. 2016. Algorithmic bias: From discrimination discovery to fairness-aware data mining. In *SIGKDD.* ACM, 2125–2126.

[9] White House. 2016. Big Data: A Report on Algorithmic Systems, Opportunity, and Civil Rights. *Washington, DC: Executive Office of the President, White House* (2016).

[10] Matthew A. Jaro. 1978. *UNIMATCH : a record linkage system : users manual.* Bureau of the Census Washington.

[11] Dustin Lange and Felix Naumann. 2011. Frequency-aware Similarity Measures: Why Arnold Schwarzenegger is Always a Duplicate. In *CIKM.* ACM.

[12] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

[13] Chen Li, Jiaheng Lu, and Yiming Lu. 2008. Efficient merging and filtering algorithms for approximate string searches. In *ICDE.* IEEE, 257–266.

[14] Margaret. Odell and Robert Russell. 1918. The Soundex coding system. *US Patents* 1261167 (1918).

[15] Evaggelia Pitoura, Panayiotis Tsaparas, Giorgos Flouris, Irini Fundulaki, Panagiotis Papadakos, Serge Abiteboul, and Gerhard Weikum. 2018. On Measuring Bias in Online Information. *SIGMOD Rec.* 46, 4 (Feb. 2018), 16–21.

[16] Andrea Romei and Salvatore Ruggieri. 2014. A multidisciplinary survey on discrimination analysis. *The Knowledge Engineering Review* 29, 05 (2014), 582–638.

[17] Chakkrit Snae. 2007. A comparison and analysis of name matching algorithms. *International Journal of Applied Science. Engineering and Technology* 4, 1 (2007), 252–257.

[18] Julia Stoyanovich, Serge Abiteboul, and Gerome Miklau. 2016. Data, Responsibly: Fairness, Neutrality and Transparency in Data Analysis. In *EDBT.*

[19] William E Winkler. 1990. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. (1990).

Rock – Let the points roam to their clusters themselves

Anna Beer
LMU Munich
Munich, Germany
beer@dbs.ifi.lmu.de

Daniyal Kazempour
LMU Munich
Munich, Germany
kazempour@dbs.ifi.lmu.de

Thomas Seidl
LMU Munich
Munich, Germany
seidl@dbs.ifi.lmu.de

ABSTRACT

In this work we present Rock, a method where the points **Ro**am to their **c**lusters using **k**-NN. Rock is a draft for an algorithm which is capable of detecting non-convex clusters of arbitrary dimension while delivering representatives for each cluster similar to, e.g., Mean Shift or k-Means. Applying Rock, points roam to the mean of their k-NN while k increments in every step. Like that, rather outlying points and noise move to their nearest cluster while the clusters themselves contract first to their skeletons and further to a representative point each. Our empirical results on synthetic and real data demonstrate that Rock is able to detect clusters on datasets where either mode seeking or density-based approaches do not succeed.

1 INTRODUCTION

Mode seeking clustering algorithms aim for detecting the modes of a probability density function (pdf) from a finite set of sample points. This type of clustering algorithm serves the purpose of detecting local maxima of density of a given distribution in the data set. In contrast density-based methods have the capability of detecting density-connected clusters of arbitrary shape. Rock positions itself between mode-seeking methods and density-based approaches. It is capable of detecting clusters of non-convex shapes *and* arbitrary densities at the same time. Plus it provides representatives for the detected clusters. Such an approach contributes additional expressiveness to the density-connected clusters. Further Rock provides the possibility to observe the directions of the data points roaming to the representatives. As an example, Figure 1 shows the path every point travels to its cluster representatives over time when Rock is applied. As Rock is not relying on any predefined density kernel it is able to detect non-convex clusters, which is a major advantage regarding other clustering methods.

The remainder of this paper is organized as follows: We provide a brief overview on the mode seeking method Mean Shift and other k-NN-based mode seeking algorithms. We then proceed explaining the Rock algorithm. Following to the elaboration on the algorithm we evaluate the performance of Rock against k-Means, Mean Shift and DBSCAN. Here the linking-role of Rock becomes clear, as our algorithm detects non-convex shaped areas where e.g., k-Means and Mean Shift fail and yet provides a representative of each detected cluster similar to the centroids in k-Means or modes in Mean Shift. We conclude this paper by providing an overview of the core features of our method and further elaborating on potential future work.

The main contributions of Rock are as follows:

- It finds non-convex shaped clusters of any density, which neither density based clustering methods nor mode or centroid based clustering methods are capable of
- The number of clusters does not have to be given by the user
- It provides a representative for every cluster

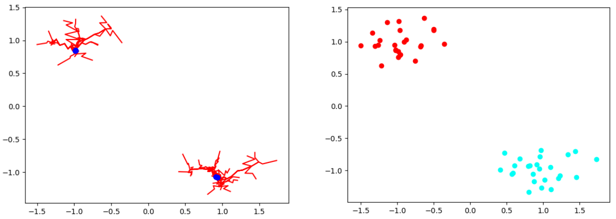


Figure 1: All points of a dataset with two clusters: overlapping final positions and the trajectories of all points on the left, initial positions and clustering on the right

2 RELATED WORK

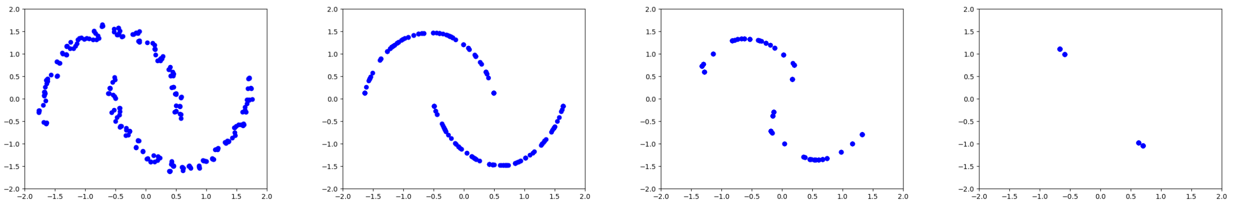
The most popular related work is Mean Shift, which has been initially proposed in [5]. Its core idea is to regard the points in an area around each point and iteratively shift the center of this area to the mean of the points lying in the previously regarded area. The Mean Shift method became popular in the field of clustering by the work of [2]. The core concept of Mean Shift is to determine the maxima (referred to as modes) of a given density function. In contrast to Rock, Mean Shift requires a kernel function K by which the weights of the nearby points are determined in order to re-estimate the mean. Further the used kernel requires a size of the area to be regarded around each point, which is also referred to as bandwidth. The choice of the bandwidth is not trivial, additionally there can be the need of an adaptive bandwidth size.

Another algorithm for detecting modes is developed in [3]. In contrast to the Mean Shift approach they rely on k-Nearest Neighbors (k-NN). While Mean Shift relies on a kernel density estimate (KDE) the k-NN approach relies on a k-NN density estimate. The algorithm in [3] defines for each input point x_i a pointer to the point within the k-Nearest Neighbors with the highest k-NN density. The process is repeated until a pointer points to itself, which represents then the local mode of x_i. However, in contrast to Rock, this method also relies on a density definition, where the density of a point is defined as inversely proportional to the distance of the k-th nearest neighbor. In a more recent work, [8] proposed a k-NN clustering method relying on consensus clustering, in which results of several clustering trials are combined in order to get a better partition.

Regarding methods which generate a backbone of a cluster, in a more recent work named ABACUS [1] the authors propose a method which is specialized in identifying skeletons of clusters using k-NN. However ABACUS and Rock differ in two major aspects. First, Rock does not primarily aim for the skeletons of a cluster but for a representative for each detected cluster (body to representative). In contrast ABACUS is targeted to find first a

Series ISSN: 2367-2005

10.5441/002/edbt.2019.73



(a) t=1 **(b) t=4** **(c) t=7** **(d) t=14**

Figure 2: Actual positions of the points at the given time t. First “skeletons” of the clusters emerge, which then tighten to divisible clusters.

backbone of a cluster and then expand from the backbone until a border of a cluster is reached (backbone to body). Secondly, ABACUS uses a static k for the k-NN approach. In our work in progress, we provide a method where k changes over each iteration step.

In this paper we compare Rock with Mean Shift and k-Means [7]. K-Means is a partitioning clustering algorithm where the user provides the number of clusters k which shall be detected. The cluster model of k-Means provides as a result the cluster centroids μ and the assignment of the datapoints to their centroids. Finally we evaluate the performance of Rock compared to DBSCAN [4], which is a density based clustering method. We compare to density based clustering since it is in contrast to other types of clustering algorithms capable of detecting non-convex shaped clusters. DBSCAN requires an ε-range and a minimum number of points ($minPts$) which shall be located within this ε-range. Through these two parameters the user can control how dense the detected clusters have to be.

3 THE ALGORITHM

Rock implements the best of both worlds, partitioning and density-based clustering. It is able to detect non-convex clusters and clusters of highly different densities. Since we use the k-Nearest Neighbors as main indication for cluster membership the algorithm works for arbitrary numbers of dimensions. Given a query point p and a number of neighbors k, the k-Nearest Neighbors at timepoint t (we will explain that later) are defined as a set $kNN_t(p,k) \subseteq DB$ with exactly k objects, such that for all of these objects $\{o_1, o_2, ..., o_k\}$ the distance to the query point p is less or equal to the distance of any other data point o' to p in our data base DB:

$$\forall o \in kNN_t(p,k), \forall o' \in DB \setminus kNN_t(p,k) : d_t(p,o) \leq d_t(p,o')$$

t refers to the timepoint at which the data base DB is looked at, since each point will have a different position at different timepoints in our algorithm. Similarly we use $d_t(p,o)$ to describe the distance between points p and o for the status of the data base at timepoint t. For detecting the k-Nearest Neighbors, our implementation of Rock uses a Balltree [9] as index structure according to the sklearn reference.

The idea of Rock is, that every point moves gradually to the representative point ζ of its cluster by “wandering” into the center of its k-Nearest Neighbors in every step. So Rock does not need any other information than where the k-NN of each point are located at time step t, which makes it easily parallelizable. Equation 1 gives us the position ζ of a point p at a time t using $k(t)$ as the number of regarded neighbors: we get the new position $\zeta(p,k,t)$ of p by calculating the mean position of all objects $o \in kNN_{t-1}(p,k(t))$, i.e. of all k-NN of this point p *at the previous time step.*

$$\zeta(p,k,t) = \sum_{o \in kNN_{t-1}(p,k(t))} \frac{\zeta(o,k,t-1)}{k(t)} \qquad (1)$$
$$\zeta(p,k,0) = initialPosition(p)$$

We increase k linearly over time and stop if the algorithm converged or a given maximal time $tmax$ is reached. Of course we do not only store the actual position at a time of each point p but also its initial position $initialPosition(p)$. With that we are in the end able to put all points which have almost the same actual position into one cluster. This is when the distance is smaller than ε, which is described in equation 2. Therefore we regard the 1-NN-distances of all points, which is the distance to the first nearest neighbor each, and use half of the average 1-NN distances of all points as ε:

$$\varepsilon = \frac{\sum_{p \in DB} \left(\sum_{o \in kNN(p,1)} dist(p,o) \right)}{2 \cdot |DB|} \qquad (2)$$

We choose this ε since it delivers an intuitively good measure for how close two points have to be to be "mergeable" in regards to their initial distribution.
In summary the algorithm looks like follows:

Algorithm 1 Rock(pointset, tmax)

```
changed= true
t=0
while changed and t < tmax do
    k=k(t)
    changed=false
    for p in pointset do
        oldPosition= p.getOldPosition()
        newPosition= meanOfKNN(pointset, p, k, t)
        p.setActualPosition(newPosition)
        if distance(oldPosition, newPosition)<$\varepsilon$ then
            changed=true
    t++
```

Note that $tmax$ is not mandatory a hyperparameter and can be set beforehand to the same value for various experiments as we will show in section 4. If we took a static k, most of the points would only shrink together with their exact k-Nearest Neighbors. Thus, we increase k over time, to receive stable clusters. The minimal value with which we will also start should be $k = 3$, since observing less nearest neighbors (including the point itself) would not lead to anything. The highest useful value for k is $\frac{n}{2}$ for a dataset of n points since if more neighbors would be regarded, it would lead to only one big cluster. We want to reach that highest value at the maximal number of iterations $tmax$, thus we obtain for linear increase:

$$k(t) = \frac{0.5n - 3}{tmax} \cdot t + 3$$

In order that k actually increases in every step the slope of $k(t)$ should be at least 1. Therefore $tmax$ should be chosen such that $tmax < 0.5n - 3$. Choosing $tmax$ significantly too low would mean to early regard a high number k of nearest neighbors, which potentially belong to other clusters, and therefore merging several clusters into one. Vice versa, a significantly too high $tmax$ could result in clusters splitting up.

Figure 2 shows the positions of all points at expressive points in time t while clustering the two moons dataset, which is further described in section 4.1. Regarding that, we already note an additional advantage: The skeleton which emerges already in early steps for each cluster could be used for anytime results or elimination of outliers, which we plan to examine further in future work.

3.1 Complexity

Having described our algorithm, we now elaborate on the runtime complexity of Rock and compare it on a theoretical basis to k-means and MeanShift. Focusing first on the used index structure, according to [9] the runtime complexity of a Ball tree is $O(d \log(N))$ where d denotes the dimensionality and N the number of data points. The runtime complexity of MeanShift is $O(iN^2)$ where i denotes the number of iterations. k-Means has a runtime complexity of $O(Nkid)$ where k refers to the number of clusters which shall be found. For Rock, using the Balltree structure, we get a runtime complexity of $O(kNd \log(N))$ where k denotes here the number of k-NN. Conclusively it can be stated that our method performs with regards to its time complexity between the performance of k-means and MeanShift. Since this is a work in progress, it may be viable to consider query-strategies which rely on different structures, such as e.g., Locality-Sensitive-Hashing (LSH) as used in [10].

4 EVALUATION

We tested Rock on several datasets and compared our results to the ones obtained by DBSCAN, k-Means and Mean Shift, since Rock positions itself between them: it finds non-convex shaped clusters as DBSCAN, but as well finds clusters of different densities with according representatives as k-Means and Mean Shift. As quality measures we use the Normalized Mutual Information (NMI) and the Adjusted Rand Index (ARI) in regards to the ground truth. All experiments were executed with $tmax = 15$, which is an empirically determined good value for the normalized data we use (i.e. removing the mean and scaling to unit variance). For all comparative methods the best parameters have been determined by iterating over a parameter range [1] and choosing those parameters which yield the best NMI resp. the best ARI for each of the methods. This shows again the simplicity of choosing the one hyperparameter Rock requires in comparison to the several parameters of our competitors, as Rock is not too sensitive to $tmax$, as we will show in future work due to the lack of space.

4.1 Two Moons

The dataset as shown in Figure 3 is a classical and well known dataset consisting of two moon-shaped clusters. For Mean Shift a bandwidth of 0.6 yielded the best NMI and a bandwidth of 1.2 yielded the best ARI. For DBSCAN $\varepsilon = 0.3$ and $minPts = 2$ yielded an NMI and ARI of 1 and for k-Means we chose $k = 2$. We tested the dataset with 250 datapoints and 5% noise, obtaining the optimal result with Rock as well as with DBSCAN. The non-density based methods k-Means and Mean Shift assigned a part of each moon to the false cluster as shown in Figure 3. As that, k-Means reaches an NMI of 0.36 and an ARI of 0.46. Mean Shift performed slightly better with an NMI of 0.53 (bandwidth 0.6) and an ARI of 0.50 (bandwidth 1.2). It may seem unintuitively first why Rock finds such shaped clusters even though regarding "only" the k-NN, but it makes sense considering that we slowly increase k. Figure 2 shows the development of the points' positions: First, the noise gets eliminated and the skeleton of the two clusters emerges. Then the ends of the clusters shrink up more and more into the middle of the clusters each. When the algorithm finishes at $tmax = 15$, there are only the two representatives of the clusters left as Figure 2d implies already.

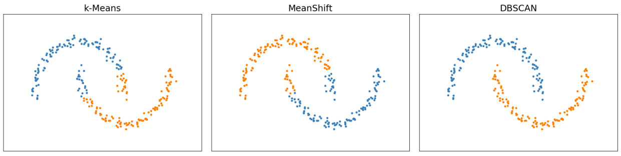


Figure 3: The clustering of two moons as received by k-Means, Mean Shift and DBSCAN (in this order)

4.2 Mouse

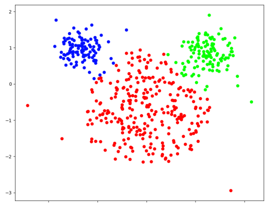


Figure 4: Result Rock returns for the mouse dataset

	NMI	ARI
k-Means	0.58	0.53
Mean Shift	0.67	0.66
DBSCAN	0.71	0.68
Rock	0.81	0.86

Table 1: Performances of the algorithms on the mouse dataset

To show how Rock deals with colliding clusters, we chose the mouse dataset. It consists of three round clusters of different densities which blend into each other at two points, where the "ears" of the mouse touch the "head". We conducted the experiment with Mean Shift using a bandwidth of 0.9, with DBSCAN using an ε-range of 0.2 and $minPts = 4$, and k-Means with $k = 3$. While Rock is able to distinguish the three clusters without a problem, the other algorithms cannot, as Figures 4.2 and 5 show: DBSCAN does not find the correct dividing line between head and right ear, k-Means puts a lot of the points from the head to the ears, and Mean Shift performs only slightly better than k-Means. The exact values of the resulting NMI and ARI can be seen in table 1.

4.3 Iris

Iris [6] may be the most famous dataset and one of the most challenging ones, since two of the three clusters are very difficult

[1] We tested ε with steps of 0.1 in a range of $[0.1, 2.0]$, $minPts$ with steps of 1 in a range of $[2, 10]$ and the bandwidth with steps of 0.1 in a range of $[0.1, 2.0]$

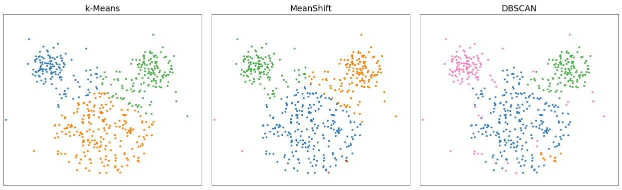


Figure 5: Clustering of the mouse dataset as obtained by the comparative methods k-Means, Mean Shift and DBSCAN (in this order).

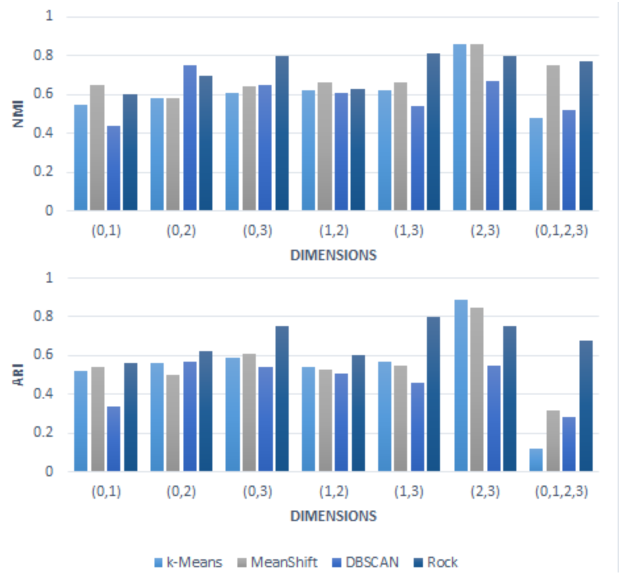


Figure 6: The NMIs (top) and ARIs (bottom) reached by the comparative methods and Rock, using different combinations of dimensions of the Iris dataset respectively.

to separate for they are overlapping in virtually every dimension. Nevertheless, even here Rock delivers good results: depending on which two of the four given dimensions it regards, we receive an NMI between 0.6 and 0.8 and an ARI between 0.56 and 0.8. These results are very good in comparison to the other methods, as Figure 6 shows: regarding the NMI we achieve at least the second best value, but most of the times the very best, no matter which two dimensions of the dataset are regarded. Concerning the ARI we mostly surpass the other algorithms, partially by far, as for example by using dimensions 1 and 3 (which is also reflected in the NMI). Since we claimed in Section 3 that Rock works with higher number of dimensions, we conducted an experiment on the full four-dimensional Iris dataset. Here our method yields the best results compared to its competitors, especially with regards to the computed ARI as can be seen in the rightmost case in Figure 6. The exact parameters for the comparative methods are shown in table 2 in the appendix.

5 CONCLUSION

In conclusion we developed an algorithm which is able to autonomously find clusters of various kind. Due to the successive increase of neighbors being decisive to a point, we are able to find non-convex shaped clusters. Without this successive increase Rock would be equivalent to a simple k-Means, but with the increase we only obtain the advantages of k-Means: being simple and providing kind of a cluster centroid which gives us even more information about the found clusters than DBSCAN provides. With the right use of index structures Rock is not only effective but also efficient. We showed that Rock surpasses good methods like Mean Shift, DBSCAN and k-Means in regards to non-convex shaped clusters, colliding clusters of different densities and even overlapping clusters like in the Iris dataset. Since it regards k-NN and no other distance measures it works on high dimensional datasets as well as on two dimensional ones. The rate of increase of k will be treated in future work, as a logarithmic or hyperbolic increase could lead to interesting results. Moreover we are also curious to know the impact of using reverse k-Nearest Neighbors instead of the k-NN, as well as using the median or mode of the k-NN instead of the mean. The influence of the parameter $tmax$ was also not yet analyzed thoroughly. And, last but not least, the feature of Rock to create skeletons of the clusters before reducing them further to representatives could be used to detect advanced structures, eliminate noise, or deliver anytime results.

A APPENDIX

Dimensions	Mean Shift	DBSCAN		k-Means
	bandwidth	minPts	ε	k
(0, 1)	0.9	2, 8	0.4, 0.5	3
(0, 2)	0.8, 0.6	2	0.5	3
(0, 3)	0.7, 0.8	2	0.5	3
(1, 2)	1.0	2	0.59	3
(1, 3)	0.8	2	0.7	3
(2, 3)	0.5	2, 7	0.3, 0.2	3
(0, 1, 2, 3)	0.7	3	0.4	3

Table 2: Parameters used for the respective comparative methods on Iris. Multiple values indicate that the first value is optimizing the NMI and differed from the second value, which optimizes the ARI.

ACKNOWLEDGMENTS

This work has been funded by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IS18036A. The authors of this work take full responsibilities for its content.

REFERENCES

[1] Vineet Chaoji, Geng Li, Hilmi Yildirim, and Mohammed J. Zaki. [n. d.]. *ABACUS: Mining Arbitrary Shaped Clusters from Large Datasets based on Backbone Identification.* 295–306.

[2] Yizong Cheng. 1995. Mean shift, mode seeking, and clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17, 8 (Aug 1995), 790–799.

[3] Robert P. W. Duin, Ana L. N. Fred, Marco Loog, and Elżbieta Pękalska. 2012. *Mode Seeking Clustering by KNN and Mean Shift Evaluated.* Springer Berlin Heidelberg, Berlin, Heidelberg, 51–59. https://doi.org/10.1007/978-3-642-34166-3_6

[4] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Kdd*, Vol. 96. 226–231.

[5] K. Fukunaga and L. Hostetler. 1975. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on Information Theory* 21, 1 (January 1975), 32–40.

[6] M. Lichman. 2013. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[7] S. Lloyd. 1982. Least Squares Quantization in PCM. *IEEE Trans. Inf. Theor.* 28, 2 (1982), 129–137.

[8] Jonas Nordhaug Myhre, Karl Øyvind Mikalsen, Sigurd Løkse, and Robert Jenssen. 2015. *Consensus Clustering Using kNN Mode Seeking.* Springer International Publishing, Cham, 175–186. https://doi.org/10.1007/978-3-319-19665-7_15

[9] Stephen M. Omohundro. 1989. *Five Balltree Construction Algorithms.* Technical Report.

[10] Ilan Shimshoni, Bogdan Georgescu, and Peter Meer. 2006. Adaptive Mean Shift Based Clustering in High Dimensions. *Nearest-neighbor methods in learning and vision: theory and practice* (2006), 203–220.

Short Paper

Fast Trajectory Range Query with Discrete Fréchet Distance

Jiahao Zhang[†‡] Bo Tang[†] Man Lung Yiu [‡]

[†] Department of Computer Science and Engineering, Southern University of Science and Technology

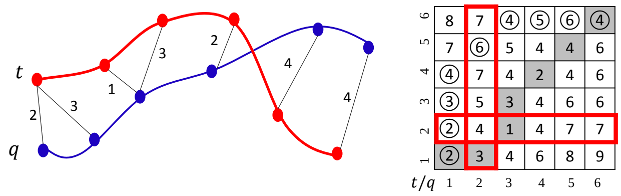
tangb3@sustc.edu.cn

[‡]Department of Computing, The Hong Kong Polytechnic University

{csjhzhang,csmlyiu}@comp.polyu.edu.hk

ABSTRACT

The discrete Fréchet distance (DFD) is widely used to measure the similarity between two trajectories. Trajectory range query has been extensively studied in trajectory analytical applications, e.g., outlier detection, movement pattern analysis. With the discrete Fréchet distance, the above applications are computation bound rather than disk I/O bound. In this work, we propose new lower and upper bound functions to speedup the evaluation of trajectory range queries. Experimental studies on three real datasets demonstrate the superiority of our proposal.



(a) DFD illustration

6	8	7	④	⑤	⑥	④
5	7	⑥	5	4	4	6
4	④	7	4	2	4	6
3	③	5	3	4	6	6
2	②	4	1	4	7	7
1	②	3	4	6	8	9
t/q	1	2	3	4	5	6

(b) DFD computation

Figure 1: DFD illustration and computation

1 INTRODUCTION

The range query problem on trajectory dataset is a core subroutine in trajectory analytical applications, e.g., anomaly monitoring [3], traffic analysis [4]. In particular, given a query trajectory q and trajectory database T, the trajectory range query problem returns a subset $S \subseteq T$ such that for each $t \in S$, the distance between t and q is within given threshold θ, i.e., $dist(t, q) \leq \theta$. Specifically, we use the discrete Fréchet distance (DFD) d_F, a widely used trajectory similarity measure [1, 5]. Given two trajectories $t = \langle t[1], t[2], \cdots, t[n] \rangle$ and $q = \langle q[1], q[2], \cdots, q[m] \rangle$, a coupling L between t and q is a sequence $\langle (t[a_1], q[b_1]), (t[a_2], q[b_2]), \cdots, (t[a_l], q[b_l]) \rangle$, where $a_1 = 1, b_1 = 1, a_l = n, b_l = m$, and for all $i = 1, \cdots, l$, we have $a_{i+1} = a_i$ or $a_{i+1} = a_i + 1$, and $b_{i+1} = b_i$ or $b_{i+1} = b_i + 1$. The discrete Fréchet distance between t and q is defined as

$$d_F(t, q) = \min_{L \in \Omega} \max_{(t[a_i], q[b_i]) \in L} ||t[a_i] - q[b_i]||,$$

where Ω is the set of all possible couplings between t and q.

Figure 1(a) illustrates the discrete Fréchet distance (DFD) between two trajectories t and q. Computing discrete Fréchet distance between t and q is equivalent to find a path from $(t[1], q[1])$ to $(t[n], q[m])$ in a free space diagram such that (i) the path travels along non-decreasing positions, and (ii) the maximum $||t[a_i] - q[b_i]||$ along the path is minimized [5]. For instance, the value of $d_F(t, q)$ in Figure 1 is determined by the gray path in Figure 1(b). Since it can be computed via dynamic programming, the time complexity of DFD computation is $O(mn)$.

In the literature, several lower and upper bounds have been proposed for DFD [2, 5]. We will introduce these bounds in Section 2. Nevertheless, the trajectory range query problem is still a computationally intensive problem, taking $O(|T|mn)$ time, where $|T|$ is the cardinality of trajectory dataset. To improve the query performance, we devise new lower and upper bound functions in this paper. Our experimental studies on real datasets reveal that our proposal improves the range query performance on DFD by up to an order of magnitude when compared to a baseline approach based on existing techniques.

The remainder of this paper is organized as follows. Section 2 formulates trajectory range query problem and presents the baseline solution adapted from existing works. We present our novel techniques in Section 3. Section 4 demonstrates the efficiency of our proposal with experiments on real datasets. Finally, we conclude the paper in Section 5.

2 PRELIMINARIES

In this section, we first define the trajectory range query problem in Section 2.1, then we present the baseline solution for it by adapting existing techniques in Section 2.2.

2.1 Problem Definition

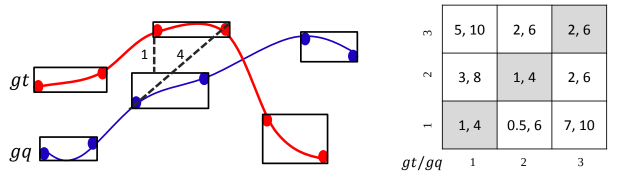
Definition 2.1 (Trajectory Range Query Problem). Given a query trajectory q, and a trajectory dataset T and distance threshold θ, the trajectory range query returns a subset $S \subseteq T$ such that for each trajectory $t \in S$, the discrete Fréchet distance (DFD) between t and q is at most θ, i.e., $d_F(t, q) \leq \theta$.

A straightforward approach for this problem (cf. Definition 2.1) is to compute the exact discrete Fréchet distance (DFD) between each trajectory $t \in D$ and the query trajectory q. Its time complexity is $O(|T|mn)$, rendering it impractical for a large trajectory dataset.

2.2 Baseline Solution

In this section, we briefly introduce a baseline approach for the trajectory range query problem (cf. Definition 2.1) which employs existing techniques in the literature. It follows the filter-and-refinement paradigm. Let $LB(t, q)$ and $UB(t, q)$ denote the lower and upper bounds of $d_F(t, q)$ respectively, i.e., $LB(t, q) \leq d_F(t, q) \leq UB(t, q)$. A candidate t cannot be a result if $LB(t, q) > \theta$. A candidate t is definitely a result if $UB(t, q) \leq \theta$. In these two cases, we save an expensive exact DFD computation.

10.5441/002/edbt.2019.74



3	5, 10	2, 6	2, 6
2	3, 8	1, 4	2, 6
1	1, 4	0.5, 6	7, 10
gt/gq	1	2	3

(a) Group-based trajectory (b) Group-based bounds

Figure 2: Group-based DFD illustration and computation

We introduce existing lower and upper bounds as follows.

Cell-based lower bound LB_{cell} (from [5]): The cell-based lower bound is defined as

$$LB_{cell}(t, q) = \max(||t[1] - q[1]||, ||t[n] - q[m]||).$$

The idea of LB_{cell} is that any coupling L between q and t must start from $(1, 1)$ and end at (n, m). Its computation cost is $O(1)$.

Row-based lower bounds LB_{row} and LB_{cross} (from [5]): The row-based and column-based lower bounds are defined as follows:

$$LB_{row}(t, q) = \min_{b_i \in [1,m]} (||t[2] - q[b_i]||).$$

$$LB_{col}(t, q) = \min_{a_i \in [1,n]} (||t[a_i] - q[2||).$$

The path leading to DFD pass through the second column and row certainly contribute these two bounds. Both of them have $O(n)$ time complexity.

For example, the row-based and column-based lower bounds for trajectory t and q are $LB_{row}(t, q) = \min\{2, 4, 1, 4, 7, 7\} = 1$ and $LB_{col}(t, q) = \min\{7, 6, 7, 5, 4, 3\} = 3$, respectively.

By combining row-based and column-based lower bounds, we obtain a cross-based lower bound as follows.

$$LB_{cross}(t, q) = \max(LB_{row}(t, q), LB_{col}(t, q)).$$

Group-based bounds LB_g and UB_g (from [5]): The idea is to partition a trajectory t (in Figure 1(a)) to a grouped trajectory $gt = \{gt[1], gt[2], gt[3]\}$ (in Figure 2(a)), which can be represented by a sequence of minimum bounding rectangles (MBRs). The length of group-based trajectory is denoted as τ. The minimum and maximum distance between $gt[a_i]$ and $gq[b_i]$ (two MBRs) are denoted as $d_G^{lb}(gt[a_i], gq[b_i])$ and $d_G^{ub}(gt[a_i], gq[b_i])$.

They can be derived in $O(1)$ cost. For example, in Figure 2 $d_G^{lb}(gt[2], gq[2])$ and $d_G^{ub}(gt[2], gq[2])$ are 1 and 4, respectively.

The DFD computing procedure with group-based trajectory adopts the minimum or maximum distances between MBRs leads a lower or upper bound of $d_F(t, q)$, as follows:

$$LB_g(t, q) = \min_{L \in g\Omega} \max_{(gt[a_i], gq[b_i]) \in L} d_G^{lb}(gt[a_i], gq[b_i]),$$

$$UB_g(t, q) = \min_{L \in g\Omega} \max_{(gt[a_i], gq[b_i]) \in L} d_G^{ub}(gt[a_i], gq[b_i])$$

As shown in Figure 2(b), the lower and upper bound DFD distance between group-based trajectory gt and gq are $LB_g(t, q) = 2$ and $UB_g(t, q) = 6$, respectively. It reduces computation cost as $O(\tau^2) < O(mn)$.

Greedy-based upper bound UB_{greedy} (from [2]): In addition to the above bounds, [2] proposed a greedy algorithm to compute an upper bound for $d_F(t, q)$ with $O(n)$ cost.

The idea of greedy algorithm is simple, e.g., the discrete Fréchet distance (DFD) goes as follows: in every step, make the move that minimizes the current distance, where a "move" is a step in either one sequence or in both of them. The greedy algorithm guarantees $2^{\Theta(n)}$-approximation of the discrete Fréchet distance with linear time cost $O(n)$. As shown in Figure 1(b), the greedy algorithm produces the path shown as the circled numbers.

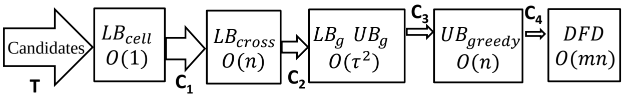
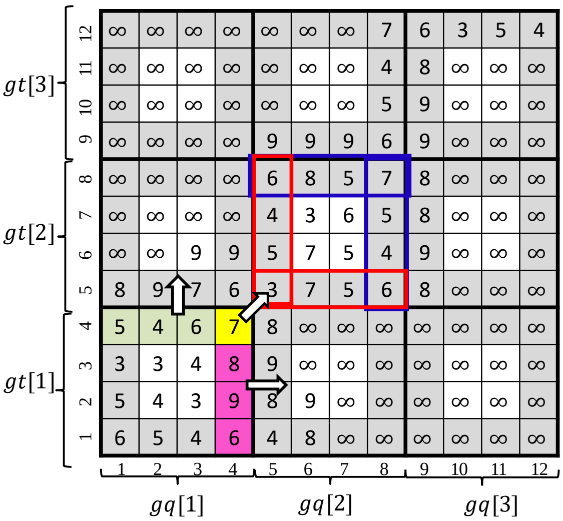


Figure 3: Baseline approach for trajectory range query



	1	2	3	4	5	6	7	8	9	10	11	12
12	∞	∞	∞	∞	∞	∞	∞	7	6	3	5	4
11	∞	∞	∞	∞	∞	∞	∞	4	8	∞	∞	∞
10	∞	∞	∞	∞	∞	∞	∞	5	9	∞	∞	∞
9	∞	∞	∞	∞	9	9	9	6	9	∞	∞	∞
8	∞	∞	∞	∞	6	8	5	7	8	∞	∞	∞
7	∞	∞	∞	∞	4	3	6	5	8	∞	∞	∞
6	∞	∞	9	9	5	7	5	4	9	∞	∞	∞
5	8	9	7	6	3	7	5	6	8	∞	∞	∞
4	5	4	6	7	8	∞	∞	∞	∞	∞	∞	∞
3	3	3	4	8	9	∞	∞	∞	∞	∞	∞	∞
2	5	4	3	9	8	9	∞	∞	∞	∞	∞	∞
1	6	5	4	6	4	8	∞	∞	∞	∞	∞	∞

Figure 4: Running example

Baseline approach: We construct the baseline approach by exploiting existing techniques. It applies a filter-and-refinement framework, as shown in Figure 3. It computes exact DFD distance for a candidate t only if it cannot be dismissed by lower bounds (i.e., LB_{cell}, LB_{cross}, LB_g) or upper bounds. Specifically, the computation cost of employed bounds are in an order from quick-but-dirty one to slow-but-accurate one.

3 PROPOSED OPTIMIZATIONS

To improve the performance of trajectory range query (cf. Definition 2.1), we propose several novel optimizations in this section.

3.1 Group-based border bounds

The lower bound distance between two grouped trajectories gt and gq is $LB_g(t, q) = \min_{L \in g\Omega} \max_{(gt[a_i], gq[b_i]) \in L} d_G^{lb}(gt[a_i], gq[b_i])$, where $d_G^{lb}(gt[a_i], gq[b_i])$ is the lower bound of ground distance between $gt[a_i]$ and $gq[b_i]$, shown as the dashed line with the label 1 in Figure 2(a). The effectiveness of the group-based lower bound pruning depends on term $LB_g(t, q)$.

Group-based border lower bound. Instead of using $d_G^{lb}(gt[a_i], gq[b_i])$ in baseline approach, we devise a tighter lower bound for ground distance between two groups $(gt[a_i], gq[b_i])$ by the crucial observation as follows. Consider $(gt[2], gq[2])$ in Figure 4, the minimum distance of $(gt[2], gq[2])$ is determined by the gray cells, as if the DFD path of trajectory t and q passes though the group pair $(gt[2], gq[2])$, $d_F(t, q)$ must be larger than

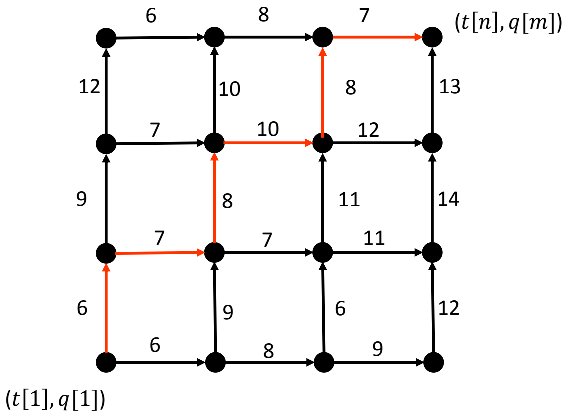


Figure 5: Group-based border upper bound

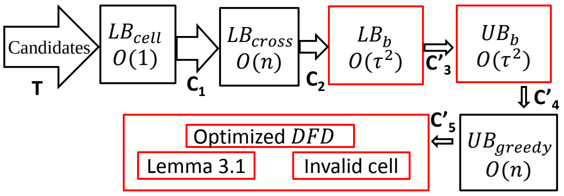


Figure 6: Advanced approach for trajectory range query

or equal to the minimum value in entrance cells (red rectangles) and exit cells (blue rectangles).

In particular, the minimum value of entrance cells and exit cells in given pair $(gt[a_i], gq[b_i])$ is defined as $d^{en}_{min}(gt[a_i], gq[b_i])$ and $d^{ex}_{min}(gt[a_i], gq[b_i])$, respectively. Thus, the entrance and exit cell based lower bound of ground distance of group pair $(gt[a_i], gt[b_i])$ is

$$d^{lb}_{ee}(gt[a_i], gq[b_i]) = \max\{d^{en}_{min}(gt[a_i], gq[b_i]), d^{ex}_{min}(gt[a_i], gq[b_i])\}.$$

Consequently, we have a tighter lower bound $LB_b(t, q)$ by

$$LB_b(t, q) = \min_{L \in g\Omega} \max_{(gt[a_i], gq[b_i]) \in L} d^{lb}_{ee}(gt[a_i], gq[b_i]).$$

There is a trade-off between the tightness of $d^{en}_{min}(gt[a_i], gt[b_i])$ (the same as $d^{ex}_{min}(gt[a_i], gt[b_i])$) and its computation cost. The quick-and-loose one is taking the smaller distance of $t[\frac{a_i n}{\tau}]$ with the MBR of $gq[b_i]$ and the MBR of $gt[a_i]$ with $q[\frac{b_i m}{\tau}]$. The slow-and-tight one is computing the smallest distance of $t[\frac{a_i n}{\tau}]$ with all points (i.e., 3, 7, 5, 6 in red rectangle) in $gq[b_i]$ and all points in $gt[a_i]$ with $q[\frac{b_i m}{\tau}]$ (i.e., 3, 5, 4, 6 in red rectangle). In terms of time complexity, the former one is $O(\tau^2)$ and the later one is $O(\tau n)$ for every group pairs in gt and gq.

Group-based border upper bound. As shown in Figure 5, the upper bound of ground distance between $t[\frac{a_i n}{\tau}]$ with the MBR of $gq[b_i]$ and the MBR of $gt[a_i]$ with $q[\frac{b_i m}{\tau}]$ for each $(gt[a_i], gq[b_i])$ pair are computed with $O(\tau^2)$ cost. The upper bound of the DFD path from $(t[1], q[1])$ to $(t[n], q[m])$ is equivalent to find a path from left-bottom corner to right-top corner, where the maximum value in that path is minimized, as the red path shown.

We denote that DFD upper bound as $UB_b(t, q)$, it is 10 in the example of Figure 5.

3.2 DFD path transfer directions

Take $(gt[1], gq[1])$ in Figure 4 as an example, the minimum distance between $gt[1]$ and $gq[1]$ is $\max\{\min\{3, 4\}, \min\{4, 6\}\} = 4$. Computing $LB_g(t, q)$ is equivalent to find a DFD path from $(gt[1], gq[1])$ to $(gt[\tau], gq[\tau])$ among the minimum distances of each group pairs, i.e., $d^{lb}_G(gt[a_i], gq[b_i])$. Even the minimum distance of each group pair is improved by entrance and exit cell bounds as above, it still can be optimized by exploiting the path transfer direction. For example, as shown in Figure 4, if the path is from $(gt[1], gq[1])$ to $(gt[1], gq[2])$, the lower bound contributes from $(gt[1], gq[1])$ is the minimum value among the magenta cells and yellow cells. Similarly, The path from $(gt[1], gq[1])$ to $(gt[2], gq[2])$ must pass the yellow cell. Obviously, the minimum value from $(gt[1], gq[1])$ to $(gt[1], gq[2])$, and from $(gt[1], gq[1])$ to $(gt[2], gq[2])$ are 6 and 7, which is larger than 4, i.e., the distance between $gt[1]$ and $gq[1]$ by entrance and exit cells bound. In summary, the transfer-based minimum values are used to tight the group-based DFD lower bound (i.e., $LB_b(t, q)$) during the DFD computation on grouped trajectories gq and gt.

3.3 DFD computation acceleration

As illustrated in Figure 3, for these candidate trajectories cannot be pruned by its lower bound or be detected as true results, it will incur expensive DFD computation $O(mn)$. Specifically, it computes the distance between any two points among t and q, i.e., each cell in Figure 4, then computes the $d_F(t, q)$ via dynamic programming. In the subsequential section, we devise two novel techniques, namely early termination and invalid cell ignoring, to reduce the DFD computation cost.

Early termination. We define $layer_{i,j}$ among trajectory t and q (cf. Figure 4) as either $t[i]$ or $q[j]$ are considered in that cells. Formally, the lower bound of $layer_{i,j}$ is defined as:

$$LB^{i,j}_{layer}(t, q) = \min\{\min_{k \in [1,j]}\{||t[i]-q[k]||\}, \min_{k \in [1,i]}\{||t[k]-q[j]||\}\}.$$

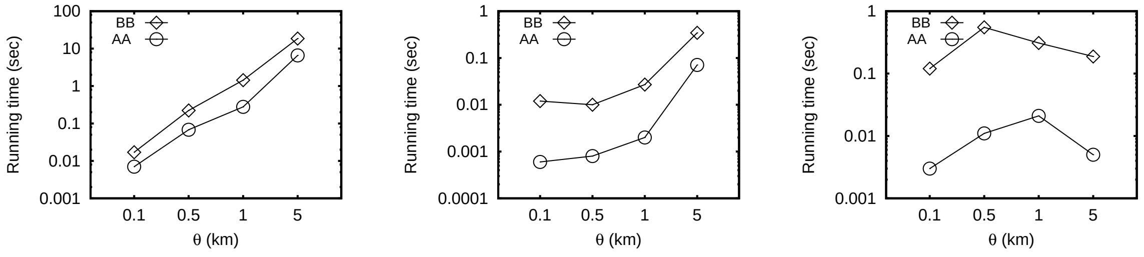
Lemma 3.1. $LB^{i,j}_{layer}(t, q) \leq d_F(t, q)$

Proof. The proof is trivial as the path from $(t[1], q[1])$ to $(t[n], q[m])$ must pass through the at least one cell of $layer_{i,j}$, thus, $d_F(t, q)$ is not smaller than the minimum value in $layer_{i,j}$. □

Thus, we terminate DFD computation if $LB^{i,j}_{layer}(t, q) \geq \theta$ with Lemma 3.1. Suppose the distance threshold $\theta = 2$, consider the $layer_{3,3}$ in Figure 4, $LB^{3,3}_{layer}(t, q) = \min\{3, 3, 4, 3, 4\} = 3$. We terminate the exact DFD computation earlier as $LB^{3,3}_{layer}(t, q) > \theta$. It improves computation cost by ignoring a lot of ground distance pair computation and the exact DFD computation.

Invalid cell ignoring. Consider the example in Figure 4, suppose the distance threshold $\theta = 7$. Given cell $(t[i], q[j])$ its distance computation can be avoided if $||t[i], q[j-1]||$, $||t[i-1], q[j]||$, and $||t[i-1], q[j-1]||$ are larger than θ. We define such kind of cells as invalid cells, set their distances as ∞, as shown in Figure 4 without incurring expensive distance computation.

Put all it together: We present the framework of our advanced approach (AA) for trajectory range query (with discrete Fréchet distance) in Figure 6. Our proposed techniques are enclosed by red rectangles, e.g., $LB_b(t, q)$, $UB_b(t, q)$, and optimized exact DFD computation.



(a) **GeoLife**

(b) **OSM-DE**

(c) **OSM-FULL**

Figure 7: Response time vs. distance threshold θ

4 EXPERIMENTAL EVALUATION

We evaluate the performance of the baseline approach (BB, cf. Section 2) and our advanced approach (AA, cf. Section 3) for trajectory range queries. In our experiments, we report the average measurements over 10 different query trajectories.

Dataset: We used three real world trajectory datasets with diverse geographical coverage and scales of sizes.

GeoLife.[1] The GeoLife project in Microsoft Research Asia collected this dataset from 182 users over five years (April 2007-August 2012). It has 18,655 trajectories and 24.9 million GPS points.

OSM-FULL.[2] This dataset contains 7.5 years of OpenStreetMap trajectory data around the world. It includes total 2.4 million trajectories constructed by 2.7 billion GPS points.

OSM-DE. It is a subset of **OSM-FULL**. We extracted trajectories that are located in Germany. **OSM-DE** has the highest data density among all regions in **OSM-FULL** [6]. It has 0.5 million trajectories and 0.5 billion GPS points in total.

We used C++ for the implementation and conducted all experiments (with single thread) on a machine with AMD A10-7850K 3.70GHz processor and 16GB main memory.

Overall performance evaluation: We compare the performance of BB and AA for trajectory range query problem on all three real world datasets by varying distance threshold in Figure 7. AA is faster than BB by 2.42 to 50.1 times. In addition, the performance gap between AA and BB becomes large with the rise of distance threshold θ. It also confirms AA is scalable. The trajectories in **GeoLife** are much denser than other two datasets, and that is the reason why range queries perform slower though **OSM-FULL** and **OSM-DE** have large sizes.

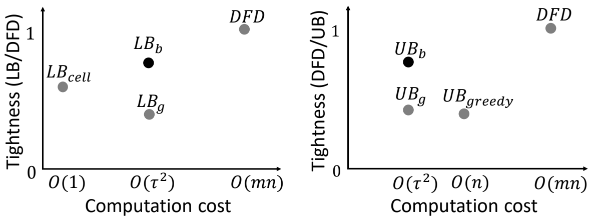
Effect of optimizations: We then evaluate the effectiveness of our proposed optimization techniques, e.g., group border-based bound, path transfer direction-aware bound, etc. Due to space limitation, we omit the experiment results on **OSM-DE** and **OSM-FULL** as they are similar to **GeoLife**. The group-size τ is 8 in all the experiments.

Table 1 illustrates the number of DFD executions (per query) of BB and AA with regard to distance threshold θ, respectively. Obviously, our advanced approach outperforms baseline approach.

We then investigate the effectiveness of DFD lower and upper bounds. Our proposed LB_b is much better than LB_g while both with $O(\tau^2)$ cost. In addition LB_b with $O(\tau n)$ cost is slightly better than with $O(mn)$, as shown in Figure 8(a). UB_b is better than UB_{greedy} and UB_g as illustrated in Figure 8(b).

Table 1: DFD **executions on GeoLife dataset**

θ	0.1 km	0.5 km	1km	5 km
BB	6.60	48.97	125.33	550.01
AA	**1.87**	**21.15**	**19.32**	**88.1**



(a) Lower bounds

(b) Upper bounds

Figure 8: Tightness of DFD **lower and upper bounds**

5 CONCLUSION

In this paper, we propose several novel techniques (e.g., group border-based bound, DFD computation acceleration) to speedup the trajectory range query problem on discrete Fréchet distance. Our advanced approach is up to 50 times faster than the baseline solution we construct from the literature. A promising direction for future work is to devise error-guaranteed approximate solutions for the trajectory range query problem.

ACKNOWLEDGEMENT

This work was supported by the National Science Foundation of China (NSFC No. 61802163) and Guangdong Natural Science Foundation (Grant No. 2018A030310129).

REFERENCES

[1] Pankaj K Agarwal, Kyle Fox, Kamesh Munagala, Abhinandan Nath, Jiangwei Pan, and Erin Taylor. 2018. Subtrajectory Clustering: Models and Algorithms. In *PODS*. 75–87.

[2] Karl Bringmann and Wolfgang Mulzer. 2015. Approximability of the discrete Fréchet distance. In *Journal of Computational Geometry*, Vol. 7. 46–76.

[3] Yingyi Bu, Lei Chen, Ada Wai-Chee Fu, and Dawei Liu. 2009. Efficient anomaly monitoring over moving object trajectory streams. In *KDD*. 159–168.

[4] Jae-Gil Lee, Jiawei Han, and Kyu-Young Whang. 2007. Trajectory clustering: a partition-and-group framework. In *SIGMOD*. 593–604.

[5] Bo Tang, Man Lung Yiu, Kyriakos Mouratidis, and Kai Wang. 2017. Efficient motif discovery in spatial trajectories using discrete fréchet distance. In *EDBT*. 378–389.

[6] Dong Xie, Feifei Li, and Jeff M Phillips. 2017. Distributed trajectory similarity search. *PVLDB* 10, 11 (2017), 1478–1489.

[1] https://bit.ly/2E4sntq

[2] https://bit.ly/2zwrKVH

Repairing of Record Linkage: Turning Errors into Insight[*]

Quyen Bui-Nguyen, Qing Wang, Jingyu Shao, Dinusha Vatsalan
Australian National University, Canberra, Australia
quyen.m.bui.nguyen@gmail.com,{qing.wang,jingyu.shao,dinusha.vatsalan}@anu.edu.au

ABSTRACT

Linking records from different data sources, referred to as record linkage, is a longstanding but not yet satisfactorily resolved question in many fields of science. For practitioners, it is difficult to ensure the quality of linkage at the time of applying linkage techniques in real world applications. Instead, linkage errors are often detected later on, mostly by users of the applications. This not only requires us to repair errors, but also provides us with opportunities to observe the linkage quality and uncover why such errors occur. In viewing that record linkage is a complex and evolving process, we study how to acquire insights from linkage errors for achieving high-quality linkage. We propose a generic repairing framework which allows us to start with imperfect linkage models, and dynamically repair linkage models and errors for improved linkage quality. We have evaluated our repairing framework over three real-world datasets and the experimental results show that the performance of the proposed tree-structured classifier SVM-tree outperforms the baseline methods.

1 INTRODUCTION

Determining which records from one or more datasets correspond to the same real-world entities, referred to as *record linkage*, is a fundamental problem arising in many fields of computer science, e-commerce, health and social sciences [4]. Current research on record linkage mainly focuses on developing accurate and efficient linkage techniques, which are constrained by a number of factors (e.g., concerns about quality of data, ambiguity of domain knowledge and unavailability of training labels) and fails to capture the full variety of record linkage [8, 11]. Therefore, it is generally difficult to guarantee linkage quality at the time when record linkage techniques are applied.

One question that often arises is how to handle linkage errors which cannot be detected at the time of applying linkage techniques, but are reported later on, particularly by users of the record linkage based applications. Since data may be enriched with additional information over time, we can find linkage errors that are hidden in the linkage process. Essentially, record linkage is not a static and one-off task, but rather a complex, continuous and evolving process. Based on the insight acquired from linkage errors we may build a repairing framework to improve linkage quality through repairing errors.

Nevertheless, building such a repairing framework is challenging. Can we generalise errors into insight and leverage such insight into improving linkage models? Not all errors are equally useful for finding insight. Some errors may be outliers and generalising such errors may even deteriorate the performance of linkage models. Thus, we need a means for assessing the genericity of linkage errors - to what extent a linkage error can act as being a representative example of similar linkage errors. In this paper, we propose a dynamic repairing framework, which aims to turn errors into insight for improved linkage models and results. The central idea is to leverage detected errors to derive similar and relevant errors for maximally repairing an “imperfect" linkage model. Figure 1 presents a high-level overview of our repairing framework which incorporates a tree-structured classifier and an error analyser for supporting an iterative repairing process. More specifically, linkage errors such as false matches or non-matches may be detected by a user or governed by integrity rules and sent to an error analyser. The error analyser uses an error normality measure to determine whether an error can be generalised for improving the linkage model. Then qualified linkage errors are sent to the tree-structured classifier, and through refining the tree structure of the classifier, the linkage quality can be improved.

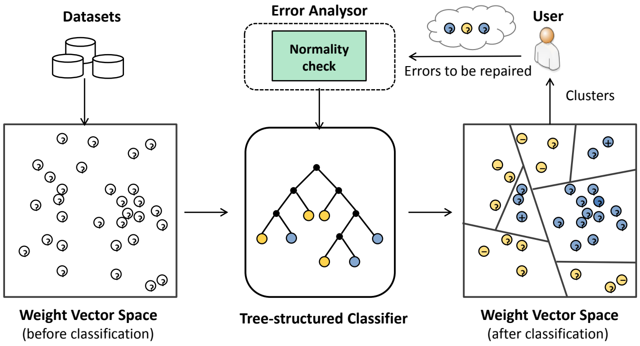


Figure 1: Proposed repairing framework, where each circled question mark represents an unlabeled weight vector, and each weight vector labeled as matches is shown with a blue $\oplus$, and non-matches with a yellow $\ominus$.

Our contributions are as follows: (1) We develop a generic repairing framework which allows us to start with imperfect linkage models and actively repair linkage models and errors for improved linkage quality. (2) We design a novel measure to assess the genericity of an linkage error, which indicates the probability of generalising errors into insight. (3) We have experimentally validated the effectiveness of the proposed repairing framework using several real-world datasets.

2 RELATED WORK

Record Linkage (RL) has long been central to the study of data integration and data cleaning [2, 5, 13]. Previous research has studied various aspects of the RL process such as: blocking, similarity comparison, classification, evaluation, active learning, and training data selection [6, 7, 9, 10, 12]. Nonetheless, these works primarily focused on preventing rather than repairing errors in linkage results.

In this paper, instead of only repairing detected errors, we also study the problem of repairing linkage models through detected linkage errors, i.e., linkage errors are leveraged to improve the performance of linkage models. It is worthy to note that,

[*] Produces the permission block, and copyright information

10.5441/002/edbt.2019.75

different from database repair [1, 3], we do not stipulate a minimal change requirement on our repairs. Existing approaches of database repair mostly deal with data errors either by obtaining consistent answers to queries [1] or by developing automated methods to find a repair that satisfies required constraints and also "minimally" differs from the original database [3]. These approaches are often computationally expensive and not applicable to repairing linkage models in real-world applications.

3 PROBLEM STATEMENT

Let R be a finite, non-empty set of records from one or more data sets. We assume that a set $F = \{f_1, \ldots, f_n\}$ of features is selected for performing record linkage tasks and each record $r \in R$ has a number of feature values $(v_1, \ldots, v_n)$ over F. Accordingly, each pair of records (r_i, r_j) over R is associated with a n-dimensional *weight vector* $\mathbf{x_{ij}} = (x_1, \ldots, x_n) \in [0,1]^n$ where each x_k represents the similarity between the feature values of f_k in r_i and r_j. For example, a pair of records (r_1, r_2) may have three features $\{\mathit{fname}, \mathit{sname}, \mathit{age}\}$ with $r_1.\mathit{fname} = \mathit{Rob}$, $r_1.\mathit{sname} = \mathit{Smith}$, $r_1.\mathit{age} = 30$, $r_2.\mathit{fname} = \mathit{Robert}$, $r_2.\mathit{sname} = \mathit{Smith}$ and $r_2.\mathit{age} = 31$, and correspond to a 3-dimensional weight vector $(0.5, 1, 0.5)$.

Let X consist of the weight vectors to which all pairs of records in R. Some blocking technique may also be applied for efficiency [9, 11]. In viewing that weight vectors are a set of data points in X, a *weight vector space* (X, δ) can be defined for X together with a *similarity metric* $\delta : X \times X \mapsto [0,1]$ that satisfies the non-negativity, identity, symmetry and triangle inequality properties as defined in [14]. Thus, the higher two data points $\mathbf{x_1}$ and $\mathbf{x_2}$ are similar, the larger the value $\delta(\mathbf{x_1}, \mathbf{x_2})$ is. A *partition* of X is a set $\{X_1, \ldots, X_q\}$ of pairwise disjoint subsets with $\bigcup_{1 \le i \le m} X_i = X$. We call each (X_i, δ) $(i \in [1, q])$ a *weight vector subspace* of (X, δ).

The label of a weight vector $\mathbf{x} \in X$ is denoted as $y(\mathbf{x})$ and $y(\mathbf{x}) \in \{1, -1\}$, where 1 refers to a true match and -1 refers to a true non-match. In this paper, we consider a linkage model as a binary classifier $c : X \longrightarrow \{-1, 1\}$, which can classify a weight vector $\mathbf{x}$ either as 1 or as -1. Thus, two kinds of errors may occur in record linkage: (1) false positives (i.e. false matches) and (2) false negatives (i.e. false non-matches):

- $fp(c) = \{\mathbf{x} \in X | c(\mathbf{x}) = -1 \wedge y(\mathbf{x}) = 1\}$;
- $fn(c) = \{\mathbf{x} \in X | c(\mathbf{x}) = 1 \wedge y(\mathbf{x}) = -1\}$.

Definition 3.1. (RL repair problem) Let c be a linkage model and E be a set of detected errors. Then the *RL repair problem* is to find a linkage model c^* such that the following objective function is minimised:

$$\operatorname*{argmin}_{c} \frac{fp(c^*) + fn(c^*)}{fp(c) + fn(c)} \tag{1}$$

The focus of the RL repair problem is on maximally repairing linkage models through detected errors, i.e., gain maximum insights from errors, rather than repairing only detected errors.

4 REPAIRING FRAMEWORK

We propose a novel framework for solving the RL repair problem.

4.1 Error Analyser

Generally, linkage errors may be caused by various reasons, e.g., poor data quality, biased classifiers and insufficient training data. Although it is desirable to repair all errors, linkage errors are not equally informative for improving a linkage model. For instance, repairing a linkage model based on errors that are indeed outliers often leads to the overfitting problem. On the other hand, errors that repeatedly occur under certain similar conditions may suggest a pattern of these errors which is more informative for repairing a linkage model. To quantify the informativeness of errors, we thus develop the notion of error normality to measure the degree errors can be generalised from specific cases to more structured ones.

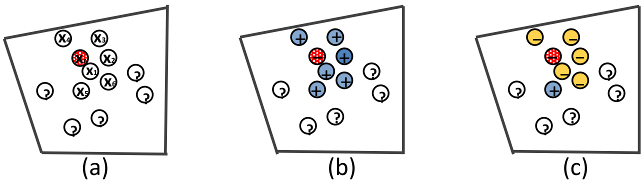


Figure 2: A false match, denoted as $\mathbf{x_r}$ and highlighted as being red, is detected in a weight vector subspace X_i.

Definition 4.1. Let (X, δ) be a weight vector subspace, $\mathbf{x} \in X$, and $N(\mathbf{x}, X)$ refer to the set of k nearest neighbours of $\mathbf{x}$ in (X, δ). Then the *normality* of $\mathbf{x}$ in X, denoted as $\rho(\mathbf{x})$, is defined as:

$$\rho(\mathbf{x}) = y(\mathbf{x}) \cdot \frac{\sum\limits_{\mathbf{x}' \in N(\mathbf{x}, X)} \delta(\mathbf{x}, \mathbf{x}') \cdot y(\mathbf{x}')}{\sum\limits_{\mathbf{x}' \in N(\mathbf{x}, X)} \delta(\mathbf{x}, \mathbf{x}')} \tag{2}$$

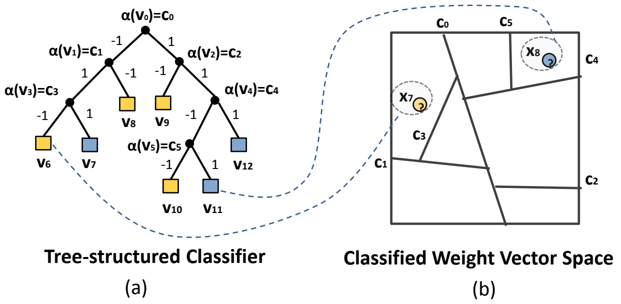
The values of normality range from -1 to 1, indicating the possibility of generalising an error $\mathbf{x}$ into improving a linkage model: (1) $\rho(\mathbf{x}) = -1$ means that all the nearest neighbours in $N(\mathbf{x}, X)$ have the opposite label to $\mathbf{x}$; (2) $\rho(\mathbf{x}) = 1$ means that all the nearest neighbours in $N(\mathbf{x}, X)$ have the same label as $\mathbf{x}$. In addition to the labels, the distances of $\mathbf{x}$ and its nearest neighbours are also taken into account by the normality. For each neighbour in $N(\mathbf{x}, X)$, the close its distance is to $\mathbf{x}$, the more influence it has on $\rho(\mathbf{x})$. Intuitively, the normality of an error $\mathbf{x}$ is measured by a weighted majority vote of their nearest neighbours, i.e., the margin between strength of voting on the same label $y(\mathbf{x})$ and strength of voting on the opposite label $-1 \cdot y(\mathbf{x})$.

Example 4.2. Suppose that, for the detected error $\mathbf{x_r}$ in Figure 2, we have $N(\mathbf{x_r}, X_i) = \{\mathbf{x_1}, \ldots, \mathbf{x_6}\}$, and $\delta(\mathbf{x_r}, \mathbf{x_i})$ is 0.9, 0.8, 0.8, 0.8, 0.7 and 0.7 for $i = 1, 2, 3, 4, 5, 6$, respectively. Then $\rho(\mathbf{x_r}) = 1$ in Figure 2.b and $\rho(\mathbf{x_r}) = (0.9 + 0.8 + 0.8 + 0.8 + 0.7 - 0.7)/(0.9 + 0.8 + 0.8 + 0.8 + 0.7 + 0.7) = 0.7$ in Figure 2.c.

4.2 Tree-structured Classifiers

To repair linkage models effectively, the repairing framework needs to offer great flexibility and expressive power in refining linkage models. We thus propose to use a tree-based structure for repairing classifiers. A tree-structured classifier can be formalised as a tuple (T, C, α, β), where T is a binary tree, C is a set of binary classifiers, α is a labeling function that assigns a classifier $c_i \in C$ to each internal vertex v_i of T, i.e., $\alpha(v_i) = c_i$, and β assigns a label in $\{-1, 1\}$ to each edge (v_i, v_j) of T, i.e., $\beta(v_i, v_j) \in \{-1, 1\}$. Therefore, given a weight vector $\mathbf{x} \in X$, a tree-structured classifier $h = (T, C, \alpha, \beta)$ classifies $\mathbf{x}$ using the following condition: $h(\mathbf{x}) = \beta(v_{n-1}, v_n)$ if there exists a path $\langle v_0, v_1, \ldots, v_n \rangle$ in T starting from the root vertex v_0 of T and ending at a leaf vertex v_n such that $\alpha(v_k) = c_k$ and $\bigwedge_{k \in [0, n-1]} c_k(\mathbf{x}) = \beta(v_k, v_{k+1})$.

Example 4.3. Consider two weight vectors $\mathbf{x_7}$ and $\mathbf{x_8}$ from the weight vector space depicted in Figure 3.b. The tree-structured classifier in Figure 3.a has classified this weight vector space into many smaller weight vector subspaces, each being assigned a



Tree-structured Classifier

(a)

Classified Weight Vector Space

(b)

Figure 3: A schematic layout of a tree-structured classifier and the corresponding classified weight vector space.

label of either −1 (highlighted by a yellow square) or 1 (highlighted by a blue square). We have $h(\mathbf{x}_7) = -1$ due to a path $\langle v_0, v_1, v_3, v_6\rangle$ with $c_0(\mathbf{x}_7) = -1, c_1(\mathbf{x}_7) = 1$ and $c_3(\mathbf{x}_7) = -1$. Similarly, we have $h(\mathbf{x}_8) = 1$ because there is a path $\langle v_0, v_2, v_4, v_5, v_{11}\rangle$ with $c_0(\mathbf{x}_8) = 1$, $c_2(\mathbf{x}_8) = 1$, $c_4(\mathbf{x}_8) = -1$, and $c_5(\mathbf{x}_8) = 1$.

Initially, the binary tree T in a tree-structured classifier only has one internal vertex as the root, which represents the initial classifier c_0, and two leaf vertices that represent two weight vector subspaces X_1 and X_2 of the given weight vector space X classified by the initial classifier c_0 such that $\forall \mathbf{x}_1 \in X_1(c_0(\mathbf{x}_1) = 1)$ and $\mathbf{x}_2 \in X_2(c_0(\mathbf{x}_2) = -1)$. Then when a linkage error $\mathbf{x}$ is detected, c can be repaired by extending T to T' such that one of the leaf vertices of T (say v_t) is replaced by an internal vertex with two leaf vertices. Accordingly, the weight vector subspace associated with v_t is divided into two smaller weight vector subspaces (one with the label 1 and the other with the label -1). This process is iteratively conducted when repairing errors.

As a result, an important property of tree-structured classifiers is to support *bounded repairs*, i.e., only weight vectors in a specified subspace are affected by a repair. This property can reduce the risk of incorrect repairs and lead to convergence on quality of a linkage model after a finite number of repairing iterations.

4.3 Repairing Algorithm

Our repairing algorithm is provided in Figure 4. Given a weight vector space X which is partitioned into $\{X_1, \ldots, X_n\}$ by an initial classifier c_0, the algorithm iteratively leverages errors to repair a tree-structured classifier until the labelling budget is used up or no more errors are detected (Line 2).

In this algorithm, E_h refers to detected errors of h, q keeps record of the number of labels requested from the human oracle, $\textsc{select\_error}(E_h)$ randomly selects an error e from E_h, and $\textsc{find\_subspace}(\mathbf{X}, e)$ searches for a subspace X_{cur} of X which contains e (Lines 3-4). If $|X_{cur}|$ is greater than S_{min} (Line 5), then $\textsc{nn\_select}(e, X_{cur}, k)$ finds k nearest neighbours of e in X_{cur}, and their labels are requested (Lines 6-7). If the normality $\rho(e)$ of e is greater than the threshold t (Line 9), $\textsc{fn\_select}(e, X_{cur}, k)$ finds k farthest neighbours of e and their labels are requested (Lines 10-11). If the purity of X_{cur} is lower than the threshold p (Line 13), the classifier c_{q+1} is trained and extends c_q by classifying X_{cur} into two subspaces X^M_{cur} (for matches) and X^N_{cur} (for non-matches) (Lines 14-16). Then accordingly, the tree-structured classifier h is repaired by adding c_{q+1} (Line 17). We consider errors whose normality values are lower than the threshold them as outliers. The algorithm returns a tree-structured classifier h with improved linkage quality (Line 23).

```
Input: initial classifier: $c_0$, normality threshold: $t \in [-1.1]$,
       human oracle for labelling: $\textsc{oracle}()$, budget limit: $b$,
       number of nearest neighbours: $k$, purity threshold $p$,
       minimum size of weight vector subspaces: $s_{min}$
Output: a tree-structured classifier $h$
1:  $\mathbf{X} \leftarrow \{X_1, \ldots, X_n\}$, $q \leftarrow 1$, $h \leftarrow c_0$
2:  while $q \le b$ and $|E_h| > 0$ do:
3:      $e \leftarrow \textsc{select\_error}(E_h)$
4:      $X_{cur} \leftarrow \textsc{find\_subspace}(\mathbf{X}, e)$
5:      if $|X_{cur}| \ge s_{min}$ then:
6:          $V_{nn} \leftarrow \textsc{nn\_select}(e, X_{cur}, k)$
7:          $V^L_{nn} \leftarrow \textsc{oracle}(V_{nn})$
8:              $V^L \leftarrow V^L \cup V^L_{nn}$
9:          if $\rho(e) \ge t$ then:
10:             $V_{fn} \leftarrow \textsc{fn\_select}(e, X_{cur}, k)$
11:             $V^L_{fn} \leftarrow \textsc{oracle}(V_{fn})$
12:             $V^L \leftarrow V^L \cup V^L_{fn}$
13:             if $purity(X_{cur}) < p$ then
14:                 $c_{q+1} \leftarrow \textsc{train}(c_q, X_{cur}, V^L)$
15:                 $X^M_{cur}, X^N_{cur} \leftarrow c_{q+1}.\textsc{classify}(X_{cur})$
16:                 $\mathbf{X} \leftarrow \mathbf{X} - \{X_{cur}\} \cup \{X^M_{cur}, X^N_{cur}\}$
17:                 $h = \textsc{repair\_classifier}(h, c_{q+1})$
18:             endif:
19:         endif:
20:     endif:
21:     $q \leftarrow q + 1$
22: endwhile:
23: return $h$
```

Figure 4: Repairing Algorithm

5 EXPERIMENTS

We have implemented the repairing framework to experimentally validate its effectiveness on three real-world datasets.

Baseline methods. We have used support vector machines (SVM) with three different kernels: linear, polynomial, and Gaussian (RBF) and a decision tree as the baseline methods. For our tree-structured classifier, we used SVMs with linear kernel as binary classifiers for its internal vertices, called SVM-tree. We also used the following parameters: $s_{min} = 25$, $p = 0.97$, and $k = 50$.

Sampling methods. We use KNN+FNN to refer to the k nearest and farthest neighbours sampling method described in the repairing algorithm in Figure 4. To evaluate its effectiveness, we compared it with three other sampling methods: (1) KNN, (2) FNN, and (3) random, in which k nearest, k farthest, and k random neighbours of an error are sampled, respectively.

Datasets. We have used three datasets: (1) Cora, which contains 1,295 publication records and is publicly available [1], (2) DBLP_ACM, which contains 4910 bibliographic records from the DBLP and ACM websites [8], and (3) NCVR, using two datasets with 1048576 and 613767 records respectively from the North Carolina Voter Registration (NCVR) database[2]. We did not apply any blocking on Cora but used *publication year* and the 1st char of *title* for DBLP_ACM, and *last_name* and *first_name* for NCVR.

Performance of classifiers. Figure 5 shows the performance of SVM-tree in comparison to the baseline methods. SVM-tree

[1] Available from: http://secondstring.sourceforge.net

[2] Available from: ftp://alt.ncsbe.gov/data/

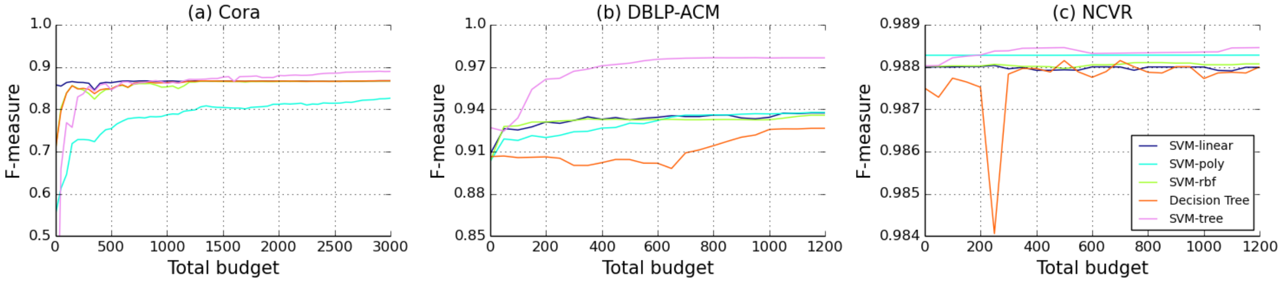


Figure 5: Comparison to the baseline methods, where the sampling method is KNN+FNN and normality threshold is -0.6.

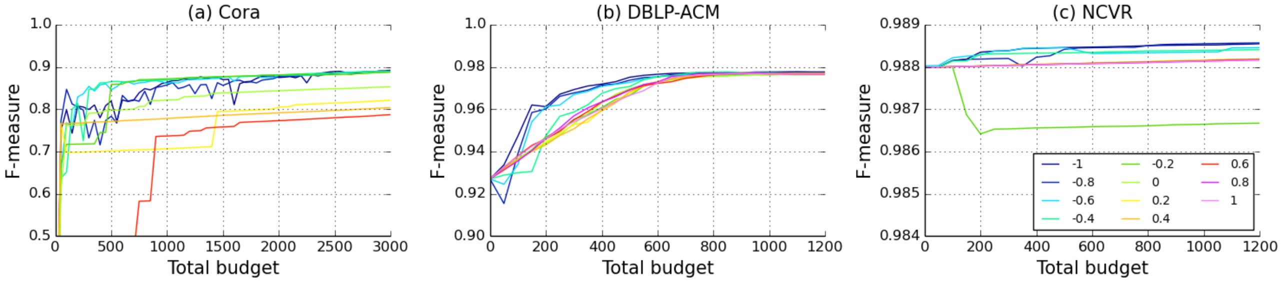


Figure 6: SVM-tree with different normality thresholds, where the sampling method is KNN+FNN.

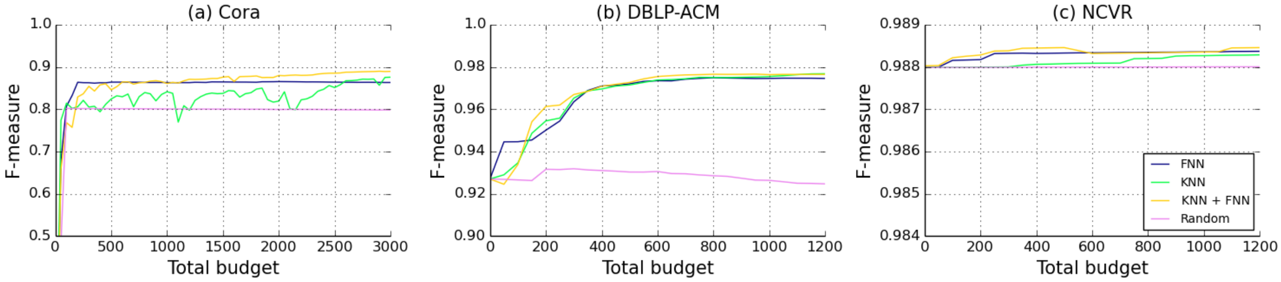


Figure 7: SVM-tree with different sampling methods, where the normality threshold is -0.6.

generally performs better than the other methods for all datasets. For Cora, SVM-tree performs increasingly better when more errors are detected. For DBLP_ACM, it has a significant increase in performance. For NCVR, it is a clean dataset that produces high quality results (around 0.988 f-measure) for all methods; nonetheless, SVM-tree is still marginally better.

