ligDB—Online Query Processing Without (almost) any Storage*

Evica Milchevski
University of Kaiserslautern
Kaiserslautern, Germany
milchevski@cs.uni-kl.de

Sebastian Michel
University of Kaiserslautern
Kaiserslautern, Germany
smichel@cs.uni-kl.de

"By letting it go it all gets done.
The world is won by those who let it go."
(Laozi)

ABSTRACT

In the big-data era data is arriving at such a high pace and volume that data exploration and querying can only be feasible if data loading and indexing happens reasonably quick—if at all. Recent research on handling large scientific data suggests ignoring any database indexing or even data-loading processing steps but rather turns toward processing raw data as it is handed in by scientists, manually or by semi-automated means—if needed in multiple, iterative steps. In this paper, we describe the anatomy and research challenges of a system coined LIGDB[1] that is operating purely on incomplete database tables, JSON documents, or sets of SPO triplets that are being filled over time. There is no data stored per se; the only data stored is stemming from previously posed queries over the stream of arriving data; kept as long as it is used by forthcoming queries and otherwise evicted. A key point is that velocity dimension of "big data" allows queries being processed as they are posted, with higher-level queries processed on historic query results (views) and live data. Data that is not touched by any posted query is immediately discarded.

1. INTRODUCTION

The big data challenge is about making sense of large amounts of digital content, in a timely fashion, for business intelligence or other forms of knowledge-seeking tasks. Data is generated at various sites and continuously growing; for instance through crowdsourcing missing entries of a database table, by contributing facts in Wikipedia pages, or by mentioning entity-centric properties in Tweets. The common problem dimensions imposed by the properties of what is commonly referred to as "Big Data" are best sketched by the "4Vs", most prominently volume and velocity (and variety and veracity). Data too often arrives in big volumes at high bandwidth, too much to apply traditional store-first, process-later approaches. With the advance of technology the volume and velocity of data will just keep growing, requiring a drastic shift in the current ways of data storing and processing. Recent works on handling scientific data [25, 4] have already emphasized the huge overhead of (re-)indexing for one-time queries over quickly changing data and propose processing queries on raw input data, if required through multiple iterations (parsings); or using access-driven data fetching [1]. Other attempts aim at explorative data analysis, with tools guiding the querying process and supporting approximative query results with tunable time budgets [29].

LIGDB represents a radically new approach to handling the data deluge: it is designed to let go data (hence its name) that is not required by any query and only the submission of a query triggers data gathering and processing. No other data is stored, unlike earlier works, like the ones mentioned above, where the "entire" data is available and waiting to be queried (if needed). In LIGDB only the results of queries are stored/cached, hence are treated as data by subsequent queries, until replaced in the store if not used. Think of queries that aim at finding restaurant ratings/critics or the temperature in a certain city, information on who is the fiancé of Angelina Jolie, the most promising stock to invest in, or the hottest posts in Facebook. In the massive amount of digital content that is created, commented-on, or simply re-invented (i.e., replicated) at literally every second, why should we store everything that has been produced thus far, when those questions can potentially be answered on the fly. That is, it appears possible that data is arriving at such a volume and velocity that queries can actually *wait for it.*

Once fired, queries are getting filled with result tuples until a user-specific quality or response-time criteria is met and report back to users. Ultimately, queries and their results can stay in the system and form a basis of further querying; essentially forming, traditionally speaking, a purely view-based database system over an update/data stream.

The following are the signature characteristics of LIGDB:

- **No Storage:** no data is stored per-se
- **No Schema:** data arrives in form of SPO triplets or JSON documents
- **Query-based:** queries trigger data gathering to answer the query

*This work has been partially supported by the German Research Foundation in project MI 1794/1-1.

[1]The prefix lig in LIGDB stands for "let it go" and *to lig* can also mean *to live on others*; both meanings capturing together the two corner stones of LIGDB

10.5441/002/edbt.2015.69

- **Result Caching:** query results are cached, statically or dynamically (as views to be updated), and evicted at some point
- **Live or Historic Results**: current queries can re-use, entirely or partially, historic query results
- **Query by Example:** queries are semi-automatically formulated, through ontological concepts and the query by example paradigm. For instance by specifying previously obtained results and waiting for the updated ones.

