UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1343

Scalable Queries over Log Database Collections

MINPENG ZHU

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2016

ISSN 1651-6214
ISBN 978-91-554-9472-8
urn:nbn:se:uu:diva-275044

Dissertation presented at Uppsala University to be publicly examined in 2446, Department of Information Technology, Polacksbacken (Lägerhyddsvägen 2), Uppsala, Wednesday, 30 March 2016 at 13:00 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Emeritus Professor Peter Gray (The School of Natural and Computing Sciences, University of Aberdeen).

Abstract
Zhu, M. 2016. Scalable Queries over Log Database Collections. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1343. 51 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-554-9472-8.

In industrial settings, machines such as trucks, hydraulic pumps, etc. are widely distributed at different geographic locations where sensors on machines produce large volumes of data. The data produced is stored locally in autonomous databases called log databases. The collection of log databases is dynamically changing when new sites are dynamically added or removed from the federation.

In this application context, an efficient way to search and analyze passed behavior of products in use is desired. To enable scalable queries over collections of distributed and autonomous log databases we developed the FLOQ (Fused LOg database Query processor) system, which provides a global view of the working status of all machines on the sites through a meta-database integrating the dynamic log database collection. A particular challenge in this scenario is a scalable way to process numerical queries that identify anomalies by joining data from the meta-database with data selected from the collection of distributed and autonomous log databases. The Thesis describes the architecture of FLOQ. In particular different strategies to execute numerical queries over log database collections are investigated. FLOQ allows both the meta-database and the log databases to be stored in multiple formats using different kinds of data managers. FLOQ provides general and extensible mechanisms for efficient processing of queries over different kinds of distributed data sources.

Minpeng Zhu, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-554-9472-8
urn:nbn:se:uu:diva-275044 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-275044)

To my parents and great friends
好人好运

List of Papers

This Thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I Zhu, M., Risch, T. (2011) Querying Combined Cloud-Based and Relational Databases, The 2011 International Workshop on Data Cloud (D-CLOUD 2011), at 2011 International Conference on Cloud and Service Computing (CSC), Hong Kong, China, December 12-14, 2011, *In Proc. CSC 2011*, pp. 330-335.

II Zhu, M., Stefanova, S., Truong, T., Risch, T. (2014) Scalable Numerical SPARQL Queries over Relational Databases, 4th International workshop on linked web data management (LWDM 2014) in conjunction with the EDBT/ICDT 2014 Joint Conference, Athens, Greece, March 28, 2014, *In Proc. LWDM 2014*, pp. 257-262.

III Zhu, M., Mahmood, K., Risch, T. (2015) Scalable Queries Over Log Database Collections, 30th British International Conference on Databases (BICOD 2015), Edinburgh, UK, July 6-8, 2015, *In Proc. BICOD 2015*, pp. 173-185.

IV Mahmood, K., Risch, T, Zhu, M. (2015) Utilizing a NoSQL Data Store for Scalable Log Analysis, 19th International Database Engineering & Applications Symposium (IDEAS 2015), Yokohama, Japan, July 13-15, 2015, *In Proc. IDEAS 2015,* pp. 49-55.

Reprints were made with permission from the respective publishers. All papers are reformatted to the one-column format of this book.

Contents

Abbreviations

Amos	Active Mediator Object System
CDM	Common Data Model
DBMS	Database Management System
FLOQ	Fused LOg database Query processor (Paper III)
GAE	Google App Engine (Paper I)
JDBC	Java Database Connectivity (Paper III)
RDF	Resource Description Framework (Paper II)
RDFS	RDF Schema (Paper II)
SQL	Structured Query Language
SPARQL	Simple Protocol and RDF Query Language (Paper II)
URI	Uniform Resource Identifier (Paper II)
URL	Uniform Resource Locator (Paper I)
C_{Join}	Total cost of a join (Paper III)
B_i	Binding stream to site *i* (Paper III)
R_i	Result stream from site *i* (Paper III)
C_i	Total cost at site *i* (Paper III)
$C_{\sigma i}$	Cost of executing σ_i in RDB_i (Paper III)
$C_{\bowtie i}$	Cost of local join at site *i* (Paper III)
$C_{Bulkload_i}$	Cost of bulk loading in RDB_i (Paper III)
C_{σ_β}	Selection Cost for a single binding β (Paper III)
RDB_i	The relational log database at site *i* (Paper III)
C_{FLOQ}	Total execution cost in the FLOQ server (Paper III)
C_S	Cost of splitting the binding stream B in the FLOQ server (Paper III)
β	A single binding from the binding stream B_i (Paper III)
C_m	Cost of merging result streams R_i in the FLOQ server (Paper III)
C_{JDBC}	Cost of JDBC call for a single binding β
σ_i	The query to RDB_i (Paper III)
C_{B_i}	Cost of transferring binding stream B_i to site *i* (Paper III)
C_{R_i}	Cost of transferring result stream R_i from site *i* (Paper III)
C_{Net}	Network communication overhead cost for a single binding β (Paper III)

1 Introduction

Modern product development generates high volumes of data during its life cycle, from development and manufacturing, through use and maintenance, to reengineering and recycling. The ability to represent, search, and analyze many different kinds of data generated during a product's life cycle is critical for high quality and high availability. Within this context, an important issue is scalable approaches to collect, process, and analyze data produced in the manufacturing process.

Recently, there is a rise of manufacturing industry transformation. Research initiatives were developed and applied. The ideas such as Germany's Industry 4.0 [4] [20], China's Made in China 2025 [21], and US's Industrial Internet [19] are proposed. They share similar ideas of improving productivity, efficiency, and quality of products.

This Ph.D. work is from a real-world industrial scenario [30], where machines such as trucks, hydraulic pumps, cutting tools, etc. are widely distributed at different geographic locations and where sensors on machines produce large volumes of data. The data produced at each site describe time stamped sensor readings of machine components (e.g. oil temperature and pressure) and is stored locally in autonomous databases called *log databases.* The log databases are used to search and analyze abnormal behaviors of the monitored machines distributed over many sites. Furthermore, the collection of log databases is dynamically changing when new sites are added or removed from the federation.

In order to search and analyze data from the log databases there is need for a meta-database that describes properties of the monitored equipment and their log databases, e.g. the machine configurations at the sites, descriptions of the installed sensors, measurement tolerances, etc. The meta-database provides a global view of the working status of all machines on the sites. It represents meta-data integrating the log database collection. A particular challenge in this scenario is a scalable way to process queries that join data selected from the meta-database with data selected from the collection of log databases.

Abnormal machine behaviors can often be detected by identifying significant deviations between measured values stored in log databases and corresponding expected values stored as meta-data. Such deviations can be expressed as numerical expressions in query conditions identifying when measured values are outside the tolerances for a sensor model during a time

period. This requires a way to process such numerical query expressions over collections of log databases integrated through the meta-database.

The following research questions are investigated:

1. The overall research question is: how should the meta-database over the federation of log databases be represented and how should queries to the federation be expressed and processed?
2. How can different kinds of data managers be used for representing the log databases as well as the meta-database?
3. How can a cloud-based data repository be used for representing the meta-database?
4. How can queries that join the meta-database with data selected from collections of the distributed log databases be executed efficiently?
5. How can numerical queries to determine anomalies in measurements be executed efficiently over log databases?

To approach these challenges and answering research question one we developed a system, *Fused LOg database Query processor (FLOQ),* Figure 1. FLOQ integrates collections of dynamic, distributed, and autonomous log databases through a meta-database called the *FLOQ ontology*. The ontology is managed by the *FLOQ server*.

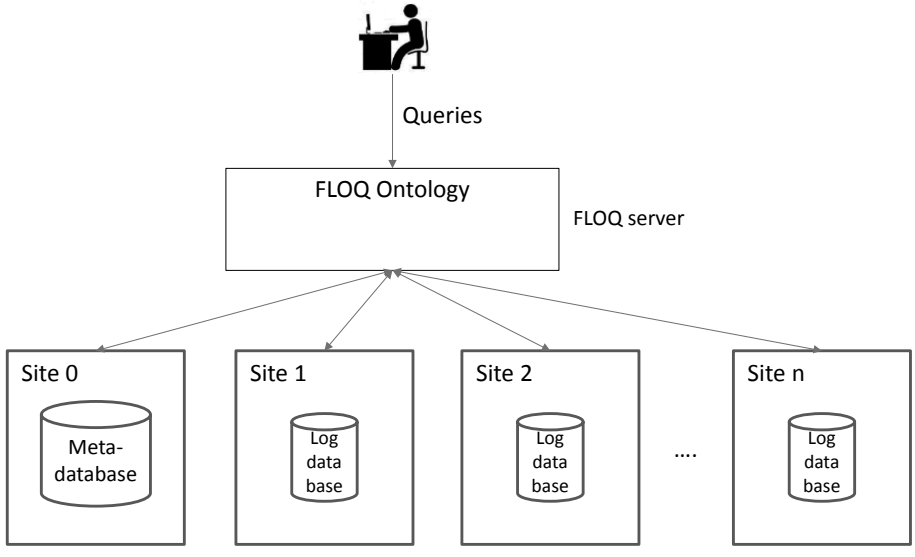


Figure 1. FLOQ Overview

The FLOQ ontology describes meta-data and physical properties about industrial equipment located at different sites. The FLOQ ontology is an application independent and can describe different kinds of industrial equipment. At each site enumerate 1,2,…,n there is equipment that produces sensor

measurement stored in local *log databases*. The log databases are locally maintained at each site and are described by the FLOQ ontology. Also parts of the ontology is defined as an external *meta-database* located at site 0. Independent FLOQ site servers encapsulate the back-end data managers to process queries at the sites. The user sends *queries* to the FLOQ server searching the log databases. The queries are expressed in terms of the ontology. Such a query might include searching and combining data from both the meta-database and the log databases. FLOQ uses the ontology to locate the log databases involved in answering a query. The queries often analyze data in log databases to find anomalies and other properties about the equipment. Such queries frequently involve numerical expressions, e.g., to detect deviations between measured and expected sensor readings.

The FLOQ ontology uses a domain-calculus based functional *common data model (CDM)* [28] to integrate all meta-data. FLOQ allows both the meta-database and the log databases to be stored in multiple formats using different kinds of data managers. The external data representations are mapped to the ontology represented by the CDM. Thus FLOQ provides general and extensible mechanisms for efficient processing of queries over different kinds of data sources, such as relational databases or MongoDB [22] [25]. This answers research question two.

In order to make meta-data widely accessible parts of the meta-database can also be stored in external data sources, such as Google's Bigtable [7]. In particular, in a world-wide organization the meta-database should be highly available and universally accessible from any location. Cloud-based data stores such as Bigtable provide high availability, universal access, and scalability. The FLOQ approach allows to map external meta-database representation to the FLOQ ontology [40]. For example, meta-data can be represented using either FLOQ's native CDM, as a relational database, or as cloud-based data manager such as Google Bigtable. The ability of FLOQ to represent meta-database using different formats answers research question three.

FLOQ provides a query processor for efficient, scalable, and distributed query execution. A general extensibility mechanism based on plug-ins allows the system to split a query into sub-queries accessing different kinds of data sources. Queries to the ontology are decomposed into sub-queries to different log databases. We propose two new join strategies, *parallel bind-join* (PBJ) and *parallel bulk-load join* (PBLJ) [42] for parallel execution of queries joining meta-data with data from autonomous log databases using standard DBMS APIs. This query processing over log database federations answers research question four.

For scalable execution of numerical queries over relational databases (RDBs), numerical operators should be pushed into SQL rather than executing the filters as post-processing outside the RDB; otherwise the query execution is slowed down, since a lot of data is transported from the RDB servers and furthermore indexes on the servers are not utilized. The *NUMTrans-*

lator algorithm [41] converts numerical expressions in numerical domain calculus queries into corresponding SQL expressions. We show that *NUMTranslator* improves substantially the scalability of numerical queries based on a benchmark that analyses numerical logs stored in an RDB. This answers research question five.

This Thesis overview is organized as follows. Chapter 2 gives an overview of technologies related to FLOQ. Chapter 3 describes the FLOQ ontology based on the application scenario motivating the FLOQ approach. Chapter 4 gives an overview of the architecture of FLOQ, including its ability to utilize different data managers through its extensibility mechanisms. Chapter 5 summarizes paper I, II, III, IV, describing technical contributions of each paper and my contributions to each paper. Chapter 6 describes conclusions and future work.

2 Background

This chapter describes the technical background related to this Thesis project. Relational database management systems in general are overviewed first. Federated and distributed database systems are then described by their architectures and query processing strategies. Furthermore, NoSQL databases, numerical databases, and temporal databases are described. Finally, the Amos II system is described, which this Thesis work extends substantially.

2.1 Database Management Systems

A database is a collection of data that can be stored [11]. A *database management system (DBMS)* is a software system that enables database creation, manipulation, and maintenance. For example, it enables the following:

- Creating a database with specified schema structure through a *Data Definition Language (DDL)*.
- Manipulating a database, for example, querying, inserting, deleting, and updating the data.
- Multi-user access to a database with different kinds of authentication control.
- Recovery control to restore the database back to a specific time point.

2.1.1 Data Models and Query Languages

A *data model* defines the structure and format of data used by a database management system. Different DBMSs have different data models. The most common data model is the *relational data model* where the database is represented as a set of tables.

The *database schema* describes the data stored in the database. In the relational data model it describes the names of tables and columns, and data types of row attributes (or column values) stored in the database. Each row in a table is identified with a *key*, which is one or several columns uniquely identifying a row. A *foreign key* is one or several attributes in a table referencing the attribute(s) in another table.

To be able to combine data from many log databases, FLOQ uses a functional and object-oriented data model [28] to represent meta-data about collections of log databases: the FLOQ ontology described in Chapter 3.

Each data model supports some query language. For example, the standard query language for the relational data model is SQL (Structured Query Language). In SQL, queries are issued over tables as predicates constraining table rows (tuples). SQL is a declarative language based on a predicate calculus called *tuple calculus* [8] [11] where variables are bound to tuples (rows) in tables and the user specifies how to match and constrain tuples. An alternative is *domain calculus* [8] [11], which is a predicate calculus where variables are bound to atomic values, rather than tuples as in tuple calculus. Since numerical expressions are easy to formulate using variables bound to numbers, queries over numerical expressions are simplified by using domain calculus, rather than the tuple calculus used by SQL where all variables are bound to tuples.

The functional data model used in FLOQ uses the domain calculus query language AmosQL [28].

A common domain calculus language used in the semantic web community is SPARQL [33]. SPARQL is a promising domain calculus based query language for scientific applications [1] [2]. FLOQ supports both SQL and SPARQL as alternative query languages for queries to the FLOQ ontology. Both SQL and SPARQL queries are translated by a parser into AmosQL queries for further processing.

Another common domain calculus language is Datalog [11]. Amos II (and FLOQ) uses a Datalog dialect, ObjectLog [28], as internal representation of queries. The query processor transforms ObjectLog expressions to improve performance.

2.1.2 Query Processing

For a given query, the DBMS takes care of how to efficiently retrieve the information from the database. The *query processor* of the DBMS transforms the query into an efficient program (execution plan) specifying how to retrieve data. Indexing is a commonly technique to improve the query execution performance over very large databases. For analytical tasks, [26] demonstrates that indexing improves query performance. Typical query processing steps in a DBMS are shown in Figure 2.

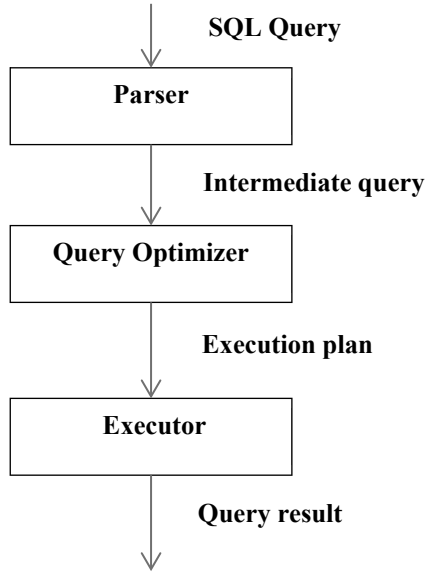


Figure 2. Query processing in DBMS

First, the parser checks the query's syntactic correctness and checks that tables and columns are correctly referenced. The result of parsing and validation is an *intermediate query*, usually a logical calculus expression to be executed. The query optimizer takes the intermediate query and produces an efficient program called an *execution plan* among the many feasible execution plans. Cost-based optimization [12] estimates the cost of executing a plan according to some cost model based on knowledge about database statistics, internal data representations, and search algorithms used in the plan. The query optimizer aims to generate an execution plan with minimal cost according to the cost model. To estimate the query plan cost, the optimizer needs, for example, the approximate number of disk block accesses, central processing unit (CPU) usage, etc. The number of disk block accesses is affected by the execution order inside the execution plan. Therefore, valid statistics of the database, e.g. number of rows in tables and number of different values in columns are very important for the query optimizer to estimate the query plan cost.

Finally, the executor interprets the execution plan and produces the query result.

A central part of FLOQ is novel specialized query processing mechanisms to process numerical queries over collections of distributed log databases described by a common meta-database. Domain calculus query transformations are utilized for transforming domain calculus queries into SQL tuple calculus queries. Query fragments are translated into sub-queries ac-

cessing the log databases and then joined by FLOQ. Special query transformations are used for processing numerical queries. In the Thesis various query processing strategies for this are evaluated.

2.2 Federated Databases

A *federated database (FDB)* [11] is a union of independent and autonomous databases. Each database in a federation has a local database schema, called *local conceptual schema*. A central federated database provides a *global conceptual schema* that integrates subsets of the local conceptual schemas to enable queries over the integrated database federation, i.e. the global conceptual schema implements an ontology represented by a common data model (CDM) that enables mapping participating databases representations to the CDM and integrates information from the participating databases. To integrate data from different databases, the global conceptual schema needs to solve *semantic data reconciliation* issues on how to combine similar or same information represented differently in the different participating databases. Since the participant database schemas are designed before the global schema, the global conceptual schema is designed in a bottom up fashion. The global conceptual schema enables query transparency to the user without showing the underlying conceptual schemas in the federation. However, it can be difficult to define such a global conceptual schema if the number of different participating databases is large. Finally, external schemas (views) can be defined by users on top of the global conceptual schema. Figure 3 shows how different schemas relate in a federated database.

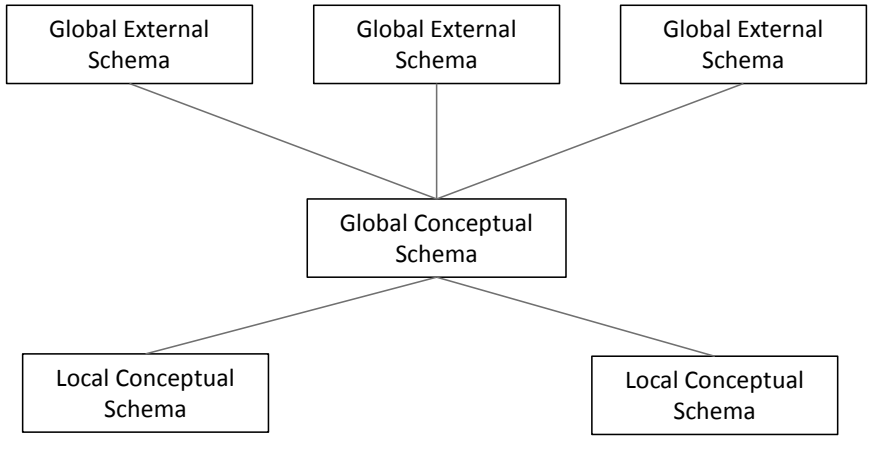


Figure 3. Federated Database Schemas

In FLOQ the log databases are independent and autonomous databases. The FLOQ ontology provides a global conceptual schema over the collection of

log databases in the federation. The global conceptual schema of FLOQ contains a uniform representation of all the databases storing the logged data. One particular problem addressed in FLOQ is that the collection of log databases is dynamic so that new log databases can be added and removed over time as new sites are included or removed. In particular, whereas federated databases traditionally have been used for integrating a fixed set of existing databases, FLOQ's log database collections are dynamic where log databases at different sites can join or leave the federation.

Federated DBMSs may include primitives to integrate databases implemented in different kinds of DBMSs, having different data models. For example, data from relational databases may need to be integrated with data from NoSQL databases [32] and text files. NoSQL data managers such as MongoDB [25] often use a data model where data is represented as JSON objects. In order to integrate MongoDB with, e.g. relational databases, the federated database system needs primitives to transform both the relational database model and the MongoDB data model into a CDM used by the federated DBMS.

In FLOQ's case the CDM is an extension of the functional and object-relational data model used in Amos II [28]. The FLOQ ontology is expressed in this CDM and defines meta-data including descriptions of log databases using different data models.

2.3 Distributed Database Systems

A distributed database (DDB) [11] [37] is a set of database processing nodes connected by a computer network. The database processing nodes are often geographically distant. The user sees the distributed database as a central database, while the database administrator (DBA) is responsible for deciding on which nodes different fragments of tables reside. Each table in a distributed database is partitioned logically, for example, a table can be fragmented by rows or columns where each fragment is stored in different distributes database nodes, which is called horizontal and vertical fragmentation, respectively. For example, a company-wide table may have different row fragments in different sites defined by the DBA. For any fragmentation method, a query to a fragmented table must return the same result as if the table is stored in a non-distributed database, i.e. the fragmentation scheme needs to be transparent in queries.

To increase query availability tables may be replicated at different sites. Data replication is also useful in speeding up query answering by accessing data available close to or at the site where queries are issued. Replication may however significantly reduce update speed, since distributed transactions might need to be propagated to several of the replicas to enable consistency.

Since the data is physically distant in distributed databases, special distributed query processing strategies are required. Often several distributed database processing nodes are involved to answer a query. For example, distributed joins ship data from one site to another and to perform the join there. Join approaches such as semijoin [11] uses a strategy to first reduce the data shipped from a site by projecting the join column and removing duplicate values. Then a semijoin between the local data and the shipped data is performed. Finally, the semijoin result is sent to the other site for performing the final join there.

FLOQ uses two distributed query processing approaches namely *parallel bind-join* (PBJ) and *parallel bulk-load join* (PBLJ) to perform queries that join meta-data from a common meta-database with the data selected from the collections of distributed autonomous log databases. This generalizes the central bind-join [14], where data is joined by binding values when accessing external data sources.

2.4 NoSQL Databases

Not only SQL (NoSQL) databases [32] propose non-relational data models to provide availability and scalability of distributed databases. NoSQL databases such as MongoDB are designed to perform simple tasks with high scalability [6]. For providing high performance updates, NoSQL databases generally sacrifice strong consistency by providing so called eventual consistency compared with the ACID transactions of regular DBMSs.