Effect of normality. Figure 6 presents the experimental results for classification using SVM-tree with varying thresholds. The KNN+FNN sampling method is used in this experiment. A normality threshold of -0.4 to -1 produces the best performance for all datasets. The threshold of -0.6 produces consistently stable good results over all datasets.

Performance of sampling. Figure 7 shows the performance of SVM-tree using different sampling methods and the normality threshold is set to -0.6. KNN+FNN generally produces the best and most stable results among all the sampling methods, particularly over the datasets Cora and DBLP_ACM. When the budget is low, the performance of FNN is comparable to KNN+FNN. The performance of KNN is not stable on Cora. The random sampling has the worst performance among all the methods.

ACKNOWLEDGEMENT

This work was partially funded by the Australian Research Council (ARC) under Discovery Project DP160101934.

REFERENCES

[1] Foto N Afrati and Phokion G Kolaitis. 2009. Repair checking in inconsistent databases: algorithms and complexity. In *ICDT*. 31–41.

[2] I. Bhattacharya and L. Getoor. 2007. Collective entity resolution in relational data. *TKDD* 1, 1 (2007), 5.

[3] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. 2007. Improving data quality: Consistency and accuracy. In *PVLDB*. 315–326.

[4] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate Record Detection: A Survey. *IEEE TKDE* 19 (2007), 1–16.

[5] I.P. Fellegi and A.B. Sunter. 1969. A theory for record linkage. *J. Amer. Statistical Assoc.* 64, 328 (1969), 1183–1210.

[6] Jeffrey Fisher, Peter Christen, and Qing Wang. 2016. Active learning based entity resolution using Markov logic. In *PAKDD*. 338–349.

[7] Jeffrey Fisher, Peter Christen, Qing Wang, and Erhard Rahm. 2015. A Clustering-Based Framework to Control Block Sizes for Entity Resolution. In *KDD*. 279–288.

[8] Hanna Köpcke, Andreas Thor, and Erhard Rahm. 2010. Evaluation of entity resolution approaches on real-world match problems. *VLDB* 3, 1-2 (2010), 484–493.

[9] George Papadakis, Jonathan Svirsky, Avigdor Gal, and Themis Palpanas. 2016. Comparative analysis of approximate blocking techniques for entity resolution. *VLDB* 9, 9 (2016), 684–695.

[10] Jingyu Shao and Qing Wang. 2018. Active Blocking Scheme Learning for Entity Resolution. In *PAKDD*. 350–362.

[11] Qing Wang, Mingyuan Cui, and Huizhi Liang. 2016. Semantic-Aware Blocking for Entity Resolution. *TKDE* 28, 1 (2016), 166–180.

[12] Qing Wang, Dinusha Vatsalan, and Peter Christen. 2015. Efficient Interactive Training Selection for Large-Scale Entity Resolution. In *PAKDD*. 562–573.

[13] Steven Euijong Whang and Hector Garcia-Molina. 2010. Entity resolution with evolving rules. *VLDB* 3, 1-2 (2010), 1326–1337.

[14] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. 2006. *Similarity search: the metric space approach*. Vol. 32. Springer.

Short Paper

Streaming HyperCube: A Massively Parallel Stream Join Algorithm

Yuan Qiu
HKUST
yqiuac@cse.ust.hk

Serafeim Papadias
HKUST
spapadias@cse.ust.hk

Ke Yi
HKUST
yike@cse.ust.hk

ABSTRACT

Streaming join is an essential operation in many real-time applications. A lot of research has been devoted to the study of distributed streaming join algorithms. However, existing solutions rely on heuristics and cannot handle data skew optimally. On non-streaming, static data, the HyperCube algorithm ensures a balanced load across all processors in an optimal way. In this paper, we extend this algorithm to the streaming setting, which can adapt the HyperCube configuration depending on the current data distribution. Some preliminary experimental results are provided to demonstrate the efficiency our algorithm.

1 INTRODUCTION

The prevalence of applications in financial trading, sensor networks, traffic analysis and web data management has brought attention to query processing over data streams. Various Distributed Stream Processing Systems (DSPSs) have emerged such as Flink [1], Spark Streaming [2] and Apache Storm [3]. As one of the most critical operations in database management systems, join has been extensively studied in the literature both in static models [5–8, 14, 16] and streaming models [11, 12, 17, 18].

Low latency and high throughput are two essentials for an efficient streaming join algorithm. In a massively distributed system, a key to achieving these goals is to ensure a good load balance among the workers. Over static data, the HyperCube algorithm [6] and its extensions [8] achieve an optimal load balance. However, this algorithm crucially depends on the data statistics, in particular, the set of heavy hitters and their frequencies. In a dynamic setting like streaming data, these statistics are changing all the time, which renders the HyperCube algorithm inapplicable.

In this paper, we propose Streaming HyperCube (SHC), an adaption of the HyperCube algorithm to the streaming setting. A key challenge in designing SHC is how to adaptively change the HyperCube configuration as the data statistics change over time. On the one hand, we want to keep the configuration as close as possible to the optimal setting the HyperCube uses for static data. On the other hand, we also want to avoid too frequent configuration changes, since each configuration change requires migration of states, which introduces communication overhead and stalls streaming processing. The SHC algorithm supports both full-stream joins as well as joins over a sliding window.

Below, we first review the related work in Section 2, in particular the static HyperCube algorithm. In Section 3 we describe the Streaming HyperCube algorithm. We have implemented the algorithm in Flink. In Section 4 we describe its implementation and show some preliminary experimental results comparing with some state-of-the-art stream join algorithms.

2 RELATED WORK

2.1 Parallel Hash Join

The Parallel Hash Join (PHJ) algorithm is the default join algorithm used in many state-of-the-art DSPSs [2, 3, 10, 19], including Flink. Consider a (natural) join $R(A,B) \bowtie S(B,C)$. Let p be the number of workers in the system, numbered $1, 2, \dots, p$. The algorithm utilizes a random hash function $h : B \rightarrow [p] = \{1, \dots, p\}$ to assign tuples $(a,b) \in R$ and $(b,c) \in S$ to worker $h(b)$. Each worker is responsible for producing the join results on all tuples assigned to it. More precisely, it maintains two hash tables built on attribute B, one for tuples from R, one for tuples from S. Upon the arrival of a tuple $(a,b) \in R$, it inserts it to the hash table on R, and probes the hash table on S with key b to find all tuples in S that can join with (a,b).

When there is no skew, this algorithm performs relatively well. However, when there are heavy hitters, namely, many tuples in R or S share the same value on attribute B, the load may no longer be balanced, since all tuples with the same b value must be sent to the same worker. In the worst case, if all tuples have the same value on b, the join degenerates into a Cartesian product, and one worker will be doing all the work while the other $p-1$ workers are idle.

2.2 HyperCube

The worst case scenario of the aforementioned PHJ algorithm happens when there is only a single value $b \in B$. In this case, the join degenerates into a Cartesian product. This problem was solved in [6] by the following HyperCube algorithm.

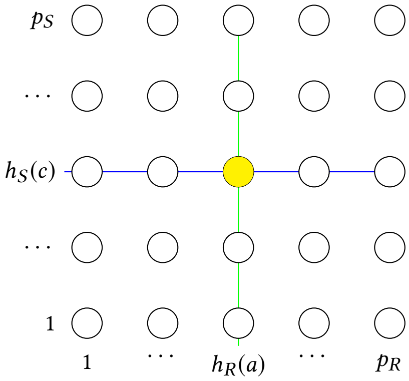


Figure 1: The HyperCube algorithm. Tuple (a,b) is sent to all workers in the column $h_R(a)$, and the tuple (b,c) is sent to all workers in the row $h_S(c)$. The worker at coordinate $(h_R(a), h_S(c))$ produces the join result (a,b,c).

The p workers are organized into a $p_R \times p_S$ grid (see Figure 1), where $p_R = |R|\sqrt{\frac{p}{|R|\cdot|S|}}$, $p_S = |S|\sqrt{\frac{p}{|R|\cdot|S|}}$. Two hash functions $h_R : A \rightarrow [p_R]$, $h_S : C \rightarrow [p_S]$ are used. Each tuple $(a,b) \in R$ is sent to all workers with coordinates $(h_R(a), *)$ and each $(b,c) \in S$

Series ISSN: 2367-2005

10.5441/002/edbt.2019.76

is sent to all workers $(*, h_S(c))$. As figure 1 shows, each output tuple is handled by exactly one worker. With high probability, each worker receives

$$\tilde{O}\left(\frac{|R|}{p_R} + \frac{|S|}{p_S}\right) = \tilde{O}\left(\sqrt{\frac{OUT}{p}}\right)$$

tuples, where the output size $OUT = |R| \cdot |S|$ for Cartesian product. Assuming $|R| = |S|$, this is better than the $O(|R|)$ load of the PHJ algorithm by an $O(\sqrt{p})$ factor.

Beame et al. [8] have generalized this algorithm to general joins. Let $R(b) = \sigma_{B=b}R$, and similarly define $S(b)$. The idea is to first partition the join attribute B into heavy hitters B^H and light hitters B^L. Let N denote the total number of tuples:

$$B^H = \left\{ b \in B \;\middle|\; |R(b)| > \frac{N}{p} \text{ or } |S(b)| > \frac{N}{p} \right\}$$

and B^L is the remaining attribute values. The light hitters B^L will be handled by the PHJ algorithm. Since there is no skew in the light hitters, the load is bounded by $\tilde{O}(N/p)$ for any worker. For each heavy value $b \in B^H$, $p_{(b)}$ workers are allocated to handle it using the *shares* algorithm, where

$$p_{(b)} = \frac{OUT(b)}{OUT^H} \cdot p = \frac{|R(b)| \cdot |S(b)|}{OUT^H} \cdot p$$

and $OUT^H = \sum_{b \in B^H} OUT(b)$ denotes the output size generated by all heavy hitters. The load of each worker for handling heavy hitters is upper bounded by $\tilde{O}(\sqrt{OUT/p})$. Therefore, the total load is $\tilde{O}\left(\frac{N}{p} + \sqrt{\frac{OUT}{p}}\right)$, which has been shown to be optimal.

However, this algorithm does not work in the streaming model, because it needs all the heavy hitter information to decide the configuration of each cube, namely how to set $p(b)$ for all the b's, and how to set the dimensions of each cube. Furthermore, the heavy hitter set may change throughout stream processing, which requires the HyperCube configuration to change correspondingly. These will be solved by our algorithm described in section 3.

2.3 BiStream Join

Lin et al. [17] proposed the Join-Biclique (JB) algorithm for computing joins over distributed streaming data. Their BiStream system divides the join into two stages, a routing stage for storage and a joining stage for producing outputs. Their work outperforms existing systems [11] in terms of memory consumption, scalability, latency and throughput. However, the algorithm relies on heuristics and does not work well on highly skewed data. We will compare SHC with a Flink implementation of the Join-Biclique algorithm in Section 4.

2.4 Cell Join

Cell Join presented by Gedik et al. [12] solves the windowed streaming join problem specifically on Cell processors. Following the three-step procedure described by Kang et al. [15], the algorithm parallelizes the scan task of streams over available workers to achieve high performance. However, since re-partition is needed whenever a new tuple arrives, this algorithm suffers from scalability issues. Although it supports band joins and can be extended to multi-way joins, its framework is solely designed for the IBM's cell processor.

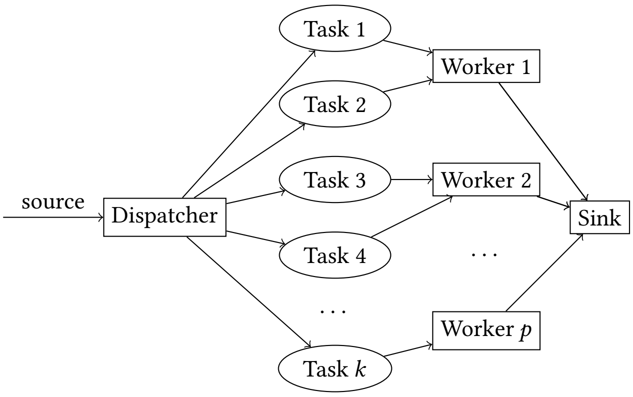


Figure 2: System Architecture.

2.5 Handshake Join

Roy et al. [18] presented a highly parallelizable handshake join for window streams. Hardware acceleration was implemented to achieve high data throughput. The algorithm uses similar window semantics to the ones in [15] by dividing the window into subwindows. The two streams flow in opposite directions so that each pair of subwindows has a handshake. This algorithm adopts the batch-processing model and may introduce disorder in the output tuple sequence, meaning it is not suitable for applications requiring instant outputs.

3 STREAMING HYPERCUBE

3.1 System Architecture

Figure 2 describes the architecture of our system. There are p workers along with a dispatcher and a sink. The dispatcher preprocesses input tuples after receiving them. The tuples are partitioned into *tasks*, which are then dynamically allocated to physical workers at runtime. The workers compute join results on-the-fly and pass them to the sink. Our system adopts the event-triggered model of Flink, where the output is updated immediately after the arrival of each tuple to ensure low latency.

3.2 Algorithm

The dispatcher runs an approximate heavy hitters tracking algorithm [9] to keep track of the heavy hitters and their approximate frequencies. Specifically, it classifies a join value $b \in B$ to be heavy hitter if $N(b) = |R(b)| + |S(b)| > \frac{N}{p}$ and light hitter if $N(b) < (\frac{1}{p} - \epsilon)N$, N is the current stream length (for full stream joins) or window length (for sliding window joins), and ϵ is some small constant. A value with frequency in between may be either heavy or light. As long as $\epsilon \leq \frac{1}{2p}$, there are at most $2p$ heavy hitters. Note that the heavy hitter tracking algorithm also maintains approximated frequencies $|R(b)|$ and $|S(b)|$ for each heavy hitter b, each with at most ϵN additive error. Therefore the corresponding output size $OUT(b) = |R(b)| \cdot |S(b)|$ can be estimated with at most $2\epsilon N$ additive error. Summing over all the heavy values, we obtain an estimate of OUT^H, the output size of all heavy hitters, with no more than $2p\epsilon N$ additive error.

Similar to the HyperCube algorithm, the dispatcher treats heavy and light hitters differently. The light hitters are handled by the PHJ algorithm using p tasks and an associated hash function

$h^L : B \longrightarrow [p]$.[1] Each heavy value b occupies $p_{(b)}$ tasks organized into a $p_{R(b)} \times p_{S(b)}$ grid. Two hash functions $h_{R(b)} : A \longrightarrow [p_{R(b)}]$ and $h_{S(b)} : C \longrightarrow [p_{S(b)}]$ are involved.

Suppose a tuple $(a, b) \in R$ arrives at time t. The dispatcher first reflects the update to the heavy-hitter tracking algorithm, so that the heavy hitter set and its statics are still valid approximations. If b is a light hitter, it is sent to task $h^L(b)$ for light hitters and the assigned worker produces the output. If b is a heavy hitter, it is sent to all tasks labeled $(h_{R(b)}(a), *)$ in the $p_{R(b)} \times p_{S(b)}$ grid allocated to b.

A key difference between SHC and the static HyperCube algorithm is that the number of tasks allocated to each heavy hitter, as well as the grid dimensions, may change over time. This is done by a *state migration* between tasks. To see why this is necessary, consider a value b, which starts light but continuously receives tuples over time. If only a single task is allocated to handle it, that worker will have a huge load. For efficient state migration, the dispatcher stores the current number of tasks $p_{R(b)} \times p_{S(b)}$ allocated for each heavy hitter b. According to the estimated statistics, it can also calculate a *desired* number of tasks for b, namely

$$p^d_{R(b)} = \frac{|R(b)|}{\sqrt{OUT^H}}\sqrt{p}, \text{ and } p^d_{S(b)} = \frac{|S(b)|}{\sqrt{OUT^H}}\sqrt{p}$$

When the estimates of OUT^H, $|R(b)|$, and $|S(b)|$ change over time, the dispatcher will compare the currently assigned grid dimension with the desired one. If $p_{R(b)} > 2p^d_{R(b)}$, we reduce $p_{R(b)}$ to half. Specifically, each task

$$(x, y), x = \frac{1}{2}p_{R(b)} + 1, \ldots, p_{R(b)}, y = 1 \ldots, p_{S(b)}$$

sends all its tuples in $R(b)$ to task $(x - \frac{1}{2}p_{R(b)}, y)$ and then gets released. If $p_{R(b)} < \frac{1}{2}p^d_{R(b)}$, double $p_{R(b)}$. Specifically, create $p_{(b)}$ new tasks and extend the grid to $2p_{R(b)} \times p_{S(b)}$. Each task

$$(x, y), x = 1, \ldots, p_{R(b)}, y = 1 \ldots, p_{S(b)}$$

sends all its tuples in $S(b)$ to $(x + p_{R(b)}, y)$. For each tuple $(a', b) \in R(b)$ in the task, apply a new hash function $h' : A \longrightarrow \{0, 1\}$ to it, where 0 means that the tuple stays, and 1 means the tuple gets sent to task $(x + p_{R(b)}, y)$. The new hash function associated with $p_{R(b)}$ is $h_{R(b)} + p_{R(b)} \cdot h' : A \longrightarrow 2p_{R(b)}$. It is similar for $p_{S(b)}$.

Finally, the dispatcher manages assignment of tasks to workers. Since the tasks are almost balanced, the dispatcher keeps a count on the total number of tasks each worker is currently handling. When a new task is created, it will be sent to the worker with the minimum number of tasks. When a task is released, the dispatcher decreases the count of the corresponding worker by 1. Note that some tasks may be assigned to the same worker, so that they are only logically separated, but the communication between them can be easily achieved. The worker nodes receiving tasks compute join results, and send it to the sink.

3.3 A Running Example

Figure 3 gives a running example of state migration. Suppose at some point of the algorithm, 500 tuples have been received, where 200 tuples contain b_1 and 250 tuples contain b_2. There are 50 other tuples of light hitters. The output size from heavy hitters is

$$OUT^H = |R(b_1)| \cdot |S(b_1)| + |R(b_2)| \cdot |S(b_2)| = 20k$$

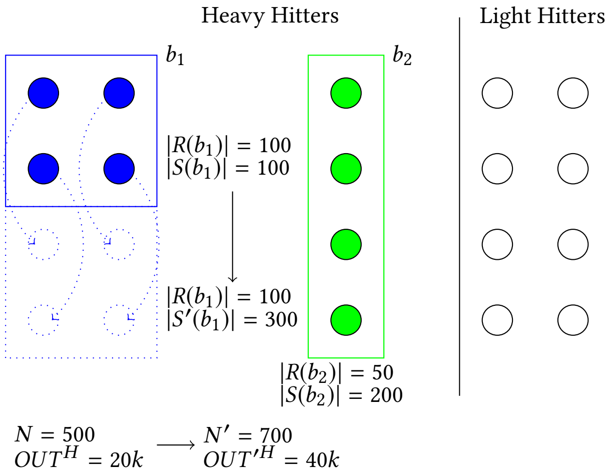


Figure 3: Example task allocation for $p = 8$. There are 8 tasks for light hitters. 2 heavy values b_1 and b_2 each occupies 4 tasks. The dotted circles represents the new tasks created in state migration.

Also assume task allocation is optimal at this time. Indeed, $p_{R(b_1)} = 2, p_{S(b_1)} = 2, p_{R(b_2)} = 1, p_{S(b_2)} = 4$. Note that the task layouts are different between two heavy hitters because they have different internal skew.

Suppose after this state, we received 300 more tuples from $S(b_1)$ and no other tuples. Consider b_1, the dispatcher calculates the new desired number of tasks as:

$$\begin{aligned} p^d_{R(b_1)} &= \frac{|R(b_1)|}{\sqrt{OUT^H}}\sqrt{p} \\ &= \frac{100}{\sqrt{40,000}}\sqrt{8} = 1.414 \end{aligned}$$

Since $2 = p_{R(b_1)} \leq 2p^d_{R(b_1)}$, state migration does not need to be performed for $p_R(b_1)$. However,

$$\begin{aligned} p^d_{S(b_1)} &= \frac{|S(b_1)|}{\sqrt{OUT^H}}\sqrt{p} \\ &= \frac{300}{\sqrt{40,000}}\sqrt{8} = 4.243 \end{aligned}$$

indicates that $2 = p_{S(b_1)} < \frac{1}{2}p^d_{S(b_1)}$. The dispatcher decides $p_S(b_1)$ should be doubled, and creates 4 more tasks to handle it. As Figure 3 illustrates, the 2×2 grid is extended to a 2×4 one, where arrows denote the flow of tuples.

For b_2, the desired numbers are 0.707 and 2.828 respectively. No migration is needed since the original 1×4 grid is sill an optimal configuration, up to a factor of 2.

4 EVALUATION

In this section, we discuss implementation details and provide comparative evaluations of our algorithm with the widely implemented PHJ algorithm and state-of-the-art Join-Biclique (JB) [17] algorithm.

4.1 Implementation

In the current implementation of our SHC algorithm, we maintain a copy of each stream in the dispatcher node in the form of a hash table. The dispatcher is responsible for redistributing

[1] $[p] = \{1, \ldots, p\}$

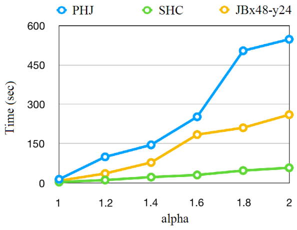


Figure 4: Zipf, varying α.

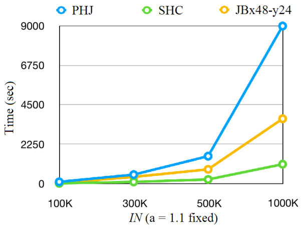


Figure 5: Zipf, varying IN.

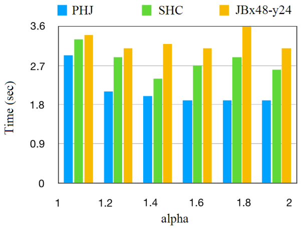


Figure 6: TPC-H, varying α.

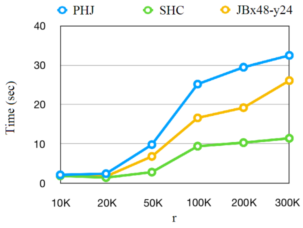


Figure 7: COREL, varying r.

certain parts of the state hashtables to the join processing workers according to state migration policies. In addition, the dispatcher may invalidate the state of certain heavy hitters from specific workers.

4.2 Experimental Setup

Environment. We conduct all experiments on an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz server that has 12 cores with 4 threads each and runs Red Hat 4.8.5. The server has 256 GB of memory, which suffices for maintaining all raw and intermediate data even if all tuples are sent to one processor only. Flink 1.4.2 is installed on the server and the implemented algorithms have maximum parallelism of $p = 48$ processing units.

Data sets. Our preliminary experiments focus on full-stream joins. We use two synthetic and one real data sets to evaluate our algorithm.

The first synthetic data set is generated from the Zipf distribution with varying value for α, which controls the level of the skewness. The second synthetic data set is the TPC-H benchmark [4]. Specifically, we choose to experiment on the Q5.

For the real data set, we use the same COREL dataset as in [13] for applying SRHC on similarity joins. The data set contains a set of 15,000 points with dimentionality 64. Each point corresponds to a histogram of a unique color image collected from the COREL repository. Based on a provided similarity threshold we apply a locality-sensitive hash function on every point, and afterwards we perform a self-join on the produced hash codes.

4.3 Performance Analysis

4.3.1 Zipf Distribution. Figure 4 and 5 demonstrates the performance of three algorithms on the generated Zipf data set, with respect to varying skewness and input size. For the JB algorithm, we present the performance of the best setting, namely JBx48-y24. The y parameter determines the number of groups that the processing units are organized into. SHC clearly outperforms both existing methods in terms of execution time.

4.3.2 TPC-H Benchmark. All TPC-H queries consists of *primary key-foreign key* joins, i.e., the join key is not skewed in one side of each join. Under this scenario, SHC will only assign a $1 \times p_{(b)}$ grid for each heavy value, which is simply broadcasting the primary key tuple to all allocations of the other stream. For JB, we observe that the more groups used, the less efficient the algorithm is. PHJ seems to yield the best performance. We stress that SHC algorithm is better on many-many joins, where skewness exists in both relations.

4.3.3 COREL data set. We perform self similarity join on this high dimensional data set utilizing a $(r, 10r, 0.9, 0.1)$-sensitive family of hash functions, i.e. points within r distance are hashed to the same bucket with at least 0.9 probability and points with more than $10r$ distance have only 0.1 probability to be hashed to the same bucket. This similarity join is a many-many join. Again, SHC algorithm outperforms the other two in terms of execution time. It is better especially when r is large, meaning a higher skewness.

5 CONCLUDING REMARKS

In this paper, we proposed the Streaming HyperCube algorithm that extends the static HyperCube algorithm to the streaming setting. The algorithm continuously maintains an approximation (up to a factor of 2) of the optimal HyperCube configuration, while inuring small state migration overhead. In the future, we plan to conduct a more thorough theoretical analysis and extend it to multi-way joins.

REFERENCES

[1] [n. d.]. Apache Flink Project. https://flink.apache.org/
[2] [n. d.]. Apache Spark Project. https://spark.apache.org/
[3] [n. d.]. Apache Storm Project. https://storm.apache.org/
[4] [n. d.]. The TPC-H benchmark. http://www.tpc.org/tpch/
[5] Foto Afrati, Manas Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D Ullman. 2014. GYM: A multiround join algorithm in mapreduce. *arXiv preprint arXiv:1410.4156* (2014).
[6] Foto N Afrati and Jeffrey D Ullman. 2010. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*. ACM, 99–110.
[7] Foto N Afrati and Jeffrey D Ullman. 2011. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering* 23, 9 (2011), 1282–1298.
[8] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in parallel query processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 212–223.
[9] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding frequent items in data streams. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1530–1541.
[10] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[11] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and adaptive online joins. *Proceedings of the VLDB Endowment* 7, 6 (2014), 441–452.
[12] Buğra Gedik, Rajesh R Bordawekar, and Philip S Yu. 2009. CellJoin: a parallel stream join operator for the cell processor. *The VLDB Journal—The International Journal on Very Large Data Bases* 18, 2 (2009), 501–519.
[13] Aristides Gionis et al. [n. d.]. Similarity search in high dimensions via hashing.
[14] Xiao Hu, Yufei Tao, and Ke Yi. 2017. Output-optimal parallel algorithms for similarity joins. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 79–90.
[15] Jaewoo Kang, Jeffrey F Naughton, and Stratis D Viglas. 2003. Evaluating window joins over unbounded streams. In *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, 341–352.
[16] Paraschos Koutris, Paul Beame, and Dan Suciu. 2016. Worst-case optimal algorithms for parallel query processing. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 48. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[17] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable distributed stream join processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 811–825.
[18] Pratanu Roy, Jens Teubner, and Rainer Gemulla. 2014. Low-latency handshake join. *Proceedings of the VLDB Endowment* 7, 9 (2014), 709–720.
[19] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, et al. 2017. The Myria Big Data Management and Analytics System and Cloud Services.. In *CIDR*.

Exploring Fairness of Ranking in Online Job Marketplaces

Shady Elbassuoni *1, Sihem Amer-Yahia #2, Ahmad Ghizzawi *1, Christine El Atie *1
*American University of Beirut, Lebanon, #CNRS, Univ. Grenoble Alpes, France,
1 se58,@aub.edu.lb, cee11@mail.aub.edu, ahg05@mail.aub.edu,
2 sihem.amer-yahia@univ-grenoble-alpes.fr

ABSTRACT

We study fairness of ranking in online job marketplaces. We focus on group fairness and aim to algorithmically explore how a scoring function, through which individuals are ranked for jobs, treats different demographic groups. Previous work on group-level fairness has focused on the case where groups are pre-defined or where they are defined using a single protected attribute (e.g., Caucasian vs Asian). In this paper, we argue for the need to examine fairness for groups of people defined with any combination of protected attributes. To do this, we formulate an optimization problem to find a partitioning of individuals on their protected attributes that exhibits the highest unfairness with respect to the scoring function. The scoring function yields one histogram of score distributions per partition and we rely on Earth Mover's Distance, a measure that is commonly used to compare histograms, to quantify unfairness. Since the number of ways to partition individuals is exponential in the number of their protected attribute values, we propose two heuristic algorithms to navigate the space of all possible partitionings to identify the one with the highest unfairness. We evaluate our algorithms using a simulation of a crowdsourcing platform and show that they can effectively quantify unfairness of various scoring functions.

INTRODUCTION AND POSITIONING

Online job marketplaces are gaining popularity as mediums to hire people to perform certain tasks. Examples include freelancing platforms such as Qapa and MisterTemp' in France, and TaskRabbit and Fiverr in the USA. On those platforms, workers can find temporary jobs in the physical world (e.g., looking for a plumber), or in the form of virtual "micro-gigs" such as "help with HTML, JavaScript, CSS, and JQuery". A person who needs to hire someone for a job can formulate a query and is shown a ranked list of people. The resulting ranking naturally poses the question of fairness. Algorithmic fairness has recently received great attention from the data mining, information retrieval and machine learning communities (See for instance [3, 5, 8, 10]). The most common definition of fairness was introduced in [1, 11] as *demographic parity*, that is the unfair treatment of a person based on *belonging to a certain group of people*. Groups are defined using protected attributes such as gender, age, ethnicity or location. We carry these definitions in our work and define unfairness in online marketplaces as the unequal treatment of people by a scoring function based on their protected attributes. This definition is inline with what is also commonly referred to as *group unfairness* [2].

Most previous work on group-level fairness have either assumed that groups are pre-defined [8] or that they are defined using a single protected attribute (e.g., Caucasian vs Asian) [4]. In this work, we consider groups of people defined with any combination of protected attributes (the so-called *subgroup fairness* [5]). The scoring function yields one histogram per demographic group as score distributions. We use the Earth Mover's Distance (EMD) [7], a measure that is commonly used to compare histograms, to quantify distances between groups. Our intuition is that if score distributions between groups are significantly different, the scoring function does not treat the individuals in these groups equally. For instance, consider two groups only, namely young males and females living in France. Unfairness can be computed as the distance between the score distributions of those two groups.

Since we do not want to focus only on pre-defined groups, we must exhaust all possible ways of partitioning individuals on their protected attributes to quantify unfairness. For example, a scoring function might treat both men and women equally but might be unfair towards older Asian Americans compared to younger White Americans. We define an optimization problem as finding a partitioning of the ranking space, i.e., individuals and their scores, that exhibits the highest average EMD between its partitions. Exhaustively enumerating all possible partitionings is exponential in the number of values of protected attributes. Therefore, we propose two heuristic algorithms, BALANCED that generates a balanced tree of partitions, and UNBALANCED that generates an unbalanced tree of partitions. At each step, our algorithms greedily split individuals on the worst attribute, i.e., the one that results in the partitioning with the highest EMD between score distributions. This local decision is akin to the one made in decision trees using gain functions [6]. The algorithms stop when there are no further attributes left to split on or when the current partitioning of individuals exhibits more unfairness than it would if its partitions were split further.

PROBLEM DEFINITION

To quantify unfairness in online job marketplaces, we model the problem as computing the highest average distance between the score distributions of all possible partitions of individuals. Unlike previous work where partitions were defined or known a priori (e.g., [4]), we explore the space of all possible groups defined by a combination of values of the individuals' protected attributes. The goal becomes finding an unfair partitioning of individuals under the scoring function. We cast this goal as an optimization problem as follows.

DEFINITION 1 (MOST UNFAIR PARTITIONING PROBLEM). *We are given a set of individuals W, where each individual is associated with a set of protected attributes $A = \{a_1, a_2, ..., a_n\}$ and observed attributes $B = \{b_1, b_2, \ldots, b_m\}$. The protected attributes are inherent properties of the individuals such as gender, age, ethnicity, origin, etc. The observed attributes represent the skills of individuals for jobs and could include, for instance, the reputation and writing skills of an individual. We are also given a scoring function $f : W \to [0, 1]$, which is defined using observed attributes*

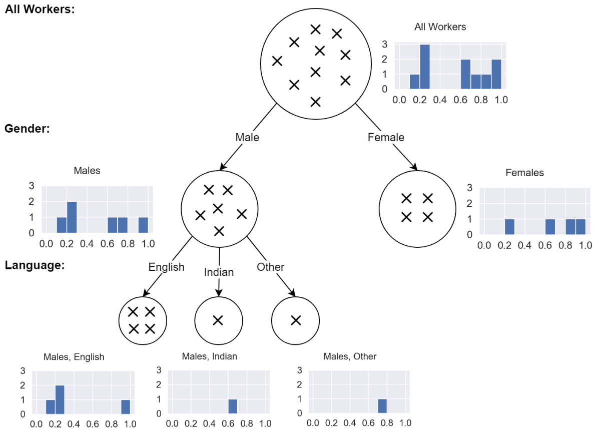


Figure 1: Optimum Partitioning of the Toy Example Data

as follows: $f(w) = \sum_{i=1}^{m} \alpha_i b_i$, *where* α_i *is a user-defined weight for observed attribute* b_i. *A weight of zero indicates that the corresponding attribute is not relevant for the user in ranking the individuals. Our goal is to* fully *partition the individuals in* W *into* k disjoint *partitions* $P = \{p_1, p_2, \ldots, p_k\}$ *based on their protected attributes in* A *using the following optimization objective:*

$$\begin{aligned} &\underset{P}{\text{argmax}} && \textit{unfairness}(P, f) \\ &\textit{subject to} && \forall i, j \; p_i \bigcap p_j = \phi \\ & && \bigcup_{i=1}^{k} p_i = W \end{aligned}$$

We now define how to compute the amount of unfairness of a function f for a partitioning P, or *unfairness*(P, f) in the above optimization problem.

Definition 2 (Average Pairwise Unfairness). *For a set of individuals* W, *a full-disjoint partitioning of the individuals* $P = \{p_1, p_2, \ldots, p_k\}$ *and a scoring function* f, *unfairness of* f *for the partitioning* P *is quantified as the average pairwise Earth Mover's Distance (EMD) between the distribution of scores in the different partitions of* P, *which is computed as follows:*

$$\textit{unfairness}(P, f) = \underset{i,j}{\text{avg}} \quad \textit{EMD}(h(p_i, f), h(p_j, f))$$

where $h(p_i, f)$ *is a histogram of the scores of individuals in* p_i *using* f.

Figure 1 shows a toy example of the optimum partitioning of 10 workers in a freelancing platform, which are ranked based on their qualification for some task using a scoring function f. The optimum partitioning is the one resulting from splitting the workers based on Gender first, and then splitting only the Male partition based on Language to get the following partitioning of workers: *Male - English, Male - Indian, Male - Other,* and *Female.* To arrive to this partitioning, one must exhaust all possible *full disjoint* partitionings of workers based on the protected attributes A and for each possible partitioning compute the average EMD between any two partitions. To do that, we generate a histogram for each partition as indicated in Figure 1 based on the function scores by creating equal bins over the range of f and counting the number of workers whose function values $f(w)$ fall in each bin.

Algorithm 1 Balanced (W: a set of individuals, f: a scoring function, A: a set of attributes)

```
$a = worstAttribute(W, f, A)$
$A = A - a$
$current = split(W, a)$
$currentAvg = averageEMD(current, f)$
while $A \neq \emptyset$ do
    $a = worstAttribute(current, f, A)$
    $A = A - a$
    $children = split(current, a)$
    $childrenAvg = averageEMD(children, f)$
    if $currentAvg \geq childrenAvg$ then
        break
    else
        $current = children$
        $currentAvg = childrenAvg$
    end if
end while
Add $current$ to $output$
```

Once the partitioning with highest average pairwise unfairness has been identified, it is up to the user, requester or platform developer, to decide on the right subsequent action.

ALGORITHMS

Our optimization problem for finding the most unfair partitioning is hard since there are is an exponential number of possible partitionings P. For this reason, we propose heuristics-based algorithms to identify a partitioning of individuals with respect to our optimization objective within reasonable time.

We first propose Balanced (Algorithm 1), an algorithm that generates a partitioning of the individuals in a greedy manner using the EMD of the partitions. Balanced is based on decision trees with EMD as utility [6]. It starts by splitting the individuals on the *worst* attribute with respect to EMD. This is done by trying out all possible attributes one at a time, and associating to each attribute-value partition, one histogram of the scores of all the individuals it contains. For each candidate attribute, Balanced computes the average pairwise EMD over histograms associated to the partitions obtained with the values of that attribute. It then returns the attribute with the highest average pairwise EMD and splits on that attribute. In the subsequent splitting steps, Balanced iteratively partitions the individuals using the other attributes in the same manner and only stops when the average pairwise EMD of the current partitioning is greater than that of the next candidate partitioning.

Balanced results in a partitioning of all the individuals using the same set of attributes (i.e., a balanced partitioning tree) since each splitting uses the same attribute over all current partitions. We also developed Unbalanced (Algorithm 2), another algorithm that partitions the individuals in a non-homogenous manner by locally deciding for each partition whether to further split it or not (i.e., resulting in an unbalanced partitioning tree).

Unbalanced is a *recursive* algorithm that decides to split a given partition by comparing the average EMD of that partition with its siblings to that of its children with its siblings. The intuition behind this is that it assesses what would happen to unfairness as measured by the average EMD if the partition was replaced by its children. It only splits a partition if its average pairwise EMD with its siblings is less than the average pairwise

Algorithm 2 UNBALANCED (*current*: a partition, *siblings*: a set of partitions, f: a scoring function, A: a set of attributes)

```
if $A = \emptyset$ then
    Add current to output
else
    $currentAvg = averageEMD(current, siblings, f)$
    $a = worstAttribute(current, f, A)$
    $A = A - a$
    $children = split(current, a)$
    $childrenAvg = averageEMD(children, siblings, f)$
    if $currentAvg \geq childrenAvg$ then
        Add current to output
    else
        for each partition $p \in children$ do
            UNBALANCED $(\{p\}, children - \{p\}, f, A)$
        end for
    end if
end if
```

EMD of its potential children with the partition's siblings. To invoke the algorithm, we first split the given set of individuals using the worst attribute as in the case of BALANCED and then the algorithm UNBALANCED is called once for each resulting partition. After all recursive calls of the algorithm terminate, the output is returned as the final partitioning of the individuals.

EVALUATION

To evaluate the effectiveness of our approach in quantifying unfairness, we run a simulation of a crowdsourcing platform using two sets of *active* workers and various scoring functions that rank those workers based on their qualification for tasks.

Setting. We generate two sets of active workers $\mathcal{W}$ of different sizes: 500 and 7300 (the estimated number of Amazon Mechanical Turk workers who are active at any time [9]). Each w in $\mathcal{W}$ has 6 protected attributes: Gender = {Male, Female}, Country = {America, India, Other}, Year of Birth = [1950, 2009], Language = {English, Indian, Other}, Ethnicity = {White, African-American , Indian, Other}, and Years of Experience = [0,30], and two observed attributes: LanguageTest = [25,100] and ApprovalRate = [25,100].

The values of those attributes are populated randomly so as to avoid injecting any bias in the data ourselves. Moreover, we define 5 different task qualification functions of the form $f = \alpha b_1 + (1 - \alpha)b_2$, where b_1 = Language Test and b_2 = Approval Rate and $\alpha \in \{0, 0.3, 0.5, 0.7, 1\}$.

We compare our two proposed algorithms UNBALANCED and BALANCED to a set of baselines. The first two baselines, which we refer to as R-BALANCED and R-UNBALANCED, are copies of our two algorithms BALANCED and UNBALANCED that use a random attribute instead of the worst attribute to split the workers at each step. The third baseline, which we refer to as ALL-ATTRIBUTES, is an algorithm that splits the workers based on *all* their protected attributes resulting in a full partitioning. Note that we also implemented an exhaustive algorithm that solves our optimization problem exactly by generating all possible partitionings in a brute-force manner and then returning the one with the highest average EMD. However, this algorithm failed to terminate after running for two days with only 6 attributes as in our simulation, even when each attribute had only a maximum of 5 values.

Simulation Results. Our first observation from Tables 1 and 2 is that for both datasets, functions f_4 and f_5 exhibit the highest unfairness as measured by the average pairwise EMD for all the partitionings retrieved by all the algorithms. Recall that these two functions are the ones that rely on one observed attribute only (LanguageTest in case of f_4 and ApprovalRate in case of f_5). *This indicates that if the scoring function uses fewer observed attributes, the chance of unfairness increases. In our simulation, since the attribute values were generated at random, there is a higher chance that the function scores correlate with a single protected attribute.*

Second, we observe that our two algorithms UNBALANCED and BALANCED consistently outperform or do as good as all other baselines for all datasets and functions. For the case of 500 workers, the UNBALANCED outperforms all other algorithms for the last three functions f_3, f_4 and f_5. On the contrary, the BALANCED returns the partitioning with the highest average EMD in the case of f_1. In the case of f_2, both BALANCED and R-BALANCED return the highest average EMD over the partitionings they find. *This allows us to validate the stopping condition used in our algorithms.*

Finally, in the case of 7300 workers, all the algorithms behave similarly, with the BALANCED and ALL-ATTRIBUTES returning the partitionings with slightly higher average EMD compared to all other algorithms. Upon investigating the returned partitioning by the different algorithms, we observed that in most cases all the algorithms returned the full partitioning tree, i.e., using all protected attributes, which is the same as the partitioning returned by the ALL-ATTRIBUTES algorithm. *We conjecture that it is due to the random values of all attributes.*

In terms of efficiency, the BALANCED algorithm took the most time to terminate compared to all other algorithms. In addition, the larger the dataset, the more time it took for all algorithms to finish. This is very intuitive given that the larger the dataset, the larger the individual histograms and the more time it takes to compute the pairwise EMD between them. Moreover, the deeper the partitioning tree, the larger the number of histograms that need to be compared. Finally, our two algorithms, BALANCED and UNBALANCED incur additional time since at each splitting step, they need to examine all remaining attributes to determine the worst one (i.e., the one which might result in the highest average EMD). All these factors contributed to the increased time to execute the BALANCED compared to all others. It is worth noting nonetheless that BALANCED terminates in less than 1.6 hours in the worst case (for the case of 7300 workers).

Qualitative Results. In addition to our simulation where we used a set of *random* task qualification functions, we also ran our algorithms on the following set of carefully-constructed functions, which are *unfair* by design:

- f_6: this function discriminates against females by setting the task qualification of workers as follows: $f_6(w) > 0.8$ if w is male and $f_6(w) < 0.2$ if w is female.
- f_7: this function sets the qualification of workers in a biased manner based on their gender and nationality as follows: $f_7(w) > 0.8$ if w is male and American, $f_7(w) < 0.2$ if w is female and American, $0.5 < f_7(w) < 0.7$ if w is Indian, either male or female, $f_7(w) > 0.8$ if w is female with any other nationality, and $f_7(w) < 0.2$ if w is male with any other nationality.
- f_8 designed as follows: $f_8(w) > 0.8$ if w is female and American, $0.5 < f_8(w) < 0.8$ if w is female and Indian and $f_8(w) < 0.2$ if w is female with another nationality.

Table 1: Average EMD and runtime for 500 workers and random functions

Algorithm	Average EMD					time (in secs)				
	f_1	f_2	f_3	f_4	f_5	f_1	f_2	f_3	f_4	f_5
UNBALANCED	0.195	0.191	**0.179**	**0.247**	**0.257**	20.987	23.715	22.823	29.504	28.845
R-UNBALANCED	0.193	0.193	0.177	0.243	0.253	28.33	26.871	28.354	27.333	28.372
BALANCED	**0.196**	**0.194**	0.177	0.246	0.253	**311.17**	**323.16**	**326.68**	**330.61**	**327.22**
R-BALANCED	0.195	**0.194**	0.177	0.246	0.253	131.87	122.49	119.97	127.06	124.46
ALL-ATTRIBUTES	0.195	0.193	0.177	0.246	0.253	42.708	42.494	42.597	42.235	42.337

Table 2: Average EMD and runtime for 7300 workers and random functions

Algorithm	Average EMD					time (in secs)				
	f_1	f_2	f_3	f_4	f_5	f_1	f_2	f_3	f_4	f_5
UNBALANCED	0.161	0.162	**0.151**	0.208	0.209	1169.224	1246.651	1205.963	1292.506	1245.037
R-UNBALANCED	0.162	**0.163**	**0.151**	0.208	0.209	1401.36	1391.541	1358.795	1290.977	1397.894
BALANCED	**0.163**	**0.163**	**0.151**	**0.210**	**0.211**	**5733.528**	**5745.611**	**5693.681**	**5840.131**	**5808.715**
R-BALANCED	**0.163**	**0.163**	0.122	**0.210**	**0.211**	3174.327	3240.727	2358.744	3115.123	3120.553
ALL-ATTRIBUTES	**0.163**	**0.163**	**0.151**	**0.210**	**0.211**	1453.626	1449.466	1450.712	469.839	1467.606

Table 3: Average EMD for 7300 workers & biased functions

Algorithm	Average EMD			
	f_6	f_7	f_8	f_9
UNBALANCED	0.040	0.164	**0.460**	0.317
R-UNBALANCED	0.399	0.362	0.322	0.350
BALANCED	**0.800**	**0.427**	**0.460**	**0.359**
R-BALANCED	0.496	0.368	0.330	0.301
ALL-ATTRIBUTES	0.420	0.368	0.337	**0.359**

- f_9 correlates with protected attributes ethnicity, language and year of birth similarly to previous ones.

As can be seen from Table 3, BALANCED retrieves the partitionings with the highest possible average EMD compared to all other algorithms. In addition, the resulting partitionings are the ones expected, i.e., using the attributes for which the functions were designed to correlate with. For example, for f_6, BALANCED partitions the workers on only gender for all datasets. Similarly, for f_7, it partitions the workers on both gender and country. We show only the results in the case of 7300 workers due to space limitation. Finally, we observe that *overall for all functions and algorithms, the average EMD is much higher compared to the functions used in our simulation experiment, which indicates that our optimization problem is indeed effective in capturing unfairness of the scoring functions as conjectured.* The only exception was for UNBALANCED in the case of f_6 and f_7, where the algorithm ended up splitting the workers further than it should because of the local nature of its stopping condition. In fact, since the function scores were generated at random within the specified range, various runs of the experiments resulted in different behavior, where in some cases, UNBALANCED performed as well as BALANCED.

SUMMARY AND FUTURE WORK

We set out to examine fairness of ranking in online job marketplaces. To do this, we defined an optimization problem to find a partitioning of the individuals based on their protected attributes that exhibits the highest unfairness by a given scoring function. We used Earth Mover's Distance between score distributions as a measure of unfairness. Unlike previous work, we did not assume a pre-defined partitioning of individuals and instead proposed two heuristic algorithms, BALANCED and UNBALANCED, that efficiently partition the individuals without exploring the full space of partitionings. Our immediate plan is to test our algorithms on real datasets from Qapa and TaskRabbit. We are also investigating other formulations and metrics for fairness instead of the Earth Mover's Distance. We are also studying ways of "repairing" bias in the context of ranking in online job marketplaces.

ACKNOWLEDGMENTS

This work is supported by the American University of Beirut Research Board (URB)

REFERENCES

[1] Toon Calders and Sicco Verwer. 2010. Three naive Bayes approaches for discrimination-free classification. *Data Mining and Knowledge Discovery* 21, 2 (01 Sep 2010), 277–292.

[2] Sorelle A. Friedler, Carlos Scheidegger, and Suresh Venkatasubramanian. 2016. On the (im)possibility of fairness. *CoRR* abs/1609.07236 (2016).

[3] Sara Hajian, Francesco Bonchi, and Carlos Castillo. 2016. Algorithmic Bias: From Discrimination Discovery to Fairness-aware Data Mining. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016.* 2125–2126.

[4] Aniko Hannak, Claudia Wagner, David Garcia, Alan Mislove, Markus Strohmaier, and Christo Wilson. 2017. Bias in Online Freelance Marketplaces: Evidence from TaskRabbit and Fiverr. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing, CSCW 2017, Portland, OR, USA, February 25 - March 1, 2017.* 1914–1933.

[5] Michael J. Kearns, Seth Neel, Aaron Roth, and Zhiwei Steven Wu. 2018. Preventing Fairness Gerrymandering: Auditing and Learning for Subgroup Fairness. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018.* 2569–2577.

[6] Sreerama K Murthy. 1998. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data mining and knowledge discovery* 2, 4 (1998), 345–389.

[7] Ofir Pele and Michael Werman. 2009. Fast and robust earth mover's distances. In *2009 IEEE 12th International Conference on Computer Vision.* IEEE, 460–467.

[8] Ashudeep Singh and Thorsten Joachims. 2018. Fairness of Exposure in Rankings. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018.* 2219–2228.

[9] Neil Stewart, Christoph Ungemach, Adam JL Harris, Daniel M Bartels, Ben R Newell, Gabriele Paolacci, Jesse Chandler, et al. 2015. The average laboratory samples a population of 7,300 Amazon Mechanical Turk workers. *Judgment and Decision making* 10, 5 (2015), 479–491.

[10] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez-Rodriguez, and Krishna P. Gummadi. 2017. Fairness Constraints: Mechanisms for Fair Classification. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA.* 962–970.

[11] Indre Zliobaite. 2015. A survey on measuring indirect discrimination in machine learning. *CoRR* abs/1511.00148 (2015).

Snapshot Isolation for Transactional Stream Processing

Philipp Götze
Technische Universität Ilmenau
Ilmenau, Germany
philipp.goetze@tu-ilmenau.de

Kai-Uwe Sattler
Technische Universität Ilmenau
Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

Transactional database systems and data stream management systems have been thoroughly investigated over the past decades. While both systems follow completely different data processing models, the combined concept of transactional stream processing promises to be the future data processing model. So far, however, it has not been investigated how well-known concepts found in DBMS or DSMS regarding multi-user support can be transferred to this model or how they need to be redesigned. In this paper, we propose a transaction model combining streaming and stored data as well as continuous and ad-hoc queries. Based on this, we present appropriate protocols for concurrency control of such queries guaranteeing snapshot isolation as well as for consistency of transactions comprising several shared states. In our evaluation, we show that our protocols represent a resilient and scalable solution meeting all requirements for such a model.

1 INTRODUCTION

Emerging application domains such as cyber-physical systems, IoT, and healthcare have fostered the convergence of stream and batch processing in data management. This can be observed not only by market trends such as real-time warehousing but also by architectural patterns like the lambda architecture aiming at combining batch and online processing on Big Data platforms.

This convergence means that the input to the system is treated as a continuous stream of data elements whose processing is implemented as a stream processing pipeline (query) with one or more persistent states/tables as sinks. Updates on these tables trigger further processing implemented again as stream processing pipeline. In addition, tables can be also queried in an ad-hoc fashion, e. g., for snapshot reports. Hence, the query part of an application is a mix of stream and ad-hoc queries while the data objects are streams and tables. However, concurrency and the need for providing fault tolerance require transaction support: at least stream queries writing to or reading from a table should run within a transaction context with ACID guarantees. This is necessary for recovery in case of failures and to provide a consistent view on (persistent) subsets of the stream (such as a window). Here, we refer to this processing style as *transactional stream processing* [2, 13] meaning that (a) a stream query writing to tables represents a sequence of transactions and (b) stream or batch queries on such tables require transaction isolation. Supporting such *queryable states* raises several requirements:

① state representations (tables) have to be queryable at all,
② the isolation property for concurrently running stream queries updating the state and ad-hoc queries on these states has to be guaranteed, and
③ consistency among multiple states of the same stream query is required even in the case of transaction aborts.

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

Figure 1 sketches a possible smart metering scenario which could benefit from transactional stream processing. Here, it is getting data from private households and the global infrastructure which is checked against respective specifications. It consists of three continuous and one ad-hoc query accessing various (shared) states whose semantic we explain in more detail in Section 3.

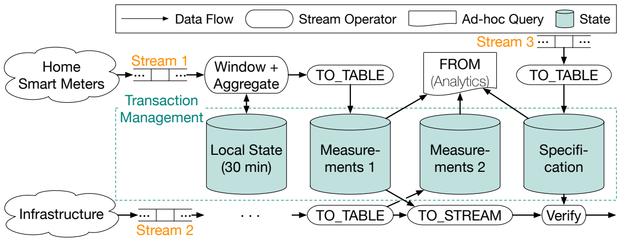


Figure 1: Possible smart metering use case.

In this paper, we present techniques to address the requirements above. Based on the work introduced in [2, 13], we propose a data-centric transaction model for data streams, discuss why and how to realize snapshot isolation on queryable states together with a consistency protocol. With the help of a micro benchmark, we show the suitability and scalability of our approach.

2 RELATED WORK

The first major investigation of a possible combination of relational and stream processing took place in the STREAM project [14]. However, when designing the system the team focused only on continuous queries instead of a hybrid query set of continuous and single point in time queries. To model the transformations of streams and relations, STREAM provides operations based on the whole relation (*RStream*), inserts (*IStream*), or deletes (*DStream*).

A more recent development of a system supporting transactions and data streaming is S-Store [13] which is built upon the main memory OLTP system H-Store [9]. S-Store inherits the ACID properties of H-Store by implementing data streaming concepts like windows and streams as time-varying H-Store tables with a timestamp as ordering property. Continuously running stream queries are implemented as stored procedures. Together with an input batch, which is the unit of a stream with the same timestamp, the execution of the stored procedure over such a batch forms a transaction over stream data.

Transactional Stream Processing [2] is a different notion to combine streaming and pure transaction processing. Their approach defines a unified transaction model which assigns a timestamp to each transaction and transforms continuous queries into a series of one-time queries, where each one-time query reads all required data before writing the result set. Transactional Stream Processing is based on a storage manager which transfers data processing into a producer-store-consumer scenario and identifies different properties to provide the proper storage for a producer-consumer combination.

In the context of scalable data stream processing platforms, Apache Flink Streaming is one of the most advanced approaches to consistent states and fault tolerance [3]. Their mechanism is based on distributed snapshots where stream barriers are moving along with the data through the dataflow. At operator level, the flow is aligned until all barriers with the same ID arrive and subsequently the current state is persisted in the state backend (e. g., HDFS). Whenever a failure occurs, the last complete snapshot is used for restoring a consistent state per streaming pipeline.

In [15] the authors present a platform combining OLTP, OLAP, and stream processing in a single system. For that, they merge Apache Spark with an in-memory transactional store and enhance it by additional features such as approximate processing.

MVCC is one of the most widely used concurrency control protocols and has been implemented in, e. g., Postgres [18], HyPer [10], Hekaton [5], and SAP HANA [12]. No precise standard exists that describes how MVCC should be implemented. There are several possibilities and adjustments that strongly depend on the expected workload. In [20] the authors present an extensive study of key design decisions when implementing MVCC in an in-memory DBMS. In particular, these are concurrency control protocol, version storage, garbage collection, and index management. We have used the findings of this study to design our own MVCC approach (see Section 4).

3 TRANSACTION MODEL

Addressing the scenario and requirements described in Section 1 requires to distinguish two basic concepts: *tables* for representing states and *streams*. Tables represent a finite collection of data structured in rows and columns, as known from relational DBMS. Streams define a potentially infinite sequence of tuples of data, where tuples carry an implicit or explicit ordering. While a table requires a physical storage representation, streams are volatile.

Linking operators. As already proposed in the query language for STREAM [1], two classes of query operators are required to link these objects: stream-to-table and table-to-stream. Here, we call them TO_TABLE and TO_STREAM.

- TO_TABLE inserts, deletes, or updates tuples from a stream in a table, e. g., to update an operator state, and
- TO_STREAM produces a stream of tuples from a table.

Whether a stream tuple is inserted or updated in a table depends on the presence of a table tuple with the same key. A delete occurs if the tuple is outdated (e. g., from a window) or explicitly removed by a *delete tuple*. TO_STREAM unifies the two cases, where stream tuples are just incrementally processed and, on the other hand, table wide operations are executed before new tuples can be emitted. Whenever a certain condition on a table is fulfilled, TO_STREAM is executed and emits a new (set of) tuple(s) to a stream. In addition to these operators, a standard ad-hoc query operator FROM is required to either attach to a stream, i. e., read all tuples of the stream starting at the point of attachment, or to read data of a table. Figure 2 illustrates these operators.

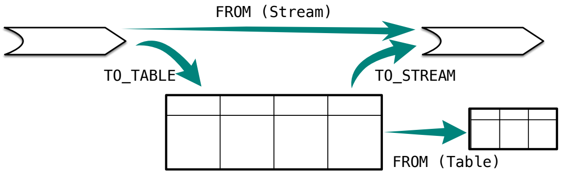


Figure 2: Linking operators.

Transaction boundaries. A first question is how to define transaction boundaries for data streams. Apart from the rather trivial case where each stream element represents its own transaction (aka "auto-commit"), two basic approaches can be distinguished. In the *data-centric* approach, transaction boundaries (BOT, COMMIT, ROLLBACK) are marked by dedicated stream elements, whereas the other stream elements are interpreted as insert or update operations. *Punctuations* [19] or *control tuples* are useful concepts for this purpose. This way each transaction can be defined over a consecutive number of stream tuples. Thereby, a transaction span can range from the length of an entire stream to the length of a sub-stream or each tuple is considered a single transaction. An alternative strategy is the traditional *query-centric* approach meaning that transaction boundaries are specified as part of the query or dataflow program. Obviously, this approach is better suited for ad-hoc and not for stream queries.

Unified tables for queryable states. For our system model, we rely on a unified table model taken from the DBMS world to cover all relevant aspects needed for transactional stream processing. However, stateful stream operators such as windows or aggregates also require structures for maintaining operator related data. Such operators exploit tables as internal structures to publish their state as a table allowing to share their content with other queries or to query the content. Besides sharing operator states, this approach also provides the advantage of re-using persistence and recovery mechanisms.

Transactional semantics. Inserts, deletes, or updates on states/tables need to run in a transactional context to guarantee atomicity for writes to a table and isolation for reads. The only way to modify a table in our model is provided by using the TO_TABLE operator which has to guarantee atomicity based on the transaction boundaries. Because a single stream query might contain multiple operators maintaining a persistent state, consistency among multiple states is a further requirement. This means that all states updated within the same stream query should be updated atomically with each commit, i. e., another query reading these states should see updates from the same (most recent) committed transaction. For reads of the FROM operator, we have to consider isolation properties. This also applies if FROM provides access to a data stream: here different isolation levels should provide different levels of visibility. For reads in TO_STREAM, we have to consider the trigger policy in addition to the isolation property. Trigger policy means the condition when a stream element is produced (TO_STREAM) or modifications in form of one or more transactions are generated into a back-to-the-table-directed stream. Here, possible policies are to consider each tuple modification or to rely on transaction commits.

4 TRANSACTIONAL STATE MANAGEMENT

From the transactional semantics described in Section 3, the following requirements can be derived that correspond to the ACID principle applied to the transactional stream processing model. First, there is atomicity, which occurs in several aspects. As mentioned before, the scope of a transaction that should be executed atomically must be marked via certain transaction boundaries. In addition, the individual operations must also be performed atomically in order to ensure both fault tolerance and consistency in case of concurrent access. This leads directly to the next two requirements: persistence and isolation. The latter property must ensure that both continuous and ad-hoc queries do not influence each other in terms of correctness and consistency. Furthermore, consistency must be ensured even if a query accesses multiple states, also in the event of transaction aborts. The persistence requirement means that the results of successfully committed transactions are still available after a system restart or crash. This goes hand in hand with recoverability, which must ensure that the states are brought back or always stay in a consistent form. Taking these requirements into consideration, we have realized a snapshot isolation approach comprising three components:

- multi-versioned data structures for queryable states,
- a transaction protocol to access these states, and
- a protocol guaranteeing consistency among multiple states.

These components are prototypically implemented as part of our data stream processing framework PipeFabric[1]. We opted for an MVCC approach as it has proven to be the most scalable and widely used concurrency control protocol in the literature for DBMSs [4, 16, 20]. We expect it to behave similarly in a transactional stream processing environment. However, this still has to be revealed, which we will pursue in the following sections.

4.1 Data Structures

As transactional state representation, we have designed a table wrapper as shown in Figure 3. For the base table, any existing backend structure with a key-value mapping can be used. Therefore, every state type can use a suitable underlying structure making our design extremely versatile. Here, each key is mapped to an MVCC object. An entry in this object corresponds to the typical structure for MVCC [17, 20] $(\equiv\ <[cts, dts], value>)$. The commit and deletion timestamp (CTS/DTS) indicate the lifetime of the value version. With the help of a bit vector (UsedSlots) the available free version slots are managed.

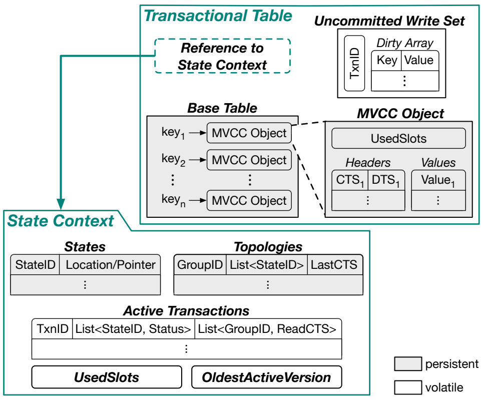


Figure 3: Transaction components.

Before new versions become visible to readers the changes are transiently stored as Uncommitted Write Set. This enables simple and fast aborts and also prevents the mixing of committed and uncommitted versions. Furthermore, the transactional table has a reference to the global context which contains all necessary runtime information. It consists of metadata about the states, topologies, and active transactions. For the states it contains general information such as their ID and physical location (e. g., file system path). In PipeFabric a query is written by defining a so-called *Topology*. It can be seen as graph where each node is an operator and the edges represent their subscribed streams. Each state is part of at least one topology for which we track the states that must be written together atomically. This is necessary for the correct application of the consistency protocol (see Section 4.3) for which the last committed transaction (LastCTS) per group is recorded. For recovery purposes, this information needs to be persistent. At the beginning of each transaction, it is assigned a unique timestamp (TxnID). All timestamps are logical and generated by a global atomic counter. For currently running transactions, we save a list of accessed states containing their ID and status (Active, Abort, or Commit) as well as the global commit timestamp at the time of reading for each topology. Again, we use a bit vector[2] to atomically manage the available slots. For garbage collection, we clean up old versions on demand (using OldestActiveVersion), i. e., if a new version has to be created and no space is available in the version array. The state context of the transaction management is completely latch-free and solely uses atomic instructions.

4.2 Concurrency Protocol

Before considering global consistency, we look at the basic operations required, namely *read*, *write*, *commit*, and *abort*. For a better separation of the protocols, we first assume that only a single state needs to be accessed consistently at a time. We further assume a beginning punctuation to signal the start of a transaction that assigns a timestamp and registers it in the context. To synchronize the actual access of MVCC blocks a lightweight locking strategy with read-write locks (latches) can be used.

The *read* operation starts by checking whether the accessing transaction has already written a new value (Uncommitted Write Set) and returns it. Otherwise, the latest visible version will be looked up using the commit and deletion timestamps. To achieve snapshot isolation, the first read version timestamp of this transaction must be transiently stored and used for subsequent reads (readCTS). The found value is finally returned to the caller.

When a transaction wants to *write* a new value, it is merely appended to its write set (Dirty Array). In case of a single writer, no exclusive locks are required and writes never block. For multiple writers, it could be checked if write sets overlap and then prematurely abort/restart the later transaction. Alternatively, this could be done only at commit time to prevent slower writes.

As all uncommitted changes are stored transiently together, it is enough for the *abort* operation to simply clear the corresponding write set and release the memory. The *commit* operation, on the other hand, is a bit more complex since all changes must be applied atomically and isolated from other readers. For each modification the MVCC object is loaded, the position of the new value is determined, and the position of the current value is looked up. The changes are then applied in memory. If no free insert position could be found, the garbage collection is performed at this point. Subsequently, the changes are populated atomically and isolated into the base table. As a final step, the global commit timestamp of the table is changed to the committing transaction's ID. By atomically setting this timestamp, the protocol can guarantee that the changes of a transaction are either visible completely or not at all. In the case of multiple writers, additional write locks are introduced and the order of the commits must be checked. If the current version is greater than the timestamp of the transaction, it must abort (First-Committer-Wins rule).

The way we designed our operations eliminates the need for undo operations within the actual table. In addition, read operations are generally not blocked by write operations and vice versa. Only during the commit time, a short synchronization is required. Therefore, we expect the performance to remain stable even for high contention situations.

4.3 Consistency Protocol

If now a continuous query needs to update multiple states, they must become visible together to maintain consistency. Assume a simplified case where a data stream query writes two states and

[1] PipeFabric: https://github.com/dbis-ilm/pipefabric

[2] In fact, it is a 64-bit integer, which is updated by CAS operations.

a second (ad-hoc) query reads from both states. We coordinate the operators with the help of the state context (see Figure 3). Whenever a commit arrives the corresponding status flag for this transaction and state is set to `Commit`. The modifications are not persisted until all states registered for this transaction are ready for commit. The operator that sets the last status flag to `Commit` becomes the coordinator and is responsible for the global commit. In addition, a transaction must be aborted globally as soon as `Abort` has been flagged for at least one state. In this respect, this is a modified version of the 2-Phase-Commit protocol [11] relying on proven concepts which adds almost no overhead in our case.

Readers can see the last completed transaction using `LastCTS` for each topology which is set at the end of a commit (instead of the transaction ID as described before). They must also check in the context which states are written together. For these, the version must be the same, otherwise, a commit has been executed in the meantime. Therefore, the read version is noted within the context (`ReadCTS`) and is only set at the first read per topology. Thus, every operation reads from the same snapshot and interleaved commits do not pose a problem. If there is an overlap when reading multiple topologies with different versions (`LastCTS`), the older version must be read to guarantee consistency.

5 EVALUATION

In this section, we present a micro benchmark to substantiate the suitability and scalability of our approach. For this, we evaluate our MVCC protocol against a simple strict two-phase locking (S2PL) [6] and a backward-oriented optimistic concurrency control (BOCC) [8] protocol. We have made every effort to implement optimized versions of each protocol. All concurrency control protocols use fundamentally the same consistency protocol for multiple states as described in Section 4.3.

5.1 Setup and Workloads

The experiments were run on a 2-socket Intel Xeon E5-2630 each 6 cores à 2 threads, 128 GB DDR3, Linux kernel 4.15, and GCC 7.3. As a base table, we use a persistent key-value store, namely RocksDB[3]. It has a log-structured merge-tree (LSM) design and provides many configuration options for a wide range of requirements. We kept the default configuration and only set the *sync* option to true to guarantee failure atomicity. As a benchmark, we use a scenario having one stream continuously writing to two states and multiple ad-hoc queries reading from these states. Both are initialized with a table size of one million key-value pairs (4 Byte key, 20 Byte value). During the experiments, we vary the number of parallel ad-hoc queries and the contention rate using a Zipfian distribution ($\theta = 2.9 \mathrel{\hat{=}} 82\%$ the same key) [7].

5.2 Performance Study

In the following, we compare the performance of all concurrency control protocols for transactions of medium length (10 operations each). An excerpt of our measurements is shown in Figure 4. Due to the synchronous writing, the readers (mostly only accessing memory) contribute almost exclusively to the total throughput. While the other two protocols are dropping, the MVCC protocol provides consistently a good performance. Interestingly, the BOCC protocol is slightly faster (~5%) than MVCC with little contention and many concurrent ad-hoc queries. However, this is logical as it is designed for scenarios with few conflicts. If the number of threads and the contention increases, it brings the S2PL and BOCC to their knees relatively quickly. It is noticeable that at least for MVCC caching effects are visible with a higher contention. Overall, it shows that our MVCC approach combined with the consistency protocol is highly scalable and resilient, making it well suited for transactional state management.

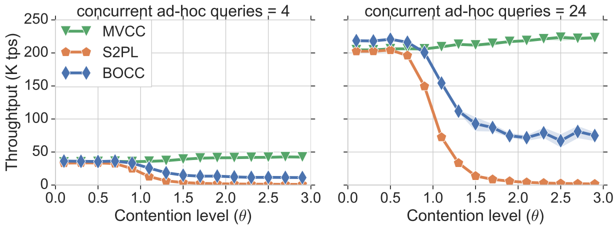


Figure 4: Contention and scalability check with persistent synchronous writes and medium-sized transactions.

6 CONCLUSION

Transactional stream processing systems itself offer a wide range of research opportunities. In this paper, we presented a data-centric transaction model and investigated concurrency and consistency aspects within this model. We have found that our snapshot isolation approach meets all our initial requirements. We designed a versatile state representation which is queryable by both continuous and ad-hoc queries. Even under high parallelism and contention, the ACID properties could always be maintained with great performance even when involving multiple states.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (DFG) within the SPP2037 under grant no. SA 782/28.

REFERENCES

[1] A. Arasu et al. 2003. CQL: A Language for Continuous Queries over Streams and Relations. In *DBPL*. 1–19.
[2] I. Botan et al. 2012. Transactional Stream Processing. In *EDBT*. 204–215.
[3] P. Carbone et al. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *PVLDB* 10, 12, 1718–1729.
[4] M. J. Carey and W. A. Muhanna. 1986. The Performance of Multiversion Concurrency Control Algorithms. *TOCS* 4, 4, 338–378.
[5] C. Diaconu et al. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*. 1243–1254.
[6] K. P. Eswaran et al. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11, 624–633.
[7] J. Gray et al. 1994. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD*. 243–252.
[8] T. Härder. 1984. Observations on Optimistic Concurrency Control Schemes. *Inf. Syst.* 9, 2, 111–120.
[9] R. Kallman et al. 2008. H-Store: A High-performance, Distributed Main Memory Transaction Processing System. *VLDB* 1, 2, 1496–1499.
[10] A. Kemper and T. Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*. 195–206.
[11] B. Lampson and H. E Sturgis. 1979. Crash Recovery in a Distributed Data Storage System. In *XEROX Research Report*.
[12] J. Lee et al. 2013. High-Performance Transaction Processing in SAP HANA. *IEEE Data Eng. Bull.* 36, 2, 28–33.
[13] J. Meehan et al. 2015. S-Store: Streaming Meets Transaction Processing. *PVLDB* 8, 13, 2134–2145.
[14] R. Motwani et al. 2003. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *CIDR*.
[15] B. Mozafari et al. 2017. SnappyData: A Unified Cluster for Streaming, Transactions and Interactice Analytics. In *CIDR*.
[16] A. Pavlo and M. Aslett. 2016. What's Really New with NewSQL? *SIGMOD Record* 45, 2, 45–55.
[17] D. P. Reed. 1983. Implementing Atomic Actions on Decentralized Data. *TOCS* 1, 1, 3–23.
[18] M. Stonebraker and L. A. Rowe. 1986. The Design of Postgres. In *SIGMOD*. 340–355.
[19] P. A. Tucker et al. 2003. Exploiting Punctuation Semantics in Continuous Data Streams. *TKDE* 15, 3, 555–568.
[20] Y. Wu et al. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *PVLDB* 10, 7, 781–792.

[3] RocksDB (version 5.15.10): https://github.com/facebook/rocksdb

Recurrent Neural Networks for Dynamic User Intent Prediction in Human-Database Interaction

Vamsi Meduri
Arizona State University
vmeduri@asu.edu

Kanchan Chowdhury
Arizona State University
kchowdh1@asu.edu

Mohamed Sarwat
Arizona State University
msarwat@asu.edu

ABSTRACT

Prediction of human intent during a user interaction session with the database received a significant amount of attention in the recent past [1, 8]. State-of-the-art intent detection approaches such as [5] insist that the human intent is dynamic and is constantly changing throughout the user session. While the usage of classifiers like SVMs and decision trees have been proposed to capture static user intent [2], such models become ineffective in predicting dynamic or ever-changing human intent. Recurrent Neural Networks (RNNs) are powerful temporal predictors and have recently been prominent in the database research community for tasks such as entity matching [3, 6]. In this work, we discuss the application of RNNs to the problem of dynamic user intent prediction during a human-database interaction. We propose two variants of SQL-specific embedding vectors for RNNs. We also propose *active learning* strategies for RNNs which consume a fraction of the held-out training data to produce competitive prediction quality as full training or *supervised learning*. Our experiments on real user sessions upon the NYCTaxiTrip dataset [9] evaluate the effectiveness of vanilla, LSTM and GRU based RNNs.

1 INTRODUCTION

Prediction of user intent during a Human-Database Interaction (HDI) session helps prefetching the results of the anticipated queries [4] thus making the interaction seamless. Existing works such as [1, 2] define the user intent as the last (target) query asked by a user in an interaction session and the prediction of user intent as a binary classification problem. The test data at hand is classified as interesting or not by using a decision tree or a Support Vector Machine (SVM) based on the training the classifier has undergone. The positive class predictions are evaluated against the results of the target SQL query held as ground truth. This line of work assumes that the user intent is static, contrary to which [5] emphasizes that the user can refine and adapt her needs constantly until the termination of an HDI session i.e., the target query changes continuously. Capturing dynamic multi-user intent using [1, 2] requires a binary classifier for each user session. Instead, we can model dynamic intent discovery as a temporal prediction task using a single Recurrent Neural Network (RNN) for all the user sessions because of its ability to train on and predict several sequences of data.

For a query qu_i issued by the user at timestep i, we not only retrieve the results of qu_i but also predict the intent vector of the subsequent query at timestep $i + 1$. We use RNN as the intent prediction middleware between the user and the database as illustrated in Figure 1. A prior embedding layer can convert raw text either into real-valued embeddings or *one-hot* vectors or words in a *vocabulary* which are fed to the RNNs as intent vectors.

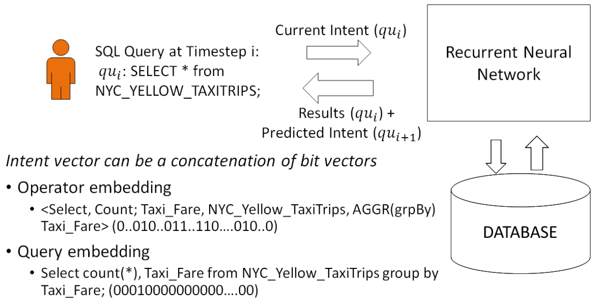

SQL Query at Timestep i:
qu_i: SELECT * from NYC_YELLOW_TAXITRIPS;
Current Intent (qu_i)
Results (qu_i) +
Predicted Intent (qu_{i+1})
Recurrent Neural Network
DATABASE
Intent vector can be a concatenation of bit vectors
• Operator embedding
• <Select, Count; Taxi_Fare, NYC_Yellow_TaxiTrips, AGGR(grpBy) Taxi_Fare> (0..010..011..110....010..0)
• Query embedding
• Select count(*), Taxi_Fare from NYC_Yellow_TaxiTrips group by Taxi_Fare; (00010000000000....00)

Figure 1: Intent Prediction Pipeline using RNNs

The performance of an RNN is dependent on the granularity at which these embeddings and the corresponding vocabulary are created. For instance, either an entire SQL query or each distinct substring or character in the query can be considered as a word in the vocabulary. In the latter case, external libraries such as word2vec (https://code.google.com/archive/p/word2vec/) or GloVe (https://nlp.stanford.edu/projects/glove/) can be used to borrow pre-trained real-valued embeddings [6].