A system such as LIGDB is ideally able to behave, from a user perspective, like a traditional data management system: it can serve ad-hoc explorative queries and analytical queries. On the other hand, it is clear that due to the forgetful data handling, it also puts natural limits on its applicability to more traditional scenarios.

In order to build such a system, various core research areas have to be integrated. In fact, there is work on almost all aspects of this system in separation. Like data integration, result caching, view maintenance, and processing queries over data streams. Old concepts like query by example are revived to allow users querying large amounts of very sparsely (if at all) described data; a general problem for which also ontologies can help. In this work, we sketch the overall idea of LIGDB, the characteristics of its core system components, application cases, and challenges.

This paper is organized as follows. In Section 2, we review recent work on processing data on raw files or in a lazy fashion along with an overview of fundamental techniques. In Section 3 the overall architecture and core components of LIGDB are presented; including a sketch of a possible implementation. In Section 4 we discuss challenges that need to be addressed. We summarize the paper with a brief conclusion in Section 5.

2. RELATED WORK

Kersten et al. [29] envision the next generation database systems as systems that shift away from the goals of completeness and correctness, aiming towards explorative and interactive data analysis. They propose several research directions: a one-minute database kernel, producing query results within a limited time; multi-scale queries—breaking the query into multiple smaller stages; post processing of the result sets; query morphing—creating variations of the issued query and finding their results as well; and providing query suggestions for more effective data exploration.

Lazy processing: Cheung et al. [15] present Sloth, a system that extends lazy evaluation with the purpose of reducing network latency in web applications. Using dynamic program analysis, they identify the queries to be issued by the application, batch them, and postpone their processing until it is absolutely necessary. Only then they execute batches of queries, thus reducing network latency. Kargin et al. [28] propose lazy ETL, a technique for extracting, transforming and loading in a data warehouse only data that is necessary to answer the issued query. Initially, only the metadata from the queries are loaded into the warehouse. When the user issues a query, the selection predicate is imposed on the metadata to decide which files need to be loaded; the data is transformed using relational views on the extracted data; and stored into the internal data structure of the warehouse.

NoDB [25, 4] avoids data indexing and instead operates on raw csv files, if required through multiple iterations. They argue that the initial indexing in traditional DBMS poses a huge overhead, too high for one-time queries over frequently changing data. However, they still assume that data is residing on disk or another storage container and waiting to be queried. Similarly, modern SQL-on-HDFS engines like Google Dremel [35] and Cloudera Impala [16] do not own data, but execute queries over files stored on a distributed file system. Abouzied et al. [1] propose to load data for processing based on queries. While the query-driven nature is similar to what we propose with LIGDB, we do not assume that data is stored anywhere and waiting to be accessed. Instead, LIGDB operates solely on live data, as handed in by scientists or as otherwise generated. When data arrives there is a one-time chance that it gets consumed—or it is eliminated otherwise.

Approximate query processing: In [3] the authors address the problem of overestimating or underestimating the error in sampling-based approximate query processing (S-AQP). They show that existing approximation methods used by S-AQP can sometimes show errors which are overestimated or underestimated. They propose a diagnosis technique to estimate the failure of error estimation for the query, while still providing an interactive mode of answering the queries. Using the diagnostics tool, the DB system switches to non-approximate methods for answering queries, when the diagnostic tool sees that the error estimation will be unreliable.

Willis et al. [32] consider partial query results from a different perspective, i.e., partial-results generation in a case of data-access failures. They provide a taxonomy of partial-results, classifying them based on the cardinality and the correctness of the partial result with respect to the true result. They further discuss how to assess the correctness and cardinality class of partial results, and whether this can be done at a finer granularity level (e.g., per row or per column).

Ge and Golab [20] propose a framework for maintaining data structures in main-memory, sliding-window data warehouses. The proposed framework aims at combining the benefits of existing sliding-window maintenance techniques, namely single data structure for all-time window partition, and one data structure per partition.

