SAMUEL: A Sharing-based Approach to processing Multiple SPARQL Queries with MapReduce*

InA Kim
Dep't of Computer Engineering
Chungnam National University
Daejeon, 34134, Korea
dodary0214@gmail.com

Kyong-Ha Lee†
Korea Institute of Science and Technology Information
Daejeon, 34141, Korea
bart7449@gmail.com

Kyu-Chul Lee
Dep't of Computer Engineering
Chungnam National University
Daejeon, 34134, Korea
kclee@cnu.ac.kr

ABSTRACT

The volume of RDF data is now growing tremendously. It is thus considered prudent to store and process massive RDF data with distributed SPARQL engines instead of relying on a single-machine system. Many sophisticated index and partitioning schemes have also been proposed to support SPARQL query evaluations. However, existing SPARQL engines have mainly followed *one-at-a-time* scheme so that query evaluation is focused only on processing each query separately. We showcase *SAMUEL*, a distributed SPARQL engine that simultaneously evaluates many SPARQL queries for a massive RDF dataset with MapReduce. SAMUEL provides an efficient optimization algorithm to evaluate many SPARQL queries simultaneously in a shared and balanced way. Extensive experiments present that without any sophisticated partitioning or index mechanisms, our approach significantly outperforms other MapReduce-based SPARQL engines as well as an *ad-hoc* query engine equipped with various indexes and partitioning tools for evaluating multiple SPARQL queries.

1 INTRODUCTION

The Resource Description Framework (RDF) is a versatile graph data model that enables users to express facts and their relationships in the form of triples ⟨ ***S**ubject*, ***P**redicate*, ***O**bject* ⟩ where a predicate (***P***) expresses a relationship between a subject (***S***) and an object (***O***) [2]. Many knowledge bases are now defined in the RDF format, *e.g.*, DBpedia [16], Bio2RDF [6] and UniProt [7] and they shape a very large set of graphs having millions of triples interlinked each other and are queried by using SPARQL query language [1]. It has been a major challenge to find subgraph patterns described in given SPARQL queries from a massive set of RDF graphs with supporting both efficiency and scalability. To address the issue, numerous SPARQL query engines have been devised based on MapReduce [15] or their proprietary distributed architectures [3]. Meanwhile, multiple SPARQL queries often need to be evaluated together as the queries can be prepared before runtime in some scenarios [13, 20]. Motivated by these facts and our previous work on XML data [8], we devise a MapReduce-based SPARQL engine called *SAMUEL*, which simultaneously evaluates many SPARQL queries in a *shared* and *balanced* way. Major features of SAMUEL are as follows :

Support of parallel multi-SPARQL query processing
SAMUEL provides an efficient means to process a massive set of RDF data in parallel. It does not require any sophisticated partitioning or index mechanisms for SPARQL query evaluation. Nonetheless, SAMUEL easily outperforms other MapReduce-based distributed SPARQL engines by simultaneously evaluating multiple SPARQL queries with a short series of MapReduce jobs.

Sharing input scans and intermediate results
SAMUEL enables computing nodes to share input scans and their intermediate results with each other. While joining RDF triples for evaluating queries, a group of join operations assigned to each reducer share their input and intermediate results associated with distinct subquery patterns that multiple queries commonly contain. Consequently, it saves many I/Os by removing many redundant subquery matchings while evaluating queries.

Runtime load balancing and multi-query optimization
In a distributed system, a straggling task delays overall job execution. This problem deteriorates with the use of MapReduce since MapReduce typically enforces barrier synchronization between Map and Reduce tasks. MapReduce's native runtime scheduling algorithm also proved to be inefficient especially in the reduce stage [12, 15]. To address the issue, SAMUEL rather exploits *dynamic shuffling* scheme that balances workloads across reducers at each MapReduce job in accordance with the cardinality of RDF triples that each reducer consumes for joining. It decomposes a given set of SPARQL queries into distinct triple patterns and then *gradually* builds a bushy query plan tree that covers all RDF join operations required to evaluate the given SPARQL queries. Since processing join operations at each level in the query plan tree requires a single M/R job, SAMUEL always tries to build a bushy plan tree with the lowest possible height. Then, it partitions the join operations at each level into n groups and then assigns them to n reducers. To balance workloads across reducers that actually perform join operations, SAMUEL computes the cost of each join operation at each level of the plan tree before actual joining. It then assigns join operations into reducers at each level such that every reducer has the same overall cost of join operations and the lowest communication cost by avoiding redundant RDF triples being transferred to multiple reducers so far as possible.