We create two variants of SQL-specific binary embeddings for dynamic intent vector representation customized to relational databases rather than relying on external libraries which are SQL-agnostic. In the first variant of query embedding, we use a one-hot vector which is a bitmap of 1s and 0s to represent each query intent. It only has a single dimension set to 1 that corresponds to a specific query among the collective set of queries issued by the users until a given point of time. The second variant of operator-based embedding breaks down a SQL query into several smaller bitmaps each corresponding to a distinct family of SQL operators. A concatenation of all the operator-specific bitmaps produces an operator embedding as the user intent for that query.

RNNs are known to consume a lot of training data and time. To reduce the amount of training data, we use active learning on RNN to empirically test if it can make effective predictions with limited user sessions. Active learning selectively includes *ambiguous* (hard-to-classify) examples into the training data and incrementally refines the classifier based on the additional training data. Our approach differs from active learning for static intent in [2] as we select query sequences that is more scalable than labeling tuples. To minimize the training time and enhance interactivity, we propose using an incrementally trained RNN. We present our results on 139 sessions we collected from real user interactions on the NYCTaxiTrip [9] dataset. The remaining paper is organized as follows: we first present the details of the experimental dataset and the construction of the embedding vectors followed by the application of supervised learning and active learning on RNN for dynamic intent prediction. We conclude the paper with a discussion on the experimental results.

2 DATASET AND SYSTEM OVERVIEW

We collected 139 user sessions from 30 real world users with 4 to 5 average sessions per user. Each of them interacted with a sample of 100,000 trips loaded from a total of 10M taxi trips that span over 5 weeks in June and July 2016. We used the visual interface of Tableau [7] connected to the database engine of PostgreSQL 9.5.10. The reason for sampling is to increase the interactivity between the user and the database without overloading the Tableau interface. The visual interactions are translated into SQL queries and stored in logs by Tableau. We obtained a total of 1958 SQL queries with 1190 distinct queries from the user interactions. Our offline vocabulary creation step ensures that the embedding vectors of all the queries have the same pre-fixed dimensionality. We do not have to handle "UNK"(unknown) tokens as RNNs conventionally do for out-of-vocabulary strings, because of the offline step.

Query Embedding: Query-based intent uses the actual query for intent representation. Each query is thus assigned a specific dimension within the entire vocabulary space of 1190 distinct SQL queries in the dataset. A bitmap of 1190 bits is created for each query of which a single bit is set to 1 specific to the query.

Operator Embedding: The NYCTaxiTrips dataset is stored as a single table and hence each SQL query intent vector is a composite bit vector of all possible operators allowed over a single relation and can be written as $vec(qu_{Aggr})$ = π_{vec} $Aggr_{vec}$ σ_{vec} $GROUP\ BY_{vec}$ $ORDER\ BY_{vec}$ $HAVING_{vec}$ $LIMIT_{vec}$. Each query vector $vec(qu_{Aggr})$ is a concatenation of bit vectors for operators such as PROJECT, AGGREGATE, GROUP BY, ORDER BY, HAVING and LIMIT. Every single operator out of these six has a dimensionality equal to the number of attributes $|Attr|$ in the database schema (indicating the columns that the operator can be associated with) except AGGREGATE and LIMIT. $Aggr_{vec}$ is a concatenation of five most common aggregate operators - AVG, MIN, MAX, SUM and COUNT and thus has a dimensionality of $|Attr| * 5$ bits. LIMIT operator contributes only to a single bit in the operator vector as it is not associated with any attribute and is more like a Boolean recording its presence in the query. The total dimensionality of an operator embedding bitmap is thus $|Attr|$x10+1 which is 18x10+1=181 for the NYC taxi trip dataset.

Operator	**Operator sub-vector**	**#Dimensions**
π_{vec}	000000000000100000	#Columns = 18
$Aggr_{vec}$	0...0111111111111111111	#Columns x 5 = 90
σ_{vec}	0000..0	#Columns = 18
$GROUP\ BY_{vec}$	000000000000100000	#Columns = 18
$ORDER\ BY_{vec}$	0000..0	#Columns = 18
$HAVING_{vec}$	0000..0	#Columns = 18
$LIMIT_{vec}$	0	1

The above table lists the operator (sub-)vectors for each expected category of SQL operators for the example query qu_{i+1} from Figure 1, qu_{Aggr}: *SELECT COUNT(*), taxi_fare FROM nyc_yellow_tripdata GROUP BY taxi_fare*. Let taxi_fare be the 13^{th} attribute among the list of 18 possible attributes. We can see the corresponding bit position set to 1 in the 18-bit operator sub-vector for the projection and group by operator while setting all the attributes for the count operator to 1.

2.1 Supervised Learning

In this section, we describe how RNNs are trained and tested for dynamic intent prediction. We use Keras API (https://keras.io/) with TensorFlow library (https://www.tensorflow.org/) for RNNs. Figure 2 illustrates RNN for two consecutive timesteps T_i and T_{i+1} in intent prediction. We use a "many-to-one" sequential RNN with an input layer, a hidden layer and an output layer. The input layer is fed with the sequence of query/operator embedding vectors until the current timestep in the session to get a temporal prediction of the intent for the next timestep. The output layer emits a probability vector with same dimensionality as the intent vector. The value of each dimension indicates the probability of that dimension being a 1. For example, in Figure 2, the output vector $< 0.6, 0.4, 0.01, 0.01, 0.9 >$ at T_i denotes the probabilities of each of its five dimensions being 1.

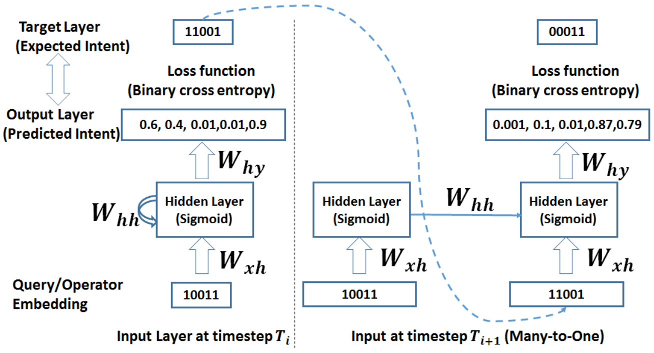


Figure 2: Recurrent Neural Network for Intent Prediction

During the training phase, the predicted probabilistic output vector is compared to the target query which is the successor query from the next timestep or the expected intent vector output (of 11001 at T_i in the figure) using a loss function. The hidden layer weights (w_{xh}, w_{hh}, w_{hy}) are updated using backpropagation either by fitting the model simultaneously on the sequences across all the timesteps so far ($fully\ trained$) or updating the model obtained thus far by fitting it only on the latest sequence ($incrementally\ trained$). We limit the number of learning epochs to 10 during training for interactivity. We append the target vector of timestep T_i to the session sequence as input to the RNN at the subsequent timestep T_{i+1}. We empirically choose sigmoid activation function for hidden layer and binary cross entropy as the loss function at the output layer. During the test phase, instead of predicting the most likely intent vector, we predict the top-K candidate output intent vectors having the highest cosine similarity to the probabilistic output vector. These top-K vectors are picked from the historical sessions that the RNN has been trained on, and this ensures that arbitrary bitmaps which may not correspond to a meaningful SQL query are not predicted.

2.2 Active Learning

Active learning can be applied to RNN to test if it can perform well with limited training data. Instead of learning the RNN upon all the training session queries, the training set can be divided into two parts - session query sequences which are made available to the RNN, and query sequences held out from the RNN. By training the RNN upon the available query sequences and selecting informative sequences from the held-out data for inclusion into the training set, the RNN is incrementally refined to verify if it achieves high F1-scores on the test set without exhausting the entire training set. It should be noted that by query qu_i, we refer to its intent vector throughout this section.

For a given user session with queries qu_1, qu_2 and qu_3, the temporal sequences constructed are $qu_1 \longrightarrow qu_2$ and $qu_1, qu_2 \longrightarrow qu_3$ as shown in Figure 3. A held-out session means that the antecedent (e.g., qu_1, qu_2 in $qu_1, qu_2 \longrightarrow qu_3$) of a temporal sequence dependency is available while keeping the consequent (e.g, qu_3 in $qu_1, qu_2 \longrightarrow qu_3$) invisible. Thus, the trained RNN is supposed to predict the consequent of each temporal dependency given its

Algorithm 1: Active Learning for intent prediction

```
input  : $Tr_A$: Available training set of query sequences
         $Tr_H$ : Held-out training set of query sequences
         $Test$: Test set of query sequences
         L: learning algorithm for a classifier
output : $C^*$: An optimal temporal predictor of query intents
         $qu_{predicted}$: Test query successors predicted by $C^*$
1  $i \leftarrow 0$
2  while $\{Tr_H\} \neq \phi$ do                    // stopping criterion
3      $C_i \leftarrow$ learnClassifier(L, $Tr_A$)
4      $seq_{ambig} \leftarrow$ pickAmbiguousQuerySequence(L,$Tr_H$)
5      $seq_{labeled} \leftarrow$ obtainSuccessors($seq_{ambig}$)
6      $Tr_H \leftarrow Tr_H - seq_{labeled}$
7      $Tr_A \leftarrow seq_{labeled} \cup Tr_A$
8      $qu_{predicted} \leftarrow$ predictSuccessors($Test$, $C_i$)
9      $Test_{F1} \leftarrow$ computeF1($Test$, $qu_{predicted}$)
10     $i \leftarrow i + 1$
11 $C^* \leftarrow C_{i-1}$
12 return $C^*$, $qu_{predicted}$
```

antecedent. Among all such held-out sequences, if the RNN finds a particular sequence to be the hardest to predict the successor for, it is deemed to be an ambiguous sequence for RNN. The consequent or the successor query of that particular temporal sequence is made available to the RNN and is included into the available training set. In Figure 3, $qu_{11}, qu_{12}, ..., qu_{20} \longrightarrow qu_{21}$ is the hardest held-out temporal sequence and hence, it is included into the training set.

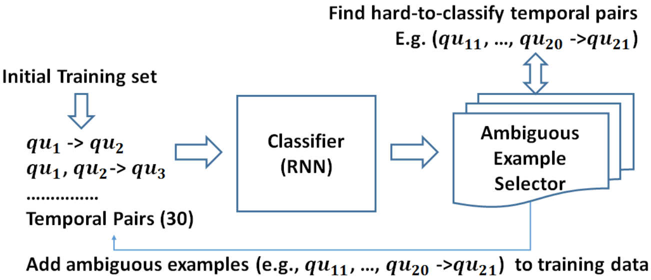

Find hard-to-classify temporal pairs
E.g. (qu_{11}, ..., qu_{20} ->qu_{21})
Initial Training set
qu_1 -> qu_2
qu_1, qu_2-> qu_3
....
Temporal Pairs (30)
Classifier
(RNN)
Ambiguous
Example
Selector
Add ambiguous examples (e.g., qu_{11}, ..., qu_{20} ->qu_{21}) to training data

Figure 3: Active learning for RNNs

Algorithm 1 describes the active learning based intent prediction algorithm using RNN. Tr_A and Tr_H are the available and held-out training datasets. In each active learning iteration, the most ambiguous query sequence is picked from Tr_H for which the successor query is the hardest to predict and the corresponding successor query is made available to RNN (lines 4 and 5). The query sequence is removed out of Tr_H and is added to Tr_A (lines 6 and 7). At the end of each iteration, the RNN learned thus far is evaluated on the fold test data until the held-out training set is exhausted. The purpose of using active learning algorithm is to verify if all the training set needs to be completely exhausted during learning to achieve a significantly high enough test F-measure. While ambiguous sequence selection is time consuming, if it helps active learning reach an early convergence to high F-measures, it is a trade-off worth considering.

2.2.1 Ambiguous Example Selection Strategy (Minimax). We have described in section 2.1 that the output layer of the RNN emits a probability vector. We compute the cosine similarity between the probability vector and the intent vectors of all the historical session queries that the RNN has learned from until then, and select the top-K intent vectors with the highest cosine similarity as the predicted intents. Hence, we can use the value of the maximum cosine similarity as the confidence measure and the inverse of the confidence value denotes the ambiguity. Among several temporal held-out sequences $seq_1, seq_2, ..., seq_n$, the one for which predicting the successor query is the hardest is inferred based on the value of the highest cosine similarity between the weight vector and the historical intent vectors. If the corresponding values of the highest cosine similarities are $maxSim_1, maxSim_2, ..., maxSim_n$, the most ambiguous query sequence is seq_i if i=$arg\ min_{i=1}^{n}$ $\{maxSim_i\}$, i.e., it has the least maximum cosine similarity (minimax) among all the sequences.

3 EXPERIMENTS

All our experiments were conducted on a Mac machine with a 4-core 2.8 GHz Intel Core i5 processor, 16GB RAM and 1.02 TB hard disk. Our experiments for supervised learning were conducted on queries arriving in an interleaved order from concurrent user sessions. Active learning experiments were conducted on 10-fold splits of the data into 10 different training and test set pairs with 90% and 10% of the user session queries respectively. We compute the precision and recall as follows:

$Precision = \frac{\#\{PredDim_i=1 \cap ActualDim_i=1\}}{\#\{PredDim_i=1\}}$

$Recall = \frac{\#\{PredDim_i=1 \cap ActualDim_i=1\}}{\#\{ActualDim_i=1\}}$

Each dimension $PredDim_i$ in the predicted intent vector is compared to the corresponding dimension $ActualDim_i$ in the actual vector, and the hits and misses are measured based on the fraction of 1s predicted accurately. We report F1-scores in our results.

Concurrent User Sessions: The queries arrive in batches of 10 and hence the 1958 queries from the NYCTaxiTrip dataset are divided across 196 learning episodes. While the RNN gets updated at the end of each learning episode with additional training data obtained from the queries during the episode, it is tested throughout the episode with the arrival of new query. For each new query, its embedding (query or operator) vector is fed to the RNN, which predicts the top-K intent vectors for the next query. K is set to 3 and we report the maximum F-measure among the top-3 predictions by averaging it across all the queries in each episode. Response time is the sum of the intent vector creation and execution times for the current query added to the intent vector prediction time for the next query. It is heavily dominated by the RNN prediction time which also includes its cumulative training time based on all the queries seen thus far. For incrementally trained vanilla (simple backpropagation) RNN, the training time depends only on the batch of queries from the latest episode.

LSTM (Long Short Term Memory) and GRU (Gated Recurrent Unit) perform same as vanilla RNN on the operator embedding intent vector (Figure 5) but GRUs perform slightly better than LSTM and vanilla on query embedding (Figure 4) vectors. The reason for this is that the average user session length is ≈18 queries per session which is not lengthy enough for advanced RNN variants to exploit the long short term dependencies. On an average, vanilla RNN seems to give competitive F-measures while also consuming lesser response time than LSTM and GRU (Figure 6). Incrementally trained RNN sacrifices test F-measures on query embedding but produces comparable quality as fully trained RNNs on operator embedding while incurring very low response times. Time charts on query embedding demonstrate similar patterns as operator embedding for both supervised and active learning which is why we omitted them for brevity.

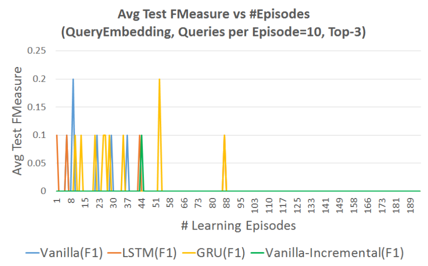


Figure 4: Concurrent Sessions (Query)

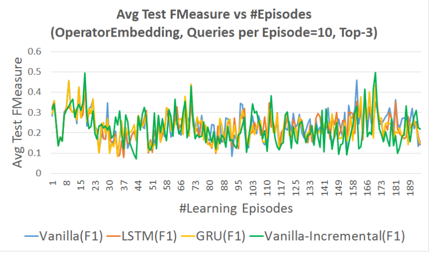


Figure 5: Concurrent Sessions (Operator)

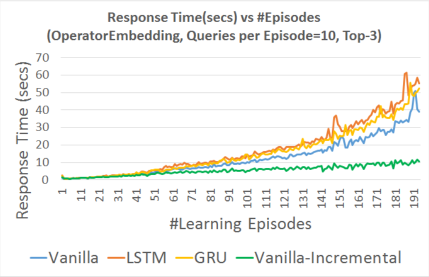


Figure 6: Response Times (Operator)

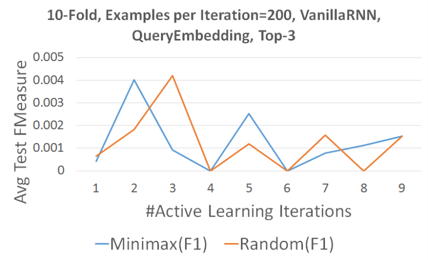


Figure 7: Active Learning (Query)

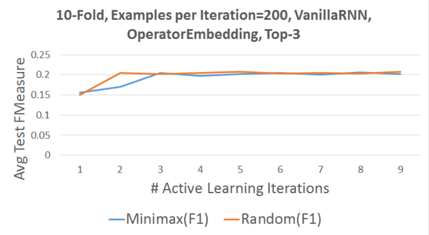


Figure 8: Active Learning (Operator)

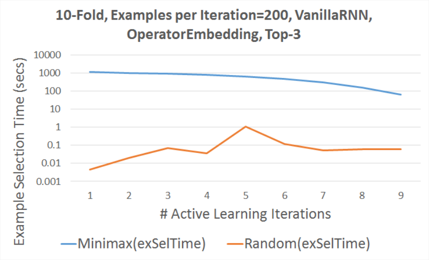


Figure 9: Selection Times (Operator)

A general observation is that operator embedding is more expressive and produces better F-measures than query embedding because predicting SQL operators in a query is easier than predicting the query in its entirety. We can also notice that the initial queries in concurrent sessions are easier to predict than the later ones. In the real world sessions we collected, each user arrives at a distinct goal by the end of the exploratory session. The first few queries may be similar across users, but each session becomes specialized with unique queries towards the end, thus inducing little overlap across sessions. This in turn makes workload prediction harder as the sessions progress concurrently.

Active Learning: Our 10-fold active learning experiments were upon the fully trained vanilla version of RNN using standard backpropagation algorithm. There were ≈1648 examples (temporal query sequence dependencies) in the training set and ≈190 test examples. We started with a seed available training set of 30 randomly chosen training examples and 1618 held-out training examples. We selected 200 examples in each iteration from the hold-out set and added them to the available set. We plot the average test F-measures over all the test sessions across 10 folds at each active learning iteration in Figures 7 and 8.

We compared the minimax example selection strategy described in section 2.2.1 with random selection. Our observations show that random strategy performs competitively to minimax on operator embedding as shown in Figure 8 and achieves earlier convergence to the eventual F-measure in the 2^{nd} iteration with 230 examples (14% training data). Minimax achieves slower convergence by iteration 3 with 430 examples and 26% training on operator embedding. Examples chosen in an iteration are used for training in the next iteration i.e., #training examples in iteration i = 30 + 200 x (i-1). Figure 7 shows that both strategies perform similarly on query embedding. A major drawback of using query embedding for active learning is the non-monotonic behavior with increasing iterations. This is due to the heavy sparsity in query embedding vectors (1 bit set out of 1190 dimensions) and fitting RNNs to more training points does not necessarily yield higher test F-measures. Minimax strategy incurs more example selection time than random selection (Figure 9) because of the expensive prediction done over the hold-out set which shrinks with increasing iterations thus also decreasing the selection times.

4 CONCLUSION

In this paper, we proposed the application of RNNs for dynamic intent prediction and two SQL specific embedding techniques for intent vector creation. Our experiments show that operator embedding is more effective than query embedding and that vanilla RNNs perform better than LSTMs or GRUs for user sessions of moderate length (#queries). We also show that an incrementally updated vanilla RNN model achieves substantially lesser response times than fully trained RNN models while making little to no sacrifice in prediction quality. Our active learning experiments show that random selection strategy achieves earlier convergence to competitive test F-measures as full training with just 14% of the training examples on operator embedding and lesser example selection time than minimax strategy.

REFERENCES

[1] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2014. Explore-by-example: An Automatic Query Steering Framework for Interactive Data Exploration. In *SIGMOD*. 517–528.

[2] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2016. AIDE: An Active Learning-Based Approach for Interactive Data Exploration. *IEEE TKDE* 28, 11 (2016), 2842–2856.

[3] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed Representations of Tuples for Entity Resolution. *Proc. VLDB Endow.* 11, 11 (July 2018), 1454–1467.

[4] Niranjan Kamat, Prasanth Jayachandran, Karthik Tunga, and Arnab Nandi. 2014. Distributed and interactive cube exploration. In *ICDE*. 472–483.

[5] Ben McCamish, Vahid Ghadakchi, Arash Termehchy, Behrouz Touri, and Liang Huang. 2018. The Data Interaction Game. In *SIGMOD*. 83–98.

[6] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD*. 19–34.

[7] Dan Murray. 2013. *Tableau Your Data!: Fast and Easy Visual Analysis with Tableau Software* (1st ed.). Wiley Publishing.

[8] Mohamed Sarwat, Raha Moraffah, Mohamed F. Mokbel, and Jamed L. Avery. 2017. Database System Support for Personalized Recommendation Applications. In *ICDE*. 1320–1331.

[9] Taxi and Limousine Commission. 2016. NYC Taxi Trip Dataset. {http://www.nyc.gov/html/tlc/html/technology/raw_data.shtml}. (2016).

Optimal Algorithm for Profiling Dynamic Arrays with Finite Values

Dingcheng Yang, Wenjian Yu, Junhui Deng
BNRist, Dept. Computer Science & Tech.,
Tsinghua Univ., Beijing, China
ydc15@mails.tsinghua.edu.cn,yu-wj@tsinghua.edu.cn,
deng@tsinghua.edu.cn

Shenghua Liu
CAS Key Lab. Network Data Science & Tech.,
Inst. Computing Technology, Chinese Academy of
Sciences, Beijing, China
liushenghua@ict.ac.cn

ABSTRACT

How can one quickly answer the most and top popular objects at any time, given a large log stream in a system of billions of users? It is equivalent to find the mode and top-frequent elements in a dynamic array corresponding to the log stream. However, most existing work either restrain the dynamic array within a sliding window, or do not take advantages of only one element can be added or removed in a log stream. Therefore, we propose a profiling algorithm, named S-Profile, which is of $O(1)$ time complexity for every updating of the dynamic array, and optimal in terms of computational complexity. With the profiling results, answering the queries on the statistics of dynamic array becomes trivial and fast. With the experiments of various settings of dynamic arrays, our accurate S-Profile algorithm outperforms the existing methods, showing at least 2X speedup to the heap based approach and 13X speedup to the balanced tree based approach.

1 INTRODUCTION

Many online systems, especially with billions of users, are generating a large stream of logs [6], recording users' dynamics in the systems, e.g. users (un)follow other users, "(dis)like" objects, enter (exit) live video channels, and click objects. Then, a question is raised:

How can we efficiently know the most popular objects (including users), i.e. mode, top-K popular ones, and even the distribution of frequency in a fast and large log stream at any time?

Mathematically, the questions converge to calculating and updating the statistics of a dynamic array of finite values. Thus, the existing fast algorithms on the statistics are as follows:

Mode of an array. The mode of an array and its corresponding frequency can be calculated by sorting the array (if it's of numeric value) and scanning the sorted array in $O(n \log n)$ time, where n is the length of array [7]. Notice that through judging the frequency of mode we can solve the element distinctness problem, which was proven to have to be solved with $\Omega(n \log n)$ time complexity [12, 15]. Therefore, calculating the mode of an array has the lower bound of $\Omega(n \log n)$ as well. If the elements of array can only take finite values, the complexity of calculating the mode can be reduced. Suppose they can only take m values. One can use m buckets to store the frequency of each distinct element. Then, the mode can be calculated in $O(n + m)$ time by scanning the m buckets.

The problem of range mode query, which calculates the mode of a sub-array $A[i \dots j]$ for a given array A and a pair of indices (i, j), has also been investigated [4, 10, 13]. The array with finite values was considered. With a static data structure, the range mode query can be answered in $O(\sqrt{n/\log n})$ time [4].

Majority and frequency approximation. The majority is the element whose frequency is more than half of n. An algorithm was proposed to find majority in $O(n)$ time and $O(1)$ space [3]. Many work on the statistics like frequency count and quantiles, are under a setting of sliding window [1, 2, 5, 8, 11]. They consider the most recently observed data elements (within the window) and calculate the statistics. Space-efficient algorithms were proposed to maintain the statistics over the sliding window on a stream.

However, those existing work slow down their algorithms without considering that the increase and decrease of object frequency are always 1 at a time in log streams. Therefore, we propose an algorithm S-Profile to keep profiling the dynamic array. With such a profile, we can answer the queries of the statistics: mode, top-K and frequency distributions.

In summary, S-Profile has the following advantages:

- **Optimal efficiency:** S-Profile needs $O(1)$ time complexity and $O(m)$ space complexity to profile dynamic arrays, where m is the maximum number of objects.
- **Querying Statistics:** With the profiling, we have sorted frequency-object pairs, and can simply answer the queries on mode, top-K, majority and other statistics in $O(1)$.
- **Applicable:** Our S-Profile can be plugged into most of log streams in many systems, and profiling objects of interest.

In experiments, S-Profile compares with the existing methods in various settings of dynamic arrays, and shows its performance and robustness.

2 AN $O(1)$-COMPLEXITY ALGORITHM FOR UPDATING THE MODE AND STATISTICS

We define tuples (x_i, c_i) as a log stream, where x_i and c_i is the object id and action in the i-th tuple. Action c_i can be either "add" or "remove", which, for example, can indicate object x_i is "liked" or "disliked", or user x_i is followed or unfollowed. Conceptually, we could imagine a dynamic array A of objects associated with a log stream, by appending object x_i into A if c_i is "add", and deleting object x_i from A if c_i is "remove". Dynamic array A is not necessarily generated and stored, which is defined for convenient description of our algorithm.

Therefore, our problem can be described as follows:

Informal Problem 1 (Profiling dynamic array). ***Given:*** *a log stream of tuples (x_i, c_i) adding and removing an object each time,*

- ***To build:*** *a data structure profiling the dynamic array A of objects associated with the log stream,*
- ***Such that:*** *at any time answering the queries on mode, top-K and other statistics of objects is trivial and fast.*

Let m be the maximum number of distinct objects in a log stream or dynamic array A. Without loss of generality, we assume id $x_i \in [1, m]$, i.e. integers between 1 and m. For any m distinct objects, we can map them into the integers from 1 to m as ids.

We can use m buckets to store the frequency of each distinct object. Let F be such a frequency array with length m. $F[i] \in F$ is the frequency of object with id i. With F, most statistics of A can be calculated without visiting A itself. For example, the mode of A refers to locations in F where the element has the maximum value. Although updating F with each tuple of a log stream trivial and costs $O(1)$, finding the maximum value in F at each time is still time consuming.

Therefore, we first introduce a proposed data structure of a profile, named "block set", which can answer the statistical queries in a trivial cost. And we show later that we can maintain such a profile in $O(1)$ time complexity and $O(m)$ space complexity.

2.1 Proposed data structure for profiling

In order to find the mode of A, we just need to care about the maximum in F. If only integers are added to A, the maximum element of F can be easily updated. However, in Problem 1 removing integer is also allowed. This complicates the calculation of mode and other statistics of A. So, the sorted array T must be employed and maintained. To facilitate the queries, T can be implemented as a binary tree. The heap and balanced tree are two kinds of binary tree, and are widely used for efficiently maintaining an sorted array. Upon a modification on A, they both can be updated in $O(\log m)$ time. The root node of a heap is the array element with the extreme value. This means the heap is only suitable for producing the A's elements with either maximum frequency (the mode) or the minimum frequency. The balanced tree is good at answering the query of median of A, and can also output the mode and top-K elements, etc. It should be pointed out, these general algorithms do not take the particularity of Problem 1 into account (the modification on F is restricted to plus 1 or minus 1). By the way, no one can maintain the sorted array under an arbitrary modification with a time complexity below $O(\log m)$, because it can be regarded as a sorting problem which has been proven to have $\Omega(m \log m)$ time complexity.

We use Figure 1 as an example to illustrate the proposed data structure for maintaining T. Suppose T has frequency values in ascending order. In order to locate the index of the i-th element of T in F and vice verse, two conversion arrays are defined: $TtoF$ and $FtoT$. In other words, we have $T[i] = F[TtoF_i]$ and $F[i] = T[FtoT_i]$. Here, we use both the subscript and bracket notation to specify an element of array. As shown in Figure 1(c), we can partition T into nonoverlapped segments according to its elements. Each such segment is called *block* here and represented by an integer triple (l, r, f), where l and r are starting and ending indices respectively, and f is the element value (frequency). So, a block $b = (l, r, f)$ always satisfies:

- $1 \leq l \leq r \leq m$
- $T_i = f, \quad \forall i \in [l, r]$
- $T_l > T_{l-1}, \quad \text{if } l > 1$
- $T_r < T_{r+1}, \quad \text{if } r < m$

There are at most m blocks, which form a *block set*. It fully captures the information in the sorted array T. An array of pointers called $PtrB$ is also needed to make a link from each element in T to its relevant block. According to the definition of block, we always have:

$$T_i = PtrB[i].f, \quad \text{and} \quad PtrB[i].l \leq i \leq PtrB[i].r \,. \qquad (1)$$

Here we use ".l" to denote the member l of a block, and so on.

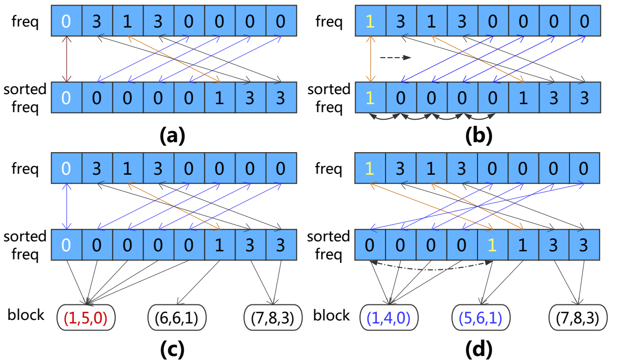


Figure 1: Illustration of the proposed *block set* for maintaining array T. (a) The initial F and T. (b) When "1"is added to A, a brute-force approach to maintain T includes four swaps of "1" rightwards and updates of $FtoT$ and $TtoF$. (c) The initial F and T, and the *block set*. F and T are up to be modified. (d) With the information of *block*, the swapping destination in T can be easily determined.

The block set B represents the sorted frequency array T, while with arrays $FtoS$, $StoF$ and $PtrB$ we no longer need to store F and T. These proposed new data structure well profile the dynamic array A. The remaining thing is to maintain them and answer the statistical queries on A in an efficient manner.

2.2 S-Profile: the $O(1)$-Complexity Updating Algorithm

We first consider the situation where an integer is added to A. As shown in Figure 1(a) and 1(b), a brute-fore approach to update T is swapping the updated frequency to its right-hand neighbor one by one, until T is in the appropriate order again. Now, with the proposed $PtrB$ and the block it points to, we can easily determine the index of T which is the destination of swapping the updated frequency. Then, we can update the relevant two blocks and pointer arrays (see Figure 1(d)).

Now, based on the situation shown as Figure 1(d), we assume a "4" is removed from A. As shown in Figure 2, we first locate the updated element in T. Then, with the information in its corresponding block we know with which it should be swapped. We further check if the updated frequency exists in T before. If it does not we need to create a new block (the case in Figure 2(b)), otherwise another block is modified.

The whole details of the algorithm for updating the data structure and returning the mode of A is described as Algorithm 1. We assume the data structures (B, $FtoS$, $StoF$ and $PtrB$) have been

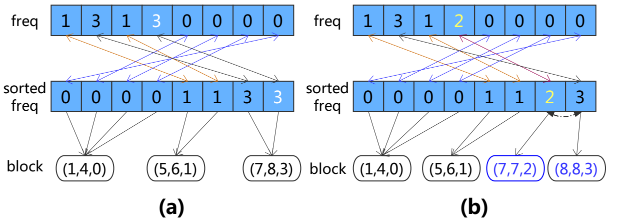


Figure 2: Illustration of the proposed data structure for a "remove" action on A. (a) The initial F and T, and the *block set*. A "4" is going to be deleted from A. (b) With the information of *block* and the pointer arrays, the *block set* can be easily updated to reflect the ordered T.

initialized, while Algorithm 1 responds to an event in the log stream and returns the updated mode and frequency.

Algorithm 1 S-Profile for updating the mode of array

Input: A tuple (x, c) in log stream, block set B, pointer arrays $FtoT$, $TtoF$ and $PtrB$, the length of sorted frequency array m
Output: The mode M, and the frequency v.

```
1:  $rank \leftarrow FtoT[x]$
2:  $b \leftarrow PtrB[rank]$
3:  $l \leftarrow b.l$ ;   $r \leftarrow b.r$;
4:  if $c$ is an "add" action then
5:      $b.r \leftarrow b.r - 1$
6:      if $b.r < b.l$ then
7:          Delete $b$
8:      end if
9:      if $r < m$ and $b.f + 1 = PtrB[r+1].f$ then
10:         $PtrB[r] \leftarrow PtrB[r+1]$
11:         $PtrB[r].l \leftarrow PtrB[r].l - 1$
12:     else
13:         Create a new block in $B$ and assign it to $PtrB[r]$
14:         $PtrB[r] \leftarrow (r,\ r,\ b.f + 1)$
15:     end if
16: else        /* It's a "remove" action */
17:     $b.l \leftarrow b.l + 1$
18:     if $b.r < b.l$ then
19:         Delete $b$
20:     end if
21:     if $l > 1$ and $b.f - 1 = PtrB[l-1].f$ then
22:         $PtrB[l] \leftarrow PtrB[l-1]$
23:         $PtrB[l].r \leftarrow PtrB[l].r + 1$
24:     else
25:         Create a new block in $B$ and assign it to $PtrB[l]$
26:         $PtrB[l] \leftarrow (l,\ l,\ b.f - 1)$
27:     end if
28: end if
29: $M \leftarrow TtoF[PtrB[m].l \ldots PtrB[m].r]$
30: $v \leftarrow PtrB[m].f$
```

As the proposed data structure maintains the sorted frequency array, it can be utilized to calculate the object with the minimum frequency (maybe a negative number) as well. We just need to replace Step 29 and 30 in Algorithm 1 with the following steps.

```
29a: $M \leftarrow TtoF[PtrB[1].l \ldots PtrB[1].r]$
30a: $v \leftarrow PtrB[1].f$
```

We can observe that the time complexity of the S-Profile algorithm is $O(1)$, as there is no iteration at all. The space complexity is $O(m)$, where m is the maximum number of objects in the log stream. Precisely, it needs $3m$ integers to store the pointer arrays and an additional storage for B. In the worst case B includes m blocks, but usually this number is much smaller than m.

Other queries on statistics of objects can also be answered. For example, the top-K order element is that whose frequency is the K-th largest. We can just use $PtrB[m - K + 1]$ to locate the block. Then, the frequency and object id can be obtained with block's member and the $TtoF$ array. Especially, the median in frequency can be located with the K/2-th element of the $PtrB$ array.

2.3 Possible Applications

For some mission-critical tasks (e.g., fraud detection) in big graphs, the efficiency to make decisions and infer interesting patterns is crucial. As a result, recent years have witnessed an increasing interest in heuristic "shaving" algorithms with low computational complexity [9, 14]. A critical step of them is to keep finding low-degree nodes at every time of shaving nodes from a graph. Thus, S-Profile can be plugged into such algorithms for further speedup, by treating a node as an object and its degree as frequency.

Furthermore, S-Profile can also deal with a sliding window on a log stream, by letting every tuple (x_i, c_i) outdated from the window be a new incoming tuple $(x_i, \bar{c}_i)$, where $\bar{c}_i$ is the opposite action of c_i.

3 EXPERIMENTAL RESULTS

We have implemented the proposed S-Profile algorithm and its counterparts in C++, and tested them with randomly generated log streams. The streams are produced with the following steps. We first randomly generate an "add" or "remove" action, with 70% and 30% probabilities respectively. Then, for each "add" action we randomly choose an object id according to a probability distribution (called *posPDF*). For each "remove" action another distribution (called *negPDF*) is used to randomly choose an object id. With this procedure, we obtained three test log streams:

- Stream1: both *posPDF* and *negPDF* are uniform random distribution on $[1, m]$.
- Stream2: both *posPDF* and *negPDF* are normal distributions with $\mu = 2m/3, m/3$ and $\sigma = m/6, m/6$, respectively.
- Stream3: *posPDF* is a normal distribution ($\mu = 4m/5$, $\sigma = m$), while *negPDF* is a lognormal distribution ($\mu = 3m/5$, $\sigma = m$).

In the following subsections, we first compare the proposed S-Profile with the heap based approach, for updating the mode and frequency. Then, the comparison with the balanced tree is presented for calculating the median. All experiments are carried out on a Linux machine with Intel Xeon E5-2630 CPUs (2.30 GHz). The CPU time (in second) of different algorithms are reported.

3.1 Comparison with the Heap

Heap is a kind of binary tree where the value in parent node must be larger or equal to the values in its children. Used to maintain the sorted frequency array, it is easy to obtain the mode (the root has the largest frequency). Notice that the balanced tree is inferior to the heap for calculating the mode.

In the experiment, a log stream is the input and the tested algorithm calculates the mode every time a tuple arrives. In Figure 3, we show the CPU times consumed by the heap based method and our S-Profile for the three log streams with varied length. The x-axis means the number of processed tuples (n). From the

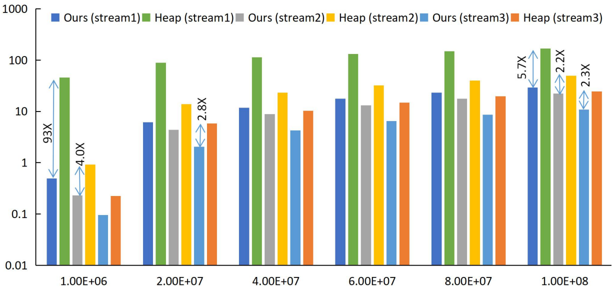


Figure 3: CPU times (in second) of the heap based method and our method for calculating the mode for n times ($m = 10^8$).

results we see that our method is at least 2.2X faster than the heap based method. Another experiment is carried out with fixed $n = 10^8$ and varied m. The results shown in Figure 4 also reveal that our S-Profile is at least 2X faster.

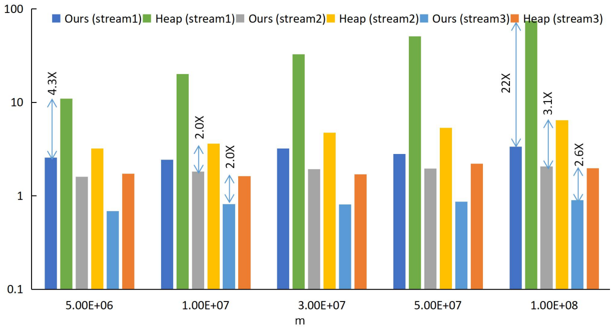


Figure 4: CPU times (in second) of the heap based method and our method for calculating the mode for $n = 10^8$ times.

For different kinds of log streams, the performance of the heap based method varies a lot. For the worst case updating the heap needs $O(\log m)$ time, despite this rarely happens in our tested streams. On the contrary, S-Profile needs $O(1)$ time for updating the data structure. This advantage is validated by the rather flat trend shown in Figure 5.

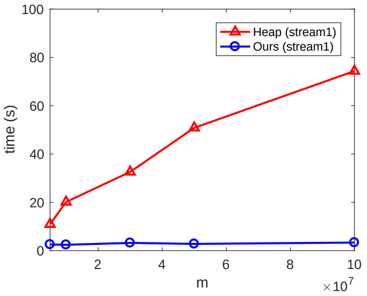


Figure 5: The trends of CPU time for varied m ($n = 10^8$).

It should be emphasized that, in addition to the speedup to the heap based method, our S-Profile possesses the advantage of wider applicability. Our method is not restricted to calculating the mode and corresponding frequency. As it well profiles the sorted frequency array, with it answering the queries on top-K and other statistics of objects is trivial and fast.

3.2 Comparison with the Balanced Tree

The proposed S-Profile can also calculate the median of the dynamic array. We compare it with the balanced tree based method implemented in the GNU C++ PBDS [16], which is more efficient than our implementation of balanced tree. The trends of CPU time are shown in Figure 6. They show that the runtime of the proposed S-Profile increases much less than that of the balanced tree based method when m increases. We can observe that the time of S-Profile is linearly depends on n, the number of modifications on array A, and hardly varies with different m. On the contrary, the balanced tree based method exhibits superlinear increase whether with n or m. Overall, the test results show that S-Profile is from 13X to 452X faster than the balanced tree based method on updating the median of a dynamic array.

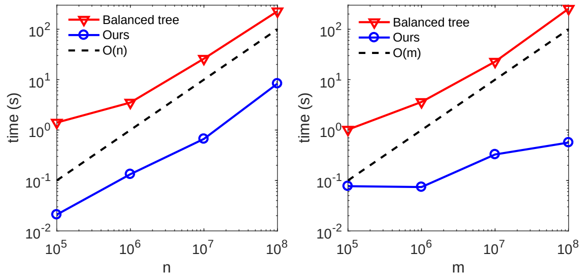


Figure 6: Comparison of the balanced tree based method and our method for calculating the median. Left: CPU time vs. n ($m = 10^6$). Right: CPU time vs. m ($n = 10^6$).

4 CONCLUSIONS

We propose an accurate algorithm, S-Profile, to fast keep profiling the dynamic array from online systems. It has the following advantages:

- **Optimal efficiency:** S-Profile needs $O(1)$ time complexity for every updating of a dynamic array, and totally linear complexity in memory.
- **Querying Statistics:** With profiling, at any time we can answer the statistical queries in a trivial and fast way.
- **Applicable:** S-Profile can be plugged into most of log streams, and heuristic graph mining algorithms.

ACKNOWLEDGMENTS

This work is supported by NSFC under grant Nos. 61872206 and 61772498, and in part by Beijing NSF No. 4172059.

REFERENCES

[1] A. Arasu and G. S. Manku. 2004. Approximate counts and quantiles over sliding windows. In *Proceedings of the 23 ACM Symposium on Principles of Database Systems (PODS)*. 286–296.

[2] B. Babcock, M. Datar, and R. Motwani. 2002. Sampling from a moving window over streaming data. In *Proceedings of the 13th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 633–634.

[3] R. S. Boyer and J. S. Moore. 1991. MJRTY—A fast majority vote algorithm. In *Automated Reasoning*. Springer, 105–117.

[4] T. M. Chan, S. Durocher, K. G. Larsen, J. Morrison, and B. T. Wilkinson. 2014. Linear-space data structures for range mode query in arrays. *Theory of Computing Systems* 55, 4 (2014), 719–741.

[5] M. Datar, A. Gionis, P. Indyk, and R. Motwani. 2002. Maintaining stream statistics over sliding windows. *SIAM J. Comput.* 31, 6 (2002), 1794–1813.

[6] M. Dietz and G. Pernul. 2018. Big log data stream processing: Adapting an anomaly detection technique. In *International Conference on Database and Expert Systems Applications*. 159–166.

[7] D. Dobkin and J. I. Munro. 1980. Determining the mode. *Theoretical Computer Science* 12, 3 (1980), 255–263.

[8] P. B. Gibbons and S. Tirthapura. 2002. Distributed streams algorithms for sliding windows. In *Proceedings of the 14th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 63–72.

[9] B. Hooi, H. A. Song, A. Beutel, N. Shah, K. Shin, and C. Faloutsos. 2016. Fraudar: Bounding graph fraud in the face of camouflage. In *Proceedings of International Conference on Knowledge Discovery and Data Mining (KDD)*. 895–904.

[10] D. Krizanc, P. Morin, and M. Smid. 2005. Range mode and range median queries on lists and trees. *Nordic Journal of Computing* 12, 1 (2005), 1–17.

[11] X. Lin, H. Lu, J. Xu, and J. X. Yu. 2004. Continuously maintaining quantile summaries of the most recent n elements over a data stream. In *Proceedings of the 20th International Conference on Data Engineering (ICDE)*. 362.

[12] A. Lubiw and A. Rácz. 1991. A lower bound for the integer element distinctness problem. *Information and Computation* 94, 1 (1991), 83–92.

[13] H. Petersen and S. Grabowski. 2009. Range mode and range median queries in constant time and sub-quadratic space. *Inform. Process. Lett.* 109, 4 (2009), 225–228.

[14] K. Shin, B. Hooi, J. Kim, and C. Faloutsos. 2017. DenseAlert: Incremental dense-subtensor detection in tensor streams. In *Proceedings of International Conference on Knowledge Discovery and Data Mining (KDD)*. 1057–1066.

[15] J. M. Steele and A. C. Yao. 1982. Lower bounds for algebraic decision trees. *Journal of Algorithms* 3, 1 (1982), 1–8.

[16] A. Tavory, V. Dreizin, and B. Kosnik. [n. d.]. Policy-Based Data Structures. https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/

Publishing Differentially Private Datasets via Stable Microaggregation

Masooma Iftikhar, Qing Wang, Yu Lin
Australian National University, Canberra, Australia
{masooma.iftikhar,qing.wang,yu.lin}@anu.edu.au

ABSTRACT

In recent years, differential privacy has emerged as one formal notion of privacy. Data release based on differential privacy can help researchers to perform statistical analysis on sensitive data of individuals. To publish differentially private datasets, there is a need for preserving data utility, along with data privacy. However, the utility of differentially private datasets is often limited, due to the amount of noise being added to the results of queries. In this paper, we address this issue by proposing a microaggregation-based framework that incorporates microaggregation and differential privacy into the data publishing process. We formulate a new notion of *stable microaggregation* to characterize a desired property of microaggregation and further develop a simple yet effective *stable microaggregation algorithm*. We experimentally verify the utility reduction of our proposed framework on real-world datasets. The experiments show that the proposed framework outperforms the state-of-the-art methods by providing better with-in cluster homogeneity and also reducing noise added into differentially private datasets significantly.

1 INTRODUCTION

Publishing data about individuals often poses a privacy threat because data may contain the sensitive information about individuals, e.g., medical history, and publishing them would intrude upon individual privacy. Thus, to preserve data privacy of individuals, various anonymization techniques have been proposed for data publishing, such as k-anonymity and its extensions [10]. Particularly, with the emerging of differential privacy in recent years [3, 5], a number of works have considered to release differentially private datasets [6, 11]. Such differentially private datasets can guarantee differential privacy controlled by a privacy parameter ε in a robust statistical way.

Broadly speaking, there are two common methods used for generating ε-differentially private datasets in the literature: one is based on differential privacy compliant histograms [11] and the other is based on record perturbation [9]. Histogram-based approaches have some limitations, including: being limited to histogram queries and the exponential growth of the number of histogram bins with the number of attributes [8]. On the other hand, record perturbation based approaches require a large amount of noise being added into the results of queries [9], though these approaches are not limited to histogram queries and allow dealing with any type of attributes.

Nevertheless, when generating differentially private datasets, there always is a trade-off being made between privacy and utility of published data. Ideally, we want to preserve the privacy of individuals while still maintaining the usefulness of data for performing statistical analysis. The utility of ε-differentially private datasets is however limited due to the amount of noise being added to guarantee differential privacy. To enhance the utility of ε-differentially private datasets, in [9] a microaggregation-based mechanism, i.e., *insensitive microaggregation*, has been proposed. It uses microaggregation to achieve k-anonymity in which a certain correspondence between clusters in the microaggregated datasets of two neighboring datasets is imposed. In doing so, the noise added to guarantee differential privacy can be greatly reduced. However, insensitive microaggregation still has the limitations: (1) it yields worse within-cluster homogeneity due to a total order relation required for the distance function [9], and (2) the minimum cluster size k grows with the size n of the dataset and one thus needs $k \geq \sqrt{n}$ to reduce noise.

Contributions. In this paper, we consider the problem of generating ε-differentially private datasets by incorporating microaggregation into the data publishing process. Our work makes the following contributions:

- We present a microaggregation-based framework for generating ε-differentially private datasets and formulate a novel notion of *stable microaggregation* to characterize the correspondence of clusters in microaggregated datasets.
- We propose a stable microaggregation algorithm that can ensure the correspondence of clusters in the microaggregated datasets of two neighboring datasets.
- We experimentally verify the utility reduction of our proposed framework on two real-world datasets containing numerical data. It shows that our algorithm can effectively enhance the utility of released data by providing better within-cluster homogeneity and reducing the amount of noise, in comparison with the state-of-the-art methods.

Related work. Among early works on data anonymization, k-anonymity [10] is a privacy model widely applied to guarantee data privacy of individuals. The popularity of k-anonymity has led to various attempts to address the limitations of k-anonymity [10]. On the other hand, differential privacy [3, 5] is a recent privacy notion that allows statistical analysis of sensitive data while providing strong privacy guarantees. A number of works [8, 9] have combined k-anonymity and differential privacy to enhance the utility of data release. One of these works used microaggregation to achieve k-anonymity, which can reduce the amount of noise added to differentially private datasets [2]. Microaggregation [1] is a family of anonymization algorithms that group similar (homogeneous) records into clusters, then replace each record with its cluster representative. MDAV [2] is the most widely used microaggregation algorithm. The target of a microaggregation algorithm is to yield minimum information loss by maximizing with-in cluster homogeneity. However, the existing works, including MDAV [2] and insensitive microaggregation [9], either produce a low degree of with-in cluster homogeneity or fail to reduce the amount of noise independent of the size of a dataset. Our work in this paper can alleviate both issues.

10.5441/002/edbt.2019.81

2 PROBLEM FORMULATION

Let $\mathcal{D}$ be a class of possible datasets. A dataset $X \in \mathcal{D}$ consists of a set of records, each $r_i \in X$ being associated with a set of attributes A. Each individual has only one record in a dataset X.

Definition 2.1. (*Neighboring datasets*) Two datasets $X, Y \in \mathcal{D}$ are said to be *neighboring*, denoted as $X \sim Y$, if $|X| = |Y| = n$, but X and Y differ in one record.

Given a dataset X, we want to generate X_ε (an anonymized version of X) that can provide ε-differential privacy guarantee for protecting the privacy of individuals' records in X.

Definition 2.2. (*Differentially private datasets*) A randomized mechanism $\mathcal{K} : \mathcal{D} \to \mathcal{D}$ provides ε-differentially private datasets, if for each pair of neighboring datasets $X \sim Y$, and all possible outputs $\mathcal{D}_\varepsilon \subseteq range(\mathcal{K})$, it holds

$$Pr[\mathcal{K}(X) \in \mathcal{D}_\varepsilon] \leq e^\varepsilon \times Pr[\mathcal{K}(Y) \in \mathcal{D}_\varepsilon] \qquad (1)$$

where $\varepsilon > 0$ is the differential privacy parameter. Smaller values of ε provide stronger privacy guarantees [4].

ε-differential privacy [3] was originally proposed as a privacy model to protect the responses of interactive queries to a dataset. A query is a function f that extracts data against records in the dataset. A standard way for achieving ε-differential privacy is by adding random noise to the true response of f, and the random noise is calibrated according to the *sensitivity* (Δ) of f, e.g. L_1-sensitivity [5]. For numerical data, the addition of noise can be drawn from a Laplace distribution by first computing the answer $f(X)$ and then generating the noisy answer $f(X) = f(X) + Lap(\Delta(f)/\epsilon)$ to provide ε-differential privacy. Although ε-differential privacy was not initially adapted for the purpose of generating anonymized datasets, but later in [7, 8] differentially private datasets were generated by considering data publishing as the answers to subsequent queries for each record in the dataset.

The L_1-sensitivity of f measures the maximum variation in the query f between two neighboring datasets $X \sim Y$ as follows.

Definition 2.3. (L_1-*sensitivity*) The L_1-sensitivity of a query $f : \mathcal{D} \to \mathbb{R}^d$ is the smallest number $\Delta(f)$ such that for all neighboring datasets $X \sim Y \in \mathcal{D}$

$$\|f(X) - f(Y)\|_1 \leq \Delta(f), \qquad (2)$$

where $\|.\|_1$ denotes the L_1-norm.

Given a dataset X, a microaggregated dataset $\overline{X}$ is created by a microaggregation algorithm $\mathcal{M}$ in two stages. First, X is partitioned into a set of clusters C_X, such that each cluster in C_X has at least k records, where k is a preset constant value, and the records within each cluster are as similar as possible (homogeneous). Second, it aggregates each cluster in C_X by replacing each record with the representative record of the cluster.

In this paper, we aim to generate ε-differentially private datasets by using microaggregation for improving data utility. As illustrated in Figure 1, a microaggregated dataset $\overline{X}$ resulting from running $\mathcal{M}$ over X is added between X and X_ε to increase utility of X_ε. In doing so, the original query f is approximated by $f \circ \mathcal{M}$, since f is run on the microaggregated dataset $\overline{X}$ rather than the original dataset X. This thus introduces two kinds of errors: one is the random noise, which depends on the sensitivity $\Delta(f)$ of query f to guarantee ε-differential privacy, and the other one is due to computing f over $\overline{X}$ instead of X. As will be discussed in Section 4, the first kind of error is much larger than the second kind of error in terms of the information loss in ε-differentially private datasets. To increase the overall utility, the key challenge is how to reduce $\Delta(f \circ \mathcal{M})$ such that $\Delta(f \circ \mathcal{M}) \leq \Delta(f)$.

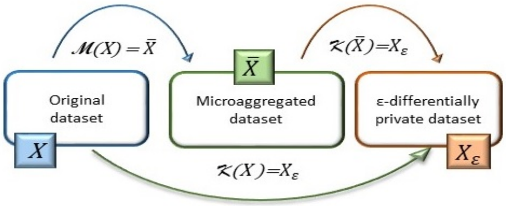

$\mathcal{M}(X) = \bar{X}$
$\mathcal{K}(\bar{X}) = X_\varepsilon$
Original dataset
Microaggregated dataset
ε-differentially private dataset
X
$\bar{X}$
X_ε
$\mathcal{K}(X) = X_\varepsilon$

Figure 1: Problem setting.

3 PROPOSED FRAMEWORK

In this section we present the details of the proposed framework.

3.1 Stable Microaggregation

Given $X \sim Y$ that only differ in a single record, their microaggregated datasets $\overline{X}$ and $\overline{Y}$ however may generate considerably different clusters, leading to a much larger $\Delta(f \circ \mathcal{M})$ than $\Delta(f)$. Suppose that we modify a record x in X to x' in Y, i.e., $X \sim Y$. As depicted in Figure 2, a microaggregation algorithm $\mathcal{M}$ (e.g. MDAV [2]) with $k = 4$ can generate C_X and C_Y over X and Y, respectively. Although X and Y only differ in one record, the clusters in C_X and C_Y are completely unrelated. The maximum variation between one cluster from C_X and another unrelated cluster from C_Y is $\Delta(f)$. Since there are n/k clusters in C_X and C_Y, $\Delta(f \circ \mathcal{M}) = n/k \times \Delta(f)$, which can be significantly higher than $\Delta(f)$ when the datasets are large, i.e., n is large.

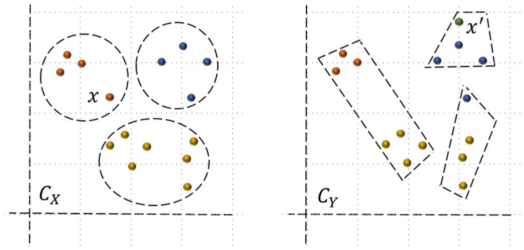


Figure 2: Clusters C_X and C_Y generated by $\mathcal{M}$ over $X \sim Y$.

To address the above issue, the notion of *insensitive* microaggregation was proposed [9]. A microaggregation algorithm $\mathcal{M}$ is said to be *insensitive* if, for every pair of neighboring datasets $X \sim Y$, there is a bijection between C_X and C_Y such that each pair of corresponding clusters differs at most in a single record. This implies that the maximum variation between each pair of corresponding clusters is reduced to $1/k \times \Delta(f)$. Since there are still n/k clusters, $\Delta(f \circ \mathcal{M})$ is $n/k \times \Delta(f)/k$. As a result, insensitive microaggregation may greatly reduce sensitivity as compared with $n/k \times \Delta(f)$ for standard microaggregation.

However, insensitive microaggregation still has some limitations. First, to achieve $\Delta(f \circ \mathcal{M}) \leq \Delta(f)$ as desired, $(n/k \times \Delta(f)/k) \leq \Delta(f)$ must hold. Therefore, one needs $k \geq \sqrt{n}$ in order to reduce added noise in comparison with directly applying $\mathcal{K}$ over X [8]. For large datasets, k thus needs to be large enough for reduced sensitivity. Second, as noted in the work [8] and will also be discussed in Section 4, the clusters generated by insensitive microaggregation are often less homogeneous than the clusters generated by standard microaggregation, such as MDAV [2]. This is because, to ensure the insensitive property, the distance function used by insensitive microaggregation algorithms must be consistent with the total order relation $\leq_X$ [9]. To alleviate these limitations, we define the notion of *stable* microaggregation.

Definition 3.1. (*Stable microaggregation*) Let $\mathcal{M}$ be a microaggregation algorithm, $C_X = \{c_1, ..., c_n\}$ be the set of clusters that results from running $\mathcal{M}$ on X, and $C_Y = \{c'_1, ..., c'_n\}$ be the set of clusters that results from running $\mathcal{M}$ on Y. $\mathcal{M}$ is *stable* if, for every pair of neighboring datasets $X \sim Y$, there is a bijection between C_X and C_Y such that at most two pairs of corresponding clusters in C_X and C_Y differ in a single record.

Since stable microaggregation affects at most two pairs of corresponding clusters in C_X and C_Y, $\Delta(f \circ \mathcal{M})$ is further reduced to $(2 \times \Delta(f)/k)$ as compared to $(n/k \times \Delta(f)/k)$ for insensitive microaggregation. Thus, when $k \geq 2$, the addition of noise can always be reduced in comparison with directly applying $\mathcal{K}$ over X, regardless of the size of a dataset.

Algorithm 1: *Stable Microaggregation Algorithm*

```
Input: $X \sim Y$ where $r := X - Y$ and $r' := Y - X$
       $\mathcal{M}$ : a standard microaggregation algorithm
Output: $\overline{X}, \overline{Y}$
1  $C_X \leftarrow \{c_1, ..., c_n\}$ generated by $\mathcal{M}_p$ over $X$
2  $C_Y \leftarrow replace(C_X, r, r')$
3  $D, L := \phi$
4  foreach $c_i \in C_X$ do
5      $D := D \cup \{(dist(r', r_{c_i}), c_i)\}$
6  $d_{min}, c_{min} \leftarrow \mathcal{F}_{min}(D)$
7  if $r' \in c_{min}$ then
8      $\overline{Y} \leftarrow \mathcal{M}_a(C_Y)$
9  else
10     $c_i := c(r')$
11     $D := D - \{(\mathcal{G}_{dist}(D, c_i), c_i)\}$
12     foreach $c_j \in C_X \setminus \{c_i\}$ do
13         $d_j \leftarrow \mathcal{G}_{dist}(D, c_j)$
14         $D := D - \{(d_j, c_j)\} \cup \{(dist(r_{c_i}, r_{c_j}) + d_j, c_j)\}$
15     $d_{min}, c_{min} \leftarrow \mathcal{F}_{min}(D)$
16     foreach $r_i \in c_{min}$ do
17         $swap(C_Y, r', r_i)$
18         $\overline{Y}_i \leftarrow \mathcal{M}_a(C_Y)$
19         $L := L \cup \{(\mathcal{I}_{loss}(Y_i, \overline{Y}_i), \overline{Y}_i)\}$
20     $\overline{Y} \leftarrow \mathcal{F}_{min}(L)$
21 $\overline{X} \leftarrow \mathcal{M}_a(C_X)$
22 Return $\overline{X}, \overline{Y}$
```

3.2 Algorithm Description

Our proposed *stable microaggregation algorithm* is described in Algorithm 1. Given $X \sim Y$, we start with partitioning the dataset X into C_X by $\mathcal{M}_p$, i.e., the partition function of a microaggregation algorithm $\mathcal{M}$. Then we replace the record $r \in C_X$ with r' and initialize D and L (Lines 1-3). For each cluster $c_i \in C_X$, by means of function *dist*(), we compute distance between r' and r_{c_i}, where r_{c_i} is the representative record of c_i. Then, we compute d_{min}, i.e., the minimum distance in D, and c_{min}, i.e., the cluster in C_X with d_{min}, by means of $\mathcal{F}_{min}$ function (Lines 4-6). If r' is in the cluster c_{min} of C_Y, then we aggregate C_Y by $\mathcal{M}_a$ that is the aggregation function of the microaggregation algorithm $\mathcal{M}$ (Lines 7-8). In this case, only one pair of corresponding clusters in C_X and C_Y is affected. Otherwise, for each cluster $c_j \in C_X \setminus \{c_i\}$ where $r' \in c_i$, we compute the distance between the representative records of clusters c_i and c_j, i.e., r_{c_i} and r_{c_j}. We proceed with updating D by summing up both distances of the corresponding clusters, excluding the distance of c_i obtained by function $\mathcal{G}_{dist}$ (Lines 10-14). In order to get the cluster c_j that is of the minimum distance from c_i, we find d_{min}, i.e., the minimum distance, and c_{min}, i.e., the cluster $c_j \in C_X$ with d_{min} from D, by means of $\mathcal{F}_{min}$ function. After that, we swap r' in c_i of C_Y with each record r_i in c_{min} of C_Y, and compute C_Y with the minimum information loss by function $\mathcal{I}_{loss}$ (Lines 15-20). In this case, at most two pairs of clusters differ at most in a single record. The algorithm terminates by returning the microaggregated datasets $\overline{X}$ and $\overline{Y}$ that have the minimum information loss.

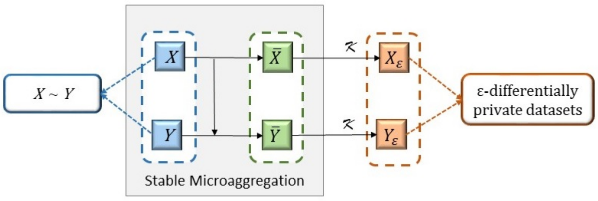


Figure 3: Proposed framework to generate ε-differentially private datasets via stable microaggregation.

A high-level description of our proposed framework is presented in Figure 3, in which stable microaggregation is applied to generate $\overline{X}$ and $\overline{Y}$ by running *Algorithm 1* over $X \sim Y$. Then ε-differentially private datasets X_ε and Y_ε are generated by applying $\mathcal{K}$ over $\overline{X}$ and $\overline{Y}$, respectively.

4 EXPERIMENTS

We evaluated the proposed framework to study how stable microaggregation enhances the utility of differentially private datasets.

Datasets. We used two datasets in the experiments: (1) CENSUS dataset[1] contains 1,080 records [2, 8, 9]. As in [9] we took 4 numerical attributes FEDTAX (Federal income tax liability), FICA (Social security retirement payroll deduction), INTVAL (Amount of interest income) and POTHVAL (Total other persons income). (2) EIA dataset[1] contains 4,092 records [1]. We took 4 numerical attributes attributes RESREVENUE (Revenue from sales to residential consumers), RESSALES (Sales to residential consumers), TOTREVENUE (Revenue from sales to all consumers), and TOTSALES (sales to all consumers).

Following [9], we consider the sensitivity of an attribute to be the difference between the lower bound (i.e. 0) and upper bound (1.5 × the maximum value) of the attribute. For both CENSUS and EIA datasets, the value of k is set to between 2 and 100.

Evaluation measure. We used the measure $IL1s$ [12] to compute the *information loss* between the original and differentially private datasets. Formally, for each record r_i,

$$IL1s = \frac{1}{|A| \cdot n} \sum_{i=1}^{n} \sum_{j=0}^{|A|} \frac{|x_{ij} - x'_{ij}|}{\sqrt{2} S_j} \tag{3}$$

where $|A|$ is the number of attributes, n is the number of records in the dataset, x_{ij} is the value of attribute $a_j \in A$ for record r_i in the original dataset, x'_{ij} is the value of attribute $a_j \in A$ for record

[1] http://neon.vb.cbs.nl/casc/CASCtestsets.htm

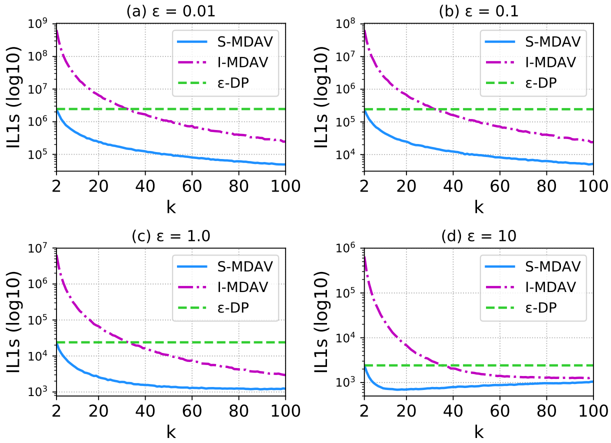


Figure 5: Evolution of $IL1s$ using S-MDAV, I-MDAV and ε-DP for different values of k and ε in CENSUS.

r_i in the corresponding differentially private dataset, and S_j is the standard deviation of attribute $a_j \in A$ in the original dataset.

Baseline Methods. We considered the following baseline methods: (1) MDAV, which is a standard microaggregation algorithm [2], (2) I-MDAV, which is an insensitive microaggregation algorithm proposed in [9], and (3) ε-DP, which is a standard ε-differential privacy algorithm in which noise is added using the Laplace mechanism [5]. We use S-MDAV to refer to our proposed stable microaggregation algorithm, which extends MDAV in partitioning and aggregation.

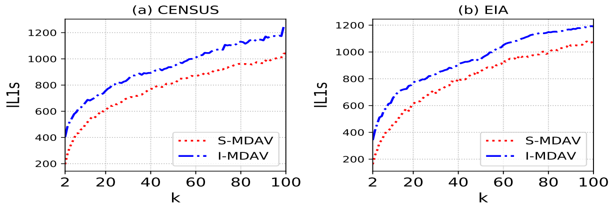


Figure 4: Evolution of $IL1s$ using MDAV and I-MDAV for different values of k: (a) CENSUS and (b) EIA.

Experimental results. We first conducted experiments to compare the information loss of microaggregated datasets that are generated by MDAV and I-MDAV under varying k between 2 to 100. The results are shown in Figure 4. We observe that, for both CENSUS and EIA datasets, the information loss of microaggregated datasets is less with MDAV as compared to I-MDAV. This is because the clusters generated by MDAV are more homogeneous than the clusters generated by I-MDAV. As we used MDAV in our algorithm S-MDAV to generate the clusters in C_X as well as most of the clusters in C_Y, S-MDAV decreases the sensitivity of $f \circ \mathcal{M}$ and thus reduces the errors caused by microaggregation.

Then, to verify the overall utility of ε-differentially private datasets, we conducted experiments to compare the information loss between the original and ε-differentially private datasets generated by using our algorithm S-MDAV and the baseline methods I-MDAV and ε-DP. Figures 5 and 6 present our experimental results. For ε-DP, we used the following privacy parameters $\varepsilon = [0.01, 0.1, 1.0, 10.0]$, which cover the range of differential privacy levels widely used in the literature [4, 7, 8]. For each parameter setting of ε, we ran 3 times and take the average result. The information loss for ε-DP is displayed as horizontal lines, as ε-DP does not depend on k.

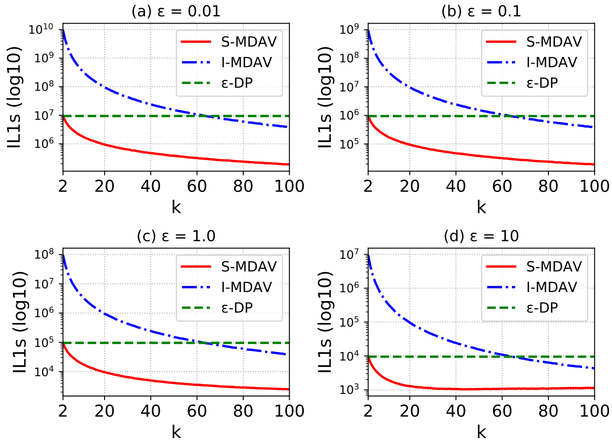


Figure 6: Evolution of $IL1s$ using S-MDAV, I-MDAV and ε-DP for different values of k and ε in EIA.

Regarding the evolution of $IL1s$ values shown in Figures 5 and 6, we can see that, for every value of ε, I-MDAV is only able to achieve $\Delta(f \circ \mathcal{M}) \leq \Delta(f)$ if $k \geq \sqrt{n}$, i.e., ($k = \sqrt{1,080} \approx 33$ for CENSUS and $k = \sqrt{4,092} \approx 64$ for EIA). This is consistent with the previous discussion in Section 3. Nonetheless, this also means that for large datasets I-MDAV requires k to be enough large in order to effectively reduce $\Delta(f \circ \mathcal{M})$, i.e., the size of k grows with the size of a dataset n. In contrast, for S-MDAV, as stated in Section 3, one needs $k \geq 2$ to reduce $\Delta(f \circ \mathcal{M})$ as compared to ε-DP. As the experiments show that our proposed algorithm S-MDAV leads to less information loss for every value of ε as compared to I-MDAV and ε-DP in both CENSUS and EIA datasets. This is because the sensitivity $\Delta(f \circ \mathcal{M})$ is significantly reduced when S-MDAV is used for microaggregation.

We have also noticed that by approximating a query f to $f \circ \mathcal{M}$ via microaggregation, the errors caused by random noise that depends on the sensitivity of $f \circ \mathcal{M}$ dominate the impact on the utility of differentially private datasets generated via microaggregation, compared to the errors existing between the original and microaggregated datasets.

REFERENCES

[1] Josep Domingo-Ferrer, Antoni Martínez-Ballesté, Josep Maria Mateo-Sanz, and Francesc Sebé. 2006. Efficient multivariate data-oriented microaggregation. *The VLDB Journal* 15, 4 (2006), 355–369.

[2] Josep Domingo-Ferrer and Vicenç Torra. 2005. Ordinal, continuous and heterogeneous k-anonymity through microaggregation. *KDD* 11, 2 (2005), 195–212.

[3] Cynthia Dwork. 2006. Differential Privacy. In *ICALP.* 1–12.

[4] Cynthia Dwork. 2011. A firm foundation for private data analysis. *Communications of the ACM-* 54, 1 (2011), 86–95.

[5] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *TCC.* 265–284.

[6] Ashwin Machanavajjhala, Xi He, and Michael Hay. 2017. Differential privacy in the wild: A tutorial on current practices & open challenges. In *SIGMOD.* 1727–1730.

[7] Ashwin Machanavajjhala, Daniel Kifer, John Abowd, Johannes Gehrke, and Lars Vilhuber. 2008. Privacy: Theory meets practice on the map. In *ICDE.* 277–286.

[8] Jordi Soria-Comas and Josep Domingo-Ferrer. 2018. Differentially private data publishing via optimal univariate microaggregation and record perturbation. *Knowledge-Based Systems* 153 (2018), 78–90.

[9] Jordi Soria-Comas, Josep Domingo-Ferrer, David Sánchez, and Sergio Martínez. 2014. Enhancing data utility in differential privacy via microaggregation-based k-anonymity. *The VLDB Journal* 23, 5 (2014), 771–794.

[10] K Wang, R Chen, BC Fung, and PS Yu. 2010. Privacy-preserving data publishing: A survey on recent developments. *ACM Computing Surveys-* (2010).

[11] Jia Xu, Zhenjie Zhang, Xiaokui Xiao, Yin Yang, and Ge Yu. 2012. Differentially Private Histogram Publication. In *ICDE.* IEEE, 32–43.

[12] William E Yancey, William E Winkler, and Robert H Creecy. 2002. Disclosure risk assessment in perturbative microdata protection. In *Inference control in statistical databases.* 135–152.

Range Query Processing for Monitoring Applications over Untrustworthy Clouds

Hoang Van Tran
Univ Rennes 1, IRISA, Lannion, France

Tristan Allard
Univ Rennes 1, IRISA, Rennes, France

Laurent D'Orazio
Univ Rennes 1, IRISA, Lannion, France

Amr El Abbadi
UC Santa Barbara, California, USA

ABSTRACT

Privacy is a major concern in cloud computing since clouds are considered as untrusted environments. In this study, we address the problem of privacy-preserving range query processing on clouds. Several solutions have been proposed in this line of work, however, they become inefficient or impractical for many monitoring applications, including real-time monitoring and predicting the spatial spread of seasonal epidemics (e.g., H1N1 influenza). In this case, a system often confronts a high rate of incoming data. Prior schemes may thus suffer from potential performance issues, e.g., overload or bottleneck. In this paper, we introduce an extension of PINED-RQ to address these limitations. We also demonstrate experimentally that our solution outperforms PINED-RQ.

1 INTRODUCTION

Reducing the impact of seasonal epidemics (e.g., H1N1 influenza) is demanding for public health officials. Early detection of spatial spread of the epidemics could help alleviate severe consequences. It is thus important to track and predict the spread of such diseases in the population. To do that, an individual can interact with a website or mobile application to report personal data (e.g., age, phone, sex, symptoms, travel plan, social network user name, ...), to be utilized for real-time predicting analyses. Systems usually run in very short periods, a few days or weeks after an epidemic emerges. Due to the need of significant computing capacity and the high speed of incoming data, it is desirable to use cloud services for managing and exploiting submitted data. However, cloud computing suffers from privacy issues, e.g., sensitive information can be exploited by cloud's administrators.

Encrypting outsourced data is a common solution to handle privacy issues in clouds. In this study, we focus on range queries over encrypted data since it is a fundamental operation. Over the last years, different approaches have attempted to strike a trade-off between security and practical efficiency [2, 10, 11]. Index-based schemes [5, 13, 15] have also been proposed to increase query performance while ensuring strong security. Nevertheless, prior schemes cannot cope with the high rate of incoming data that occurs in a wide range of monitoring applications, especially in the proposed context.

To solve the drawback of existing works, we propose a solution based on PINED-RQ [15] that enables the building of a secure index over sensitive data at a trusted component (hereinafter referred to as a collector). The collector then publishes the secure index and the encrypted data to the cloud for serving range queries. Our choice is motivated by the fact that PINED-RQ offers strong privacy protection while it has significantly faster range query processing and requires less storage space, compared to its counterparts [5, 13]. Nonetheless, PINED-RQ has to publish data in batches and partially processes data at the collector. Consequently, a bottleneck may occur as incoming data and query requests arrive at a high rate. Moreover, since PINED-RQ partially evaluates queries at the collector, which often has limited resources, they may also confront scalability problems. Publishing small batches may help PINED-RQ ease those potential problems, however, because it is built upon differential privacy [6, 8], small batches would cause large aggregation noise, destroying index's utilities.

Therefore, in order to adapt PINED-RQ to the targeted context, we aim to shift heavy workload from the collector to the cloud, that is able to provide on-demand capacity. In particular, instead of publishing data in batches, when a new tuple arrives, the collector immediately sends it to the cloud. The challenge to our approach is how to build PINED-RQ's index for the new tuples that are previously moved to the untrusted cloud.