Publish/subscribe is a widely used communication paradigm for large-scale distributed systems. In the publish subscribe interaction scheme, subscribers register for an event and publishers publish events. Subscribers are asynchronously notified when an event of interest is published. There are many variants of this system. Eugster et al. [17] identify and summarize the commonalities and differences of different publish-subscribe systems. Vargas et al. [43] propose an architecture for integrating databases with a publish-subscribe system. They integrate Hermes, a publish subscribe system, with PostgreSQL, allowing users to subscribe to events denoted by changes that occur in the underlaying database

Data Streams and Continuous Queries: Data streams have emerged as an answer to applications that do not fit the traditional data model. Golab and Özsu [22] summarize data-stream management systems from different perspectives, and further identify possible research challenges. PSoup [13] is a system for streaming queries over streaming data. PSoup treats data and queries symmetrically: when new query is registered to the system, it is probed over the historical data to find possible results. Similarly, when a new streaming data tuple arrives to the system it is first probed over the pending queries; the result is materialized; and stored in the system. A result is returned only when a user requires one, and then only the results belonging to a time

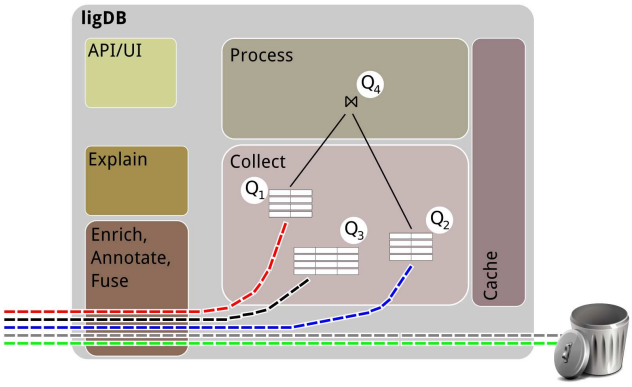

Figure 1: high level architecture

window are returned. PSoup supports joins over different data streams as well, by storing each stream in a corresponding data SteM. Bonet et al. [11, 12] discuss streaming data and the kinds of queries that can be issued. They describe types of queries over streaming data—snapshot and long-running queries. Snapshot queries are defined as those over a set of streaming sources but in a single point in time in the past. On the other hand, they describe long-running queries as queries that continually return answers as new data arrives. LIGDB is not meant to be a data-stream processing system, rather it is supposed to act like a non-streaming, traditional database system, from a user perspective; with the difference that no data is stored and only query results are cached (treated as data or, better, materialized views) and can be queried as normal "data". Terry et al. [42] propose the concept of continuous queries; a permanent query for which the user gets results whenever there is a matching tuple. They further define monotone continuous queries as queries which result is strictly non decreasing over time.

3. LIGDB ARCHITECTURE

The high-level architecture of LIGDB is shown in Figure 1. The system smoothly blends data processing on raw input streams with traditional query processing on top of data gathered in a query-driven way. We believe that the query-driven data gathering and subsequent processing in LIGDB is very reasonable as data is not created by a "big bang" but is being built up over time, for instance, crowdsourced [19], measured by sensors, or created by user actions in social networks or Wikipedia.

3.1 Core Components

LIGDB has the following core components (cf., Figure 1):

- **Enrich, Annotate, Fuse:** We consider small *data fragments* in form of sets of object-oriented (entity centric) key-value pairs as the generic data format. For instance, in form of JSON objects; however, ideally in form of full-fledged relational tables with clear schemas. In general, input data can be further annotated/enriched, e.g., through object/entity disambiguation, cleaned or standardized to general concepts using ontologies. This is very generic, and does not assume any fixed schema with full-fledged relational tables and foreign-key constraints to be present.
- **Collect:** As there is no data stored per se, initially queries are purely data-gathering queries, i.e., they define materialized views, until the query is answered to a satisfactory level. Subsequently, the query results are returned. Results can be statically cached or the query can remain in the system and needs to be continuously updated. This forms a data basis in the otherwise empty-storage LIGDB.
- **Cache:** The cache is responsible to handle the previously mentioned results of historic queries. It has space, time, and runtime constraints. That is, it has limited (in-memory) storage, evicts too-old-to-be-useful query results, and limits the amount of views to be continuously maintained [9, 30] and not frequently used.
- **Process:** Once queries are present in the system, either running or cached, newly posted queries can (fully or partially) reuse previous query results. When the query cannot be fully answered by historic results, the entire query or parts of it are posted in the data-collector component over the data stream.
- **Explain:** Querying heterogeneous, schema-free (or not well understood ones) data requires mechanisms that guide the query-phrasing process. We believe that the querying process should be driven by either examples and/or be guided by general-purpose and specialized ontologies (that can be uploaded in the system).
- **API/UI:** Users can assemble queries using the above Explain component and push them to the system. If they can solely be answered based on cached data, the result is returned instantaneous. Otherwise, the query is registered and necessary data is gathered. The user is notified if the query is answered to a satisfiable degree.