The rest of this paper is organized as follows. Section 2 introduces previous studies directly related to our work. Section 3 describes how we evaluate multiple SPARQL queries together at a time and our system architecture that provides workload balancing as well as multi-SPARQL query processing. Section 4 presents our demonstration scenario including the major results of performance evaluations.

2 RELATED WORK

Numerous distributed SPARQL engines have been devised to store RDF data and to evaluate SPARQL queries so far [3, 9, 11, 14, 19, 21, 22]. Readers are referred to a recent survey on distributed SPARQL query engines [3]. These systems fall into

*This work is equally supported by KISTI and NRF grant (No. NRF-2015R1A2A2A01004879) funded by the Korea government.
†Corresponding author

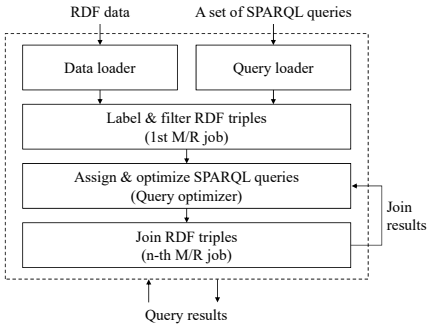


Figure 1: System architecture

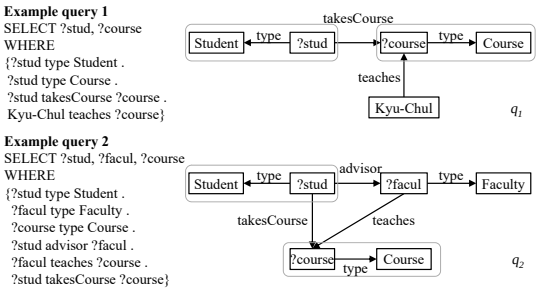


Figure 2: Example of common subquery patterns in two SPARQL queries

two categories in the aspect which framework these systems rely on : (i) general-purpose framework such as MapReduce and (ii) specialized RDF systems. In this paper, we only deal with MapReduce-based RDF engines due to the limit of space.

MapReduce-based RDF engines

MapReduce is a popular parallel processing tool that provides high scalability as well as simple abstraction. Therefore, many studies have been done with MapReduce although it has some inherent limitations [15]. It is noteworthy that most M/R-based RDF engines do not well utilize index mechanisms since MapReduce is originally devised for *batch processing* rather than *ad hoc queries*. Indexes are considered inefficient for batch jobs due to their expensive building cost only for one-time use.

SHARD [21] is a distributed RDF engine built with MapReduce. All the RDF triples are stored in a single file in HDFS and RDF triples are hash-partitioned across nodes. SPARQL queries are evaluated as M/R job iterations. A subquery pattern is evaluated within a M/R job and its results are transferred to a subsequent M/R job. SHARD does not utilize any indexing scheme so that it needs to scan the entire dataset for query evaluation. HadoopRDF [11] is another RDF engine built with MapReduce. It partitions RDF triples into multiple files in the way that each file contains all the triples that have the same predicate. It also locates non-duplicate binary joins together into a M/R job to minimize MR job iterations. SHAPE [14] also uses MapReduce but it uses semantic hash partitioning scheme to group vertices on the basis of URI hierarchy for improving data locality. However, these systems suffer from workload imbalance if a few triple patterns dominates overall data distribution. H2RDF+ [19] is a distributed engine based on both MapReduce and HBase, the open sourced version of BigTable. It materializes combinations of RDF triples and store them into HBase tables in order to utilize some features of HBase, *e.g.*, sorted keys and range-partitioned tables based on the keys. However, H2RDF still join RDF triples with M/R job iterations thus a complex queries can expand the iterations. CliqueSquare [9] partitions RDF triple in three ways, by hashing them on the basis of subject, predicate and object for increasing data locality. It exploits the data replication feature of HDFS to locally process all first-level joins on each node. It also tries to minimize the number of M/R job iterations for RDF joins with a shallow query plan tree and multi-way join operations.