In this paper, we propose PINED-RQ++, an extension of PINED-RQ, to mainly prevent potential bottlenecks at the collector while ensuring a secure index for new data. The key idea behind our prototype is to reverse the process of constructing PINED-RQ's index. This allows the sending of new data to the cloud as soon as possible without sacrificing privacy. The experimental results give promising performance, e.g., the publishing time of the NASA dataset (~0.5M tuples) [1] is reduced up to ~35x while maximum data rate at the collector experiences a reduction of up to ~2.7x. In particular, our solution eliminates query processing at the collector, making the system more scalable. The contributions of this paper are as follows.

(1) We introduce a notion of index template within PINED-RQ to support frequently published data.
(2) We propose a mechanism, PINED-RQ++, for updating the index template while still retaining privacy protection for frequently published data comparable to PINED-RQ.
(3) We also develop a parallel version of PINED-RQ++ to improve the throughput of the system.
(4) We implement (non-)parallel PINED-RQ++ to show the superiority of our solutions compared with PINED-RQ.

The paper is structured as follows. In Section 2, we briefly review background. We then introduce our solution in Section 3. In Section 4, we present our experimental results before giving conclusion and future work in Section 5.

2 BACKGROUND

2.1 Related Work

Various schemes have been developed to preserve privacy for processing range queries in clouds over the last years. Hidden vector encryption approaches [4, 16] use asymmetric cryptography to conceal data's attributes in an encrypted vector. These

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.82

methods incur prohibitive computation costs. Many bucketization schemes [9–11] have been proposed for range query processing in clouds. These solutions partition an attribute domain into a finite number of buckets. The range query retrieves all data falling within the range. However, bucketing approaches disclose data distribution and suffer from large aggregation false positives. Agrawal *et al.* [2] and Boldyreva *et al.* [3] present order-preserving encryption schemes that preserve the relative order of plain data under encryption. A downside of these schemes is that they leak the total order of plain data to the cloud. This is vulnerable to statistical attacks. Meanwhile, Li *et al.* [13] and Demertzis *et al.* [5] propose index-based strategies for answering range queries over outsourced data. Unfortunately, both suffer from prohibitive storage cost. On the other hand, Sahin *et al.* [15] present PINED-RQ for serving efficient range query processing in clouds via secure indexes. Nonetheless, the high rate of new data is not discussed in this work. Based on PINED-RQ, we develop our solution to tackle the limitations of the current works.

2.2 Differential Privacy

Definition 1 (ϵ-differential privacy [6, 8]): A randomised mechanism M satisfies ϵ-differential privacy, if for any set $O \in Range(M)$, and any datasets D and D' that differ in at most one tuple,

$$Pr[M(D) = O] \leq e^{\epsilon}.Pr[M(D') = O]$$

where ϵ represents the privacy level the mechanism offers.

Laplace Mechanism [7]: Let D and D' be two datasets such that D' is obtained from D by adding or removing one tuple. Let $Lap(\beta)$ be a random variable that has a Laplace distribution with the probability density function $pdf(x, \beta) = \frac{1}{2\beta}e^{-|x|/\beta}$. Let f be a real-valued function, the Laplace mechanism adds $Lap(max \parallel f(D) - f(D') \parallel_1 / \epsilon)$ to the output of f, where $\epsilon > 0$.

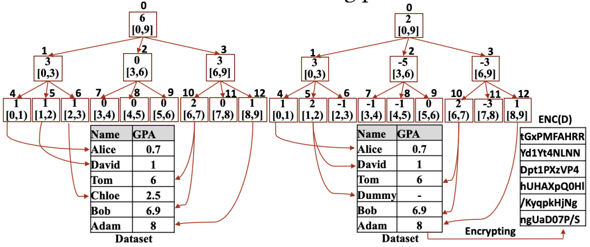
Theorem 1 (Sequential Composition [14]): Let $M_1, M_2, ..., M_r$ denote a set of mechanisms and each M_i gives ϵ_i-differential privacy. Let M be another mechanism executing $M_1(D), M_2(D), ..., M_r(D)$. Then, M satisfies $(\sum_{i=1}^{r} \epsilon_i)$-differential privacy.

2.3 PINED-RQ

We briefly describe PINED-RQ [15], which is built upon differential privacy [6]. We only focus on the insertion operation in this study. There are two main steps for building a PINED-RQ index.

(a) Building an index: Given a dataset at the collector, a PINED-RQ's index is constructed based on a B+Tree. In PINED-RQ, the set of all nodes is defined as a histogram covering the domain of an indexed attribute. For example, the students' GPA is used to build histograms (see Figure 1a). Each leaf node has a count representing the number of tuples falling within its interval. It also keeps pointers to those tuples. Likewise, the root and any internal node have a range and a count, combining the intervals and the counts of their children, respectively.

(b) Perturbing an index: All counts in the index are independently perturbed by Laplace noise [7]. The noise may be positive or negative, thus, after this step, the count of a node may increase or decrease, respectively. As shown in Figure 1b, the count of node 6 changes from 1 to -1 while the count of node 5 changes from 1 to 2. Such changes consequently lead to inconsistencies between a leaf node's noisy count and the number of pointers it holds. To address that issue, PINED-RQ adds dummy tuples (fake tuples) to the dataset as a leaf node receives positive noise. Otherwise, if a leaf node receives negative noise, real tuples are moved from the dataset to the corresponding overflow array. An overflow array [15] of a leaf node is a fixed-size array, which is randomly filled with dummy tuples at the publishing time to conceal removed tuples from the adversary. As illustrated in Figure 1b, the tuple (Chloe) belonging to node 6 is deleted from dataset while one dummy tuple is added and linked to node 5. Finally, the perturbed index is sent to the cloud along with the encrypted dataset. Notably, PINED-RQ satisfies $(\epsilon, \delta)_n$-Probabilistic-SIM-CDP privacy model [15], a variant of differential privacy [8]. Intuitively, this variant results from the introduction of the encryption and the overflow arrays to the index building process.



(a) Clear index **(b) Secure index**

Figure 1: Example of PINED-RQ index

Clearly, an update directly to such published indexes would violate differential privacy [6, 8], thus, PINED-RQ cannot support live updates. Furthermore, PINED-RQ is also reluctant to publish very small datasets since the aggregation noise would destroy index's utilities. These properties make PINED-RQ impractical for high speed monitoring applications. In contrast, our proposal aims to send new data immediately to the cloud. Thus, one technical challenge is how to manage such dummy and removed tuples, which help protect the privacy of the index, as new data are stored at the cloud instead of at the collector.

3 PINED-RQ++

We focus on the architecture as depicted in Figure 2. Data generators produce raw data and send them to a collector. The incoming data are then pre-processed prior to being sent to a cloud. A consumer poses range queries to the cloud. In PINED-RQ++, we assume that the cloud is *honest-but-curious* while the other components are trusted. Thus, the adversary can use all information exchanged between the cloud and the other trusted components to deduce anything in a computationally-feasible way.

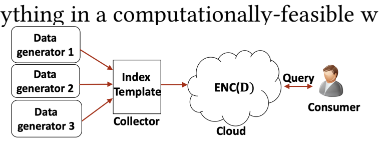


Figure 2: Proposed architecture

At the beginning, an index template is built at the collector. Whenever a new tuple arrives, the index template is updated with that tuple. Next, the tuple is encrypted and forwarded to the cloud. When the index template is published at a later time, the cloud associates it with unindexed data to produce a secure index as described in Section 2.3. The collector then initiates a new index template for future incoming data.

A query is only processed at the cloud which holds both indexed and unindexed data at time. As a result, as a consumer issues a query, it is first evaluated on indexed data (as in PINED-RQ's query processing [15]), the result of this evaluation and all the unindexed data are returned to the consumer. Finally, the consumer decrypts and filters the returned data for the final results.

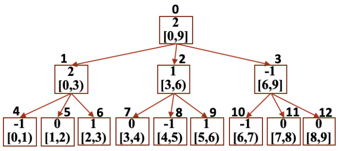
3.1 Index Template

The process of building an index template is typically the same as in PINED-RQ (see Section 2.3). However, since initially there are no data, its count variables only contain Laplace noise [7] and its leaves have no pointers. Such count variables and pointers are updated during a publishing time interval, which is defined as the period from when an index template is initiated to when it is published. This poses several challenges to our approach, for instance, how to publish dummy tuples generated during the index template building process or how to ensure that pointers between leaves and the new data do not leak privacy. Section 3.2, and 3.3 discuss these challenges as well as possible solutions.

3.2 Matching Table

Recall that when an index template is published, the cloud associates it with unindexed data to form a PINED-RQ's index for those data. To prepare for this association, the collector needs to keep the pointers between unindexed data and leaves. To do that, a simple way is to mark the ciphertext of a new tuple by the id of the leaf node to which the tuple belongs, and send the marked ciphertext to the cloud. Later, the cloud can rebuild pointers from marked ciphertexts when the index template is published. However, these marked ciphertexts reveal the real pointers between unindexed data and leaves during a time interval. PINED-RQ++ consequently discloses more extra information, e.g., the actual distribution of the incoming time of real data, when compared to PINED-RQ.

To prevent the leakage of such information, we use unique random numbers which are viewed as temporary ids of tuples and a matching table (see Figure 3). The first column in this matching table stores leaves' id while each row of the second column holds the temporary id of tuples belonging to the corresponding leaf node. For instance, tuples 1 and 7 belong to node 6 while tuple 5 belongs to node 9. In particular, when a tuple arrives, the collector encrypts it, generates a unique random number, and sends the *<random number, ciphertext>* pair to the cloud. This number is stored in the corresponding row in the matching table at the collector. The randomness guarantees that no useful information about the index template is leaked to the adversary. When an index template and its matching table are published, the cloud simply loops over the matching table and replaces random numbers with the leaves' pointers.



Matching Table

Leaf node	Tuple IDs
4	{}
5	{}
6	{1,7}
7	{}
8	{}
9	{5}
10	{}
11	{}
12	{}

ID	Tuples
1	tGxPMFAHR
5	Yd1Yt6NLNN
7	Mpt0POzMR

Figure 3: Perturbed index template and matching table

3.3 Noise Management

One challenge to our approach is that the collector initiates and perturbs the index template without any existing data. This means that no tuples are available to be deleted in case of negative noise. Also, when leaves receive positive noise, the collector can initially generate dummy tuples. However, when those dummy tuples are published to ensure privacy protection must be considered. To this end, we present two approaches as follows.

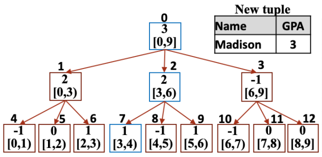
Regarding positive noise, the collector can immediately generate and send dummy tuples to the cloud at any random time point within a time interval. However, the arrival of all dummy tuples at the same time could be unsafe since the distribution of arrivals may be exploited by an adversary. Instead, we randomly release dummy tuples over the time interval. For instance, given 10 dummy tuples and a time interval of 100ms, then 10 discrete points can be randomly chosen between 1 and 100. At each chosen time point, a dummy tuple will be published along with a unique random number. It is, however, true that the privacy would be leaked as a dummy tuple might arrive at a chosen time at which real tuples are improbable. To avoid that case, the collector sends dummy data according to the actual distribution of the sending time of the real tuples. With this approach, when a *<random number, ciphertext>* pair comes to the cloud, the adversary cannot distinguish which pair is dummy or real data.

For negative noise, if a leaf node initially receives negative noise c, the collector moves the first c tuples (when they arrive) of that leaf node to the corresponding overflow array. At publication time, the collector randomly fills all overflow arrays with dummy tuples and sends these overflow arrays to the cloud. Notably, the movement of tuples only occurs at the collector and the adversary does not know which nodes receive negative noise. Thus, the privacy of such movements is also preserved.

3.4 Index Template Update Management

The main goal is to guarantee that the counts of PINED-RQ++'s index template are the same as those of PINED-RQ's index when the index template is published. In PINED-RQ, a leaf node's count represents the number of tuples falling within its interval. The count of internal nodes and the root is a summation of their children's counts. All counts are then perturbed by noise. In contrast, an index template only contains noise at first and it will increase its counts as soon as a tuple arrives at the collector.

Basically, when a tuple arrives at the collector, the leaf node to which the tuple belongs is determined. Then, the count of that leaf node and all its ancestors will be increased by 1. As shown in Figure 4, as the new tuple <Madison, 3> has GPA lying within the interval of node 7, the count of node 7 and all its ancestors (node 2 and node 0) are increased to 1, 2 and 3, respectively.



New tuple

Name	GPA
Madison	3

Matching Table

Leaf node	Tuple IDs
4	{}
5	{}
6	{1,7}
7	{9}
8	{}
9	{5}
10	{}
11	{}
12	{}

ID	Tuples
1	tGxPMFAHR
5	Yd1Yt6NLNN
7	Mpt0POzMR
9	Gp=0MOzLR

Figure 4: Index template and matching table are updated after the arrival of a new tuple

3.5 Parallel PINED-RQ++

Since the collector, that is assumed to be a small private node with a few cores, updates the index template, its throughput would be impacted as the rate of incoming tuples increases. We therefore parallelize the construction of the index template to improve the collector's throughput. Instead of keeping only one index template for updating, we create many clones of the original, each of which is independently updated. As a result, incoming tuples are equally distributed to clones for local updating. At publish time, all clones are merged together and sent to the cloud.

3.6 Privacy Analysis

As compared to PINED-RQ, both PINED-RQ++ and its parallel version only leak extra information of *<random number, ciphertext>* pairs and the time, when such pairs arrive at the cloud, to the adversary. Random numbers will not disclose any information about the data. Besides, as discussed in Section 3.3, when a pair arrives at the cloud, an adversary cannot distinguish between a dummy and a real tuple. Thus, PINED-RQ++'s privacy protection is similar to that of PINED-RQ.

4 EVALUATION RESULTS

4.1 Benchmark Environment

We ran our experiments on a cluster, whose configuration is illustrated in Table 1.

Table 1: Experimental environment

Component	CPU (2.4 GHz)	Memory (GB)	Disk (GB)
Collector	12	16	20
Cloud	16	16	40
Data generator	4	8	80
Consumer	4	16	10

We evaluate our proposal on four metrics namely network traffic, the time needed to publish an index (template), time response latency, and throughput. We use two real datasets NASA log [1] (1569898 tuples, five attributes) and Gowalla [12] (6442892 tuples, three attributes) for our experiments. We use the reply byte and check-in time as indexed attributes, respectively. Based on the values in the datasets, the reply byte's domain is divided into 350 bins while that of check-in time into 1502 bins. The fanout is set to 16. We use a time interval of 1 minute.

4.2 Results

(a) Network traffic: The network traffic metric gives an idea of the stability of the overall system, which is crucial for analytical processing. In this scenario, the data generator sends 3K tuples/second. Network traffic in terms of data rate is monitored at the collector over ten minutes. Figure 5 shows that the network traffic in PINED-RQ++ is much more stable than that in PINED-RQ. The maximum data rate is reduced by up to ~2.7x (NASA) and ~2.5x (Gowalla) in PINED-RQ++.

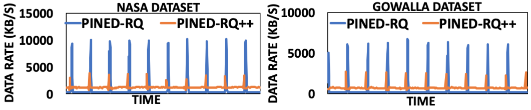


Figure 5: Network traffic over a ten minutes period

(b) Publishing time: We compare the time required to publish an index (template) according to dataset size. This metric is essential for monitoring applications since a long delay may cause bottlenecks at the collector. Different sizes of datasets are obtained by adjusting the incoming data speed and time interval parameter. As shown in Figure 6, when the dataset size increases, the time gradually rises in PINED-RQ while the publishing time in PINED-RQ++ remains almost unchanged. In particular, the publishing time is reduced by up to ~35x for NASA (514972 tuples) and ~16x reduction for Gowalla (1116907 tuples). Notably, when the dataset size rises, the gap between the two prototypes goes up.

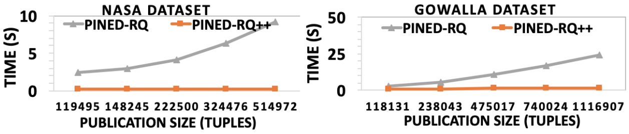


Figure 6: Average publishing time of 10 datasets

(c) Response time latency: We turn our attention to query latency. The data generator produces 2K tuples/second. The consumer sends one query per second. The query range is randomly chosen between 25% and 100%. In Figure 7, the results indicate that PINED-RQ has lower latency for small ranges compared to PINED-RQ++. However, our approach performs slightly better with large ranges (100%) because PINED-RQ's collector experiences higher workload for processing consumers' queries.

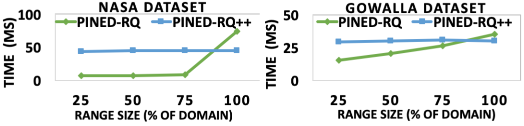


Figure 7: Average response time latency of 1000 queries

(d) Throughput: We compare the parallel PINED-RQ++'s throughput with the non-parallel version. The incoming data rates are chosen to be higher than the maximum throughput of the non-parallel version, 6.5K and 18K tuples/second for NASA and Gowalla, respectively. The different data rates are chosen since the tuple size of NASA is larger than that of Gowalla. The parallel version always has better throughput than the non-parallel version. The results in Figure 8 show that when the number of clones increases, the throughput is improved. The two-clone setting gives the best throughput, increasing by about 18% (to ~3K tuples/second) for NASA and about 47% (to ~8.3K tuples/second) for Gowalla when compared to the non-parallel version. Using a larger number of clones is not meaningful due to the merging process's costs.

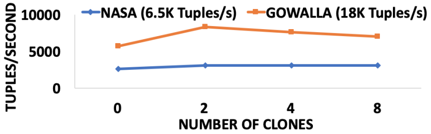


Figure 8: Parallel PINED-RQ++

5 CONCLUSION

We developed PINED-RQ++ to address the challenges of high rates of incoming data for processing range queries in clouds and the scalability problems of the prior schemes. The experimental results show that our solution provides better performance than PINED-RQ while privacy is also protected. This proves that PINED-RQ++ is appropriate for real-world monitoring applications. Besides, we introduce the parallel version, that helps enhance throughput.

Future work includes improving query performance, caching techniques and a dynamic adaptation to the query workload.

REFERENCES

[1] 1996. NASA Log. (1996). http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html

[2] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order Preserving Encryption for Numeric Data. In *SIGMOD*. 563–574.

[3] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. 2011. Order-preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *CRYPTO*. 578–595.

[4] Dan Boneh and Brent Waters. 2007. Conjunctive, Subset, and Range Queries on Encrypted Data. In *TCC*. 535–554.

[5] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos Garofalakis. 2016. Practical Private Range Search Revisited. In *SIGMOD*. 185–198.

[6] Cynthia Dwork. 2006. Differential Privacy. In *ICALP*. 1–12.

[7] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *TCC*. 265–284.

[8] Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Found. Trends Theor. Comput. Sci.* 9, 3&#8211;4 (2014), 211–407.

[9] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad . 2002. Executing SQL over Encrypted Data in the Database-service-provider Model. In *SIGMOD*. 216–227.

[10] Bijit Hore, Sharad Mehrotra, Mustafa Canim, and Murat Kantarcioglu. 2012. Secure Multidimensional Range Queries over Outsourced Data. In *VLDB*. 333–358.

[11] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. 2004. A Privacy-preserving Index for Range Queries. In *VLDB*. 720–731.

[12] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

[13] Rui Li, Alex X. Liu, Ann L. Wang, and Bezawada Bruhadeshwar. 2014. Fast Range Query Processing with Strong Privacy Protection for Cloud Computing. *In VLDB* (2014), 1953–1964.

[14] Frank D. McSherry. 2009. Privacy Integrated Queries: An Extensible Platform for Privacy-preserving Data Analysis. In *SIGMOD*. 19–30.

[15] Cetin Sahin, Tristan Allard, Reza Akbarina, Amr El Abbadi, and Esther Pacitti. 2018. A Differentially Private Index for Range Query Processing in Clouds. In *ICDE*. 857–868.

[16] M. Wen, R. Lu, K. Zhang, J. Lei, X. Liang, and X. Shen. 2013. PaRQ: A Privacy-Preserving Range Query Scheme Over Encrypted Metering Data for Smart Grid. *IEEE Transactions on Emerging Topics in Computing* 1, 1 (June 2013), 178–191.

Towards Augmented Database Schemes by Discovery of Latent Visual Attributes

Visionary

Tomáš Grošup, Ladislav Peška, Tomáš Skopal
SIRET Research Group, Faculty of Mathematics and Physics,
Charles University, Prague, Czech Republic
{grosup,peska,skopal}@ksi.mff.cuni.cz

ABSTRACT

When searching for complex data entities, such as products in an e-shop, relational attributes are used as filters within structured queries. However, in many domains the visual appearance of an item is important for a user, while coverage of visual appearance by relational attributes is left to database designer at design time and is by nature an incomplete and imperfect representation of the entity. Recent advances in computer vision, dominated by deep convolutional neural networks (DCNNs), are a promising tool to cover the gaps. It has been shown that activations of neurons of DCNNs correspond to understandable visual-semantic features of an input image. We envision that activations of neurons are of great use for search queries in domains with strong visual information, even when obtained from DCNNs models pre-trained on general imagery. Locally scoped visual features obtained using them can be combined to form search masks which would correlate to what humans understand as an attribute, when applied on the entire dataset. Ultimately, combination of visual features can be identified automatically and formed into immediate suggestion of a new relational attribute, leaving one last task for humans to turn this into augmentation of the database schema – putting a label on it.

1 INTRODUCTION

Many approaches to information retrieval tasks, e.g., querying or exploration, assume structured data entities. These are modelled using composition of simple attributes and relations to other structured entities and as such, they represent a simplified model of the real world entity. Object's attributes either comply to a fixed schema in relational database model, or there is an implicit schema defined by existence of attributes in non-relational models such as document databases, key-value stores or linked data. Even though some data representations (e.g., JSON, XML) might be referred as schema-less and compliance is not always enforced, applications are working against an expected structure of the data and their attributes. In either case, these attributes are defined in a supervised manner by domain experts.

As an example application, let us consider an e-shop with a search interface. Within its search interface, users can define structured queries, which operate on a pre-defined set of attributes, utilizing range filters, exact matching or similar filtering mechanisms. An example query might be "*category = 'shoes' AND price between 100 and 200 AND color = 'black' AND description matches 'summer'*". Usability of the querying mechanism is limited both by the expressiveness of the database schema as well as capability of the user. In many domains (e.g., art, cars, dating, decorations, fashion or furniture), schema expressiveness is rather inferior due to the low information gain of available structured data. Therefore, the final filtering step of a retrieval task has to be executed by a human via examination of non-structured data (images, audio, video, etc). Scientific communities have been trying to reveal the internal structures of multimedia content in the last decades with gradually improving results. However, disclosing the multimedia object's representation in a way comprehensible for end-users remains extremely challenging. Partial solutions of this challenge are query-by-example search or multimedia exploration paradigms, accompanying the classical structured search as a refinement step.

Recent advances in neural network architectures, such as deep convolutional neural network AlexNet [8] represent a promising direction in revealing the internal structure of multimedia. Due to their layered nature, learned concepts are more high-level, and can bridge the semantic gap in computer vision. Although the task of many pioneer architectures was to classify data into a fixed set of pre-defined categories, it has been shown that the trained models can generalize to unknown domains, if the activation of neurons is used as a high-dimensional descriptor of objects [3]. Existing network models can also be fine-tuned to operate on a finer subset of data, increasing the precision for a particular dataset [4]. Finally, a layered networks' architecture allows to generate descriptors on different levels of a semantic scale [13, 19], ranging from low-level visual features (e.g., edges) up to high-level concepts, such as a cat. It is also possible to focus on individual regions of an image, denoted as patches [5, 18].

In our vision, we propose to extract information encoded in these descriptors via identification of commonly occurring combination of features. In order to analyze the behaviour of descriptors in the domain of product imagery, we implemented a multi-example similarity search mechanism [15]. This was also used to better understand applicability of different layers of the network by evaluating precision of their descriptors against manually selected proposals of new attributes acting as a test dataset. In this first step, the similarity was evaluated only globally, by extracting and comparing descriptors representing the whole object. As a second step, evaluation of individual image parts, patches, was done [14]. Human users were tasked to manually highlight interesting part of the image when selecting a set of images as an input for the search. Selected parts of an image were powering the search mechanism, which operated on an aggregation of global representations of images as well as on individual descriptors of each image patch, matching patches in the input query and in the overall database. This was partly done as an analytical step to better understand the behaviour and patterns within the data and how they correlate to what humans understand as an attribute.

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

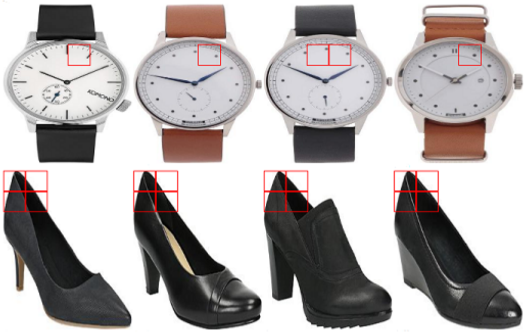


Figure 1: Examples of two proposed attributes. First row are "hand watches without labels". Second row are "lady shoes with sharp corner in the ankle area".

In the final stage of our vision, patterns of descriptors occurring frequently in the dataset can be automatically suggested as candidates for new attributes, immediately augmenting the original database schema (see Figure 1). Once the schema is augmented, all of the existing applications using it as a building block, e.g., recommender systems, data analytics, search functions and many others, can utilize this augmentation and therefore cummulatively increase the added value of the newly discovered "virtual" attribute. This implies benefits from an overall data-engineering perspective, and not just an application-specific improvement of a search function.

2 RELATED WORK

To the best of our knowledge, there have been no attempts to augment database schema using multimedia features. However, there are many similar disciplines that were used as inspiration and many technologies that could be used as building blocks to help implementing the vision. We want to highlight that rather than solving a single application-specific task (e.g., using images for product recommendation) we define the problem from database-engineering perspective, so that each service utilizing a database schema could benefit from its refinement.

Deep learning approaches have been evolving in the last years. Recent advances include inception modules (Googlenet), residual networks (ResNet, ResNeXt, DenseNet), and meta-learning approaches for searching an optimal architecture automatically (NAS), with better architectures being proposed every year. The target of our vision is to work with any network pre-trained on general imagery, making the choice of network architecture orthogonal to the main problem.

One of anticipated outcomes of our vision is an improvement of product recommendation techniques. Several papers focusing on fashion recommendation were published recently [6, 7, 9, 12, 17]. In fashion domain, visual-similarity based recommending approaches managed to considerably improve conversions of visitors to buyers [12]. In [6, 7, 12], authors focused on similarity and identity matching of "wild" images (i.e., pictures taken on streets or non-canonical representations of objects), while our scenario allows us to restrict on canonical representations of objects shot from similar angles with uniform background, etc.

Somewhat similar to our work is the approach by Yu et al. [17] on utilization of aesthetic-based features of objects together with common CNN features. However, using sole aesthetic features did not improve over CNN features and therefore we kept focusing on local visual similarity of objects instead of their aesthetics. The main difference between our vision and mentioned related work is the aim on materializing relevant similarity pattern instead of just utilizing them as a part of similarity calculations.

The topic of schema augmentation is not new, however, it has not been targeted via latent visual information so far. Selke et al. [11] focused on crown-enabled databases, which could enrich database schema at the query-time by utilization of manual labor to power search execution. Yakout et al. [16] automatically augmented entities via scraping of online available HTML tables and matching their information.

3 ROAD MAP

In this section, we overview the main points of our vision, describe the context, where it brings the most benefit, and define individual milestones and already completed goals, which altogether form a road map towards turning the vision into reality.

3.1 Vision

The body of our vision is to provide global as well as a user-based schema extensions by virtual attributes not present in the original database schema. Virtual attributes are to be derived from a visual information in domains, where it can provide a significant information gain, e.g., fashion, art, decorations, furniture, etc. We do not meant to replace classical search or recommendation methods, but rather boost them by providing additional relevant features. In addition to the applications in automated content processing methods, virtual attributes (if properly labeled) can be revealed to the end-users or utilized in data analytics.

Implicit feedback collected in an application-specific scope, such as browsing history in an e-shop, can be used as crowd-based evidence for the suggested attributes. If an attribute correlates with contents of search history, it is likely to have a meaning to humans. The evidence could naturally grow from single random visitors, over long-term users, up to large groups of similar users and thus increasing the level of confidence for the suggestion.

A particular objective aims at straightforward applicability for general usage; not relying on extensive training phase and pre-existing definitions of attributes/classes. The main challenge relies in an operation on previously unseen and undefined attributes, and an evolution of the model at runtime. Generally, training images and classes are essential for today's state-of-the-art computer vision models such as neural networks. However, the real world has unlimited number of classes assignable to objects, and it is impossible to prepare such data upfront. At the same time, requiring a special design and training phase would implicitly mean a need for a trained machine learning specialist, which limits the benefits to a small portion of data owners. We believe that it is possible to finalize the envisioned method in a set of unsupervised-pipeline steps, leading to a deployment-enabled black-box component which can be integrated into a broad range of today's applications.

3.2 Context

The envisioned approach of automatically detecting attributes is not guaranteed to be beneficial within all domains, not even when restricting the idea to products with visual information. The main preconditions are that visual information is an important factor influencing users' decisions and that information gain of existing relational attributes is low compared to information retrieved from multimedia data. This condition disqualifies product domains such as computer components, where the visual information is available, but existing relational attributes cover most of the users' information needs.

The applicability of our approach rises in scenarios, where objects may contain multiple important visual attributes and therefore it is not possible to simply label objects w.r.t. some class hierarchy. It is the dynamic point of view making it a challenge, since different users looking at the exact same image might have different understanding what is the attribute of interest for them. For example, several virtual attributes, e.g., leather-build, high heel, floral texture and black zipper may be detected for a single shoe, however for various users, only some of the attributes may be relevant in their decisions.

3.3 Milestones

In the following we summarize the milestones of the road map, also sketched in Figure 2.

Descriptor extraction (a). As a first step of our vision, proper image descriptors need to be selected. However, due to the broad range and complexity of today's techniques, it is almost impossible to evaluate all possible variants and combinations. Even with simple DCNN like AlexNet, there are several layers to choose from and different aggregation methods to turn convolutional layers into a single vector. We have evaluated performance of different layers in [15] with the conclusion that there is no single dominating layer for all search tasks. Therefore, this aspect still deserves more research attention. Nonetheless, as all further tasks are independent on the choice of image descriptors, it is plausible to start with some simple selections, e.g., based on AlexNet and research this area in parallel with further tasks.

Image patches (b) are regions of an image that enable to focus on a particular part of it and, as a consequence, evaluate its similarity locally. Within standardized product datasets, it can be sufficient to cut images using regular grids, since all images have same orientation, centering and background. If we move towards "images in the wild" scenario, some trainable image segmentation approach might be necessary. In either case, the final image representation is a list of patches' descriptors. Another challenge in this task is to define methods to aggregate similarity w.r.t. different patches. We already reported results of some basic aggregation methods in [14].

Similarity sets (c), defined by image patches that are similar to each other in a given feature-space, could be the first step towards a virtual attribute. There are multiple ways to model similarity between image patches, e.g., by evaluating only the distance between their respective descriptors or by weighting it together with global similarity of the entire entity.

Noise removal (d) in the area of similarity sets is a challenging problem given the unsupervised nature of the desired pipeline, i.e., the only available information are images, their descriptors, and similarity values. The distribution of distance values can be a useful heuristic to filter out sets which are unlikely to form a good attribute, such as near-duplicates (distance close to zero), trivial patches (small distance to single-color patches) or sets being too large (attributes would no longer be discriminatory). Another possibility is to eliminate noise by cross-checking data coverage by existing relational attributes, e.g., to remove suggestion for an attribute that is already known in the data and correlates highly with the new suggestion.

Frequent patterns (e) within the filtered similarity sets should be prioritized in order to provide a ranked list of suggested attributes to the domain administrators. Existing approaches to the frequent patterns mining, such as the Market Basket Analysis [1], could be utilized. The algorithm should be optimized to reduce the number of suggested patterns/rules, while preserving a good coverage of entities in the database and a reasonable confidence level that can be used for sorting purposes.

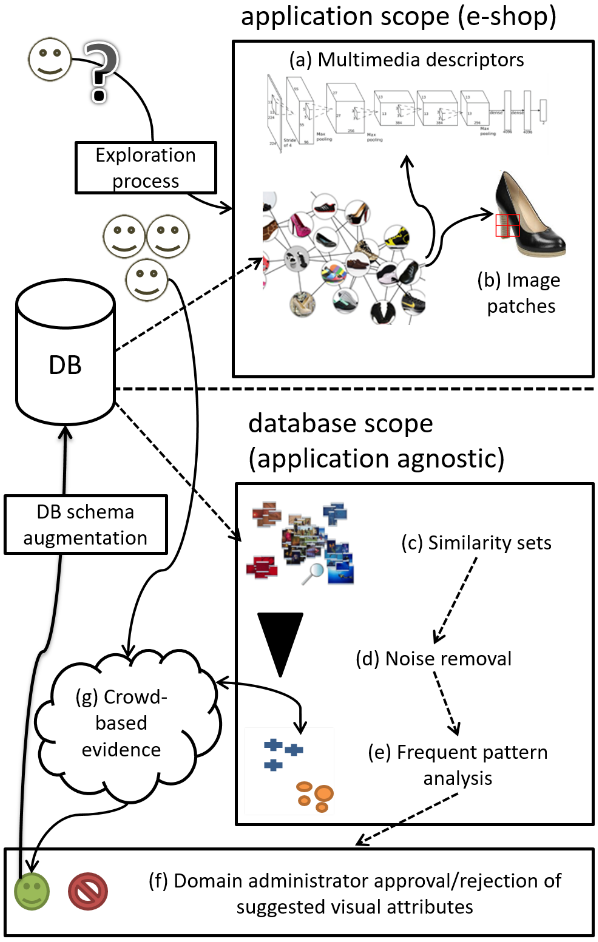

application scope (e-shop)
(a) Multimedia descriptors
Exploration process
(b) Image patches
DB
database scope (application agnostic)
DB schema augmentation
(c) Similarity sets
(d) Noise removal
(g) Crowd-based evidence
(e) Frequent pattern analysis
(f) Domain administrator approval/rejection of suggested visual attributes

Figure 2: Road map/architecture of the vision.

Domain administrator (f) could be presented with n-tuples of items that match a suggested attribute. Furthermore, if the feedback from users is available (e.g., their browsing history or shopping baskets), frequently co-occurring items possessing a suggested attribute may be prioritized. The interface could also highlight the features on which the similarity of the set is based, e.g., by highlighting relevant image patches. If the suggestion gets approved and labeled, the similarity set could be turned into a search mask, evaluated w.r.t. the full database of descriptors to turn it into a new attribute. Since the database of descriptors contains also individual patches, certain attributes could be detected multiple times within a single object (a 1:N relation).

Collaborative techniques (g) based on implicit user feedback could not only refine suggestions to the domain administrators, but could also establish sets of latently approved virtual attributes specific for a cluster of users, or a single long-term user. In such a way, we may postpone the work of domain administrators and label the attributes only upon a request from a component that disclose attributes to the users (e.g., data analytics or attribute search). Challenges of this task are balancing the granularity of collected feedback (e.g., a single user, top-k similarity, user clustering), maintaining different augmented schemes for multiple user scopes and proposing methods capable to aggregate more granular augmented schemes (e.g., on a level of single users) and bring them to the higher level, eventually reaching the main database schema.

3.4 Challenges

In this section, we summarize the major challenges of the proposed vision and offer ideas on how to address them. The challenges are evaluation, noise reduction, scalability, personalization, continuous schema evolution and applicability on other forms of content.

By far the most significant challenge for database augmentation is the lack of an established evaluation framework and standardized datasets. The nature of human-system interaction will require time-consuming user studies in order to collect sufficient training data and feedback. The evaluation metrics could measure the overall user satisfaction, impact on speed in information retrieval tasks, as well as indirect impact obtained due to improved recommendation and search, e.g., the conversions ratio. Also, the practically unbound volume of possible attribute suggestions and the lack of training data for ranking the candidates makes human-in-the-loop a critical part of the evaluation.

The second challenge is the reduction of noise. The amount of attributes within a dataset is subject to a combinatorical explosion of common patterns across the dataset. The system can verify their meaningfulness using automated techniques operating on similarity data and visual descriptors, but it might be very difficult to systematically estimate if a pattern is indeed a new attribute, or if it was a random set of data points.

In order to deal with a large volume of possible patterns, the system must be efficient and scalable in all parts of data processing pipeline. In the existing work, the MapReduce paradigm was used in the implementation, allowing the computation to run on a large cluster of machines. The computationally intensive step of similarity self-join identifying common patterns across the dataset, was executed using the Hadoop MapReduce algorithms of Čech et al. [2].

The fourth challenge is personalization. Whenever the system starts inferring new attributes based on the implicit feedback input, the correct scope must be identified. The trivial cases are single-user scoped and globally scoped attributes. However, collaborative approaches may consider various, possibly overlapping clusters of similar users. This would require more extensive verification of an attribute across different user clusters, in order to estimate its benefit for the respective user groups.

The fifth challenge is the continuous schema evolution, fostered by a stream of new attributes, and its propagation into classical database tasks, e.g., similarity search. Therefore, the inclusion of new attributes must be done in a transparent way, so the successive techniques can automatically incorporate it without explicit intervention. Transitively, this has also impact on all other components operating on a set of relational attributes, such as data exploration or recommendation engine.

The last challenge is to generalize the proposed model into other forms of multimedia, such as sound or video. This might be appealing in domains, where short video material represents an inherent part of information retrieval tasks. One example might be a database of movies with their trailers. Recently, there have been significant improvements in video browsing techniques, such as the work of Lokoč et al [10]. The proposed schema augmentation approach could be beneficial in such scenarios as well.

4 CONCLUSION

This vision paper outlines a novel research problem, which aims at augmenting database schemes by attributes extracted from visual information. Initial attempts have outlined a possible direction for future research and identified several sub-problems and challenges to solve.

Acknowledgments

This research has been supported by Czech Science Foundation (GAČR) project Nr. 17-22224S.

REFERENCES

[1] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. 1997. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD '97)*. ACM, New York, NY, USA, 255–264.

[2] Přemysl Čech, Jakub Maroušek, Jakub Lokoč, Yasin N. Silva, and Jeremy Starks. 2017. Comparing MapReduce-Based k-NN Similarity Joins on Hadoop for High-Dimensional Data. In *Advanced Data Mining and Applications*. Springer, 63–75.

[3] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. 2013. DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition. *CoRR* abs/1310.1531 (2013). arXiv:1310.1531

[4] Z. Ge, C. McCool, C. Sanderson, and P. Corke. 2015. Subset feature learning for fine-grained category classification. In *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 46–52.

[5] Xufeng Han, T. Leung, Y. Jia, R. Sukthankar, and A. C. Berg. 2015. MatchNet: Unifying feature and metric learning for patch-based matching. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 3279–3286.

[6] J. H. Hsiao and L. J. Li. 2014. On visual similarity based interactive product recommendation for online shopping. In *2014 IEEE International Conference on Image Processing (ICIP)*. 3038–3041. https://doi.org/10.1109/ICIP.2014.7025614

[7] M. H. Kiapour, X. Han, S. Lazebnik, A. C. Berg, and T. L. Berg. 2015. Where to Buy It: Matching Street Clothing Photos in Online Shops. In *2015 IEEE International Conference on Computer Vision (ICCV)*. 3343–3351.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 1097–1105.

[9] Nick Landia. 2017. Building Recommender Systems for Fashion: Industry Talk Abstract. In *Proceedings of the Eleventh ACM Conference on Recommender Systems (RecSys '17)*. ACM, 343–343. https://doi.org/10.1145/3109859.3109929

[10] J. Lokoc, W. Bailer, K. Schoeffmann, B. Muenzer, and G. Awad. 2018. On influential trends in interactive video retrieval: Video Browser Showdown 2015-2017. *IEEE Transactions on Multimedia* (2018).

[11] Joachim Selke, Christoph Lofi, and Wolf-Tilo Balke. 2012. Pushing the Boundaries of Crowd-enabled Databases with Query-driven Schema Expansion. *Proc. VLDB Endow.* 5, 6 (Feb. 2012), 538–549.

[12] Devashish Shankar, Sujay Narumanchi, H A Ananya, Pramod Kompalli, and Krishnendu Chaudhury. 2017. Deep Learning based Large Scale Visual Recommendation and Search for E-Commerce. (2017). arXiv:1703.02344

[13] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2013. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. *CoRR* abs/1312.6034 (2013).

[14] Tomás Skopal, Ladislav Peska, and Tomás Grosup. 2018. Interactive Product Search Based on Global and Local Visual-Semantic Features. In *Similarity Search and Applications - 11th Int. Conference, SISAP 2018, Lima, Peru*. 87–95.

[15] Tomáš Skopal, Ladislav Peška, Gregor Kovalčík, Tomáš Grosup, and Jakub Lokoč. 2017. Product Exploration Based on Latent Visual Attributes. In *Proc. of the 2017 ACM on Conference on Information and Knowledge Management (CIKM '17)*. ACM, 2531–2534. https://doi.org/10.1145/3132847.3133175

[16] Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. 2012. InfoGather: Entity Augmentation and Attribute Discovery by Holistic Matching with Web Tables. In *Proc. of the 2012 ACM SIGMOD Int. Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 97–108.

[17] Wenhui Yu, Huidi Zhang, Xiangnan He, Xu Chen, Li Xiong, and Zheng Qin. 2018. Aesthetic-based Clothing Recommendation. In *Proceedings of the 2018 World Wide Web Conference (WWW '18)*. Int. World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 649–658.

[18] Sergey Zagoruyko and Nikos Komodakis. 2015. Learning to Compare Image Patches via Convolutional Neural Networks. *CoRR* abs/1504.03641 (2015).

[19] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and Understanding Convolutional Networks. In *Computer Vision – ECCV 2014*, David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars (Eds.). Springer, 818–833.

Short Paper

Workload-Driven and Robust Selection of Compression Schemes for Column Stores

Martin Boissier
Hasso Plattner Institute
Potsdam, Germany
martin.boissier@hpi.de

Max Jendruk
Hasso Plattner Institute
Potsdam, Germany
max.jendruk@hpi-alumni.de

ABSTRACT

Modern main memory-optimized column stores employ a variety of compression techniques. Deciding for one compression technique over others for a given memory budget can be challenging since each technique has different trade-offs whose impact on large workloads is not obvious. We present an automated selection framework for compression configurations. Most database systems provide means to automatically choose a compression configuration but lack two crucial properties: The compression selection cannot be constrained (e.g., by a given storage budget) and robustness of the compression configuration is not considered. Our approach uses workload information to determine robust configurations under the given constraints. The runtime performance of the various compression techniques is estimated using adapted regression models.

1 COLUMN COMPRESSION IN HYRISE

Two of the main driving forces of current database development – both industrial and research – are autonomous database systems and cloud-based installations. Both topics are strongly connected as database vendors are increasingly interested in optimizing their operational costs for large self-hosted database installations.

One way to lower the costs – especially for main memory-optimized database systems – is to reduce the memory consumption of large databases. Such a reduction allows storing databases on smaller and thus less expensive server machines or adding more instances to a shared server. But the sheer size of large cloud installations hampers manual optimization of compression configurations by database administrators. This development has recently sparked the research on autonomous database systems.

The work presented in this paper is an intermediate step to approach the issue of optimizing memory consumption while still retaining the performance advantages of main memory-optimized databases. When cost considerations are gaining importance, the optimization objective for compression configurations is less runtime performance rather than to retain the current runtime performance while minimizing the storage requirements. With the goal of automatically finding a compression configuration for a given memory budget, this project intends to provide the building blocks for autonomous systems.

The area of data compression has been thoroughly studied for decades in database research. Virtually all modern database systems implement various techniques to compress data and most commercial systems further provide means to adjust the compression level (e.g., Oracle's declarative policies for the *automatic data compression* (ADO), cf. [12], or SQLServer's *database engine tuning advisor* (DTA), cf. [11]). However, we see two distinct issues that remain open from a research perspective: **(i)** workload- and constraint-based compression configurations and **(ii)** determination of configurations whose runtime performance is robust to changing workloads.

We present and discuss the three main components in our research database Hyrise [10] with which we approach workload-driven and robust compression configurations:

- We introduce Hyrise's compression framework which implements an efficient and maintainable interface for various column compression techniques (Section 2).
- We present our runtime estimation, which predicts the performance of compression techniques (Section 3).
- We discuss the applicability of existing approaches for the optimization of physical database designs and how they perform for the task of compression selection (Section 4).

2 COLUMN COMPRESSION FRAMEWORK

Virtually every database management system for hybrid transactional and analytical processing (HTAP) employs a variety of compression schemes. Besides the advantage of reducing the main memory footprint, light-weight compression can even improve runtime performance, e.g., by reducing the memory traffic (cf. [2, 4]) or broadening applicability of vectorization (cf. [17]).

But supporting a variety of compression schemes is challenging as it needs to balance maintainability and efficiency. Most existing approaches optimize either (i) for performance while hampering maintainability and increasing complexity or (ii) provide unified interfaces for improved maintainability which potentially introduces runtimes issues.

2.1 Hyrise's Storage Concept

Hyrise is a main memory-optimized database with a column-major storage format [10]. Each table in Hyrise is horizontally partitioned into n chunks with a predefined maximum size. Each attribute of a table is hence distributed over all chunks whereby a column in a chunk is referred to as a segment. Modifications (i.e., insertions or MVCC-enabled updates) are appended to the most recent mutable chunk. When this chunk reaches its size limit, the chunk is considered immutable and a new mutable chunk is created. Immutable chunks might be compressed asynchronously. Hyrise encodes and compresses segments independently.

2.2 Balancing Performance and Maintainability

There are multiple approaches to integrate column compression schemes into the database system. One is to decompress vectors before an operator accesses the data, eliminating the need to handle different compression schemes in every operator. While this approach might be sufficient for analytical purposes which

Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.84

are dominated by sequential operations, it is not feasible for HTAP processing where single row accesses are frequent.

Another approach is to adapt operators so that they can directly execute on compressed data using late materialization. This approach lowers the memory bandwidth required and further allows to exploit encoding-specific optimizations (e.g., early exiting a predicate when the searched value does not exist in the dictionary). While this approach promises the best performance, it also increases maintenance efforts significantly.

The middle ground between the upfront decompression of segments and encoding-specialized operators are abstraction layers that provide a unified interface to access data (cf. [2, 14]). However, this approach usually introduces dynamic polymorphism and thus virtual method calls per tuple, which are prohibitively expensive in analytical scenarios [5]. Dynamic polymorphism adds instructions, impedes cache utilization, and further hinders compilers to automatically apply vectorization primitives.

2.3 Integrating Compression Schemes

To overcome the issues mentioned in the previous section, Hyrise implements an efficient abstraction layer that provides a unified interface while still allowing encoding-specific optimization when desirable. The implementation uses *zero-cost abstractions* based on C++ metaprogramming and templating.

The column compression framework handles both encoding as well as decoding of compressed segments. The framework separates the various concerns by splitting data storage, encoding, and decoding into separate components. To decode columns, Hyrise uses C++'s iterator concept which – amongst other advantages – allows to apply algorithms of the standard library as if data would simply reside unencoded in an `std::vector`. Moreover, iterators can have state which enables block-based compression schemes to cache the most recently decoded block for potential upcoming accesses to the same block.

In Hyrise, column segments are usually accessed in two ways: fully sequentially or semi-randomly via a position list. As a consequence, each compression scheme provides these two access paths via a *sequential iterator* and a *point-access iterator* which accepts a position list. The impact of providing a positional access path over upfront decompression is shown in Figure 1.

Following the separation presented in [8], Hyrise distinguishes between logical-level and physical-level compression techniques and allows to cascade them. The logical-level techniques currently implemented are dictionary, frame of reference, and run length encoding. The physical-level techniques include fixed-size byte-aligned (FSBA) compression and SIMD-BP128 compression (cf. [8] for more details on the mentioned techniques).

Implementation Aspects: Hyrise is written using C++17 and uses of Boost Hana[1] for metaprogramming. To provide static interfaces within the encoding framework, we use the *curiously recurring template pattern* (CRTP). The runtime effects of static over dynamic polymorphism are shown in Figure 1. The combination of Boost Hana, CRTP, and C++14's generic lambda expressions allows us to avoid typical type resolving patterns such as the visitor pattern or hardly maintainable switch/case statements.

Hyrise's Execution Model: The mentioned iterators cover both major reading access patterns: full sequential and point accesses. The basic execution model in Hyrise (with few exceptions for the query compilation engine) follows the principles used in most modern columnar databases (cf. [1]). Predicates are descendingly ordered by their estimated costs and executed successively where each operator passes a list of qualifying positions to the next operator (i.e., position lists instead of materialized vectors). When parallel reads are preferable (e.g., as SIMD can be used), filters are executed in parallel and the results are intersected afterwards. Hence, the sequential iterator is typically used for the first filter predicate, while the following operators (e.g., following filters, joins, and aggregates) use the point-access iterator.

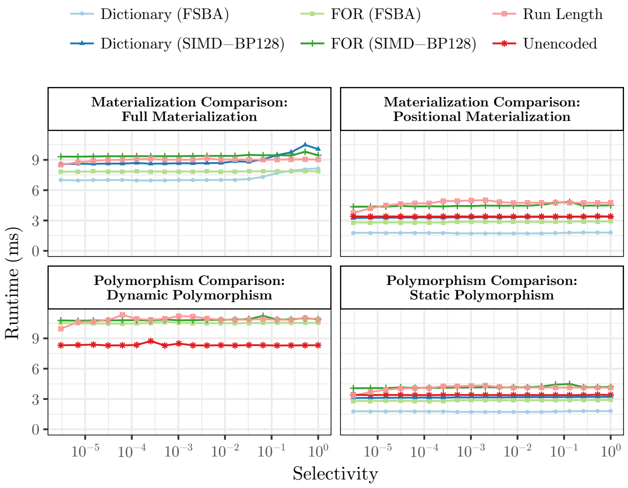


Figure 1: Positional aggregation (i.e., aggregating 25% of the tuples via a given position list) on a vector of 1M integers. Top: impact of decompressing the full column upfront vs. positional accesses. Bottom: impact of dynamic polymorphism vs. static polymorphism for row accesses.

3 ESTIMATING PERFORMANCE

In order to decide for one compression technique over another, the runtime performance as well as the resulting storage requirements need to be estimated. Estimating runtime of a particular action is implemented in any database that optimizes incoming queries and needs to decide on the order of actions to some extent. However, we think that existing approaches are by far too inaccurate for our goals. The reason is that most databases do not optimize for the accuracy of each runtime prediction they make, but rather optimize to correctly estimate *the order of alternative decisions* to take. As soon as the order is sufficiently accurate, better models with smaller errors do not provide any further advantages. We argue, however, that for any reasonable decision on the physical database design, both the potential advantages (here, reduction of allocated memory) as well as the drawbacks (here, potentially increased runtimes) need to be known upfront.

While storage requirements are rather straightforward to estimate for most compression techniques (assuming knowledge about, e.g., the row count and the number of distinct elements), we found manually crafted cost models for runtime predictions to be problematic. Due to the various CPU and compiler optimization techniques on modern platforms (e.g., out of order execution, branch prediction, code reordering) manually crafted runtime models often turn out to be too inaccurate and cumbersome.

To accurately estimate runtimes nonetheless, we create an array of regression models for each compression technique (e.g., for different data types). We measured the runtimes of sequential as well as random accesses to compressed data structures. Note

[1] Boost Hana: https://boostorg.github.io/hana/

that we do not estimate the compression runtime or write performance of encoded schemes for two reasons. First, Hyrise ensures that mutable chunks are never compressed but only immutable chunks are. Hence, no writes to immutable chunks occur apart from MVCC modifications. Second, the decoding of segment columns for reading happens significantly more often than encoding. Thus, we consider the compression costs to be amortized soon after anyways and ignore them.

We evaluated three established regression methods: gradient-boosted regression trees (in our case, XGBoost [7]) and two linear regressions with different minimization objectives. One linear regression variant uses ordinary least squares (OLS) and the other has been adapted to use a relative (normalized) error metric. The reason to use a model minimizing relative errors is that non-normalizing models are less suitable in our case as they tend to optimize the prediction of long runtimes.

Typically, the runtimes of database operators are, however, heteroscedastic, meaning that the variance differs significantly for varying input parameters (e.g., the selectivity or the table size). As a result, when estimating runtimes of operations with very short expected runtimes (e.g., a scan operation on a small dimension table with a low selectivity), extrapolating models often estimate negative runtimes. In our case, a relative error metric is a better fit as estimation errors are equally important on short and long running operations.

Table 1 shows the error rates on the example of the dictionary-encoded model for integer values. We evaluated three data sets (each set has previously been split for training and testing), one including all measurements, one including measurements below the median runtime, and one including measurements equal to or above the median runtime. For typical regression metrics, such as the *mean squared error* (MSE), XGBoost shows the best results for all data sets. However, looking at the relative error metrics *mean average percentage error* (MAPE) and *Ln Q* [15] shows that the adapted linear regression has a lower error on two of the three data sets. In Hyrise, we consider relative error metrics to be superior as the resulting model's applicability increases. On top of determining accurate rankings during access path selection, it can also be used to estimate the runtime of complex queries (e.g., queries containing nested queries which are short running but executed many times).

Quartiles	Metric	Linear Regression		XGBoost
		(OLS)	(adapted)	
$(Q1-Q4)$	MSE	498	499	323
	MAPE	1.18	1.03	2.07
	Ln Q	0.000 250	0.000 172	0.001 72
$(Q1, Q2)$	MSE	3.34	3.34	2.80
	MAPE	1.39	1.10	3.29
	Ln Q	0.000 352	0.000 196	0.003 35
$(Q3, Q4)$	MSE	993	995	643
	MAPE	0.971	0.969	0.855
	Ln Q	0.000 148	0.000 148	0.000 099 0

Table 1: Comparison of three regression models and varying error metrics (model for dictionary-encoded columns storing integers; green showing the best model).

As a consequence, we decided for the adapted linear regression model as it yields lower estimation errors and has two further advantages over more sophisticated approaches such as gradient-boosted trees or net-based approaches. First, most tree-based approaches do not support inter- and extrapolating predictions of out of sample values, which regularly happens in our scenario. Second, both learning as well as predicting of linear models is efficient, fast, and can be implemented without additional dependencies in a comparably short time frame.

While the current approach to estimate runtimes is suitable for the scenario described here, i.e., finding the best configuration for a given memory budget, it is not sufficiently covering other important scenarios. Besides storage constraints, one common constraint is limiting the expected end-to-end runtime for a given workload to be at most *n*% larger than the runtime with the current configuration. Think of cloud scenarios where a database system can be redeployed on another (potentially virtualized) server at any time. A given workload in this example can consist of a set of queries with tight runtime constraints (e.g., caused by existing service level agreements).

4 COMPRESSION SELECTION

Selecting a suitable compression configuration requires knowledge about the particular data characteristics as well as the workload. Most current databases use simple heuristics to choose a compression scheme for a given column. However, those approaches neglect three major issues that we deem crucial: Compression configurations ought to be (i) adaptable with respect to a given memory budget, (ii) consider its impact on the decisions the query optimizer is going to make given the configuration, and (iii) consider opportunity costs usually exploitable in real-world systems and workloads.

While cost considerations always played an important role in database systems, the current trend towards self-adapting cloud systems emphasizes the need to reduce the memory consumption (amongst others) as much as possible while ensuring acceptable performance. In fact, a large database vendor told us that for many cloud installations, TCO (total cost of ownership) reductions are a far more pressing issue than performance.

As such, the database must understand the performance impact of varying memory budgets. It needs to understand the interplay between space consumption and performance, e.g., to apply maximum compression to a table that is virtually never accessed. At the same time, such a memory budget-driven system should degrade gracefully for decreasing budgets.

Our selection framework accepts a given memory budget that should not be exceeded. The system uses heavier compression schemes for data that is rarely accessed or whose access patterns do not suffer from heavy compression, while frequently accessed tables might use light-weight compression schemes or no compression at all. With the workload at hand, it might make sense to lose certain performance by applying heavier compression for a less often accessed table and invest that gained space to frequently accessed tables (cf. opportunity costs).

4.1 Greedy Selection Heuristics

The goal of compression selection is to determine a compression configuration for a given data set and workload. The selected configurations should gracefully degrade for decreasing memory budgets. The memory consumption of the resulting compression configuration ought to be within the given memory budget. We evaluated two heuristics and static configurations for a synthetically generated CH-benCHmark-like workload.

To gather workload information, Hyrise parses the database's query plan cache. This information is fed to the selection heuristics and includes for each physical column, e.g., which operations are executed with which access path taken (e.g., full linear accesses or random probing accesses).

We implemented two greedy heuristics which have been proposed for a related field of physical design optimization: index selection. As the first heuristic, we implemented a density-based heuristic that chooses the first compression technique from all applicable techniques ordered ascendingly by their size-to-performance improvement ratio, comparable to [16]. As the second heuristic, we implemented a performance-greedy heuristic that selects compression schemes ordered ascendingly by their expected performance improvement, comparable to [11].

The results are shown in Figure 2. We make two observations. First, the "All FOR (FSBA)" configuration has shown to be a good trade-off between performance and memory consumption. No heuristic was able to find a comparable configuration as FOR encoding is often neglected by both greedy heuristics. Second, simple greedy heuristics are not sufficient as they (i) fall short in covering the whole range of permitted memory budgets and (ii) can even be outperformed by simple static heuristics.

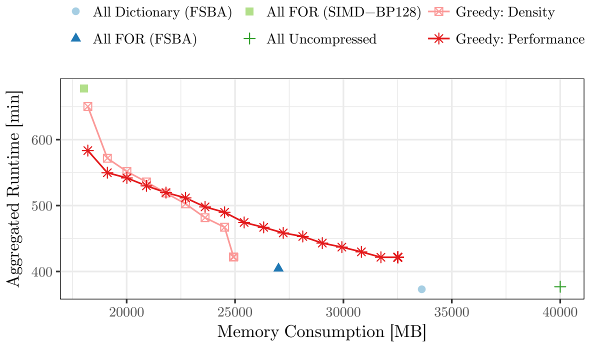


Figure 2: Comparison of static compression configurations and budget-driven heuristics.

The reason is that simple heuristics do not incorporate interactions affecting the optimization of query plans (cf. *index interaction*). One simple example is the effect of ordering conjunctive filter chains when the predicate with the lowest selectivity (hence, preferably being executed first) happens to be heavily compressed and hence slow to access. Optimizers using non-logical cost models – as done in Hyrise – might yield completely different query plans. To cover such cases, we think more elaborate selection approaches (e.g., recursive approaches as [3] and [6], or ILP-based approaches as [9]) are a necessary next step.

4.2 Robustness

The robustness of compression configurations is another crucial aspect. The more a segment is compressed as the expected workload is infrequently or not accessing it at all, the more expensive this decision might turn out when workloads shift.

To provide robust configurations, we use a simple framework to generate additional workloads which are evaluated together with the actually provided workload. Instead of selecting the compression configuration with the lowest runtime for the given workload, we choose the configuration minimizing the aggregated runtime of all workloads.

The creation of alternative workloads is done by shuffling and adding queries. First, for every query being part of the provided workload, we randomly select a new execution count based on the normal distribution around the actual execution count. Second, to counter the problem of heavy-weight compression for infrequently accessed segments, we manually add a query for each table to the workload that projects all columns. Both the number of generated workloads as well as the number of executions for the manually added query are configurable.

5 RELATED WORK

The object of integrating and selecting compression schemes has been researched in several previous works. Abadi et al. presented their C-Store extension to support various compression schemes [2] allowing operations directly on compressed data. A unified interface allows exploiting several compression scheme-unique optimizations while some operations have to fall back to a virtual method call-based interface. Further, the authors proposed a decision tree for the selection of compression schemes which uses both workload and data properties, but which does not adapt to changing environments nor considers memory budgets.

Lemke et al. presented a performance-optimized approach for TREX, the predecessor to SAP HANA [14]. At the cost of code complexity and maintenance efforts, compression schemes are explicitly handled within most database operators allowing to fully exploit vectorization and other optimizations.

Lang et al. presented data blocks for HyPer [13]. The authors focus the interplay of vectorizing operation on compressed vectors and their query compilation engine. HyPer selects compression schemes based on data characteristics.

6 CONCLUSION

We presented the current state of column compression in Hyrise. We think that compression selection will be an increasingly crucial topic. Mostly caused by (i) an increasing autonomy of database systems adapting themselves to changing environments and (ii) the move from on-premise to cloud-based installations which emphasizes the need for reduced main memory footprints.

REFERENCES

[1] D. Abadi, , P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations & Trends in Databases* 5, 3 (2013), 197–280.
[2] Daniel Abadi et al. 2006. Integrating compression and execution in column-oriented database systems. In *Proc. SIGMOD*. 671–682.
[3] Martin Boissier et al. 2018. Hybrid Data Layouts for Tiered HTAP Databases with Pareto-Optimal Data Placements. In *Proc. ICDE*. 209–220.
[4] Peter A. Boncz et al. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. VLDB*. 54–65.
[5] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85.
[6] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proc. VLDB*. 146–155.
[7] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *KDD*. ACM, 785–794.
[8] Patrick Damme et al. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *Proc. EDBT*. 72–83.
[9] Debabrata Dash et al. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *PVLDB* 4, 6 (2011), 362–372.
[10] M. Dreseler et al. 2019. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *Proc. EDBT*.
[11] Hideaki Kimura, Vivek R. Narasayya, and Manoj Syamala. 2011. Compression Aware Physical Database Design. *PVLDB* 4, 10 (2011), 657–668.
[12] Tirthankar Lahiri et al. 2015. Oracle Database In-Memory: A dual format in-memory database. In *ICDE*. IEEE Computer Society, 1253–1258.
[13] Harald Lang et al. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*. 311–326.
[14] Christian Lemke et al. 2010. Speeding Up Queries in Column Stores - A Case for Compression. In *Proc. DAWAK*. 117–129.
[15] Chris Tofallis. 2015. A better measure of relative prediction accuracy for model selection and model estimation. *JORS* 66, 8 (2015), 1352–1362.
[16] Gary Valentin et al. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proc. ICDE*. 101–110.
[17] Thomas Willhalm et al. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB* 2, 1 (2009), 385–394.

CLRL: Feature Engineering for Cross-Language Record Linkage

Öykü Özlem Çakal
Technische Universität Berlin
o.cakal@campus.tu-berlin.de

Mohammad Mahdavi
Technische Universität Berlin
mahdavilahijani@tu-berlin.de

Ziawasch Abedjan
Technische Universität Berlin
abedjan@tu-berlin.de

ABSTRACT

Record linkage aims at identifying duplicate records across datasets. Most existing record linkage techniques have been designed for monolingual datasets. In this paper, we propose a novel approach, CLRL, that links the records in a cross-language setting, where each input dataset is in a different language. CLRL combines monolingual similarity measures with multilingual cross-language word embedding similarities to identify the correspondence of records across datasets. As our experiments show, CLRL outperforms baseline approaches in cross-language data integration settings.

1 INTRODUCTION

Record linkage is one of the most relevant tasks in a data integration process. The goal of record linkage is to identify records from two different datasets that represent the same real-world entity. One of the major challenges in record linkage is to identify similarity heuristics that are effective at approximating the equality of two heterogeneously represented entities [8]. Numerous similarity measures have been proposed so far to capture various similarity levels such as character-based and phonetic-based similarities [3]. These measures are effective in the monolingual setting, where similar words have a high lexical similarity. However, they are often ineffective in a cross-language settings, where each dataset adheres to a different language.

A naive solution to overcome this problem is to first translate one dataset into the other corresponding language and then apply an off-the-shelf record linkage approach. However, this approach suffers from two major problems.

(1) *Ambiguity in translation.* Short texts in structured datasets do not usually provide enough context for machine translation models to translate accurately.
(2) *Out-of-vocabulary terms.* The machine translation model cannot translate out-of-vocabulary terms, such as the concatenation "firstsight", which is not among the standard language vocabulary.

Motivation example. Table 1 illustrates two movie datasets in English (dataset *A*) and German (dataset *B*). Among all the 4 possible pairs of records in $A \times B$, only the second record of the English dataset should be linked to the first record of the German dataset. Traditional record linkage approaches would fail to identify their correspondence as there is no lexical similarity between the title "Forbidden Planet" and the title "Alarm im Weltall". Translating one of the datasets and then applying a traditional record linkage approach also fails because the translation of "Alarm im Weltall" is "Alarm in the Universe"; it is still not lexically similar to "Forbidden Planet".

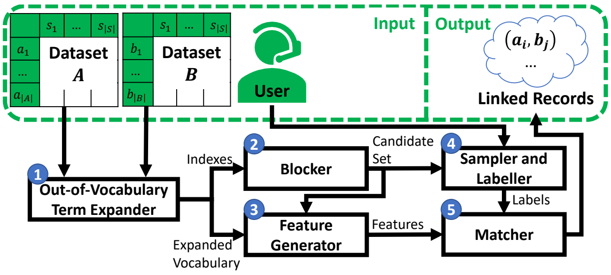


Figure 1: The workflow of CLRL.

Table 1: Two movie datasets in English and German.

ID	Name	Year	ID	Name	Jahr
1	Heat	1995	1	Alarm im Weltall	1956
2	Forbidden Planet	1956	2	Der Pate	1972

However, our approach is able to identify the correspondence by using the latent similarity of these two multilingual short movie titles based on cross-language word embedding models [14]. In particular, our maximum vector similarity feature (see Section 3.2) captures the similarity of the words "Planet" and "Weltall" (means "universe" in German) as these semantically similar words have similar word embedding vectors. □

Contributions. In this paper, we propose a novel approach to link the records across multilingual datasets. To this end, we make the following contributions:

- We design a term expansion scheme (Section 3.1) to expand each out-of-vocabulary term into a set of in-vocabulary terms. In fact, CLRL leverages a three-step policy to expand different types of out-of-vocabulary terms differently.
- We propose an effective set of similarity features (Section 3.2) for cross-language record linkage problem. In addition to state-of-the-art monolingual similarity measures, we include four multilingual similarity measures, adopted from cross-language word embedding models.
- We empirically evaluate our approach on six real-world datasets (Section 4). In particular, we show that CLRL outperforms three existing record linkage approaches in cross-language setting.

2 CLRL OVERVIEW

Figure 1 illustrates the record linkage procedure with CLRL. The multilingual datasets and the user feedback are the input and the set of linked records is the output of the approach.

CLRL adheres to the well-known pipeline of existing record linkage approaches, i.e., preprocessing, blocking, and matching. In addition to standard preprocessing operations, such as value normalization and identifying corresponded attributes, our approach can apply a novel preprocessing step to expand out-of-vocabulary terms into in-vocabulary terms (Step 1). We will detail this step in Section 3.1. In the blocking step, a state-of-the-art blocker is used to generate a set of candidate links between record

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.85

pairs of the input datasets (Step 2). Then, CLRL generates features for each candidate pair (Step 3). We will detail this step in Section 3.2. Depending on the sampling strategy and user-interaction model, a sample set of candidates are chosen to be labeled by the user as matches or non-matches (Step 4). Finally, a state-of-the-art classifier takes the features and labeled record pairs to classify all record pairs in the candidate set (Step 5).

3 FEATURE ENGINEERING

We first explain how out-of-vocabulary terms are expanded into in-vocabulary terms. Then, we describe our extensive feature set.

3.1 Out-of-Vocabulary Term Expansion

Out-of-vocabulary terms are those terms that could not be found in formal vocabularies and therefore are not translatable. The goal of out-of-vocabulary term expansion is to transform these non-translatable terms into in-vocabulary terms. This way, we can later leverage cross-language word embedding models to capture the similarity of these terms as well. CLRL applies the following steps to the out-of-vocabulary terms.

Morphological checking. CLRL first tries to find morphological variants of the out-of-vocabulary term. In morphological check, the out-of-vocabulary term is transformed into its morphemes (i.e., primitive units), if applicable. For example, the out-of-vocabulary term "firstsight" is transformed into in-vocabulary terms "first" and "sight". We leverage Polyglot Python module [16], which supports 100 languages, to conduct morphological transformations. Note that our morphological checking already covers lighter lemmatization and stemming transformations as well.

Spell checking. Spelling errors could be the emerging cause of many out-of-vocabulary terms. Therefore, in case of failure in morphological transformation, CLRL tries to fix spelling errors. To this end, the approach collects all the in-vocabulary terms whose Damerau-Levenshtein edit distance to the out-of-vocabulary term is less than equal to $\theta_{dist} = 1$. To minimize the risk of replacing an out-of-vocabulary term with the wrong in-vocabulary term, we restrict the threshold to the minimum possible distance, i.e., $\theta_{dist} = 1$, and replace the term only when exactly one in-vocabulary candidate has been found.

Ostrich policy. CLRL ignores transforming all the other out-of-vocabulary terms that could not be transformed by the previous treatments. These out-of-vocabulary terms are mainly numbers (e.g., "2001" in "2001: A Space Odyssey") or named entities (e.g., "Lebowski" in "The Big Lebowski") that do not need any transformation.

3.2 Feature Vector

Each pair of records in the candidate set is mapped to a feature vector that contains all the similarity scores of the two records. Let $A = \{a_1, a_2, \dots, a_{|A|}\}$ and $B = \{b_1, b_2, \dots, b_{|B|}\}$ be two relational datasets with different languages, where each $a \in A$ or $b \in B$ is a record. Let $S = \{s_1, s_2, \dots, s_{|S|}\}$ be the set of mapped attributes in these datasets. Therefore, the data cell $a[s]$ refers to the record $a \in A$ and the attribute $s \in S$. Let f be a similarity function that takes two data cells $a[s]$ and $b[s]$ and returns a similarity score $f(a[s], b[s]) \in [0, 1]$. Therefore, the feature vector of a candidate record pair (a, b) is

$$V(a, b) = [f(a[s], b[s]) \mid \forall f \in F \wedge \forall s \in S], \tag{1}$$

where F is the set of all the similarity functions. Due to the cross-language setting, the similarity functions should be able to capture, not only the monolingual similarity, but also the multilingual similarity of terms in different languages. That is why, we leverage monolingual and multilingual similarity functions.

3.2.1 Monolingual Similarity Functions. Monolingual similarity functions are typical lexical similarity measures. The goal of incorporating these measures is to capture lexical similarity of named entities, such as "Brad Pitt", which are written similarly in languages with the same scripting system. In particular, we calculate Jaccard, Levenshtein, Jaro, Jaro-Winkler, Needleman-Wunsch, Smith-Waterman, and Monge-Elkan similarity measures [5].

3.2.2 Multilingual Similarity Functions. Multilingual similarity functions capture the similarity of data values across different languages. We leverage cross-language word embedding models [14] to capture the similarity of short multilingual data values of datasets. Word embedding models, such as word2vec [12], learn to map each term into a dense vector in a way that terms with similar context (i.e., surrounding words) have similar vector representations as well. Cross-language word embedding models, such as fastText [7], are a special kind of word embedding models. These models share the same cross-language space for two different languages so that similar words from different languages can have similar vector representations. Thus, a cross-language word embedding model m can take a word w and returns its correspondent vector $m(w)$ in a cross-language shared space. In this shared space, not only monolingual similar words such as "Dog" and "Puppy" would have close vector representations, but also cross-language similar words such as "Dog" and "Hund" (means "dog" in German) would have close vector representations, i.e., $m(\text{"Dog"}) \approx m(\text{"Hund"})$.

Now, let m be the cross-language word embedding model and let $P = \{p_1, p_2, \dots, p_{|P|}\}$ and $Q = \{q_1, q_2, \dots, q_{|Q|}\}$ be the sets of words in data cells $a[s]$ and $b[s]$, respectively. We define the following four similarity functions.

Mean vector similarity (MeVS). The Mean of word vectors in a data cell is a representative vector for the whole data cell. Considering each data cell as a set of word vectors, we calculate the cosine similarity of mean vectors of data cells. Formally,

$$MeVS(P, Q) = \text{cosine}\Big(\frac{1}{|P|}\sum_{p \in P} m(p), \frac{1}{|Q|}\sum_{q \in Q} m(q)\Big). \tag{2}$$

Maximum vector similarity (MaVS). When the data cell contains noisy (i.e., irrelevant) words, it is desirable to represent the data cell by the vector of its most important word. For example, "Forbidden Planet" and "Alarm im Weltall" are the same movie titles in English and German. Since "Weltall" means "universe" in German, $m(\text{"Weltall"})$ is a more accurate representation for the German movie title rather than mean of all the word vectors inside the complete German movie title. This similarity function outputs the maximum cosine similarity between all the pairs of words $P \times Q$. Formally,

$$MaVS(P, Q) = \max_{(p,q) \in P \times Q} \text{cosine}\Big(m(p), m(q)\Big). \tag{3}$$

Optimal alignment similarity (OAS). If two data cells are matched, they might have an optimal one-to-one alignment of words, where each word in the first data cell corresponds directly to one word in the other data cell. This similarity function looks for such an optimal alignment between words of P and Q, where the sum of similarity scores between aligned word pairs is maximal. Finding the optimal one-to-one alignment of words is the classical assignment problem. We leverage the Hungarian algorithm [9] to find the optimal one-to-one alignment of words in two data cells. Since the aligning needs both P and Q to have the same length of words, the shorter one is padded with arbitrary out-of-vocabulary words, hence inducing dissimilarity. Let us assume that P is the shorter one and its padded version is P'. Formally,

$$OAS(P,Q) = \frac{\sum_{(p,q)\in L(P',Q)} \text{cosine}\Big(m(p), m(q)\Big) \times (|P| + |Q|)}{2 \times |P| \times |Q|}, \tag{4}$$

where $L(P', Q) = \{(p, q) \mid p \in P', q \in Q\}$ is the optimal one-to-one alignment of words in P' and Q. Note that since this similarity function depends on the word count in P and Q, we normalize its value by the harmonic mean of these word counts, as suggested in literature [4].

Maximum alignment similarity (MAS). Two matching data cells might not necessarily have an optimal one-to-one alignment of words. For example, although the English data value "Purchase Price" is matched to the German data value "Kaufpreis", there is no optimal one-to-one alignment for words. Instead, here we have a two-to-one alignment for the words. Thus, in general, we also need a similarity function that can capture the similarity of m-to-n alignments. This similarity function allows each word in the first data value be aligned to the most similar word in the second data value, regardless of whether these two words are already aligned to any other words or not. Formally,

$$\begin{aligned} MAS(P,Q) = \frac{1}{2}\Big(\frac{1}{|P|}\sum_{p\in P} \text{cosine}\Big(m(p), m(q_p^*)\Big) + \\ \frac{1}{|Q|}\sum_{q\in Q} \text{cosine}\Big(m(q), m(p_q^*)\Big)\Big), \end{aligned} \tag{5}$$

where, $q_p^* \in Q$ is the most similar word to word $p \in P$, i.e., $q_p^* = \max_{q\in Q} \text{cosine}\Big(m(q), m(p)\Big)$.