3.2 Object/Entity-Centric Input Fragments

As input, we specifically consider data represented as generic JSON objects, i.e., a bag of possibly nested key-value pairs. These might or might not come with globally unique identifiers that allows to gather data specifically related to one unique object. If ids are not given directly, to accumulate key-value pairs for the same object (entity), methods for determining the correct entity based on the data context are required. Consider for instance the JSON object in Figure 2 that gives details on a business in Phoenix, AZ, as given by the Yelp academic dataset [45]. Portals like Yelp that harness crowd input are an excellent example why data is not created in one time point, but evolves. For instance, business categories, here "Food" and "Grocery" might be added later on, review counts grow over time, and the field "open" can change over time, too. Other information such as "city" and "state" in this example are redundant, here with "full_address", but might be useful for querying. To extract such information and to bring their naming to a common ground is part of the *Enrich, Annotate, Fuse* component. There has been many recent works on understanding Web tables [33, 44] and on matching and disambiguating named entities [31, 24] that can be harnessed in LIGDB.

3.3 Query Types and Query Publishing

Per se, there is no restriction on the kind of queries that should be processed in LIGDB. Apparantly, however, in a scenario like the one addressed, where it is reasonable to throw away any historic, unused data, queries will likely be mostly of analytical, explorative nature.

The most basic query in LIGDB are so called data gathering queries that extract information out of the underlying data stream. Since it is not reasonable to assume a fixed schema, as described, one way to work is with examples/templates, for instance,

```
{
  "business_id": "usAsSV36QmUej8–yvN-dg",
  "full_address": "845 W SouthernAve Phoenix,
                                    AZ 85041",
  "open": true,
  "categories":["Food", "Grocery"],
  "city" :"Phoenix",
  "review_count": 5,
  "name": "Food City".
  "state": "AZ",
  "type": "business"
}
```

Figure 2: Excerpt of a JSON object of the Yelp dataset

```
{
  "business_id": ?
  "name" :?,
  "city" :"Chicago",
}
```

that are simple *selection&projection queries with predicates*. Users can also re-use existing query results to gather additional information, or to refresh/enrich previous results.

It is clear that the above samples and the focus on JSON in this work is not a restriction; likewise, we could also consider RDF Subject-Predicate-Object (SPO) triples and queries expressed in SPARQL, or even data integration/fusion into relational tables and querying with SQL. Still, it is hard to phrase meaningful queries. Below, we review some works on query advisors and ontologies [41, 7].

More advanced queries can be in a form of (top-k) aggregation queries [27], arbitrary join queries (particularly semi-join–based data pruning appears very useful), queries that gather/compute data statistics, such as our recent work on computing correlation values for entity or tag occurrences [5], up to "scientific queries" like interpolations that act as input to visualization, or data cleaning/predication on the moving data using methods such as Kalman filters; and any other queries that are traditionally processed over data streams such as running sums or quantiles. We do not explicitly rule out sliding windows to be used in LIGDB, but this, rather traditional and also orthogonal (to query-drive data gathering) concept, is not the focus in this proposal. Obviously, the selectivity of the query can impair the performance, and, in case of a *select **, turn LIGDB into a traditional store. It is the job of the below described query advisor to guide the user, considering the selectivity of the queries.