Multi-query optimizations

Multi-query optimization on MapReduce is regarded as an extended version of the classical *job-shop problem* [5] that has a long history. A few studies related to multi-query optimization on MapReduce have been reported in the literature [18, 23] in the context of relational processing. They provide generalized grouping techniques that merge multiple jobs into a single job thereby enabling the merged jobs to share input scans and common mapped outputs. Some studies address the issue in the context of SPARQL queries [13, 20]. Multi-query optimization for SPARQL is also proven to be *NP-hard* so that they rather propose heuristic algorithms that partition a set of queries into groups such that queries in each group can be optimized together. However, their algorithms [13, 20] are focused only on working on a single-machine engine rather than a distributed environment such as MapReduce. Therefore, all the approaches are hard to be directly applied to distributed SPARQL engines. On the contrary, we focus on multi-query optimization on MapReduce-based distributed SPARQL engines. To the best of our knowledge, this is the first work that provides a solution for the multi-query optimization problem in MapReduce-based SPARQL engines.

3 MULTI SPARQL QUERY PROCESSING

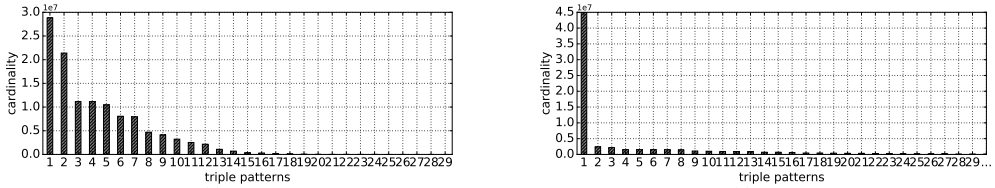
SAMUEL performs its query evaluation in three phases: (i) preprocessing phase, (ii) labeling & filtering phase, and (iii) iterative joining phase (see Fig. 1). In the preprocessing phase, RDF data and a set of SPARQL queries are loaded on HDFS. In our system RDF triples are simply stored in a single file where each triple is recorded as a single line as it is, except that redundant URIs and labels are substituted by unique IDs for saving storage volume and I/Os. Note that we do not use any partitioning or index mechanisms for RDF data since we only show the effect of our approach, distinguished from the benefits that we can get with the mechanisms. Query loader decomposes SPARQL queries into a set of distinct triple patterns, and then associates each triple pattern with a set of queries that contains the triple pattern like Fig. 2. It also builds an in-memory radix tree where a tree node at each level represents a unique ***S***, ***P***, and ***O*** of triple patterns, respectively. With the radix tree, SAMUEL rapidly finds triple patterns matched to an input RDF triple while query evaluation.

The labeling & filtering phase is implemented with a single M/R job. In the phase, RDF triples are labeled with IDs for their corresponding triple patterns by traversing the radix tree and also are filtered out if they have no corresponding triple patterns. This work has an analogy to filtering stream data. Since reducers aggregate RDF triples that has the same pattern, it allows us to easily compute the cardinality of each triple pattern in the phase.