NoSQL databases often have limited schemas where attributes in collections are dynamic, compared to relational databases where tables must have all columns specified in the schema before populating the database. For example, MongoDB [25] provides dynamic schemas which allows new attributes for data to be dynamically added to existing databases. It means records can be in different schema even in the same collection. This feature enables flexible insertions of the data into the database. NoSQL databases cover different kind of data store families, such as document stores, graph databases, cloud-based data stores.

[6] gives an overview of list features on the state-of-the-art NoSQL databases such as MongoDB [25], Cassandra [5], Redis [27], HBase [16], Memcached [24], and CouchDB [9]. However, Cassandra [5], Redis [27], HBase [16], Memcached [24], and CouchDB [9] do not provide full secondary indexing, which is essential for scalable performance of numerical queries. MongoDB [25] provides both a query language along with primary and secondary indexing. This is well suited for analyzing persisted logs.

In this Thesis, FLOQ investigates the approach to store parts of the common meta-data using Google's Bigtable cloud-based NoSQL database. Furthermore, FLOQ allows the log databases to be stored in different forms,

such as relational databases and MongoDB. A NoSQL database such as MongoDB may be useful for typical historical analysis of log data or numerical log analytics where transactional consistency conforming ACID compliance is not required.

2.5 Numerical and Temporal Databases

The content of a relation in a relational database can be changed over the time by inserting new tuples, deleting existing tuples, or updating existing tuples. A regular database maintains a snapshot of the current data. Temporal databases [31] are databases that maintain histories of data values over time for each table. There are different approaches to store time attributes, such as a time instant is stored together with each tuple in a table, or the *valid time* for a tuple is defined by an associated starting time and ending time instant. In FLOQs log databases valid time is important for defining during what time period a measured value is valid. The FLOQ ontology thus associates a valid time interval with each row in a log database table.

Numerical and scientific databases [29] typically are used to store complex objects such as array data representations. Queries over numerical databases can contain query conditions such as matrix operations, numerical computations, etc. Such query conditions can be, for example, a linear equation or a numerical computation involving numerical operators comparing values. Query optimization techniques such as compile time evaluation and query rewrites can be applied to improve the scalability of query execution.

In our applications sensors in equipment located at the sites produce measurement values of industrial equipment. The measured values are stored in the local log databases for analyzing the abnormal equipment behavior in the past. Each measurement has an associated valid time interval. To observe equipment abnormalities, queries to FLOQ often involve numerical query conditions. To improve query scalability, numerical query conditions are pushed down to the log databases. In FLOQ the *NUMTranslator* algorithm [41] utilizes a table driven approach to extract and translate numerical domain calculus operators into numerical tuple calculus operators, which are translated to SQL expressions executed by a relational DBMS.

2.6 Overview of Amos II

Amos II [28] is an extensible main-memory database system which in used in our prototype implementation of FLOQ. Amos II provides a functional and object-oriented data model where *objects, types,* and *functions* are the essential concepts. Types classify different kinds of objects stored in the database and functions define properties and computations over the objects.

FLOQ utilizes the Amos II data model as a common data model (CDM) for storing the FLOQ ontology to integrate collections of log databases.

Each object is an instance of some types and all object instances of a type represent the *extent* of the type. Functions model object properties, relationships between objects, and computation over objects. A function is defined by a *signature* and an *implementation*. The signature defines the input and result parameter types and names. The implementation defines rules how to relate inputs and outputs. For example, *stored* functions are used to represent object attributes stored in an Amos II database as a table. *Derived* functions are side-effect free, precompiled, and optimized queries in terms of other Amos II functions. *Foreign* functions enable low-level interfaces for accessing external data sources through data manager.

AmosQL [28] is the domain calculus based query language in Amos II, where queries are expressed in terms of functions over variables bound to typed objects. The query processor internally represents queries as domain calculus ObjectLog [28] expressions, which extends Datalog [11] with types, objects, external predicates, and disjunctions. Since Amos II has an extensible engine (both data manager and query processor), new data types and operators defined by some external programming languages (C, Java, or Lisp) in new applications can be added to AmosQL. This extensibility allows wrapping different data representations of different kind of data sources.

FLOQ uses the data model of Amos II to represent the FLOQ ontology. The extensibility of Amos II is used for accessing different kinds of external data sources. In particular relational log databases are accessed through a relational *data manager interface (DMI)* [15], while Google Bigtable data stores can be access through another DMI [40]. The latter enables FLOQ to store parts of the FLOQ ontology as a cloud database. The MongoDB DMI [22] provides query processor and interfaces to MongoDB databases. It enables log databases managed by MongoDB to be queried through FLOQ.

To enable distributed query processing, many Amos II instances can be set up and communicated using TCP/IP in a federation. In FLOQ, the Amos II instances in the federation are called *FLOQ site servers* and the *FLOQ server* represents the FLOQ ontology and integrates data from a collection of FLOQ site servers.

The query processing of Amos II is illustrated in Figure 4.

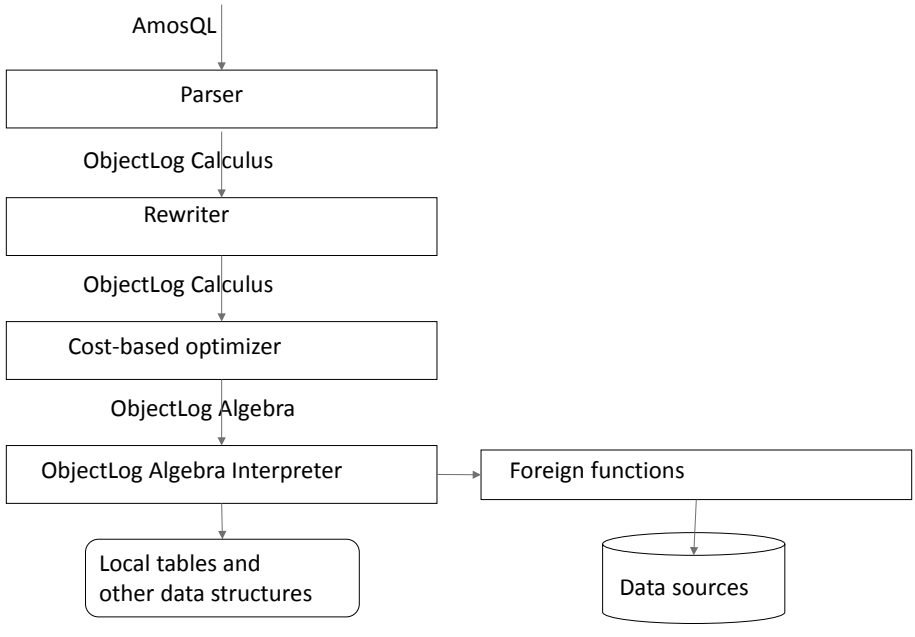


Figure 4. Amos II query processing

The system first parses the query into an *ObjectLog Calculus* expression. The *rewriter* applies logical query transformations on ObjectLog expressions. For example, it expands views to expose indexes, eliminates common subexpressions, and evaluates expressions at compile time. The *cost-based optimizer* applies optimization algorithms on the transformed ObjectLog expression to generate an optimized execution plan in terms of an *ObjectLog algebra*. The *ObjectLog algebra interpreter* runs the execution plan where foreign functions provide user-defined interpretations and access to foreign data sources. Native Amos II databases are accessed or queried through *local main-memory tables* and data structures supported by the system, such as vectors and dictionaries.

In FLOQ the Amos II query processor is modified with new rewrite mechanisms [40] to automatically split ObjectLog calculus expressions into query fragments accessing different kinds of data sources. The mechanism rewrites calculus expressions into equivalent ObjectLog expressions to generate a query execution plan accessing different data sources. Thus queries to the FLOQ ontology are decomposed into ObjectLog sub-queries to different data sources.

In our application, queries discovering abnormal machine behaviors often involve query include numerical expressions, inequalities, comparisons, etc. in query filter. For scalable execution of numerical queries to log databases,

the numerical expression should be extracted and translated to push it down to the log database. The *NUMTranslator algorithm* [41] extends the rewrite strategy [40] to translate numerical operators in domain calculus queries into numerical operators into SQL tuple calculus expressions through system tables of FLOQ.

FLOQ provides a rewrite strategy implementing special query optimization strategies [42] to support scalable queries that join meta-data from a common meta-database with data selected from a collection of distributed autonomous log databases. Two new join strategies for parallel execution of queries joining meta-data with data from autonomous log databases using standard DBMS APIs are proposed and implemented: parallel bind-join (PBJ) and parallel bulk-load join (PBLJ).

3 The FLOQ Ontology

Recently, there is a rise of manufacturing industry transformation. Research initiatives were developed and applied. The ideas such as Germany's Industry 4.0 [4] [20], China's Made in China 2025 [21], and US's Industrial Internet [19] are proposed. They share similar ideas of improving productivity, efficiency, and quality of products.

Within this context, an important issue is scalable approaches to collect, process, and analyze data produced in the manufacturing process. For example, industrial equipment (machines) with sensors installed on its components report the working status by delivering its measured data as data streams to some monitoring center where it is analyzed. Such data describe time stamped sensor readings of the monitored machines' components, such as the pressure of hydraulic motor pumps, the oil pressure in oil tankers, and the temperature of the engines in wheel loaders. These measurements reflect how the components function. The data is especially useful for analyzing the working status of monitored machine components. Data values outside certain machine-dependent constraints represent abnormal behavior and are therefore identified and used for further analyses. Thus, one way to analyze the measured data is to collect and store them for historical analyses. The stored measured data is called *log data* and a database storing such log data is called a *log database*. The data in log databases will be interpreted and analyzed to improve the monitored equipment's reliability. Since the sensors deliver the measured data as data streams, a *Data Stream Management System* can be used for real-time analyses of the data streams [38]. The scope of this Thesis work is focusing on analyzing stored log data rather than real-time stream data processing.

In Section 3.1 a common scenario from an industrial setting is presented to show the need to analyze historical log data in order to find abnormal machine behavior. Log data from embedded sensors is stored in a local log database at each site. Based on this scenario, in Section 3.2 the general conceptual schema description of monitored equipment, called the *FLOQ ontology*, is defined. Section 3.3 describes an application scenario log data set.

3.1 Industrial Application Scenario

To improve productivity, efficiency, and quality in manufacturing industry data is collected from sensors installed in industrial equipment, which enable monitoring and predicting their behavior to support, e.g., preventive maintenance [30]. The computation process provides monitoring, analysis, and control with feedback to the manufacturing process. Within this context, an important issue is a scalable approach to collect, process, and analyze the data produced in the manufacturing process.

As an example [30], Bosch Rexroth Mellansel AB (Hägglunds) [17] produce hydraulic drive systems, which are used in different areas of heavy industry, such as recycling, material handling, mining, etc. Figure 5 shows a wood waste shredder that is smashing wood waste to produce shredded wood that can be used, e.g., for animal bedding or as top of soil to improve fertility and preserve moisture. It is driven by a hydraulic motor which is connected and power supplied with a *hydraulic drive unit* containing a hydraulic pump driven by an electric motor. The hydraulic drive unit includes a *Spider control system* [18], which controls and monitors the hydraulic drive unit. The spider control system is a modularized control system that allows control of different kind of hydraulic drive systems, such as the one powering the shredder in Figure 5.

Figure 5. A wood waste shredder at Mellansel plant

The Spider control system contains a unit called, Spider Link, which collects the measured data and provides a serial log channels interface for data download. The collected data are in CSV format and can be transferred via USB-memory or GPRS-link. Figure 6 shows a scenario of data collected from a

hydraulic drive unit via a spider link and transferred by a GPRS modem to be stored in a log database at the site.

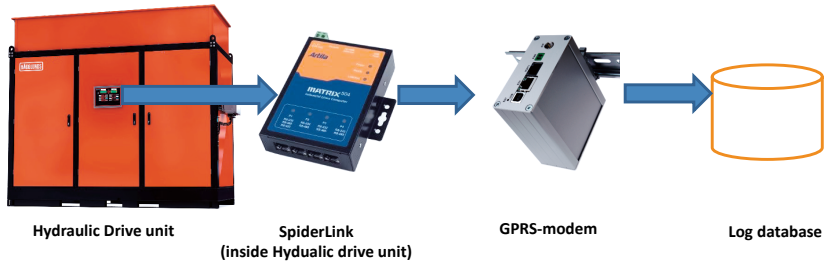


Figure 6. An example of data collection and transportation

Figure 7 shows parts of the inside of a hydraulic drive unit with an electric motor, a hydraulic pump, a heater, a set of filters, etc. To monitor the working status of the hydraulic drive unit, sensors are installed that continuously deliver measured values for the hydraulic drive unit components. While the wood waste shredder is working, sensors are continuously generating data monitoring the equipment. If measured data is not in the desired range of each component it reflects abnormal machine behavior. For example, the hydraulic drive unit may stop working because of too high pressure in its pump. Then experts need to be brought in to analyze the problem and take decisions to avoid an abrupt breakdown.

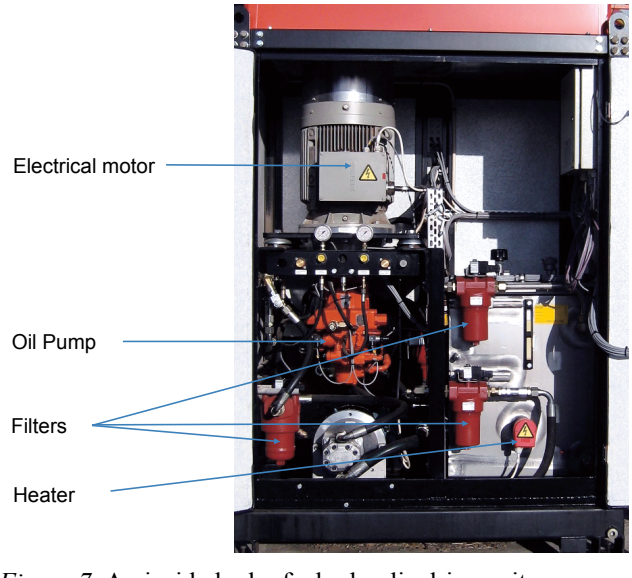


Figure 7. An inside look of a hydraulic drive unit

One particular goal of FLOQ is to improve the efficiency of the industrial process, for example by reducing the equipment stopping time and managing the maintenance of the machines. This requires analyzing logged sensor readings produced by the equipment to check if it is working normally. The FLOQ ontology provides a universal view of logged sensor data. This enables the test engineer to, e.g., analyze collected historical sensor data to see how long time the components was previously working under certain conditions. Based on these analyses a prediction regarding a component's failure can be estimated and prepared for solutions for equipment outages to make the equipment up and running again. To determine how the equipment has previously behaved in a given situation, the measured values of certain components need to be compared with its expected values. If the result is not within a certain tolerance threshold, it is probably an indication that the component did not function as normal.

In our application scenario, hydraulic drive units are widely distributed and used to supply power to hydraulic motors at each site. To analyze the collected measured data from a hydraulic drive unit, each site maintains its own collected measured data stored in an autonomous log database. Figure 8 shows such a scenario where spider control systems in the hydraulic drive units send measured data through GPRS to a local log database at each site. It allows the head office to analyze the working status of the equipment through the collection of log databases.

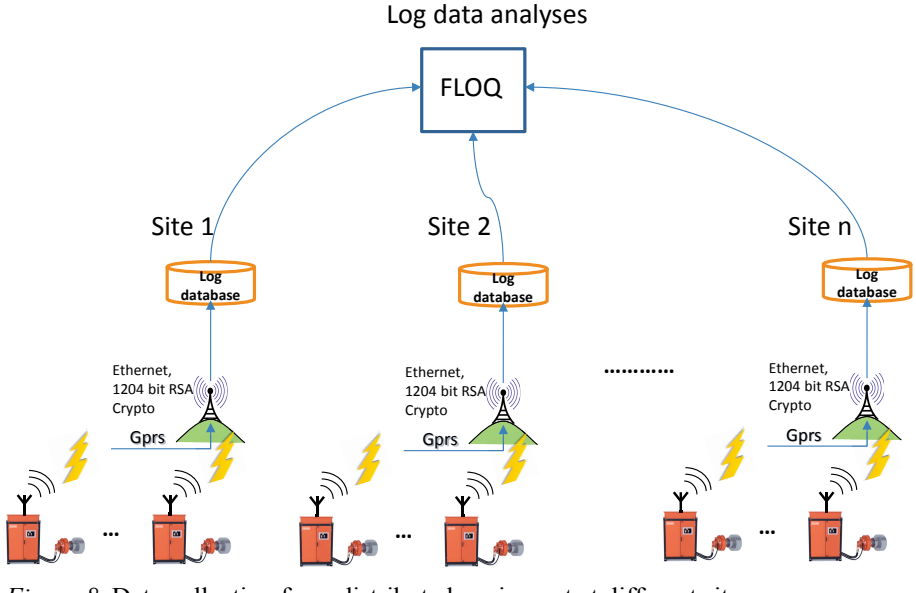


Figure 8. Data collection from distributed equipment at different sites

To ensure that collected data is useful and reveal abnormalities, the measured data needs to be collected at adequate frequencies. The fault detection and diagnosis on the collected measured data are also based on other factors, such as how the machine is used including operating cycle time, average velocity, etc.

Based on this and similar industrial scenarios [30] the FLOQ ontology was defined to represent meta-data about sensor readings from collections of log databases.

3.2 The FLOQ Ontology Definition

Figure 9 shows basic types and functions representing the FLOQ ontology schema for our industrial application scenario. More properties can be added to customize the ontology.

The type *MachineModel* represents different kinds of machines. It has four properties represented as stored functions: a unique machine model identifier *mm(MachineModel)->Number*, a name *name(MachineModel)->String*, a model description *descr(MachineModel)->String*, and its manufacturer *manuf(MachineModel)->String*.

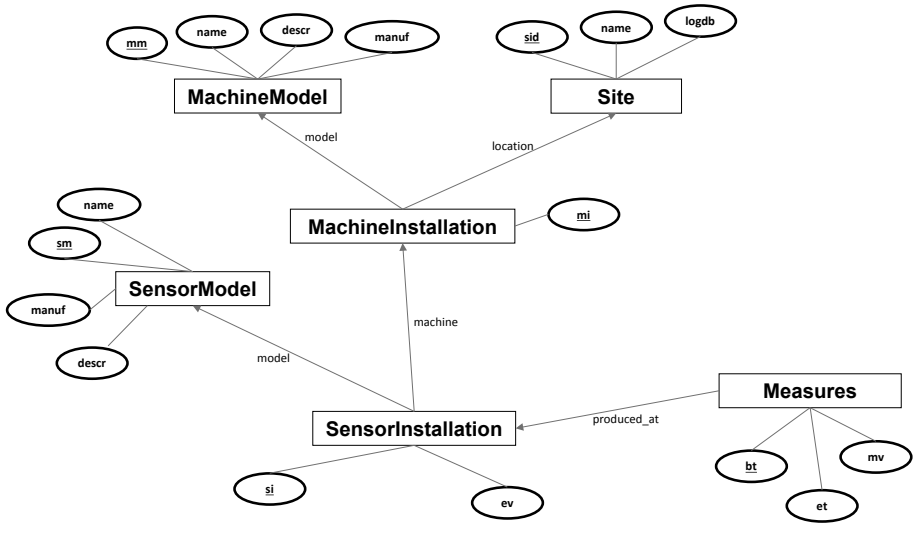


Figure 9. FLOQ ontology schema

The type *MachineInstallation* represents machine configurations at different sites. It has a unique machine installation identifier *mi(MachineInstallation)->Number*. The function *model(MachineInstallation)->MachineModel* identifies the machine model used in an installation and *location(MachineInstallation)->Site* identifies its site.

The *Site* type represents the locations of machines and log databases. It has a unique identifier *sid(Site)->Number*, a name *name(Site)->String*, and an identifier of its log database *logdb(Site)->Number*.

The type *SensorModel* represents sensor models. Sensor models have the properties: a unique identifier *sm(SensorModel)->Number*, a name *name(SensorModel)->String*, a description *descr(SensorModel)->String*, and a manufacturer *manuf(SensorModel)->String*.

The type *SensorInstallation* represents different installations of sensor models. It has a unique identifier *si(SensorInstallation)->Number* and an expected measured value *ev(SensorInstallation)->Number*. The sensor model of a sensor installation is identified by the function *model(SensorInstallation)->SensorModel,* while the machine on which a sensor is installed is identified by the function *machine(SensorInstallation)->MachineInstallation*.

The *Measures* type represents measurements from sensors installed on different machines valid in the time interval *[bt,et)*. Its attributes include the begin time *bt(Measures)->Time,* the end time *et(Measures)->Time*, and the measured value *mv(Measures)->Number*. The sensor installation where a value was measured is identified by the function *produced_at(Measures)->SensorInstallation*.

Figure 10 shows how the FLOQ ontology is represented as an external relational database schema mapped to the FLOQ ontology.

MachineModel(m, mmn, descr, mmanuf)
MachineInstallation(mi, m, sid)
SensorModel(sm, sname, descr, smanuf)
SensorInstallation(si, mi, sm, ev)
Site(sid, name, logdb)

Figure 10. Meta-database schema

Each site has its own autonomous log database table *Measures(mi, si, bt, et, mv)* (Figure 11) storing measurements from the sensors installed on the machines located at the site.

Measures(mi,si,bt,et,mv)

Figure 11. Log table at each site

The FLOQ view *VMeasures* (Figure 12) integrates the collection of log databases. It is logically a union-all of all log tables (*Measures*) on the different sites. In the view the attribute *logdb* identifies the origin of a tuple in a log database. Through the meta-database users can make queries over all log tables by joining the meta-data with the view *VMeasures*. Since the set of

log databases is dynamic and accesses many databases it is not feasible to define *VMeasures* as a static view; instead FLOQ processes queries to *VMeasures* by dynamically submitting queries to the log databases and collecting the results.

```
VMeasures(logdb,mi,si,bt,et,mv)
```

Figure 12. Integrated view in FLOQ server of all log tables

3.3 Application Scenario Log Data Sets

As test data in this PhD project, in the experiments we populate the meta-database and the log databases with meta-data and log data from Hägglunds. Sensor data was collected from the pump in Figure 7. It contains both normal and abnormal data. The data was delivered as a set of CSV files where each file includes meta-data about the logged data such as unit names, file sequence numbers, measured parameter names, sampling rates, etc., as well as logged sensor readings having a time stamp *ts* of each measurement. When loading the logged values into a log database, the time stamps are transformed into valid time intervals [*bt, et*) (Figure 11). The data was used in papers II, III, and IV. Figure 13 shows a small sample of log data from the B-side pump pressure sensor in a time interval.