3.4 Query Processing

When considering query processing techniques, there are two important aspects of LIGDB that need to be considered. First, queries are not guaranteed to be executed within milliseconds as data is per se not given. That is, queries can run for seconds, which calls for grouping queries and sharing the load [21, 34, 14]. Second, the workload and the underlying database is very dynamic, the streaming and "historic" data as well as schemas are constantly changing. Thus, adaptive on-the-fly query optimization techniques need to be applied. Due to the dynamic nature of data streams, adaptive query processing has been addressed in [34, 8], but also for queries executed over raw data [4]. Adaptive indexing techniques [23, 26] should also be applied since, first of all, LIGDB has no fixed schemas, and second, the query workload is constantly changing—rendering fixed indexing schemes ineffective. Acosta et al. [2] discuss adaptive query processing with respect to SPARQL endpoints. Their main idea is adapting the query to the availability of the SPARQL endpoints, thus getting results even when some sources are not available, or parts of the query cannot be answered. In LIGDB, as there is no guarantee that the query can be answered as a whole, such adaptive techniques can be applied to gather answers for parts of the query.

3.5 Query Advisor

LIGDB aims at handling large amounts of heterogeneous content, not tailored to a specific, narrow application case with well designed and understood schemas. The trouble with this generic setup is that it is hard if not impossible to phrase meaningful queries without guidance. Thus, the query advisor is a key component of LIGDB. Statistics, constantly gathered from the incoming data stream (after the Enrich, Annotate, Fuse component), together with generic and domain specific ontologies, can serve as the base of the advisor. Blunschi et al. [10] employed ontologies to help keyword-based querying of complex-schema databases, for the case of business analysts in banking environments. They propose methods to find the most promising SQL-query candidates, based on input keywords. Clearly, in an arbitrary-data management system like LIGDB, this is of even higher importance.

Once a query base has been formed, and data is residing in the cache, collected data can trigger further query proposals. Sellam and Kersten propose a query advisor [39] that allows users to explore data and its statistics by posing queries that can be refined, gathering statistics and explanations of possible results. Another practical solution appears to be the application of the generic query-by-example paradigm [47] and reverse engineering of queries based on data samples [37, 46]; in addition to handing in object/entity centric (former) query results and waiting for LIGDB to refresh them. However, since all these existing techniques are designed over static data, and the nature of LIGDB implies that the data residing in the cache is dynamic, new challenges arise for adapting existing techniques to the constantly changing data.

3.6 Implementation

The input layer to LIGDB is formed by a cluster of machines that run systems such as Storm [40] or S4 [36] that aim at fault-tolerant realtime handling of big, high velocity streams of data; similar to what MapReduce is for batch processing. Such systems can scale to big loads of data by adding more machines for individual processing tasks running in parallel. Application developers have to *provide the implementation* of stream sources and operators, following the provided API. At that entry point, data is matched against registered queries and data not touched by at least a single query is immediately discarded. The Enrich, Annotate, Fuse and the Collect component is directly implemented in this streaming environment. Useful data and posted queries are indexed in an underlying key-value store by entity/object ids, including attributes and values. In the naive way, this would generate redundancy, there might be more advanced ways to store/index that information in an accessible way; such as by forming clusters or related information (e.g., in the sense of what is being done for RDF graphs). Ideally, though, a full-fledged database management system can be

employed to harness standard SQL and the research efforts of the past decades (query optimization, query advisors, indices, etc.).

4. CHALLENGES

Arguably the most characteristic facet of LIGDB is its forgetful or simply ignorant way to handle data. Only posed queries trigger the collection of query-relevant data and, hence, enables their processing. This is in strict contrast to traditional data management that collects large amounts of data, indexes it, and is able to process ad-hoc queries very efficiently; assuming data is not changing frequently. This characteristic of LIGDB is unique and poses several important research challenges that need to be addressed.

4.1 Incomplete and Time-Varying Results

Terry et al. [42] define monotone continuous queries as queries whose result is non-decreasing at any point of time. However, they consider queries over append-only databases, meaning current tuples do not get updated or deleted. In LIGDB this is not the case: The same query issued only seconds later may return completely different results due to the constantly changing flow of data. One way to solve this issue is to define *truth over time*, meaning that results are true only for a specific time point or time frame, depending on the query. This is different to queries over time windows, as queries do not get attached a time window, but rather results. It is not clear whether old, cached information (even though updated) and new streaming data should always be mixed together, or if it is better to start an entirely new query. Consider for instance the case of query results that are, despite being kept fresh, capturing a larger time span; it might not be semantically correct (or advised) to join/merge its results with a new query that has seen only very recent data.