Based on the cardinality information, our query optimizer builds a global query plan tree that has the lowest possible height to minimize the number of M/R job iterations for joining RDF triples. Redundant join operations are removed as each join operation is shared by multiple queries. When assigning binary RDF join operations to reducers, we consider both join cost and transmission cost to minimize and balance workloads across reducers. It is achieved by summing the cost of every RDF join operation

Dataset	LUBM-100M	WatDiv-1M	WatDiv-10M	WatDiv-100M	WatDiv-1B
# of triples in total	138,280,374	1,098,468	10,989,614	109,795,305	1,098,717,244
# of distinct triples	133,573,856	1,085,817	10,906,204	109,051,965	1,091,667,092
# of distinct patterns matched to queries (1,000 queries for WatDiv and 14 queries for LUBM)	23	361	355	358	355
Data size(GB)	23.02	0.14	1.45	14.63	148.23

Table 1: Statistics of RDF Datasets



(a) LUBM (b) WatDiv

Figure 3: Data skewness in two datasets

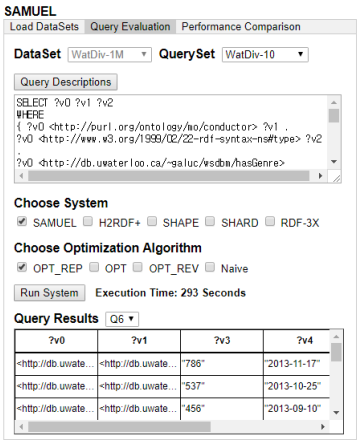


Figure 4: SAMUEL GUI

assigned to each reducer and the cost of each RDF join operation, implemented with binary hash join, is computed by summing the sizes of two input RDF triple pattern lists. Triples are grouped on the basis of distinct triple patterns and the groups are ordered in a descending order of cardinality. We devise a heuristic optimization algorithm based on *first-fit decreasing* scheme whose input is a set of ordered lists of the distinct triple pattern groups. Some triple patterns have very high cardinality as shown in Fig 3. We thus allow splitting a very large triple pattern list into multiple small lists for better workload balance. Note that our global query plan tree is gradually built. That is, the n-th level in the tree is computed right before running the n-th joining phase. In each joining phase, implemented by another M/R job, mappers read grouped RDF triples and tag reducer IDs to the triples according to the global plan tree. Since mapped outputs are shuffled by intermediate keys, triples tagged by the same reducer ID go to the same reducer together and are then joined. Again, each reducer computes the cardinality of its joined outputs and the cardinality information and joined results are stored in HDFS. Then, our optimizer repeats building the $n+1$-th level of the plan tree with the cardinality of the results of the n-th joining phase.

4 DEMONSTRATION

During the demonstration, audience are invited to compare our system with other distributed SPARQL engines based on the MapReduce framework, interacting with the system to run queries and to check the influences of optimization techniques.

Hardware setup: We implemented our system with Hadoop version 1.2.1. SAMUEL as well as compared systems were installed and run on a cluster of 15 nodes, each of which was equipped with an Xeon E5-2620 2.1GHz CPU, 64GB memory and an 1TB 7200RPM HDD, running on Ubuntu 12.04. All the nodes were connected via Gigabit switching hub and a node is designated as a master node. We basically used the same settings for our cluster for fair comparison. However, the settings were sometimes tuned for showing the best performance of compared systems.

Dataset: We used two datasets, which had been widely used for measuring SPARQL engines in the literature [4, 10]. Table 1 and Figure 3 present the statistics for datasets used in our demonstration and their data distributions. Both datasets exhibited high skewness in their data distributions in that only a few of triple patterns dominated most of the data distributions (see Fig. 3).

Compared systems: In our demonstration, we compare our system with RDF-3X [17], a single-machine SPARQL engine that utilizes various indexes, and three other MapReduce-based systems, *i.e.,* SHARD [21], SHAPE [14], and H2RDF+ [19]. For evaluation, the performance of each system was averaged over five runs excluding the maximum and minimum values.