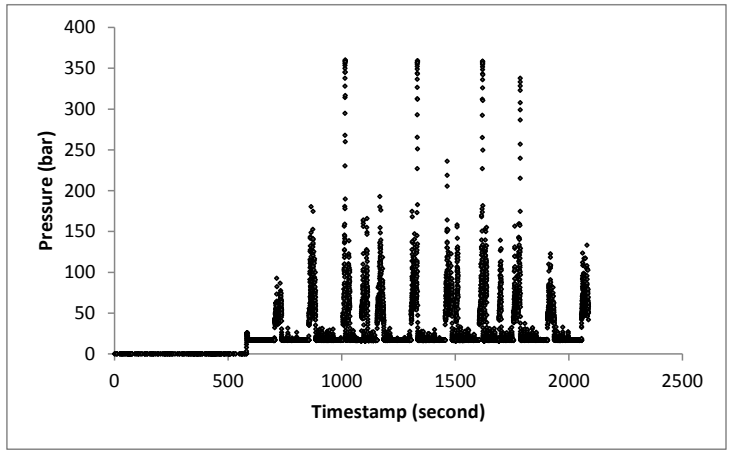


Figure 13. Sampled measured B-side pump pressures

There is an initial warm-up time in the figure of 581.1 seconds. After that, the pressure value starts to climb up and down. Abnormal situations are detected when the measured value *mv* is larger than the configured expected

value *ev*. For example, when *ev=359.44* it means the tolerance is wide and in this case all points are normal. If *ev=0* all points become abnormal.

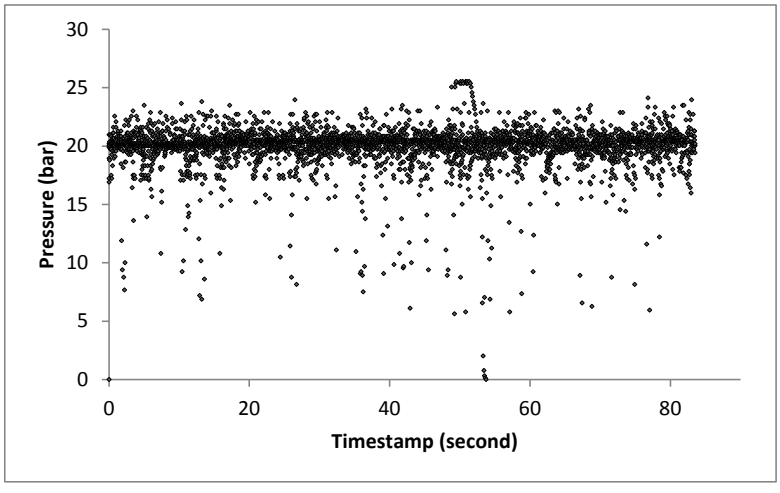


Figure 14. An example of sampled measured data of pressure charge pump

Figure 14 illustrates an example of a scatter plot of the measured values for pressure charge (B-side) of the pump in Figure 7. In this case, the expected value is set to 20.0. Here the threshold value is used to indicate absolute or relative deviation from the expected value. For example, all points in Figure 14 become abnormal when *ev=0*, while all points are normal when *ev=20*.

4 The FLOQ System

4.1 Architecture

Figure 15 illustrates the FLOQ architecture. It provides a uniform view of measurement data from a collection of log databases located at different *sites*. The user specifies *queries* to the federation in terms of the *FLOQ ontology*, which is managed by the *FLOQ server*.

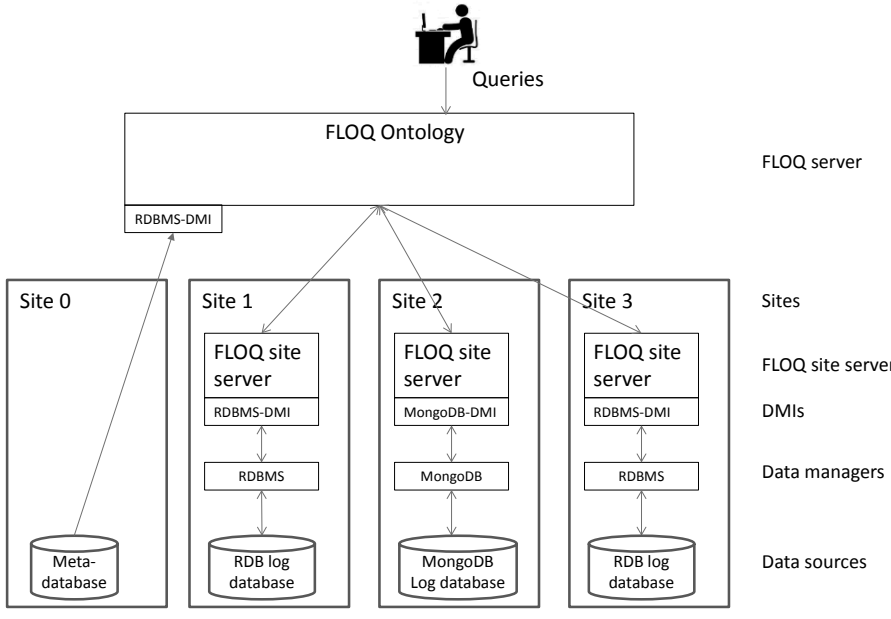


Figure 15. FLOQ system architecture

To enable processing queries over different kinds of *data sources* managed by different kinds of *data managers*, FLOQ supports plug-ins of data manager interfaces, *DMIs*, for each kind of data manager. For example, in Figure 15 the log databases at site one and three are managed by relational data managers, *RDBMSs*, interfaced using the *RDBMS-DMI,* while the log database at site two is managed by a *MongoDB* data manager [25] interfaced through the *MongoDB-DMI.* Furthermore, parts of the FLOQ ontology itself

is stored in a relational database at site zero. FLOQ uses the RDBMS-DMI to map the meta-data in site zero to the FLOQ ontology.

To process local queries each site has a *FLOQ site server,* which is a FLOQ system that contains schema mappings between the log database at the site and the FLOQ ontology view. There is no site server for site zero since the meta-data is mapped directly to the FLOQ ontology and then queried directly from the FLOQ server.

Processing queries joining meta-data in the FLOQ ontology with data from different log databases requires sending sub-queries from the FLOQ server to the corresponding FLOQ site servers. FLOQ processes such joins by accessing the FLOQ ontology to find the identifiers of the log databases that need to be accessed to answer the query. Then sub-queries are generated for each data source and sent to the FLOQ site servers encapsulating them. The query processor in a FLOQ site server translates a received sub-query into a local execution plan that contains calls to its log database through the DMI. It sends back to the FLOQ server the result of executing the query as a stream of tuples. Parallel processing is provided since the FLOQ site servers work independently of each other. The results from many FLOQ site servers are asynchronously merged by FLOQ server while emitting the result to the user.

Figure 16 illustrates meta-data about DMIs stored in the FLOQ ontology. This *DMI meta-data* enables FLOQ to process queries over different kinds of data managers.

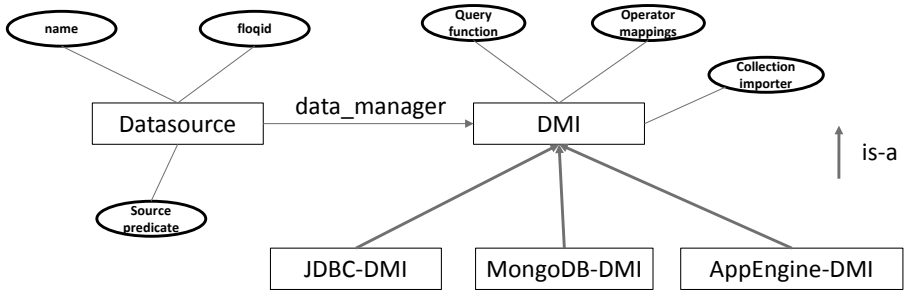


Figure 16. DMI meta-data

Data sources registered with FLOQ are represented by instances of type *Datasource*. Each data source has a unique *name* assigned by the user, and a *floqid* number assigned by FLOQ. The *DMI* of the system managing a data source is obtained by the function *data_manager()*. Each data manager provides an interface to execute queries by calling a *query function* implemented as a foreign FLOQ function. Depending on the capabilities of a data manager its query function can process queries having different query operators mapped to corresponding FLOQ functions through the *operator mappings*.

When a new collection in a data source is made accessible to FLOQ the user calls a *collection importer* to import meta-data of a data source using its data manager. The collection importer generates a *source predicate* for each accessed data source collection. The source predicate is a derived function that retrieves the tuples in the collection. For example, if the data source represents a log database the tuples represent the rows of the *Measures* table in Figure 11, while source predicates representing FLOQ meta-data will return the corresponding tuples of the meta-database tables in Figure 10. For a given query to the FLOQ ontology the extensible query processor generates query fragments to different sources by rewriting the source predicates.

The FLOQ ontology stores general meta-information about the locations and names of all FLOQ site servers in the federation, while each FLOQ site server has its own local schema describing its local data source. Distributed FLOQ site servers can be set up communicating using TCP/IP. A FLOQ site server joins the federation by registering itself to the FLOQ server by remotely calling the collection importer in the FLOQ server for the DMI of the site. The collection importer creates a new instance of type *Datasource* representing the site server along with the source predicate representing the *Measures* table of the new site. After the registration the FLOQ ontology has all required meta-data about the new FLOQ site server needed to process queries accessing the site server. The FLOQ site servers have full query processors, which enables processing sub-queries submitted from the FLOQ server.

4.2 FLOQ Query Processor

The query processor of FLOQ extends the query processor of Amos II in the following ways:

- FLOQ provides novel specialized query processing mechanisms for different kinds of DMIs. The mechanism is based on plug-ins called *extractors* (named 'absorbers' in Paper I, [40]) and *finalizers*.
- By developing a DMI for Google App Engine it is possible to store parts of the FLOQ ontology in an external cloud-based datastore. (Paper I, [40])
- A streamed interface to Google App Engine provides queries to Bigtable data repositories returning large data volumes. (Paper I, [40])
- A table driven approach used by the extractors and finalizers provides the *NUMTranslator* mechanism to translate numerical domain calculus operators into numerical SQL tuple calculus operators. (Paper II, [41])
- The parallel bind-join (PBJ) provides streamed parallel joins between meta-data in the FLOQ ontology and the dynamic set of autonomous, distributed log databases utilizing standard DBMS APIs. (Paper III, [42])

- The parallel bulk-load join (PBLJ) utilized the bulk load capabilities of a site data manager to provide scalable joins between the meta-database and the log databases. (Paper III, [42])

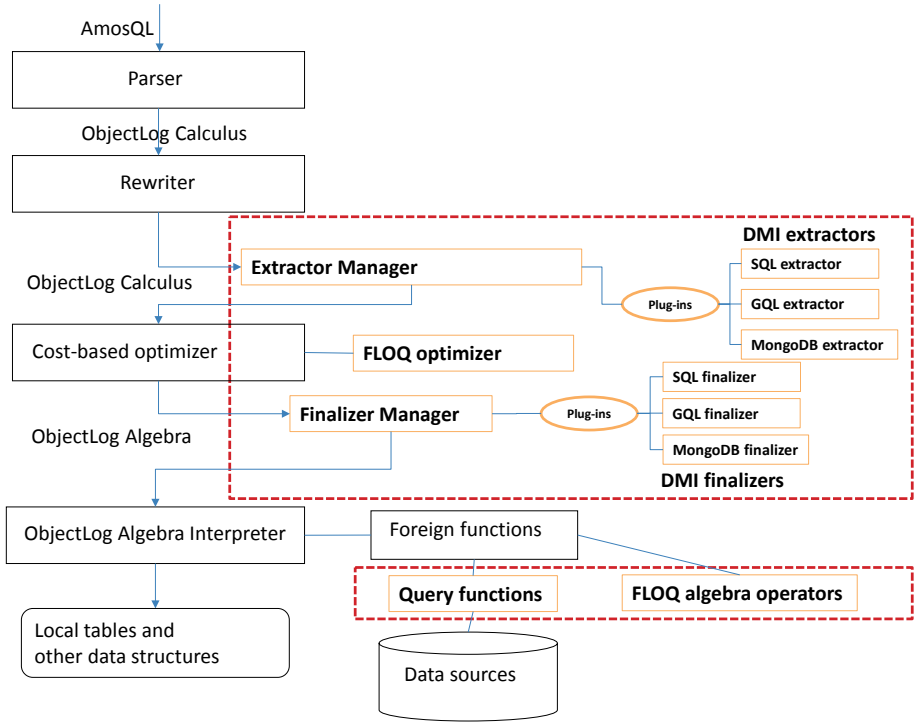


Figure 17. FLOQ query processor extensions of the Amos II query processor

Figure 17 illustrates how FLOQ extends the query processing steps of Amos II. The extensions are highlighted by the dot-lined rectangles in the figure. The following modules are added:

1. The ObjectLog *rewriter* of Amos II is extended with an *extractor manager* that automatically transforms ObjectLog calculus fragments into sub-queries accessing the different kinds of data sources. For each DMI there is a specialized *DMI extractor* plugged-into the extractor manager. The extractor manager takes an ObjectLog query and, for each source predicate referenced in the query, calls the corresponding DMI extractor to collect from the query the predicates that can be executed by the site server, based on the capabilities of the DMI (paper I, [40]).
2. After the cost-based optimization the execution plan is passed to the *finalizer manager*. It traverses the optimized ObjectLog algebra expression to translate algebra fragments into calls to the query function of the DMIs used to access queried data sources. As for the extractors, each DMI has a specialized *DMI finalizer* plugged-into the finalizer manager. A DMI finalizer transforms into query function calls the fragments of an

algebra expression that can be translated based on the query capabilities of the DMI's data manager. For example, for translating numerical ObjectLog algebra expressions to SQL, the *NUMTranslator algorithm* transforms numerical ObjectLog algebra operators into SQL tuple calculus expressions through the operator mappings table of FLOQ in Figure 16. The fragments of an ObjectLog algebra expression that cannot be translated into query function calls remain in the FLOQ algebra expression. Thus the rewritten query plan will be an ObjectLog algebra expression with calls to query functions (paper II, [41]).

3. To speed up queries combining meta-data with distributed logged sensor readings, sub-queries to the log databases should be run in parallel. Two join strategies PBJ and PBLJ for parallel execution of queries joining meta-data with data from autonomous log databases using standard DBMS APIs are proposed and their performance evaluation are analyzed (Paper III, [42]). In general, PBLJ performs better than PBJ, details in [42]. To implement PBJ and PBLJ, the *FLOQ optimizer* (Figure 17) was developed. It rewrites queries to the *VMeasures* view (Figure 12) to generate calls to *FLOQ algebra operators,* using PBJ or PBLJ to enable parallel sub-queries to the log databases. The extractor and finalizer plug-ins of a RDBMS or MongoDB (Figure 17) are utilized by the FLOQ optimizer to generate corresponding sub-queries to RDBs or MongoDB log databases, respectively.
4. It was investigated of how the state-of-the-art NoSQL DBMS MongoDB was suitable as a scalable log database manager. The performance of using MongoDB to store the *Measures* tables at a site (Figure 11) was compared with using two different relational DBMSs with various configurations. MongoDB was shown to have similar performance as a state-of-art RDBMS (Paper IV, [23]). A MongoDB query function interface was developed [22] along with a MongoDB extractor and finalizer plug-ins (Figure 17).

5 Technical Contributions

The technical contributions of this Thesis are summarized below, together with summaries of the published papers to guide the reader on how the included papers relate to the research questions.

5.1 Paper I

Zhu, M., Risch, T. (2011) Querying Combined Cloud-Based and Relational Databases, The 2011 International Workshop on Data Cloud (D-CLOUD 2011), at 2011 International Conference on Cloud and Service Computing (CSC), Hong Kong, China, December 12-14, 2011, *In Proc. CSC 2011*, pp. 330-335.

5.1.1 Summary

An increasing amount of data is stored in cloud repositories, which provide high availability, accessibility, and scalability. The paper investigates the possibility to store and query part of the FLOQ ontology in a cloud based storage, Google Bigtable [7]. To interface Bigtable, a DMI for the Google App Engine [13] was developed to access FLOQ ontology elements stored in a Bigtable repository. To compensate for the limited query capabilities of the GQL [34], the query language of Google App Engine, novel specialized query processing mechanism based on plug-ins called absorbers (later renamed to extractors) and finalizers were developed. Furthermore, a streamed communication protocol provides queries to Bigtable returning large data volumes.

Paper I answers research question three and partly answers research question two.

I am the primary author of this paper. The other authors contributed to discussion and paper writing.

5.2 Paper II

Zhu, M., Stefanova, S., Truong, T., Risch, T. (2014) Scalable Numerical SPARQL Queries over Relational Databases, 4th International workshop on

linked web data management (LWDM 2014) in conjunction with the EDBT/ICDT 2014 Joint Conference, Athens, Greece, March 28, 2014, *In Proc. LWDM 2014*, pp. 257-262.

5.2.1 Summary

The paper investigates the problem of detecting past machine anomalies by querying historical sensor readings stored in a relational database. Typically anomaly detection queries include numerical expressions, inequalities, string matching, and set membership tests inside query conditions. We call such queries *numerical queries*. For scalable execution of numerical queries, numerical operators should be pushed into SQL rather than executed as post-processing filters outside the RDB; otherwise the query execution is slowed down since a lot of data is transported back from the RDB server before the filtering. In addition indexes on the server are not utilized.

The paper presents the *NUMTranslator* algorithm, which transforms numerical and other domain calculus operators into corresponding SQL expressions. The experiments show that *NUMTranslator* substantially improves the query performance in particular when the numerical expressions inside query conditions are highly selective. The algorithm uses a table driven approach to translate numerical domain calculus expressions into corresponding numerical SQL expressions. We compared the performance of the numerical queries with and without applying *NUMTranslator*. We also compared our approach with other systems. In the paper the query language SPARQL was used rather than SQL, showing the FLOQ can process different query languages. Only D2RQ [3] could execute numerical SPARQL queries over RDBs, but substantially slower, since D2RQ does not employ an approach similar to *NUMTranslator*.

Paper II answers research question five.

I am the primary author of this paper, while the other authors contributed with discussions and paper writing. Silvia Stefanova helped with related work while Thanh Truong contributed with some initial implementation work.

5.3 Paper III

Zhu, M., Mahmood, K., Risch, T. (2015) Scalable Queries Over Log Database Collections, 30th British International Conference on Databases (BICOD 2015), Edinburgh, UK, July 6-8, 2015, *In Proc. BICOD 2015*, pp. 173-185.

5.3.1 Summary

Two new join strategies are proposed, *parallel bind-join* (PBJ) and *parallel bulk-load join* (PBLJ), for parallel execution of queries joining meta-data with data from autonomous databases using standard DBMS APIs. A cost model is proposed to guide and evaluate the efficiency of the join strategies. The performance of the two methods is evaluated using data from a real-world application [30], where sensor readings are collected from machines at distributed sites and joined through the FLOQ meta-database at a central site in order to detect unexpected behaviors. For the performance evaluation we define typical fundamental queries to detect anomalies and investigate the impact of our join strategies guided by the cost model. The experimental results validate the cost model. In general, PBLJ performs better than PBJ when the number of bindings from the meta-database is increased and the number of result tuples is small.

Paper III answers research question four and partly answers research question two.

I am the primary author of this paper, while the co-authors helped with discussions and paper writing. Khalid Mahmood contributed to the cost-model for join strategies.

5.4 Paper IV

Mahmood, K., Risch, T., Zhu, M. (2015) Utilizing a NoSQL Data Store for Scalable Log Analysis, 19th International Database Engineering & Applications Symposium (IDEAS 2015), Yokohama, Japan, July 13-15, 2015, *In Proc. IDEAS 2015*, pp. 49-55.

5.4.1 Summary

A potential problem for persisting large volume of data logs with a conventional relational database is that loading massive logs produced at high rates is not fast enough due to the strong consistency model and high cost of indexing. As a possible alternative, a modern NoSQL data store, which sacrifices transactional consistency to achieve higher performance and scalability, can be utilized. In this paper, we investigate to what degree a state-of-the-art NoSQL database can achieve high performance persisting and fundamental numerical queries to analyze data in log databases. For the evaluation, a state of-the-art NoSQL database, MongoDB, is compared with a relational DBMS from a major commercial vendor and with a popular open source relational DBMS. MongoDB is chosen as it provides both primary and secondary indexing needed for anomaly detection, which is essential for scalable processing of queries over large log databases. Our results reveal that relaxing

the consistency does not provide substantial performance enhancement for any of the systems. For high-performance loading of data logs MongoDB is shown to have similar performance as a state-of-the-art relational database, while the query performance of the relational database is usually better.

The main contribution of the paper is a performance evaluation of persisting and analyzing data logs under different consistency configurations, as needed by log databases.

Paper IV partly answers research question two.

I contributed to discussion, reviewed, and proposed changes in the paper writing.

6 Conclusions and Future work

In this Ph.D. project, I started investigating querying data stored in a cloud data store with limited query capabilities compared to a regular relational database. I developed the BigIntegrator system to enable queries combining data from a cloud data store with regular relational databases. The architecture of the system included novel query processing mechanisms based on plug-ins called extractors and finalizers, which must be implemented for each new data manager. The extensions compensate for limited query capabilities of different data managers. Furthermore, due to the quota limitation in transferring data from a cloud data store per request, a streamed communication interface was implemented to enable queries returning big volume of data.

In order to analyze passed behavior of monitored equipment, sensor readings can be stored in relational databases and analyzed with queries. However, queries for machine anomaly detection often involve numerical expressions inside query conditions. The continuation of BigIntegrator, i.e. the FLOQ system, is able to process in a scalable way numerical queries that analyze logged data stored in collections of different kinds of databases.

To efficiently process numerical queries over log databases, the *NUMTranslator* algorithm was developed, which extracts and translates numerical domain calculus expressions into corresponding numerical SQL expressions by using a table driven approach. The approach was evaluated on a benchmark scenario in an industrial setting where logged data stored in a relational database was analyzed using numerical queries. The experiments show that *NUMTranslator* substantially improves the query performance of numerical queries, in particular when the numerical expressions inside query conditions are highly selective.

To process queries over geographically distributed log databases, I developed two new join strategies, parallel bind-join (PBJ) and parallel bulk-load join (PBLJ). For the performance evaluation I defined typical fundamental queries and investigated the impact of the join strategies. A cost model was used to guide and evaluate the efficiency of the strategies. The experimental results validated the cost model. In general, PBLJ performs better than PBJ when the number of bindings from the meta-database is increased and the returned result is small.

Finally, it was shown that a NoSQL data store such as MongoDB is a suitable alternative to relational databases for storing log databases. Based

on this, a MongoDB-DMI [22] was developed to enable FLOQ queries combining logged data stored in MongoDB databases with other data sources.

As future work, the system should be extended to handle new kinds of data sources by developing new DMIs, for example, DMIs to call MapReduce systems [10] that process large data logs as parallel batch jobs. Furthermore, logged data should be combined with streaming data to match on-line data from equipment with historical data in log databases in order to identify how similar situations were previously handled when anomalies are detected in streaming data. To handle expensive analyses over streaming and stale data novel parallel query processing strategies such as *parasplit* [39] can be utilized.

In the experiments a rather small set of autonomous log databases were used. The impact of having a very large number of log databases should be further investigated. Different strategies to improve communication overheads, e.g. by compression, should be investigated.