In LIGDB, we are also facing the problem of (almost) never having the complete query answer. With exception of some type of queries, results in LIGDB are never complete, by design of the system. For instance, a query asking for cities with hotel prices below some value cannot be complete as new hotel offers may always arrive to the system. The question is when query results are complete enough to be returned to users, considering a benefit/cost tradeoff between runtime and completeness/quality. One approach could investigate the level of convergence of the query result to a specific value or size, or if the tuples of the result become "stable" enough.

4.2 Scale Independence

Queries in LIGDB trigger the data collection for their processing and the amount of data they "subscribe" to is, thus, performance critical. In the sense of the concept of scale independence [6, 18], ideally, queries in LIGDB can be answered with consumption of a bounded number of input data, produce only bounded intermediate results and have also size-bounded output. Without additional knowledge and constraints imposed by the application logic, as in PIQL [6], this is not possible, and it is unclear to what extent this can be implemented "in the wild" over schema-free, heterogeneous data. In addition, queries in LIGDB should behave well in the time dimension, that is, the time required for gathering data to allow a satisfactory query answering should be ideally bounded, too. Knowing the rough output cardinality of a query helps in so far, as the degree of completion or the full completion is known. Still, this does not immediately tell when exactly data arrives that would be required to finalize the results. In fact, it might happen that a query never gets answered within reasonable time. This needs to be determined as soon as possible to return to the query initiator. The system should also be interactive in the sense that it periodically returns incremental results or time-to-completion information.

4.3 Cold-Start Problem

In recommender systems, a cold-start problem (cf., [38]) occurs when a new item (user) is added to the system which cannot be recommended as no one has rated the item so far. In LIGDB there is a cold-start problem when a query is posted for which only few or no data at all is present in the system, as in the recent past no related query has been issued. With changing data and user interests this problem is expected to occur virtually at any time. In that case, all query-related data need to be first gathered from scratch and no previously cached query result is useful to answer the query at least partially. After this cold-start phase it is assumed that most queries can be answered at least partially by harnessing cached query results. Partially answerable means that previous queries allow finding a subset of the true query answers (objects) with potentially missing key-values pairs in the individual result entries. It is to be decided in this case if it is reasonable to postpone the result delivery to the query initiator in order to gather additional results and attributes, to already report the incomplete results, and/or to start data gathering to aim at more complete results. Another issue which rises with the cold-start problem is how and which queries to phrase: Phrasing queries when there is no schema available or some previous knowledge of the streaming data is not trivial. This could make the user resort to posing general data-gathering queries. Thus the LIGDB query advisor, ideally, by using the gathered statistics and/or ontologies, should be able to suggest queries in this case as well.

5. CONCLUSION

In this work, we sketched the core ideas and research challenges behind LIGDB, a data-management architecture that makes the case for storing only data for which a related query is posted to the system. This is in strong contrast to common data management systems, that follow an index-first-query-later paradigm or are running continuous queries (often of statistical nature) on data streams. LIGDB is designed to ideally act as a traditional data management system, serving ad-hoc queries in acceptable response times but can also harness long-running data stream analytics. The forgetful data handling and the pay-as-you-go building up of a data repository to be reused render LIGDB appealing for handling explorative and analytical queries over large input data. The price to pay for such a data processing principle is its susceptibility to data-to-query discrepancy and a not sufficiently large data input rate.

6. REFERENCES

[1] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible loading: access-driven data transfer from raw files into database systems. *EDBT*, 2013.

[2] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. Anapsid: An adaptive query processing engine for sparql endpoints. *ISWC*, 2011.

[3] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. I. Jordan, S. Madden, B. Mozafari, and I. Stoica.

Knowing when you're wrong: building fast and reliable approximate query processing systems. *SIGMOD*, 2014.