Demo scenarios and interaction

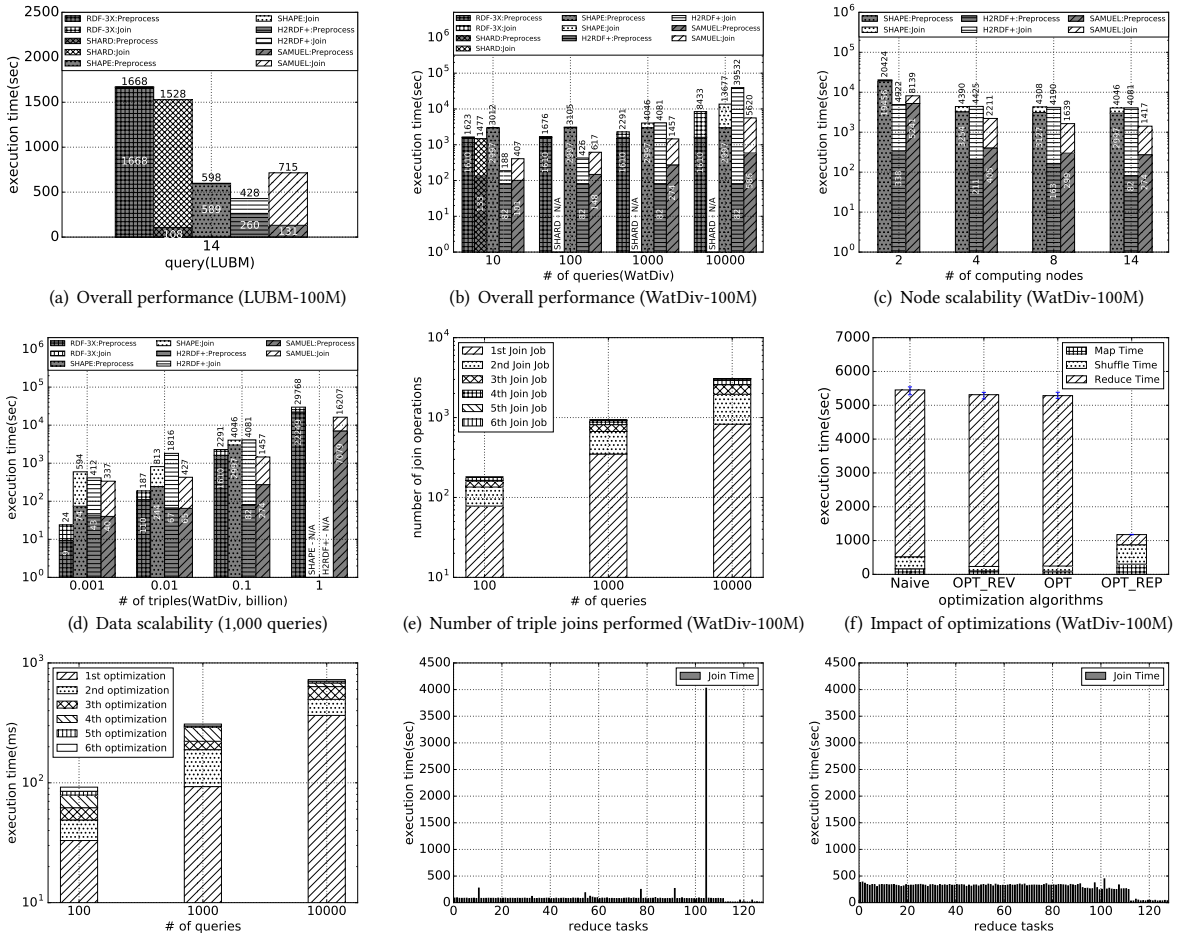
We provide a user interface shown in Fig. 4 to demonstrate the performance of SAMUEL using large-scale RDF datasets, *i.e.,* LUBM and WatDiv. In our demonstration, we however use only a few fractions of the two datasets due to the limited time and computing resources. However, we still present our evaluation results performed with all the datasets (see Fig 5). Currently, SAMUEL supports a subset of SPARQL language, *i.e.,* basic graph pattern matching. In our demonstration, users will be given a list of SPARQL queries generated from WatDiv for the datasets in Table 1. Users are also allowed to load their own queries and RDF data into the system and run the queries themselves. During the processing, users will be explained with Hadoop GUI and our own UI how our system processes multiple SPARQL queries simultaneously. Users will also check how features of SAMUEL affect the overall performance as they turn on and off the features, *i.e.,* sharing input scan and filtered solutions, optimization policies, and so on.