Extensible indexing techniques can be used for improving the performance of complex numerical queries over logs [35] [36].

Summary in Swedish

Modern produktutveckling genererar stora mängder data under alla dess olika faser, från utveckling och tillverkning, genom användning och underhåll, till vidareutveckling och återvinning. Det är mycket viktigt att kunna lagra, söka och analysera de olika slags data som genereras under produktcykeln för att skapa högkvalitativa och tillförlitliga produkter. Det behövs även metoder för att samla ihop, bearbeta och analysera producerade data.

Avhandlingen är baserad på ett reellt industriellt scenario [30] där maskiner, t.ex. hjullastare, hydrauliska pumpar, eller skärverktyg är spridda över olika anläggningar vid olika geografiska platser och där sensorer på maskinerna producerar stora volymer mätvärden. De data som generas vid varje anläggning representerar tidsstämplade sensorvärden från olika maskinkomponenter (t.ex. oljetemperatur eller tryck) och lagras i lokala *loggdatabaser* vid anläggningen. Dessutom behövs ett effektivt sätt att övervaka och validera att övervakad utrustning fungerar som avsett. Loggdatabaserna används för att finna och analysera onormalt beteende hos de övervakade maskinerna på de geografiskt distribuerade anläggningarna. Samlingen av loggdatabaser är vidare dynamisk i den meningen att nya anläggningar tillkommer och försvinner över tiden.

För att kunna analysera och jämföra data från loggdatabaserna behövs en övergripande s.k. meta-databas som beskriver egenskaper hos övervakad utrustning och dess loggdatabaser, t.ex. olika maskinkonfigurationer vid anläggningarna, vilka typer av sensormodeller som är installerade på de olika maskinerna och vilka toleranser som är aktuella. Meta-databasen tillhandahåller en global vy av tillståndet hos alla maskiner vid alla anläggningar. Genom meta-databasen kan man ställa frågor som spänner över loggdatabaserna och identifierar när övervakade maskiner uppträder eller har uppträtt onormalt. En speciell utmaning som behandlas i avhandlingen är skalbar hantering av frågor som kombinerar data i den övergripande meta-databasen med data från de distribuerade loggdatabaserna.

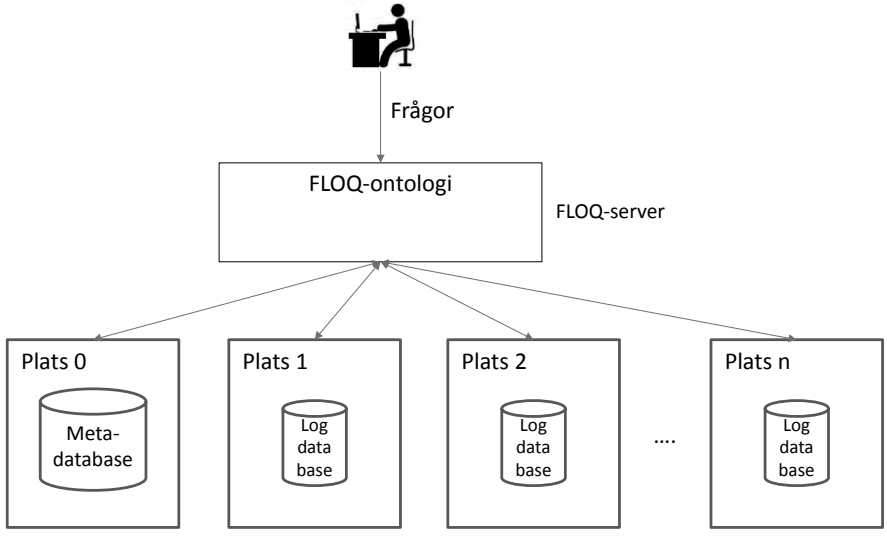
Ett onormalt beteende hos maskiner kan ofta upptäckas genom att identifiera onormala avvikelser i uppmätta värden som lagrats i loggdatabaserna. Sådana avvikelser kan uttryckas som databasfrågor innehållande numeriska villkor, exempelvis att sensorvärden avviker utanför toleransmarginalen för en viss sensormodell under en viss sammanhängande tidsperiod. Detta kräver att systemet kan utföra numeriska frågor över en mängd av loggdatabaser

beskrivna av en meta-databas som innehåller exempelvis dessa toleranser och information om aktuella typer av maskiner.

Följande forskningsfrågeställningar undersöks i avhandlingen:

1. Den övergripande forskningsfrågan är: Hur representerar man en meta-databas som beskriver distribuerad industriell utrustning och dess loggdatabaser?
2. Hur kan olika sorters mjukvara för att hantera databaser användas för lagring av både loggdatabaserna och meta-databasen?
3. Hur kan moln-baserad datalagring användas för att representera meta-databasen?
4. Hur kan systemet effektivt och skalbart utföra frågor som kombinerar meta-databasen med data från de distribuerade loggdatabaserna?
5. Hur kan systemet effektivt utföra numeriska frågor som identifierar onormala mätvärden i loggdatabaserna?

Som en ansats för dessa utmaningar och för att besvara den första forskningsfrågan har vi utvecklat ett system, *FLOQ (Fused Log database Query processor)* illustrerat i Figur 18. FLOQ integrerar samlingar av dynamiska, distribuerade och separata loggdatabaser genom en övergripande meta-databas som kallas *FLOQ-ontologin*. FLOQ-ontologin hanteras av ett system som kallas *FLOQ-servern*.



Figur 18. FLOQ-översikt

FLOQ-ontologin beskriver meta-data och fysiska egenskaper hos industriell utrustning på geografiskt distribuerade anläggningar. FLOQ-ontologin är generell och kan beskriva olika sorters industriell utrustning. Varje anläggning på plats 1, 2, …, n har utrustning som producerar mätvärden från givare lagrade i separata *loggdatabaser*. De olika loggdatabaserna underhålls lokalt

per anläggning oberoende av andra databaser. FLOQ-ontologin inkluderar beskrivningar av dessa loggdatabaser. Även delar av FLOQ-ontologin kan lagras i en extern *meta-databas* som finns på plats 0. Användaren sänder *frågor* till FLOQ-servern för att söka i loggdatabaserna och i meta-databasen. Frågorna formuleras i termer av FLOQ-ontologin. De analyserar ofta data i loggdatabaserna för att upptäcka onormala avvikelser i lagrade mätvärden. Sådana frågor innehåller ofta numeriska villkor, t.ex. för att identifiera onormalt stora skillnader mellan uppmätta och förväntade givarvärden.

FLOQ-ontologin är representerad i en generell datamodell som kan beskriva alla sorters meta-data. Vidare kan både meta-databasen och loggdatabaserna lagras i olika format m.h.a. olika sorters databashanterare. De olika externa datarepresentationerna är avbildade till FLOQ-ontologin. FLOQ tillhandahåller en generell och utbyggbar mekanism för effektivt utförande av frågor över olika sorters databaser, som t.ex. MySQL eller MongoDB [22][25]. Detta ger ett svar på den andra forskningsfrågan.

För att göra delar av FLOQ-ontologin globalt tillgängligt för världsomspännande organisationer tillåter FLOQ att en del av ontologin lagras i en extern moln-databas som Googles Bigtable [7]. Sådana moln-databaser tillhandahåller hög universell tillgänglighet och skalbarhet. FLOQ gör det möjligt att avbilda extern datarepresentation i Bigtable eller andra datarepresentationer till FLOQ-ontologin [40]. Meta-data kan således lagras antingen direkt i FLOQ-ontologins datamodell, i en relationsdatabas, eller i en moln-databas som Bigtable. Denna möjlighet av FLOQ att representera ontologin på olika sätt besvarar forskningsfråga tre.

FLOQ tillhandahåller ett delsystem för att effektivt och skalbart utföra databasfrågor över distribuerade loggdatabaser i termer av FLOQ-ontologin. En mekanism för att modulärt plugga in beskrivningar av olika sorters databashanterare gör det möjligt för systemet att automatiskt dela upp en fråga till FLOQ-ontologin i separata delfrågor som sänds till de distribuerade loggdatabaserna. Två olika strategier förslås för att dela upp frågor som kombinerar meta-data i ontologin med data i loggdatabaserna, *parallel bind-join* (PBJ) och *parallel bulk-load join* (PBLJ) [42]. Denna frågebearbetning över distribuerade loggdatabassamlingar besvarar forskningsfråga fyra.

Skalbar bearbetning av numeriska frågor över loggdatabaser representerade som relationsdatabaser (t.ex. MySQL eller Oracle) kräver att numeriska villkor i största möjligaste mån utförs som lokala SQL-frågor direkt över de olika loggdatabaserna snarare än att data transporteras till FLOQ-servern för filtrering där. *NUMTranslator* algoritmen [41] konverterar numeriska uttryck i termer av meta-databasmodellen till motsvarade SQL-frågor. I [41] visas att *NUMTranslator* förbättrar prestanda väsentligt baserat på frågor som identifierar avvikelser i loggdatabaser lagrade i en relationsdatabas. Detta besvarar forskningsfråga fem.

Acknowledgements

First I would like to thank Professor Tore Risch, you as my main supervisor is most influential to my research project. Thanks for giving me this opportunity to do research in database technology. When given the opportunity, I started my PhD journey in database research. During these years, I have learned a lot coding and system development skills from you. Of course, writing scientific research papers is something new to me at the beginning. But, I have improved a lot during these years, working as a PhD student. Thank you. I would also like to thank the former and current UDBL members, Kjell Orsborn, Erik Zeitler, Manivasakan Sabesan, Silvia Stefanova, Lars Melander, Cheng Xu, Andrej Andrejev, Thanh Truong, Mikael Lax, Sobhan Badiozamany, and Khalid Mahmood. I would like to thank Kjell Orsborn for helping me in the PhD Thesis. Thanks Silvia Stefanova for your help and supervision. I have been roommate with Lars and Andrej and like the time we have been sharing office. Thank you Lars for helping me drive car, buy and transport the furniture when I needed to settle down with my life in Uppsala. Thanks Cheng, I enjoyed talking to you and thanks for your help, especially when I was in some urgent situation. Thanks Thanh, I came to the group a year earlier than you, but slowly I realized we share the most similar view on some aspects, which are not usually analyzed. For political discussion, I was most of the time quiet and try to keep the conversation short. Thanks for the conversation we had and I appreciate it very much. Thanks Mikael for the help and discussion. Thanks Sobhan for the discussion and having lunches together. Thanks Khalid for help and discussion in research and the jokes that made me laugh so much. Thanks for friendship from all you guys. Thanks for the advices given during these years in research or even beyond research. These advices are very good and have helped me a lot in my PhD journey. Thanks for listening to me especially when I had problems in research and felt a bit down.

I would like to thank my friend Jun He. We used to work in the same floor in ICT building 1. Thanks for being soul mate and nice friend to me. I enjoyed so much to talk with you and shared my happiness and sadness. I like your family as well. I appreciate the friendship very much.

I would like to thank Matteo Magnani. I enjoyed the Italian food and had lot of fun in the parties. I like your family as well.

I would like to thank Ulrika Andersson. Thank you for helping me in the issues while living in Sweden. I also thank you for your advices. I also like to thank Anne-Marie Jalstrand and Anna-Lena Forsberg for your help.

Of course, there are a lot of friends and hard to write about everyone here. But I would like to thank you all, thanks for the time having fun with you guys.

For my parents, I would like to credit them with huge gratitude. It was in November 2003 I started the journey of studying and living in abroad. That was the first time I had lived apart from you outside of China and it was such a long distance from you. Without support at the beginning, it might have been impossible for me to continue my journey until today. This journey really changed the track of my life and thanks to my parents supporting me all the time no matter whether I was up or down. Thank you for your love and care. I love you so much and can't express my love to you in words. You and my future family are the most important part in my life.

I would like to thank myself as well, for being persistent and consistent in going through the journey that started in November 2003. I have really learned a lot in research and beyond. It is a unique and beneficial experience that I got from this journey.

Minpeng Zhu

Bibliography

1. Andrejev, A., Risch, T.: Scientific SPARQL: Semantic Web Queries over Scientific Data. In: DESWEB workshop, pp. 5-10 (2012)
2. Andrejev, A., Toor, S., Hellander, A., Holmgren, S., Risch, T.: Scientific Analysis by Queries in Extended SPARQL over a Scalable e-Science Data Store. In: eScience, pp. 98-106 (2013)
3. Bizer, C., Cyganiak, R., Garbers, G., Maresch, O., Becker, C.: The D2RQ Platform v0.7 - Treating Non-RDF Relational Databases as Virtual RDF Graph, http://www4.wiwiss.fu-berlin.de/bizer/d2rq/spec/
4. Brettel, M., Friederichsen, N., Keller, M., Rosenberg, M.: How Virtualization, Decentralization and Network Building Change the Manufacturing Landscape: An Industry 4.0 Perspective. J. International Journal of Mechanical, Aerospace, Industrial, Mechatronic and Manufacturing Engineering. 8(1), 37-44 (2014)
5. Cassandra, http://cassandra.apache.org/
6. Cattell, R.: Scalable SQL and NoSQL data stores. J. ACM SIGMOD Record. 39(4):12-27, 2010
7. Chang, F., et al.: Bigtable: A distributed storage system for structured data. In: OSDI, pp. 205-218 (2006)
8. Codd, E. F.: Relational completeness of database sublanguages. J. Communications of the ACM. 13(6):377-387, 1970
9. CouchDB, http://couchdb.apache.org/
10. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI, pp. 107-113 (2004)
11. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall, Upper Saddle River, NJ, USA (2009)
12. Graefe, G.: Query evaluation techniques for large databases. J. ACM Computing Surveys (CSUR). 25(2):73-169, 1993
13. Google App Engine, http://code.google.com/appengine/docs/whatisgoogleappengine.html
14. Haas, L., Kossmann, D., Wimmers, E., Yang, J.: Optimizing queries across diverse data source. In: VLDB, pp. 276-285 (1997)
15. Hansson, M.: Wrapping External Data by Query Transformations, Master Thesis, Computer. Science no. 260, ISSN 1100-1836, Uppsala University (2003)
16. HBase, http://hbase.apache.org/
17. Hägglunds, http://www.boschrexroth.com/en/xc/company/haegglunds-landingpage/haegglunds-landingpage

18. Hägglunds direct drive system,
http://dc-america.resource.bosch.com/media/us/press_kits_1/workboat/ brochures_1/Rexroth-Hagglunds-Drive-Systems.pdf
19. Industrial Internet: Pushing the boundaries of minds and machines,
http://www.ge.com/docs/chapters/Industrial_Internet.pdf
20. Lee, J., Bagheri, B., Kao, H. -A.: A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing system. J. Manufacturing Letters. 3: 18-23, 2015
21. Made in China 2025,
http://csis.org/publication/made-china-2025
22. Mahmood, K.: Scalable Persisting and Querying of Streaming Data by Utilizing a NoSQL Data Store, Master Thesis IT 14 021, Department of Information Technology, Uppsala University (2014)
http://www.it.uu.se/research/group/udbl/Theses/KhalidMahmoodMSc.pdf
23. Mahmood, K., Risch, T., Zhu, M.: Utilizing a NoSQL Data Store for Scalable Log Analysis. In: IDEAS, pp. 49-55 (2015)
24. Memcached,
http://www.memcached.org/
25. MongoDB,
http://docs.mongodb.org/
26. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., Dewitt, D.J., Madden, S., Stonebraker, M.: A Comparison of Approaches to Large-Scale Data Analysis. In: SIGMOD, pp. 165-178 (2009)
27. Redis,
http://redis.io/
28. Risch, T., Josifovski, V., Katchaounov, T.: Functional data integration in a distributed mediator system. In: Gray, P.M.D., Kerschberg, L., King, P.J.H., Poulovassilis, A. (eds.): The Functional Approach to Data Management - Modeling, Analyzing, and Integrating Heterogeneous Data. pp. 211-238. Springer, Verlag (2004)
29. Shoshani, A., Rotem, D.: Scientific Data Management: Challenges, Technology, and Deployment. Taylor and Francis Group, Boca Raton, FL, USA (2009)
30. Smart Vortex Project, http://www.smartvortex.eu/
31. Snodgrass, R.T.: Temporal databases. In: Frank, A.U., Campari, I., Formentini, U. (eds.): Theories and Methods of Spatio-Temporal Reasoning in Geographic Space. vol. 639, pp. 22-64. Springer, Verlag (1992)
32. Stonebraker, M.: SQL databases v. NoSQL databases. J. Communications of the ACM. 53(4):10-11, 2010
33. SPARQL 1.1 Query Language,
https://www.w3.org/TR/sparql11-query/
34. The GQL reference web page,
http://code.google.com/appengine/docs/python/datastore/gqlreference.html
35. Truong, T., Risch, T.: Scalable Numerical Queries by Algebraic Inequality Transformations. In: DASFAA, pp. 95-109 (2014)
36. Truong, T., Risch, T.: Transparent inclusion, utilization, and validation of main memory domain indexes. In: SSDBM, Article No. 21 (2015)
DOI= 10.1145/2791347.2791375
37. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems. Springer Science+Business Media, New York (2011)

38. Xu, C., Wedlund, D., Helgoson, M., Risch, T.: Model-based Validation of Streaming Data. In: DEBS, pp. 107-114 (2013)
39. Zeitler, E., Risch, T.: Massive scale-out of expensive continuous queries. J. ACM VLDB. 4(11):1181-1188, 2011
40. Zhu, M., Risch, T.: Querying combined cloud-based and relational databases. In: CSC, pp. 330-335 (2011)
41. Zhu, M., Stefanova, S., Truong, T., Risch, T.: Scalable Numerical SPARQL Queries over Relational Databases. In: LWDM workshop, pp. 257-262 (2014)
42. Zhu, M., Mahmood, K., Risch, T.: Scalable Queries Over Log Database Collections. In: BICOD, pp. 173-185 (2015)

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1343

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-275044

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2016

Paper I

DOI= 10.1109/CSC.2011.6138543

The paper is reformatted for typographic consistency.

Querying Combined Cloud-Based and Relational Databases

Minpeng Zhu and Tore Risch
Department of Information Technology, Uppsala University, Sweden
Minpeng.Zhu@it.uu.se Tore.Risch@it.uu.se

Abstract— An increasing amount of data is stored in cloud repositories, which provide high availability, accessibility, and scalability. However, for security reasons enterprises often need to store the core proprietary data in their own relational databases, while common data to be widely available can be stored in a cloud data repository. For example, the subsidiaries of a global enterprise are located in different geographic places where each subsidiary is likely to maintain its own local database. In such a scenario, data integration among the local databases and the cloud-based data is inevitable. We have developed a system called BigIntegrator to enable general queries that combine data in cloud-based data stores with relational databases. We present the design and working principle of the system. A scenario of querying data from both kinds of data sources is used as illustration. The system is general and extensible to integrate data from different kinds of data sources. A particular challenge being addressed is the limited query capabilities of cloud data stores. BigIntegrator utilizes knowledge of those limitations to produce efficient query execution.

Keywords: cloud data repository; relational database; data integration; Bigtable;

I. Introduction

Cloud based repositories such as Google's Bigtable [1] allow widely accessible distributed data stores to be queried by the query language GQL [7]. This is done by web-based applications managed by the Google App Engine (GAE) [10]. GAE provides an application environment and query language to manage data stored in Google's cloud. It is easy to write web-based applications that access and update these cloud-based databases.

Cloud repositories such as Google's Bigtable are particularly useful to store data that has to be globally available. For example, in industrial settings, machines such as engines, trucks, cutting tools, etc., produce many different kinds of data and the machines are often geographically widely

distributed and maintained locally. To check that distributed equipment works properly, it is crucial to analyze its working status by searching the data produced by the equipment. Since the equipment is widely distributed, properties about the equipment should be stored in an environment that provides high availability and universal access, such as a cloud-based data store.

Relational database systems (RDBMSs) have the limitation that they must run in some central server site and therefore require substantial maintenance efforts to provide high availability. As an alternative approach, we propose to store common data, for instance equipment properties, in a cloud-based data store, such as Bigtable. By using such a cloud service the data becomes universally available and can easily be maintained. However, the data stored in the cloud often needs to be combined with data stored in regular databases. For example, cloud-based data is used for finding the locations of a particular machine, while the information about the machines' operating environments is stored in local relational databases. A maintenance engineer may wish to make queries combining relational data with the cloud-based data. To enable this, there is need for a system supporting queries combining cloud-based data and data in relational databases. We have developed such a system, BigIntegrator, to transparently process queries combining data stored in Bigtable data stores and data stored in relational databases.

BigIntegrator utilizes a novel query processing mechanism to provide easy extension of data integration from different kinds of data sources. The mechanism is based on plug-ins called *absorbers* and *finalizers*. The limited expressiveness of GQL has to be taken into account by BigIntergrator's query processor, which is the challenge being addressed by the absorbers and finalizers.

GQL has some similarities with SQL but has very limited query expressions in order to provide for scalable processing. BigIntegrator can process queries to such data sources with limited back-end query languages support. The absorber and finalizer for Bigtable data sources know the limitations of GQL and will pre and post-process those operations that cannot be processed by the data sources. For this, BigIntegrator generates integrating execution plans containing calls to relational databases, Bigtable data stores, and local operators.

In summary the contributions of our work are:

- The BigIntegrator system provides query capabilities over combined cloud-based and relational databases.
- A novel query processing mechanism based on plug-ins for absorbers and finalizers is developed to allow easy extensions for each new kind of data source that provide a restricted query language.
- A client-server architecture for scalable querying of Bigtable data repositories is developed.

The rest of the paper is organized as follows: Section II discusses related work. Section III illustrates the system by a scenario from an industrial equipment point of view. Section IV overviews the BigIntegrator system architecture and describes its query processing. Conclusions and future work are described in section V.

II. Related work

There are several cloud-based storage systems available, such as Dynamo [8], PNUTS [5], and Bigtable [1]. These systems have very limited query languages as a compromise for very high scalability. The restricted queries do not allow joins and there are restrictions on how to specify the query conditions. In contrast, the BigIntegrator pushes as much query processing as possible to the data sources and compensates the lacking query capability of a data source by doing post-query processing with its own query engine. Similar approaches can be applied on [8, 5] as well.

Some cloud-based storage systems such as Cloudy [3] provide rather complete SQL capabilities. It offers key-value, SQL, and XQuery interfaces to manipulate its cloud data. Microsoft SQL Azure [2] offers full SQL language support for its cloud-based relational database. Unlike Cloudy and SQL Azure, the purpose of BigIntegrator is to allow joining of data from a restricted cloud-based data store such as Bigtable with relational databases, by generating execution plans that combine queries sent to the data sources.

Unlike classical work on mediator/wrapper techniques over conventional databases such as [4], BigIntegrator provides data integration between cloud-based data repositories and relational DBMSs. Furthermore, a novel query plug-in mechanism based on absorbers and finalizers is developed to provide easy extensions for new kinds of data sources providing restricted query languages.