[4] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: efficient query execution on raw data files. *SIGMOD*, pages 241–252, 2012.

[5] F. Alvanaki and S. Michel. Tracking set correlations at large scale. In *SIGMOD*, 2014.

[6] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. Piql: Success-tolerant query processing in the cloud. *PVLDB*, 5(3), 2011.

[7] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. Dbpedia: A nucleus for a web of open data. *ISWC/ASWC*, 2007.

[8] S. Babu and J. Widom. Streamon: An adaptive engine for stream query processing. *SIGMOD*, 2004.

[9] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. *SIGMOD*, 1986.

[10] L. Blunschi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger. Soda: Generating sql for business users. *PVLDB*, 5(10), 2012.

[11] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. *MDM*, 2001.

[12] P. Bonnet and P. Seshadri. Device database systems. *ICDE*, 2000.

[13] S. Chandrasekaran and M. J. Franklin. Psoup: A system for streaming queries over streaming data. *The VLDB Journal*, 12(2), 2003.

[14] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. *SIGMOD*, 2000.

[15] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: being lazy is a virtue (when issuing database queries). *SIGMOD*, 2014.

[16] Cloudera Impala. `https://github.com/cloudera/impala`.

[17] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2), 2003.

[18] W. Fan, F. Geerts, and L. Libkin. On scale independence for querying big data. *PODS*, 2014.

[19] A. Feng, M. J. Franklin, D. Kossmann, T. Kraska, S. Madden, S. Ramesh, A. Wang, and R. Xin. Crowddb: Query processing with the vldb crowd. *PVLDB*, 4(12), 2011.

[20] C. Ge and L. Golab. Lazy data structure maintenance for main-memory analytics over sliding windows. *DOLAP*, 2013.

[21] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann. Shared workload optimization. *PVLDB*, 7(6), 2014.

[22] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2), 2003.

[23] G. Graefe and H. A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. *EDBT*, 2010.

[24] J. Hoffart, Y. Altun, and G. Weikum. Discovering emerging entities with ambiguous names. *WWW*, 2014.

[25] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my data files. here are my queries. where are my results? *CIDR*, 2011.

[26] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9), 2011.

[27] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[28] Y. Kargín, M. Ivanova, Y. Zhang, S. Manegold, and M. Kersten. Lazy etl in action: Etl technology dates scientific data. *PVLDB*, 6(12), 2013.

[29] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou. The researcher's guide to the data deluge: Querying a scientific database in just a few seconds. *PVLDB*, 4(12), 2011.

[30] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2), 2014.

[31] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1), 2010.

[32] W. Lang, R. V. Nehme, E. Robinson, and J. F. Naughton. Partial results in database systems. *SIGMOD*, 2014.

[33] G. Limaye, S. Sarawagi, and S. Chakrabarti. Annotating and searching web tables using entities, types and relationships. *PVLDB*, 3(1), 2010.

[34] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. *SIGMOD*, 2002.

[35] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54(6), 2011.

[36] S4: Distributed stream computing platform. `http://incubator.apache.org/s4/`.

[37] A. D. Sarma, A. G. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. *ICDT*, 2010.

[38] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock. Methods and metrics for cold-start recommendations. *SIGIR*, 2002.

[39] T. Sellam and M. L. Kersten. Meet charles, big data query advisor. *CIDR*, 2013.

[40] Storm: Distributed and fault-tolerant realtime computation. `http://storm-project.net/`.

[41] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. *WWW*, 2007.

[42] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. *SIGMOD*, 1992.

[43] L. Vargas, J. Bacon, and K. Moody. Integrating databases with publish/subscribe. *DEBS*, 2005.

[44] P. Venetis, A. Y. Halevy, J. Madhavan, M. Pasca, W. Shen, F. Wu, G. Miao, and C. Wu. Recovering semantics of tables on the web. *PVLDB*, 4(9), 2011.

[45] Yelp Academic Dataset. `https://www.yelp.com/academic_dataset`.

[46] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. *SIGMOD*, 2013.

[47] M. M. Zloof. Query-by-example: the invocation and definition of tables and forms. *VLDB*, 1975.