4 EVALUATION

Experimental setup. We evaluate our approach on six real-world datasets, which are described in Table 2. *Universities* and *Universités* contain information of universities around the world in English and French, respectively. An example of matched universities in these datasets is "Technical University of Berlin" and "Université technique de Berlin". *Movies* and *Películas* contain information on movies in English and Spanish, respectively. An example of matched movies in these datasets is "The Godfather" and "El padrino". *Wikipedia Titles* and *Wikipedia-Titel* contain wider domains including titles of Wikipedia pages in English and German, respectively. An example of matched titles in these datasets is "1982 World Snooker Championship" and "Snookerweltmeisterschaft 1982". We extracted these datasets from the DBpedia knowledge base [10]. We leveraged inter-language links inside DBpedia to obtain the ground truth for these datasets, i.e., the pairs of records that are actually linked. We evaluate our approach with precision $P = \frac{\text{the number of correctly identified linked records}}{\text{the number of all outputted linked records}}$, recall $R = \frac{\text{the number of correctly identified linked records}}{\text{the number of all actual linked records}}$, and F_1 measure $F_1 = \frac{2\times P\times R}{P+R}$. We mainly report only the F_1 measure, which combines the precision and recall, due to the space constraints.

Table 2: Datasets.

Name	Language	#Rows	#Common Attributes	Candidate Set Size	#Actual Linked Records
Universities	English	8758	16	124559	940
Universités	French	3957			
Movies	English	1273	14	59198	72
Películas	Spanish	15334			
Wikipedia Titles	English	1976	2	51211	83
Wikipedia-Titel	German	2159			

We apply cross-validation and report the mean and standard deviation of these measures. As the default parameter setting, we setup our approach with all the introduced features. We also leverage fastText [7] as the cross-language word embedding model and XGBoost [1] as the classifier. Our prototype is available online[1].

Effectiveness versus baselines. We compare the effectiveness of CLRL to the following three baseline approaches:

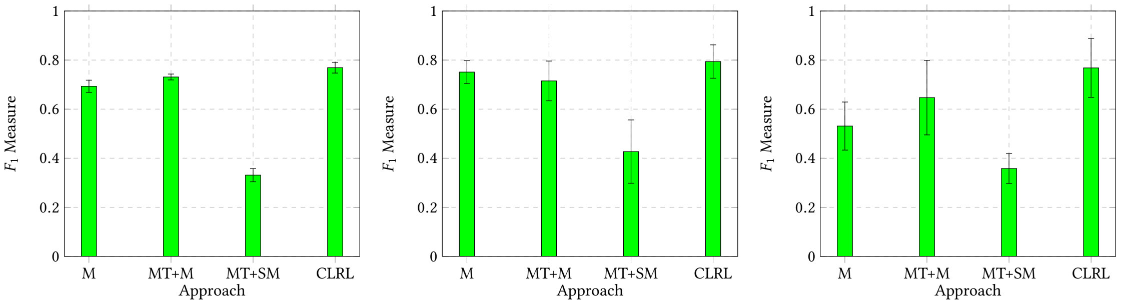
(1) **Magellan (M).** Magellan is an end-to-end entity matching system that uses monolingual lexical similarity features to learn links between records [8]. In our experiments, we include the following default set of features: Jaccard, Levenshtein, Jaro, Jaro-Winkler, Needleman-Wunsch, Smith-Waterman, and Monge-Elkan similarity measures.
(2) **Machine translation plus Magellan (MT+M).** This approach first translates non-English language datasets into English using the Joshua machine translation toolkit [11] and then applies Magellan on two English datasets.
(3) **Machine translation plus semantic matching (MT+SM).** This approach also leverages machine translation to have both datasets in English. However, instead of using Magellan, it applies a monolingual word embedding model to link records [15].

Figure 2 illustrates the effectiveness of CLRL in comparison to these baseline approaches. CLRL always outperforms the other approaches as it leverages a broad set of features to capture monolingual and multilingual similarities. This superiority is more obvious on *Wikipedia Titles/Wikipedia-Titel* datasets, as they contain more linguistically different content. In fact, in university and movie domains there are many named entities such as "Berlin" and "Brad Pitt" that remain the same in many languages. Therefore, even a traditional record linkage approach, such as Magellan, with lexical similarity measures can capture the similarity. However, in Wikipedia title domain there are fewer named entities, hence cross-language techniques are more promising.

Out-of-vocabulary term expansion analysis. Figure 3 illustrates the influence of out-of-vocabulary term expansion on the effectiveness of CLRL. Leveraging out-of-vocabulary term expansion, CLRL has a higher F_1 measure as the similarly functions can capture the similarities with higher recall, i.e., CLRL can identify more linked records. Again, the improvement is higher on *Wikipedia Titles/Wikipedia-Titel* datasets as the out-of-vocabulary terms in these datasets are mainly morphological and spell errors, which are transformed into in-vocabulary terms.

Feature analysis. Table 3 illustrates the effectiveness of CLRL when it leverages different features separately. In general, all the similarity features are informative for the task as CLRL works best with all the features. Furthermore, the proposed multilingual similarity features provide higher F_1 measure than the traditional monolingual features.

[1] https://github.com/BigDaMa/clrl



(a) Universities/Universités

(b) Movies/Películas

(c) Wikipedia Titles/Wikipedia-Titel

Figure 2: Effectiveness of different approaches on different datasets.

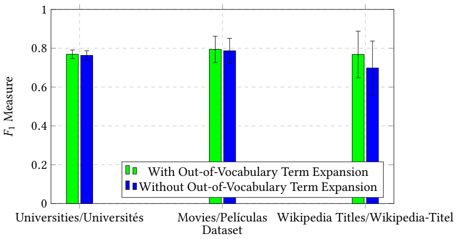


Figure 3: F_1 measure of CLRL with and without out-of-vocabulary term expansion.

Table 3: F_1 measure of CLRL with different feature groups.

Feature Name	Universities/Universités	Movies/Películas	Wikipedia Titles/ Wikipedia-Titel
Monolingual	0.69 ± 0.02	0.75 ± 0.04	0.53 ± 0.10
MeVS	0.70 ± 0.01	0.76 ± 0.08	0.67 ± 0.09
MaVS	0.71 ± 0.02	0.79 ± 0.09	0.60 ± 0.13
OAS	0.72 ± 0.02	0.75 ± 0.03	0.59 ± 0.10
MAS	0.75 ± 0.01	0.80 ± 0.04	0.70 ± 0.14
Full	**0.77 ± 0.02**	**0.80 ± 0.06**	**0.77 ± 0.12**

5 RELATED WORK

Cross-language record linkage. There are only few pieces of work on cross-language record linkage because this is relatively a new topic. Song et al. [15] translated the Japanese datasets into English and then applied monolingual word embedding models to identify linked records. As shown in the experiments, CLRL outperforms this approach because of two reasons. First, CLRL does not rely on direct translation of datasets, which can be ambiguous as explained earlier. Second, instead of only one monolingual word embedding-based similarity feature, CLRL leverages various monolingual and multilingual similarity features to capture the similarities of multilingual records more accurately.

Record linkage. Numerous works have tackled the similarity representation challenge of record linkage task by different similarity measures [3]. In addition to these common monolingual similarity measures, CLRL leverages multilingual word embedding-based similarity measures as well-suited similarity features for cross-language setting. We showed the benefit of the new similarity features in our experimental comparison.

Cross-language matching. Cross-language matching has been mainly studied for unstructured text data in tasks such as information retrieval [6] and entity matching [13]. While the entity is usually surrounded with a rich context of words in these tasks, in structured datasets the texts are mainly short, which make the cross-language matching task more challenging. That is why, CLRL leverages cross-language word embedding models to capture the semantic similarity of short multilingual texts accurately.

Out-of-vocabulary term expansion. Out-of-vocabulary term expansion has been addressed for the record linkage problem using the top-K co-occurring words with the out-of-vocabulary term [2]. CLRL does not hold any assumption on the term frequency of the out-of-vocabulary terms in the dataset.

6 CONCLUSION

We addressed the problem of cross-language record linkage. In addition to the monolingual similarity measures, we leverage four novel cross-language word embedding-based similarity measures. As our experiments show, CLRL outperforms three record linkage baseline approaches in cross-language setting. In future, we plan to extend the blocking step. Since the records are in different languages, simple blocking heuristics, such as having a word in common, do not work effectively in cross-language setting.

ACKNOWLEDGMENTS

This work has been partially funded by the German Ministry for Education and Research as BBDC II (01IS18025A).

REFERENCES

[1] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *SIGKDD*. 785–794.

[2] Muhammad Ebraheem et al. 2017. DeepER–deep entity resolution. *arXiv preprint arXiv:1710.00597*.

[3] Ahmed K Elmagarmid et al. 2007. Duplicate record detection: A survey. *TKDE* 19, 1, 1–16.

[4] Goran Glavaš et al. 2018. A resource-light method for cross-lingual semantic textual similarity. *Knowledge-Based Systems* 143, 1–9.

[5] Wael H Gomaa and Aly A Fahmy. 2013. A survey of text similarity approaches. *IJCA* 68, 13, 13–18.

[6] Gregory Grefenstette. 2012. *Cross-language information retrieval.* Springer Science & Business Media.

[7] Armand Joulin et al. 2018. Loss in translation: Learning bilingual word mapping with a retrieval criterion. In *EMNLP*. 2979–2984.

[8] Pradap Konda et al. 2016. Magellan: Toward building entity matching management systems. *PVLDB* 9, 12, 1197–1208.

[9] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval research logistics quarterly* 2, 1-2, 83–97.

[10] Jens Lehmann et al. 2015. DBpedia–a large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web* 6, 2, 167–195.

[11] Zhifei Li et al. 2009. Joshua: An open source toolkit for parsing-based machine translation. In *StatMT*. 135–139.

[12] Tomas Mikolov et al. 2013. Distributed representations of words and phrases and their compositionality. In *NIPS*. 3111–3119.

[13] Daniel Rinser et al. 2013. Cross-lingual entity matching and infobox alignment in Wikipedia. *IS* 38, 6, 887–907.

[14] Sebastian Ruder et al. 2017. A survey of cross-lingual word embedding models. *arXiv preprint arXiv:1706.04902*.

[15] Yuting Song et al. 2016. Cross-language record linkage using word embedding driven metadata similarity measurement.

[16] Sami Virpioja et al. 2013. Morfessor 2.0: Python implementation and extensions for Morfessor Baseline. Aalto University.

Short Paper

Operational Stream Processing: Towards Scalable and Consistent Event-Driven Applications

Asterios Katsifodimos
Delft University of Technology
Netherlands
a.katsifodimos@tudelft.nl

Marios Fragkoulis
Delft University of Technology
Netherlands
m.fragkoulis@tudelft.nl

ABSTRACT

In the last decade we are witnessing a widespread adoption of architectural styles such as microservices, for building event-driven software applications and deploying them in cloud infrastructures. Such services favor the separation of a database into independent silos of data, each of which is owned entirely by a single service. As a result, traditional OLTP systems no longer fit the architectural picture and developers often turn to ad-hoc solutions that rarely support ACID transaction consistency.

At the same time, we are witnessing the gradual maturation of distributed streaming dataflow systems. These systems nowadays have departed from the mere analysis of streaming windows and complex-event processing, employing sophisticated methods for managing state, keeping it consistent, and ensuring exactly-once processing guarantees in the presence of failures.

The goal of this paper is threefold. First, we illustrate the requirements of stateful software services in terms of consistency and scalability. Second, we present how well existing solutions meet those requirements. Finally, we outline a set of challenging problems and propose research directions for enabling event-driven applications to be developed on top of streaming dataflow systems. We strongly believe that streaming dataflows can have a central place in service-oriented architectures, taking over the execution of ACID transactions, ensuring message delivery and processing, in order to perform scalable execution of services.

1 INTRODUCTION

Event-driven Applications (EDA) are software applications that act on incoming events. EDAs nowadays span a multitude of areas. Some very common examples are GUI applications (for web services, gaming, design, etc), complex event processing (for fraud detection, pattern matching, etc.), computations on streaming graphs and machine learning, and analytical queries over streams, such as window aggregation. In this paper we focus on two emerging application types named after the architectural pattern they follow, namely *microservices* [20] and actor-based systems [1], such as Erlang [22], Akka,[1], but most importantly, higher lever abstractions such as Microsoft's Orleans [3]. Our work is motivated by the observation that these emerging architectural patterns do not receive the necessary amount of attention from the database community, although they are extremely ubiquitous and growing in popularity by the day.

Microservices and actors have a surprising number of commonalities. Microservices, like actors, are founded on the principle of separation of concerns: each microservice (or actor) manages its own data and implements a set of endpoints (actors offer function calls). The only way for a microservice to get data from another microservice is to make a call on an endpoint (a function call/shipping for an actor). However, the two have a considerable amount of differences. More specifically, microservices favor communication via synchronous REST API calls[2] and ensure fault tolerance by relying on an external database for persistence. Actors, on the other hand, communicate via asynchronous messages and typically persist state in a local data structure or even in an external storage system [3]. The local state of actors can be used for recovering after failures, but also for migrating to different machines. To alleviate data management and consistency issues from actors, Bernstein et. al. [10] proposed the concept of virtual actors, backed by actor-oriented database systems, with the goal of integrating database concepts inside actor systems.

Stateful microservices can scale particularly well, but almost always implement eventual consistency, such as SAGAS [11]. In case that strict consistency is required, orchestration and Two-Phase Commit (2PC) take place at the application level: users hard-code database logic in their applications, using APIS such as Java XA, by implementing commit, rollback, and prepare for 2PC to work. Worse, transactions today contain more and more complex business logic. However, encoding complex business logic in a stored procedure is not preferred nowadays [2]. As a result, strict consistency – an old and difficult problem that was once only the responsibility of very few database programmers – is now part of daily work for application programmers. Finally, stateful services (including virtual actors) require state locality in order to achieve low latency - not only when writing data, but most importantly when reading data. As applications become more and more interactive, reacting to state changes renders latency requirements even stricter. At the same time, serverless computing [23], an emerging trend allowing the execution of user-supplied functions as a service, is proven to be a bad fit for stateful microservices. This is because it necessitates shipping data to the code and forces communication between executing components through the storage layer, which is slow compared to a direct network connection [14].

The aforementioned shortcomings call for a principled solution that will allow implementing EDAs with innate support for transactions, loose-coupling of service modules with local state, and consistent global state. In this paper we argue that such a fabric can be based on streaming dataflows. More specifically, modern streaming dataflow systems, such as Apache Flink [7] and Samza [17], execute a topology of continuously executing operators with local state and maintain a consistent snapshot of their global state. Operators cooperate to provide analytics on bounded and unbounded data. Moreover, features like Flink's Savepoints [6] or Kafka's Streams offer a deterministic time machine for debugging and replaying dataflow executions, and can be used to hide failures from application developers by offering

[1] https://akka.io

[2] Certain microservice implementations such as reactive microservices opt for asynchronous messaging.

exactly-once processing guarantees. Finally, dataflow systems scale extremely well. As a result, at the time of writing, we are witnessing a trend towards building stateful applications on top of streaming engines.

In this paper we present the vision of *operational stream processing* whose goal is to render stream processors full-fledged data management engines, capable of executing transactions, performing analytics, and embedding complex business logic of stateful services inside dataflow operators. We organize this paper as follows: in Section 2 we focus on current best practices for implementing EDAS and their requirements for scalability and consistency. In Section 3 we review existing possible solutions to fill those requirements. Finally, in Section 4 we argue that streaming dataflow systems, such as Apache Flink [7] and Samza [17], can serve as an efficient, and scalable backend for executing EDAS, given that our community tackles a set of important challenges.

2 REQUIREMENTS OF EDAS

In this section we first briefly outline the main requirements of EDAS with respect to the backend technology required for their execution. We then focus on a set of advanced operational requirements of microservices and actors.

2.1 General Requirements

The following requirements manifest themselves in almost all EDAS. We believe that a fabric that can be used as a backend for EDAS should at least provide support for the following.

Fault-Tolerant State. State is a first-class citizen in virtually every event-driven computation. State in a streaming computation can be counters (e.g., counting elements in a stream), database contents in a microservice or the current computation state of an actor program. At the same time a distributed event-processing application, needs to ensure that, even in the presence of failures, the state of the system remains consistent and the application continues its operations from where it left off. Whenever possible, failures should be transparent to the application programmer.

Event Partitioning & Scaling Out. Computations over partitioned data are typically used for computing aggregates, and for scaling-out actor instances and load-balancing user requests (e.g., by partitioning per user id). Similarly, scalability in microservices can be typically achieved by running multiple service instances, balancing the load among them using HTTP proxies.

2.2 Advanced Requirements

Apart from the general requirements, which are needed by most EDAS, most modern applications demand a special set of requirements for the operation of the services that comprise them.

Transactions. One of the largest problems in running services is the lack of coordination schemes in order to perform *transactions* and retain consistent state across services.

State Locality. Access to local state is important for boosting the performance of services [8], yet it is not always leveraged in microservice or actor architectures in favor of keeping those services stateless. In those cases, the state is offloaded to an external storage system. However, stringent latency requirements of interactive server applications require state to remain embedded in the service.

Global State View & Analytics. Services often need to consolidate data from multiple other services in a bulk fashion. For instance consider the case of joining orders with transactions, in order to obtain insight about sales. Since each service owns its data, consolidating that state via multiple calls to service endpoints is currently slow and cumbersome.

Loose coupling. One of the most important reasons that services are so popular is that they allow developers to develop, test, and deploy them in a loosely-coupled fashion. We consider this a defining trait of services that must be respected.

Debugging and Auditing. Given their distributed nature, services inherit the difficulty to reason, understand, and debug. To this end, services need inherent support for debugging and testing in a reproducible manner.

Dynamic (re)configuration. During the lifetime of a service, the load of the service (e.g., due to churn), its assigned hardware (e.g., due to failures), and even the service itself (e.g. due to updates) can change dramatically. Services need to transparently adapt to those changes without being affected in terms of performance and availability.

3 EXISTING SOLUTIONS

Many existing solutions meet the general requirements presented in Section 2.1. The advanced requirements of Section 2.2 are a lot more challenging to support though. This section presents which of the advanced requirements are fulfilled by existing systems, namely microservices frameworks, OLTP systems, actor programming frameworks, and stream processors.

3.1 Microservices frameworks

Transactions. The most widespread solutions for performing transactions in microservices frameworks are SAGAS [11] and application-level transaction managers implementing the Java Transactions API (JTA). A transaction in the SAGA pattern allows microservices to make local state changes (e.g., book a flight/hotel) independently of the rest of the microservices taking part in the transaction. On failure, all microservices that have already updated their state, need to issue compensating actions (e.g., cancel the flight booking). SAGAS pose two main issues: i) state consistency across microservices cannot be achieved and ii) not every action can be compensated. Various web application development frameworks, such as Spring, offer eXtended Transactions (XA), an implementation of 2PC with ACID-like properties for web applications [18]. The solutions based on application-level transaction managers require a tight coupling of services, and demand from developers to implement 2PC functions (e.g., `prepare` and `abort`) in application-level code which is very error prone and necessitates deep understanding of database concepts.

State Locality. The state locality requirement contradicts the design of stateless microservices, which dictates handing over the responsibility of persisting state to an external database system. Thus, current microservice frameworks fail to fulfill this requirement. Scaling-out microservices requires very sophisticated design of the backing database architecture. The service's state in these cases is external; accessing it requires a call to an external system, introducing latency.

Global State View. Microservices frameworks are not suitable for providing global state views. Since each microservice owns its data and state, having a complete view over the complete system's data state requires manually making a snapshot of all services' individual databases and importing those in a central database to perform analytics. Such a process cannot be done without special software and protocols; naively dumping all individual databases into a data warehouse without taking down all services in advance will most certainly result into data inconsistencies.

Debugging & Auditing. Developing microservices using event-sourcing and Command Query Responsibility Segregation (CQRS) [5] allows developers to replay message exchanges between services and debug their applications by rebuilding the state of an application at the time that a bug occurred.

3.2 Distributed OLTP Systems

Distributed OnLine Transaction Processing (OLTP) systems such as VoltDB [21] and H-Store [15], can be used as a backend for EDAS since they provide scalability, consistency, global view of the application state, and analytics. However, they introduce a number of issues. First, OLTP systems require that transactional code is included in the database as a stored procedure. However, transactional code is often an indivisible part of application code, a microservice's business logic for instance. Pushing the business logic into the database is largely disliked [2]. Second, distributed OLTP systems partition their state as they see fit, yet in order to achieve low latency, the state of each service should be locally available and even in the same memory space as the service itself. Finally, having one distributed database that manages the state of all applications goes against loose-coupling and requires that different services agree on a given schema, database system, etc.

3.3 Actor programming frameworks

Actors are good examples of loosely-coupled systems, that can be reconfigured in the presence of failures and environment changes. In addition, effective debugging can be achieved by use of the event sourcing and CQRS patterns, like in reactive microservices. Erlang, Akka, and Microsoft Orleans are popular actor-based programming frameworks. In the rest of this section, we focus on Orleans, which offers the highest level of abstraction among actor-based systems.

Transactions. Actors in general do not provide means for executing distributed transactions. Orleans appears to be implementing some form of ACID transactions with 2PC based on batching [10].

Global State View & Analytics. Actors can offer state locality but they lack the capability to provide a consolidated view of the global state and perform analytics on that state. For instance, Orleans can save the persistent state of actors locally or in inexpensive cloud storage that can be replicated for scalability and fault tolerance. Adopting the encapsulation principle, each actor has local access to its own data and state and restricted access to the state of other actors as per their public interface. This organization works for established actor-local operations, but leaves much to be desired in terms of a consistent global state view that can be used to answer queries on the complete state of an application. Performing analytics on global state in an actor system would be similar to a microservices.

3.4 Stream Processing Systems

So far stream processors are primarily known for their capacity to support high-throughput and low-latency analytics. However, modern stream processors such as Apache Flink [7], also support distributed state consistency via exactly-once state processing guarantees. Stream processors can serve as a platform for running EDAS in a scalable [16] and consistent fashion [6]. We argue that additional research needs to be performed in order for stream processors to be able to satisfy the operational requirements of EDAS, namely transactions, query-able global state, and loose coupling at the API level. Current stream processors lack appropriate programming models that allow developing microservice architectures. At present, they only offer functional APIS focused on bulk event processing, rather than message exchange that allows for loose microservice coupling. Moreover, stream processor systems need transactional facilities to support advanced business logic and coordination. The only two exceptions that are available at the time of writing are i) a closed-source implementation of multi-key transactions in Apache Flink, as well as S-Store [19], which provides ACID guarantees on shared mutable state on a single machine. Having a global state view of an application is also missing. Apache Flink, for instance, only supports external querying of a single operator's state.

Finally, stream processors can serve as a good basis for debugging distributed services. For instance, message brokers, such as Apache Kafka [17], can be used for storing all messages that are exchanged among microservices and replaying them during debugging. Moreover, Apache Flink's savepoints[3] can be used to replay events from a specific past consistent state of the stream topology in order to debug an application.

4 THE ROAD AHEAD

This section summarizes the research directions our community needs to take in order to realize the operational stream processing vision. We envision EDAS to be authored in a service-oriented API. Such an API would enable a service to have custom business logic, to communicate with other services via (a)synchronous message exchange, and to have access to local state.

A set of services authored in that API can be compiled into a dataflow graph, an example of which is depicted in Figure 1. Edges represent channels of message exchange among services. Vertices execute custom business logic, and are given access to managed local state. The dataflow engine takes care of the state's consistency, as well as the routing and exactly-once processing of messages within services. For the services executing in the dataflow graph to send and receive requests to/from external systems, they have to write or read to a message queue/log (e.g., Apache Kafka). This, not only enables exactly-once processing and delivery of messages, but it can also hide failures from the application programmers who can assume that, if a message has been sent, their application will receive an answer to that message exactly-once. Finally, certain parts of the inputs can be replayed in order to debug/audit a given service deployment.

One would argue that such an architecture could be implemented on existing systems such as Apache Flink, Samza, or Kafka. However, a set of requirements that we introduced in the previous section remain unsatisfied and require further research.

Transactions. Current dataflow systems guarantee consistency of single-event changes on a given partition of state. In order to guarantee consistency during multi-key, multi-partition state changes, we need to extend existing approaches of consistent snapshots [6] drawing ideas from distributed systems [9], and traditional OLTP systems. Some form of timestamp-based concurrency control could be employed [4, 13], especially given that processing- or event-time is a first-class citizen of all events entering a stream processor. Furthermore, we can invent 2PC-like protocols that take advantage of the FIFO connections between computation vertices in dataflow graphs, and ensure the order in which transactions arrive and state changes are applied.

[3] https://ci.apache.org/projects/flink/flink-docs-stable/ops/state/savepoints.html

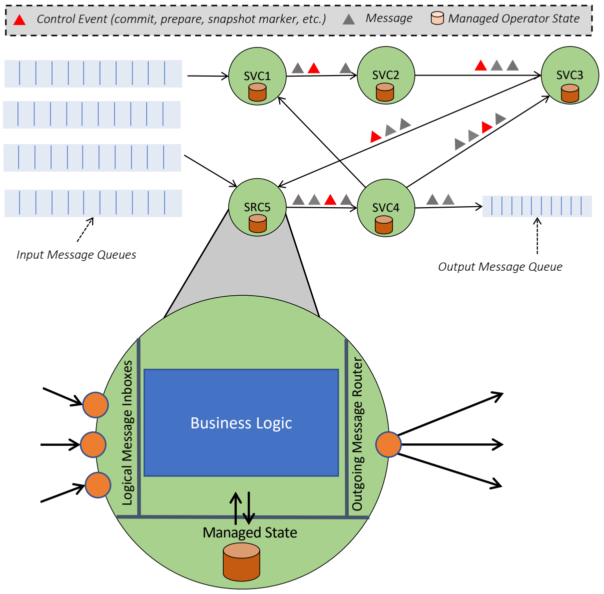


Figure 1: A set of services running inside a streaming dataflow graph. The business logic of services runs as an operator that processes messages and produces responses, the state of the service is managed by the dataflow engine, and all inputs and outputs of a given system are logged.

(Non-Dataflow) APIs for Loosely-coupled Services. Current dataflow systems only allow developers to author data pipelines, by defining data dependencies among operators and user-defined functions such as Map and Reduce in an explicit manner. To allow developers to develop loosely-coupled data-intensive services, we need novel APIs which will allow developers to individually develop, test, and debug services. Those services can be automatically compiled into a single, efficient, and scalable dataflow graph. To this end, we can derive the data dependencies from the defined messages/endpoints that applications send to each other, building the dataflow graph dynamically.

Consistent Access to Global State. Consistent snapshots [6] of transactional dataflows need to provide safe and consistent read access to global state, i.e., we need the means to execute distributed queries over the state of different operators. More specifically, we could provide the means for services to publish (views over) their state on top of which other services can build materialized views. Materialized views can maintain fresh results [12] and guarantee locality.

Dynamic (re)configuration. Currently, dataflow systems build a dataflow graph statically and then parallelize and deploy it, since all the data dependencies among operators are pre-defined. However, frequent and independent updates of services necessitate highly dynamic graphs with network channels that can be created or destroyed at any given time during the execution of a service. The ultimate goal is that the performance of existing services is not affected by changes in the dataflow graph. An important use case of dynamic configuration is the automatic parallelization of services. As we mentioned earlier, each service obtains access to managed local state. However, both the state and the input messages that are directed to a given service should be partitioned whenever possible, in order to ensure parallel execution. To this end, given a key for each state object, we aim at automatically parallelizing and even replicating service deployments and optimizing them for throughput and latency, without sacrificing consistency.

5 CONCLUSIONS & FUTURE WORK

In this paper we made a case for using streaming dataflow systems as a backend for stateful event-driven applications, such as microservices. We listed a set of requirements that include ACID transactions, global state consolidation, and the need for debugging and auditing. We then used those requirements to draw a rough outline of the future work that we believe has to take place, to materialize the vision of operational stream processing.

REFERENCES

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

[2] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *SIGMOD '15*.

[3] P. A. Bernstein and S. Bykov. 2016. Developing Cloud Services Using the Orleans Virtual Actor Model. *IEEE Internet Computing* 20, 5 (2016).

[4] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (June 1981).

[5] Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. 2013. *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure* (1st ed.). Microsoft patterns & practices.

[6] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *PVLDB* 10, 12 (2017).

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink®: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015).

[8] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD*. ACM.

[9] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (1985).

[10] Tamer Eldeeb and Phil Bernstein. 2016. *Transactions for Distributed Actors in the Cloud.* Technical Report.

[11] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *SIGMOD '87*.

[12] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *OSDI '18*.

[13] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An evaluation of distributed concurrency control. *PVLDB* 10, 5 (2017).

[14] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR '19*.

[15] Robert Kallman, Hideaki Kimura, and Jonathan Natkins et al. 2008. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *PVLDB* 1, 2 (Aug. 2008).

[16] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *ICDE '18*.

[17] Martin Kleppmann and Jay Kreps. 2015. Kafka, Samza and the Unix Philosophy of Distributed Data. *IEEE Data Eng. Bull.* 38, 4 (2015), 4–14.

[18] X/Open Company Ltd. 1991. "Distributed Transaction Processing: The XA specification", X/Open CAE Specification.

[19] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, et al. 2015. S-Store: streaming meets transaction processing. *PVLDB* 8, 13 (2015).

[20] Sam Newman. 2015. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.".

[21] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36 (2013), 21–27.

[22] Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent Programming in ERLANG (2Nd Ed.).* Prentice Hall International (UK) Ltd.

[23] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *USENIX ATC '18*.

Metropolis-Hastings Algorithms for Estimating Betweenness Centrality

Mostafa Haghir Chehreghani
LTCI, Télécom ParisTech
Paris
mostafa.chehreghani@gmail.com

Talel Abdessalem
LTCI, Télécom ParisTech
Paris
talel.abdessalem@telecom-paristech.fr

Albert Bifet
LTCI, Télécom ParisTech
Paris
albert.bifet@telecom-paristech.fr

ABSTRACT

Recently, an optimal probability distribution was proposed to sample vertices for estimating betweenness centrality, that yields the minimum approximation error. However, it is computationally expensive to directly use it. In this paper, we investigate exploiting Metropolis-Hastings technique to sample based on this distribution. As a result, first given a network G and a vertex $r \in V(G)$, we propose a Metropolis-Hastings MCMC algorithm that samples from the space $V(G)$ and estimates betweenness score of r. The stationary distribution of our MCMC sampler is the optimal distribution. We also show that our MCMC sampler provides an (ϵ, δ)-approximation. Then, given a network G and a set $R \subset V(G)$, we present a Metropolis-Hastings MCMC sampler that samples from the joint space R and $V(G)$ and estimates relative betweenness scores of the vertices in R. We show that for any pair $r_i, r_j \in R$, the ratio of the expected values of the estimated relative betweenness scores of r_i and r_j with respect to each other is equal to the ratio of their betweenness scores. We also show that our joint-space MCMC sampler provides an (ϵ, δ)-approximation of the relative betweenness score of r_i with respect to r_j.

1 INTRODUCTION

Centrality is a structural property of vertices (or edges) in a network that quantifies their relative importance. For example, it determines the importance of a person within a social network, or a road within a road network. Freeman [14] introduced and defined *betweenness centrality* of a vertex as the number of shortest paths from all (source) vertices to all others that pass through that vertex. He used it for measuring the control of a human over the communications among others in a social network [14]. Betweenness centrality is also used in some well-known algorithms for clustering and community detection in social and information networks [16].

Although there exist polynomial time and space algorithms for betweenness centrality computation, the algorithms are expensive in practice. Currently, the most efficient existing exact method is Brandes's algorithm [5]. Time complexity of this algorithm is $O(nm)$ for unweighted graphs and $O(nm + n^2 \log n)$ for weighted graphs with positive weights, where n and m are the number of vertices and the number of edges of the network, respectively. This means exact betweenness centrality computation is not applicable, even for mid-size networks. However, there exist observations that may improve the computation of betweenness scores in practice.

- First, in several applications it is sufficient to compute betweenness score of only one or a few vertices. For instance, this index might be computed for only core vertices of communities [23] in social/information networks or for only hubs in communication networks. Chehreghani [9] has discussed some situations where it is required to compute betweenness score of only one vertex. Note that these vertices are not necessarily those that have the highest betweenness scores. Hence, algorithms that identify vertices with the highest betweenness scores [21] are not applicable. While exact computation of this index for one vertex is not easier than that for all vertices, Chehreghani [9] and later Riondato and Kornaropoulos [21] respectively showed that this index can be estimated more effectively for one arbitrary vertex and for k vertices that have the highest scores.
- Second, in practice, instead of computing betweenness scores, it is usually sufficient to *compute betweenness ratios* or *rank vertices* according to their betweenness scores [21]. For example, Daly and Haahr [12] exploited betweenness ratios for finding routes that provide good delivery performance and low delay in Mobile Ad hoc Networks. The other application is handling cascading failures [1].

While the above mentioned observations do not yield a better algorithm when exact betweenness scores are used, they may improve approximate algorithms. In the current paper, we exploit both of these observations to design more effective approximate algorithms. In the first problem studied in this paper, we assume that we are given a vertex $r \in V(G)$ and we want to estimate its betweenness score. In the second problem, we assume that we are given a set $R \subset V(G)$ and we want to estimate the ratios of betweenness scores of vertices in R. The second problem is formally defined as follows: given a graph G and a set $R \subset V(G)$, for any two vertices r_i and r_j in R, we want to estimate the *relative betweenness score* of r_i with respect to r_j, denoted by $BC_{r_j}(r_i)$ (see Equation 8 of Section 4.3 for the formal definition of relative betweenness score). The ratio of the expected values of our estimations of $BC_{r_j}(r_i)$ and $BC_{r_i}(r_j)$ is equal to the ratio of betweenness scores of r_i and r_j.

In [9], Chehreghani presented the optimal probability distribution for estimating betweenness centrality, that yields the minimum approximation error 0. However, this distribution cannot be directly used, as computing the constant factor of probability densities is computationally expensive. A natural solution for this problem is to use Metropolis-Hastings sampling [18]. In this short paper, we investigate the possibility of using such a sampling method for the two aforementioned problems and theoretically analyze the resulted algorithms. More precisely, our key contributions are as follows.

- Given a graph G and a vertex $r \in V(G)$, in order to estimate betweenness score of r, we develop an MCMC sampler

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

that samples from the space $V(G)$. Unlike existing work, our samples are non-iid and the stationary distribution of our MCMC sampler is the *optimal probability distribution* [9]. We also show that our MCMC sampler provides an (ϵ, δ)-approximation of the betweenness score of r ($\epsilon \in \mathbb{R}^+$ and $\delta \in (0, 1)$).
- Given a graph G and a set $R \subset V(G)$, in order to estimate relative betweenness scores of all pairs of vertices in R, we develop an MCMC sampler that samples from the joint space R and $V(G)$. This means each sample (state) in our MCMC sampler is a pair $\langle \mathfrak{r}, \mathfrak{v} \rangle$, where $\mathfrak{r} \in R$ and $\mathfrak{v} \in V(G)$. For any two vertices $r_i, r_j \in R$, we show that our joint-space MCMC sampler provides an (ϵ, δ)-approximation of the relative betweenness score of r_i with respect to r_j.

Techniques similar to our algorithm (the second algorithm that estimate *relative* betweenness centrality) have already been used in *statistical physics* to estimate *free energy differences* [3]. However, they are new in the context of network analysis. Our current work takes the first step in bridging these two domains. This step can be further extended by proposing algorithms similar to our work for estimating other network indices. As a result, a novel family of techniques might be introduced to the field of network analysis. We leave efficient implementations of our proposed algorithms and evaluating their empirical efficiency for future work.

2 PRELIMINARIES

Throughout the paper, G refers to a graph (network). For simplicity and without loss of generality, we assume that G is an undirected, connected and loop-free graph without multi-edges. Also, we assume that G is an unweighted graph, unless it is explicitly mentioned that G is weighted. $V(G)$ and $E(G)$ refer to the set of vertices and the set of edges of G, respectively. For a vertex $v \in V(G)$, by $G \setminus v$ we refer to the set of connected graphs generated by removing v from G. A *shortest path* between two vertices $u, v \in V(G)$ is a path whose size is minimum, among all paths between u and v. For two vertices $u, v \in V(G)$, we use $d(u, v)$, to denote the size (the number of edges) of a shortest path connecting u and v. By definition, $d(u, u) = 0$ and $d(u, v) = d(v, u)$. For $s, t \in V(G)$, σ_{st} denotes the number of shortest paths between s and t, and $\sigma_{st}(v)$ denotes the number of shortest paths between s and t that also pass through v. *Betweenness centrality* of a vertex v is defined as:

$$BC(v) = \frac{1}{|V(G)| \cdot (|V(G)| - 1)} \sum_{s,t \in V(G) \setminus \{v\}} \frac{\sigma_{st}(v)}{\sigma_{st}}.$$

A notion which is widely used for counting the number of shortest paths in a graph is the directed acyclic graph (DAG) containing all shortest paths starting from a vertex s (see e.g., [5]). In this paper, we refer to it as the *shortest-path-DAG*, or *SPD* in short, rooted at s. For every vertex s in graph G, the *SPD* rooted at s is unique, and it can be computed in $O(|E(G)|)$ time for unweighted graphs and in $O\left(|E(G)| + |V(G)| \log |V(G)|\right)$ time for weighted graphs with positive weights [5]. Brandes [5] introduced the notion of the *dependency score* of a vertex $s \in V(G)$ on a vertex $v \in V(G) \setminus \{s\}$, which is defined as: $\delta_{s\bullet}(v) = \sum_{t \in V(G) \setminus \{v,s\}} \delta_{st}(v)$, where $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$. We have: $BC(v) = \frac{1}{|V(G)| \cdot (|V(G)|-1)} \sum_{s \in V(G) \setminus \{v\}} \delta_{s\bullet}(v)$.

A Markov chain is a sequence of dependent random variables (states) such that the probability distribution of each variable given the past variables depends only on the last variable. An MCMC has *stationary distribution* if the conditional distribution of the $k + 1^{\text{th}}$ state given the k^{th} state does not depend on k. Let $\mathbb{P}[x]$ be a probability distribution defined on the random variable x. When the function $f(x)$, which is proportional to the density of $\mathbb{P}[x]$, can be efficiently computed, the Metropolis-Hastings algorithm is used to draw samples from $\mathbb{P}[x]$. In a simple form (with symmetric proposal distribution), the Metropolis-Hastings algorithm first chooses an arbitrary initial state x_0. Then, iteratively: i) let x be the current state. It generates a candidate x' using the *proposal distribution* $q(x'|x)$, and ii) it moves from x to x' with probability $\min\left\{1, \frac{f(x')}{f(x)}\right\}$. The *proposal distribution* $q(x'|x)$ defines the conditional probability of proposing a state x' given the state x. In the *Independence Metropolis-Hastings algorithm*, $q(x'|x)$ is independent of x, i.e., $q(x'|x) = g(x')$.

3 RELATED WORK

Brandes [5] introduced an efficient algorithm for computing betweenness centrality of a vertex, which is performed in $O(|V(G)||E(G)|)$ and $O(|V(G)||E(G)| + |V(G)|^2 \log |V(G)|)$ times for unweighted and weighted networks with positive weights, respectively. Çatalyürek et. al. [7] presented the *compression* and *shattering* techniques to improve efficiency of Brandes's algorithm for large graphs. The two natural extension of betweenness centrality to sets of vertices are *group betweenness centrality* [13] and *co-betweenness centrality* [8]. Brandes and Pich [6] and Bader et.al. [2] proposed approximate algorithms based on selecting k source vertices and computing dependency scores of them on the other vertices in the graph. To estimate betweenness score of vertex v, Chehreghani [9] presented a non-uniform sampler, defined as follows: $\mathbb{P}[s] = \frac{1/d(v,s)}{\sum_{u \in V(G) \setminus \{v\}} 1/d(v,u)}$, where $s \in V(G) \setminus \{v\}$. Similar to these algorithms, our proposed algorithms are *source vertex samplers*, too. However, they use a new mechanism for sampling which is based on the Metropolis-Hastings algorithm. Riondato and Upfal [22] introduced a *pair sampler* for estimating betweenness scores of all (or top-k) vertices in a graph. Riondato and Kornaropoulos [21] and Borassi and Natale [4] presented *shortest path samplers* for estimating betweenness centrality of all vertices or the k vertices that have the highest betweenness scores. The algorithm of [4] uses balanced bidirectional BFS (bb-BFS) to sample shortest paths. In bb-BFS, a BFS is performed from each of the two endpoints s and t, in such a way that they are likely to explore about the same number of edges. Finally, Chehreghani et.al. [11] presented exact and approximate algorithms for computing betweenness centrality in directed graphs.

4 MCMC ALGORITHMS FOR ESTIMATING BETWEENNESS CENTRALITY

In this section, we present our MCMC sampler for estimating betweenness score of a single vertex; and our joint-space MCMC sampler for estimating relative betweenness scores of vertices in a given set.

4.1 Betweenness centrality as a probability distribution

Chehreghani [9] presented a randomized algorithm that admits a probability mass function as an input parameter. Then, he proposed an optimal sampling technique that computes betweenness score of a vertex $r \in V(G)$ with error 0. In optimal sampling, each vertex v is chosen with probability

$$\mathbb{P}_r\left[v\right] = \frac{\delta_{v\bullet(r)}}{\sum_{v'\in V(G)} \delta_{v'\bullet(r)}} \tag{1}$$

In other words, for estimating betweenness score of vertex r, each source vertex $v \in V(G)$ whose dependency score on r is greater than 0, is chosen with probability $\mathbb{P}\left[v\right]$ defined in Equation 1.

In the current paper, for $r \in V(G)$ we want to estimate $BC(r)$ and also for all pairs of vertices r_i, r_j in a set $R \subset V(G)$, the ratios $\frac{BC(r_i)}{BC(r_j)}$. For this purpose, we follow a *source vertex sampling* procedure where for each vertex r, we consider $\mathbb{P}_r\left[\cdot\right]$ defined in Equation 1 as the target probability distribution used to sample vertices $v \in V(G)$. It is, however, computationally expensive to calculate the normalization constant $\sum_{v'\in V(G)} \delta_{v'\bullet(r)}$ in Equation 1, as it gives the betweenness score of r. However, for two vertices $v_1, v_2 \in V(G)$, it might be feasible to compute the ratio $\frac{\mathbb{P}_r[v_1]}{\mathbb{P}_r[v_2]} = \frac{\delta_{v_1\bullet}(r)}{\delta_{v_2\bullet}(r)}$, as it can be done in $O(|E(G)|)$ time for unweighted graphs and in $O(|E(G)| + |V(G)| \log |V(G)|)$ time for weighted graphs with positive weights. This motivates us to propose Metropolis-Hastings sampling algorithms that for a vertex r, sample each vertex $v \in V(G)$ with the probability distribution $\mathbb{P}_r\left[v\right]$ defined in Equation 1.

4.2 A single-space MCMC sampler

In this section, we propose an MCMC sampler, defined on the space $V(G)$, to estimate betweenness centrality of a single vertex r. Our MCMC sampler consists of the following steps:

- First, we choose a vertex $\mathfrak{v}_0 \in V(G)$, as the initial state, uniformly at random.
- Then, at each iteration t, $1 \le t \le T$:
  - Let $\mathfrak{v}(t)$ be the current state of the chain.
  - We choose vertex $\mathfrak{v}'(t) \in V(G)$, uniformly at random.
  - With probability $\min\left\{1, \frac{\delta_{\mathfrak{v}'(t)\bullet}(r)}{\delta_{\mathfrak{v}(t)\bullet}(r)}\right\}$ we move from state $\mathfrak{v}(t)$ to the state $\mathfrak{v}'(t)$.

The sampler is an iterative procedure where at each iteration t, one transition may occur in the Markov chain. Let M be the multi-set (i.e., the set where repeated members are allowed) of samples (states) accepted by our sampler. In the end of sampling, betweenness score of r is estimated as

$$\ddot{BC}(r) = \frac{1}{(T+1)(|V(G)|-1)} \sum_{v\in M} \sum_{u\in V(G)\setminus\{v\}} \frac{\sigma_{vu}(r)}{\sigma_{vu}}. \tag{2}$$

This estimation does not give an unbiased estimation of $BC(r)$, however as we discuss below, by increasing T, $\ddot{BC}(r)$ can become arbitrarily close to $BC(r)$. In the rest of this section, we show that our MCMC sampler provides an (ϵ, δ)-approximation of $BC(r)$, where $\epsilon \in \mathbb{R}^+$ and $\delta \in (0, 1)$.

Theorem 4.1. *Let $\overline{\delta(r)}$ be the average of dependency scores of vertices in $V(G)$ on r, i.e., $\overline{\delta(r)} = \frac{\sum_{v\in V(G)} \delta_{v\bullet}(r)}{|(V(G)|}$, and $\Delta(r)$ be the maximum dependency score that a vertex in G has on r. Let also $\mu(r)$ denote $\frac{\Delta(r)}{\overline{\delta(r)}}$. Then, for a given $\epsilon \in \mathbb{R}^+$, by our MCMC sampler and starting from any arbitrary initial state, we have*

$$\mathbb{P}\left[|\ddot{BC}(r) - BC(r)| > \epsilon\right] \le 2\exp\left\{-\frac{T}{2}\left(\frac{2\epsilon(|V(G)|-1)}{\mu(r)\Delta(r)} - \frac{3}{T}\right)^2\right\}. \tag{3}$$

Due to space limitations, in this short paper we omit all the proofs. However, the interested reader may find them in a longer version of this text in [10]. In general, our (ϵ, δ)-approximation proofs are based on a theorem presented in [17] for the concentration analysis of MCMC samples and a theorem presented in [19] for the uniformly ergodicity of Independence Metropolis-Hastings algorithms.

Note that Inequality 3 does not depend on the initial state. Furthermore, in Inequality 3 it is not required to discard an initial part of the chain, called *burn-in*. More details on this can be found in [10]. T is usually large enough so that we can approximate $\frac{3}{T}$ by 0. Hence, Inequality 3 yields that for given values $\epsilon \in \mathbb{R}^+$ and $\delta \in (0, 1)$, if T is chosen such that

$$T \ge \frac{\mu(r)^2 \Delta(r)^2}{2\epsilon^2(|V(G)|-1)^2} \ln \frac{2}{\delta} \tag{4}$$

our MCMC sampler will estimate the betweenness score of r within an additive error ϵ with a probability at least $1 - \delta$.

4.3 A joint-space MCMC sampler

In this section, we present an MCMC sampler to estimate the ratios of betweenness scores of the vertices in a set $R \subset V(G)$. Each state of this sampler is a pair $(\mathfrak{r}, \mathfrak{v})$, where $\mathfrak{r} \in R$ and $\mathfrak{v} \in V(G)$. Since this sampler is defined on the joint space R and $V(G)$, we refer to it as *joint-space MCMC sampler*. Given a state $\mathfrak{s}$ of the chain, we denote by $\mathfrak{s}.\mathfrak{r}$ the first element of $\mathfrak{s}$, which is a vertex in R; and by $\mathfrak{s}.\mathfrak{v}$ the second element of $\mathfrak{s}$, which is a vertex in $V(G)$.

Our joint-space MCMC sampler consists of the following steps:

- First, we choose a pair $\langle \mathfrak{r}_0, \mathfrak{v}_0 \rangle$, as the initial state, where $\mathfrak{r}_0$ and $\mathfrak{v}_0$ are chosen uniformly at random from R and $V(G)$, respectively.
- Then, at each iteration t, $1 \le t \le T$:
  - Let $\mathfrak{s}(t)$ be the current state of the chain.
  - We choose elements $\mathfrak{r}(t) \in R$ and $\mathfrak{v}(t) \in V(G)$, uniformly at random.
  - With probability $\min\left\{1, \frac{\delta_{\mathfrak{v}(t)\bullet}(\mathfrak{r}(t))}{\delta_{\mathfrak{s}(t).\mathfrak{v}\bullet}(\mathfrak{s}(t).\mathfrak{r})}\right\}$ we move from state $\mathfrak{s}(t)$ to the state $\langle \mathfrak{r}(t), \mathfrak{v}(t) \rangle$.

Techniques similar to our joint-space MCMC sampler have been used in *statistical physics* to estimate *free energy differences* [3]. Our joint-space MCMC sampler is a Metropolis-Hastings algorithm that possesses a *unique stationary distribution* [15, 20] defined as follows:

$$\mathbb{P}\left[r, v\right] = \frac{\delta_{v\bullet}(r)}{\sum_{r'\in R} \sum_{v'\in V(G)} \delta_{v'\bullet}(r')}. \tag{5}$$

All samples that have a specific value r for their $\mathfrak{r}$ component form an Independence Metropolis-Hastings chain that possesses the stationary distribution defined in Equation 1. Samples drawn by our MCMC and joint-space MCMC samplers are non-iid. In Theorem 4.2, we show how our joint-space MCMC sampler can be used to estimate the ratios of betweenness scores of the vertices in R.

Theorem 4.2. *In our joint-space MCMC sampler, for any two vertices $r_i, r_j \in R$, we have:*

$$\frac{BC(r_i)}{BC(r_j)} = \frac{\mathbb{E}_{\mathbb{P}_{r_j}[v]}\left[\min\left\{1, \frac{\delta_{v\bullet}(r_i)}{\delta_{v\bullet}(r_j)}\right\}\right]}{\mathbb{E}_{\mathbb{P}_{r_i}[v]}\left[\min\left\{1, \frac{\delta_{v\bullet}(r_j)}{\delta_{v\bullet}(r_i)}\right\}\right]} \tag{6}$$

where $\mathbb{E}_{\mathbb{P}_{r_i}[v]}$ (respectively $\mathbb{E}_{\mathbb{P}_{r_j}[v]}$) denotes the expected value with respect to $\mathbb{P}_{r_i}\left[v\right]$ (respectively $\mathbb{P}_{r_j}\left[v\right]$).

The proof of Theorem 4.2 is based on the *detailed balance property* of Metropolis-Hastings algorithms and can be found in [10].

Let $r_i, r_j \in R$, and $M(i)$ and $M(j)$ be the multi-sets of samples taken by our joint-space MCMC sampler whose $\mathfrak{r}$ components are respectively r_i and r_j. Equation 6 suggests to estimate $\frac{BC(r_i)}{BC(r_j)}$ as the ratio:

$$\frac{\frac{1}{|M(j)|} \times \sum_{\mathfrak{s} \in M(j)} \min \left\{1, \frac{\delta_{\mathfrak{s}.\mathfrak{v}}(r_i)}{\delta_{\mathfrak{s}.\mathfrak{v}}(r_j)}\right\}}{\frac{1}{|M(i)|} \times \sum_{\mathfrak{s} \in M(i)} \min \left\{1, \frac{\delta_{\mathfrak{s}.\mathfrak{v}}(r_j)}{\delta_{\mathfrak{s}.\mathfrak{v}}(r_i)}\right\}}. \tag{7}$$

We use Equation 7 to estimate the *ratio of the betweenness scores* of r_i and r_j. We then define the *relative betweenness score* of r_i with respect to r_j, denoted by $BC_{r_j}(r_i)$, as follows:

$$BC_{r_j}(r_i) = \frac{1}{|V(G)|} \sum_{v \in V(G)} \min \left\{1, \frac{\delta_{v \bullet}(r_i)}{\delta_{v \bullet}(r_j)}\right\}. \tag{8}$$

When we want to compare betweenness centrality of vertices r_i and r_j, using *relative betweenness score* makes more sense than using the *ratio of betweenness scores*. In relative betweenness centrality, for each $v \in V(G)$, the ratio of the dependency scores of v on r_i and r_j is computed and in the end, all the ratios are summed. Hence, for each vertex v independent from the others, the effects of r_i and r_j on the shortest paths starting from v are examined. Note that the notion of *relative betweenness score* can be further extended and presented as follows:

$$BC_{r_j}(r_i) = \frac{\sum_{v \in V(G)} \sum_{t \in V(G) \setminus \{v\}} \min \left\{1, \frac{\delta_{vt}(r_i)}{\delta_{vt}(r_j)}\right\}}{|V(G)| \cdot (|V(G)| - 1)}.$$

In the following, we show that the numerator of Equation 7, i.e.,

$$\frac{1}{|M(j)|} \sum_{\mathfrak{s} \in M(j)} \min \left\{1, \frac{\delta_{\mathfrak{s}.\mathfrak{v}}(r_i)}{\delta_{\mathfrak{s}.\mathfrak{v}}(r_j)}\right\},$$

can accurately estimate $BC_{r_j}(r_i)$. We refer to this value as $\ddddot{BC}_{r_j}(r_i)$. For a pair of vertices $r_i, r_j \in R$, in Theorem 4.3 we derive an error bound for $\ddddot{BC}_{r_j}(r_i)$.

Theorem 4.3. *Let $r_i, r_j \in R$, $M(j)$ be the multi-set of samples whose $\mathfrak{r}$ components are r_j, and $\overline{\delta(r_j)}$ be the average of dependency scores of vertices in $V(G)$ on r_j, i.e., $\overline{\delta(r_j)} = \frac{\sum_{v \in V(G)} \delta_{v \bullet}(r_j)}{|(V(G)|}$. Suppose that there exists some value $\mu(r_j)$ such that for each vertex $v \in V(G)$, the following holds: $\delta_{v \bullet}(r_j) \leq \mu(r_j) \times \overline{\delta(r_j)}$. Then, for a given $\epsilon \in \mathbb{R}^+$, by our joint-space MCMC sampler and starting from any arbitrary initial state, we have*

$$\mathbb{P}\left[|\ddddot{BC}_{r_j}(r_i) - BC_{r_j}(r_i)| > \epsilon\right] \leq 2 \exp \left\{-\frac{|M(j)| - 1}{2} \left(\frac{2\epsilon}{\mu(r_i)} - \frac{3}{|M(j)| - 1}\right)^2\right\}. \tag{9}$$

Similar to Inequality 3, Inequality 9 does not depend on the initial state and it holds without need for burn-in. Furthermore, for given values $\epsilon \in \mathbb{R}^+$ and $\delta \in (0, 1)$, if we have

$$|M(j)| \geq \frac{\mu(r_j)^2}{2\epsilon^2} \ln \frac{2}{\delta},$$

then our joint-space MCMC sampler can estimate relative betweenness score of r_i with respect to r_j within an additive error ϵ with a probability at least $1 - \delta$.

5 CONCLUSION

In this paper, first given a network G and a vertex $r \in V(G)$, we proposed a Metropolis-Hastings MCMC algorithm that samples from the space $V(G)$ and estimates betweenness score of r. We showed that our MCMC sampler provides an (ϵ, δ)-approximation. Then, given a network G and a set $R \subset V(G)$, we presented a Metropolis-Hastings MCMC sampler that samples from the joint space R and $V(G)$ and estimates relative betweenness scores of the vertices in R. We showed that for any pair $r_i, r_j \in R$, the ratio of the expected values of the estimated relative betweenness scores of r_i and r_j with respect to each other is equal to the ratio of their betweenness scores. We also showed that our joint-space MCMC sampler provides an (ϵ, δ)-approximation of the relative betweenness score of r_i with respect to r_j. We leave efficient implementations of our proposed algorithms and evaluating their empirical efficiency for future work.

ACKNOWLEDGMENTS

This work has been funded by the ANR project IDOLE.

REFERENCES

[1] Manas Agarwal, Rishi Ranjan Singh, Shubham Chaudhary, and Sudarshan Iyengar. 2014. Betweenness Ordering Problem : An Efficient Non-Uniform Sampling Technique for Large Graphs. *CoRR* abs/1409.6470 (2014). http://arxiv.org/abs/1409.6470

[2] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. 2007. Approximating betweenness centrality. In *WAW*. 124–137.

[3] C. H. Bennett. 1976. Efficient estimation of free energy differences from Monte-Carlo data. *J. Comput. Phys.* 22 (1976), 245.

[4] Michele Borassi and Emanuele Natale. 2016. KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation. In *ESA*. 20:1–20:18.

[5] U. Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology* 25, 2 (2001), 163–177.

[6] U. Brandes and C. Pich. 2007. Centrality estimation in large networks. *Intl. Journal of Bifurcation and Chaos* 17, 7 (2007), 303–318.

[7] Ümit V. Çatalyürek, Kamer Kaya, Ahmet Erdem Sariyüce, and Erik Saule. 2013. Shattering and Compressing Networks for Betweenness Centrality. In *Proceedings of the 13th SIAM International Conference on Data Mining*. 686–694.

[8] Mostafa Haghir Chehreghani. 2014. Effective co-betweenness centrality computation. In *Seventh ACM International Conference on Web Search and Data Mining*. 423–432.

[9] Mostafa Haghir Chehreghani. 2014. An Efficient Algorithm for Approximate Betweenness Centrality Computation. *Comput. J.* 57, 9 (2014), 1371–1382.

[10] Mostafa Haghir Chehreghani, Talel Abdessalem, and Albert Bifet. 2017. Metropolis-Hastings Algorithms for Estimating Betweenness Centrality in Large Networks. *CoRR* abs/1704.07351 (2017). arXiv:1704.07351 http://arxiv.org/abs/1704.07351

[11] Mostafa Haghir Chehreghani, Albert Bifet, and Talel Abdessalem. 2018. Efficient Exact and Approximate Algorithms for Computing Betweenness Centrality in Directed Graphs. In *Advances in Knowledge Discovery and Data Mining (PAKDD)*. 752–764.

[12] Elizabeth M. Daly and Mads Haahr. [n. d.]. Social Network Analysis for Information Flow in Disconnected Delay-Tolerant MANETs. *IEEE Trans. Mob. Comput.* 8, 5 ([n. d.]), 606–621.

[13] M. Everett and S. Borgatti. 1999. The centrality of groups and classes. *Journal of Mathematical Sociology* 23, 3 (1999), 181–201.

[14] L. C. Freeman. 1977. A set of measures of centrality based upon betweenness, Sociometry. *Social Networks* 40 (1977), 35–41.

[15] W. R. Gilks, S. Richardson, and D. J. Spiegelhalter. 1996 (ISBN: 0-412-05551-1). *Markov Chain Monte Carlo in Practice.* Chapman and Hall, London.

[16] M. Girvan and M. E. J. Newman. 2002. Community structure in social and biological networks. *Natl. Acad. Sci. USA* 99 (2002), 7821–7826.

[17] Krzysztof Łatuszyński, Błażej Miasojedow, and Wojciech Niemiro. 2012. *Nonasymptotic Bounds on the Mean Square Error for MCMC Estimates via Renewal Techniques.* Springer Berlin Heidelberg, Berlin, Heidelberg, 539–555.

[18] K. L. Mengersen and R. L. Tweedie. 1996. Rates of convergence of the Hastings and Metropolis algorithms. *Ann. Statist.* 24, 1 (02 1996), 101–121.

[19] K. L. Mengersen and R. L. Tweedie. 1996. Rates of convergence of the Hastings and Metropolis algorithms. *The Annals of Statistics* 24, 1 (Feb. 1996), 101–121.

[20] S. P. Meyn and R. L. Tweedie. 1993. *Markov chains and stochastic stability.* Springer-Verlag, London.

[21] Matteo Riondato and Evgenios M. Kornaropoulos. 2016. Fast approximation of betweenness centrality through sampling. *Data Mining and Knowledge Discovery* 30, 2 (2016), 438–475.

[22] Matteo Riondato and Eli Upfal. 2016. ABRA: Approximating Betweenness Centrality in Static and Dynamic Graphs with Rademacher Averages. In *KDD*. 1145–1154.

[23] Y. Wang, Z. Di, and Y. Fan. 2011. Identifying and Characterizing Nodes Important to Community Structure Using the Spectrum of the Graph. *PLoS ONE* 6, 11 (2011), e27418.

Short Paper

From Copernicus Big Data to Extreme Earth Analytics

Visionary Paper

Manolis Koubarakis[1], Konstantina Bereta[1], Dimitris Bilidas[1], Konstantinos Giannousis[1], Theofilos Ioannidis[1], Despina-Athanasia Pantazi[1], George Stamoulis[1], Seif Haridi[2], Vladimir Vlassov[2], Lorenzo Bruzzone[3], Claudia Paris[3], Torbjørn Eltoft[4], Thomas Krämer[4], Angelos Charalabidis[5], Vangelis Karkaletsis[5], Stasinos Konstantopoulos[5], Jim Dowling[6], Theofilos Kakantousis[6] Mihai Datcu[7], Corneliu Octavian Dumitru[7], Florian Appel[8], Heike Bach[8], Silke Migdall[8], Nick Hughes[9], David Arthurs[10], Andrew Fleming[11]

[1] National and Kapodistrian University of Athens, [2] KTH Royal Institute of Technology, Stockholm
[3] University of Trento, [4] University of Tromsø, [5] National Center for Scientific Research - Demokritos
[6] LogicalClocks, [7] German Aerospace Center, [8] VISTA Remote Sensing in Geosciences GmbH
[9] Norwegian Meteorological Institute, [10] Polar View, [11] British Antarctic Survey
koubarak@di.uoa.gr

ABSTRACT

Copernicus is the European programme for monitoring the Earth. It consists of a set of systems that collect data from satellites and in-situ sensors, process this data and provide users with reliable and up-to-date information on a range of environmental and security issues. The data and information processed and disseminated puts Copernicus at the forefront of the big data paradigm, giving rise to all relevant challenges, the so-called 5 Vs: volume, velocity, variety, veracity and value. In this short paper, we discuss the challenges of extracting information and knowledge from huge archives of Copernicus data. We propose to achieve this by scale-out distributed deep learning techniques that run on very big clusters offering virtual machines and GPUs. We also discuss the challenges of achieving scalability in the management of the extreme volumes of information and knowledge extracted from Copernicus data. The envisioned scientific and technical work will be carried out in the context of the H2020 project ExtremeEarth which starts in January 2019.

1 INTRODUCTION

Copernicus is the European programme for monitoring the Earth. It consists of a set of systems that collect data from satellites and in-situ sensors, process this data and provide users with reliable and up-to-date information on a range of environmental and security issues. The Earth observation satellites that provide the data of Copernicus are the *Sentinels*, which are developed for the specific needs of the Copernicus programme, and the *contributing missions*, which are operated by national, European or international organizations. The access to Sentinel data is regulated by EU law and it is full, open and free. Information extracted from Copernicus data is made available to users through the *Copernicus services* addressing six thematic areas: land, marine, atmosphere, climate, emergency and security.

The data and information processed and disseminated puts Copernicus at the forefront of the big data paradigm, giving rise to all relevant challenges, the so-called *5 Vs*, discussed below.

Volume: The repository of Sentinel products managed by the European Space Agency (ESA) has so far published more than 5 million products, and it has more than 100 thousand users who have downloaded more than 50 PB of data since the start of the operations of the system. This volume will increase in the following years, as new Sentinel satellites are launched.

Velocity: Copernicus data has to be delivered and processed in a short time frame to allow the provision of 24/7 information to users requiring fast responses. By the end of 2016, 6 TB of data were generated and 100 TB of data were disseminated every day from the Sentinel product repository. These rates will increase in forthcoming years as new Sentinel satellites are launched.

Variety: The Sentinel satellites have different types of sensors (e.g., radar and optical) and different levels of processing (from raw data to advanced products). Moreover, datasets used for geospatial applications can be not only satellite data but also aerial imagery, in-situ data and other collateral information (e.g., public government data). This wealth of data is processed by Earth Observation actors to extract information and knowledge. This *information and knowledge is also big* and similar big data challenges apply. For example, 1PB of Sentinel data may consist of about 750.000 datasets which, when processed, about 450TB of content information and knowledge (e.g., classes of objects detected) can be generated.

Veracity: Decision-making and operations require reliable sources. Thus, assessing the quality of the data is important for the whole information extraction chain.

Value: The extraction of information from the Copernicus data has direct economic benefits for Europe. Several economic studies have concluded that the Copernicus programme has the potential to significantly impact job creation, innovation and growth. The Copernicus Market report of 2016 estimates that the overall investment in Copernicus will reach EUR 7.4 billion in the years 2008-2020, while the cumulative economic value generated by it in the same period will be around EUR 13.5 billion, and it will support 28.030 job years in the Earth Observation sector.

An important activity related to Copernicus is the *thematic exploitation platforms (TEPs)* of the European Space Agency (ESA). A TEP is a collaborative, virtual work environment addressing a class of users and providing access to EO data, algorithms and computing/networking resources required to work with them, through one coherent interface. The fundamental principle of the TEPs is to move the user to the data and tools as opposed to the traditional approach of downloading, replicating, and exploiting data “at home”. Now the user community is present and visible in the platform, involved in its governance and and enabled to share

10.5441/002/edbt.2019.88

and collaborate. There are currently 7 TEPs addressing the following application areas: coastal, forestry, hydrology, geohazards, polar, urban themes, and food security.

Another important development in the context of Copernicus is the implementation of five *Copernicus Data and Information Access Services (DIAS).* The European Commission has awarded in December 2017 four contracts to industrial consortia for the development of four cloud-based platforms for Copernicus DIAS. The fifth DIAS is built by EUMETSAT in collaboration with Mercator Ocean and the European Centre for Medium-Range Weather Forecasts. Like the TEPs, these five platforms will also bring computing resources close to the data and enable an even greater commercial exploitation of Copernicus data.

Although the TEPs and DIAS activities funded by ESA have been welcomed by the EO data user community, they both have a *significant disadvantage:* they target users that are experts in EO data and technologies, and ignore the myriad of software developers that might not be experts in EO but still have a lot to gain by integrating EO data in their applications. Therefore, *opening up the TEPs and DIASs* by extracting information and knowledge hidden in the data, publishing this information and knowledge using *linked data technologies*, and interlinking it with data in other TEPs and DIASs and other non-EO data, information and knowledge can be an important way of making the development of downstream applications easy for both EO and non-EO experts.

In the last few years, there have been four highly successful European research projects that have pursued this idea: the FP7 projects TELEIOS , LEO and Melodies , and the on-going project Copernicus App Lab . These projects pioneered the use of linked geospatial data in the EO domain, and demonstrated the potential of linked data and semantic web technologies by developing prototype environmental and business applications. However, *none of these projects has faced the challenges of big data, information and knowledge that users and application developers are facing in the context of the Copernicus program.* The above four projects have developed tools for knowledge discovery and data mining from satellite images and related geospatial data sets, as well as tools for linked geospatial data integration, querying and analytics. However, *none of these tools scales to the many PBs of data, information and knowledge present in the Copernicus context.* For example, the state-of-the art geospatial and temporal RDF store Strabon implemented in TELEIOS [15] can only handle up to 100 GBs of point data and still be able to answer simple geospatial queries (selections over a rectangular area) efficiently (in a few seconds). Competitor systems like GraphDB by company OntoText performs similarly [2]. If the complexity of geometries in the dataset increases (i.e., we have multi-polygons), not even the aforementioned performance can be achieved for both Strabon and GraphDB.

In addition, contrary to multimedia images, for which highly scalable Artificial Intelligence techniques based on deep neural network architectures have been developed by big North American companies such as Google and Facebook recently [7, 8], *similar architectures for satellite images, that can manage the extreme scale and characteristics of Copernicus data, do not exist today.* The deep neural network architectures can classify effectively and efficiently multimedia images because they have been trained using extremely large benchmark datasets consisting of millions of images (e.g., ImageNet) and have utilized the power of big data, cloud and GPU technologies. *Training datasets consisting of millions of data samples in the Copernicus context do not exist today* and published deep learning architectures for Copernicus satellite images typically run using one GPU and do not take advantage of recent advances like distributed scale-out deep learning [8].

2 MAIN OBJECTIVE AND TECHNICAL CHALLENGES

The main objective of ExtremeEarth is to go beyond the four projects mentioned above by developing *extreme Earth analytics techniques and technologies that scale to the PBs of big Copernicus data, information and knowledge, and applying these technologies in two of the ESA TEPs: Food Security and Polar.* The technologies to be developed will extend the HOPS data platform [9, 12, 13, 17] to offer unprecedented scalability to extreme data volumes and scale-out distributed deep learning for Copernicus data. The extended HOPS data platform will run on a DIAS selected after the project starts and will be available as open source to enable its adoption by the strong European Earth Observation downstream services industry.

The detailed scientific and technical challenges of ExtremeEarth are the following.

Challenge C1. To develop scalable deep learning and extreme earth analytics techniques for Copernicus big data. The constellations of Sentinel-1/2/3 satellites have the important capability to acquire long time series of multispectral and Synthetic Aperture Radar (SAR) images where the temporal dimension plays a very important role for the characterization of the information content of the image (e.g., land cover or sea ice) and its dynamics. Moreover, this results in the availability of very large archives of images covering long time periods. Another key aspect of the Sentinel missions is the multimodal structure of the platforms. Different kinds of sensors (radar, optical, multi/multispectral) are available and can be used in synergy. Each modality provides specific information that can be used to cope with the limitations of another. ExtremeEarth will advance the state of the art in this area [20] by developing distributed scale-out deep learning techniques for the classification of remote sensing images based on architectures that can effectively exploit the spatial, spectral, temporal and multimodal properties of Sentinel data.

Two deep learning architectures for the classification of Sentinel remote sensing images will be developed; one for determining crop boundaries and type, and one for sea ice mapping. These will be used in the two applications described in the Challenges A1 and A2 below. The developed algorithms will be supported by a scale-out open-source platform for distributed deep learning and big data, based on the HOPS data platform [12].

Challenge C2. To develop very large training datasets for deep learning architectures targeting the classification of Sentinel images. In deep learning architectures, the availability of large amounts of high quality training data is equally important to the learning models. Computer vision and other image processing areas have developed during the last few years huge training data sets (e.g., ImageNet) consisting of many millions of objects. Satellite remote sensing is largely lacking this development since the complexity and physical meaning of the sensor data make the generation of training datasets much more complex. Moreover, from an operational viewpoint it is not feasible to assume the availability of enough ground truth or annotated labeled data for training a deep network. In this area, the largest benchmark dataset is Eurosat which was proposed recently [11]. It uses Sentinel 2 data and covers 13 different spectral bands and 10 land cover classes with a total of 27,000 labeled images.

In ExtremeEarth, we will develop tools to generate EO training datasets by enlarging existing datasets currently in development by the German Aerospace Center [4, 18] and by leveraging existing cartographic/thematic products which are now available at continental or planetary scale (e.g., OpenStreetMap). Two training datasets consisting of millions of samples will be developed aimed at the two deep learning architectures of Challenge C1. The datasets will be published as open source to be used by the whole Remote Sensing community.

Challenge C3. To develop techniques and tools for linked geospatial data querying, federation and analytics that scale to big Copernicus data, information and knowledge. The paradigm of linked geospatial data and relevant technologies has been pioneered by previous projects TELEIOS, LEO, Melodies and Copernicus App Lab mentioned above. The technologies developed include state-of-the-art systems for transforming geospatial data into RDF (GeoTriples [16]), interlinking with other geospatial data sources (geospatial/temporal extensions of Silk [21]), visualizing (Sextant [5]), querying, federating (Semagrow [3]) and performing data analytics (Strabon [15] and Ontop-spatial [1]). These systems are open-source and they currently represent the international state-of-the-art in the area of linked geospatial and EO data [6, 14]. In ExtremeEarth, the systems GeoTriples and Strabon will be re-engineered so that they scale to big linked geospatial data and extreme geospatial analytics. In addition, the JedAI linking framework [19] will be extended to enable the scalable discovery of geospatial relations in big geospatial RDF data sources. Finally, the engine Semagrow will be extended so that it can manage efficiently federations of big geospatial data sources and answer extreme geospatial analytical queries. To achieve the required extreme scalability, we will develop these three systems on top of the highly scalable HOPS data platform [12].

Challenge C4. To extend the capabilities for EO data discovery and access with semantic catalogue services that scale to the big data, information and knowledge of Copernicus. Currently, Copernicus data catalogues (e.g., the Copernicus Open Access Hub or the catalogues of the various TEPs) allow a user to access data by drawing an area of interest on the map and specifying search parameters such as sensing date, mission, satellite platform, product type etc. The new semantics-based catalogue we will develop in ExtremeEarth will expose the knowledge hidden in Sentinel satellite images and related data sets, and will allow a user to ask sophisticated queries such as "How many icebergs were embedded in the Norske ÃŸer Ice Barrier at its maximum extent in 2017?" which currently cannot be answered by the catalogue of the Norwegian Meteorological Institute, although all this knowledge is available in the Sentinel archive and related European data sets. We will demonstrate how to develop semantics-based catalogues and how to implement them efficiently for the extreme scale of the Copernicus context using the advances of ExtremeEarth in the area of big linked geospatial data discussed in Challenge C3 above. Two semantic catalogues (one for each TEP) will be developed operating in the selected DIAS platform and scaling to trillions of metadata records.

Challenge C5. To integrate the big data and extreme earth analytics technologies of Challenges C1-C4 in the HOPS data platform and deploy them in the selected DIAS and the two TEPs. The ExtremeEarth technologies presented above will be implemented and evaluated in the elastic cloud environment of the startup LogicalClocks participating in the consortium. This cloud environment is managed by LogicalClocks and currently provides the HOPS data platform [9, 12, 13, 17] together with significant storage, compute and GPU resources that will be made available to the project. Copernicus data will also be made available in the same environment and will be used to develop the ExtremeEarth technologies. The HOPS data platform provides services to move the processing to where the data is and it is based on a cloud computing platform-as-a-service approach. HOPS supports state-of-the-art parallel processing on big data with Apache Spark and deep learning with TensorFlow/Keras , as well as distributed deep learning using TensorFlow's distribution strategies, including collective allreduce and parameter server . HOPS also provides its own libraries for parallel deep learning experiments (hyperparameter search and model-architecture search). Once the ExtremeEarth technologies are integrated in HOPS, they will be deployed in the two TEPs and the selected DIAS.

The technologies discussed above will be demonstrated in two application areas: Food Security and Polar addressed by the relevant ESA TEPs. The challenges to be addressed in these two applications are the following.

Challenge A1. To develop high resolution water availability maps for agricultural areas allowing a new level of detail for wide-scale irrigation support. The maps will be available as linked data together with other geospatial layers (e.g., OpenStreetMap, field boundaries, crop types etc.) and made available to farmers. ExtremeEarth will tackle the combination of hydrological and agricultural monitoring, both in the sense that the thematic areas will be brought together, but also in the sense that the federation of the data sources and the cloud platforms used for the processing will be integrated. Both TEPs are already up and running, with the Food Security TEP, as the youngest of the TEPs, still in its second development phase, and the Polar TEP being in pre-operational mode. On both platforms, the first pre-processing chains for the information needed in ExtremeEarth are already running. These will be the baseline for bringing the two applications together in one combined application for irrigation. In practice this means that processing has to be widened to include whole watersheds (or catchment areas), to include all necessary Copernicus satellite input data from radar and optical imagery and to span the whole year instead of just the winter season or vegetation period. Additionally, scalable deep learning techniques discussed in Challenge C1 will be used to derive field boundaries and crop types, making it possible for the processing chains to include this information as linked data on a large scale (formerly, this information was only available at farm level). This will allow crop type specific deduction of crop variables, and thus a higher degree of accuracy for each field. The information generated with the ExtremeEarth approach will then be fed into the PROMET model [10] to provide high resolution (10m) water availability maps for the agricultural area in the whole watershed, allowing a new level of detail for wide-scale irrigation support. This application, which combines different TEPs, different Earth observation types and remotely sensed information and land surface modelling can be seen as a blueprint for further such applications, where the focus will lie on a federation of specialized knowledge and working environments (data and workflows). This type of federation of TEPs with methods, tools and data specialised for their topic rather than one broad platform for everything is seen by us as the way into the future.