To conclude, most work on cloud-based databases concentrates on providing scalability, availability and consistency as storage services inside a cloud. No other system addresses the problem of integrating data from cloud-based databases having restricted query languages with relational databases. We show the extensibility of the system and the advantages of its novel query plug-in mechanisms.

III. Scenario

In this section, we present a scenario combining data from Bigtable and a local relational database. An enterprise is responsible for maintaining geographically widely distributed industrial equipment. Some generally availa-

ble data about the equipment is stored in a cloud repository, while data about local personnel is in relational databases. BigIntegrator enables queries combining these databases.

The database schema for the cloud based database is shown in Fig. 1 and for the relational one in Fig. 2. The cloud table *Machine(Model, Name, Manufacturer)* stores general data about industrial machines such as its model identifier, name, and manufacturer. The table *Site(SID, Name, Country, Region)* stores information about each site such as site ID, its name, and the country and region where it is located. The table *MachineInstallation(MID, Model, SID)* stores information about each installation of a machine at some site, i.e the identifier of the machine, its model, and the identifier of the site where it is located *(SID)*. The attribute *Model* is foreign key from *MachineInstallation* to *Machine* and the attribute *SID* is foreign key from *MachineInstallation* to *Site*. The tables *Machine, Machineinstallation,* and *Site* provide globally accessible common data and are therefore stored in the cloud.

A country maintains its local personnel database in the relational database in Fig. 2. The table *Operator(PID, Name, Skill, Operates)* stores the identifier of a machine operator along with his name, specialty, and the machine he is currently operating. The attribute *Operates* is foreign key from the local database table *Operator* to the cloud database table *MachineInstallation*. Fig. 3 shows all the tables in this scenario with populated data.

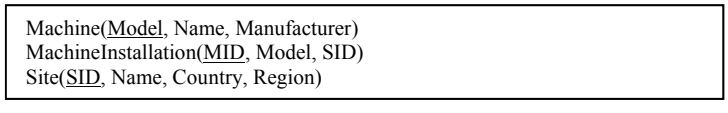


Figure 1. Cloud database schema

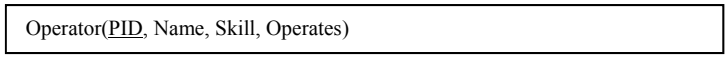


Figure 2. Relational database schema

Model	Name	Manufacturer
1	M1	Volvo
2	M2	Volvo
3	M3	Volvo
4	M4	Volvo
5	M5	Volvo

Machine table

SID	Name	Country	Region
1	Uppsala	Sweden	Uppland
2	Chengdu	China	Si Chuan
3	Campinas	Brazil	Sao Paulo
4	Chapaevsk	Russia	Samara
5	Monki	Poland	Bialystok

Site table

MID	Model	SID
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	1	1
7	2	2
8	3	3
9	4	4
10	5	5

MachineInstallation table

PID	Name	Skill	Operates
1	John	Operation	1
2	Oliver	Operation	2
3	Bruce	Operation	3
4	Carl	Operation	4
5	Thomas	Operation	5
6	Jens	Operation	6
7	Lucas	Operation	7
8	Alex	Operation	8
9	Ryan	Operation	9
10	Wes	Operation	10

Local table Operator

Figure 3. Scenario database schema

The following is an SQL query to BigIntegrator that combines data from the cloud-based tables at a data source named *A* (the Bigtable data source) and the relational database table at data source named *B* (the country's local data source):

```
select i.Mid, o.Name
from Machine_A m, MachineInstallation_A i, Site_A s, Operator_B o
where m.Name = 'M1' and
        m.Manufacturer like 'V%' and
        s.Region = 'Uppland' and
        s.Sid = 1 and
        m.Model = i.Model and
        i.Sid = s.Sid and
        i.Mid = o.Operates
```

The query retrieves identities of machines of model "M1" along with the operators' names, where the machines' manufacturer names starts with "V", the machines are installed in the region "Uppland", and the site code is equal to one.

Every time a data source is accessed the system automatically generates a set of relations called the *source predicates* representing the collections inside the source. The source predicates are references as tables in the SQL queries. In the example there are the source predicates *Machine_A, MachineInstallation_A, Site_A,* and *Operator_B*. The name of each collection in a source named *X* is suffixed by *"_X"*. After the query is executed, BigIntegrator returns the following query result:

1, John
6, Jens

IV. System Overview

The BigIntegrator system architecture is shown in Fig. 4. The system contains two sub-systems: The *RDBMS wrapper* and the *Bigtable wrapper*. A wrapper needs to be implemented for each *kind* of data source to be queried from the BigIntegrator system. The RDBMS wrapper generates SQL queries sent to a back-end RDBMS, while the Bigtable wrapper generates GQL queries to data stored in Bigtable.

The system receives SQL queries, which are processed to generate a query execution plan that contains calls to the underlying relational and Bigtable databases. The wrapper modules have plug-ins that know how to generate queries to each kind of data source.

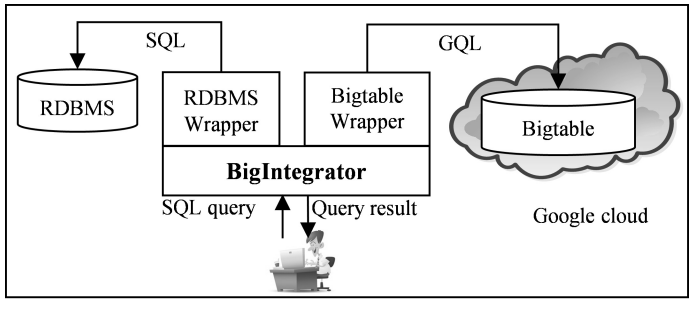


Figure 4. BigIntegrator architecture

A. BigIntegrator Wrappers

Fig. 5 shows the components of a wrapper definition for a BigIntegrator data source.

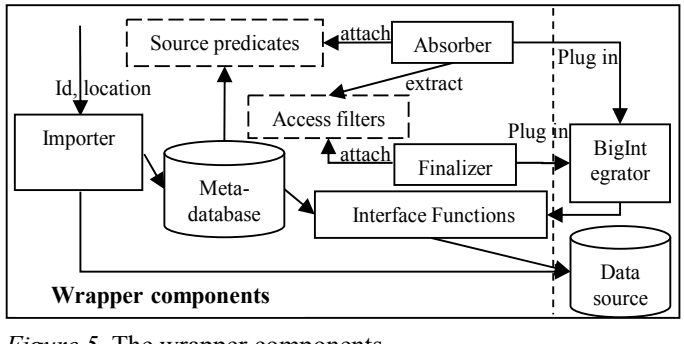


Figure 5. The wrapper components

For each new kind of data source the components *importer, absorber, finalizer,* and *interface function* have to be developed. Once a wrapper is defined any data source of that kind can be wrapped by creating a source identifier *id* for the source and then calling a system procedure *import(id,location),* which accesses the location and imports system catalog data to the local *meta-database*. When a data source is wrapped it can be used in SQL queries and joined with other wrapped data sources.

Each data source can contain many collections presented to the system as *source predicates*. The importer creates the source predicates and stores them in the local meta-database. Each wrapper has one *absorber*, which is a plug-in that from a user query extracts a subquery, called the *access filter*. It selects data from a particular source predicate, based on the capabilities of the source. Each wrapper also has a *finalizer*, which is a plug-in that translates each access filter in the plan to an algebra operator called an *interface function*, specific for each kind of source. The interface function sends a query to the data source (i.e. a GQL or SQL query).

B. The BigIntegrator query processor

The steps of the query processor in BigIntegrator are shown in Fig. 6.

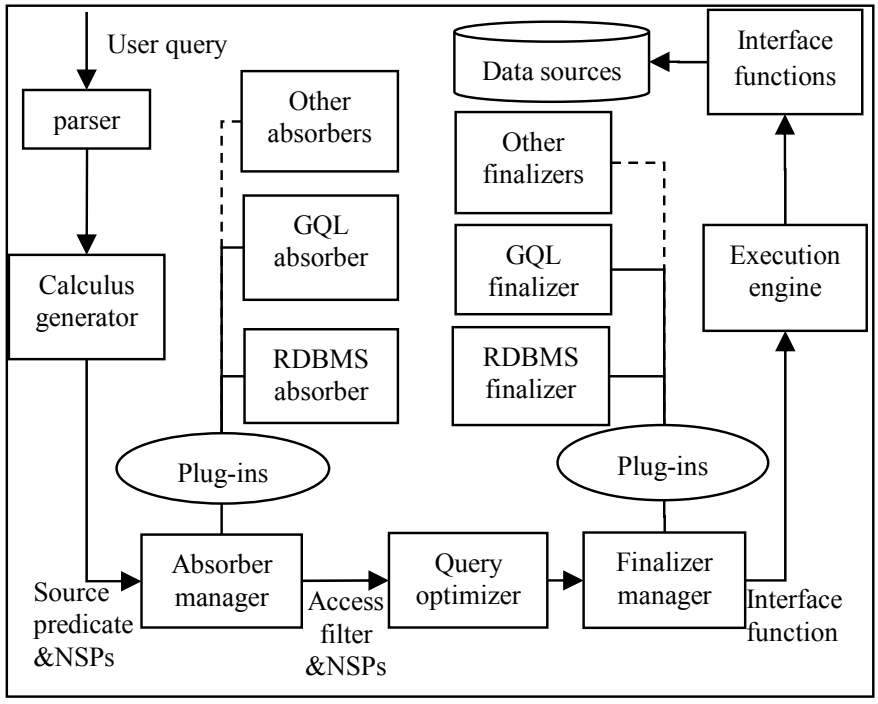


Figure 6. Query processing in BigIntegrator

The *parser* translates the SQL query into a parse tree, which the *calculus generator* transforms into a Datalog [6] query. The Datalog query will contain both source predicates and *non source predicates (NSPs)*. The *absorber manager* takes the Datalog query and, for each source predicate referenced in the query, calls the corresponding absorber of its wrapper. In order to replace the source predicate with an access filter, the absorber collects from the query the source predicates and the possible other predicates, based on the capabilities of the data source. The *query optimizer* reorders the access filters and other predicates to produce an algebra expression containing calls to both access filters and NSP operators. The *finalizer manager* takes the algebra expression and, for each access filter operator referenced in the algebra expression, calls the corresponding finalizer of its wrapper. The finalizer transforms the access filters into interface function calls. To access the different data sources, the *execution engine* interprets the finalized algebra expression calling the interface functions.

The example query is transformed by the parser and calculus generator into the following Datalog query:

```
Query1(mid, name3) :-
Machine_A(model,name1,manufacturer) AND
MachineInstallation_A(mid,model,sid)AND
Site_A(sid, name2, country, region) AND
Operator_B(pid,name3,skill,operates)AND
name1 = 'M1' AND
manufacturer like 'V%' AND
region = 'Uppland' AND sid = 1
```

The NSPs are in bold phase. Unique variable names are generated when needed, e.g. *name1, name2* and *name3*.

In this example, the GQL absorber for the source predicate *Machine_A(model, name1, manufacturer)* will absorb *name1 = 'M1'* since the predicate = can be handled by a GQL data source and both predicates share the same parameter *name1*.

The capabilities of a data source can vary widely, e.g. joins are allowed in RDBMS data sources but not in Bigtable data sources. If joins are allowed, as in SQL, an access filter is formed as a conjunction of all relational source predicates and supported NSPs. If joins are not allowed, as for Bigtable sources, each source predicate forms its own access filter based on GQL language constraints.

The access filters are represented as Datalog rules. In the example there will be one access filter created for each of the source predicates *Machine_A* (filter *F1*), *MachineInstallation_A* (filter *F2*), *Site_A* (filter *F3*), and *Operator_B* (filter *F4*):

```
F1(model,name1,manufacturer):-
  Machine_A(model,name1,manufacturer) AND
  name1='M1'

F2(mid,model,sid):-
  MachineInstallation_A(mid,model,sid)AND
  sid = 1

F3(sid,name2,country,region):-
 Site_A(sid,name2,country,region) AND
 region ='Uppland' AND sid =1

F4(pid,name3,skill,operates):-
Operator_B(pid, name3, skill, operates)

Query1(mid,name3):-
F1(model,name1,manufacturer)AND
F2(mid,model,sid)AND
F3(sid,name2,country,region)AND
F4(pid,name3,skill,operates)AND
manufacturer like 'V%'
```

The possible NSPs are placed in all the access filters for which they have a shared source predicate parameter. For example, *Sid = 1* is placed in both *F2* and *F3*. In other word, the NSPs can be absorbed into one or several access filters.

If an NSP cannot be placed in any access filter, it will remain as a separate predicate in the query and post-processed by BigIntegrator. In the example, *Manufacturer like 'V%'* remains as a separate predicate even though it shares variable *manufacturer* with the GQL access filter *F1*, since GQL does not support *like* predicates. An absorber contains rules about what NSPs can be absorbed into the access filter according to the query capability of the data source. GQL queries have the following restrictions [7]:

> Suppose A, B, and C are attributes names of a table in a GQL data source, and x, y, a, and z are constants or strings. Then the following *where* clauses of a GQL query are allowed:
> where A = x
> where A < x
> where A > x and A < y
> where A > x and A < y and B = z
> where A > x and A < y and B = z and C = a etc

Accordingly, we define the following heuristic algorithm for the GQL absorber:

1. Absorb all equalities having one variable in common with the source predicate while the other parameter is known.

2. Absorb the first inequality having one variable in common with the source predicate also having the other parameter known.
3. If an inequality is absorbed in 2. then also absorb the first inverse inequality for the same variable.

Unlike GQL, SQL can handle joins. Therefore, the absorber for the RDBMS wrapper absorbs several source predicates to produce joins. This is not elaborated here.

The access filters (F1, F2, F3 and F4) and the NSPs that cannot be absorbed into any access filter, are combined into a conjunctive form and sent to the *query optimizer* for optimization. A greedy query optimization method [6] is employed to find an optimized plan fast.

The finalizer manager takes the optimized algebra expression and, for each access filter referenced in the algebra expression, calls the finalizer of the access filter's wrapper. The finalizer translates the access filter into an interface function call to the source.

In the final plan, BigIntegrator's query execution engine calls the interface functions. An interface function sends the query to a data source for execution. For the example query, the finalizer manager finalizes the query execution plan shown in Fig. 7. Bind joins [12] in this example combine each result tuple of F3 and F5 as the input for F2.

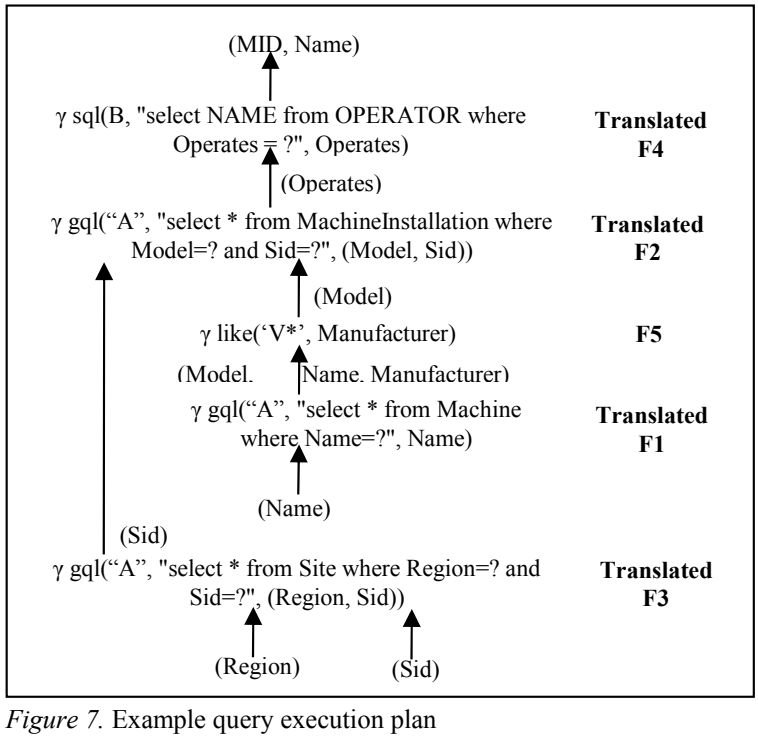


Figure 7. Example query execution plan

The execution plan contains several algebra expression with calls to the *apply* operator γ [11].

The interface function *gql* is an interface function with the signature:

gql(Charstring dsn, Charstring query, Tuple params) -> Stream result

The *gql* function sends a parameterized GQL *query* with parameter *params* to the Bigtable data source *dsn* for execution and returns a stream of tuples, *result*. The *"?"* in a GQL string is substituted with a corresponding parameter value.

Analogously the interface function *sql* has the signature:

sql(Relational ds, Charstring query, Vector params) -> Stream result

The function *sql* sends a parameterized SQL *query* with parameter *params* to RDBMS data source *ds* for execution and returns a stream of tuples, *result*.

In the execution plan the interface function call *gql("A", "select * from Site where Region=? and Sid=?", (Region,Sid))* returns a stream of tuples *(Sid)*. The interface function *gql("A", "select * from Machine where Name=?", Name)* returns a stream of tuples *(Model, Name, Manufacturer)*. The *like* operator returns the filtered stream of tuples *(Model)*. Each combination of tuples from *(Model, Sid)* is input for the interface function call *gql("A", "select * from MachineInstallation where Model=? and Sid=?", (Model, Sid))*, producing a stream of tuples *(Operates)*, which is fed to the interface function call *sql(B, "select NAME from OPERATOR where Operates = ?", Operates),* producing the final result.

The BigIntegrator automatically generates algebra operators for the NSPs that can't be absorbed into any access filter to post-process them by its query engine. For example, *like('V*', Manufacturer)*. This compensates for the lack of a *like* function in GQL.

C. The Bigtable wrapper

1) Architecture

The Bigtable wrapper includes the server and client components shown in Fig. 8.

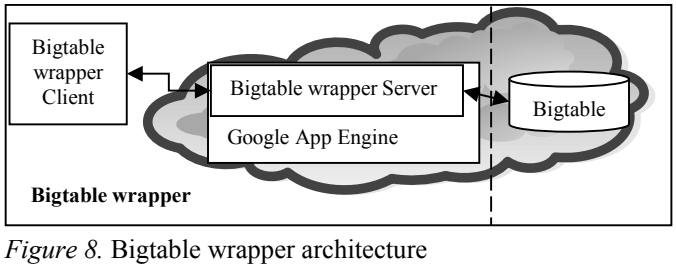


Figure 8. Bigtable wrapper architecture

The Bigtable wrapper server is a web application written in Python served by GAE. It manages the requests from the Bigtable wrapper client. The http protocol is used for the communication between the client and the server. The interface function sends a query request to the server, which forwards the GQL query to Bigtable using the Python Datastore API [9]. The Bigtable wrapper server then sends back the query result to the Bigtable wrapper client.

GAE limits the size of a query result. This is a problem when a GQL query returns a large result. Another problem is that there is a 30 seconds limit on the response time for a request. This is a problem if the server is running longer time than the limit or returns a too large result. Therefore the server delivers the query results in chunks. This is implemented through the cursor facility of the Python Datastore API. Fig. 9 illustrates the Bigtable wrapper client and server communication.

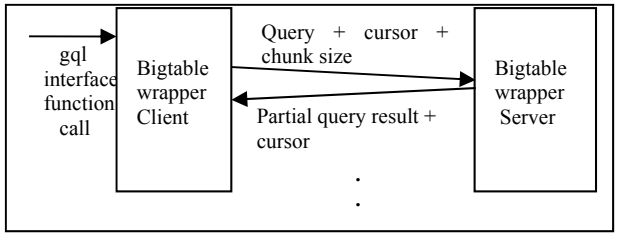


Figure 9. Bigtable wrapper client-server communication

For a given *gql* interface function call, the client sends the GQL query, the cursor information, and the chunk size to the server. The Bigtable wrapper server retrieves the chunks one by one by several *next* requests from the Bigtable wrapper client until the entire result is transmitted to the client. To be able to separate cursors from different queries the cursor handle is shipped back with each result and used in the *next* calls to move the cursor forward.

2) The Bigtable wrapper client and server components

Fig. 10 illustrates the Bigtable wrapper client and server components.

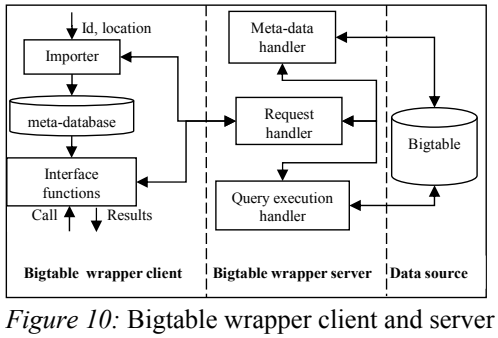


Figure 10: Bigtable wrapper client and server components

Every web application in GAE has its own application identifier (e.g. *http://application-id.appspot.com/*), which is specified when the application is created. A Bigtable wrapper server is a GAE web application and therefore has a unique URL. The location (URL) is used by the importer of the Bigtable wrapper client to establish an http connection to the Bigtable wrapper server. The *importer* first sends a request to the Bigtable wrapper server to collect the meta-data of the Bigtable database. The *request handler* routes the request to the *meta-data handler*. The retrieved meta-data is sent back to the importer by the request handler. The importer stores the meta-data (e.g. source predicate definitions) in the client's *meta-database*. The request handler passes query requests to the *query execution handler*, which calls the Python Datastore API to execute the GQL query. The query results are then sent back to the client through the request handler.

V. Conclusions and Future Work

We presented the BigIntegrator system, which enables SQL queries joining data stored in a Bigtable data repository and in local relational databases. A novel query processing mechanism based on plug-ins for absorbers and finalizers implements extensions for each new kind of data source having limited query capabilities. We presented the architecture of the system. The Bigtable wrapper provides communication between a client computer running the BigIntegrator engine and a Bigtable wrapper server managed by GAE running in a cloud. A communication mechanism provides streamed communication between the Bigintegrator system and the Bigtable wrapper server.

As future work, we plan to evaluate the scalability of the system and develop strategies to improve the system's performance by parallelization.

ACKNOWLEDGMENT

This work is supported by the Swedish Foundation for Strategic Research under contract RIT08-0041.

REFERENCES

1. F. Chang et al, "Bigtable: A distributed storage system for structured data," in OSDI, 2006, pp. 205-218.
2. D. Campbell, G. Kakivaya and N. Ellis, "Extreme scale with full SQL language support in Microsoft SQL Azure," in SIGMOD, 2010.
3. D. Kossmann, T. Kraska, S. Loesing, S. Merkli, R. Mittal, and F. Pfaffhauser, "Cloudy: A modular cloud storage system," in *Proc*. VLDB 2010, Vol. 3, No. 2.
4. V. Josifovski, P. Schwarz, L. Haas, and E. Lin, "Garlic: A new flavor of federated query processing for DB2," in ACM SIGMOD, 2002.