REFERENCES

[1] 2008. SPARQL Query Language for RDF. https://www.w3.org/TR/rdf-sparql-query/. (2008).

[2] 2014. Resource Description Framework (RDF). https://www.w3.org/RDF/. (2014).

[3] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. 2017. A Survey and Experimental Comparison of Distributed SPARQL Engines for



(a) Overall performance (LUBM-100M)
(b) Overall performance (WatDiv-100M)
(c) Node scalability (WatDiv-100M)
(d) Data scalability (1,000 queries)
(e) Number of triple joins performed (WatDiv-100M)
(f) Impact of optimizations (WatDiv-100M)
(g) Elapsed time for multi-query optimization
(h) Reducer tasks without multi-query optimization
(i) Reducer tasks with multi-query optimization

Figure 5: Performance evaluation

Very Large RDF Data. *Proceedings of the VLDB Endowment* 10, 13 (2017).

[4] Güneş Aluç, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee. 2014. Diversified stress testing of RDF data management systems. In *International Semantic Web Conference*. Springer, 197–212.

[5] David Applegate and William Cook. 1991. A computational study of the job-shop scheduling problem. *ORSA Journal on computing* 3, 2 (1991), 149–156.

[6] François Belleau, Marc-Alexandre Nolin, Nicole Tourigny, Philippe Rigault, and Jean Morissette. 2008. Bio2RDF: towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics* 41, 5 (2008), 706–716.

[7] Emmanuel Boutet, Damien Lieberherr, Michael Tognolli, Michel Schneider, and Amos Bairoch. 2007. UniProtKB/Swiss-Prot: the manually annotated section of the UniProt KnowledgeBase. *Plant bioinformatics: methods and protocols* (2007), 89–112.

[8] Hyebong Choi, Kyong-Ha Lee, Soo-Hyong Kim, Yoon-Joon Lee, and Bongki Moon. 2012. HadoopXML: a suite for parallel processing of massive XML data with multiple twig pattern queries. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 2737–2739.

[9] François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge-Arnulfo Quiané-Ruiz, and Stamatis Zampetakis. 2015. Cliquesquare: Flat plans for massively parallel RDF queries. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 771–782.

[10] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web* 3, 2 (2005), 158–182.

[11] Mohammad Husain, James McGlothlin, Mohammad M Masud, Latifur Khan, and Bhavani M Thuraisingham. 2011. Heuristics-based query processing for large RDF graphs using cloud computing. *IEEE Transactions on Knowledge and Data Engineering* 23, 9 (2011), 1312–1327.

[12] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2012. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 25–36.

[13] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. 2012. Scalable multi-query optimization for SPARQL. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 666–677.

[14] Kisung Lee and Ling Liu. 2013. Scaling queries over big RDF graphs with semantic hash partitioning. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1894–1905.

[15] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. 2012. Parallel data processing with MapReduce: a survey. *AcM SIGMOD Record* 40, 4 (2012), 11–20.

[16] 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.

[17] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal?The International Journal on Very Large Data Bases* 19, 1 (2010), 91–113.

[18] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. 2010. MRShare: sharing across multiple queries in MapReduce. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 494–505.

[19] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. 2013. H2RDF+: High-performance distributed joins over large-scale RDF graphs. In *Big Data, 2013 IEEE International Conference on*. IEEE, 255–263.

[20] Xuguang Ren and Junhu Wang. 2016. Multi-query optimization for subgraph isomorphism search. *Proceedings of the VLDB Endowment* 10, 3 (2016), 121–132.

[21] Kurt Rohloff and Richard E Schantz. 2010. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. In *Programming Support Innovations for Emerging Distributed Applications*. ACM, 4.

[22] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. 2016. S2RDF: RDF querying with SPARQL on spark. *Proceedings of the VLDB Endowment* 9, 10 (2016), 804–815.

[23] Guoping Wang and Chee-Yong Chan. 2013. Multi-query optimization in mapreduce framework. *Proceedings of the VLDB Endowment* 7, 3 (2013), 145–156.