Challenge A2. To produce high resolution ice maps from massive volumes of heterogeneous Copernicus data. The maps will be made *available as linked data and will be combined with other information such as sea surface temperature and wind information for informing maritime users.* The anticipated economic development of the Arctic, partially driven by reductions in sea ice cover, will see an increase in maritime shipping activity. High quality, timely and reliable information about sea ice and iceberg conditions is vital to ensure that vessels navigate efficiently and safely with minimal risk to the environment. This information is required by vessels in many sectors, including cargo transport, fisheries, tourism, research vessels, resource exploration and extraction, destination shipping and national coast guard vessels.

The functionality provided by both the DIAS infrastructure and the ESA Polar TEP as part of the ExtremeEarth infrastructure are well suited to answering the challenges of this application. Access to the required data interfaces is already established for some data sources in Polar TEP and further work to reinforce this will happen with integration of the DIAS infrastructure. Effort will be required to ensure data access is optimised for these purposes, for example ensuring near-real-time access to all required datasets. Building on the Polar TEP and DIAS as part of the ExtremeEarth infrastructure will also provide access to compute resources for processing. Since this is potentially going to be a significant processing load, but for limited periods of time as data is acquired and becomes available, then processing resources will need to be on demand and scalable to ensure efficiency. This will be achived by building on top of the HOPS data platform. Integration of established delivery systems into the ExtremeEarth infrastructure will support delivery of information products to polar users, such as tourist ships and fishing vessels operating in ice infested waters. This will include systems for information delivery and visualisation such as the Polar Code Decision Support System (PCDSS) which is currently being developed by company Polar View. PCDSS is designed to be used over restricted communication links, to bridge between the service production and users onboard ships in the Polar Regions. The scalable deep learning algorithms for sea ice classification, discussed in Challenge C1 above, will be integrated in the HOPS data platform to produce high resolution ice maps from massive volumes of heterogeneous Copernicus data. The aim is to deliver sea ice concentration and type maps, displaying stage of development (in accordance with the World Meteorological Organization - WMO Sea Ice Nomenclature), including fraction of leads and ridges, over the Polar Regions, at a resolution of 1 km or better.

3 THE EXTREMEEARTH CONSORTIUM

ExtremeEarth brings together a consortium of two companies leading the activities of the Food Security and Polar TEPs (VISTA and Polar View), one organization specializing in polar science for the Arctic and the Antarctic (British Antarctic Survey), one company specializing in big data, analytics and deep learning technologies (LogicalClocks), the German Aerospace Center with its TerraSAR-X satellite and expertise in SAR and multispectral EO (DLR), five top European academic institutions specializing in big data, linked data, Artificial Intelligence, deep learning and extreme earth analytics (National and Kapodistrian University of Athens, University of Trento, University of Tromsø, KTH and DLR), one research institute specializing in big data and Artificial Intelligence (National Center for Scientific Research - Demokritos), and one research institute specializing in big data, high performance computing applications, and provision of Metocean information, including sea ice (Norwegian Meteological Institute). The consortium is led by the National and Kapodistrian University of Athens.

ExtremeEarth starts in January 1, 2019 and will have a duration of 3 years. The project consortium is fully aware of the huge challenges that lay in front of us, and it is looking forward to meet them!

4 CONCLUSIONS

We have presented the vision of Horizon 2020 European project ExtremeEarth addressing the challenges of how to extract information and knowledge from the PBs of satellite data of the Copernicus programme using deep learning techniques, how to manage this information and knowledge efficiently using the HOPS data platform, how to develop two applications with economic and environmental importance (Food Security and Polar), and how to deploy these applications on the two relevant ESA TEPs and DIAS.

ACKNOWLEDGMENTS

This project has received funding from the European Union′s Horizon 2020 research and innovation programme under grant agreement No 825258.

REFERENCES

[1] K. Bereta and M. Koubarakis. 2016. Ontop of Geospatial Databases. In *ISWC.*

[2] K. Bereta, M. Koubarakis, S. Manegold, G. Stamoulis, and B. Demir. 2018. From Big Data to Big Information and Big Knowledge: The Case of Earth Observation Data. In *CIKM.*

[3] A. Charalambidis, A. Troumpoukis, and S. Konstantopoulos. 2015. SemaGrow: optimizing federated SPARQL queries. In *SEMANTICS.*

[4] C. Dumitru, G. Schwarz, and M. Datcu. 2018. SAR Image Land Cover Datasets for Classification Benchmarking of Temporal Changes. *J-STARS* 11 (2018).

[5] C. Nikolaou et al. 2015. Sextant: Visualizing time-evolving linked geospatial data. *J. Web Sem.* 35 (2015), 35–52.

[6] M. Koubarakis et al. 2016. Managing Big, Linked, and Open Earth-Observation Data: Using the TELEIOS/LEO software stack. *IEEE Geoscience and Remote Sensing Magazine* 4, 3 (2016).

[7] O. Russakovsky et al. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[8] P. Goyal et al. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR* (2017).

[9] S. Niazi et al. 2017. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *FAST.*

[10] T. Hank, H. Bach, and W. Mauser. 2015. Using a Remote Sensing-Supported Hydro-Agroecological Model for Field-Scale Simulation of Heterogeneous Crop Growth and Yield: Application for Wheat in Central Europe. *Remote Sensing* 7, 4 (2015), 3934–3965.

[11] P. Helber, B. Bischke, A. Dengel, and D. Borth. 2018. Introducing Eurosat: A Novel Dataset and Deep Learning Benchmark for Land Use and Land Cover Classification. In *IGARSS.*

[12] M. Ismail, E. Gebremeskel, T. Kakantousis, G. Berthou, and J. Dowling. 2017. Hopsworks: Improving User Experience and Development on Hadoop with Scalable, Strongly Consistent Metadata. In *ICDCS.*

[13] M. Ismail, S. Niazi, M. Ronström, S. Haridi, and J. Dowling. 2017. Scaling HDFS to more than 1 million operations per second with HopsFS. In *CCGRID.*

[14] M. Koubarakis, K. Bereta, G. Papadakis, D. Savva, and G. Stamoulis. 2017. Big, Linked Geospatial Data and Its Applications in Earth Observation. *IEEE Internet Computing* 21, 4 (2017).

[15] K. Kyzirakos, M. Karpathiotakis, and M. Koubarakis. 2012. Strabon: A Semantic Geospatial DBMS. In *ISWC.*

[16] K. Kyzirakos, D. Savva, I. Vlachopoulos, A. Vasileiou, N. Karalis, M. Koubarakis, and S. Manegold. 2018. GeoTriples: Transforming geospatial data into RDF graphs using R2RML and RML mappings. *J. Web Sem.* (2018).

[17] S. Niazi, M. Ronström, S. Haridi, and J. Dowling. 2018. Size Matters: Improving the Performance of Small Files in Hadoop. In *Middleware.*

[18] A. Oca, R. Bahmanyar, N. Nistor, and M. Datcu. 2017. Earth observation image semantic bias: A collaborative user annotation approach. *J-STARS* 10 (2017).

[19] G. Papadakis, K. Bereta, T. Palpanas, and M. Koubarakis. 2017. Multi-core Meta-blocking for Big Linked Data. In *SEMANTICS.*

[20] C. Persello and L. Bruzzone. 2014. Active and Semisupervised Learning for the Classification of Remote Sensing Images. *IEEE Trans. Geoscience and Remote Sensing* 52, 11 (2014), 6937–6956.

[21] P. Smeros and M. Koubarakis. 2016. Discovering Spatial and Temporal Links among RDF Data. In *LDOW.*

Neuromorphic Hardware As Database Co-Processors: Potential and Limitations

Visionary

Thomas Heinis[¶], Michael Schmuker[†]

[¶]*Imperial College, London, United Kingdom* [†]*University of Hertfordshire, Hatfield, United Kingdom*

ABSTRACT

Today's databases excel at processing data using simple, mostly arithmetic operators. They are, however, not efficient at processing that includes pattern matching, speech recognition or similar tasks humans can execute quickly and efficiently. One of the few ways to integrate such powerful operators into data processing is to simulate neural networks or to resort to the crowd.

Neuromorphic hardware will become common-place in complementing traditional computing infrastructure and will, for the first time, enable the time-efficient emulation of neural networks. Given the impact hardware accelerators like FPGAs and GPUs had on querying, the question is if neuromorphic devices can be used to a similar effect by enabling richer database operators.

In this paper we thus discuss how neuromorphic devices can be used as database co-processors. The goal of this paper is to understand their potential as well as limitations, what future developments will make them suitable to accelerate databases and what the research challenges are. We also evaluate a prototype implementation of a query operator on neuromorphic hardware.

1 INTRODUCTION

While today computers have a clear advantage over the brain in terms of raw computing power, the brain is superior in error resilience and speed for approximate tasks like understanding speech or recognizing objects. It is for these reasons but also due to the limitations of current CPU designs (pin bottleneck limiting data transfer, limited heat dissipation, power supply) that scientists have developed neuromorphic devices which for the first time enable the energy and time-efficient simulation of spiking neural networks (SNNs).

The vision is that neuromorphic devices will be as prevalent as FPGAs or GPUs are today: in a first instance neuromorphic chips can be plugged into existing systems and computations are offloaded (e.g., image recognition). Later, neuromorphic chips may share sockets and caches to accelerate exchange of data. With Intel, IBM and others developing neuromorphic hardware, the question is not if, but rather when, it will become commonplace.

Research has developed methods by which hardware accelerators are used to accelerate queries. GPUs are used to offload parallel computation (e.g., joins [14]) while FPGAs are used to compute histograms when data is read from storage [13, 20].

The question we thus ask is how neuromorphic hardware will support query execution in databases. In this paper we first discuss what neuromorphic hardware is and how it can be used to simulate spiking neural networks. We then discuss how neuromorphic hardware has the potential to act as a co-processor for databases to offload computation which is best executed using a SNN. We finally also discuss a prototype application.

2 SIMULATING NEURAL NETWORKS

The simulation of neural networks and neuronal activity is not a new development. Perceptron-based multilayer networks have been used for decades in applications like pattern recognition and memory. With neuromorphic hardware, however, it is for the first time possible to efficiently emulate large SNNs.

In neural networks, neurons communicate via spikes, that is, discrete events that occur at defined times. Depending on the weight of the synapse, these events either increase or decrease the probability that the receiving neuron will produce a spike.

Over time, neuron models of increasing bio-realism and thus complexity have been proposed. Early models did not use voltage spikes, but used neurons as threshold gates that simply add input and fire once the threshold is exceeded [18]. These neuron models only take binary input, produce binary output and can thus be used to compute any boolean function.

Next generation neuron models based on the perceptron [24] use an activation function (typically sigmoid or a linear saturation function) on the sum of weighted inputs to compute a continuous, differentiable output. Concatenated ensembles of perceptron units are able to approximate arbitrary functions and can thus be used as universal classifiers in pattern recognition.

More powerful than this are networks with spiking neurons. They use a model of a neuron that receives several spikes, adds them up to the local potential P over time and fires a spike once P exceeds a given threshold. Information can thus not only be encoded in the number of spikes, the firing rate, but also in the timing of spikes [10]. From an information theory point of view we can encode more information and reduce the number of neurons to perform the same computation with this approach.

Additionally, the time-dependence of spiking neural models enables using additional learning algorithms that operate in the time domain. In neural networks that use non-spiking models, learning is achieved through backpropagation of errors [25], where synaptic weights are iteratively adjusted until the applied stimulus leads to the desired output. In spiking networks, this learning rule is extended with spike timing or spike-timing-dependent plasticity (STDP). In STDP, the weight of a synapse changes as a function of the time difference between spikes produced by the pre- and the postsynaptic neuron (neuron before and after the synapse). In STDP, a synapse is strengthened when the pre-synaptic neuron fired a spike shortly before the post-synaptic neuron fired. Vice versa, the synapse is weakened if the post-synaptic spike precedes the pre-synaptic one [2]. STDP thus extends firing-rate based learning by including spike timing.

3 NEUROMORPHIC DEVICES

The brain is a massively parallel system of highly interconnected but computationally simple neurons. Neurons act entirely event driven and thus operate asynchronously, integrating incoming spikes and sending spikes to other neurons.

Integrating spikes is very efficiently done on von Neumann based hardware (traditional CPUs or GPUs) while other aspects cannot be executed efficiently. First, to simulate, a very high number of very small messages (i.e., spikes) must be sent between neurons. While spikes are encoded with a few bits, most

10.5441/002/edbt.2019.89

communication protocols require an order of magnitude more bits for header and routing information alone. Moreover, today's communication protocols lack efficient means for multi-cast communication which is crucial to send the same spike to a large set of neurons. Second, while neurons are only active when receiving and processing spikes, traditional CPUs or GPUs are always on, rendering the simulation of SNNs energy inefficient.

3.1 Brain-like Hardware

With these challenges, traditional CPUs or GPUs cannot efficiently simulate SNNs. As a consequence, *neuromorphic hardware* is being developed to better support simulation of SNNs. While the design depends on the manufacturer, all proposals are driven by similar ideas.

Computation: Key to neuromorphic hardware is massive parallelism, i.e., a large number of computational units/cores. Each of the cores has the relatively simple task of simulating several neurons and it can thus be comparatively wimpy (few FLOPS).

Communication: Unlike traditional communication between cores or computers, the payload sent between neurons is very small as only the timing of the spike matters. This information can be encoded in a few bits (40-72). Key to neuromorphic hardware thus is an efficient communication optimized for small payloads.

Given the massive number of connections between neurons, multicast communication must also be efficiently supported.

Energy Efficiency: Since the computational requirements to model neurons are modest, the computing infrastructure can be of low complexity. Also, since neurons operate fully event-driven, there is no need for global synchronization. These properties allow for making neuromorphic devices very energy efficient.

3.2 State of the Art Devices

The prototypes developed share key features like a massively parallel infrastructure with a fast interconnect for small messages.

For example, the SpiNNaker architecture is based on a large number of ARM processors [8]. Memory with little capacity is local to each processor (which itself has several cores) while there is also slower, global memory used for communication between processors. More important is a very efficient communication infrastructure for small packets (~24 Bytes for a spike).

IBM's TrueNorth project is a platform with 4096 cores each simulating 256 neurons [19]. Similar to SpiNNaker, memory, computation and networking is handled locally by each core, therefore moving past the von Neumann architecture. With an event driven model where cores are only powered when needed, TrueNorth reduces energy consumption similarly to SpiNNaker.

Intel's design reduces energy consumption by using spin devices to simulate the neurons (thereby restricting the neuron models that can be used) as well as memristor as local memory (to store synapse weight) [27].

Spikey [23] takes energy efficiency even further by using a mixed-signal approach. Neurons are implemented using analog elements. While this makes them more bio-realistic, it limits the flexibility to use arbitrary neuron models. New models cannot simply be implemented through programming, but require changes in the hardware. Nevertheless, Spikey has proven to be versatile enough to implement a wide range of networks [23].

4 APPLICATIONS IN DATA MANAGEMENT

Spiking neural networks are capable of modelling arbitrary complex processes thanks to their ability to represent different information dimensions, such as time, space, frequency and phase.

In this section we discuss applications where neuromorphic hardware can support databases. In the applications we discuss, extracting all features a priori and storing them in a database may at first seem an option but is unfeasible as the feature space will explode, thus requiring excessive storage space.

The simulation of SNNs is thus key. Any application based on a SNN can be simulated on a CPU or GPU but only inefficiently. The benefit of using neuromorphic hardware is that it can run bigger SNNs faster and crucially, substantially more energy efficient.

While there is a plethora of potential applications which can be simulated with SNNs, our discussion is primarily driven by application domains where SNNs are used successfully.

4.1 Multimedia Databases

Neural networks can be used for data classification [26] but they prove particularly useful for recognising media. Several SNNs have been trained based on supervised learning methods [3, 21, 22, 26] and have been tested on imaging benchmarks. SNNs frequently outperformed non-spiking classification methods [3].

Content-based Image Retrieval Research has produced multiple approaches for content-based image retrieval in multimedia databases [4]. Most approaches are based on two phases: feature extraction (i.e., color, shape and others) and the indexing of the features [17]. Feature extraction is application specific [4] while indexing is general and based on high-dimensional indexes for similarity searches (R-Trees [11] and variants).

State-of-the-art approaches generally have two shortcomings. First, the semantic gap [6] means that it is very challenging to determine the semantics of an image from its low-level features, i.e., there is little connection between pixel statistics and the semantics of an image. Second, due to the high number of dimensions of extracted features and the curse of dimensionality, content-based retrieval of images is not efficient and scales poorly [5].

Spiking neural networks, however, are very useful for image and pattern recognition [1]: using a network with spike-timing dependent plasticity, one of the images is presented to a two layer network in the learning phase. The presentation triggers a single spike in each neuron in the first network layer and the incoming activity propagates to the next layer. The first neuron to fire in the second layer inhibits its neighbors and triggers learning. As a result each neuron fires if again presented with the same image. The learning step is substantially faster compared to other methods (particularly backpropagation) as is the recognition step.

Using spiking neural networks for image retrieval in multimedia databases (or databases in general) thus has two major advantages. First, using SNNs, no feature extraction is necessary and the method consequently can work directly on the raw data. By doing so there is less risk of basing retrieval on a highly specific set of features. Whether the use of SNNs bridges the semantic gap, however, has not been determined yet [1] but it decouples image recognition from features. Second, by using SNNs, no high-dimensional indexes are needed and image retrieval is more efficient and scales better. By simulating SNNs efficiently, neuromorphic hardware enables their use for image retrieval.

Audio & Video Retrieval The problem of audio and video retrieval is very similar to image retrieval with the difference that both also have a time dimension, e.g., videos have consecutive different frames. Simple approaches for video retrieval are based on the same ideas as image retrieval with a additional features drawn from changes between frames. By doing so they have the same drawbacks as current image retrieval methods.

Spiking neural networks support time very well. Any stimuli of a SNN is time-based, e.g., even images need to be encoded as a succession of neuron stimuli over time. SNNs and neuromorphic hardware thus lend themselves well for audio or video [28].

4.2 Spatial Indexing

A vast number of spatial indexes has been developed [9]. Many spatial indexes (particularly based on data-oriented partitioning like the R-Tree and variants), however, suffer from limited performance due to overlap and dead space in the index [11].

SNNs have also been used for spatial navigation: through learning mental maps of the environment it enables planning of paths [12]. The particular neuron in the network is activated as a simulated animal explores different locations in the environment and connections between neurons activated in a close temporal proximity are strengthened, i.e, cells representing neighboring locations develop strong synaptic interactions. This mechanism can be used for answering nearest neighbor queries.

A SNN run on neuromorphic hardware thus has the potential to execute nearest neighbor queries and planning paths.

5 CHALLENGES

We propose to use neuromorphic hardware as a co-processor for databases. We use the hardware to efficiently simulate a SNN for a complex query operator, e.g., pattern recognition for image retrieval. The associated research challenges can be identified for neuroscience and data management research alike.

On the level of hardware, while existing neuromorphic hardware already scales substantially better than traditional von Neumann architectures, it has to be investigated how current approaches can scale to simulate SNNs beyond millions of neurons, so that more sophisticated operators can be implemented.

Regarding neuroscience, the challenge lies in finding more SNNs that solve generic computing problems and are amenable to simulation on neuromorphic hardware. Several such SNNs have been developed but more applications from human cognition need to be considered as they are useful as database operators.

On the level of databases the challenges are as follows:

5.1 Data Preparation/Encoding

The data in the database and the query need to be encoded for the neuromorphic hardware. More precisely, as is common for SNNs, the data needs to be encoded as temporal spikes that can be fed into the system. Consider, for example, pattern recognition in images. All images as well as the query need to be encoded as temporal spiking patterns for it to be matched on a SNN.

Encodings for video and audio can be computed very efficiently: because they already have a temporal structure, data like movies or sound are converted to sequences of spikes using little computational effort. Similarly, for imaging data, sensors are available that efficiently produce time series of spikes [16].

As numerous examples of functional SNNs show, it is often possible to convert arbitrary data into spikes. Whether the effort to produce such an encoding is outweighed by the gain in computational power that a SNN will provide over more conventional approaches of data processing, however, is questionable.

Furthermore, while the same encoding can be used for matching different queries, it is very likely that different query operators will benefit from encoding that is tuned to that particular operator. A more difficult challenge thus is to decide what encodings should be stored and the organization in storage. It may suffice to only store the difference between encodings instead of storing one encoding per operator.

The most difficult challenge, however, is how to execute queries on multiple attributes. Going back to the imaging example: a query may restrict the area in which to find a particular pattern. Restricting the area on an already encoded imaging is difficult as all positional information (about the area) is lost in the encoding.

5.2 Query Planning

Current prototypes of neuromorphic hardware have a very limited interface to move data making query execution challenging.

Modelling Execution Cost A crucial question is when SNNs are run faster on neuromorphic hardware compared to the CPU. The query plan has to consider a cost model with the time to transfer data to and from the device and the execution time of the simulation. Execution time, however, is not the only consideration as running an operator on the device is more energy efficient than on the CPU. A second consideration thus has to be the energy needed to move the data and to run the SNN.

Query Optimization Given the high cost of moving data, moving all data to the device should be avoided. Considering the selectivity of predicates other than SNN (e.g., metadata information) early in the query plan is therefore imperative. Further, a crucial research challenge is to investigate if features in the raw data can be identified, extracted and indexed such that they can be used early in the query execution to curb data movement. Selecting the features is very challenging without reintroducing the semantic gap known from image retrieval.

5.3 Query Execution

Setting Up New Operators To define an operator, a network needs to be trained by adjusting synaptic weights. Key to the training phase is to encode the stimulus appropriately and injecting all learning stimulus consecutively to train the network. The learning phase can also be performed in a simulated environment and the resulting weights can be transferred to the hardware.

Swapping Operators One network cannot serve all purposes so operators/networks need to be loaded and unloaded. The state of a SNN or operator is captured in the neurons, their placement, their interconnections and the synaptic weights.

To swap an operator, a new SNN needs to be set up by loading neurons and synapse weights which is a slow process. In many cases, however, it may suffice to only define rules and derive the precise connections/network on the fly.

Interpreting Results Depending on the SNN, the result is indicated differently. In some cases the result is binary, i.e., one neuron being active indicates the result. In other cases, for example in case of the spatial application, the proximity of two points is expressed by the proximity of two active neurons. Each operator must therefore come with an implementation of an interpreter able to understand the result.

The major research challenge is how ambiguous simulation results are interpreted. The result is rarely precise and this ambiguity has to be propagated to the user. One approach is the use of ideas from uncertain databases, e.g., attribute-level uncertainty.

6 CURRENT LIMITATIONS

Neuromorphic hardware still is a new proposition and most available prototypes focus on the efficient execution of SNNs as a proof of concept whereas the programmability, data transfer and others are currently only second order considerations.

Moving Data: Current hardware, in particular the SpiNNaker system, has a very limited interface to move data to and from the device (Ethernet interface with 100MBps). The issue of bandwidth will, however, be addressed if neuromorphic hardware demonstrates its usefulness.

Simulation Size: The current challenge for hardware developers is increasing the capacity of neuromorphic systems. The largest systems today can simulate at most millions of neurons. Nevertheless, it is believed that scaling the number of neurons up to the human brain (i.e., 10^{10} neurons) will give neuromorphic systems the ability to infer relationships in data of a complexity that is inaccessible to conventional computers [7].

7 PROTOTYPE IMPLEMENTATION

To show the basic feasibility of using neuromorphic hardware as a database co-processor, we implement a proof of concept based on a simple application. Clearly this is only an example with several obvious optimizations which we address in future work.

7.1 Sample Application

As our sample application we use the recognition of digits in images [26]. Images containing hand written digits are stored as bitmaps in the database and a users query to find images containing a particular digit. To answer a query, a SNN run on neuromorphic hardware infers the digit in the image. The SNN is trained offline by showing sample digits (0 - 9) from MNIST [15].

In the current implementation we use the same digits for learning and querying. We do so to avoid the challenge of interpreting the result — an open research challenge as we discussed. Hence, when training we store the response (active neurons) to the stimulus for a particular digit. When asking what images contain a particular digit, we present the stored images to the SNN and compare the stimulus response (active neurons) with stored responses. Due to the nondeterminism of SNNs, the stored and the current result may not be identical. We thus compute the share of active neurons the current and the stored response agree on and consider the result a match if it exceeds a predefined threshold.

Compared to preprocessing all data and storing the result, we remain flexible as the SNN can be updated and ran on the hardware to query the data (with potentially more accurate results).

7.2 Setup

For the current implementation we use Postgres. Given the images from MNIST are stored as bitmaps, we store them as two dimensional arrays without need for transformation.

We use a user defined C function in Postgres: given a number *n* and a images stored in a table *t*, the UDF presents all images in *t* to the classifier and returns the ones likely to contain *n*.

For the experiments we use a 4 chip SpiNNaker board [8], the smallest board but adequate for a proof of concept. The board has 72 identical Arm968 processors (18 per SpiNNaker chip) operating at 200MHz. The SpiNNaker board currently only provides an Ethernet interface which connect to the host running Postgres.

7.3 Experimental Analysis

Training the SNN takes on average 68.2 seconds. The vast majority (95.5%) of time is spent transferring data: images to the device (8.4%), reading the stimulus responses (29.2%) and reading the trained SNN (57.9%). Learning only takes 3.1 seconds as each of the hundred digits is exposed to the SNN for 20ms.

Querying for one number — classifying all images and finding the ones matching the user input — takes on average 52.5 seconds. Most time is spent on moving data, i.e., moving the trained SNN (84.4%) and sending all images (10.5%) to the device. Classification on the device takes 0.2s while the time spent in the UDF is 2.48s.

Clearly, querying (as well as learning) suffers from loading (and storing) the SNN — the main bottleneck. Even without, however, moving the images is a very costly operation. The time for learning and classifying, on the other hand, is insignificant.

7.4 Open Challenges

As simple as the example used is, it shows some of the challenges discussed. A major challenge is the transfer of data. Currently this can only be done through Ethernet and is slow but future versions will have SATA and TrueNorth, for example, uses PCI.

Related to the limited bandwidth is the challenge of reducing the data moved: only images which contain the digit with high probability should be moved. Means to index the stimuli need to be devised so that a preselection of images can be performed.

Finally, scoring the result (and the inherent uncertainty) is a challenge which we worked around. Clearly, better approaches need to be devised for more sophisticated applications.

8 CONCLUSIONS

While neuromorphic hardware systems are still maturing, the high-level design is well-defined and the benefits are clear. It is thus the right time to start assessing the viability of neuromorphic applications. In this paper we have sketched the research challenges for query execution. Some challenges are implied by the prototypic nature of the hardware, but most are due to more fundamental reasons (e.g., query planning). By addressing these challenges we believe neuromorphic hardware will enable the efficient execution of more sophisticated query operators.

REFERENCES

[1] 2001. Spike-based Strategies for Rapid Processing. *Neural Networks* 14, 67 (2001), 715 – 725.

[2] G Bi and M Poo. 2001. Synaptic Modification by Correlated Activity: Hebb's Postulate Revisited. *Annual Review of Neuroscience* 24 (Jan 2001), 139–166. https://doi.org/10.1146/annurev.neuro.24.1.139

[3] S. M. Bohte, H. La Poutre, and J. N. Kok. . Unsupervised Clustering with Spiking Neurons by Sparse Temporal Coding and Multilayer RBF Networks. *Transactions on Neural Networks* 13, 2 ().

[4] Ritendra Datta, Dhiraj Joshi, Jia Li, and James Z. Wang. 2008. Image Retrieval: Ideas, Influences, and Trends of the New Age. *Comput. Surveys* 40, 2 (May 2008).

[5] Adrien Depeursinge, Benedikt Fischer, Henning Müller, and Thomas M Deserno. 2011. Prototypes for Content-Based Image Retrieval in Clinical Practice. *The Open Medical Informatics Journal* 5 (Jul 2011).

[6] Thomas M. Deserno, Sameer Antani, and Rodney Long. 2008. Ontology of Gaps in Content-Based Image Retrieval. *Journal of Digital Imaging* 22, 2 (2008).

[7] Steven K. Esser, Alexander Andreopoulos, Rathinakumar Appuswamy, Pallab Datta, Davis Barch, Arnon Amir, John Arthur, Andrew Cassidy, Myron Flickner, and Paul Merolla. 2013. Cognitive Computing Systems: Algorithms and Applications for Networks of Neurosynaptic Cores. December (2013).

[8] S.B. Furber, F. Galluppi, and S. Temple. 2014. The SpiNNaker Project. *Proc. IEEE* 102, 5 (2014).

[9] Volker Gaede and Oliver Guenther. 1998. Multidimensional Access Methods. *Comput. Surveys* 30, 2 (1998).

[10] Wulfram Gerstner, Andreas K. Kreiter, Henry Markram, and Andreas V. M. Herz. 1997. Neural Codes: Firing Rates and Beyond. *Proceedings of the National Academy of Sciences* 94, 24 (1997).

[11] Antonin Guttman. . R-trees: a Dynamic Index Structure for Spatial Searching. *SIGMOD* ().

[12] John J. Hopfield. 2010. Neurodynamics of Mental Exploration. *Proceedings of the National Academy of Sciences* 107, 4 (2010), 1648–1653.

[13] Zsolt Istvan, Louis Woods, and Gustavo Alonso. . Histograms As a Side Effect of Data Movement for Big Data. In *SIGMOD '14*.

[14] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. . GPU Join Processing Revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN '12)*.

[15] Yann LeCun, Corinna Cortes, and Christopher JC Burges. 1998. The MNIST Database of Handwritten Digits.

[16] P. Lichtsteiner, C. Posch, and T. Delbruck. . A 128 X 128 120db 30mw Asynchronous Vision Sensor that Responds to Relative Intensity Change. In *IEEE International Solid-State Circuits Conference 2006*.

[17] M. De Marsicoi and L. Cinque. 1997. Indexing Pictorial Documents by their Content: A Survey of Current Techniques. *Image and Vision Computing* 15, 2 (1997).

[18] WarrenS. McCulloch and Walter Pitts. 1943. A Logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biophysics* 5, 4 (1943), 115–133. https://doi.org/10.1007/BF02478259

[19] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L. Jackson, and Nabil Imam. 2014. A Million Spiking-neuron Integrated Circuit with a Scalable Communication Network and Interface. *Science* 345, 6197 (2014).

[20] Rene Mueller and Jens Teubner. . FPGA: What's in it for a Database?. In *SIGMOD '09*.

[21] Emre Neftci, Srinjoy Das, Bruno Pedroni, Kenneth Kreutz-Delgado, and Gert Cauwenberghs. 2014. Event-driven Contrastive Divergence for Spiking Neuromorphic Systems. *Frontiers in Neuroscience* 7, January (2014).

[22] Bernhard Nessler, Michael Pfeiffer, Lars Buesing, and Wolfgang Maass. 2013. Bayesian Computation Emerges in Generic Cortical Microcircuits through Spike-Timing-Dependent Plasticity. *PLoS Computational Biology* 9, 4 (2013).

[23] Thomas Pfeil, Andreas Grübl, Sebastian Jeltsch, Eric Müller, Paul Müller, Mihai A. Petrovici, Michael Schmuker, Daniel Brüderle, Johannes Schemmel, and Karlheinz Meier. 2013. Six Networks on a Universal Neuromorphic Computing Substrate. *Frontiers in Neuroscience* 7 (2013), 11.

[24] Frank Rosenblatt. 1958. The perceptron: a Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review* 65, 6 (1958).

[25] Rumelhart, David E and Hinton, Geoffrey E. 1988. Learning Representations by Back-propagating Errors. *Cognitive Modeling* 5 (1988), 3.

[26] Michael Schmuker, Thomas Pfeil, and Martin Paul Nawrot. 2014. A Neuromorphic Network for Generic Multivariate Data Classification. *PNAS* 111, 6 (Feb 2014).

[27] Mrigank Sharad, Charles Augustine, Georgios Panagopoulos, and Kaushik Roy. 2012. Proposal For Neuromorphic Hardware Using Spin Devices. arXiv:arXiv:1206.3227

[28] Simei Gomes Wysoski, Lubica Benuskova, and Nikola Kasabov. 2008. Fast and Adaptive Network of Spiking Neurons for Multi-view Visual Pattern Recognition. *Neurocomputing* 71, 13 - 15 (2008).

Short Paper

Query Driven Data Labeling with Experts: Why Pay Twice?

Eyal Dushkin[1] Shay Gershtein[1] Tova Milo[1] Slava Novgorodov[2]

[1] *Tel Aviv University* [2] *eBay Research*

[1]{eyaldush, shayg1, milo}@post.tau.ac.il [2]snovgorodov@ebay.com

ABSTRACT

Data has become a major priority for customer facing businesses of all sizes. Companies put a lot of effort and money into storing, cleaning, organizing, enriching and processing data to better meet user needs. Usually in large scale systems such as big e-commerce sites these tasks involve machine learning methods, relying on training data annotated by domain experts. Since domain experts are an expensive resource in terms of monetary costs and latency, it is desired to design algorithms that minimize the interaction with them.

In this paper we address the problem of minimizing the number of annotation tasks with respect to a set of queries. We present a dedicated algorithm based on efficient labeling, that dictates the strategy for constructing a minimal set of classifiers sufficing to answer all queries. Our approach not only reduces monetary costs and latency, but also avoids data redundancy and saves storage space. We first consider a typical scenario of two expressions per query, and further discuss the challenges of extending our approach to multiple expressions. We examine two common models: *batch* and *stream* configurations, and devise *offline* and *online* algorithms, respectively. We analyze the number of annotations, and demonstrate the efficiency and effectiveness of our algorithm on a real-world dataset.

1 INTRODUCTION

Data has become a major priority for customer facing businesses of all sizes. Companies put a lot of effort and money into storing, cleaning, organizing, enriching and processing data to better meet user needs. For example, news articles published on news websites, are often annotated, e.g., tagged with "World Cup 2018" or "Elections in the United States", enabling readers to easily consume relevant content, hereby improving personalization and accessibility. E-commerce websites invest in generating a reliable catalog of products combining human and machine intelligence [4, 10, 17]. This allows potential customers to find the best matching product either by navigating through faceted categories, or by executing search queries. Since catalogs are often huge and cannot be maintained solely by human experts, automatic solutions based on machine learning (ML) are also employed. Combining both experts and ML is a widely-used approach also in fraud detection applications [9], text categorization [15] and other classification tasks [19, 20]. One of the most common usages of ML with *human in the loop* is harnessing domain experts to generate adequate labeled data as a baseline for supervised learning. Thus, generating sufficient annotations, while minimizing domain experts' effort, is highly desired. This trade-off between high quality and low cost is the holy grail of training data preparation. Much research has been devoted in the literature to minimizing the interaction with regular crowds [8, 12, 20]. However, these techniques are best suited for mundane classification tasks, whereas for questions that require particular expertise, as shown in [6], gathering multiple answers from various crowd workers does not always produce a desired accuracy level. In light of this, we present a novel query driven approach with *human experts* that avoids redundant labeling tasks and finds the minimal set of necessary tasks. We note that our approach may be combined in a regular crowdsourced schema with probabilistic settings. To illustrate our approach, consider the following example.

Example 1.1. A database *Products* contains a relation *Shirts* with attributes *product_id*, *product_title*, *product_description*, *product_image*, *product_price*, *color* and *material*. This relation contains a list of shirts sold on an e-commerce website. While the product title and description are provided by the seller, its color and material are usually missing and should be automatically extracted from the title, description or image using ML classifiers. Assume a customer performs a keyword-based search for orange cotton shirts, which translates into the following SQL query via NLP-based methods[1]:

```
SELECT * FROM Shirts
WHERE `color` = 'Orange'
AND `material` = 'Cotton';
```

Providing an answer to such queries, should rely on correctly enriched color and material attributes values for every product. Toward this end, one may train various classifiers with respect to suitable labeled data. Classifiers shall be incorporated to answer "Is this product's color is X and material is Y?". Given that any classifier needs N labeled examples for training, in order to answer the query above one can either train *two* classifiers: one for detecting orange shirts and one for detecting cotton shirts, and separately apply them for each product. This requires $2N$ labeled examples. Alternatively, one could train a more specific classifier that detects orange cotton shirts, using only N labeled examples. Clearly, for this specific case, it is better to choose the latter. However, given a general list of queries with different values of colors or materials, minimizing the number of classifiers may be difficult.

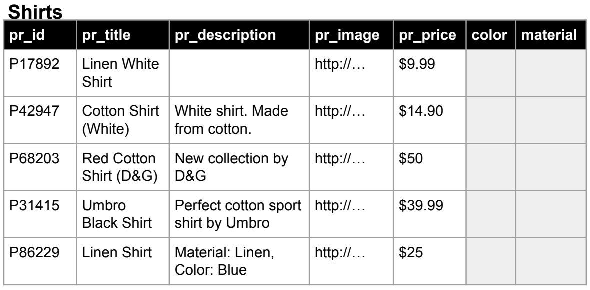
In this paper we propose an algorithm that minimizes the number of classifiers (labeling tasks) that are sufficient to answer all queries. We focus on the case of having at most two expressions per query, which is the most common case in e-commerce search [3]. The extension of our approach to multi-criteria queries is discussed in the future work section. We present a dedicated algorithm based on efficient labeling, that dictates the strategy for constructing a minimal set of classifiers sufficing to answer all queries. Our approach not only reduces monetary costs and latency, but also avoids data redundancy and saves storage space entailed in the enriched attribute values. The contributions of this paper can be summarized as follows:

- We formulate the problem of experts labeling effort minimization with respect to a database and a list of queries.

[1] This methods involve NER-based solutions and assumed to be given, hence it is out of the scope of this paper.

Series ISSN: 2367-2005

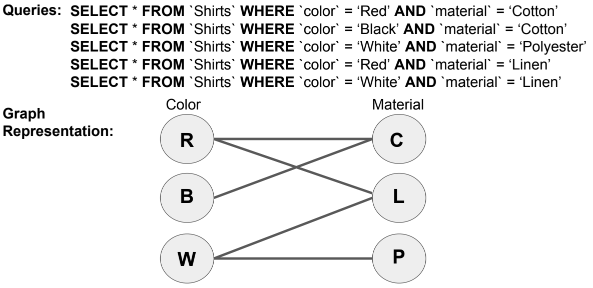
10.5441/002/edbt.2019.90



Shirts

pr_id	pr_title	pr_description	pr_image	pr_price	color	material
P17892	Linen White Shirt		http://…	$9.99		
P42947	Cotton Shirt (White)	White shirt. Made from cotton.	http://…	$14.90		
P68203	Red Cotton Shirt (D&G)	New collection by D&G	http://…	$50		
P31415	Umbro Black Shirt	Perfect cotton sport shirt by Umbro	http://…	$39.99		
P86229	Linen Shirt	Material: Linen, Color: Blue	http://…	$25		

Figure 1: 'Shirts' relation example.



Queries: **SELECT** * **FROM** `Shirts` **WHERE** `color` = 'Red' **AND** `material` = 'Cotton'
SELECT * **FROM** `Shirts` **WHERE** `color` = 'Black' **AND** `material` = 'Cotton'
SELECT * **FROM** `Shirts` **WHERE** `color` = 'White' **AND** `material` = 'Polyester'
SELECT * **FROM** `Shirts` **WHERE** `color` = 'Red' **AND** `material` = 'Linen'
SELECT * **FROM** `Shirts` **WHERE** `color` = 'White' **AND** `material` = 'Linen'

Figure 2: Queries and their graph representation.

- We propose an algorithm that solves the special (yet highly common) case of having at most two expressions per query and provide theoretical analysis of this algorithm.
- We extend the solution to a streaming scenario, where queries are processed piece-by-piece in a serial fashion, and provide approximation guarantees for our approach.
- We conduct an experimental evaluation with a real-world setting, demonstrating the efficiency and effectiveness of our approach, with respect to the number of annotation tasks and the additional storage required.

The paper is organized as follows. The next section defines the model and the problem statement and describes the technical details of the solution (Section 2). We then present our experimental evaluation (Section 3). Finally, we discuss related and future work (Sections 4-5).

2 OUR APPROACH

We start by explaining the data model composed of queries and attributes, followed by the description of the algorithm and its online fashion variant for the streaming scenario.

2.1 Preliminaries and Model

Data: Our model consists of a database $\mathcal{D}$ with relation $\mathcal{R}$, a set of attributes F occupied with values for all $t \in R$ and a set MF of attributes with missing values for some (or all) $t \in R$. Each of the attributes has its domain, the set of all possible values per attribute, i.e., $dom(A_i) = \{v_i^1, v_i^2, ..., \}$.
Binary Classifier: Missing attribute values can be discovered with full certainty by constructing a proper *binary* classifier based on labeled data generated by domain experts[2]. Formally, a binary classifier maps every tuple $t \in R$ with its predefined attributes values to $\{0, 1\}$, and can be used for filling holes in missing attribute values. For example, followed our running example, the predefined attributes F are all the $product\_*$ attributes (title, image, price, etc.) while two missing attributes are *color* and *material*. In order to reveal missing values for colors, one can learn binary classifiers for various colors that indicate whether a tuple t has the objective color, e.g., $C_{red}(t) = 1$ if t is red. Figure 1 depicts a small sample of such relation that contains products from "Shirts" category. Note that since data is provided by various sellers, the information is concealed within different patterns, usually semi-structured or free-text fields. In some cases the relevant missing values exist only in the title, in other in the description, or in a combination of both. Hence, extracting the desired attributes is a difficult task, which needs well-trained classifiers. We assume that the construction of every classifier is the same. We discuss how to relax this assumption in Section 5, which is a part of our ongoing work.

Queries: In addition to the information stored in $\mathcal{D}$, there is a set of queries Q that is used to extract the relevant tuples from $\mathcal{D}$. The queries assumed to be only over missing attribute values[3]. Formally, the query is of the following form:

```
SELECT * FROM R WHERE `A_1` = 'v_1_j1' AND
`A_2` = 'v_2_j2' AND ... AND `A_n` = 'v_n_jn';
```

We denote the queries that specify all the attributes from MA as *full queries*, while the queries that specify only subset of MA as *partial queries*.
Query Expressiveness: We assume that queries may specify up to two equality expressions over a set of missing attributes values MF. While this sounds limiting, having one or two tokens is the most common case both in e-commerce [3] and general purpose search engines [7]. Extending our solution to multi-criteria queries is a part of the open problems and ongoing effort discussed in Section 5.
Graph Model: We represent the set of queries in an undirected graph $G = (V, E)$, where the vertices V are the values that appear in the queries (unique per attribute) and the edges E correspond to the queries such that if a query clause consists of two equality expressions with values v_1^j and v_2^l of attributes A_1 and A_2, resp., we construct an edge between their corresponding nodes in the graph. To support partial queries with one equality expression, we use self edges on the corresponding node. Figure 2 illustrates the set of unique queries and their corresponding graph representation. The graph consists of 6 nodes, 3 unique value per each of the attributes and 5 edges (the number of queries). Note that the graph is not necessarily connected. For example, removing in the mentioned example the "white linen shirt" query, splits the graph into two separate connected components. In addition, the provided example has no self-loops since all queries have two expressions. Adding a query such as:

```
SELECT * FROM `Shirts` WHERE `color` = 'Black';
```

generates a self-loop in the graph ("B" node).

2.2 The Algorithm

We start by presenting the *offline* algorithm assuming the set of queries is given as a whole. We extend our solution to support streaming fashion in the next subsection. Considering the general case, the algorithm should determine the set of suitable binary classifiers: either a *query-oriented* classifier corresponding to specific query (denoted by "label the edges") or a *predicate-oriented* classifier corresponding to single attribute value (denoted by "label the nodes"). The goal is to minimize the number of classifiers that are sufficient to answer all queries. Note that in order to answer all queries, our algorithm must determine which edges and nodes to assign a classifier.

Assuming a relation $\mathcal{R}$ and a set of queries Q. The algorithm first constructs a graph representation of the queries, as described

[2] A common method of *multi-label classification* amounts to independently training one binary classifier for each label [16]. There are other solutions, e.g., reduction to the *multi-class classification* problem or adaption methods, which are out of the scope of this work.

[3] Clauses that involve F attributes can be filtered without any classification.

in previous section. For every connected component C in the graph, the algorithm determines whether labeling edges or nodes (in accordance with query-oriented and predicate-oriented classifiers, resp.) by calculating $|E_C|$ and $|V_C|$. If $|E_C| \geq |V_C|$, the algorithm labels nodes[4] and edges otherwise. Since we focus on connected components, the number of edges $|E_C|$ is at least $|V_C| - 1$. Therefore, the algorithm labels edges if and only if $|E_C| = |V_C| - 1$, i.e., if the connected component is a tree.

Lemma 2.1. *Given a relation $\mathcal{R}$ and a set of queries Q with a corresponding graph $G = (V, E)$, the algorithm minimizes the number of classifiers that are sufficient to answer all queries.*

Proof. Since connected components are independent, it suffices to prove correctness with respect to a single connected component C. Assume that there is a better solution with a set of classifiers T, $|T| < min(|E_C|, |V_C|)$. If C is a tree with $|E_C| = |V_C| - 1 < |V_C|$ then there is an edge (v_{q_i}, v_{q_j}) with no query-oriented classifier from T and at least one of v_{q_i} or v_{q_j} does not have a predicate-oriented classifier in T. Thus, the query corresponds to this edge is insoluble. On the other hand, let $|V_C| \leq |E_C|$, then some v_{q_i} does not have a predicate-oriented classifier in T, thus all $(v_{q_i}, u) \in E$ must have query-oriented classifiers, $d = |(v_{q_i}, u) \in E|$. Since a query-oriented classifier matches a single edge, the number of edges is reduced by d while the number of nodes is reduced by at most d, since otherwise v_{q_i} and its neighbours form a tree. Thus, applying these query-oriented classifiers induces a sub-graph $C' \subset C$ with at least $|V| - d$ nodes and $|E| - d$ edges. Since $min(|V| - d, |E| - d) > |T| - d$ it follows, by induction, that the set of classifiers is insufficient to answer all queries. □

Corollary 2.2. *Let q be a partial query, v_q its corresponding vertex and C_{v_q} the connected component containing v_q, then the set of classifiers derived by labeling all nodes in C_{v_q} is a minimal set that suffices to answer all queries projected to C_{v_q}.*

Proof. Since partial query with one attribute corresponds to a self-loop in a graph, a cycle is created. Therefore, the connected component is not a tree and its nodes should be labeled. □

2.3 Online Algorithm

We now assume that the queries arrive in a streaming manner, i.e., piece-by-piece in a serial fashion, and the algorithm makes a decision based on limited information.

The practical motivation for this setting is the common scenario where the already existing classifiers are not sufficient to provide accurate answers to users' query, thus a deficient result is retrieved based on a simple full-text query against a text index. to improve for future times where such a query will be issued, suitable labeling tasks are generated.

Here again we assume a relation $\mathcal{R}$ and a set of queries Q arriving in a streaming manner, one at a time, and our algorithm makes labeling decisions that may later turn out to be sub-optimal. Our algorithm initializes an empty graph and for every query q being processed, it updates (or creates) the connect component C_q with the corresponding edge and nodes. Similar to the offline version, it labels the new edge if C_q is a tree, and the nodes o/w.

Lemma 2.3. *Given a query q being processed from a stream of queries, v_q its corresponding vertex and C_{v_q} the connected component containing v_q. If the algorithm decides to label the node v_q, the optimal strategy henceforth w.r.t to C_{v_q} is labeling nodes.*

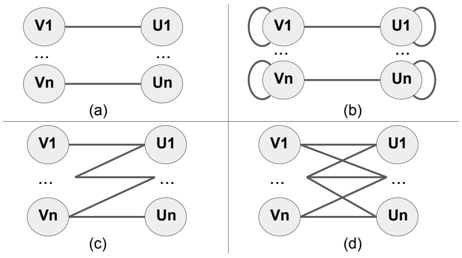


Figure 3: Worst case analysis of "label the edges" strategy.

Proof. Given a query q with vertex v_q within connected component C. Assume that $|E_C| \geq |V_C|$ for the first time immediately after updating the graph, and the algorithm decides to label v_q. Now, since every processed query q' with v'_q in C contributes at most one vertex and at least one edge to C, it follows that $|E_C| - |V_C|$ can only increase and $|E_C| \geq |V_C|$ holds. □

For every connected component C, we refer to the point when C abandoned its tree structure, as the *swapping point*, since our strategy swapped from label edges to label nodes.

Relative Bounds Analysis: First, as proven in Lemma 2.3, once we have swapped strategy from labeling edges to labeling nodes, the optimal strategy is labeling nodes henceforth. Thus, our error derives in between those states, if exist. Till the swapping point $|V_C| - 1$ edge classifiers were generated and in the worst case, each one of them turns out to be redundant. On the other hand, since every solution must contain at least $|V_C| - 1$ classifiers, our approximation ratio is at most 2.

We now illustrate two unfortunate cases. In Figure 3a there are $|E|$ connected components, each consists of two nodes and one edge. Figure 3b makes the original strategy sub-optimal, where each connected component has one redundant edge classifier. In this case, the ratio is $(n + 2n)/(2n) = 3/2$. Figures 3c and 3d present the worst case scenario where the graph is a tree till the swapping point. Therefore, the approximation ratio in the case is $(|V_C| - 1 + |V_C|)/|V_C| = 2 - 1/|V_C|$, demonstrating we can make the relative error as close as we like to 2.

3 EXPERIMENTAL EVALUATION

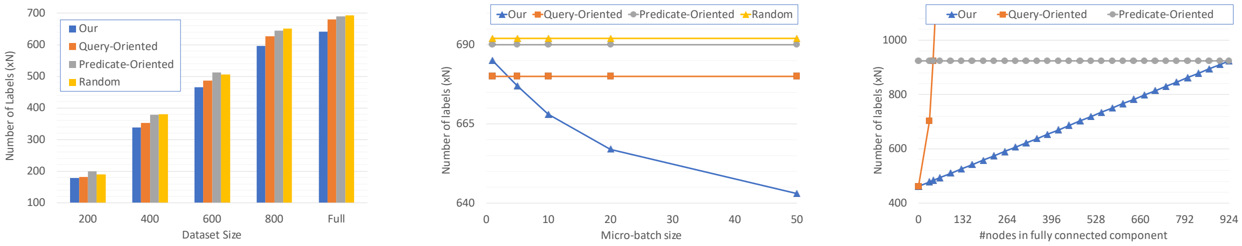
3.1 Experimental setup

Competitors: To evaluate our approach we implemented our algorithm and its three natural competitors. Given a query graph:

- **Predicate-Oriented** - Regardless of any relationships among the queries, the algorithm always labels nodes.
- **Query-Oriented** - Regardless of any relationships among the queries, the algorithm always labels edges.
- **Random** - The algorithm randomly decides whether to label nodes or edges, till it has sufficient information to answer all queries.

Dataset: The dataset contains a real-world public dataset taken from BestBuy with around 1000 search queries over the electronics domain [1]. Each query is written in a structured format, e.g., the query "LG TV" is represented as {"*Brand*" : "*LG*", "*Category*" : "*TV*"}), which allows a straightforward execution of our experiment. The dataset contains 7 different attributes (Category, Model, Brand, ScreenSize, Price, Storage, RAM) with 97% of the queries specify up to 3 attributes: Category, Model, Brand. Out of this subset, around 95% of the queries are of length one or two (66.5% and 29.2% respectively), which corresponds to our assumption on the number of attributes in the e-commerce search queries being small. Interestingly, the longest query in the dataset contains only four attributes (Category, Brand, Model, ScreenSize).

[4] This decision has no effect in the offline case, but helps in the streaming scenario.



(a) Offline results. **(b) Online results.** **(c) Effect of queries dispersion.**

Figure 4: Experimental results.

3.2 Evaluation results

We evaluate our algorithm and its competitors with respect to two configurations: offline and online. Considering the offline configuration, we first execute the algorithms against two types of sets: (i) the full set of relevant queries (that contain one or two expressions), and (ii) randomly selected subsets of varying sizes (200, 400, 600, 800). Considering the online configuration, we simulate random arrivals of successive queries from a stream of queries, and measure the number of classifiers generated by every competitor. Furthermore, we evaluate a *micro-batching* approach, where each iteration is executed with respect to a window of queries with varying sizes (5, 10, 20, 50).

Offline Evaluation: Results are depicted in Figure 4a. As expected, our algorithm outperforms the competitors and reduces the number of classifiers as the dataset size increases. Observe that the query-oriented algorithm is constantly superior to the predicate-oriented algorithm. In general, the predicate-oriented algorithm performs better when there is a small set of attributes which covers multiple queries. On the other hand, for queries that do not share many attributes the query-oriented algorithm performs better. Since our algorithm decides the best strategy per connected component, it outperforms both competitors. While reducing “only” 5%, it may save hundreds of thousands of dollars annually in workers cost [2] and terabytes of storage (entailed in the labeled data and the additional attributes in the relation).

Streaming Scenario Evaluation: Results are depicted in Figure 4b. We can see that even in small micro-batches of size 5, our algorithm outperforms all competitors, and significantly surpasses the competitors for larger windows. Observe that only in the extreme case of completely online scenario it is sub-optimal.

Queries Dispersion Evaluation: In this set of experiment, we examine the effect of attributes overlapping between queries on the number of labeling tasks. To examine it, we fix the size of the graph with n nodes and vary its density by increasing the size of connected components, from 2 to n, which corresponds to the queries dispersion. Figure 4c depicts the results for our algorithm and its two deterministic competitors: predicate-oriented and query-oriented algorithms. First, we can see that our algorithm is consistently superior with comparable competitors in the two extreme cases. In between those cases, our algorithm beats the competitors by large margins and when $|E| = |V|$, it conducts approximately two times less labeling tasks.

4 RELATED WORK

In recent years, crowd workers and human experts are widely employed with various tasks to amend the performance of supervised ML models, e.g., contributing to feature selection [13], learning of semantic attributes [18] and others. Several systems propose hybrid mechanisms [5, 14, 17] that interweave humans and machines. One family of algorithms focus on reducing the error of crowd annotators [4, 8], e.g., combining crowd and machines for multi-predicate classification tasks [11]. In contrast to the probabilistic models employed in these algorithms, they are less suitable for the experts-based setting. To the best of our knowledge, this is the first attempt that aim at minimizing the effort of experts in annotation tasks and the required storage entailed by obtaining classifiers sufficing to answer a set of queries.

5 FUTURE WORK

The most intriguing direction is extending our approach to support longer queries, which is our main ongoing process. Simply extending our graph representation to queries with multiple equality expressions results in a hypergraph representation, thus other approaches might be more suitable. In addition, supporting general weights for various classification tasks, is interesting. Finally, extending our evaluation to additional datasets with different types of queries is a part of our future work.

Acknowledgements This work has been partially funded by the Israel Innovation Authority, the Israel Science Foundation, Len Blavatnik and the Blavatnik Family foundation.

REFERENCES

[1] 2017. BestBuy dataset. https://dataturks.com/projects/Mohan/Best%20Buy%20E-commerce%20NER%20dataset. (2017).

[2] 2017. How to Organize Data Labeling. https://altexsoft.com/blog/datascience/how-to-organize-data-labeling-for-machine-learning-approaches-and-tools/. (2017).

[3] 2017. Retail Site Search Queries Are Getting Shorter. https://www.marketingcharts.com/industries/retail-and-e-commerce-76709. (2017).

[4] Ron Bekkerman and Matan Gavish. 2011. High-precision phrase-based document classification on a modern scale. In *KDD 2011*.

[5] Justin Cheng and Michael S Bernstein. 2015. Flock: Hybrid crowd-machine learning classifiers. In *CSCW*. ACM, 600–611.

[6] Benoît Groz, Ezra Levin, Isaac Meilijson, and Tova Milo. 2016. Filtering with the Crowd: CrowdScreen revisited. In *ICDT 2016*.

[7] Ido Guy. 2016. Searching by Talking: Analysis of Voice Queries on Mobile Web Search. In *SIGIR 2016*. 35–44.

[8] Chien-Ju Ho, Shahin Jabbari, and Jennifer Wortman Vaughan. 2013. Adaptive task assignment for crowdsourced classification. In *ICML*. 534–542.

[9] Ariel Jarovsky, Tova Milo, Slava Novgorodov, and Wang-Chiew Tan. 2018. GOLDRUSH: Rule Sharing System for Fraud Detection. *PVLDB* 11, 12 (2018).

[10] Ece Kamar, Severin Hacker, and Eric Horvitz. 2012. Combining human and machine intelligence in large-scale crowdsourcing. In *AAMS 2012*. 467–474.

[11] Evgeny Krivosheev, Fabio Casati, Marcos Baez, and Boualem Benatallah. 2018. Combining Crowd and Machines for Multi-predicate Item Screening. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 97.

[12] Guoliang Li, Jiannan Wang, Yudian Zheng, and Michael J Franklin. 2016. Crowdsourced data management: A survey. *TKDE* 28, 9 (2016), 2296–2319.

[13] Besmira Nushi, Adish Singla, Andreas Krause, and Donald Kossmann. 2016. Learning and feature selection under budget constraints in crowdsourcing. In *HCOMP 2016*.

[14] Akash Das Sarma, Ayush Jain, Arnab Nandi, Aditya Parameswaran, and Jennifer Widom. 2015. Surpassing humans and computers with JELLYBEAN: Crowd-vision-hybrid counting algorithms. In *HCOMP 2015*.

[15] Fabrizio Sebastiani. 2002. Machine learning in automated text categorization. *ACM computing surveys (CSUR)* 34, 1 (2002), 1–47.

[16] Mohammad S Sorower. 2010. A literature survey on algorithms for multi-label learning. *Oregon State University, Corvallis* 18 (2010).

[17] Chong Sun, Narasimhan Rampalli, Frank Yang, and AnHai Doan. 2014. Chimera: Large-scale classification using machine learning, rules, and crowdsourcing. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1529–1540.

[18] Tian Tian, Ning Chen, and Jun Zhu. 2017. Learning Attributes from the Crowdsourced Relative Labels.. In *AAAI*, Vol. 1. 2.

[19] Xiaohang Zhang, Guoliang Li, and Jianhua Feng. 2016. Crowdsourced top-k algorithms: An experimental evaluation. *PVLDB* 9, 8 (2016), 612–623.

[20] Yudian Zheng, Jiannan Wang, Guoliang Li, Reynold Cheng, and Jianhua Feng. 2015. QASCA: A quality-aware task assignment system for crowdsourcing applications. In *SIGMOD 2015*. 1031–1046.

Short Paper

A Galaxy of Correlations

Detecting Linear Correlated Clusters through k-Tuples Sampling using Parameter Space Transform

Daniyal Kazempour
Ludwig-Maximilians-University Munich
kazempour@dbs.ifi.lmu.de

Lisa Krombholz
Ludwig-Maximilians-University Munich
Lisa.Krombholz@campus.lmu.de

Peer Kröger
Ludwig-Maximilians-University Munich
kroeger@dbs.ifi.lmu.de

Thomas Seidl
Ludwig-Maximilians-University Munich
seidl@dbs.ifi.lmu.de

ABSTRACT

In different research domains conducted experiments aim for the detection of (hyper)linear correlations among multiple features within a given data set. For this purpose methods exist where one among them is highly robust against noise and detects linear correlated clusters regardless of any locality assumption. This method is based on parameter space transformation. The currently available parameter transform based algorithms detect the clusters scanning explicitly for intersections of functions in parameter space. This approach comes with drawbacks. It is difficult to analyze aspects going beyond the sole intersection of functions, such as e.g. the area around the intersections and further it is computationally expensive. The work in progress method we provide here overcomes the mentioned drawbacks by sampling d-dimensional tuples in data space, generating a (hyper)plane and representing this plane as a single point in parameter space. By this approach we no longer scan for intersection points of functions in parameter space but for dense regions of such parameter vectors. By this approach in future work well established clustering algorithms can be applied in parameter space to detect e.g. dense regions, modes or hierarchies of linear correlations in parameter space.

1 INTRODUCTION

Typing into Google Scholar [1] the query **"linear correlation between"** yields around 343.000 scientific works from various domains such as medical science, chemistry, biology, pharmacology, electric engineering, economics, physics. Further limiting the search by adding "multivariate" to the previous query reduces the results down to around 20 scientific works. The insights are here twofold: first, there is a demand for detecting linear correlations in various domains and second, as for now only few scientific works have investigated linear correlations between multiple variables. One real-world example for linear correlations among multiple features is in the wages data set[2]. It contains the statistics of potentially influencing factors of wages from 1985 Current Population Survey. Visualizing the data reveals that there are linearly correlated clusters among the features "years of education", "years of work" and age. As a second example in the scientific domain of water research in a work by [4] the authors revealed a linear correlation between the hydroxyl-radical concentration and the inactivation time of E.coli in a photocatalytic disinfection substance.

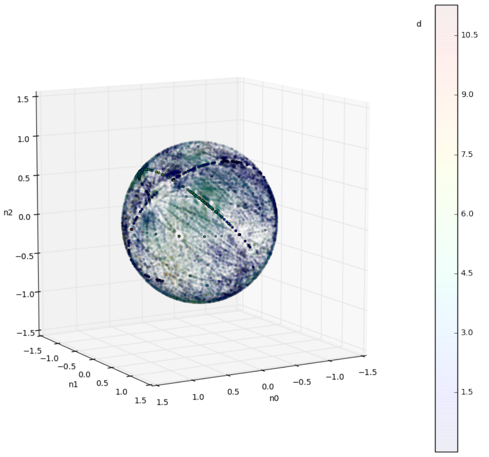


Figure 1: Triple of 3D data points sampled in data space yield this spherical galaxy of correlations in a Hessian Normal Form parameter space. Highly dense regions represent areas with high correlations between data points.

Among the various methods for detecting linear correlated clusters in high dimensional data there is one particular method which relies on parameter space transform: CASH [1]. This method comes, compared to its competitors, with advantages that it is highly robust against noise and detects global linear correlated clusters, being independent of any kind of locality assumption. As we shall elaborate more in detail in section 3 this method works by projecting data points from data space to parameter space becoming a data point functions. The parameter space is then scanned for intersections of such data point functions which represent data points being linearly correlated. However the scan for intersections in parameter space is computationally expensive and the capabilities to further analyze the area around the intersection areas are not given by this approach. We provide the following two major contributions in this work:

- Providing a novel approach for detecting regions of intersection by generating d-dimensional samples from which a linear function is derived. This linear function is then represented in parameter space as a vector eliminating the need to scan for intersections resulting in galaxy-like shapes as seen in Figure 1 and
- An opportunity to analyze further aspects on the detected clusters such as e.g. different densities and thus variances

[1] scholar.google.com
[2] http://lib.stat.cmu.edu/datasets/CPS_85_Wages

Table 1: Selected Parameter Transform Methods

Method	Strategy	Strengths	Weaknesses
Hough Transform	grid-based (accumulators)	Simple strategy	No pruning
CASH	iterative parameter axis splitting	efficiency from DFS	Slow on high-noise data
D-MASC	De-noising with Mean-Shift, Rasterization of functions	Effective against high levels of noise	slow on low-jitter and low-noise data

of linear correlated clusters as well as density connected clusters and their semantics.

2 RELATED WORK

The parameter transform, which is also known as Hough transform, has been first introduced in a patent by Paul V.C. Hough in [6] in context of edge detection on images. The first application of parameter transform in context of detecting (hyper)linear correlated clusters was in the work by [1]. This approach despite its high level of sophistication suffered regarding its runtime if the data has a high amount of noise and jitter. To approach this issue in a recent work [7] the authors provide a method in which the data is pre-aggregated using Mean Shift [3] in data space which yields modes. These modes are transformed into mode-functions in parameter space, which then are rasterized into cells. Those cells from mode functions which are overlapping most with other mode-function cells are considered as candidates for linear correlated clusters. All the mentioned related works aim primarily at finding intersections in parameter space at a specific resolution. The intuition behind this resolution is that the smaller the detected cells in parameter space, the more are the points located on a specific line, plane or hyperplane. An overview on the mentioned linear parameter transform methods are provided in Table 1. It would be of interest to detect e.g. chains of dense regions by applying DBSCAN [5], or centroids by applying centroid or mode based methods such as e.g. MeanShift, or determining hierarchies of linear correlated clusters by applying e.g. single-link to the parameter space. Since we are dealing with functions and not with points in parameter space, we can not apply the mentioned methods.

3 PROJECTING SAMPLED K-TUPLES TO PARAMETER SPACE

Having given an overview on the related work, we elaborate in this section on our work in progress in more detail and compare it to the currently used approach. In the methods described in the related work section, each data point p_i in data space $\mathbb{D}$ is projected to parameter space $\mathcal{P}$ as a data point function $p_i \mapsto f_{p_i}$ where $f_{p_i} := b = y - m \cdot x$, where m represents the slope of a line and b the intercept. An intersection of several of such data point functions at a specific point (m_s, b_s) means that their corresponding data points are located on a line with a common slope and common intercept as it can be seen in Figure 2. Since data points are rarely located perfectly on a line, in the related work it is looked for at least *minpoint* data point functions intersecting within a maximum (m, b)-range. Here *minpoint* is a hyperparameter set by the domain experts. Further regarding the range, the intuition is the following: the smaller the range for the slope and intercept, the more precise are the data points located on an explicit line, and the higher the linear correlation.

In contrast to the related work, in our method all k-Tuples (here all pairs) of data points are taken. From these tuples (p_i, p_j), for each of them a line with a specific slope and intercept is calculated. From the lines we obtain the slope and intercept for each tuple which is a point in parameter space. A point or region in parameter space where these slope-intercept coordinates are densely located represent a correlation in data space as it can be seen in Figure 2. In a formalized manner:

$$\forall (p_i, p_j), where p \in \mathcal{D} : (p_i, p_j) \mapsto f_{p_i, p_j} = (m_{p_i, p_j} . b_{p_i, p_j}) \in \mathcal{P} \tag{1}$$

,

For detecting the dense regions in parameter space we apply density based clustering algorithms such as DBSCAN [5] and OPTICS [2]. In DBSCAN we have two hyperparamters, namely *minpoints* which defines the minimum number of data points which are expected to be located within an *ϵ-neighborhood*. In context of the parameter space, the effects of controlling minpoints and ϵ are the following:

$$\begin{aligned} minpoints &\mapsto |S| \in Corr_{(m,b)}, \\ where \quad S &:= \{(p_0, p_1), ..., (p_i, p_j)\} \subseteq DB \\ and \quad \epsilon &\mapsto \sigma(Corr_{(m,b)}) \end{aligned} \tag{2}$$

Here the intuition is as follows: the minpoints in DBSCAN represent the minimum number of data point tuples which are expected to have the same parameter values (or value ranges) and thus belonging to the same linear correlation. The ϵ hyperparameter represents the variance σ or resolution we allow for the data points around a linear correlation.

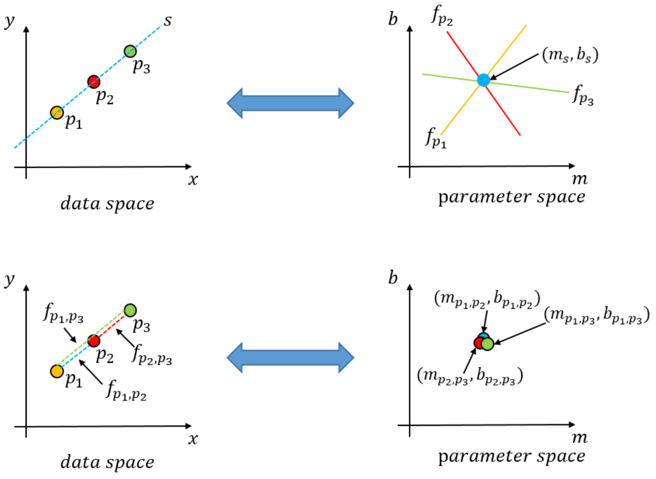


Figure 2: Comparison of current parameter space transform approaches (top) to our method (bottom).

At this point we have to highlight that a question which may rise immediately is: how do we choose the two parameters? This can be partially addressed by using OPTICS which eliminates mostly the issue of determining a proper ϵ value and enables the detection of density hierarchies. Having determined the dense regions or sampled linear correlations, our method computes the median of such regions. The median comes with a pleasant effect that it weeds out the influence of outlier k-tuple correlations in parameter space. As for now our method can be summarized into the following steps as it can be seen in Figure 3

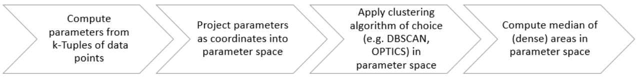

Compute parameters from k-Tuples of data points
Project parameters as coordinates into parameter space
Apply clustering algorithm of choice (e.g. DBSCAN, OPTICS) in parameter space
Compute median of (dense) areas in parameter space

Figure 3: Pipeline with its single stages of our method.

The parameter space provided in our example is the slope-intercept form. There is a variety of linear parameter space representations. Since the slope-intercept form comes with various drawbacks (unbounded dimensions, unable to project y-axis parallel linear correlations, non-ambiguous representations in higher dimensions etc.) we use in this work the Hessian Normal Form (HNF) since it comes with the advantages of the slope-intercept and other representations but with none of their disadvantages as stated in [8]. In the HNF representation a line or (hyper)plane is represented through:

$$\begin{aligned} \delta =< p, n >, where p := (p_0, p_1, ..., p_d) \in \mathcal{D}^d, \\ and \quad n := (n_0, n_1, ..., n_d) \in \mathcal{P}^d \end{aligned} \tag{3}$$
,

Here n represents the normal vector and δ represents the distance from origin orthogonal to the hyperplane.

Further we shall see in the complexity section why a full k-Tuple construction is computationally infeasable, especially in higher-dimensional settings.

4 COMPLEXITY

The runtime complexity of the related work CASH is in worst case $O(s \cdot c \cdot n \cdot 2^d)$ where s reflects the number of intersections and thus the resolution of the grid, c denotes the number of found clusters, n represents the number of data points in the data set and d stands for the dimensionality of the data space. In comparison D-MASC has a runtime complexity of $O(Tn\log(n)+(\frac{len(bounds_d)}{w})^d m^2)$ with T denoting the number of iterations of the MeanShift algorithm for initially reducing noise and jitter in data space, $len(bounds_d)$ representing the range of the parameter space range in which we are looking for intersections, w standing for the width of the cells being generated in a rasterization process and m for the number of resulting modes after applying MeanShift. Our method requires for computing the parameter coordinates for all data point all k-tuples, where k corresponds to the dimensionality d of the original data set. With regards to the dimensionality, we require in a 2D data set all two-tuples, in a 3D data set all three-tuples etc. This yields a runtime of $O(\binom{n}{d})$. DBSCAN requires with an indexing structure that executes the neighborhood query an overall runtime of $O(n\log(n))$. Thus we get for our method in total a runtime of $O(\binom{n}{d} + n\log(n))$. From a runtime point of view, our method has an exponential runtime with regards to the dimensionality like CASH and D-MASC. However instead of generating all $\binom{n}{d}$ tuples, one strategy is to sample over the data points. In some preliminary experiments we could observe that sampling yielded in most cases as accurate results as performing a full enumeration of all k-tuples. As for now, our assumption is, that if data points are correlated in data space, so do their samples reflect the correlation up to a certain extent. However, since this is a work in progress, an exhaustive analysis on theoretical as well as on experimental level is required to prove the assumptions.

5 EXPERIMENTS AND DISCUSSION

Now that we have elaborated on our method and its runtime complexity we provide here experiments which focus on the quality of the detected clustering results. As a first experiment we take one of the data sets which is used in D-MASC. The two-dimensional data set consists of 100 data points contributing to three linear correlated clusters with irregular densities. To these 100 data points 90% noise is contributed. According to the pipeline mentioned in section 3, first all 2-tuples of data points are created, then projected into the parameter space. In the parameter space we can already observe dense regions. Applying OPTICS we get the following plot as seen in Figure 4.

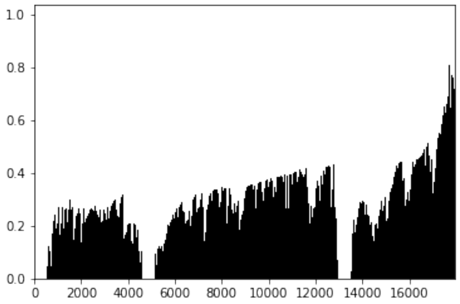


Figure 4: OPTICS plot of the parameter vectors in parameter space.

The three valleys indicate three almost equally high-dense regions in parameter space. As a result of DBSCAN with $minpoints = 33$ and $\epsilon = 0.015$ we obtain the following regions in parameter space which are marked with an 'x' in Figure 5

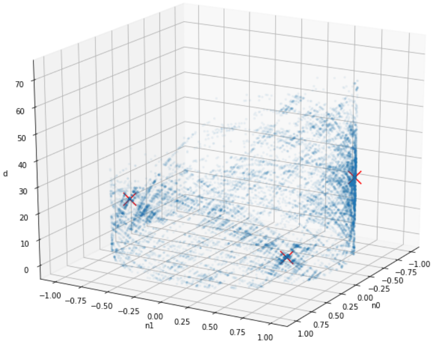


Figure 5: Detected three high-density regions in parameter space.

After having computed the median from each of the dense regions, our method was capable of detecting all three linear correlated clusters with all the data points being assigned to their respective cluster as it can be seen in Figure 6.

As a teaser for its performance on data sets with a dimensionality higher than two, we have a three dimensional data set

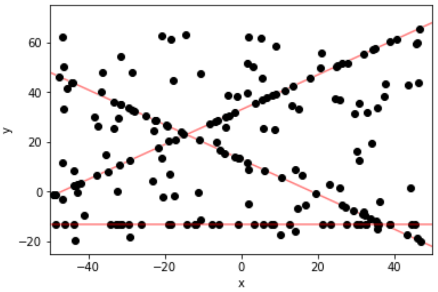


Figure 6: Detected three linear correlated clusters in data space.

consisting of 66 data points. From which 36 data points belong to planar correlations and 30 data points are randomly generated noise. The 36 data points belong to two planar correlated clusters, with 18 data points per correlation. In parameter space we generate thus $\binom{66}{3} = 45760$ parameter vectors in parameter space. Our method detects in parameter space two clusters as it can be seen in Figure 7 where two very deep valleys can be seen. The first figure in the introduction of the paper is the actual parameter space of this data set. The three axes represent each a dimensional of the normal vector. The color (black) represents the distance δ. The parameters for the density based clustering were $minpoints = 250$ and $\epsilon = 0.01$.

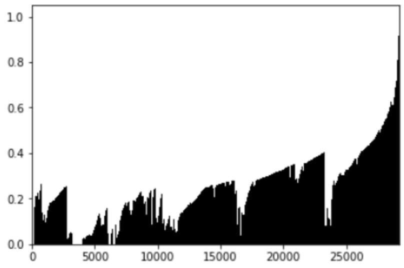


Figure 7: OPTICS plot of the 3D data set with two highly-dense clusters.

As an result we get both planes detected correctly and all points assigned to their corresponding planar correlated clusters as seen in Figure 8.