5. B. F. Cooper et al, “PNUTS: Yahoo!’s hosted data serving platform,” in VLDB, 2008.
6. W. Litwin and T. Risch, “Main memory oriented optimization of OO queries using type data log with foreign predicates,” in IEEE Transactions on Knowledge and Data engineering, 1992, pp. 517-528.
7. The GQL reference web page, published online http://code.google.com/appengine/docs/python/datastore/gqlreference.html
8. G. DeCandia et al, “Dynamo: amazon's highly available key-value store,” in SOSP, 2007.
9. The Python Datastore API reference web page, published online http://code.google.com/appengine/docs/python/datastore/
10. Google App Engine, published online http://code.google.com/appengine/docs/whatisgoogleappengine.html
11. G. Fahl and T. Risch, “Query processing over object views of relational data,” The VLDB Journal, Vol. 6 No.4, November 1997, pp. 261-281.
12. L. Haas, D. Kossmann, E.Wimmers, and J. Yang, “Optimizing queries across diverse data source,” in *Proc*. VLDB 1997, Athens, Greece.

Paper II

Minpeng Zhu, Silvia Stefanova, Thanh Truong, and Tore Risch.
2014. Scalable Numerical SPARQL Queries over Relational Databases. In Proceedings of the 4th International workshop on linked web data management (LWDM 2014) in conjunction with the EDBT/ICDT 2014 Joint Conference, Athens, Greece, March 28, 2014.

Scalable Numerical SPARQL Queries over Relational Databases

Minpeng Zhu, Silvia Stefanova, Thanh Truong, Tore Risch
Department of Information Technology, Uppsala University
Box 337, SE-75105 Uppsala, Sweden
{Minpeng.Zhu, Silvia.Stefanova, Thanh.Truong, Tore.Risch}@it.uu.se

ABSTRACT
We present an approach for scalable processing of SPARQL queries to RDF views of numerical data stored in relational databases (RDBs). Such queries include numerical expressions, inequalities, comparisons, etc. inside FILTERs. We call such FILTERs *numerical expressions* and the queries - *numerical SPARQL queries*. For scalable execution of numerical SPARQL queries over RDBs, numerical operators should be pushed into SQL rather than executing the filters as post-processing outside the RDB; otherwise the query execution is slowed down, since a lot of data is transported from the RDB server and furthermore indexes on the server are not utilized. The *NUMTranslator* algorithm converts numerical expressions in numerical SPARQL queries into corresponding SQL expressions. We show that NUMTranslator improves substantially the scalability of SPARQL queries based on a benchmark that analyses numerical logs stored in an RDB. We compared the performance of our approach with the performance of other systems processing SPARQL queries to RDF views of RDBs and show that NUMTranslator improves substantially the scalability of numerical queries compared to the other systems' approaches.

Keywords
SPARQL queries; RDF views of relational databases; numerical expressions; query rewrites; query optimization

1 Introduction

The Semantic Web provides uniform data representation for integrating data from different data sources by using established well-known formats like RDF, RDFS, OWL, and the standard query language SPARQL. Semantic Web seems promising to integrate and search industrial data [2].

Our application scenario is from the industrial domain, where sensors on machines such as trucks, pumps, kilns, etc., produce large volumes of log

data. Such log data describes measured values of certain components at different times and can be used for analyzing machine behavior. Furthermore, the geographic locations of machines are often widely distributed and maintained locally in autonomous RDBs called *log databases*. We are developing the FLOQ (Federated LOg database Query) system, which is a system for historical analyses over federations of autonomous log databases using SPARQL queries. To discover abnormal machine behaviors, a user of FLOQ defines SPARQL queries to these log databases. FLOQ processes a SPARQL query by first finding the relevant log databases containing the desired data, then sending local SPARQL queries to them, and finally collecting the local query results to obtain the final result.

In this paper we concentrate on scalable historical analyses by SPARQL queries of log data stored in a single relational database. Suspected abnormal machine behaviors are discovered and analyzed by specifying numerical SPARQL queries to an RDF view of the RDB. The queries analyze log data through numerical FILTERs containing numerical operators [11]. For example, query *Q1* retrieves the machine identifiers *m* for which a sensor has measured values *mv* of measurement class *A* higher than the expected values *ev* by a threshold value *@thA* during the time from *bt* to time *et*. Here <*prod*> denotes the URI for the RDF view of the RDB.

```
Q1:
SELECT ?m ?bt ?et
FROM <prod>
WHERE {?measuresA  log:mA_BySensor  ?sensor.
       ?measuresA  log:mA/bt        ?bt.
       ?measuresA  log:mA/et        ?et.
       ?measuresA  log:mA/m         ?m.
       ?measuresA  log:mA/mv        ?mv.
       ?sensor     log:sensor/ev    ?ev.
       FILTER (?mv > (?ev + @thA))         }
```

In FLOQ, SPARQL queries to RDBs are processed by generating a local execution plan containing calls to one or several SQL queries sent to a backend RDBMS for evaluation. SPARQL queries that cannot be completely processed by SQL are instead partially processed by an execution plan interpreter in FLOQ. However, in order for the SQL queries to return the minimal required data, it is desirable that as much as possible of the SPARQL query is translated to SQL [8].

In FLOQ numerical SPARQL queries are defined over an automatically generated RDF view over an RDB expressed in *ObjectLog* [6], which is a Datalog dialect that supports objects for representing URIs and typed literals [9], disjunctive queries for UNION expressions, and foreign predicates to represent numerical operators in queries. The SPARQL queries are parsed into ObjectLog queries to the RDF view. Internally representing queries in ObjectLog permits domain calculus query transformations and optimizations

before generating the execution plan. Calls to tuple calculus SQL query strings are made as foreign predicates. Foreign predicates are also used for accessing URIs in the execution plan. Doing all processing in the RDB is complicated, and requires implementing SPARQL operators not supported by SQL as RDB-specific UDFs. We show that ObjectLog query transformations enable scalable execution by the RDBMS.

Numerical SPARQL queries contain variables bound to numbers and calls to numerical functions and operators. For scalable execution, it is important that such numerical expressions are pushed into corresponding SQL expressions and executed on the RDBMS server, which is the subject of this paper. The NUMTranslator algorithm converts numerical SPARQL queries into SQL queries where numerical expressions are pushed into SQL. For example, *Q1* is converted into SQL query *SQL1*, where the numerical expression in the SPARQL FILTER is translated into a corresponding SQL expression.

```
SQL1:
SELECT m.m, bt, et
FROM MeasuresA m, SENSOR s
WHERE m.m=s.m AND
        m.s=s.s    AND
        m.mv > s.ev + @thA
```

A particular problem is that SPARQL and ObjectLog are domain calculus languages where variables can be bound to numbers, while SQL is a tuple calculus language where variables have to be bound to tuples in relations. The NUMTranslator algorithm translates domain calculus expressions into corresponding SQL tuple calculus expressions after having applied domain calculus transformation on the ObjectLog representation.

We show that NUMTranslator improves substantially the query performance for numerical SPARQL queries compared to other approaches used by other systems.

In summary the contributions are:

- We propose a table driven approach to translate numerical domain calculus operators into numerical SQL tuple calculus operators.
- We present the NUMTranslator algorithm that extracts numerical ObjectLog expressions and translates them into corresponding numerical SQL expressions.
- We compare the performance of numerical SPARQL queries to RDF views of RDBs with and without applying NUMTranslator, and show that the algorithm substantially improves the query performance.
- We compare the performance of our approach with the performance of other systems processing SPARQL queries over RDF views of RDBs and show substantially better performance.

The rest of this paper is organized as follows: Section 2 presents a scenario where the approach is applicable. Section 3 overviews the system architecture. Section 4 describes the NUMTranslator algorithm. Section 5 discusses performance experiments. Section 6 describes related work. Conclusions and future work are described in section 7.

2 Motivating Scenario

We present a common scenario from an industrial setting where it is desirable to analyze historical log data in order to find abnormal machine behavior. Log data from embedded sensors is stored in a relational log database.

Figure 1 shows the schema of the RDB storing log data measured by sensors embedded in machine installations. Table *Machine(m, mm)* stores metadata about each machine installation, i.e. machine identifier and model name. The table *Sensor(m, s, sm, mc, ev, ad, rd)* stores information about each sensor installation, i.e. the machine installation *m* where a sensor *s* is embedded, sensor model name *sm*, the kind of measurement (measurement class) *mc*, expected sensor value *ev*, absolute error *ad* and relative error *rd*. The attribute *mc*, measurement class is used to identify different kind of measurements, e.g. oil pressure, temperature, etc. The tables *MeasuresA(m, s, bt, et, mv)* and *MeasuresB(m, s, bt, et, mv)* store log data of kind *A* and *B* read from sensors *s* embedded in machine installations *m*. The begin time *bt* and the ending time *et* for a sensor reading are also stored, while the measured value for a certain time stamp is denoted by *mv*. The columns *m*, (*m, s*), and (*m, s, bt*) are primary keys in the tables *Machine*, *Sensor*, and *MeasuresA and MeasuresB*, respectively. The column *m* in tables *MeasuresA*, *MeasuresB,* and *Sensor* references the column *m* in the table *Machine* as foreign key. Furthermore, columns (*m, s*) in tables *MeasuresA* and *MeasuresB* reference columns (*m, s*) in table *Sensor* as a composite foreign key.

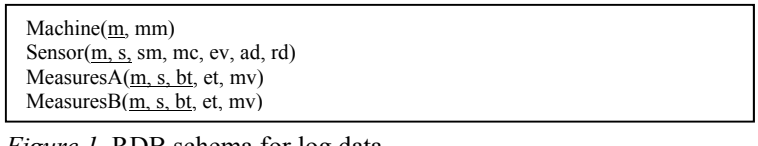


Figure 1. RDB schema for log data

The RDF view of the RDB is illustrated by the RDF graph in Figure 2.

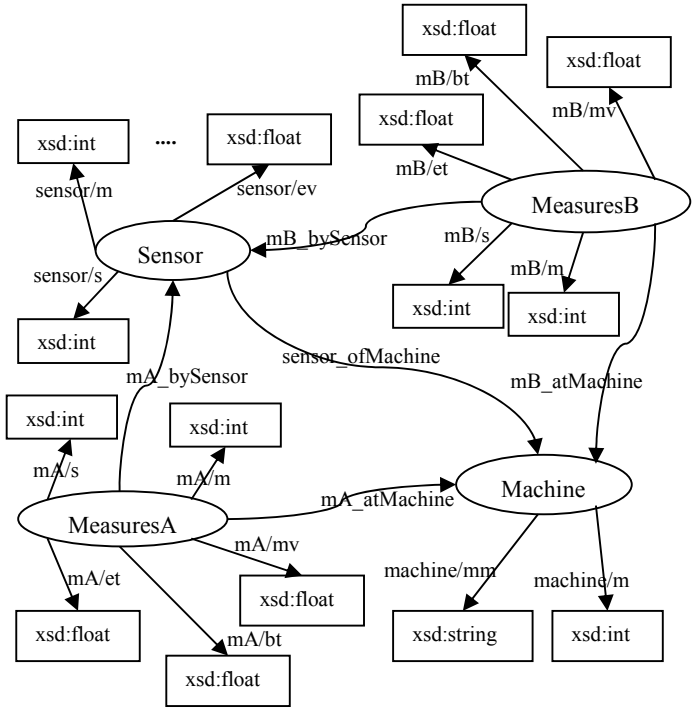


Figure 2. RDF graph of the RDF view for the example RDB

Next we define two more typical numerical SPARQL queries to the log database, *Q2* and *Q3,* that discover abnormal machine behaviors. Query *Q2* identifies a potential failure by retrieving for machine models *M_1, M_2,* and *M_3* those *machineid* where, during the time interval *(bt, et),* the measured value *mv* was above 75% of the allowed deviation *@thA* from the expected value *ev*.

```
Q2:
SELECT ?machineid ?bt ?et
FROM <prod>
WHERE{?measuresA log:mA_bySensor  ?sensor.
      ?measuresA log:mA/bt        ?bt.
      ?measuresA log:mA/et        ?et.
      ?measuresA log:mA/mv        ?mv.
      ?measuresA log:mA_atMachine ?machineid.
      ?machineid log:machine/mm   ?mm.
      FILTER (?mm in ('M_1','M_2','M_3')).
      ?sensor    log:sensor/ev    ?ev.
      FILTER (?mv > (?ev + 0.75*@thA))    }
```

Query *Q3* identifies abnormal behaviors of machines of a measurement class based on absolute deviations: when and for which machine identifiers did the pressure reading of class *B* deviate more than *@thB* from its expected value *ev*?

```
Q3:
SELECT ?m ?bt ?et
FROM <prod>
WHERE {?measuresB  log:mB/bt       ?bt.
       ?measuresB  log:mB/et       ?et.
       ?measuresB  log:mB/mv       ?mv.
       ?measuresB  log:mB_bySensor ?sensor.
       ?sensor     log:sensor/m    ?m.
       ?sensor     log:sensor/ev   ?ev.
       BIND ((?mv-?ev) as ?temp).
       FILTER (abs(?temp) > @thB)       }
```

3 FLOQ Overview and Query Processing

Figure 3 illustrates processing of numerical SPARQL queries by FLOQ.

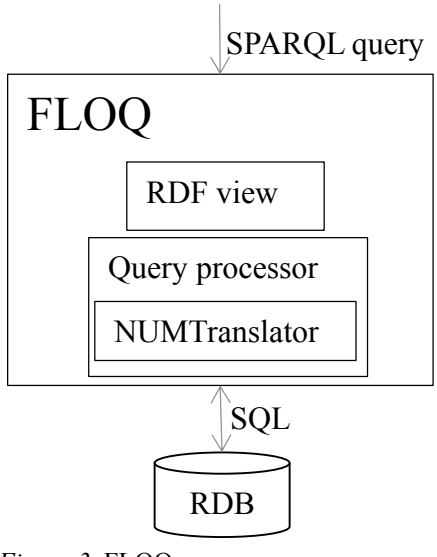


Figure 3. FLOQ query processor

The *RDF view* over the RDB is automatically generated based on the database schema and ontology mapping tables in FLOQ. The used mappings conform to the direct mapping recommended by W3C [10].

We define a unique RDFS class for each relational table, except for link tables [10] representing set-valued properties as many-to-many relationships. In addition, RDF properties are defined for each column in a table. For example, the RDFS class with the URI *<log:mA>* represents the table *MeasuresA*, while *<log:mA/bt>* and *<log:mA/et>* represent the columns *bt* and *et* in *MeasuresA,* respectively.

The RDF view is defined in terms of:

- *Source predicates* $R(a_1, a_2, ..., a_n)$ that represent the content of each referenced relational database table *R* where the tuple $(a_1, ..., a_n)$ represents a row in *R*.
- *URI-constructor* predicates that construct URIs to identify rows in tables.
- *Mapping tables* that map relational schema elements to RDF concepts.

The complete RDF view definitions can be found in [9]. The query processing steps in FLOQ are shown in Figure 4.

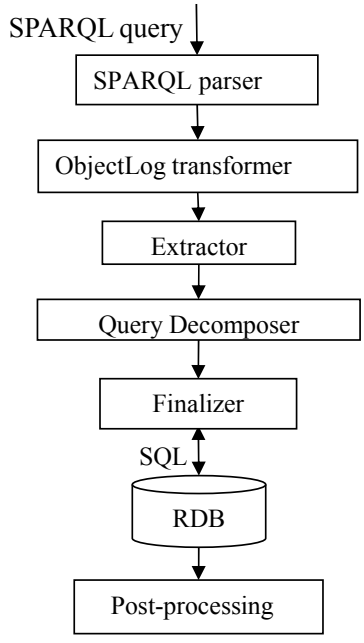


Figure 4. Query processing steps

The *SPARQL parser* first transforms the SPARQL query into an ObjectLog expression where each triple pattern in the query becomes a reference to the RDF view of the RDB. Then the *ObjectLog transformer* generates a simplified disjunctive normal form (DNF) predicate. The NUMTranslator algorithm performs the *extractor* and *finalizer* steps. The extractor collects from conjunctions predicates that can be translated to SQL, called *access filters*. The *query decomposer* then optimizes the query, producing a query execution plan where access filters are called. The finalizer traverses the execution plan to translate the extracted predicates in the access filters into SQL expressions. When the execution plan is interpreted, the generated SQL statements are sent to the RDB for execution. The non-extracted predicates are not translated to SQL and have to be processed outside the RDB by *post-processing* operators. For example, such operators are URI-constructors and numerical expressions not supported by the SQL engine.

4 The NUMTranslator Algorithm

The NUMTranslator uses a table-driven approach to define which SPARQL operators to extract and translate into corresponding SQL operators and functions. Table 1 defines the SPARQL to SQL operator translations:

Table 1. **SPARQL to SQL operators to translate**

SPARQL	SQL	INFIX	FUNCTION
>	>	True	False
<	<	True	False
=	=	True	False
!=	<>	True	False
+	+	True	True
-	-	True	True
ABS	ABS	False	True
UCASE	UPPER	False	True
etc.			

In Table 1 there is one row for each SPARQL operator or function (column *SPARQL)* that can be translated into SQL. The column *SQL* defines the corresponding SQL operator or function. A value in the column *INFIX* is true when the corresponding SQL operator is an infix operator *op* on operands *x* and *y*, i.e. *x op y* (e.g. x+y); otherwise it is an SQL function on format *f(x,y,..)*. The column *FUNCTION* is true when the operator is a non-Boolean function returning a value.

4.1 The NUMTranslator extractor

The extractor is applied on each ObjectLog conjunction in the simplified predicate received by the ObjectLog transformer. The extractor collects predicates that can be translated to SQL. Such predicates are i) source predicates *SPs* representing RDB tables, and ii) *non-source predicates (NSPs)* that are defined in Table 1 as translatable to SQL.

Figure 5 shows the ObjectLog representation of *Q1* after it has been transformed by the ObjectLog transformer.

```
Q1(m, bt, et):-
1  MeasuresA(m, s, bt, et, mv)          and
2  mv > v36                             and
3  v36 = ev + @thA                      and
4  Sensor(m, s, _, _, ev, _, _)
```

Figure 5. ObjectLog of query *Q1*

In this case all predicates in *Q1* are translatable to SQL since *MeasuresA* and *Sensor* are SPs, and > and + are NSPs defined in Table 1.

The steps of the extractor are the following:

1. Initialize a variable *Xpreds* for the first found SP, denoted *R1*, in the conjunction and bind a variable *Rest* to the other predicates.
2. Iteratively extract from *Rest* the predicates that have some common variable with some extracted predicate in *Xpreds*, which are either SPs or NSPs defined in Table 1.
3. Construct an *access filter* of all extracted predicates in *Xpreds* since those can be fully translated to SQL.
4. While there are some remaining SP, *R2*, in *Rest*, re-initialize *Xpreds* by *R2* and *Rest* by the remaining predicates, and repeat steps 2-3.
5. Finally, construct a conjunction of the access filters and *Rest*.

For example, for *Q1* the predicates in *Xpreds* are extracted in the following order:

1. *MeasuresA(m, s, bt, et, mv)* (line 1), since it is an SP.
2. *>(mv, v36)* (line 2) since > is defined in Table 1 and the variable *mv* is common with the extracted *MeasuresA*.
3. *Sensor(m, s, _, _, ev, _, _)* (line 4) since it is an SP having common variables (*m* and *s*) with *MeasuresA()*.
4. *V36 = ev + @thA* (line 3) since + is defined in Table 1 and the variable *ev* is common with the extracted *Sensor* predicate.

Then the following conjunctive access filter *F1* is formed by the predicates in *Xpreds*:

```
  F1(m,s,bt,et,mv,ev):-
1   MeasuresA(m, s, bt, et, mv)       and
2   Sensor(m, s, _, _, ev, _, _)      and
3   v36= ev + @thA                    and
4   mv > v36
```

No non-translatable predicates remain in *Rest*.

4.2 Query decomposition

To optimize the query produced by the extractor, the query decomposer uses cost-based optimization [6] to produce an optimized execution plan. Based on heuristics and statistic of the queried RDB, execution cost and selectivities of access filter are estimated. Default cost parameters are used by the optimizer to estimate the execution cost and selectivities of predicates if no statistic is available. The decomposer will then reorder the access filters and the post processed predicates to generate an optimized execution plan. We do not further elaborate the query decomposer here.

4.3 The NUMTranslator finalizer

The finalizer translates access filters in the decomposed execution plan into calls to an SQL interface operator, *sql* that sends generated SQL strings to the back-end RDB for execution.

ObjectLog numerical expressions are translated into SQL numerical expressions by recursively replacing all ObjectLog domain variables that represent numerical expressions with their bound expressions. For example, the variable *v36* in line 4 in *F1* doesn't represent a relational column and is replaced by its bound expression in line 3, and then the obtained expressions is *mv* > *ev* + *@thA*. Thus for *Q1* the execution plan *P1* becomes the following:

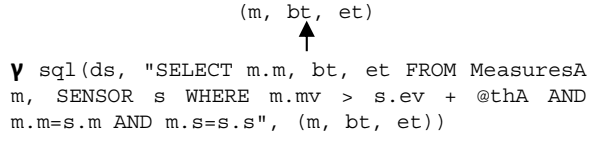


Figure 6. Execution plan *P1* with NUMTranslator

The execution plan contains an algebra expression where the *apply* operator γ *fn(..)* calls the *foreign predicate sql(ds, q, result)* implemented in Java. The foreign predicate *sql* sends an SQL query *q* to the RDBMS data source *ds* for execution and iteratively returns bindings of tuples, *result*.

If NUMTranslator had not been applied, all numerical operators would have to be post-processed, which would slow down the query execution since filtering cannot be made in the database server.

For example, if NUMTranslator is turned off, for *Q1* the following execution plan *P2* is produced that doesn't contain any numerical SQL operators corresponding to numerical SPARQL operators, which are instead post-processed:

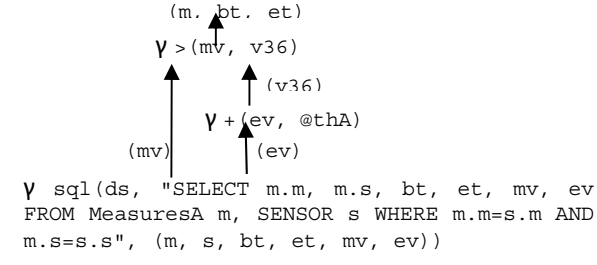


Figure 7. Execution plan *P2* without NUMTranslator

Comparing the two execution plans *P1* and *P2* it can be seen that the *sql* operator in *P2* retrieves much more data than *P1*, so if NUMTranslator is turned off lots of data needs to be filtered out outside the RDB server. Furthermore, the utilization of indexes on the SQL numerical expression by the

back-end database server makes significant performance difference. We show in the next section that applying NUMTranslator substantially improves the query performance of numerical SPARQL queries.

5 Performance Measurements

We compared the performance for executing the numerical queries *Q1, Q2,* and *Q3* in FLOQ with and without applying NUMTranslator. Furthermore, we compared the query performance of FLOQ with the query performance of D2RQ [1] for *Q1, Q2,* and *Q3,* for the same back-end relational database. We tried to run the queries with both ontop [7] and Virtuoso [3] as well, but none of our numerical SPARQL queries could be run, indicating that those systems do not provide full support for processing numerical SPARQL queries.

All experiments are carried out on a MS SQL Server 2008 R2 installed on a server machine with 8 AMD Opteron™ 6128 processors, 2.00 GHz CPU and 16GB RAM. The RDB is populated by loading sensor data into the MS SQL server. B-tree indexes are created on the columns *mm, mv, bt, et, ev, ad,* and *rd* to speed up the queries.

All measurements were taken both for cold and warm runs. The cold runs were made immediately after the RDBMS server was started, which implied that there were no data cached in the buffer pool and the executed query wasn't optimized by the RDBMS. Thus a measured query execution time for a cold run includes the time for i) reading data from disk, ii) SQL query optimization on the RDBMS server, iii) communication, and iv) post-processing of data on the client. The warm runs were made after a query was executed once. Since the back-end RDBMS has a statement cache a same SQL query executed twice will be optimized the first time it is run. Therefore, warm executions do not include RDBMS query optimization time.

The plotted values are mean values of three measurements. The standard deviation is less than 10% in all cases. To investigate the SQL query produced by all the other systems we use the system profiling tool of MS SQL server when running a query.

The following notations are used in the performance diagrams:

- *NUMTranslator:* FLOQ with NUMTranslator turned on, i.e. the SPARQL numerical expressions are translated into corresponding SQL expressions.
- *Naive:* FLOQ with NUMTranslator turned off, i.e. the SPARQL numerical expressions are not translated into corresponding SQL numerical expressions.
- *D2RQ:* D2RQ version [0.8.1] configured with the system's default mappings.

Figure 8, 9 and 10 show the execution times for both cold and warm runs for *Q1, Q3, and Q2* while scaling the databases size from 1 GB to 15 GB.

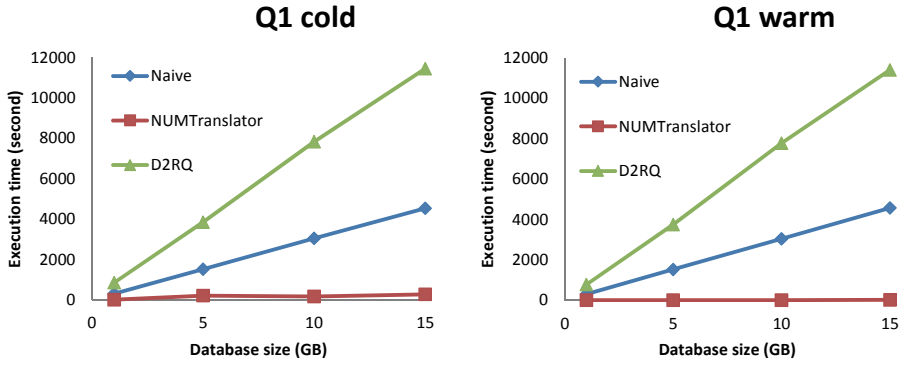


Figure 8. Execution times for *Q1*

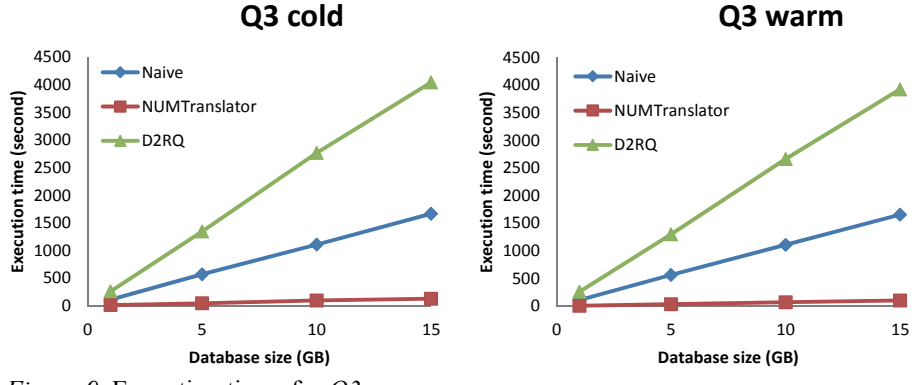


Figure 9. Execution times for *Q3*

Figure 8 and 9 show that *NUMTranslator* substantially improves the query execution scalability compared to *Naïve* for numerical SPARQL queries like *Q1* and *Q3* with highly selective numerical FILTERs: 0.04% for Q1 and 3% for Q3. In these cases pushing the numerical FILTERs to SQL is more profitable than filtering large data amounts on the client. The performance of D2RQ is worse than Naïve since D2RQ sends to the RDBMS an SQL query that doesn't contain numerical expressions, and is a much more complex query with more joins. Furthermore, *Q3* had to be manually changed for D2RQ to remove the BIND operator, since otherwise D2RQ wouldn't return correct result.

Measurement results for *Q2* are shown in Figure 10. For *Q2* the results for *NUMTranslator* and *Naïve* are presented in a separate diagram, since they are very close. It can be seen on Figure 10 that *NUMTranslator* doesn't improve the query performance for non-selective queries like *Q2* where the FILTER selects 43% of the data. In this case pushing the numerical

SPARQL filters to be executed to the RDBMS server doesn't make a significant difference compared to post-filtering data on the client.

D2RQ performs worse for *Q2* since it doesn't translate any of the FILTERs and it furthermore generates a very complex SQL query with many joins.

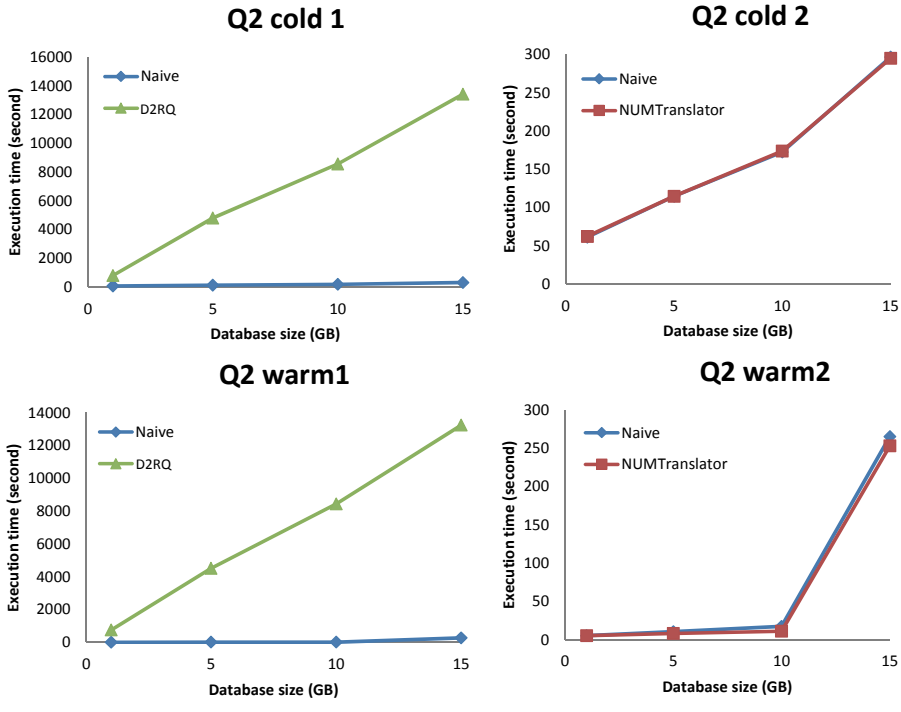


Figure 10. Execution times for *Q2*

In general, the experiments show that NUMTranslator substantially improves the query performance of numerical SPARQL queries where the numerical FILTERs have high selectivity.

6 Related Work

Virtuoso RDF Views [3] and D2RQ [1] are other systems that process SPARQL queries to RDF views of RDBs. These systems implement compilers that translate SPARQL directly to SQL. By contrast, FLOQ first generates ObjectLog queries to a declarative RDF view of the RDB, and then transforms the SPARQL queries to SQL by logical transformations.

We didn't find any publication of how D2RQ compiles numerical SPARQL queries into SQL and the documentation for Virtuoso's SQL generation is very limited [3]. However, by using the profiling tool of the RDBMS and the debug logging of Virtuoso we were able to analyze what

queries were actually sent to the RDBMS, showing that neither of those systems translates numerical SPARQL expressions into corresponding SQL expressions.

The ontop system [7] also enables SPARQL queries to RDF views of RDBs by translating SPARQL to Datalog programs, which are rewritten and translated to SQL. A difference to ontop is the table driven NUMTranslator algorithm, which makes it very easy to extend for new operators. Furthermore, FLOQ generates execution plans containing calls to SQL intermixed with expressions interpreted in the client. This enables FLOQ to interpret in the client SPARQL operators not available in SQL. In addition NUMTranslator translates the domain calculus SPARQL queries into tuple calculus SQL queries by substituting variables with their bound expressions.

7 Conclusions and Future Work

We presented the FLOQ system where the NUMTranslator algorithm uses a table driven approach to translate numerical domain calculus SPARQL expressions into corresponding numerical SQL expressions. This enables scalable processing of numerical SPARQL queries to RDF views over RDBs.

The approach was evaluated on a benchmark scenario in an industrial setting where logged data stored in an RDB was analyzed using numerical SPARQL queries. We compared the performance of the SPARQL queries with and without applying NUMTranslator. The experiments show that NUMTranslator substantially improves the query performance of numerical SPARQL queries in particular when the numerical expressions inside FILTERs are highly selective.

We also compared our approach with other systems that translate SPARQL queries to SQL. Only D2RQ could execute our queries, but substantially slower since D2RQ does not employ an approach similar to NUMTranslator.

As our next step, we will investigate numerical SPARQL queries searching large numbers of distributed log databases combined through an ontology. Another issue is creating benchmarks based on randomly generating SPARQL queries [5]. Furthermore, query processing and mediation strategies over other back-ends than RDBs [4] in our setting should be investigated.

8 ACKNOWLEDGMENT

This work is supported by EU FP7 project Smart Vortex and the Swedish Foundation for Strategic Research under contract RIT08-0041.

9 REFERENCES

1. Bizer, C., Cyganiak, R., Garbers, G., Maresch, O., and Becker, C. 2009. *The D2RQ Platform v0.7 - Treating Non-RDF Relational Databases as Virtual RDF Graph*, http://www4.wiwiss.fu-berlin.de/bizer/d2rq/spec/
2. Björkelund, A., Edström, L., etc. 2011. On the integration of skilled robot motions for productivity in manufacturing, In *Proc. of IEEE International Symposium on Assembly and Manufacturing*, Tampere, Finland.
3. Erling, O. and Mikhailov, I. 2009. RDF Support in the Virtuoso DBMS, *Studies in Computational Intelligence*, Vol. 221
4. Langegger, A., Wöß, W., and Blöchl, M. 2008. A Semantic Web Middleware for Virtual Data Integration on the Web, *5th European Semantic Web Conference ESWC 2008*.
5. Langegger, A. and Wöß, W. 2009. RDFStats – The Extensible RDF Statistics Generator and Library, *8th International Workshop on Web Semantics, DEXA 2009*, Linz, Austria, August 31-September 40.
6. Litwin, W. and Risch, T. 1992. Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6.
7. Rodriguez-Muro, M., Rezk, M., Hardi, J., Slusnys, M., Bagosi, T., and Calvanese, D. 2013. Evaluating SPARQL-to-SQL Translation in Ontop, *ORE 2013*
8. Sequeda, J. F., and Miranker, D. P. 2013. *Ultrawrap: SPARQL Execution on Relational Data*, Tech. Report, Univ. of Texas at Austin. http://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2078.pdf
9. Stefanova, S., and Risch, T. 2011. Optimizing Unbound-property Queries to RDF Views of Relational Databases. *7t International workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2011)*, Bonn, Germany.
10. Arenas, M., Bertails, A., Prud'hommeaux, E., and Sequeda, J. 2012. A Direct Mapping of Relational Data to RDF, http://www.w3.org/TR/rdb-direct-mapping/
11. Harris, S., and Seaborne, A. 2013. SPARQL 1.1 Query Language, http://www.w3.org/TR/sparql11-query/

Paper III

The paper is reformatted for typographic consistency.

Scalable Queries over Log Database Collections

Minpeng Zhu, Khalid Mahmood, and Tore Risch
Department of Information Technology, Uppsala University, Uppsala 75237, Sweden
{minpeng.zhu, khalid.mahmood, tore.risch}@it.uu.se

ABSTRACT
Various business application scenarios need to analyse the working status of products, e.g. to discover abnormal machine behaviours from logged sensor readings. The geographic locations of machines are often widely distributed and have measurements of logged sensor readings stored locally in autonomous relational databases, here called *log databases*, where they can be analysed through queries. A global meta-database is required to describe machines, sensors, measurements, etc. Queries to the log databases can be expressed in terms of these meta-data. FLOQ (Fused LOg database Query processor) enables queries searching collections of distributed log databases combined through a common meta-database. To speed up queries combining meta-data with distributed logged sensor readings, sub-queries to the log databases should be run in parallel. We propose two new strategies using standard database APIs to join meta-data with data retrieved from distributed autonomous log databases. The performance of the strategies is empirically compared with a state-of-the-art previous strategy to join autonomous databases. A cost model is used to predict the efficiency of each strategy and guide the experiments. We show that the proposed strategies substantially improve the query performance when the size of selected meta-data or the number of log databases are increased.

1 INTRODUCTION

Various business applications need to observe the working status of products in order to analyse their proper behaviours. Our application is from a real-world scenario [11], where machines such as trucks, pumps, kilns, etc. are widely distributed at different geographic locations and where sensors on machines produce large volumes of data. The data describes time stamped sensor readings of machine components (e.g. oil temperature and pressure) and can be used to analyse abnormal behaviours of the equipment. In order to analyse passed behaviour of monitored equipment, the sensor readings can be stored in relational databases and analysed with SQL. In our application

area, data is produced and maintained locally at many different sites in autonomous relational DBMSs called *log databases*. New sites and log databases are dynamically added and removed from the federation. The number of sites is potentially large, so it is important that the query processing scales with increasing number of sites. A global meta-database enables a global view of the working status of all machines on different sites. It stores meta-data about machines, sensors, sites, etc.

A particular challenge in our scenario is a scalable way to process queries that join meta-data with data selected from the collection of autonomous log databases using standard DBMS APIs. This paper proposes two strategies to perform such joins, namely *parallel bind-join* (PBJ) and *parallel bulk-load join* (PBLJ). PBJ generalizes the *bind-join* (BJ) [4] operator, which is a state-of-the-art algorithm for joining data from an autonomous external database with a central database. One problem with bind-join in our scenario is that large numbers of SQL queries will be sent to the log databases for execution, one for each parameter combination selected from the meta-database, which is slow. Furthermore, whereas bind-join is well suited for joining data from a single log database with the meta-database, our application scenario requires joining data from many sites.

With both PBJ and PBLJ, streams of selected meta-data variable bindings are distributed to the wrapped log databases and processed there in parallel. After the parallel processing the result streams are merged asynchronously by FLOQ.

- With PBJ the streams of bindings selected from the meta-database are bind-joined in the distributed wrappers with their encapsulated log databases. The bind-joins of different wrapped log databases are executed in parallel.
- With PBLJ the selected bindings are first bulk loaded in parallel into a *binding table* in each log database where a regular join is performed between the loaded bindings and the local measurements.

The strategies are implemented in our prototype system called *FLOQ (Fused LOg database Query processor)*. FLOQ provides general query processing over collections of autonomous relational log databases residing on different sites. The collection of log databases is integrated by FLOQ through a meta-database where properties about data in the log databases are stored. On each site the log database is encapsulated by a *FLOQ wrapper* to pre- and post-process queries.

To investigate our strategies, a cost model is proposed to evaluate the efficiency of each strategy. To evaluate the performance we define fundamental queries for detecting abnormal sensor readings and investigate the impact of our join strategies. A relational DBMS from a major commercial vendor is used for storing the log databases.

In summary the contributions are:

- Two join strategies are proposed and compared: parallel bind-join and parallel bulk-load join, for parallel execution of queries joining meta-data with data from collections of autonomous databases using external DBMS APIs.
- A cost model is proposed to evaluate the strategies.
- The conclusions from the cost model are verified experimentally.

The rest of this paper is organized as follows: Section 2 overviews the FLOQ system architecture and presents the scenario and queries used for the performance evaluation. Section 3 presents the join strategies and the cost model used in the evaluation. Section 4 presents the performance evaluation for the join strategies. Section 5 describes related work. Finally, Section 6 concludes and outlines some future work.

2 FLOQ

Fig. 1 illustrates the FLOQ architecture. To analyse machine behaviours, the user sends queries over the integrated log databases to FLOQ. FLOQ processes a query by first querying the meta-database to find the identifiers of the queried log databases containing the desired data, then in parallel sending distributed queries to the log databases, and finally collecting and merging the distributed query results to obtain the final result. Scalable parallel processing of queries making joins between a meta-database and many large log databases is the subject of this paper.

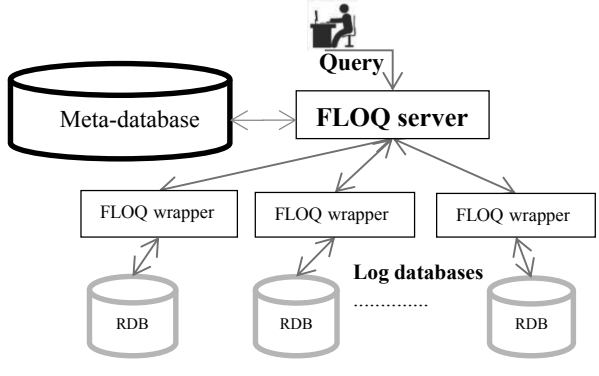


Figure 1. FLOQ system architecture

Each log database is encapsulated with a *FLOQ wrapper* called from the FLOQ server to process queries over the wrapped log database. A FLOQ wrapper contains a full query processor which enables, e.g. local bind-joins between a stream of bindings selected from the meta-database and the log database. Parallel processing is provided since the FLOQ wrappers work

independently of each other. Each FLOQ wrapper sends back to the FLOQ server the result of executing a query as a stream of tuples. The results from many wrappers are asynchronously merged by the FLOQ server while emitting the result to the user. Details of the query processor are described in [10], [13,14] and are outside the scope of this paper.

2.1 The FLOQ schema

The schema for the FLOQ meta-database is shown in Fig. 2(a). The table *MachineModel(m, mmn, descr, mmanuf)* stores data about machine models, i.e. a unique ma-chine model identifier *m*, along with its name *mmn*, description *descr*, and manufacturer *mmanuf*. The table *MachineInstallation(mi, m, sid)* stores meta-data about each machine installation, i.e. a unique machine installation identifier *mi*, its installed site *sid* and its machine model identifier *m* (foreign key). The table *SensorModel(sm, sname, smanuf)* stores information about sensor models, i.e. a unique sensor model identifier *sm*, the sensor model name *sname*, and its manufacturer *smanuf*. The table *SensorInstallation(si, mi, sm, ev)* stores the sensor installation information, i.e. a sensor installation identifier *si*, the machine installation *mi* of *si*, the sensor model *sm*, and the expected measured value *ev*. The columns *m* and *sid* in table *MachineInstallation* are foreign keys in tables *MachineModel* and *Site*, respectively. The column *mi* in table *SensorInstallation* is foreign key to *MachineInstallation*.

```
MachineModel(m, mmn, descr, mmanuf)
MachineInstallation(mi, m, sid)
SensorModel(sm, sname, smanuf)
SensorInstallation(si, mi, sm, ev)
Site(sid, name, logdb)
```

Figure 2(a). Meta-database schema

```
Measures(mi,si,bt,et,mv)
```

Figure 2(b). Log table at each site

```
VMeasures(logdb,mi,si,bt,et,mv)
```

Figure 2(c). Integrated view in FLOQ server

The table *Site(sid, name, logdb)* stores information about the sites where the log databases are located: a numeric site identifier *sid*, its *name*, and an identifier of its log database, *logdb*. A new log database is registered to FLOQ by inserting a new row in table *Site*. Each site presents to FLOQ its log data as a temporal local relation *Measures(mi, si, bt, et, mv)* (Fig. 2(b)) representing measurements from the sensors installed on the machines at the site, i.e. temporal local-as-view [5] data integration is used. For a machine installation *mi* at a particular site the local view presents the measured readings

from sensor installation *si* in the valid time interval *[bt,et)*. The columns *mi* and *si* in *Measures* are foreign keys from the corresponding columns in the meta-database tables *MachineInstallation* and *SensorInstallation*, respectively.

The view *VMeasures* (Fig. 2(c)) in FLOQ integrates the collection of log databases. It is logically a union-all of the local *Measures* views at the different sites. In *VMeasures* the attribute *logdb* identifies the origin of each tuple. Through the meta-database users can make queries over the log databases by joining other meta-data with *VMeasures*. Since the set of log databases is dynamic it is not feasible to define *VMeasures* as a static view; instead FLOQ processes queries to *VMeasures* by dynamically submitting SQL queries to the log databases and collecting the results. In the experiments we populate the meta-database and the log databases with data from a real-world application [11].

2.2 Example Queries

Q1 in Fig. 3 is a simple query that retrieves unexpected sensor readings. It returns machine identifiers *mi* together with the time intervals *[bt,et)* when a sensor on the machine has measured values *mv* higher than the expected values *ev* by a threshold parameter *th* on line 5 marked '?'.

```
Q1:
1  SELECT m.mi, m.bt, m.et
2  FROM Measures m, Site s,
3       MachineInstallation mi,
4       SensorInstallation si
5  WHERE m.mv > si.ev+? AND
6        mi.mi > ? AND
7        si.mi = mi.mi AND
8        m.si = si.si AND
9        m.logdb = s.logdb AND
10       s.sid < ?
```

Figure 3. Query Q1

```
Q2:
1  SELECT count(*)
2  FROM Measures m, Site s,
3       MachineInstallation mi,
4       SensorInstallation si
5  WHERE m.mv > si.ev+? AND
6        mi.mi > ? AND
7        si.mi = mi.mi AND
8        m.si = si.si AND
9        m.logdb = s.logdb AND
10       s.sid < ?
```

Figure 4. Query Q2

Query *Q1* is used for the basic scalability experiments. It contains a simple numerical expression over the log database view in terms of *th*. On line 6 there is a constraint on the selected machine identifiers *mi* and on line 10 the selected sites *sid* are restricted. The experiments are scaled by varying these parameters. The number of log databases is varied by restricting *sid*, the amount of data selected from each log database is varied by *th*, and the number of bindings selected from the meta-database is varied by *mi*.