6 CONCLUDING REMARKS AND FUTURE PROSPECTS

In this work in progress, we have provided a first concept for a different approach in detecting (hyper)linear correlated clusters in parameter space by sampling k-Tuples in data space, generating k-dimensional (hyper)planes. The parameters of these (hyper)planes are projected to parameter space. In the parameter space we used as an example density based methods for detecting highly dense regions. This approach of dealing with points in parameter space instead with (hyper)linear functions opens new possibilities of analysis. Primary targets for future work are evaluating sampling strategies, applying the experiments to high-dimensional and also real world data and evaluating different clustering models in parameter space (hierarchical, centroid-based, subspace etc.). We hope to encourage with this paper to

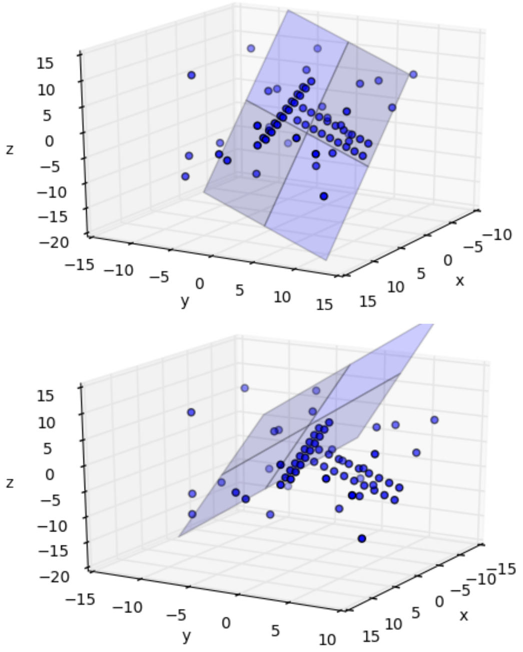


Figure 8: Two planar correlated clusters

not only develop different approaches for clustering using parameter transform itself, but also fostering the research of parameter space transformation based methods making discoveries in the galaxies of correlations.

ACKNOWLEDGMENTS

This work has been funded by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IS18036A. The authors of this work take full responsibilities for its content.

REFERENCES

[1] Elke Achtert, Christian Böhm, Jörn David, Peer Kröger, and Arthur Zimek. [n. d.]. Global Correlation Clustering Based on the Hough Transform. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 1, 3 ([n. d.]), 111–127.

[2] Mihael Ankerst, Markus M. Breunig, Hans peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points To Identify the Clustering Structure. ACM Press, 49–60.

[3] Yizong Cheng. 1995. Mean shift, mode seeking, and clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17, 8 (1995), 790–799.

[4] Min Cho, Hyenmi Chung, Wonyong Choi, and Jeyong Yoon. 2004. Linear correlation between inactivation of E. coli and OH radical concentration in TiO2 photocatalytic disinfection. *Water Research* 38, 4 (2004), 1069 – 1077.

[5] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*.

[6] Paul VC Hough. 1962. Method and means for recognizing complex patterns. (Dec. 18 1962). US Patent 3,069,654.

[7] Daniyal Kazempour, Kevin Bein, Peer Kröger, and Thomas Seidl. 2018. D-MASC: A Novel Search Strategy for Detecting Regions of Interest in Linear Parameter Space. In *Similarity Search and Applications - 11th International Conference, SISAP 2018, Lima, Peru.* 163–176.

[8] Daniyal Kazempour, Andrian Mörtlbauer, Peer Kröger, and Thomas Seidl. 2018. Mirror Mirror on the Wall, What is the Fairest Linear Parameter Space Representation of All? On Representations of Linear Parameter Space in Context of Clustering. In *Proceedings of the Conference "Lernen, Wissen, Daten, Analysen", LWDA 2018, Mannheim, Germany.* 169–173.

Insights into a running clockwork: On interactive process-aware clustering

A Vision Paper

Daniyal Kazempour
Ludwig-Maximilians-Universität München
Munich, Germany
kazempour@dbs.ifi.lmu.de

Thomas Seidl
Ludwig-Maximilians-Universität München
Munich, Germany
seidl@dbs.ifi.lmu.de

ABSTRACT

In recent years the demand for having algorithms which provide not only their results, but also add explainability up to a certain extent increased. In this paper we envision a class of clustering algorithms where the users can interact not only with the input or output but also intercept within the very clustering process itself, which we coin with the term process-aware clustering. Further we aspire to sketch the challenges emerging with such type of algorithms, such as the need of adequate measures which evaluate the progression through the computation process of a clustering method. Beyond the explainability on how the results are generated, we propose methods tailored at systematically analyzing the hyperparameter space of an algorithm, determining in a more ordered fashion suitable hyperparameters rather then applying a trial-and-error schema.

1 INTRODUCTION

Performing a query to the computer science bibliography search engine dblp with the keyword "explainable"[1] delivers an interesting insight looking at the "refine by year" area of the search result as it can be seen in Figure 1 . As for now (November 2018) the number of publications dealing with the aspect of explainability increased from 33 in 2017 up to 101 in 2018. However the scientific works are tailored towards deep learning systems. Since it can be agreed on that deep learning systems have some kind of black box character as stated in e.g. [8], classical clustering methods are fairly transparent regarding the way they generate the clustering results. In the majority of publications dealing with clustering, the whole system can be represented as in Figure 2. Data is given to a clustering algorithm additionally with hyperparameters. The algorithm of choice is executed on that data and yields a clustering result. If clustering methods are already transparent and so easy to comprehend, why should we bother then with the aspect of explainability? Despite the fact that we know how a clustering algorithm works, questions arise like e.g. "what is a good hyperparameter setting?", or "how do my clusters change (or not change) with different hyperparameter settings?", "why did the clusters change that particular way choosing different settings?". Only because we know how the clustering algorithms work, we do not yet fully utilize the potential of this knowledge. Having algorithms like e.g. MeanShift [3] domain experts may want to know what happens with the emerging clusters during the clustering process to understand the resulting clusters. In this vision paper we elaborate in the upcoming sections more in detail on different aspects which are ultimately connected to the idea of including insights from the clustering process itself. For this we address in our vision targets such as points of interaction, hyperparameter analysis, methods and measures as well as potential impacts of our vision on areas such as explainability and didactics. During our tour through these targets we will mention the current related work, and highlight potential difficulties and needs motivating the need for process-aware clustering.

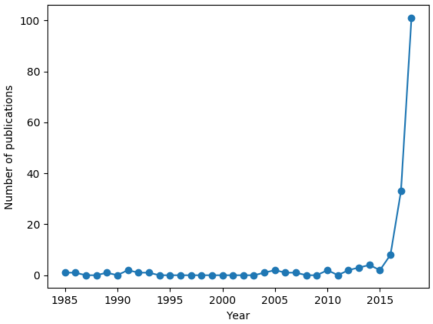


Figure 1: Number of publications indexed at dblp for the keyword explainable from 1985 to 2018.

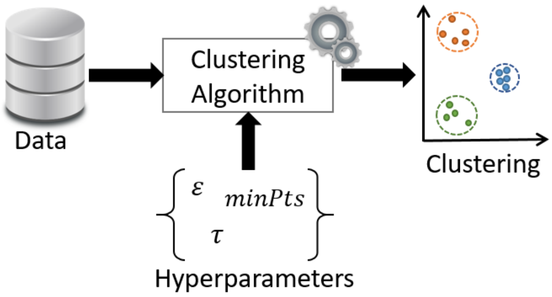


Figure 2: A classic clustering pipeline.

2 INTERACTION TARGETS

In context of interactiveness a rich body of literature is present. Exemplarily we mention iPCA as an interactive tool for PCA-based visual analytics [5]. Despite its high level of sophistication regarding the used visualization techniques, it enables the user to interact with the system *after* a PCA has been performed. The users can not browse through some imtermediate computation steps of PCA and intercept. Also advanced interactive clustering tools like VISA [2] enable human interaction for inspecting the detected clusters and subspaces but do not facilitate to intercept within the clustering process itself. Even in a more recent work

[1] https://dblp.uni-trier.de/search?q=explainable

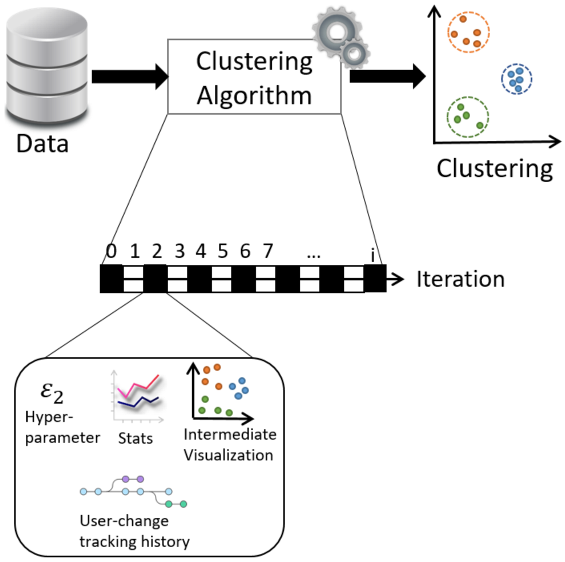


Figure 3: Clustering pipeline in an interactive setting where the users can intercept within each of the steps of the clustering.

[9] , the users can decide on modifying the results in context of hierarchical clustering, yet they do not offer a history of steps which the utilized hierarchical clustering has performed so far.

In a more recent work, a simple but also limited tool PARADISO [6] provides the users the opportunity to explore and intercept within the clustering process itself. In PARADISO our classic clustering pipeline from Figure 2 is re-defined to a pipeline as seen in Figure 3. Here the users can intercept at each of the iterations of a MeanShift algorithm. In this algorithm data points roam within a specific bandwidth (also known as Parzen window) to their mean with regards to the position of other data points located within the bandwidth. The data points thus roam themselves towards their respective mode. In PARADISO the bandwidth hyperparameter can be modified, stats regarding the current clusters are provided, and visualizations for intermediate results at each iteration step are given.

2.1 Multi-instance hyperparameter settings

One aspect from PARADISO which we'd like to emphasize on, is the capability to assign at any iteration step of the algorithm a different bandwidth value, which we coin with the term multi-instance hyperparameter setting, where multi-instance refers to either the iteration step or in general the steps of a clustering algorithm. Going beyond the classical setting where the users provide in the beginning one fixed (set of) hyperparameter(s) which remain(s) valid until the end of an algorithms run, leads us to the case of intercepting within the clustering process and deliberately changing the hyperparameter values at different times. This method becomes even more significant in streaming context. By the simple fact that in a stream setting the data changes over time, a hyperparameter which has been selected in the beginning may no longer be suitable. Thus we may need different hyperparameter values at different times while a clustering algorithm is computing at each iteration on different snapshots of the data.

Further in context of multi-instance hyperparameter settings it is vital to keep track of which changes the users have made. Similar to a version control system, the users shall be given a tool at hand to keep track of the history of changes they have made. By knowing which changes have been made at which iteration, the users can move arbitrarily through the iteration timeline and create alternative branches of changes. Despite giving new targets of interaction, like changing at arbitrary iterations the hyperparameters, the concepts of interactivenes as described so far do bare their own problems which need to be addressed: While on small amounts of data the intermediate computations can be stored, it is in-feasible to store each of the iteration steps at larger data sets. Here potential research targets are e.g. compression strategies and significance measures to determine which of the iteration steps are relevant (enough) to be kept in memory. Such measures could indicate e.g. at which iterations the most changes are taking place.

One question which may arise in context of multi-instance hyperparameter settings is: what are the implications of such an approach? What would it change if we choose different bandwidths *within* a MeanShift run vs. re-running MeanShift multiple times with different bandwidths? Suppose we are given a data set and apply the MeanShift algorithm with a specific bandwidth (e.g. 0.7) in the beginning. Depending on the data set it may happen that a subset of points collapses after a few iterations into one mode. Intercepting a few iterations beforehand by choosing a much smaller bandwidth (e.g. 0.2) may prevent the collapse to one singular mode, leading to multiple modes. Depending at which iteration the interception has taken place, a re-run of MeanShift with a bandwidth of 0.2 may not lead to the same result as in the variant where we start with 0.7 and change it after i-iterations to 0.2. However a proof for this claim is required, which is subject of future works further investigating the multi-instance hyperparameter settings aspect. Yet, we'd like to provide a brief intuition for why it can lead to different results using different bandwidths at different iterations. We envision that MeanShift executed in two instances with each having a bandwidth b_0 and b_1 is like two objects moving among distinct trajectories where time is represented through the single iteration steps. Intercepting in the MeanShift with b_1 is comparable to deflecting the object b_1 leading to a potentially different trajectory than its original path by forces from other modes acting on the object. It may lead to the same destination as MeanShift executed with b_0 but can as well have its destination at a different positions in data space.

3 MEASURING METHODS

Besides the concept of intercepting into the clustering process itself and assigning multiple instances of hyperparameters at different iteration steps, there is a need for measuring methods which are suitable in context of process-aware clustering. The requirements to such measures would be e.g. to capture the dynamics within the clustering process like data points being assigned to specific clusters or subspaces and the change of such over the course of a clustering run.

3.1 Entanglement

The first propositions for such a measure were made in [7]. Here the concept of entanglement has been raised which is intended to support interactive data clustering with the purpose to supply additional information to users. In [7], the entanglement between

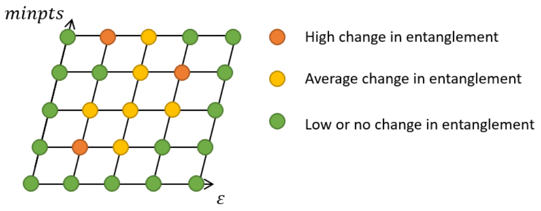

minpts
ε
High change in entanglement
Average change in entanglement
Low or no change in entanglement

Figure 4: Parameter grid, with the parameters of DBSCAN. Each point represents the entanglement difference computed to its previous entanglement values.

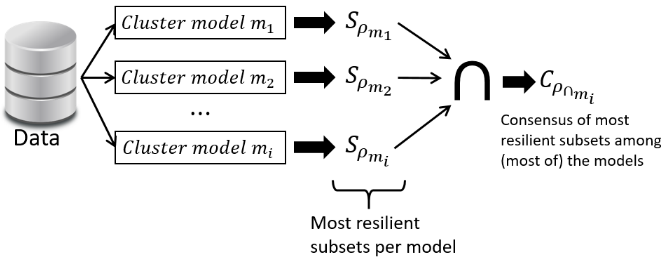


Figure 5: Resilience in context of ensemble clustering.

two data points is defined as the dynamic time warp distance between the trajectories of both points roaming over time to their modes in a MeanShift clustering algorithm. If those data point trajectory pairs are sufficiently similar to each other over the iterations, they are considered as entangled. This definition is however limited to centroid-based approaches like in MeanShift. In our vision we aspire to provide such entanglement definitions for various clustering models (e.g. density-based, hierarchical, spectral, subspace, correlation etc.) and, if possible, a more generalized definition of it.

3.2 Resilience

The entanglement of two data points can be highly varying or remaining stable. This stability depends on the chosen hyperparameters. In [7] the so called resilience has been defined, which computes the variation of the entanglement over different hyperparameter settings. The lower the variance of the entanglement at different hyperparameters, the more resilient is the specific subset of data points. However the work in [7] is highly limited to the MeanShift setting. In our vision auspicious directions include the research of resilience in context of other clustering models, especially also in context of subspaces where the resilience and entanglement is not only determined among subset of data points but also among a subset of subspaces. It remains for future work to investigate in how far the concepts of entanglement and resilience can be applied to different clustering models (e.g. density based, hierarchical, subspace, correlation, spectral etc.) and with which kind of adaptations.

3.3 Hyperparameter Analysis

The research on entanglement and resilience bares potential for the aspect of hyperparameter analysis. In many of the publications on clustering so far, the hyperparameters in e.g. the conducted experiments were chosen on an experience basis. Different settings are tried out until a good (enough) clustering result is achieved. We envision that measures like e.g. resilience can be used together with more systematic approaches in determining the quality of parameters. For example in context of density based clustering like in DBSCAN [4], where we have a minpnts and ϵ parameter, a grid-based approach can prove effective as seen in Figure 4.

Here the granularity of the grid is set by the users on how fine they want to sample the grid. It is advised to start with a coarse grid first. In an initial step, one computes for each of the (minpnts, ϵ) combinations the entanglement. Then the resilience over different parameter settings is computed. Those points on the grid (which represent (minpnts, ϵ) hyperparameters) which have the highest difference in resilience can be considered as interesting, since this specific hyperparameter setting impacts the entanglement of subsets of points. Then in a follow-up step, the grid is refined around such interesting points on the grid. The users can thus successively explore the impact of the hyperparameters. In this context the users can use clusering with a different goal in mind: given a subset of points S_{in} where we want them to be in the same cluster, and further we are given a subset of points S_{not-in} which we do not want to be in the same cluster. Under which parameter settings do we get a clustering fulfilling these constraints? In this setting it is also of interest for future research to approach the scientific problem: how can the users be informed on how "unintuitive" it would be for the clustering algorithm with its underlying model to respect the given constraints as in S_{in} and S_{not-in}?

3.4 Ensemble Settings

The concept of resilience can further be used as a measure for ensembles of clustering models. Given a dataset, in an ensemble setting, the most resilient subsets are considered per model. The intersection of the resulting resilient subsets over all the models yields a subset which is highly resilient among all (or most) of the clustering models which can be seen in Figure 5. The intuition of this intersection subset is, that it is a consensus of different cluster models at different hyperparameter settings, yielding clusters or at least subsets of clusters being valid among different cluster models. On the contrary, removing the consensus subset, leaves subsets which may be (a) not resilient among their respective models, or (b) are highly resilient, but only within their models. The latter case is exciting since this potentially facilitates the extraction of highly *model specific subsets* of data points.

4 IMPACTS

Having elaborated on interaction targets, and measuring methods, we now discuss potential impacts of our vision within the upcoming subsections providing some outlook on the potential magnitudes that this vision may trigger.

4.1 Explainability

Since we have mentioned in the beginning of this vision paper the aspect of explainability we now elaborate on this aspect as a potential impact. The propositions for methods and measures so far enable to look at different parts of an clustering algorithm. The entanglement supports to look into the clustering process itself detecting data points which have the same trajectories or are assigned to the same clusters, subspaces etc. Resilience enables to understand which data points remain (based on entanglement) together even over different hyperparameter settings. The interaction concept of multi-instance hyperparameter setting permits the users to intercept and explore the effects of choosing different values for hyperparameters during the clustering process. Each of the mentioned methods and aspects can reveal information which can, best to our knowledge, not be gained by simply adding data and some chosen hyperparameter values for a clustering algorithm that once executed only returns the clustering results. However, words of caution are also need to be stated here, since questions that remain open are e.g.: How are the information from the clustering process itself best being presented to the users, and in which form? Which other forms of interaction-tracking may be required, since with increasing number of interaction targets, also the complexity of what can be changed and observed increases.

4.2 Didactics

A connection which may not be obvious on first sight, but becomes rather evident when thinking more from an educational perspective is the relation between explainability and didactics. Besides the theory and exercises in tutorials, demonstrations of the discussed clustering algorithms significantly contribute to the understanding. For such purposes tools like e.g. ELKI [1] exist which can also be used to demonstrate how datasets are clustered and to explore the effects of different hyperparameters by re-running the algorithm every time with a different parameter setting. We are convinced that in our vision process-aware clustering can enable even more insights into the clustering process itself, understanding the behavior, the strengths and also limitations of the process-aware clustering models. It may further aid graduate students which are writing their bachelor or master thesis to evaluate the effects of potential enhancements that they develop and apply to the clustering models they are working with, providing a different approach to evaluate.

5 CONCLUSIONS

In this vision paper we have elaborated on the idea of process-aware clustering, and on its concepts which can be seen summarized in Figure 6. Regarding interaction targets, we have the pillar of multi-instance hyperparameter settings and the pillar of hyperparameter-change tracking with history. While the first pillar enables the interception into the clustering process, the latter provides the capability to track changes and explore different settings. The pillars of entanglement and resilience from the aspect of measuring methods provide the very basis for (a) hyperparameter analysis, which itself serves as the foundation for (b) ensemble settings. All the mentioned fields pose the very foundation for explainability and didactics in the field of interactive process-aware clustering. Further ideas regarding the vision would be to connect process-aware clustering with the research field of process mining. Since various methods are developed for the analysis of processes, some of them (with or without adaptations) may be beneficial to the process-aware clustering concept. Since this vision aims to reveal what happens within the clustering process itself, we conclude this vision paper with a quote from Dr. Faust from a tragic play by Johan Wolfang von Goethe:

That I may understand whatever
Binds the world's innermost core together,
See all its workings, and its seeds,
Deal no more in words' empty reeds.
−−Faust, lines 382–385.

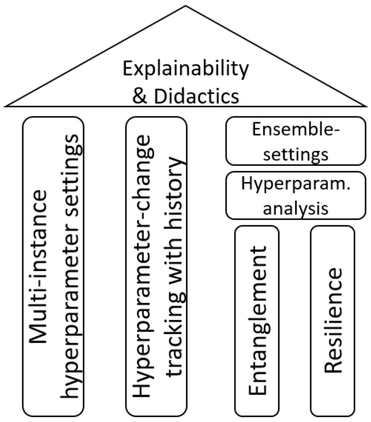


Figure 6: Pillars of process-aware clustering.

ACKNOWLEDGMENTS

This work has been funded by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IS18036A. The authors of this work take full responsibilities for its content.

REFERENCES

[1] Elke Achtert, Hans-Peter Kriegel, and Arthur Zimek. 2008. ELKI: A Software System for Evaluation of Subspace Clustering Algorithms. In *Proceedings of the 20th International Conference on Scientific and Statistical Database Management (SSDBM '08).* 580–585.

[2] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. 2007. VISA: Visual Subspace Clustering Analysis. *SIGKDD Explor. Newsl.* 9, 2 (Dec. 2007), 5–12.

[3] Yizong Cheng. 1995. Mean shift, mode seeking, and clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17, 8 (1995), 790–799.

[4] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96).* 226–231.

[5] Dong Hyun Jeong, Caroline Ziemkiewicz, Brian Fisher, William Ribarsky, and Remco Chang. 2009. iPCA: An Interactive System for PCA-based Visual Analytics. *Computer Graphics Forum* (2009).

[6] Daniyal Kazempour, Anna Beer, Johannes-Y. Lohrer, Daniel Kaltenthaler, and Thomas Seidl. 2018. PARADISO: an interactive approach of parameter selection for the mean shift algorithm. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management, SSDBM 2018, Bozen-Bolzano, Italy, July 09-11, 2018.* 26:1–26:4.

[7] Daniyal Kazempour and Thomas Seidl. 2018. Identifying Entangled Data Points on Iteration Trajectories of Clusterings. In *Proceedings of the Conference "Lernen, Wissen, Daten, Analysen", LWDA 2018, Mannheim, Germany, August 22-24, 2018.* 174–178.

[8] Ravid Shwartz-Ziv and Naftali Tishby. 2017. Opening the black box of deep neural networks via information. *arXiv preprint arXiv:1703.00810* (2017).

[9] Sharad Vikram and Sanjoy Dasgupta. 2016. Interactive Bayesian Hierarchical Clustering. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016.* 2081–2090.

Interpolation-friendly B-trees: Bridging the Gap Between Algorithmic and Learned Indexes

Ali Hadian
Imperial College London
hadian@imperial.ac.uk

Thomas Heinis
Imperial College London
t.heinis@imperial.ac.uk

ABSTRACT

Index structures such as B-trees and bloom filters are the "petrol engines" of database systems, but these structures do not fully exploit patterns in data distribution. To address this, researchers have suggested using machine learning models as "electric engines" that can entirely replace index structures. Such a paradigm shift in data system design, however, opens many unsolved design challenges, e.g. more research is needed to understand the theoretical guarantees and design efficient support for insertion and deletion.

In this paper we adopt a different position: index algorithms are good enough, and that instead of going back to the drawing board to fit data systems with learned models, we should develop lightweight "hybrid engines", where a *helping model* "boosts" classical index performance using techniques from machine learning.

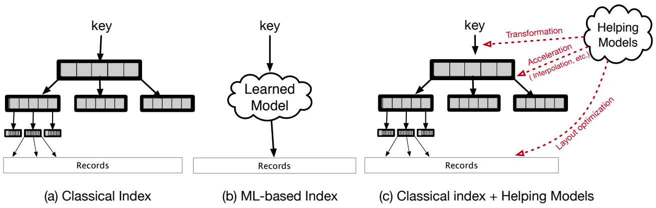
As a case study, we show how interpolation techniques can be integrated with a B-trees with negligible change to the structure and memory footprint of the base algorithm. We show that such a simple helping model, called Interpolation-Friendly B-tree (IFB-tree), can boost the speed of B-trees by up to 50%.

1 INTRODUCTION

Data engines exploit efficient implementations of algorithmic index structures, such as hash tables, B-trees, radix-trees, bloom filters, etc. Index structures fit different tasks and workloads, e.g. B-trees are efficient for range lookups and hash tables are the tool of choice for point queries. Different aspects for each of these indexes have been studied for decades. Many tuning suggestions exist, for example, for B-trees to efficiently fit hardware specifications including the latency/bandwidth ratio and cache-sizes, and various extensions have been suggested to improve the performance even further. [4, 5, 11, 12].

Recently, it was suggested that general-purpose index structures such as B-tree cannot exploit common patterns in data distribution of the real-world data, hence proposing the use of machine learning (ML) models [9]. In this approach, a learned model entirely replaces a classical index and learns how to perform the same behavior. For example, a B-tree can be replaced by a learned index (based on deep learning models) that takes the key as input and *estimates* the position of the corresponding data record in a sorted set, i.e., a clustered index or sorted list of keys.

Learned indexes can be effective for read-only lookups over many data distributions. However, lack of theoretical performance guarantees for a learned model and the challenges for handling update operations in a learned model has lead to an extensive debate in the community. In general, and similar to the analogy of 'petrol vs electric' engines, adopting machine learning techniques could yield elegant methods in data management but the design and maintenance of an ML-enhanced DBMS opens numerous challenges that require comprehensive research. In the meantime, we need hybrid engines that do not render current algorithms and indexes unnecessary. We refer to the hybrid engines as "helping models". Figure 1 illustrates a classical B-tree index (a) vs a learned index (b), the hybrid approach (c) where helping models improve a classical index on different stages.



(a) Classical Index (b) ML-based Index (c) Classical index + Helping Models

Figure 1: Alternatives for classical indexes

In this work, we try to bridge the gap between traditional index structures and the ML approach, suggesting that a helping model can be *integrated* with a traditional index without incurring undue overhead. We suggest that the ability of an ML model to make *distribution-aware* indices is not a rival for classical indexes, but indeed complements them.

We therefore present the *Interpolation-Friendly B-tree* (IFB-tree), which sticks with the traditional B-tree structures to enjoy their performance guarantees, yet is able to exploit the basic ideas of a learned index such as *bounding error-windows* for intra-node lookups. More specifically, we demonstrate how linear interpolation can be integrated with a B-tree index to reduce unnecessary operations with a negligible memory footprint. The process can be done by analyzing a B-tree and labeling *interpolation-friendly* nodes, so that further access to such nodes can be accelerated using interpolation. Our experiments show that the Interpolation-Friendly B-trees (IFB-trees) gives up to 50% improvement over B-trees.

2 TO B-TREE OR NOT TO B-TREE?

2.1 B-tree overview

The B-tree is a generic data structure. State-of-the-art B-trees are n-ary binary trees, i.e., generic data structures that do not assume any specific pattern in the underlying key distribution [4, 5, 11, 12]. B-tree lookup time consists of the time to search within each node, plus the time to follow the pointers and load the next nodes in bottom levels. This takes time and requires keeping a portion of data (inner nodes) in memory.

One of the key issues in a B-tree is that, to locate an item in each of the B-tree nodes, the entire node should be searched, either by linear scan or binary search.

In the following, we consider the alternative approaches that provide "smarter" methods for looking up keys in B-trees. The general solution for a more targeted lookup in a B-tree node (i.e., a sorted list of keys) is to use a method that effectively leverages the underlying key distribution.

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.93

2.2 Learned indexes

Machine learning provides various tools for learning a distribution. Recently, Kraska et al. [9] suggested that if a machine learning model can fully learn the CDF of the key distribution, it can directly predict the location of the queried keys in the table pages. They exploited a hierarchy of deep learned models to help shrink the problem space. Since all predictions in machine learning are susceptible to errors, the *maximum error* for the models needs to be computed in advance. Once the position is estimated, we can scan a *range* around the estimated position in the table pages. This is called the *error window*, i.e. $[\hat{pos}(key) - MaxErr_{\text{Left}}\ ,\ \hat{pos}(key) + MaxErr_{\text{Right}}]$. As shown in Figure 1(a,b), a learned index entirely replaces a classical data structure (like B-tree). If the prediction error is small enough, a learned model can ultimately beat B-tree's lookup time.

Challenges of learned indexes. A learned index can learn many complex distributions and locate the position of the key with a considerably low error margin. The idea of ML-indexes is be very effective for modeling some specific CDFs, yet ML models are not efficient for every key distribution. Moreover, state-of-the-art learned models are mostly suitable for read-only data. Despite that these models can handle a few insertions and deletions by reserving empty spaces in the sorted list, more update requests require re-training the model, which is computationally expensive. Since the model is bound to the mapping between the key's value and the position's offset, any changes to this mapping, such as inserts and deletes, make the learned model ineffective. Moreover, learned indexes require that the table pages be stored in a continuous block of memory so that records can be retrieved once the position is estimated. This assumption does not hold when the keys are not sorted, e.g. in a secondary (non-clustered) index, hence limits the applicability of a learned index.

2.3 IFB-tree

We believe that the main issue behind the challenges in a learned index is that it "replaces" an algorithmic data structure, but cannot deliver all the same operations. Alternatively, we suggest that a similar ideas of a learned index can be embedded within a data structure, such as the B-tree. Unlike the ML index approach which entirely replaces conventional index structures with a learned model over the sorted page of keys, we stick with the genuine structure of the B-tree. In fact, the B-tree itself is very effective in modeling the CDF of the key distribution by repeatedly breaking it into smaller parts. We consider the fact that a considerable share of B-tree lookup time is spent on intra-node search, i.e., to find the smallest value in a B-tree node that is larger (or equal) to the query point q. If we manage to outperform the intra-node search in a B-tree, the efficiency of B-trees can be improved without changing its theoretical performance guarantees.

Learning node distribution. Since keys are sorted in each B-tree node, we can think of a smarter lookup instead of naively doing a linear or binary search over all keys in the node. Similar to the learned indexes, a tiny model can predict the position of a key in each node, which reduces the search space within each B-tree node. However, even the simplest models (e.g. linear regression) requires keeping model parameters for each node in memory, and the cost of managing and loading the parameters does not lead to any performance improvements.

Node interpolation. Following the general idea of ML indexes, our solution is to estimate a range that guarantees the target key resides in. While *interpolation* looks like a naive approach, it could be very effective and has near-zero cost: it simply requires one bit per key, and the computation is very fast and simple. Figure 2b illustrates how interpolation can be done in a B-tree node. If the queried key v_q is between the two keys in a B-tree node, say v_i, v_{i+1} , it should be located on the next level, which could be the next node in the tree or a node in the physical table pages if the current node is a leaf. The interpolated index of the entry corresponding to v_q is: $\hat{p_q} = \left\lfloor \frac{v_q - v_i}{v_{i+1} - v_i} \times \texttt{node\_size} \right\rfloor$

The interpolation has an error, unless the keys follow an exact arithmetic progression, which is almost never the case in real-world except in case of auto-generated key sequences. Let's assume the maximum error for any value in a node is $\text{MaxErr}_{\text{Left}}$ and $\text{MaxErr}_{\text{Right}}$, respectively. This means that the area between $\hat{p_q} - MaxErr_{\text{Left}} + \hat{p_q} + MaxErr_{\text{Right}}$ should be scanned to find the actual position corresponding to v_q.

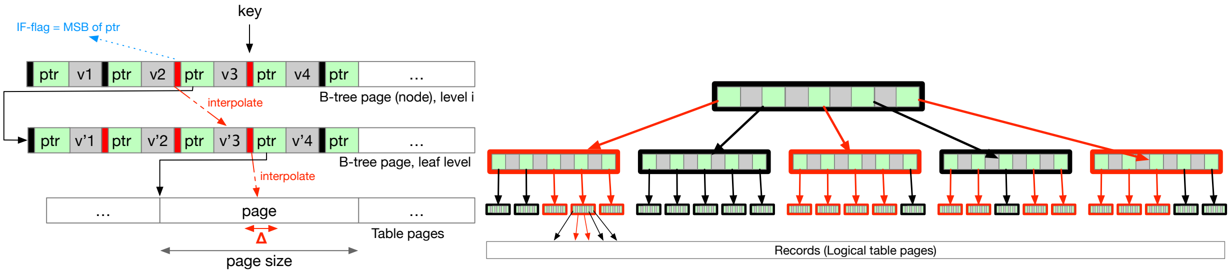
Global error window. Storing the errors values $\text{MaxErr}_{\text{Left}}$ and $\text{MaxErr}_{\text{Right}}$ alongside the keys in the B-tree node will increase the memory footprint and hence greatly reduces the performance. Moreover, having variable error values for each node makes it challenging to exploit the SIMD capabilities for interpolation and search. We prefer not to store any extra information in each node and keep the structure of B-tree untouched to the extent possible. Therefore, we define a *global* threshold for error for the entire B-tree, called the *interpolation diameter* Δ. We call a B-tree node "*interpolation-friendly*" if any key can be found with interpolation error of at most Δ, i.e. $\Delta \geq max(MaxErr_{\text{Left}}, MaxErr_{\text{Right}})$. If a node is interpolation-friendly, we can estimate the location corresponding to the query point q, say $\hat{p_q}$, and then search the area $\left[\hat{p_q} - \Delta, \hat{p_q} + \Delta\right]$, as the result is guaranteed to be in this area. Another advantage of having a pre-defined value for Δ, is that the length of the loop for searching in a node is defined in compile time, hence the program can be efficiently optimized using efficient branch-free and/or loop-unrolled code for either linear or binary search [2].

Marking IFB-tree nodes. Building an IFB-tree involves two steps. The first step is to build a simple B-tree. In the second stage, we analyze all nodes in B-tree to find which nodes are interpolation friendly. More specifically, a node is interpolation-friendly if $|p_{\text{key}_i} - i| < \Delta, \forall\ 0 \leq i \leq K$. Once a node is identified as interpolation-friendly, it should be flagged somewhere. In case that the interpolation error for a node is above Δ, the default search procedure should be used without interpolation, which can be done by either linear or binary search.

A typical B-tree node consists of a list of keys and pointers to next nodes. However, the MSB of the 64-bit pointer is rarely used in practice, because the memory address in commodity operating systems is less than 2^{64-1} bytes = 2 Exabytes. Therefore, we can use the MSB of the pointer to flag if the node is interpolation-friendly. To use the pointer, we need to mask the MSB of the pointer value before using it. Figure 2a shows the layout of an IFB-tree node. Note that if IFB-tree is used as a primary index, the data tables are sorted too, hence interpolation can be also done on the leaf node to predict the position of the result in physical table pages. Moreover, marking an IFB-tree is a lightweight process compared to B-tree build time, but it can be easily parallelized as well.

2.4 Complexity analysis

The complexity of IFB-tree is the same as the B-tree for all operations. Whenever an IFB-tree node is created or modified, the interpolation error must be re-computed for all values in the



(a) Interpolation in IFB-tree

(b) Overall layout of IFB-tree. IF nodes are marked in red.

Figure 2: Interpolation in IFB-tree

node. For a node with node size of p, the time taken to find the maximum interpolation error for all values in the node is $O(p)$. In the following, we analyze the time taken to build and update the IFB-tree and show that none of the theoretical complexities are different than that of B-tree.

Build time. . Bulk-loading data in a B-tree takes $O(n \log_p n)$ time. In IBF-trees, there is a second phase for evaluating the nodes and marking interpolation-friendly nodes, which takes $O(n)$. The complexity of bulk insertion thus is $O(n \log_p n + n) = O(n \log_p n)$.

Update time. For insertion and deletion, the complexity of the B-tree is $O(p \log_p n)$, which consists of $O(\log_p n)$ access/modification of nodes times $O(p)$ time for modifying each node. When using IFB-trees, we simply re-evaluate the interpolation-friendliness of each node when an element in the node is modified. The complexity of accessing a node thus still is $O(p + p) = O(p)$, leaving the overall $O(p \log_p n)$ complexity unchanged for update operations.

Lookup time. Finding the position corresponding to the query in each node takes $O(t)$ time if the node is interpolation-friendly (t=interpolation diameter), and $O(p)$ otherwise. Depending on how many of the nodes are interpolation-friendly. As $t < p$, the lookup time is between $O(t \log_p n)$ if all nodes are interpolation-friendly and $O(p \log_p n)$ in the worst case.

3 EVALUATION

In this section we compare the performance of the IFB-tree for both synthetic and real-world data to analyze its lookup time and the effectiveness of interpolation.

Experimental Setup. IFB-tree and B-tree are implemented in C++ and compiled with gcc (7.3.0). Note that all data resides in main memory. The range index finds the first the first indexed key that is equal or higher than the lookup key. Also, the keys on the physical layout are sorted (i.e. it is a clustered index), so that the entire result set can be returned once the first key is found.

Datasets. We used three datasets for performance evaluation, namely accessLog, lognormal, and longitudes. The accesslog data contains web-server log records from a hotel booking website. Lognormal is a synthetic data generated from lognormal distribution. Longitudes are sampled without replacement from the longitudes of over 1.5 billion locations extracted from the OpenStreetMap database [17].

Implementation details. For both B-tree and IFB-tree, we used 64-bit keys and 64-bit payload. Scanning in each node can be done by either linear or binary search. When the baseline methods use linear search, the performance improvements of IFB-trees are expected to be higher and more predictable, because shrinking the scan area will linearly reduce the lookup time in nodes. However, it is generally reported that binary search is more effective on new hardware, even for medium-sized nodes [2, 7]. We consequently use binary search for both IFB-tree and the baseline B-tree methods.

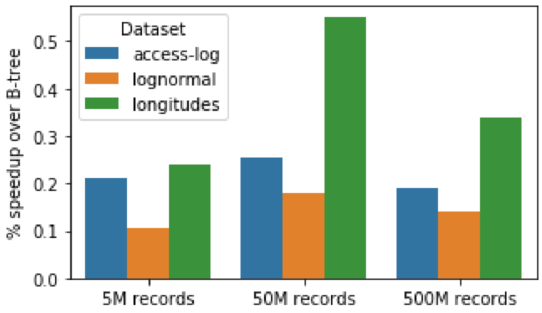


Figure 3: Speedup gained by interpolation (IFB-tree)

Speedup. Figure 3 shows the speedup obtained by the IFB-tree over the B-tree on different datasets with 5M and 500M keys. After tuning the parameters for both indexes (i.e., page size and interpolation diameter), the IFB-tree improves the performance by 10% to 55%.

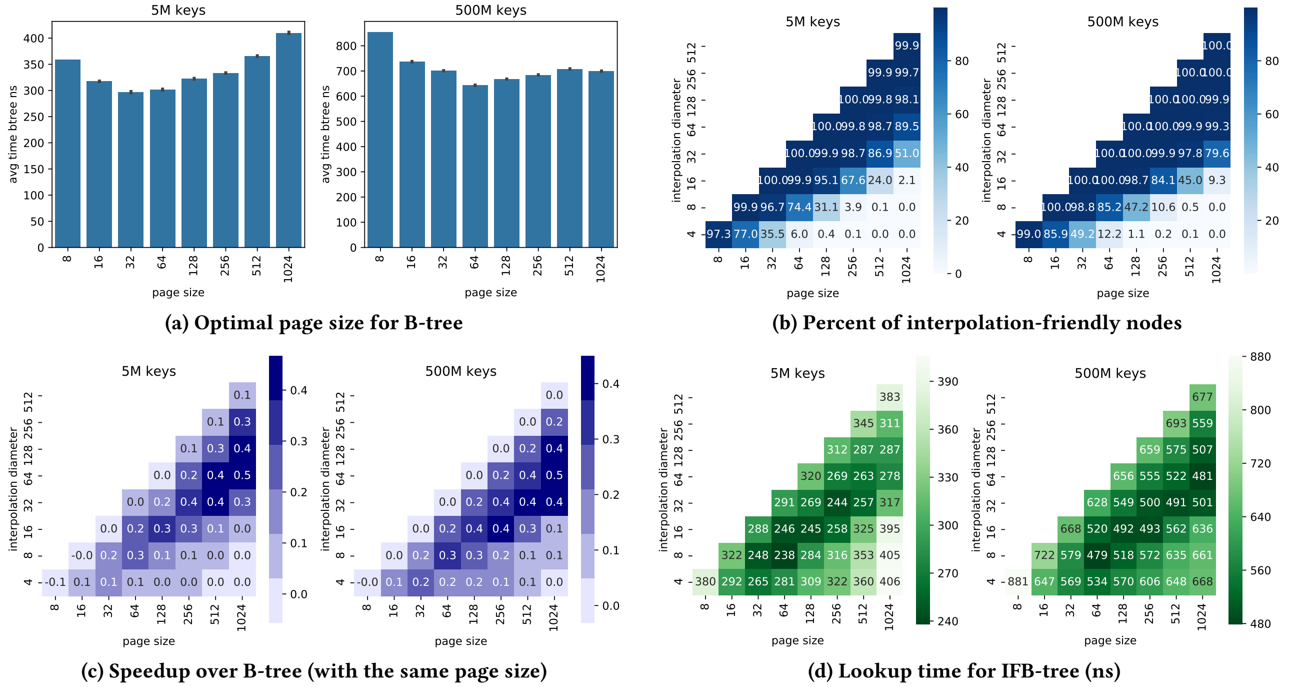
Effect of page size. The key parameter in a B-tree is the page size (= node size). Figure 4a shows how the B-tree lookup time is affected by the page size. Choosing a large page size decreases the branching factor and the tree depth, but searching through each nodes takes more time.

Speedup analysis. The performance speedup of the IFB-tree depends of B-tree nodes that are interpolation-friendly, as well as the interpolation diameter (the area that should be searched within each page). As shown in Figure 4b, the number of interpolation-friendly nodes is almost proportional to $\frac{\Delta}{\text{page size}}$, i.e., to interpolate the majority of nodes on large pages, we need a larger error window. However, a larger error window also decreases the speedup gained by interpolation. Figure 4c shows the speedup of IFB-tree against a B-tree with the same page size, suggesting that the best error window is obtained when $\Delta = \frac{1}{4} \times$ page size. The lookup times (in nanoseconds) are depicted in Figure 4d.

4 RELATED WORK

Interpolation search. Interpolation is an alternative to index structures for estimating the location of records in a clustered index. The technique is known to be very effective when the underlying data distribution is close to uniform [4].

Learning database engine. The use of machine learning on learned indices is just recently proposed. Kraska et al. suggested a learned index based on a hierarchy of deep models, called the *Recursive Model Index* (RMI) [9]. Further research is done on linear learned indexes [3], inverted indexes [18], and enhanced learned bloom filters [15, 18]. Moreover, some theoretical analysis is done on the maximum *capacity* of deep learning models for



(a) Optimal page size for B-tree

(b) Percent of interpolation-friendly nodes

(c) Speedup over B-tree (with the same page size)

(d) Lookup time for IFB-tree (ns)

Figure 4: IFB-tree analysis on Longitudes dataset

keeping index key information [20]. Machine learning has been also adopted for other database operations [6, 8, 10, 13, 13, 16, 19].

Data transformation. Following a different line of work, it is suggested that the performance of index structures can be greatly improved by pre-processing the input data and transforming the keys using a mapping function, in a way that the index structure be more efficient for indexing the distribution of the transformed keys. This idea is exploited for bloom filters [14] and multi-dimensional data [1]. Interestingly, a data transformation that makes the key distribution closer to uniform can benefit IFB-tree and further decreases the interpolation error, hence boosting the speedup of an IFB-tree against its B-tree equivalent.

B-tree enhancement techniques. Several suggestions have been made to design B-trees efficient for the modern hardware. For example, Levandoski et al. suggested BW-trees, a latch-free and cache-friendly B-tree that is used in several Microsoft data engines [12]. Optimizing B-trees can involve compression techniques such as prefix-based splitting. Leis et al. suggested Adaptive Radix Trees, which consumes less memory especially on top levels of the tree. Since all tree-based data structures follow the basic principles of B-trees, such as maintain data in sorted order and splitting the distribution using hierarchies of the table, IFB-tree can be customized for these variants, too.

5 CONCLUSION AND FUTURE WORK

We suggest that the ideas behind learned indexes can be integrated with classical index structures to make them aware of the distribution, hence boosting the performance of the current algorithm. By adopting a computationally lightweight method like interpolation, we boosted the performance of B-trees by up to 50% without modifying the overall structure of B-trees or deteriorating their theoretical performance guarantees. The intra-node interpolation idea is indeed independent from the layout of the tree and hence can be integrated in different extensions to B-tree, including B+-trees, BW-trees [12], and radix trees [11].

REFERENCES

[1] Anonymous. . Spreading vectors for similarity search. In *ICLR 2019 Submission.*

[2] Performance comparison: linear vs binary search. https://dirtyhandscoding.github.io/posts/performance-comparison-linear-search-vs-binary-search.html.

[3] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2018. A-tree: A bounded approximate index structure. *arXiv preprint arXiv:1801.10207* (2018).

[4] Goetz Graefe. 2006. B-tree indexes, interpolation search, and skew. In *DaMoN.*

[5] Goetz Graefe et al. 2011. Modern B-tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402.

[6] Milad Hashemi, Kevin Swersky, Jamie A Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning Memory Access Patterns. *arXiv preprint arXiv:1803.02329* (2018).

[7] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. 2014. Improving in-memory database index performance with Intel Transactional Synchronization Extensions. In *HPCA.* 476–487.

[8] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. *arXiv preprint arXiv:1809.00677* (2018).

[9] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *SIGMOD.* 489–504.

[10] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *arXiv preprint arXiv:1808.03196* (2018).

[11] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE.* 38–49.

[12] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE.* 302–313.

[13] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. *arXiv preprint arXiv:1803.00055* (2018).

[14] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Related Structures. *arXiv preprint arXiv:1802.00884* (2018).

[15] Michael Mitzenmacher. 2018. Optimizing Learned Bloom Filters by Sandwiching. In *NIPS.*

[16] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. 2018. DeepCache: A Deep Learning Based Framework For Content Caching. In *NetAI.* 48–53.

[17] OpenStreetMap on AWS. https://aws.amazon.com/public-datasets/osm.

[18] Harrie Oosterhuis, J. Shane Culpepper, and Maarten de Rijke. 2018. The Potential of Learned Index Structures for Index Compression. In *ADCS.*

[19] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. (2018).

[20] Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, and Hervé Jégou. 2018. Deja Vu: an empirical evaluation of the memorization properties of ConvNets. *arXiv preprint arXiv:1809.06396* (2018).

Short Paper

SynthEdit: Format transformations by example using edit operations

Alex Bogatu, Alvaro A.A. Fernandes, Norman W. Paton, Nikolaos Konstantinou
School of Computer Science, University of Manchester
alex.bogatu@manchester.ac.uk

ABSTRACT

Format transformation is one of the most labor intensive tasks of a data wrangling process. Recent advances in programming by example proposed synthesis algorithms that showed promising results on spreadsheet data. However, when employed on repositories consisting of multiple sources and large number of examples, such algorithms manifest scalability issues. This paper introduces a new transformation synthesis technique based on edit operations that enables efficient learning of transformation programs. Empirical results show comparable effectiveness and dramatic improvements in efficiency over the state-of-the art.

1 INTRODUCTION

Format transformation is a sub-task of data wrangling ([3], [8]) that carries out changes to the representation of textual information, with a view to reducing inconsistencies. Such tasks are typically coded manually by experienced users through scripts that change the representation of data. Recent advances in Programming By Example (PBE) led to algorithms such as *FlashFill* [4] and *BlinkFill* [11] that synthesize transformation programs from user given input-output examples. These algorithms represent important steps towards automatic format transformation in the fields for which they have been devised: spreadsheets. Spreadsheet processing often involves datasets of manageable size, a small number of examples and active user involvement in providing additional example data when needed. In contrast, in areas of data wrangling and data analysis, the task of format transformation is applied repetitively on large datasets from many sources, where more examples are available and user supervision is impractical [2]. Increased volume of example data reveals the opportunity for taking the human factor out of the synthesis process. Automating format transformation by increasing the number of examples represents a challenge for current synthesis algorithms due to their high complexity: exponential in the number of examples and highly polynomial in the length of the examples [10].

In this paper we contribute a solution, which we call *SynthEdit*, to the problem of automatic format transformation. We are motivated by the high computational cost of current state-of-the-art techniques when the number of available examples increases, and propose an approach based on finite state automata that scales better than algorithms such as *FlashFill* when there are more than a few examples available - typically tens or hundreds. The improved efficiency comes at the expense of expressiveness, but allows *SynthEdit* to benefit from substantial example data when available, and, therefore, to cover many inconsistency cases and to eliminate the need for user involvement.

Related work: The problem of learning transformations from examples has been addressed in a number of settings ranging from restructuring of textual information [4], [11], to table formatting [6], and text extraction [1]. More complex systems such as TDE [5] embody a wider range of transformations using external information in addition to user-provided examples.

The closest to our work are algorithms such as FlashFill which have specifically been designed for spreadsheets, assume high user involvement and use DAG-based synthesis solutions that aim to cover the entire space of possible transformations between two given strings. The user's main task in such systems is to provide additional examples for cases not covered by the previous instances. The resulting algorithms have exponential complexity w.r.t. the number of examples [10].

While recent works [2], [7] have proposed techniques that minimize the user involvement in generating examples, *SynthEdit* aims to fully automate the task of format transformation for larger repositories such as data lakes, richer in potentially useful example data, and challenging to address through manual authoring of scripts due to their size and diversity.

2 PRELIMINARIES

We start by describing a representative example where our algorithm can be employed, inspired by web real world data.

Example 2.1. 1900s NY state governor names and term years:

Source	Target
Hugh Leo Carey (74-82)	Hugh L. Carey (1974-1982)
Jay Henry Lehman (33-42)	Jay H. Lehman (1933-1942)

In Example 2.1, the task is to derive a transformation that can generate the *Target* values using the *Source* values. In this paper we refer to such pairs of strings as *example instances.*

Tokens: We view a string s as a collection of *tokens*, where each token represents a sub-string of s. Similarly to existing algorithms for format transformation, e.g., *FlashFill*, *SynthEdit* supports three types of tokens: (i) regular expression tokens - that match a predefined regular expression pattern; (ii) constant string tokens - with a value equal to the corresponding constant string, and (iii) special tokens - beginning/end of a string.

Regular expression primitives: Obtaining the set of tokens for a string s builds on a set of primitive lexical classes defined by regular expressions. The regular expression primitives we use are notated as follows: N = [0-9]+, U = [A-Z]+, L = [a-z]+, A = [A-Za-z]+, Q = [A-Za-z0-9]+, P = [.,;: /-_?!&$]+, W = \\$s$+.

Transformations: Going back to Example 2.1, we can express the source string in the first row as a collection of token types, henceforth called a *token-type representation*: A W A W A W P N P N P. Such representations will be used to derive a transformation that will produce the target string from the source string. One simple such transformation would have the following English language specification: *Replace "Leo", "74", and "82" from the source with "L", "1974", and "1982" from the target, resp.*. Although such

Series ISSN: 2367-2005

10.5441/002/edbt.2019.94

$$
\begin{aligned}
\mathit{Regex\ primitive}\ \mathtt{r} &:= \mathtt{N} \mid \mathtt{U} \mid \mathtt{L} \mid \mathtt{A} \mid \mathtt{Q} \mid \mathtt{P} \mid \mathtt{W} \\
\mathit{Position\ expression}\ \mathcal{P} &:= \mathit{Pos}(\mathtt{r}_1, \mathtt{r}_2, c) \\
\mathit{Token}\ \mathtt{t} &:= (\mathtt{r}, \mathcal{P}) \\
\mathit{String\ expression}\ \mathcal{E} &:= \mathit{Copy}(\mathtt{t}) \mid \mathit{Const}(\mathtt{s}) \\
&\quad \mid \mathit{Substr}(\mathtt{t}, i, j) \mid \mathit{Concat}(\mathcal{E}_1, \ldots, \mathcal{E}_n) \\
\mathit{Edit\ operation}\ O &:= \mathit{INS}(\mathcal{E}) \mid \mathit{DEL}(\mathtt{t}) \mid \mathit{SUB}(\mathtt{t}, \mathcal{E}) \\
\mathit{Transformation}\ \mathcal{T} &:= O_1;\ O_2;\ \ldots;\ O_n;
\end{aligned}
$$

Figure 1: Transformation language syntax

a transformation is consistent with the first example instance, it is too specific to correctly transform the string in the second row. Fortunately, we can generalize the transformation by using token types instead of actual sub-string values: *Replace the second* A*-type token, the first* N*-type token, and the second* N*-type token from the source with the first* U*-type token, the first* N*-type token, and the second* N*-type token from the target, resp.*. Now, the same transformation is consistent with both example instances.

3 TRANSFORMATION LANGUAGE

In this section we propose a transformation language that is expressive enough to describe the previous transformation using simple edit operations: *Insert* (referred to as *INS*), *Delete* (referred to as *DEL*), and *Substitute* (referred to as *SUB*). The complete syntax of the transformation language is in Figure 1, while the semantics are described below. We use the notation ϵ to denote an empty string, $len(s)$ for the length of a string s, $s[i : j]$ for the sub-string that starts at index i in s and ends at index j, and c to denote the c^{th} occurrence of a token type in a string[1].

The semantics of a position expression, $Pos(\mathtt{r}_1, \mathtt{r}_2, c)$, is to evaluate to a sub-string $s[j : k]$ of a given string s, such that $\exists\, i, j, k, l\ 0 \leq i < j < k < l \leq len(s) - 1$, where $s[i : j]$ matches $\mathtt{r}_1$ and $s[k : l]$ matches $\mathtt{r}_2$. Furthermore, $s[j : k]$ is the c^{th} such sub-string in s. If such a sub-string does not exist then the expression evaluates to ϵ. Such an expression allows us to uniquely identify each token in a given string using its neighbour tokens and, at the same time, ensures that the expression will (likely) evaluate to ϵ when applied on strings with different format representations. We can now redefine a token $\mathtt{t}$ as a pair consisting of a token type $\mathtt{r}$ and a position expression $\mathcal{P}$. Note that $\mathtt{t}$ only exists if the string value returned by $\mathcal{P}$ matches $\mathtt{r}$.

The *Copy*($\mathtt{t}$) expression evaluates to the string value of $\mathtt{t}$. *Const*($\mathtt{s}$) evaluates to a constant string s. $Substr(\mathtt{t}, i, j)$ returns the sub-string that starts at position i and ends at position $j - 1$ of the string value of $\mathtt{t}$. $Concat(\mathcal{E}_1, \ldots, \mathcal{E}_n)$ performs string concatenation on the results of the underlying string expressions $\mathcal{E}_1, \ldots, \mathcal{E}_n$. The edit operations $INS(\mathcal{E})$, $DEL(\mathtt{t})$, and $SUB(\mathtt{t}, \mathcal{E})$ perform insertion, removal, and replacement, resp. of some string resulting from the expressions given as parameters, on a given string s. Lastly, $\mathcal{T}$ is a sequence of edit operations applied on s.

Figure 2 shows 5 operations (from a total of 12) of a transformation that can be used to edit both source strings from Example 2.1 into their corresponding target strings. Considering the first row from Example 2.1, the intuition in (1) and (2) is to replace the first occurrences of an A-type token, i.e., *Hugh*, and the first white space with a copy of themselves. Similarly, in (3), the token *Leo* from the source string is replaced with the first letter of itself. In (4), a dot, ".", is inserted after the result of operation (3).

$$
\begin{aligned}
&SUB((\mathtt{A}, Pos(\hat{}\,, \mathtt{W}, 0)), Copy((\mathtt{A}, Pos(\hat{}\,, \mathtt{W}, 0)))); && (1)\\
&SUB((\mathtt{W}, Pos(\mathtt{A}, \mathtt{A}, 0)), Copy((\mathtt{W}, Pos(\mathtt{A}, \mathtt{A}, 0)))); && (2)\\
&SUB((\mathtt{A}, Pos(\mathtt{W}, \mathtt{W}, 0)), Substr((\mathtt{A}, Pos(\mathtt{W}, \mathtt{W}, 0)), 0, 1)); && (3)\\
&INS(Const(".")); && (4)\\
&SUB((\mathtt{N}, Pos(\mathtt{P}, \mathtt{P}, 0)), Concat(Const("19"), Copy((\mathtt{N}, Pos(\mathtt{P}, \mathtt{P}, 0))))); && (5)
\end{aligned}
$$

Figure 2: Transformation for Example 2.1

Operation (5) replaces the first number between two punctuation tokens, i.e., *74,* with the result of the concatenation of a constant string, *19*, and the same number to obtain *1974*. Note that the transformation in Figure 2 is applied on a copy of the source, as opposed to modifying the string in-place. This ensures that source tokens used by later operations are not lost if an early operation deletes/substitutes them.

4 SYNTHESIS ALGORITHM

In this section we describe an algorithm that, starting from the example instances provided in Example 2.1, learns the transformation from Figure 2. Given an example instance, the algorithm consists of 4 phases: 1) tokenize the source and target strings; 2) synthesize edit operations; 3) synthesize string expressions; 4) merge the results of 2) and 3) to create a transformation.

Tokenization: Both *source* and *target* strings are split into tokens using a function `Tokenize`. Specifically, the function searches for sub-strings that match one of the regular expression primitives defined in Section 2. Each such match represents a new token t_i. Next, by looking at the neighbouring matches, the function learns a position expression which uniquely identifies t_i in the parent string. Tokenizing the source string of the first row in Example 2.1 returns: $(\mathtt{A}, Pos(\hat{}\,, \mathtt{W}, \mathtt{0}))$, $(\mathtt{W}, Pos(\mathtt{A}, \mathtt{A}, 0))$, $(\mathtt{A}, Pos(\mathtt{W}, \mathtt{W}, 0))$, $(\mathtt{W}, Pos(\mathtt{A}, \mathtt{A}, 1))$, $(\mathtt{A}, Pos(\mathtt{W}, \mathtt{W}, 1))$, $(\mathtt{W}, Pos(\mathtt{A}, \mathtt{P}, 0))$, $(\mathtt{P}, Pos(\mathtt{W}, \mathtt{N}, 0))$, $(\mathtt{N}, Pos(\mathtt{P}, \mathtt{P}, 0))$, $(\mathtt{P}, Pos(\mathtt{N}, \mathtt{N}, 0))$, $(\mathtt{N}, Pos(\mathtt{P}, \mathtt{P}, 1))$, $(\mathtt{P}, Pos(\mathtt{N}, \$, 0))$

Edit operation synthesis: The top-level transformation is a collection of edit operations parameterized with tokens and/or string expressions. Given an example instance, with its corresponding token-type representation of source (T_s) and target (T_t) derived from the token sets obtained at the previous step, a function `EditSynthesis`(T_s, T_t) generates a sequence of edit operations that edits T_s into T_t. This can be done using an edit distance algorithm, such as the one proposed in [9], where the composition operation on weighted finite state automata (WFSA) is used to generate all possible edit operations that transform the source into the target. By assigning equal weights to each edit operation, the shortest path of the lattice of operations returned by the composition of WFSA denotes the simplest way to obtain the token-type representation of the target. Note that each path of the obtained lattice denotes a valid sequence of edit operations which would produce the target string, but we are only interested in the simplest one. As an example, consider again the first row from Example 2.1 with the token-type representations $T_s = \mathtt{A\ W\ A\ W\ A\ W\ P\ N\ P\ N\ P}$ and $T_t = \mathtt{A\ W\ U\ P\ W\ A\ W\ P\ N\ P\ N\ P}$. `EditSynthesis`$(T_s, T_t)$ can produce the following sequence of edit operations[2] that transform T_s into T_t:

$$
\begin{aligned}
&SUB(\mathtt{A}^{\mathtt{s}}_{\mathtt{0}}, \mathtt{A}^{\mathtt{t}}_{\mathtt{0}});\ SUB(\mathtt{W}^{\mathtt{s}}_{\mathtt{0}}, \mathtt{W}^{\mathtt{t}}_{\mathtt{0}});\ SUB(\mathtt{A}^{\mathtt{s}}_{1}, \mathtt{U}^{\mathtt{t}}_{\mathtt{0}});\ INS(\mathtt{P}^{\mathtt{t}}_{\mathtt{0}});\\
&SUB(\mathtt{W}^{\mathtt{s}}_{1}, \mathtt{W}^{\mathtt{t}}_{1});\ SUB(\mathtt{A}^{\mathtt{s}}_{2}, \mathtt{A}^{\mathtt{t}}_{1});\ SUB(\mathtt{W}^{\mathtt{s}}_{2}, \mathtt{W}^{\mathtt{t}}_{2});\ SUB(\mathtt{P}^{\mathtt{s}}_{\mathtt{0}}, \mathtt{P}^{\mathtt{t}}_{1}); \qquad (6)\\
&SUB(\mathtt{N}^{\mathtt{s}}_{\mathtt{0}}, \mathtt{N}^{\mathtt{t}}_{\mathtt{0}});\ SUB(\mathtt{P}^{\mathtt{s}}_{1}, \mathtt{P}^{\mathtt{t}}_{2});\ SUB(\mathtt{N}^{\mathtt{s}}_{1}, \mathtt{N}^{\mathtt{t}}_{1});\ SUB(\mathtt{P}^{\mathtt{s}}_{2}, \mathtt{P}^{\mathtt{t}}_{3});
\end{aligned}
$$

$\mathtt{A}^{\mathtt{s}}_{\mathtt{i}}/\mathtt{A}^{\mathtt{t}}_{\mathtt{j}}$: the i^{th}/j^{th} token of type A from source/target, $i, j \geq 0$.

Note that Expression (6) denotes the transformation between a given source and a given target, but it cannot be used directly

[1] All string positions, i.e. indexes, start from 0.

[2] Expression (6) omits the *Pos* constructors for clarity.

Algorithm 1 String expression synthesis

```
Input: Index entry: $t_i \rightarrow pairs_{t_i}$
Output: A string expression $\mathcal{E}$
 1: function ExpressionSynthesis
 2:     if $pairs_{t_i}$ == [] then return $Const(t_i)$ end if
 3:     $(s_j, lcs_j) \leftarrow$ best_pair(pairs$_{t_i}$)
 4:     if $s_j == lcs_j == t_i$ then $\mathcal{E} \leftarrow Copy(s_j)$
 5:     else if $lcs_j \subset s_j$ && $lcs_j == t_i$ then
 6:         $\mathcal{E} \leftarrow Substr(s_j$, indexOf$(lcs_j, s_j))$
 7:     else
 8:         $\mathcal{E} \leftarrow Concat($ConcatSynthesis$(t_i, pairs_{t_i}))$
 9:     end if
10:     return $\mathcal{E}$
11: end function
```

Algorithm 2 *Concat* expression synthesis

```
Input: Index entry: $t_i \rightarrow pairs_{t_i}$
Output: A list of string expression $exp$
 1: function ConcatSynthesis
 2:     $exp \leftarrow [\,]$
 3:     for all $(s_j, lcs_j) \in pairs_{t_i}$ do
 4:         if $s_j == lcs_j$ then $exp \leftarrow exp + [Copy(s_j)]$
 5:         else $exp \leftarrow exp + [Substr(s_j$, indexOf$(lcs_j, s_j))]$
 6:         end if
 7:         $t_i \leftarrow$ replace$(t_i, lcs_j$, "")
 8:     end for
 9:     if len$(t_i) > 0$ then $exp \leftarrow exp + [Const(t_i)]$ end if
10:     return $exp$
11: end function
```

when the target is not given. As such, our next objective is to express each target token in Expression (6) as a string expression applied on some source token, e.g., Figure 2. This will allow us to apply the newly obtained operations on new input strings.

String expression synthesis: Expressing a target token t_i as a processing of some source tokens requires the identification of the source token (or the group of source tokens) whose value(s) are the closest to t_i. To this end, we create an inverted index $\mathcal{I}$ in which each target token value t_i identifies a list of pairs of the form (s_j, lcs_j), where lcs_j is the *longest common sub-string* between a source token value s_j and t_i. The first two columns from Table 1 depict three index entries obtained for the first row of Example 2.1. The third column indicates the type of string expressions that can be applied on the source tokens to obtain t_i, as described next.

For each entry of I, we can apply a function `StringSynthesis` (defined in Algorithm 1) to synthesize a string expression that uses some source tokens to obtain the target token. The function at line 3 in Algorithm 1 returns a pair (s_j, lcs_j) where s_j is the source token value that best covers the target token value. Note that we only process the best such pair because its source token has the most useful value to derive the target token. For when the best pair is not the desired one, i.e. the target token has to be obtained from a different source token, we rely on multiple example instances to disambiguate and to generalize a transformation. The `indexOf`(lcs_j, s_j) function at line 6 returns the start and end indexes of lcs_j in s_j.

When there is no source token that can be used to obtain t_i, `ExpressionSynthesis` returns a *Const* expression. Alternatively, when the longest common sub-string, the source and target token values are all identical, the result is a *Copy* expression, e.g., Row 1 in Table 1. If the longest common sub-string is only equal to the target token value, it means that t_i can be obtained from the source token using a *Substr* expression, e.g., Row 2 in Table 1.

Finally, a *Concat* expression is synthesized, using Algorithm 2, when the source token value is a sub-string of the target token value. The `ConcatSynthesis` function processes all pairs of the current index entry, as opposed to only the best pair, and consumes the target token value as soon as it is able to derive a sub-string of it. For example, for Row 3 in Table 1, *74* is a source token and, therefore, the condition at line 4 in Algorithm 2 is met. At the next iteration t_i = "19" (because we consume the previous value) but there are no pairs in $pairs_{t_i}$ that cover the new value which means that the condition at line 9 evaluates to true and the next expression learned is *Const*.

Transformation synthesis: The last step of the algorithm replaces the target tokens in Expression (6) with the corresponding string expressions learned by Algorithm 1. The result, listed below, is a transformation consistent with the example instance and applicable on new input strings, similar in format representation with the source string of the example instance.

Table 1: Index entries

	t_i	$[(s_j, lcs_{(s_j,t_i)})]$	Expression
1	Hugh	[(Hugh, Hugh)]	*Copy*
2	L	[(Leo, L)]	*Substr*
3	1974	[(74, 74)]	*Concat*

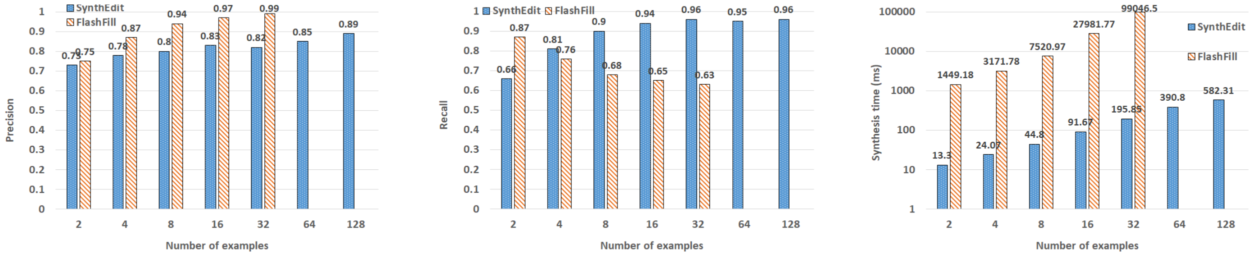
$SUB(\mathtt{A}^s_0, Copy(\mathtt{A}^s_0))$; $SUB(\mathtt{W}^s_0, Copy(\mathtt{W}^s_0))$; $SUB(\mathtt{A}^s_1, Substr(\mathtt{A}^s_1, 0, 1))$;
$INS(Const(".")); SUB(\mathtt{W}^s_1, Copy(\mathtt{W}^s_0))$; $SUB(\mathtt{A}^s_2, Copy(\mathtt{A}^s_2))$;
$SUB(\mathtt{W}^s_2, Copy(\mathtt{W}^s_0))$; $SUB(\mathtt{P}^s_0, Copy(\mathtt{P}^s_0))$;
$SUB(\mathtt{N}^s_0, Concat(Const("19"), Copy(\mathtt{N}^s_0)))$; $SUB(\mathtt{P}^s_1, Copy(\mathtt{P}^s_1))$;
$SUB(\mathtt{N}^s_1, Concat(Const("19"), Copy(\mathtt{N}^s_1)))$; $SUB(\mathtt{P}^s_2, Copy(\mathtt{P}^s_2))$;

Learning from multiple examples: *SynthEdit* assumes the existence of several example instances from which it can learn multiple transformations. Although it is possible for a valid transformation to be synthesized from a small sample of the examples, it is not always possible to automatically identify the relevant sample and, therefore, all examples have to considered. Synthesizing transformations from multiple example instances that follow more than one format representation results in a *conditional transformation program.* To create such a program, we first partition the example instances into groups with source strings that follow the same format representation and synthesize a transformation for each partition[3]. Before transforming a new input string, we match its format representation against the ones for which we have synthesized transformations and apply the transformation of the matching format. If such a format representation does not exist the input string is left unchanged.

Complexity: There are two dominant tasks in *SynthEdit* in terms of complexity. Firstly, *EditSynthesis* runs in $O(m \times n)$ time [9], where m is the length of the source string and n is the length of the target string. Secondly, the generation of inverted index I implies the identification of the longest common sub-string between two token string values which is a dynamic programming problem that runs in $O(k \times l \times u \times v)$, where k is the number of source tokens, l is the number of target tokens, u is the source token value length, and v is the target token value length.

Synthesizing string expressions using `Expression Synthesis` only for the simplest token-based transformation, previously obtained using `EditSynthesis`, gives *SynthEdit* a computational

[3] If more than one transformation is possible per partition, we pick the one consistent with the majority of the example instances of that partition



(a) Average precision

(b) Average recall

(c) Average synthesis time

Figure 3: Experimental Results

advantage over algorithms such as *FlashFill* which considers all possible transformations that are consistent with the examples before ranking them and choosing the best one.

5 EVALUATION

In this section we perform a comparative evaluation[4] involving *SynthEdit* and an implementation of *FlashFill* from PROSE SDK[5], using 33 real world datasets[6], used in a related work [12]. Each dataset consists of up to 200 example instances from several domains such as person names, websites, songs, etc.

We report the average precision, recall and synthesis time over all datasets computed using k-fold cross-validation ($k = 10$) and various number of examples. At each iteration (fold), we synthesize a transformation program from n randomly picked example instances and test on the remaining instances. For the purposes of computing precision and recall, we count as a *true positive* any input string that is correctly transformed, i.e., the result of the transformation is similar to the expected output; as a *false positive* any input string that is incorrectly transformed; and as a *false negative* any input string that is left unchanged, i.e., there is no transformation synthesized for its format representation.

Comparative effectiveness: Avg. precision and recall as the number of examples varies. The precision results are shown in Figure 3a. *SynthEdit* achieves lower precision compared with *FlashFill* for the different numbers of examples. The difference can be explained by the ability of *FlashFill* to better generalize transformations as more examples are added by using a classifier trained on example instances. This allows it to correctly transform strings with format representations not covered by the examples. Conversely, *SynthEdit* performs a strict mapping between format representations and transformations and, therefore, requires at least one example instance for each format representation it transforms. For the last two cases, i.e., 64 and 128 examples, *FlashFill* required more RAM memory than was available.

The classifier employed by *FlashFill* to pick the right transformation given a new input string can become confused when some examples are too similar to each other in terms of the features used during learning. This means that for some input strings *FlashFill* fails to identify an appropriate transformation or the transformation picked is not consistent with the input, e.g., the transformation expects a type of token that is not present in the input. The consequence is a drop in recall visible in Figure 3b as the number of examples increases. By contrast, *SynthEdit* achieves better recall because more examples enables it to better differentiate between transformation cases.

Comparative efficiency: Avg. synthesis time as the number of examples varies. Figure 3c confirms the high complexity of *FlashFill* when the number of examples increases. Conversely, *SynthEdit* proves more than two orders of magnitude faster in synthesizing transformations. As opposed to *FlashFill*, *SynthEdit* does not aim to exhaust the search space of transformations for each example instance. Our algorithm uses edit distance computations to find the shortest path between the token-type representations of the source and target strings. Consequently, the transformation language is simpler but more efficient to learn.

6 CONCLUSIONS

We have contributed an effective and efficient solution to the problem of automating format transformation given input-output examples. We have used an edit distance based approach that identifies the shortest path from a source string to a target string and uses fuzzy matching of source and target tokens to generalize transformations applicable on new input strings, similar in format representation with the examples. Results from a comparative evaluation provide evidence that *SynthEdit* performs substantially more efficiently than the state-of-the-art while achieving better recall at the cost of slightly reduced precision.

Acknowledgement: This work is supported by the VADA Grant of the UK Engineering and Physical Sciences Research Council.

REFERENCES

[1] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao. 2016. Inference of Regular Expressions for Text Extraction from Examples. *IEEE Trans. Knowl. Data Eng.* 28, 5 (2016), 1217–1230.

[2] A. Bogatu, N. Paton, A. Fernandes, and M. Koehler. 2018. Towards Automatic Data Format Transformations: Data Wrangling at Scale. *Comput. J.* (2018).

[3] T. Furche, G. Gottlob, L. Libkin, G. Orsi, and N. W. Paton. 2016. Data Wrangling for Big Data: Challenges and Opportunities. In *EDBT*. 473–478.

[4] S. Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *POPL '11*. ACM, New York, NY, USA, 317–330.

[5] Y. He, X. Chu, K. Ganjam, Y. Zheng, V. Narasayya, and S. Chaudhuri. 2018. Transform-data-by-example (TDE): An Extensible Search Engine for Data Transformations. *Proc. VLDB Endow.* 11, 10 (June 2018), 1165–1177.

[6] Z. Jin, M. R. Anderson, M. Cafarella, and H. V. Jagadish. 2017. Foofah: Transforming Data By Example. In *SIGMOD*. 683–698.

[7] M. Koehler, A. Bogatu, C. Civili, N. Konstantinou, E. Abel, A. A. A. Fernandes, J. A. Keane, L. Libkin, and N. W. Paton. 2017. Data context informed data wrangling. In *IEEE BigData, Boston, MA, USA, December 11-14, 2017*. 956–963.

[8] N. Konstantinou, M. Koehler, E. Abel, C. Civili, B. Neumayr, E. Sallinger, A. Fernandes, G. Gottlob, J. Keane, L. Libkin, and N. Paton. 2017. The VADA Architecture for Cost-Effective Data Wrangling. In *SIGMOD*. 1599–1602.

[9] M. Mohri. 2002. Edit-Distance of Weighted Automata. In *CIAA, 2002*. 1–23.

[10] M. Raza, S. Gulwani, and N. Milic-Frayling. 2014. Programming by Example Using Least General Generalizations. In *AAAI*. 283–290.

[11] R. Singh. 2016. BlinkFill: Semi-supervised Programming by Example for Syntactic String Transformations. *PVLDB* 9, 10 (June 2016), 816–827.

[12] E. Zhu, Y. He, and S. Chaudhuri. 2017. Auto-join: Joining Tables by Leveraging Transformations. *Proc. VLDB Endow.* 10, 10 (June 2017), 1034–1045.

[4] Experiments were run on a 2.60 GHz Intel Core i7-4720HQ CPU with 8 GB RAM.

[5] https://microsoft.github.io/prose/

[6] www.microsoft.com/en-us/research/wp-content/uploads/2016/12/WebTableBenchmark.zip

Triad Enumeration at Trillion-Scale using a Single Commodity Machine

Yudi Santoso, Venkatesh Srinivasan,
Alex Thomo
University of Victoria
Canada
{santoso,srinivas,thomo}@uvic.ca

Sean Chester
Norwegian University of Science and Technology
Norway
sean.chester@ntnu.no

ABSTRACT

Triad enumeration yields more detailed information than triangle enumeration. However, triad enumeration is more complex as it has to list the edges as well as the nodes of the triads. Furthermore, it is challenging to do on large graphs because of two reasons: how to deal with large amounts of data using limited memory, and how to do the computation in a reasonable amount of time. While distributed computing can take care of both problems, it requires large investment and high operating cost, as well as a distributed algorithm design which is not always possible. In this paper we show that triad enumeration of very large graphs at the web-scale can actually be done on a single commodity machine. Memory space limitation can be overcome by using data compression and partial loading. Performance can be greatly improved through optimized preprocessing and parallelization.

1 INTRODUCTION

Triangles play an important role in network analysis. For example, the presence of triangles is an indicator of communities in the network [13]. Triangles are also central to computing the connectivity of a graph [2], the clustering coefficient [18], and the transitivity [11]. There are many practical applications of these, for example, detecting fake users in social networks [19] and uncovering hidden thematic layers in the Web [9].

Most real world networks have directed relationships, and therefore we should consider directed graphs for better representations of those networks. A triad is a subgraph of three nodes in a directed graph [1, 8]. When each pair of the nodes is connected we have a closely connected triad. Since we are only interested in closely connected triads we will simply call them triads in this paper. There are seven types of triads, called by some authors as seven types of triangles [14, 16]. They are shown in Fig. 1. Note, however, that we define our own numbering here.

Enumerating triads means listing the edges as well as the nodes inside every triad. Triad enumeration would reveal a more detailed picture of the network, and hence opening up more possible applications. For example, transitivity can be more accurately analyzed by using triads [2], and directed clustering coefficient can be used as a measure of systemic risk in complex banking networks [15]. Also, triad enumeration is an important element in social network analysis [17].

Nonetheless, triad enumeration is more complex than triangle enumeration. The general belief is that it is considerably more difficult [14] and that it would take much longer running time. We found that this is not necessarily the case. Although there are some challenges, it is possible to devise an efficient algorithm which, combined with a compression framework such as WebGraph, is able to enumerate triads on a graph with a billion nodes and billions of edges using a single commodity machine.

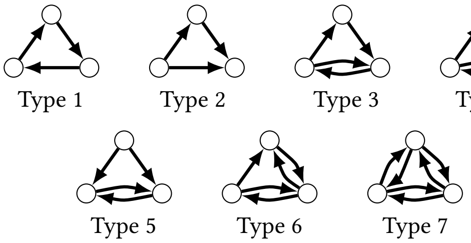


Figure 1: Seven types of triads.

1.1 Related Work

Pajek is a well known graph analysis software for triad enumeration. The triad census algorithm by Batagelj and Mrvar [1], which is employed in Pajek, has been known as the standard algorithm to enumerate triads for several years. However, this program is not suitable for very large graphs with millions of nodes and edges.

Chin et al. [7] proposed a compact data structure where both outgoing and incoming edges are listed in the same adjacency list, and the edge direction is encoded using the 2 lowest bits out of the 32 bits (i.e., using `int`) in each entry. This data structure is suitable for parallel computation using shared memory architectures.

This idea was further refined by Parimalarangan et al. [12], who proposed two types of algorithms, intersection based (AI) and marking based (AM), as the most efficient algorithms to enumerate triads on shared memory platforms.

While those ideas significantly improve the running time, there is a cost on the scalability. With two bits used for edge direction, the order of the graphs that can be processed is reduced. This becomes a problem when we want to analyse a graph of a billion nodes. Theoretically, we can switch to 64-bit integers and the problem should be moot. Practically, however, this would at least double the memory requirement. With limited memory budget, space is already a problem for analysing very large graphs.

1.2 Contributions

1. We implemented Parimalarangan et. al. AI algorithm in Java. We chose AI over AM because it uses less memory. AM is unable to work in a consumer-grade machine of 32GB memory even for graphs of moderate size. Our code is designed for execution on a single machine with parallel threads. Together with optimized preprocessing on the input graphs, we were able to run triad enumeration faster than the ones reported in Parimalarangan's paper, hence raising the known record.

2. For enormous graphs, such as ClueWeb12, AI is not able to work in a consumer-grade machine of 32GB memory. In order to address the case of such graphs we propose another algorithm, which has better scalability. It uses both the graph and transpose graph as input, and computes the node connections on the fly.
3. We are able to employ WebGraph compression which archives a more than 7-fold compression ratio and thus allows loading big graphs (or significant parts of them) in main memory. This enables our new algorithm to complete triad enumeration on ClueWeb12 in the aforementioned machine.

2 TRIAD ENUMERATION

The Batagelj and Mrvar triad census algorithm [1] assigns a code to each pair of nodes to represent the directed edges between them. For each triple of nodes it then uses a table to find the triad types based on the combined codes. Although this algorithm can do triad enumeration in subquadratic time, it is not fast enough for very large graphs with millions of nodes and edges.

Chin et al. [7] developed a compact data structure which makes it easier to parallelize the computation. They combined the adjacency list to contain both outgoing and incoming edges. The edge information or the link is coded using 2-bits: 01 (forward), 10 (backward), and 11 (both), embedded in the neighbour node labels inside the list. Suppose the nodes were labeled by using 32-bit integers. The bits are shifted to the left by two, and the two lowest bits are then used for the edge direction. Thus, only 30 bits can actually be used to label the nodes.

Parimalarangan et al. [12] took on this idea and combined it with the most efficient algorithms known for triangle enumeration on single machines [10]. They came up with two algorithms, the AI algorithm which is intersection based, and the AM algorithm which is marking based. Although AM can in some cases be faster it requires more memory than AI. Therefore, we chose AI and implemented it in Java for our base comparison. We use parallel streams introduced in Java 8 to make use of the multiple threads in our machine.

In our experiments, we use directed networks and their transpose in compressed WebGraph format. For our implementation of AI, we first build the compact data structure (of edges and their direction 2-bit-encoding) from these datasets and save it in plain text format, which is then used as input to the AI program.

The WebGraph framework [5, 6] provides compression schemes suitable for graph adjacency lists. The compression factor can be more than seven-fold, significant in saving disk space. The framework also provides some tools to work with files in WebGraph format. One of them is the `loadMapped`, which allows partial loading of the dataset using memory-mapped files. As we will see later, this is one of the key tools that enable us to process very large graph such as ClueWeb12. Note, however, that using WebGraph comes with the cost of decompressing the dataset.

To improve the performance, we did some preprocessing on the graphs (before building the compact data structure). First, note that each triad should be iterated only once. To avoid multiple counting, we assert condition $u < v < w$ for a triad (u, v, w). Consequently, we only need to consider bigger neighbours. Also, permutation on the labels should not change the number of triads in a graph. Thus, we first sort the nodes according to degrees from lowest to highest, relabel them, and then cut out smaller neighbours from the adjacency list. That is, suppose N_u is the set of neighbours of u after relabelling, then after the cut the adjacency list only contains $N_u \setminus \{v | v < u\}$. This is done simultaneously for the graph and its transpose as the relabelling must be the same for both. After this preprocessing, a node which originally has the highest degree would have zero/no neighbours. The degree distribution in the new adjacency list would have a hill shape with highest (effective) degrees gathered in the middle. When we parallelize on the first node iteration, this distribution can lead to imbalanced workload among the threads. To alleviate this, we do a further preprocessing which redistributes the nodes in steps of 10,000. That is, we pick the nodes and rearrange them in order (0, 10000, 20000, ..., 1, 10001, 20001, ...), and then relabel them as (0, 1, 2, ...).

Algorithm 1 Four Pointers Triad Enumeration

```
Input: A directed graph $G = (V, E)$ and its transpose $G^T$
Output: The number of each type of triads in $G$, $\Delta_i$.
$\Delta_1 \leftarrow 0, \ldots, \Delta_7 \leftarrow 0$
for all $u \in V$ do                                    ▷ Parallelize
    while there is next do
        Find next neighbour in $N^+(u)$ and/or $N^-(u)$: $v$.
        Code the link $uv$ as $e1$: either 01, 10 or 11
        while there is next do
            Find next common neighbour of $u$ and $v$: $w$, in $(N^+(u), N^-(u))$ and $(N^+(v), N^-(v))$.
            Code the links $vw$ as $e2$, and $wu$ as $e3$.
            Look up triad type $i$ using $e1, e2, e3$.
            enum(u,v,w,e1,e2,e3)
            $\Delta_i \leftarrow \Delta_i + 1$
return $\Delta_1, \ldots, \Delta_7$
```

The drawback of the compact data structure solution is that it leads to a reduced scalability. In Java, the 32-bit integer data type can be used to label up to 2^{31} nodes (because we can have only signed integers), but with 2 bits used for edge information, it can label only up to 2^{29} (or about 1/2 billion) nodes. This becomes problematic when we want to analyze a graph such as ClueWeb12 which has almost a billion nodes and about forty two billion edges. Theoretically, we can switch to 64-bit long data type and be able to do ClueWeb12. However, we still need to overcome the memory limitation problem. With ClueWeb12, forty two billion edges translates to more than 300GB RAM if we use 8 bytes for each, which is way beyond the typical amount of RAM in current commodity machines.

In order to process ClueWeb12, we tried several modifications of the AI algorithm, separating the edge direction information from the node labels in the adjacency list. For instance, we put it in separate list of bytes, where each byte encoding a link. We also tried to use BitSet to more compactly encode the edge and save the space, and WebGraph for the combined adjacency list. However, none of our attempts succeeded in running on ClueWeb12 using 32GB RAM.

To this end, we develop a new algorithm which computes the type of connections between each pair of nodes on the fly, using both the graph and its transpose as input. Using this algorithm, and partial loading method of WebGraph , we were able to process ClueWeb12 on our machine with a budget of 32GB RAM. The algorithm, in a simplified version, is shown in Algorithm 1. We call this Four Pointers Triad Enumeration algorithm due to the fact that we use four pointers: one on each of $N^+(u)$, $N^-(u)$, $N^+(v)$, and $N^-(v)$. Here, $N^+(u)$ is the set of out-neighbours of u

in G, while $N^-(u)$ is the set of out-neighbours of u in G^T. This algorithm is an expansion of the 2-pointer algorithm commonly used for set intersections.

Algorithm 1 iterates over the first node u. This iteration can easily be parallelized. For each, it checks both $N^+(u)$ and $N^-(u)$ to find the neighbours of u and their respective links. For each neighbour of u, v, it finds their common neighbours using four pointers. For each, w, it looks up the triad type based on the links among the three nodes (u, v, w). In line 10, `enum()` is a space holder for an enumeration or listing function.

3 EXPERIMENTS AND DISCUSSIONS

3.1 The Setup

We ran our experiments on a machine with dual Intel Xeon E5620 CPUs and 64GB RAM. Its price is less than $3K, qualifying it as a commodity machine. However, to make a better comparison with other papers, we allowed only 32GB of RAM to be used by the Java virtual machine. The Xeon CPU has a clock speed of 2.40 GHz and 8 threads (16 threads total for the dual). The OS is Linux Ubuntu 14.04.5. We used Java 8 and the WebGraph 3.6.1.

We have five programs to run, listed in Table 1. Both AI and 4P are parallelized on the first node iteration.

Name	Description
SC	Sort and cut preprocessing
RD	Node redistribution
CDt	Build compact dataset
AI	Our AI implementation
4P	Four pointers enumeration

Table 1: The programs used in this experiment.

3.2 The Datasets

We applied our algorithms on five networks in compressed WebGraph format. The datasets were downloaded from the WebGraph website [4] (http://law.di.unimi.it/datasets.php). For each, we downloaded both the graph and the transpose graph.

Here are our selection of networks:

1. **cnr-2000**: a small crawl from the Italian CNR (Consiglio Nazionale delle Ricerche).
2. **ljournal-2008** (abbreviated as **ljournal**): a snapshot from LiveJournal (https://www.livejournal.com/) in 2008.
3. **arabic-2005** (abbreviated as **arabic**): a of websites that contain arabic, performed by UbiCrawler [3] in 2005.
4. **uk-2005**: a shallow crawl of .uk domain, performed by UbiCrawler [3] in 2005.
5. **clueweb12** (abbreviated as **clueweb**): a crawl of English webpages, created by the Lemur Project (http://www.lemurproject.org/clueweb12/index.php), with outlink nodes removed.

The statistics of these datasets are listed in Table 2. Notice that G and G^T can have different sizes because the results of the compression can be different. The smallest dataset, `cnr-2000`, has 3.2M edges, while the largest dataset, `clueweb`, has more than 42B edges. The degree statistics are listed in Table 3. The statistics of the resulting graphs after preprocessing are listed in Table 4. Note that redistribution will not change these statistics.

			Size of	
Name	$\|V\|$	$\|E\|$	G	G^T
cnr-2000	325,557	3,216,152	1.2M	920K
ljournal	5,363,260	79,023,142	105M	105M
arabic	22,744,080	639,999,458	141M	96M
uk-2005	39,459,925	936,364,282	201M	140M
clueweb	978,408,098	42,574,107,469	12G	7.0G

Table 2: Dataset statistics of the directed graphs. The sizes are of the compressed WebGraph files, in bytes.

Name	$d^+_{\max}$	$d^-_{\max}$	d_{avg}	$d^-_{\max}/d^+_{\max}$
cnr-2000	2,716	18,235	9.9	6.7
ljournal	2,469	19,409	14.7	7.9
arabic	9,905	575,618	28.1	58.1
uk-2005	5,213	1,776,852	23.7	340.9
clueweb	7,447	75,611,690	43.5	10,153.3

Table 3: Degrees statistics in the original datasets.

Name	$\|E^{\text{eff}}\|$	$\|E^{T,\text{eff}}\|$	$d^{+,\text{eff}}_{\max}$	$d^{-,\text{eff}}_{\max}$
cnr-2000	2,580,192	548,518	1,336	81
ljournal	42,947,594	35,043,920	1,257	397
arabic	534,631,498	96,522,171	6,646	3,126
uk-2005	759,564,189	161,780,889	5,213	584
clueweb	36,605,200,001	5,968,907,468	5,873	4,242

Table 4: Dataset statistics of the effective directed graphs.

3.3 Results

We ran the AI, and 4P triad enumeration programs on the chosen datasets. We verified that we got the same numbers from both programs, when run with or without pre-processing. The numbers of triads for each types are listed in Table 5.

In Table 6 we list the running times. The graphs had been preprocessed by the SC program (sort and cut) before being used as input, but without node redistribution. The running times on graphs without preprocessing are not shown here. However, note that this preprocessing is important in keeping the running time low. The CDt program takes the graph and its transpose in the compressed WebGraph format and produces a compact dataset written into a plain text file which is then used as input by the AI program. The 4P program takes the WebGraph files directly as input. Therefore, we compare 4P with AI+CDt. Except for `cnr-2000`, AI+CDt is faster than 4P. This is due to repeat decompression cost in 4P. However, AI+CDt is unable to process `clueweb` for the reason that is explained in the previous section (inability to represent 1 billion nodes using 29 bits for each), while 4P can. Notice that our AI implementation can process `arabic` in a shorter time than the one reported in Parimalarangan's paper [12] (In that paper they did not report the time to build the compact data set, CDt, and excluded the loading time. Our numbers here for AI include the loading time.).

Next, we checked whether redistribution can improve the performance. In Table 7 we list the running times on the graphs that had been preprocessed and which nodes had been redistributed. The cost of redistributing the nodes is listed as RD. We see that this RD cost can be quite significant. Taking this cost into consideration, it is not always worth it to do redistribution. For example, on `ljournal`, the 4P time without RD is 81 seconds, while RD +

Name	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	Δ_7
cnr-2000	10,342	9,899,367	85,969	2,433,041	6,736,504	419,472	1,392,934
ljournal	530,051	86,777,707	10,421,919	69,748,792	44,608,271	80,177,727	118,890,977
arabic	2,668,704	6,906,765,421	30,427,662	1,571,745,235	11,765,868,185	384,594,679	16,233,290,956
uk-2005	5,335,890	5,198,533,331	48,779,535	1,773,901,843	9,499,139,863	411,396,906	4,842,278,688
clueweb	281,444,867	517,684,665,693	2,261,300,705	153,674,084,413	790,291,640,762	28,556,769,295	502,545,385,030

Table 5: The counts of triads of each types on the selected networks.

Name	SC	AI	CDt	AI+CDt	4P
cnr-2000	2.75	1.25	2.06	3.31	3.0
ljournal	74	27	28	55	81
arabic	200	429	206	635	2961
uk2005	311	354	319	673	796
clueweb	12,870	-	-	-	115,960

Table 6: The running time (in seconds) of triad enumeration using Four Pointer algorithm (4P), and AI algorithm (AI). Also listed are the preprocessing time (SC), and the time to build the compact dataset (CDt). Since, 4P does not need compact dataset, we compare 4P with AI+CDt.

$4P^{RD}$ is 86 + 30.5 = 116.5 seconds. For `arabic`, though, the RD is helpful in bringing the overall cost down.

Name	RD	AI^{RD}	CDt^{RD}	$AI+CDt^{RD}$	$4P^{RD}$
cnr-2000	3.7	0.98	1.49	2.5	3.2
ljournal	86	18.4	22.8	41.2	30.5
arabic	453	215	154	369	483
uk2005	632	312	248	560	366
clueweb	30,413	-	-	-	-*

Table 7: The running time (in seconds) of triad enumeration using Four Pointer algorithm (4P), and AI algorithm (AI) on redistributed graphs. *The run on `clueweb` cannot be done in a reasonable amount of time.

Notice that with RD our 4P becomes competitive to AI+CDt. This can be understood from the fact that workload imbalanced affects 4P more than AI since 4P has to do decompression on the adjacency list, and hence RD benefits 4P more as it reduces the imbalance. It is faster on `ljournal` and `uk2005`, and just a bit slower on `cnr-2000` and `arabic`.

The redistribution, however, has an unwanted effect on WebGraph compression. The compression works best if the distances among the neighbour nodes are not large. With redistribution, there are large distances within a neighbour set, which in turn lower the overall compression ratio. As such, we do not advise performing redistribution on very large graphs such as `clueweb`. For such graphs sort-and-cut preprocessing is all what is needed for our algorithm 4P to complete in reasonable time.

4 CONCLUSIONS

We have shown that through optimized preprocessing and parallelization we are able to run triad enumeration on very large graphs using a single commodity machine in reasonable time. We also designed an algorithm, 4P, with better scalability than the state-of-the-art, which, with some trade-off on the performance, can run on graphs of a billion nodes and billions of edges, counting trillions of triads. In our solution, the WebGraph framework plays an important role in alleviating the memory problem. In conclusion, our results show that triad enumeration can be done on a commodity machine even for very large graphs such as ClueWeb12.

REFERENCES

[1] Vladimir Batagelj and Andrej Mrvar. 2001. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social networks* 23, 3 (2001), 237–243.

[2] Vladimir Batagelj and Matjaž Zaveršnik. 2007. Short cycle connectivity. *Discrete Mathematics* 307, 3-5 (2007), 310–318.

[3] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A scalable fully distributed web crawler. *Software: Practice & Experience* 34, 8 (2004), 711–726.

[4] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiResolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web*, Srinivasan Sadagopan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[5] Paolo Boldi and Sebastiano Vigna. 2004. The Webgraph Framework I: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*. ACM, New York, NY, USA, 595–602. https://doi.org/10.1145/988672.988752

[6] Paolo Boldi and Sebastiano Vigna. 2004. The Webgraph Framework II: Codes for the world-wide web. In *Data Compression Conference, 2004. Proceedings. DCC 2004*. IEEE, 528.

[7] George Chin Jr, Andres Marquez, Sutanay Choudhury, and John Feo. 2012. Scalable triadic analysis of large-scale graphs: Multi-core vs. multi-processor vs. multi-threaded shared memory architectures. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*. IEEE, 163–170.

[8] James A Davis and Samuel Leinhardt. 1972. The structure of positive interpersonal relations in small groups. *Sociological Theories in Progress* 2 (1972), 218–251.

[9] Jean-Pierre Eckmann and Elisha Moses. 2002. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings of the National Academy of Sciences* 99, 9 (2002), 5825–5829.

[10] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science* 407, 1-3 (2008), 458–473.

[11] Mark EJ Newman, Duncan J Watts, and Steven H Strogatz. 2002. Random graph models of social networks. *Proceedings of the National Academy of Sciences* 99, suppl 1 (2002), 2566–2572.

[12] Sindhuja Parimalarangan, George M Slota, and Kamesh Madduri. 2017. Fast parallel graph triad census and triangle counting on shared-memory platforms. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 1500–1509.

[13] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. 2004. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences* 101, 9 (2004), 2658–2663.

[14] Comandur Seshadhri, Ali Pinar, and Tamara G Kolda. 2013. Fast triangle counting through wedge sampling. In *Proceedings of the SIAM Conference on Data Mining*, Vol. 4. 5.

[15] Benjamin M Tabak, Marcelo Takami, Jadson MC Rocha, Daniel O Cajueiro, and Sergio RS Souza. 2014. Directed clustering coefficient as a measure of systemic risk in complex banking networks. *Physica A: Statistical Mechanics and its Applications* 394 (2014), 211–216.

[16] Pinghui Wang, Yiyan Qi, Yu Sun, Xiangliang Zhang, Jing Tao, and Xiaohong Guan. 2017. Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage. *Proceedings of the VLDB Endowment* 11, 2 (2017), 162–175.

[17] Stanley Wasserman and Katherine Faust. 1994. *Social network analysis: Methods and applications*. Vol. 8. Cambridge University Press.

[18] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393, 6684 (1998), 440.

[19] Zhi Yang, Christo Wilson, Xiao Wang, Tingting Gao, Ben Y Zhao, and Yafei Dai. 2014. Uncovering social network sybils in the wild. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 8, 1 (2014), 2.

Fast Truss Decomposition in Large-scale Probabilistic Graphs

Fatemeh Esfahani, Jian Wu, Venkatesh Srinivasan, Alex Thomo, and Kui Wu
{esfahani,wujian,srinivas,thomo,wkui}@uvic.ca

ABSTRACT

Truss decomposition is popular for finding dense substructures in graphs. Discovering trusses in deterministic graphs has been widely discussed in the literature. However, with the intrinsic uncertainty in many networks such as social, biological, and communication networks, it is of great importance to study truss decomposition in a probabilistic context, but this has received much less attention till now. Furthermore, due to computation challenges of truss decomposition in probabilistic graphs, the state-of-the-art approaches are not scalable to large graphs.

Given a user-defined threshold, we are interested in finding all the maximal subgraphs which are a k-truss with high probability. The most important challenge, which distinguishes truss decomposition in probabilistic graphs from deterministic graphs, is computing tail probabilities of edge supports. We employ a special version of the Central Limit Theorem (CLT) to obtain the tail probabilities efficiently. Based on our CLT approach we propose a peeling algorithm for truss decomposition of a probabilistic graph that scales to very large graphs and offers significant improvement over state-of-the-art. Our extensive experimental results confirm the scalability and efficiency of our approach.

1 INTRODUCTION

Probabilistic graphs are graphs in which each edge has an existence probability [2]. Many real-world networks, such as social, trust, communication, and biological networks, feature uncertainty and thus can be modeled as probabilistic graphs.

Dense subgraph mining is an important way to analyze the structure of networks [7]. A popular notion of cohesive graph is the k-truss, which is defined as a maximal subgraph in which each edge participates in at least $(k-2)$ triangles within that subgraph. The k-truss features a variety of applications [5]. For instance, k-truss is a useful tool for visualization of complex networks [12]. Also, k-trusses are the basis of several community models [8]. It is thus important to discover k-trusses in probabilistic graphs.

Truss decomposition in deterministic graphs is a straightforward task and has been broadly studied in the literature [4, 9, 11]. However, in probabilistic graphs, truss computation is challenging and has received much less attention. We use the notation of local probabilistic (k, η)-truss introduced in [5], and will be explained in more detail in the next section.

Challenges and contributions. Probabilistic truss decomposition is associated with significant challenges due to intrinsic uncertainty in probabilistic graphs. Thus, the general idea of iterative edge removal in deterministic graphs does not work by itself in probabilistic graphs. For instance, counting the number of triangles which contain an edge is straightforward in deterministic graphs. But, in probabilistic graphs, the triangles in which an edge might participate have combinatorial nature [5]. As a result, the most difficult task is computing edge support probabilities efficiently. This becomes particularly important when the input graph is huge. In [5], support probability of an edge $e = (u, v)$ is computed using dynamic programming (DP), which has a complexity of $O((\min\{d(u), d(v)\})^2)$, where $d(u)$ and $d(v)$ is deterministic degree of u and v, respectively. Unfortunately, the values of $d(u)$ and $d(v)$ can be in the millions in many real-world social and web networks, and quadratic time complexity of DP makes it impractical for huge graphs.

Realizing the fact that each triangle in probabilistic graph can be defined as a Bernoulli random variable with an existence probability, we design a novel approach based on Lyapunov's special version of the Central Limit Theorem [6] to approximate probability distribution of the support of an edge. We show that the proposed approximation is accurate for our problem when the number of triangles is big. In addition, we derive an error bound on the approximation to ensure that the output probabilities are very close to the values obtained through exact computation.

We design a peeling algorithm for probabilistic k-truss decomposition. Our algorithm takes advantage of the fast calculation of edge support probabilities in time $O(\min\{d(u), d(v)\})$ using central limit theorem. It also uses optimized array-based data structures for storing edge information of the graph.

In summary, our contributions are as follows.

- We introduce an efficient approach based on Lyapunov's central limit theorem to compute support probabilities of edges in the input graph (Section 3.1).
- Using theoretical analysis, we obtain error bound of the approximation, which shows that the higher the number of triangles, the higher the accuracy of the approximation results (Corollary 1).
- We develop a peeling algorithm based on recursive edge deletions (Section 3.2), which, by utilizing central limit theorem and additional data structures, is able to calculate truss decomposition in very big probabilistic graphs not possible with the pure DP approach (Section 4).

2 BACKGROUND

Trusses in deterministic graphs. Let $G = (V, E)$ be an undirected graph with no self-loops. For a vertex v, let $N_G(v)$ be the set of v's neighbors in G. A triangle in the input graph is defined as a cycle of length 3, denoted by $\triangle_{uvw}$, where $u, v, w \in V$. Given an edge $e = (u, v)$, the *support of* e in G is the number of triangles that contain e. Formally, $\text{sup}_G(e) = |N_G(v) \cap N_G(u)|$.

The k-truss of G is defined as the maximal induced subgraph $T_k(G) = (V', E_{V'})$ in which each edge $e \in E_{V'}$ has support of at least $(k-2)$. The set of all k-trusses forms truss decomposition of G, where $2 \leq k \leq k_{\max}$, and $k_{\max}$ is the largest support of any edge.

Probabilistic graphs. A probabilistic graph is defined as $\mathcal{G} = (V, E, p)$, where V and E are as before and $p : E \rightarrow (0, 1]$ is a function that assigns existence probability $p(e)$ to edge e. In the most common probabilistic graph model [2], the existence probability of each edge is assumed to be independent of other edges.

To analyze probabilistic graphs, we use the concept of *possible worlds*, which are deterministic graph instances of $\mathcal{G}$. For each

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

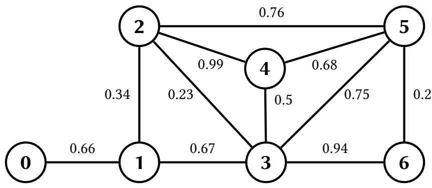


Figure 1: A probabilistic graph.

possible world $G = (V, E_G) \sqsubseteq \mathcal{G}$, where $E_G \subseteq E$, the probability of observing that possible world is obtained as follows: $\Pr(G) = \prod_{e \in E_G} p(e) \prod_{e \in E \setminus E_G}(1 - p(e))$.

Given an edge $e = (u, v)$, let $k_e = |N(u) \cap N(v)|$. We define the notion of *η-support*, denoted by *η-sup$_{\mathcal{G}}(e)$*, as the maximum k for which $\Pr_{G \sqsubseteq \mathcal{G}}[\mathrm{sup}_G(e) \geq k] \geq \eta$, where $k = 0, \ldots, k_e$, and the probability is taken over all the possible worlds $G \sqsubseteq \mathcal{G}$. In the rest of the paper, we use $\Pr[\mathrm{sup}_{\mathcal{G}}(e) \geq k]$ to denote $\Pr_{G \sqsubseteq \mathcal{G}}[\mathrm{sup}_G(e) \geq k]$.

For instance, consider Figure 1, edge $e = (2, 5)$, and $\eta = 0.5$. With the assumption that e exists (with probability $p(e) = 0.76$), edge e has support at least 2 with probability $0.76 \cdot (0.99 \cdot 0.68) \cdot (0.23 \cdot 0.75) = 0.0883$ (product of the probabilities that triangles $\triangle_{245}$ and $\triangle_{235}$ exist in a possible world), and it has support at least 1 with probability $0.76 \cdot \big(1 - (1 - 0.99 \cdot 0.68) \cdot (1 - 0.23 \cdot 0.75)\big) = 0.5545$ (complementary probability that none of the two triangles are in a possible world). Since 0.5545 is greater than the threshold η, the η-support of edge e is 1.

In probabilistic context, $\mathrm{sup}_{\mathcal{G}}(e)$ is a random variable which can take on integer values form zero to k_e. Furthermore, as k increases, the value of $\Pr[\mathrm{sup}_{\mathcal{G}}(e) \geq k]$ decreases.

Trusses in probabilistic graphs. In order to compute truss decomposition in probabilistic graphs we follow the local (k, η)-*truss* model defined in [5]:

Definition 1. *Let $\mathcal{G} = (V, E, p)$ be a probabilistic graph. Given threshold $\eta \in [0, 1]$, the local (k, η)-truss is the maximal induced subgraph $T_{(k,\eta)}(\mathcal{G}) = (V', E_{V'}, p)$ in which the η-support of each edge $e \in E_{V'}$ is at least $(k - 2)$. The set of all the local (k, η)-trusses forms the local truss decomposition of $\mathcal{G}$.*

Similar to deterministic case, local (k, η)-trusses are unique and nested into each other. The highest value of k for which e belongs to a local (k, η)-truss is called *η-truss* number or probabilistic trussness of e.

Computing η-supports using Dynamic Programming. Support probabilities are computed using $\Pr[\mathrm{sup}_{\mathcal{G}}(e) \geq k] = 1 - \sum_{i=0}^{k-1} \Pr[\mathrm{sup}_{\mathcal{G}}(e) = i]$. One way of calculating $\Pr[\mathrm{sup}_{\mathcal{G}}(e) = i]$ is to use dynamic programming as proposed in [5]. Given an edge $e = (u, v)$, time complexity of this method of computation is $O(k_e^2)$. Since $k_e \in O(\min\{d(u), d(v)\})$, where $d(u)$ and $d(v)$ are deterministic degree of u and v in $\mathcal{G}$, respectively, this method of computation is not practical when the minimum degree of two neighbors is big, say over 100, which is quite common in all our datasets. In addition, web-scale graphs have millions of such edges, and as a result, if DP is applied to each edge the total processing time increases considerably. In the next section we propose an alternative approach for fast computation of η-support of an edge e using Lyapunov central limit theorem.

3 PEELING ALGORITHM FRAMEWORK USING CENTRAL LIMIT THEOREM

We describe a CLT-focused algorithm to compute truss decomposition in probabilistic graphs. The pseudocodes/proofs are omitted due to page limit.

3.1 Computing η-Supports using Central Limit Theorem

We first show how a special version of Central Limit Theorem (CLT) can be applied to accurately estimate $\Pr\big[\mathrm{sup}_{\mathcal{G}}(e) \geq k\big]$. Then, we show theoretical bounds on the accuracy of this approximation. Specifically, we show that CLT can produce a very accurate approximation to tail probabilities of the support edge.

Based on CLT, the distribution of properly scaled sum of a sequence of random variables converges to normal distribution under specific conditions. In this paper, we consider a variant called Lyapunov CLT that can be applied when random variables are independent, but not necessarily identically distributed. Lyapunov CLT can be formally stated in the following:

Theorem 3.1. ***Lyapunov CLT.*** *Let $\zeta_1, \zeta_2, \cdots, \zeta_n$ be a sequence of independent, but non-identically distributed random variables, each with finite expected value μ_k and variance σ_k. Let*

$$s_n^2 = \sum_{k=1}^{n} \sigma_k^2, \tag{1}$$

Lyapunov CLT states that if

$$\lim_{n \to \infty} \frac{1}{s_n^{2+\delta}} \sum_{k=1}^{n} \mathrm{E}[|\zeta_k - \mu_k|^{2+\delta}] = 0, \tag{2}$$

for some $\delta > 0$, then $\frac{1}{s_n} \sum_{k=1}^{n} (\zeta_k - \mu_k)$ converges in distribution to a standard normal random variable.

Equation (2) is called Lyapunov's condition which in practice is usually tested for the special case $\delta = 1$. The proof for this theorem can be found in [3].

Computing η-supports using Lyapunov CLT. Given an edge e, to compute $\Pr\big[\mathrm{sup}_{\mathcal{G}}(e) \geq k\big]$ we assume that e exists. Thus, the true edge support probability is obtained by multiplication of $\Pr\big[\mathrm{sup}_{\mathcal{G}}(e) \geq k\big]$ with $p(e)$.

Each edge e_j in probabilistic graph has existence probability of $p(e_j)$, which is independent of the other edge probabilities. As a result, associated with each edge e_j we can define a Bernoulli random variable ζ_{e_j} which takes on 1 with probability $p(e_j)$ and 0 with probability $(1 - p(e_j))$. Since each edge is assumed to exist independently of other edges, ζ_{e_j}'s are independent. Given an edge $e = (u, v)$, let $\mathcal{T}_e$ be a set of all the common neighbors of u and v in $\mathcal{G}$. We have,

$$\mathcal{T}_e = N(u) \cap N(v) = \left\{t_1, \cdots, t_{k_e}\right\}.$$

For each common neighbor t_i, let ζ_{u,t_i} and ζ_{v,t_i} be the corresponding Bernoulli random variables to the edges (u, t_i) and (v, t_i), respectively. Let $X_i = \zeta_{u,t_i} \cdot \zeta_{v,t_i}$. The following observations hold for each random variable X_i: **(1)** X_i's are independent, since ζ_{u,t_i} and ζ_{v,t_i} are independent random variables. **(2)** X_i's are Bernoulli random variables which take on 1 with probability $p(u, t_i) \cdot p(v, t_i)$. This is because X_i can be equal to 1 when both $\zeta_{u,t_i} = 1$ and $\zeta_{v,t_i} = 1$, with probability $p(u, t_i) \cdot p(v, t_i)$. Otherwise, if at least one of them is 0, the value of X_i will become zero. The probability that at least one of these random variables become zero is $1 - (p(u, t_i) \cdot p(v, t_i))$.

Now, let us consider the triangle $\triangle_{uvt_i}$. It should be noted that only common neighbours can create a triangle containing edge e. With the assumption that e exists, the triangle $\triangle_{uvt_i}$ exists if both edges (u, t_i) and (v, t_i) exist, which is associated with $X_i = 1$. On the other hand, the triangle does not exist if at least one of edges, (u, t_i) and (v, t_i), does not exist, which corresponds to $X_i = 0$. Therefore, corresponding to each triangle $\triangle_{uvt_i}$, we can define the Bernoulli random variable X_i.

Let $p_i = p(u, t_i) \cdot p(v, t_i)$. Since X_i is a Bernoulli random variable, we know that $\mathrm{E}[X_i] = \mu_i = p_i$ and $\mathrm{Var}[X_i] = p_i(1 - p_i)$. Since $\sup_{\mathcal{G}}(e) = \sum_{i=1}^{k_e} X_i$, we have:

$$\Pr[\sup_{\mathcal{G}}(e) \geq k] = \Pr\left[\sum_{i=1}^{k_e} X_i \geq k\right]. \tag{3}$$

Bernoulli random variables X_i's are independent, but may not be identically distributed. Thus, if condition (2) is satisfied and if k_e is large enough, we can conclude that $\frac{1}{s_{k_e}} \sum_{i=1}^{k_e}(X_i - \mu_i)$ has standard normal distribution, where $s_{k_e} = \sqrt{\sum_{i=1}^{k_e} p_i(1 - p_i)}$. In order to compute the right-hand side of equation (3), we can subtract $\sum_{i=1}^{k_e} \mu_i$ from both sides of the inequality, and then divide by s_{k_e} which results in:

$$\Pr\Big[\sum_{i=1}^{k_e} X_i \geq k\Big] = \Pr\Big[\frac{1}{s_{k_e}} \sum_{i=1}^{k_e}(X_i - \mu_i) \geq \frac{1}{s_{k_e}}(k - \sum_{i=1}^{k_e} \mu_i)\Big]. \tag{4}$$

Using Lyapunov CLT and setting

$$Z = \frac{1}{s_{k_e}} \sum_{i=1}^{k_e}(X_i - \mu_i), \tag{5}$$

we can conclude that Z has standard normal distribution. Thus

$$\Pr\big[\sup(e) \geq k\big] \cong \Pr\big[Z \geq z\big], \tag{6}$$

where $z = \frac{1}{s_{k_e}}(k - \sum_{i=1}^{k_e} \mu_i)$. Using the complementary cumulative distribution function [13] of standard normal variable Z, we can simply evaluate the right-hand side of Equation (6) for each value of k. Thus, to find the η-support for an edge, we start with $k = 1$, approximate $\Pr[\sup(e) \geq k]$ using Lyapunov CLT, and find the maximum k for which the probability multiplied by $p(e)$ is above threshold η. For an edge e, the obtained value of k, which can be in range from one to k_e, is set as initial η-support for that edge. Given an edge $e = (u, v)$, time complexity of finding η-support is $O(k_e)$, where $k_e = |N(u) \cap N(v)|$. Recall that DP required $O(k_e^2)$ for this step.

In the following we show that Lyapunov's condition in Theorem 3.1 is satisfied for our problem. We set $\delta = 1$ in Equation (2) to show that this condition holds for a sequence of non-identically distributed Bernoulli random variables.

Theorem 3.2. *Given a sequence of random variables $X_i \sim$ Bernoulli(p_i), where $1 \leq i \leq n$, the Lyapunov's condition (2) for $\delta = 1$ is satisfied whenever $s_n^2 = \sum_{k=1}^{n} p_k(1 - p_k) \to \infty$.*

Accuracy of the Approximation. Using Berry–Esseen theorem [14], in the following corollary we show how to obtain an upper-bound on the maximal error while approximating the true distribution of the sum of X_i's with the normal distribution.

Corollary 1. *For each edge e in the probabilistic graph $\mathcal{G}$ with X_i's being Bernoulli random variables defined as above in this section, where $i = 1, \ldots, k_e$, the error bound on the approximation of the right-hand side of Equation (6) to the standard normal distribution is given as follows:*

$$\sup_{x \in \mathbb{R}} \left|F_{k_e}(x) - \Phi(x)\right| \leq \frac{0.56}{\sqrt{p_1(1 - p_1) + \cdots + p_{k_e}(1 - p_{k_e})}}.$$

3.2 Peeling Algorithm (PA)

In [10], a peeling algorithm was proposed to calculate the k-truss in deterministic graphs. While the algorithm is not applicable to probabilistic graphs, its optimized array-based data structures for storing edge information of the graph are useful. Our new peeling algorithm, termed as *CLT_based-PA* algorithm, is built on the same array-based data structures but utilizes central limit theorem to compute and update support probabilities of edges which participate in more than 100 triangles.[1]

The *CLT_based-PA* algorithm consists of two main parts: **(1)** initial probabilistic support computation, and **(2)** probabilistic truss computation which involves updating probabilistic support values once an edge is removed.

In initial support computation step, the η-support of each edge e is computed using CLT and Equation (6), if k_e is greater than 100. Otherwise, DP can be used safely. The details on DP approach can be found in [5]. The initial phase can be executed in parallel, since probabilistic support of each edge can be computed independently of other edges.

After initialization, the *CLT_based-PA* algorithm runs in three steps: First, sort edges in ascending order of their η-support in the array *sortedEdge*, and store their positions in the array.

Then, remove edges with the lowest η-support. The removal of an edge $e = (u, v)$ affects the η-support of all edges that can constitute triangles with (u, v). As a result, the algorithm finds all the common neighbors w of u and v, i.e., $\triangle_{uvw}$ is a triangle containing edge (u, v).

At the third step, the η-support of (u, w) and (v, w) is updated if their η-supports are greater than e's η-support. In the updating part, if the number of remaining triangles which contain edges (u, w) and (v, w) is greater than 100, we perform update phase using CLT approach. Otherwise, we apply DP. Since the η-support has been changed, we change the position of edges (v, w) and (u, w) in *sortedEdge* array in constant time [10]. The algorithm continues until all the edges in the graph are removed. Then, the trussness of each edge is obtained by adding 2 to the final η-support.

4 EXPERIMENTS

Our implementations are in Java and the experiments are conducted on a commodity machine with Intel i7, 2.2Ghz CPU, and 12Gb RAM, running Ubuntu 14.03. The hard disk is Seagate Barracuda ST31000524AS 2TB 7200 RPM.

The statistics for the datasets are shown in Table 1. We obtained flicker, dblp, and biomine from the authors of [2], and the rest of the datasets from Laboratory of Web Algorithmics.[2] Each horizontal line in the table categorizes the datasets according to their size, small (S), medium (M), and large (L). We use the Webgraph framework [1] to store these datasets. The flickr, dblp, and biomine datasets already contained edge probability values. For the other datasets we generated probability values uniformly distributed in $[0, 1]$.

[1] This value was chosen because it was large enough to keep the approximation error obtained from Corollary 1 small.

[2] http://law.di.unimi.it/datasets.php

Name	$\lvert V\rvert$	$\lvert E\rvert$
flickr	24,125	300,836
dblp	684,911	2,284,991
cnr-2000	325,557	2,738,969
biomine	1,008,201	6,722,503
ljournal-2008	5,363,260	49,514,271

Table 1: Dataset Statistics

Dataset	η	Running Time	Running Time	gain (%)
		DP_Pure	CLT_based–PA	
flickr	0.1	351	94	73%
dblp	0.1	37	34	8.50%
biomine	0.1	7642	2554	67%
cnr-2000	0.1	N.P.	7874	100%+
ljournal-2008	0.1	54627	26129	52%
	0.2	50614	27064	47%
	0.3	45052	24799	45%
	0.4	36563	21332	42%
	0.5	28773	16291	43

Table 2: Running times (sec) of DP_Pure and CLT_based–PA. The column "gain (%)" reports the gain of CLT_with_DP algorithm over DP_Pure algorithm. We use N.P. for "Not Possible".

Dataset	**η-suppmax**	**kmax**	**η**
flickr	49	47	0.1
dblp	42	14	0.1
biomine	151	33	0.1
cnr-2000	4672	13	0.1
ljournal-2008	1030	51	0.1
	1015	43	0.2
	1001	35	0.3
	980	27	0.4
	530	19	0.5

Table 3: Maximum η-support, maximum probabilistic trussness, value of the threshold η.

Table 2 represents the running times of our proposed approach, versus the running times of the state-of-the-art, which uses dynamic programming (DP) only and is referred as *DP_Pure*. The last column shows the gain of *CLT_based–PA* algorithm over *DP_Pure*. For ljournal-2008, we present the results for different values of η ranging from 0.1 to 0.5. However, for the other datasets, we only show the results for $\eta = 0.1$, and omit results for $\eta = 0.2, \ldots, 0.5$, since they are similar to those for $\eta = 0.1$ and their performance trend is similar to what we see for ljournal-2008. As can be seen, *CLT_based–PA* algorithm is significantly faster than *DP_Pure*. For instance, for biomine, which is a large dataset, the gain of our algorithm is 67 percent, making *CLT_based–PA* truss decomposition algorithm three times faster than *DP_Pure*.

CLT_based–PA produced the results in about 1.5 minutes and about 34 seconds for flickr and dblp, respectively. Although flickr is smaller than dblp in terms of the number of vertices and edges, its probabilistic maximum truss is much greater at value 47 compared to 14 in dblp. We represent the maximum probabilistic truss and maximum probabilistic support in Table 3. *These values are the same as those obtained by DP_Pure.* On biomine which is a large dataset, our proposed algorithm completed in about 43 minutes; which is quite impressive. In contrast, *DP_Pure* produced the results in more than 2 hours. The running time for ljournal-2008 increases, which is quite reasonable, because this graph has 49 million edges with probabilistic support of 1030 when $\eta = 0.1$. The same argument holds for cnr-2000 which has probabilistic support of 4672 which is significantly big. *DP_Pure wasn't able to run to completion in our machine after one day.*

Effect of η values. The running time of both algorithms increases as η becomes small. This is because as η decreases, the chance for support probabilities to pass the threshold increases, resulting in larger values of η-supports. This is particularly important in performance evaluation of *DP_Pure* algorithm– as η decreases the DP algorithm approaches its worst case time complexity, $O(k_e^2)$, for an edge e. As a result, for larger values of η, the running time of *DP_Pure* improves, but is still by far slower than *CLT_based–PA*. In terms of the effect of η on truss decomposition and support values, as can be seen in Table 3, the maximum truss and maximum initial probabilistic support decrease as η increases. As before, we report the maximum truss, and the maximum η-support for ljournal-2008 for $\eta = 0.1, \ldots, 0.5$, whereas for the other datasets we only show the values for $\eta = 0.1$.

5 CONCLUSIONS

We presented a peeling algorithm for computing truss decomposition in probabilistic graphs at web scale. Our peeling algorithm uses Lyapunov's central limit theorem to obtain the probabilistic support for an edge. Unlike the dynamic programming approach, the computation does not rely on incremental evaluation of support probabilities. In addition, it can efficiently update probabilistic support when a triangle is removed from the input graph, without the need of storing all the previously computed support probabilities. We evaluated our algorithm and showed that it is significantly faster than state-of-the-art for large datasets. For large and medium datasets our algorithm obtained approximately 50 percent gain over the proposed algorithm in the literature, completing the biomine dataset in less than one hour.

REFERENCES

[1] P. Boldi and S. Vigna. 2004. The webgraph framework I: compression techniques. In *Proc. WWW'04*. ACM, 595–602.

[2] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich. 2014. Core decomposition of uncertain graphs. In *Proc. SIGKDD*. ACM, 1316–1325.

[3] H. Cramér. 1946. *Mathematical Methods of Statistics*. PUP.

[4] X. Huang, H. Cheng, L. Qin, W. Tian, and J. Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proc. SIGMOD*. ACM, 1311–1322.

[5] X. Huang, W. Lu, and L. V. Lakshmanan. 2016. Truss decomposition of probabilistic graphs: Semantics and algorithms. In *Proc. SIGMOD*. ACM, 77–90.

[6] H. Kobayashi, B.L. Mark, and W. Turin. 2011. *Probability, Random Processes, and Statistical Analysis*. Cambridge University Press.

[7] V. Lee, N. Ruan, R. Jin, and Ch. Aggarwal. 2010. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*. Springer, 303–336.

[8] Y. Li, T. Kuboyama, and H. Sakamoto. 2013. Truss decomposition for extracting communities in bipartite graph. In *Proc. IMMM*. 76–80.

[9] J. Wang and J. Cheng. 2012. Truss decomposition in massive networks. *Proc. VLDB* 5, 9 (2012), 812–823.

[10] J. Wu, A. Goshulak, V. Srinivasan, and A. Thomo. 2018. K-Truss Decomposition of Large Networks on a Single Consumer-Grade Machine. In *Proc. ASONAM*. IEEE, 873–880.

[11] Y. Zhang and S. Parthasarathy. 2012. Extracting analyzing and visualizing triangle k-core motifs within networks. In *Proc. ICDE*. IEEE, 1049–1060.

[12] F. Zhao and A. Tung. 2012. Large scale cohesive subgraphs discovery for social network visual analysis. In *Proc. VLDB*, Vol. 6. VLDB Endowment, 85–96.

[13] D. Zwillinger and S. Kokoska. 2000. *CRC Standard probability and statistics tables and formulae*. Chapman and Hall/CRC, USA.

[14] D. Zwillinger and S. Kokoska. 2013. *Probability Theory*. Springer-Verlag, London.

Short Paper

An Experimental Study on Network Immunization

Alvis Logins
Aarhus University

Panagiotis Karras
Aarhus University

ABSTRACT

Given a network in which a undesirable rumor, disease, or contamination spreads, which set of network nodes should we *block* so as to contain that spread? Past research has proposed several methods to address this *network immunization* (NI) problem, which is to find a set of k nodes, such that the undesirable dissemination is minimized in expectation when they are blocked. As the problem is NP-hard, some algorithms utilize solely features of the network structure in a *preemptive* manner, to others that take into account the specific source of a contamination in a *data-aware* fashion. This paper presents an experimental study on NI algorithms and baselines under the independent cascade (IC) diffusion model. We employ a variety of synthetic and real-world networks with diverse graph density, degree distribution, and clustering coefficients, under realistically calculated influence probabilities. We conclude that data-aware approaches based on the construct of *dominator trees* usually perform best; however, in networks with a power-law degree distribution, preemptive approaches utilizing spectral network properties shine out by virtue of their efficiency in identifying central nodes.

1 INTRODUCTION

Real-world networks facilitate the spread of ideas, behaviors, inclinations, or diseases via *diffusion* processes [10]. Oftentimes a diffusion of malicious nature needs to be contained via countermeasures [11]. One such countermeasure is the *blocking* of a subset of network nodes. *Network Immunization* (NI) calls to find an optimal set of nodes to block so as to arrest a diffusion.

Early works on NI were motivated by epidemiology [3, 12], categorizing individuals as Susceptible $\mathbb{S}$, Infected $\mathbb{I}$, or Recovered $\mathbb{R}$. Those who are infected infect their susceptible neighbors with a transition rate β, and become recovered (hence immune) with transition rate γ. In the context of social networks [14], the *Independent Cascade* (IC) model [4] generalizes the SIR model, assigning an independent transition rate β to each edge. Kempe et al. [6] formulated the Influence Maximization (IM) problem under the IC model, where the goal is to select k seed nodes that maximize the expected diffusion spread; since then, the problem has been studied extensively [10, 14].

The NI problem is complementary to the IM problem. Certain notions are useful in both. For example, eigenvalue centrality [12] has been used to guide seed selection in IM. Similarly, Chen et al. [2] employ the first eigenvalue λ as a proxy to the objective of NI problem, scoring nodes by the eigen-drop $\Delta\lambda$ that their removal causes, leading to a succession of techniques aiming to to maximize the eigen-drop of immunized nodes [20].

We distinguish two variants on network immunization: *preemptive* immunization finds a solution before the epidemic starts; by contrast, *data-aware* immunization tailors the solution to a particular diffusion seed [19]. The state-of-the-art data-aware solution, *Data-Aware Vaccination Algorithm* (DAVA) [21] employs structures called *dominator trees*. Still, the experimental study in [21] is limited to four datasets with synthetic propagation probabilities; it is not clear how the topology of the network influences the algorithm's performance. At the same time, recent preemtive immunization methods [11, 13] significantly outperform the baselines used in [21], yet have not been compared to DAVA itself. Thus, to the best of our knowledge, no previous work has studied how data-aware and and preemptive immunization strategies fare under different graph topologies.

In this paper, we investigate the performance of state-of-the-art data-aware and preemptive NI solutions on a variety of real-world and synthetic network structures with diverse characteristics, and under realistic influence probabilities with the IC model. Our study features the *first*, to our knowledge, application of the most recent algorithm for eigen-drop maximization and a generic spectral method of activity shaping, to NI under the IC model. We demonstrate that data-aware approaches are leading in a majority of configurations, yet preemtive ones stand out under particular settings of graph density, influence probabilities, degree distribution, and clustering coefficients.

2 BACKGROUND

The classic approach to preemptive NI is the NetShield algorithm [2]. NetShield greedily selects a set of nodes S, aiming to maximize its *Shield value*:

$$Sv(S) = \sum_{i \in S} 2\lambda \mathbf{u}(i)^2 - \sum_{i,j \in S} \mathbf{A}(i,j)\mathbf{u}(i)\mathbf{u}(j)$$

where λ and $\mathbf{u}$ are the largest eigenvalue and the corresponding eigenvector of the network's adjacency matrix $\mathbf{A}$. A set S has high Sv if its elements have high eigenscore $\mathbf{u}(i)$ and are not connected to each other (zero $\mathbf{A}(i,j)$). A high eigenscore implies that their removal leads to a significant eigen-drop $\Delta\lambda$. The algorithm has a $O(n|S|^2)$ complexity, where n is the size of a network.

NetShield defines an *epidemic threshold* β' such that any edge transition probability $\beta > \beta'$ would result in a significant portion of the network being contaminated. The algorithm utilizes the fact that the *epidemic threshold* is related to the first eigenvalue of the network adjacency matrix as $\beta' = 1/\lambda$ [16]. Thus, λ expresses the *vulnerability* of the network to an epidemic. Tariq et al. [13] improved upon NetShield by approximating the eigen-drop, relying on the fact that λ can be expressed as the limit trace of the p-exponential adjacency matrix $\mathbf{A}$, which equals the number of p-sized closed walks in the graph, cw_p:

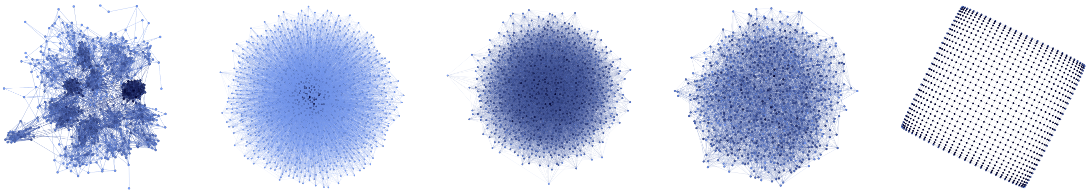
$$\lim_{p \to \inf, p \text{ even}} (trace(\mathbf{A}^p))^{1/p} = \lambda$$

$$trace(\mathbf{A}^p) = cw_p(G)$$

The proposed method greedily selects a set of nodes to block based on their contribution to closed walks, hence to network vulnerability, approximating cw_p by a submodular score function, calculated by partitioning vertices into α equal-size groups by means of a set of hash functions $i = \{1..\gamma\}$. In our experiments, we use $\alpha = 200$ and $i \in \{1..3\}$. The published version suggested using $p = 6$, yet in communication with the authors we confirmed that $p = 8$ feasibly leads to improved results; we refer to this algorithm as *Walk8*; its complexity is $O(n^2 + \gamma(n + \alpha^3) + nk^2)$.

Series ISSN: 2367-2005

10.5441/002/edbt.2019.97



(a) GRP (b) BA (c) Binomial (d) WS (e) Grid

Figure 1: Generated Graph Examples. Darker color indicates higher degree, normalized per graph.

Graph Type	$\|V\|$ $[\cdot 10^3]$	$\|E\|$ $[\cdot 10^3]$	$deg_{\text{min/avg/med/max}}$	clust. coeff. cl	infl. prob. W	seed fract. sf	k fract. kf	Other Parameters
Binomial	1.0	14.7	4/15/15/30	0.012	0.2	0.05	0.05	edge exist. $p = 0.015$
GRP	1.0	14.6	2/30/26/90	0.325	0.1	0.01	0.05	shape param. $s = 20$, $v = 0.9$, intra-group prob. $p_{in} = 0.4$, inter-group $p_{out} = 0.001$
WS	1.0	7.0	18/28/28/44	0.237	0.2	0.05	0.05	neighbors in a ring $l = 15$, rewiring prob. $p = 0.3$
BA	1.0	29.4	28/59/40/498	0.103	0.1	0.05	0.05	prob. of triangle $p = 0.2$, density $m = 15$
Grid	1.0	39.7	4/8/8/8	0.000	0.7	0.05	0.05	–
Stanford	9.9	36.9	0/7/5/555	0.392	0.2	0.2	0.2	–
Gnutella	62.6	147.9	1/4/2/95	0.007	0.2	0.2	0.2	
VK	2.8	40.9	1/29/14/288	0.235	–	0.2	0.2	

Table 1: Default parameters for graph types

DAVA [21] accepts the seed set of a network diffusion as input and builds its NI solution around *dominator trees.* A node u *dominates* a node w w.r.t. a seed node s if all paths from s to w pass through u. A *dominator tree* is a tree where each node is dominated by its ancestors. The *benefit* of removing a node is calculated as $\gamma(v) = 1 + \sum_{u \in \text{children of } v} \gamma(u) \cdot p_{vu}$, where p_{vu} stands for the probability that influence propagates along *any* path, approximated via the *most probable* path, from v to u.

DAVA iteratively removes the node of highest benefit and reconstructs the tree. In a DAVA variant, DAVA-fast, the tree is built only once and top-k nodes are selected based on their benefit in one go. A dominator tree is built in $O(E \log N)$ [8].

NetShape [11] immunizes a network via a convex relaxation approach, maximizing the eigen-drop of the network's integrated and symmetrized *Hazard matrix*, a matrix of a continuous integrable transition rate functions $\{\beta(u, v, t)\}_{u,v \in V}$, which indicate the probability that v is influenced by u at time t after u gets infected. The first eigenvalue bounds the expected spread of an infection. We apply NetShape as a heuristic for the IC model, setting the integral of the transition rate $\beta(u, v, t)$ as equal to the influence probability between u and v, and minimize the first eigenvalue by the projected subgradient descent method in the space of possible Hazard matrices *after* immunization, while setting the effect of immunizing u on the integrated hazard matrix element as 0, if u is a seed. The complexity is $O(\frac{1}{\epsilon^2} p_{max}^2 E \ln E)$, where p_{max} is the maximum propagation probability, and ϵ is a parameter affecting a step of subgradient descent.

3 METHODOLOGY

Consider a directed graph $G = (V, E)$ with set of nodes V and set of edges E. Each edge is associated with a probability of propagation. By the *independent cascade* model, a diffusion occurs in discrete time steps. In step t_0, a *seed set* $S \subset V$ becomes *activated.* Any node v activated in step t_i attempts to activate each of its inactive neighbors in step t_{i+1}, and succeeds by the probability associated with the edge from v to that neighbor. The process terminates when there are no more newly activated nodes. The *Network Immunization* (NI) problem calls to *block* a select set of k nodes $R \subseteq V \setminus S$ so as to minimize the expected *spread* of activated nodes, by a given seed set S in a graph G.

3.1 Algorithms

We compare six solutions to the NI problem in three categories:

- **Naïve**: *Degree* selects the top-k nodes with highest degree; *Random* selects k nodes uniformly at random.
- **Preemptive**: *NetShield* [2] and *Walk8* [13],
- **Data-Aware**: *NetShape* [11] and *DAVA* [21].

On NetShape, we use the default $\epsilon = 0.2$. As exact spread computation is #P-hard, we estimate spread with any solution via 1000 Monte-Carlo IC simulations. We use the original Matlab code of Walk8. As seeds cannot be blocked, we fetch $k + |S|$ nodes to be blocked with Walk8, ensuring that at least k nodes are blocked. We implemented all other algorithms in Python[1].

3.2 Data

We use both synthetic and real data obtained as follows.

3.2.1 Synthetic Data. We generated graphs of different properties using five models. By the *Erdős-Rényi model*, each edge is present with probability p; generated graphs have a low clustering coefficient and a binomial degree distribution. We refer to this generator as **Binomial**. We render the graph directed by selecting a random direction for each edge with 50% probability.

A **Gaussian Random Partition** (GRP) [1] selects edges as with Erdős-Rényi, but with a prior grouping, where group size follows a Gaussian distribution; it uses a probability value p_{in} for edges across nodes in the same group, and p_{out} otherwise, hence varying intra-group and inter-group density.

Watts Strogatz (WS) networks model self-organizing small-world systems [17], which have small average shortest path length, and are highly clustered, hence susceptible to infectious spread. The generator employs two parameters: l indicates how many nearest neighbors each node is joined with in a ring; p is a probability of edge rewiring, which induces disorder.

Barabási-Albert (BA) networks have both high clustering coefficients (as GRP and WS graphs) and power-law degree distribution, hence are better imitations of real-world social networks. We use the algorithm of Holme and Kim [5], which extends the original Barabási-Albert model, yet use the BA label as its basis; this algorithm randomly creates m edges for each node in a graph, and for created edge with a probability p adds an edge to one of its neighbors, thus creating a triangle.

Grid graphs have each node connected to four neighbors on a lattice. With this graph type, we explore the applicability of solutions on spatial graphs such as geosocial contact networks [22].

[1] Available at https://github.com/allogn/Network-Immunization

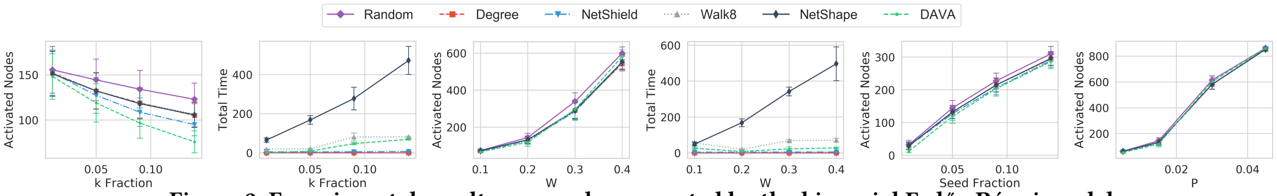


Figure 2: Experimental results on graphs generated by the binomial Erdős-Rényi model

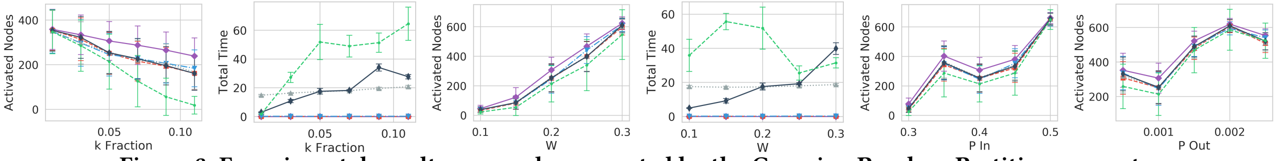


Figure 3: Experimental results on graphs generated by the Gaussian Random Partition generator

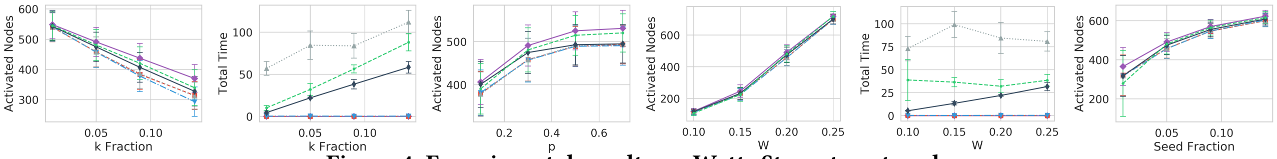


Figure 4: Experimental results on Watts Strogatz networks

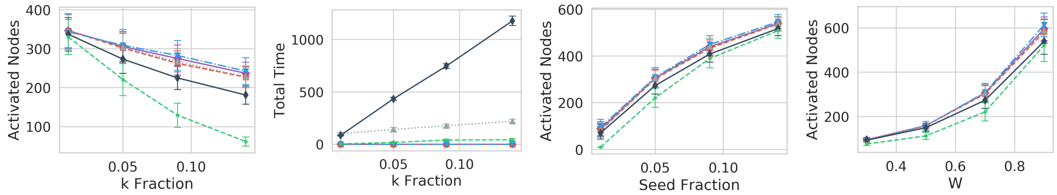


Figure 5: Experimental results on regular grids

Table 1 lists the default parameters for all models, where fractions $sf = |S|/|V|$ and $kf = k/|V|$. Figure 1 shows example graphs. All synthetic graphs have 1000 nodes, as in [16].

3.2.2 Real-World Datasets. We use 3 real-world graphs. Stanford and Gnutella, have been employed in related literature; a third, VK, provides a case of real-world propagation probabilities.

The **Stanford** data consists of pages and hyperlinks in the Stanford University website[2] [21]. The **Gnutella** peer-to-peer file sharing directed network is part of the SNAP dataset [9]. We use the biggest snapshot of 62586 nodes, with a diameter of 11 nodes and a clustering coeficient of 0.0055. It has been used in [11, 19, 21]. **vKontakte**[3] (VK) is a Russia-based social network of more than 500 million users[4]. Its public API allows to download information about public profiles, subscriptions, and posts. We fetch public posts of users to train the IC model.

3.3 Parameters

We consider **blocked node set size** k as a fraction of network size [15]. We employ *random* **seed selection** [3, 18, 21]; we pick 10 random seed sets, and show the mean and standard deviation of activated nodes. We choose **influence probabilities** uniformly at random from 0 to a maximum value W. We learn influence probabilities on the VK data using user posts as actions. We download 100 latest posts at the moment of publishing per user, resulting in 21M posts. Most posts are short, hence we can apply the same Natural Language Processing methods as for short messages. After preprocessing, we collected 536,073 non-empty messages belonging to non-isolated nodes in the VK graph, with median length of 11 words, std 187 and max 2977, leaving us with 3% of the original dataset. We define the closeness of actions by comparing the content of text messages, as in [7], to learn vector embeddings of short messages. We define term proximity as $p(w_2|w_1) = \frac{1}{|M|}\frac{c(w_1,w_2)}{c(w_1)}$, where M is a set of all posts with non-zero text content, $c(w)_m$ is the number of messages with w, and $c(w_1, w_2)$ is the number of messages with w_1 and w_2 present together. We learn stemmed term proximities and enrich the term frequency–inverse document frequency vectors of messages by increasing the probability of any words similar to words present in the message. We consider all message pairs with similarity above the median as similar. Scanning the action log to calculate the influence probability from a node u to any nove v as the ratio of successful reposts of similar messages. Filtering zero-probability edges, we select the largest component of 2.8K nodes and 40.9K edges as our VK network.

4 EXPERIMENTAL RESULTS

Here we present the results of our study. We set a timeout of 1h for all experiments for a single solver instance.

4.1 Synthetic Data

Figure 2 shows results with **Binomial** graphs. As the number of blocked nodes grows, DAVA's advantage of knowing the seeds becomes evident. Surprisingly, NetShield achieves better results than NetShape and Walk8 in this graph type. As the graph has a uniform structure, spectral-based algorithms do not perform well. This uniformity results in performance of algorithms not being dependent on the number of seeds and influence probability W. Still, as Figure 2c shows, with large W DAVA is slightly worse than preemptive approaches. DAVA assumes that the influence probability between two successive dominators in the dominator tree is equal to the probability along the shortest path. When there are many paths between two dominators, this assumption fails, hence the accuracy of the algorithm drops. We observe that NetShape is the least scalable algorithm.

[2] https://www.cise.ufl.edu/research/sparse/matrices/Gleich/ [3] http://vk.com/
[4] https://en.wikipedia.org/wiki/VKontakte

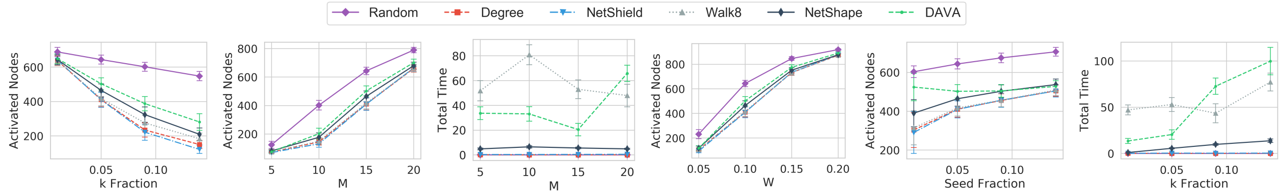


Figure 6: Experimental results on graphs generated by the Barabasi-Albert growth model

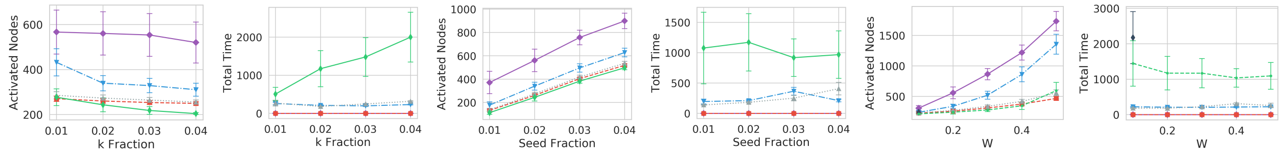


Figure 7: Experimental results on Stanford network

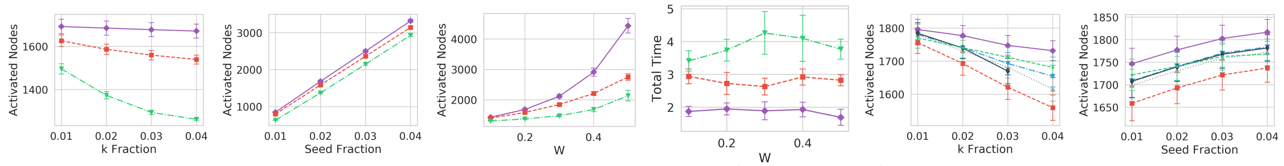


Figure 8: Experimental results on Gnutella (a-d) and VK (e-f) networks

Figure 3 shows results with **GRP** graphs. Again, the gap increases as k grows. DAVA achieves the best results on all parameters, except for the largest p_{out}. We observe that, as the inter-group probability p_{out} grows, DAVA shows slightly worse performance; in other words, as the graph forfeits its clustered structure, DAVA provides less accurate probability estimates.

Figure 4 shows results with **WS** graphs. Here, preemptive algorithms perform significantly better than DAVA, while the difference is accentuated as the number of blocked nodes k grows.

Figure 5 shows results with **Grid** graphs. All algorithms except DAVA fail to isolate seeds. NetShape outperforms other spectral approaches thanks to its data-awareness. Runtimes are similar to those in the Binomial case, with DAVA being sufficiently scalable.

Last, Figure 6 shows results with **BA** graphs; the degree heuristic and NetShield perform best. This result indicates that there are limits to the versatility of DAVA.

4.2 Real Data

Figure 7 shows results on the **Stanford** network. We employ the fast DAVA that builds a dominator tree only once so as to scale. Exploring a larger range of parameters than [21] reveals that DAVA performs similarly to the Degree heuristic, and slightly worse as W grows, due to the scale-free data topology. NetShape and Walk8 could not scale to such size. **Gnutella** has a more random topology than the Stanford network. Running on the 62K-node Gnutella snapshot, only fast-DAVA and baselines terminated within the time limit. Figure 8 shows the results, with DAVA reasserting its advantage. Our **VK** graph has high clustering coefficient and power-law degree distribution. Figure 8 shows that, on this data, DAVA is outperformed by preemptive methods. We deduce that, in real-world social networks, isolating diffusion sources is less critical than immunizing influence hubs.

5 CONCLUSIONS

We conducted an exhaustive experimental study of network immunization methods. We conclude that, while data-aware approaches stand out on networks with uniform topologies, spectral structure-based approaches are competitive on networks with power-law topologies. This result calls for further research.

REFERENCES

[1] Ulrik Brandes, Marco Gaertler, and Dorothea Wagner. 2003. Experiments on Graph Clustering Algorithms. In *ESA*. 568–579.

[2] Chen Chen, Hanghang Tong, B. Aditya Prakash, Charalampos E. Tsourakakis, Tina Eliassi-Rad, Christos Faloutsos, and Duen Horng Chau. 2016. Node Immunization on Large Graphs: Theory and Algorithms. *IEEE Trans. Knowl. Data Eng.* 28, 1 (2016), 113–126.

[3] Wen Cui, Xiaoqing Gong, Chen Liu, Dan Xu, Xiaojiang Chen, Dingyi Fang, Shaojie Tang, Fan Wu, and Guihai Chen. 2016. Node Immunization with Time-Sensitive Restrictions. *Sensors* 16, 12 (2016), 2141.

[4] Jacob Goldenberg, Barak Libai, and Eitan Muller. 2001. Talk of the Network: A Complex Systems Look at the Underlying Process of Word-of-Mouth. *Marketing Letters* 12, 3 (2001), 211–223.

[5] Petter Holme and Beom Jun Kim. 2002. Growing scale-free networks with tunable clustering. *Physical review E* 65, 2 (2002), 026107.

[6] David Kempe, Jon M. Kleinberg, and Éva Tardos. 2003. Maximizing the spread of influence through a social network. In *KDD*. 137–146.

[7] Ricardo Lage, Peter Dolog, and Martin Leginus. 2013. Vector Space Models for the Classification of Short Messages on SN Services. *WEBIST* (2013), 209–224.

[8] Thomas Lengauer and Robert Endre Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM TPLS* 1, 1 (1979), 121–141.

[9] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM TIST* 8, 1 (2016), 1.

[10] Yuchen Li, Ju Fan, Yanhao Wang, and Kian-Lee Tan. 2018. Influence Maximization on Social Graphs: A Survey. *IEEE TKDE* 30, 10 (2018), 1852–1872.

[11] Kevin Scaman, Argyris Kalogeratos, Luca Corinzia, and Nicolas Vayatis. 2017. A Spectral Method for Activity Shaping in Continuous-Time Information Cascades. *CoRR* abs/1709.05231 (2017).

[12] Paulo Shakarian, Abhinav Bhatnagar, Ashkan Aleali, Elham Shaabani, and Ruocheng Guo. 2015. The independent cascade and linear threshold models. In *Diffusion in Social Networks*. Springer, 35–48.

[13] Juvaria Tariq, Muhammad Ahmad, Imdadullah Khan, and Mudassir Shabbir. 2017. Scalable Approximation Algorithm for Network Immunization. In *PACIS*.

[14] V. Tejaswi, P. V. Bindu, and P. Santhi Thilagam. 2016. Diffusion models and approaches for influence maximization in social networks. *ICACCI* (2016).

[15] Biao Wang, Ge Chen, Luoyi Fu, Li Song, and Xinbing Wang. 2017. DRIMUX: Dynamic rumor influence minimization with user experience in social networks. *IEEE Trans. Knowl. Data Eng.* 29, 10 (2017), 2168–2181.

[16] Yang Wang, Deepayan Chakrabarti, Chenxi Wang, and Christos Faloutsos. 2003. Epidemic Spreading in Real Networks: An Eigenvalue Viewpoint. In *22nd Symposium on Reliable Distributed Systems*. 25–34.

[17] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of "small-world" networks. *Nature* 393, 6684 (1998), 440.

[18] Arie Wahyu Wijayanto and Tsuyoshi Murata. 2018. Pre-emptive spectral graph protection strategies on multiplex social networks. *Applied Network Science* 3, 1 (2018), 5.

[19] Dingda Yang, Xiangwen Liao, Huawei Shen, Xueqi Cheng, and Guolong Chen. 2018. Dynamic node immunization for restraint of harmful information diffusion in social networks. *Physica A* 503 (2018), 640–649.

[20] Yao Zhang. 2017. *Optimizing and Understanding Network Structure for Diffusion*. Ph.D. Dissertation. Virginia Tech.

[21] Yao Zhang and B. Aditya Prakash. 2015. Data-Aware Vaccine Allocation Over Large Networks. *TKDD* 10, 2, 20:1–20:32.

[22] Yao Zhang, Arvind Ramanathan, Anil Vullikanti, Laura L. Pullum, and B. Aditya Prakash. 2017. Data-Driven Immunization. In *ICDM*. 615–624.