Query *Q2* in Fig. 4 is similar to *Q1*, the difference being that it applies an aggregate function over *Q1*, i.e. it computes the number of faulty sensor readings. Here only a single value is returned from each log database. The purpose of the query is to investigate the join strategies without concerning the overhead of transferring substantial amounts of data back to the client.

```
Q3:
1 SELECT m.mi, m.bt, m.et
2 FROM Measures m, Site s,
3      MachineInstallation mi,
4      SensorInstallation si
5 WHERE abs(m.mv-si.ev)/si.ev>? AND
6       si.mi = mi.mi AND
7       m.si = si.si AND
8       m.logdb=s.logdb
```

Figure 5. Query Q3

```
Q4:
1 SELECT m.mi, m.bt, m.et
2 FROM Measures m, Site s,
3      MachineInstallation mi,
4      SensorInstallation si
5 WHERE si.mi = mi.mi AND
6       m.si = si.si AND
7       m.logdb = s.logdb AND
8       ((m.mv>(1+?)*si.ev and si.ev>0) or
9        (m.mv<(1+?)*si.ev and si.ev<0) or
10       (m.mv<(1-?)*si.ev and si.ev>0) or
11       (m.mv>(1-?)*si.ev and si.ev<0))
```

Figure 6. Transformed Q3

Query *Q3* in Fig. 5 is an example of a more complex numerical query for identifying machine failures. It detects situations where the relative deviation of sensor readings from *ev* is larger than a threshold parameter we denote *rth*. One property of *Q3* is that the query optimizer of the used DBMS cannot utilize an ordered index on the measured value *mv*, so the entire local table *Measures* on each site will be scanned entirely. This query thus has a high query execution cost for searching the log databases.

Query *Q4* in Fig. 6 is a manually transformed version of *Q3* to expose the index column *mv* of *Measures* table for query optimizer of the DBMS for scalable search. Here all parameter occurrences in the query (marked ?) refer to the supplied value of *rth*. FLOQ automatically makes this algebraic transformation by utilizing the algorithm in [12]. The difference between *Q3* and *Q4* shows the trade-off between full scan and index scan in the log databases enabled by the rewrite. *Q3* is an expensive query compared to *Q4*.

3 Join Strategies

The two strategies, PBJ and PBLJ, for parallel execution of queries joining data between the meta-database and the log databases are illustrated in Fig. 7 and Fig. 8, respectively. With both strategies FLOQ first extracts parameter bindings from the meta-database. The result is a stream of tuples is called the *binding stream B* where each tuple $(i, v_1, v_2, ..., v_p)$ is a *parameter binding*. The elements $v_1, v_2, ..., v_p$ of the binding stream are the values of the free variables in the query fragment sent to the log databases. For example, in Q1 the free variables are *(mi, si, ev)*. Each binding tuple is prefixed with a *destination site, i*, identifying where the log database RDB_i resides. The parameter binding tuples are joined with measurements in the log databases. Thus the binding stream is split into one site binding stream B_i per log database RDB_i, $B = B_1 \cup B_2 \ldots \cup B_n$, where n is the number of sites. The destination i determines to which site the rest of the tuple, $(v_1, v_2, \ldots, v_p)$, is routed. The join strategies are defined as follows:

PBJ, parallel bind-join: PBJ (Fig. 7) is a generalization of bind-join [4] to handle parallel execution between a common meta-database and a collection of wrapped relational databases RDB_i. On each site i the tuples in the binding stream B_i received by a FLOQ wrapper is bind joined (BJ) with the query σ_i sent to the database RDB_i through parameterized (prepared) JDBC calls. The tuples in the result stream R_i from the JDBC calls are then streamed back to the FLOQ server, where they are merged asynchronously with the result tuples from other sites. With PBJ, a parameterized query is executed many times in each wrapped log database, once for each parameter binding in B_i.

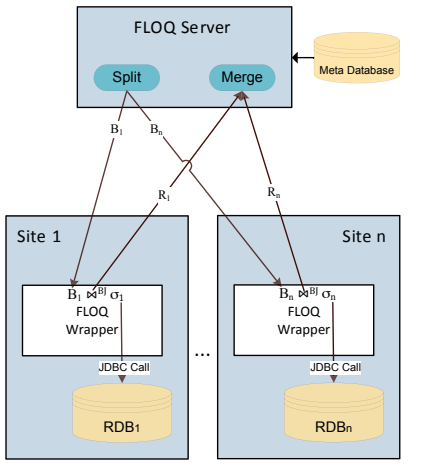


Figure 7. PBJ

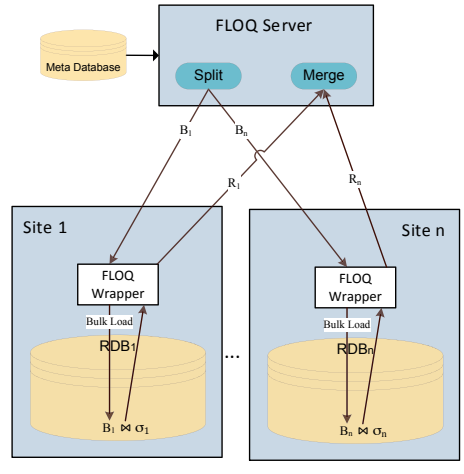


Figure 8. PBLJ

PBLJ, parallel bulk-load join: With PBLJ (Fig. 8) each FLOQ wrapper first bulk loads the entire binding stream B_i into a *binding table* in RDB_i. When all parameter bindings have been loaded, the system submits a single SQL query to the log database to join the loaded binding table with σ_i. As for PBJ, the result stream R_i is shipped back to the FLOQ server through the wrapper for asynchronous merging. Compared to PBJ, the advantage of this approach is that only one query is sent to each log database. It requires the extra step of bulk loading in parallel the entire parameter streams into each log database, which, however, should be less costly compared to calling many prepared SQL statements through JDBC with PBJ. The bulk loading facility of the DBMS is utilized for high performance.

BJ, regular bind-join: If there is a single log database, PBJ is analogous to BJ and is a baseline in our evaluations. With BJ one prepared SQL query per binding is shipped from the FLOQ wrapper to only one log database, RDB_1.

3.1 Cost Model for Join Strategies

The total cost in terms of response times of the proposed join strategies is divided between the cost of execution in the FLOQ server C_{FLOQ} and the maximum site cost C_i.

$$C_{Join} = C_{FLOQ} + max(\{C_i : i = 1, \dots, n\}) \quad (1)$$

The total cost of the FLOQ server execution is approximately divided between two major components, which are the cost of splitting the binding stream B, C_s, and the cost of merging all result streams R_i, C_m. The cost of the FLOQ server execution is independent of any join strategies, i.e.:

$$C_{FLOQ} = C_s + C_m \quad (2)$$

The variables used in analysing cost models are described in Table 1.

Table 1. **Variables used in the cost model**

Variable	**Description**
C_{Join}	Total cost of a join
B_i	Binding stream to site *i*
R_i	Result stream from site *i*
C_i	Total cost at site *i*
C_{σ_i}	Cost of executing σ_i in RDB_i
$C_{\bowtie_i}$	Cost of local join at site *i*
$C_{Bulkload_i}$	Cost of bulk loading in RDB_i
C_{σ_β}	Selection Cost for a single binding β
RDB_i	The relational log database at site *i*
C_{FLOQ}	Total execution cost in the FLOQ server
C_S	Cost of splitting the binding stream B in the FLOQ server
β	A single binding from the binding stream B_i
C_m	Cost of merging result streams R_i in the FLOQ server
C_{JDBC}	Cost of JDBC call for a single binding β
σ_i	The query to RDB_i.
C_{B_i}	Cost of transferring binding stream B_i to site *i*
C_{R_i}	Cost of transferring result stream R_i from site *i*
C_{Net}	Network communication overhead cost for a single binding β

The total site cost C_i is approximately divided between four major cost components: (i) transferring the binding stream B_i from the FLOQ server to the site, C_{B_i}, (ii) executing σ_i in the log database, C_{σ_i}, (iii) local join $C_{\bowtie_i}$ either in RDB_i ($C_{\bowtie_i}^{LogDB}$ for PBLJ) or in the FLOQ wrapper ($C_{\bowtie_i}^{Wrapper}$ for PBJ), and

(iv) transferring the result stream R_i to the FLOQ server, C_{R_i}. Thus the total site cost C_i is defined as:

$$C_i = C_{B_i} + C_{\sigma_i} + C_{\bowtie_i} + C_{R_i} \quad (3)$$

By combining equation (1), (2), and (3), the total cost of a distributed join becomes:

$$C_{Join} = C_s + C_m + max(\{ (C_{B_i} + C_{\sigma_i} + C_{\bowtie_i} + C_{R_i}) : i = 1, \dots, n\}) \quad (4)$$

For each site, the binding stream B_i is significantly smaller than the number of logged measurements in RDB_i:

$$|B_i| \ll |Measures(RDB_i)| \quad (5)$$

For PBJ, the bind-join is performed in each FLOQ wrapper, therefore, the cost of a local join $C_{\bowtie_i}$ can be replaced with the cost of a bind-join in the wrapper, $C_{\bowtie_i}^{Wrapper}$. Also the cost of executing the sub-query σ_i that selects data from a log database, C_{σ_i}, is replaced with the *BJ selection cost*, $C_{\sigma_i}^{Wrapper}$, in the site cost in (3).

$$C_i^{PBJ} = C_{B_i}^{PBJ} + C_{\sigma_i}^{Wrapper} + C_{\bowtie_i}^{Wrapper} + C_{R_i} \quad (6)$$

In PBLJ the joins and selections are combined into one sub-query to each RDB_i. Therefore, the cost of $C_{\bowtie_i}$ and C_{σ_i} in the site cost in equation (3) for PBLJ can be replaced with the cost of join and selection in the log database ($C_{\bowtie_i}^{LogDB}$ and $C_{\sigma_i}^{LogDB}$):

$$C_i^{PBLJ} = C_{B_i}^{PBLJ} + C_{\sigma_i}^{LogDB} + C_{\bowtie_i}^{LogDB} + C_{R_i} \quad (7)$$

In PBJ, the FLOQ server transfers the binding stream B_i to a FLOQ wrapper through the standard network protocol. Therefore, the cost of transferring bindings to each site, $C_{B_i}^{PBJ}$, is the aggregated network communication overhead for each binding, C_{Net}.

$$C_{B_i}^{PBJ} = |B_i| \times C_{Net}, \text{ where} |B_i| \geq 1 \quad (8)$$

In PBLJ all the bindings B_i are bulk-loaded directly into the log database. The cost of sending all bindings to site i, $C_{B_i}^{PBLJ}$, is the cost of bulk loading the bindings, $C_{Bulkload_i}$.

$$C_{B_i}^{PBLJ} = C_{Bulkload_i} \quad (9)$$

Obviously, the cost of bulk-loading in PBLJ $C_{Bulkload_i}$ is insignificant compared to sending large numbers of bindings to prepared SQL statements in PBJ:

$$C_{Bulkload_i} << |B_i| \times C_{Net}, \text{ where } |B_i| \geq 1; \text{ therefore,}$$
$$C_{B_i}^{PBLJ} \leq C_{B_i}^{PBJ} \quad (10)$$

On the other hand, the selection cost of PBLJ is also low compared to PBJ since the cost of selection performed by RDB_i is lower than the combined cost of selection and JDBC overhead for each binding β of a binding stream B_i:

$$C_{\sigma_i}^{LogDB} \leq |B_i| \times (C_{\sigma_\beta} + C_{JDBC}), \text{ where } \beta \in B_i \text{ and } |B_i| \geq 1; \text{ therefore:} \quad (11)$$
$$C_{\sigma_i}^{LogDB} \leq C_{\sigma_i}^{Wrapper} \quad (12)$$

Similarly, a local join in the relational DBMS is efficient compared to the join performed in a FLOQ wrapper since query optimization techniques can be applied inside a relational DBMS where the overhead JDBC calls are eliminated. Thus,

$$C_{\bowtie_i}^{LogDB} \leq C_{\bowtie_i}^{Wrapper} \quad (13)$$

From equation (10), (12), and (13), the total cost at site *i* for the three components, transferring bindings (C_{B_i}), selection (C_{σ_i}), and join ($C_{\bowtie_i}$) are lower for PBLJ than for PBJ. The cost C_{R_i} of transferring the result streams R_i to the FLOQ server is equal for both PBLJ and PBJ, therefore, comparing (6) and (7):

$$C_i^{PBLJ} \leq C_i^{PBJ} \quad (14)$$

From equation (1), as the cost of the execution at the FLOQ server C_{FLOQ} is equal for both PBJ and PBLJ, by combing equation (1) and (14) it can be stated that the overall cost of join in PBLJ is lower than PBJ:

$$C_{PBLJ} \leq C_{PBJ} \quad (15)$$

3.2 Discussion

According to equation (15), PBLJ should always outperform PBJ in every experiment when $|B_i| \geq 1$. Equation (8) and (11) suggest that PBLJ will perform increasingly better than PBJ when scaling the number of bindings $|B_i|$. It is evident from equation (4) that, independent the chosen join strategy, when the size of the result stream $|R_i|$ is large, the tuple transfer cost (C_{R_i}) will be a major dominating component in the cost model. Therefore, the performance trade-offs between respective join strategies, are more significant when the number of tuples returned from the log database is small.

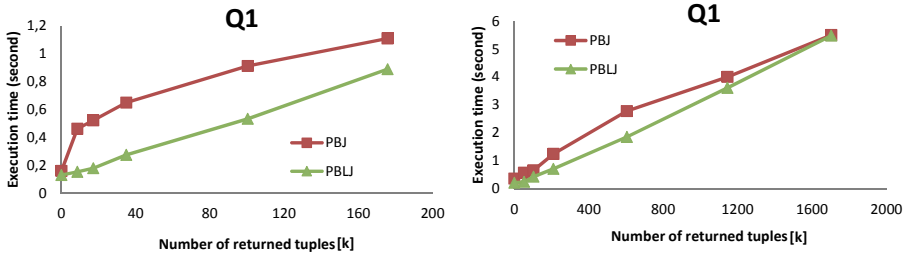
To conclude, according to the cost model, the performance evaluation should be investigated by (i) varying the number of tuples returned from the sites, (ii) scaling the number of sites, and (iii) scaling the number of bindings from the meta-database.

4 Performance Evaluation

We compared the performance of the join strategies PBJ and PBLJ based on the queries Q1, Q2, Q3, and Q4. In our real-world application each log database had more than 250 million measurements from sensor readings, occupying 10GB of raw data. The following scalability experiments were performed on six PCs (with 4 processors and 8GB main memory) running Windows 7 while: (i) scaling the number of result tuples $|R_i|$; (ii) scaling the number of sites, *n*; and (iii) scaling the number of bindings $|B_i|$.

Scaling the number of result tuples

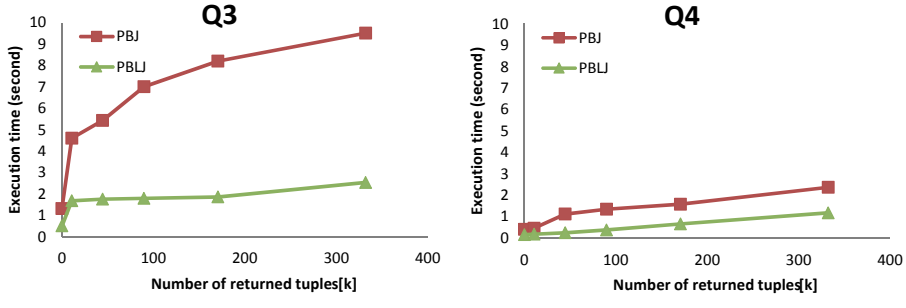
Fig. 9(a) shows the execution times of Q1 for the two join strategies over a single log database, while scaling the number of result tuples $|R|$ by adjusting *th*. As expected from equation (12), PBLJ performs better than PBJ. Since there is only one site, PBJ is equivalent to BJ.



(a) (b)

Figure 9. Q1(a) with one log database and (b) with six log databases

Fig. 9(b) compares the performance of Q1 for six log databases while scaling $|R|$. As expected PBLJ scales better than PBJ. However, as more tuples are returned from the log databases the network overhead is becoming a major dominating factor, making the performance difference of the join strategies insignificant. Notice that the number of returned tuples remains the same for both strategies; thus the network overhead is equal. However, PBLJ will always perform better (even with a small fraction) than PBJ since other overhead is larger for PBJ.



(a) (b)

Figure 10. Execution time for Q3 and Q4 with six log databases

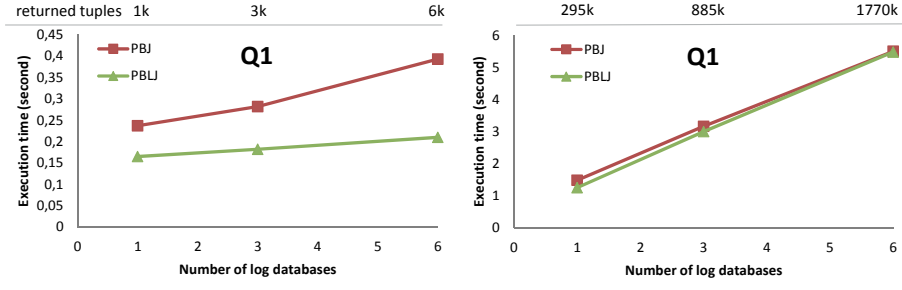
Fig. 10 compares PBJ and PBLJ for Q3 and Q4 for six log databases. Q3 is an example of a slow numerical query requiring a full scan of *Measures*, whereas Q4 is faster since it exposes the index on *Measures.mv* for query Q3. It is evident from Fig. 10 that PBLJ performs better than PBJ for both query Q3 and Q4. Fig. 10(b) shows the performance improvement due to index utilization compared to sequential scan in Q3.

To conclude, PBLJ performs better than PBJ when the number of returned tuples is increased, as also indicated by equation (15) of the cost model.

Scaling the number of log databases

Fig. 11 compares PBJ and PBLJ for Q1 when scaling the number of log databases. In Fig. 11(a) and Fig. 11(b) the total number of tuples returned from a single log database $|R_i|$ is 1K and 295K, respectively. Notice that the total number of tuples returned $|R|$ in each figure is multiplied with the fixed $|R_i|$ from each log database.

In Fig. 11(a) $|R|$ is small, so the performance difference between PBJ and PBLJ is dominating over the network cost, while in Fig. 11(b) the higher network cost makes the difference less significant.



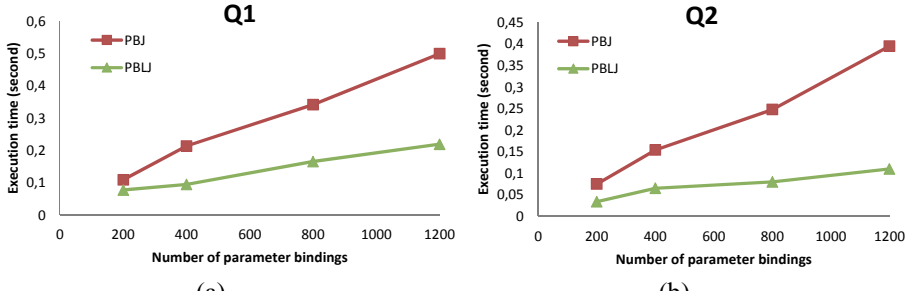
(a) 1k tuples from each database (b) 295k tuples from each database

Figure 11. Execution time for Q1 varying number of log databases and selectivity

In summary, the overall performance of PBLJ is always better while scaling number of log databases compared to PBJ.

Scaling the number of bindings

This experiment investigates the performance of PBJ and PBLJ while varying the number of bindings $|B_i|$ from the meta-database. Fig. 12 shows the execution times for Q1 and Q2 for PBJ and PBLJ for a single log database.



(a) (b)

Figure 12. Execution time for Q1 and Q2

From Fig. 12(a) it is evident that PBLJ performs significantly better while scaling $|B_i|$. The reason is that in PBJ, the FLOQ wrapper is performing $|B_i|$ bind-joins, so the overhead of the JDBC calls is multiplied with $|B_i|$. In all experiments the extra time for the bulk loading was less than 50ms irrespective of number of bindings $|B_i|$. This makes it insignificant for this small number of bindings relative to the size of the log databases. This confirms equation (8) and (11) of the cost model that PBJ will not scale compared to PBLJ when increasing the number of bindings. The experimental results of query Q2 that returns a single tuple per site are shown in Fig. 12(b). The reason of the better scalability of PBLJ than for Q1 is because the network communication overhead C_{R_i} in equation (4) is negligible since only one tuple is returned from each site.

In all experiments, the PBLJ join strategy performs better than PBJ, in particular while scaling the number of bindings $|B_i|$. This confirms equation (15) in the cost model. The performance improvement is more significant when the number of tuples returned from each log database is low.

5 Related work

Bind-join was presented in [4] as a method to join data from external databases [7]. We generalized bind-join to process in parallel parameterized queries to dynamic collections of autonomous log databases. Furthermore we showed that our bulk-load join method scales better in our setting.

In Google Fusion Tables [3] left outer joins are used to combine relational views of web pages, while [6] uses adaptive methods to join data from external data sources. In [9] the selection of autonomous data sources to join is based on market mechanisms. Our case is different because we investigate strategies to join meta-data with data from dynamic collections of log databases without joining the data sources themselves.

Vertical partitioning and indexing of fact tables in monolithic data warehouses is investigated in [1]. One can regard our *VMeasures* view as a horizontally partitioned fact table. A major difference to data warehouse techniques is that we are integrating data from dynamic collections of autonomous log databases, rather than scalable processing of queries to data uploaded to a central data warehouse.

In [2] the problem of making views of many autonomous data warehouses is investigated. The databases are joined using very large SQL queries joining many external databases. Rather than integrating external databases by huge SQL queries, our strategies are based on simple queries over a view (*VMeasures*) of dynamic collections of external databases, i.e. the local-as-view approach [5].

A classical optimization strategy used in distributed databases [8] is to cost different shipping alternatives of data between non-autonomous data servers before joining them. By contrast, we investigate using standard DBMS APIs (JDBC and bulk load) to make multi-database joins of meta-data with dynamic sets of autonomous log databases using local-as-view.

6 Conclusions

Two join strategies were proposed for parallel execution of queries joining meta-data with data from autonomous log databases using standard DBMS APIs: parallel bind-join (PBJ) and parallel bulk-load join (PBLJ). For the performance evaluation we defined typical fundamental queries and investigated the impact of our join strategies. A cost model was used to guide and evaluate the efficiency of the strategies. The experimental results validated the cost model. In general, PBLJ performs better than PBJ when the number of bindings from the meta-database is increased.

In the experiments a rather small set of autonomous log databases were used. Further investigations should evaluate the impact of having very large number of log databases and different strategies to improve communication overheads, e.g. by compression.

Acknowledgements

This work is supported by EU FP7 project Smart Vortex and the Swedish Foundation for Strategic Research under contract RIT08-0041.

References

1. Datta, A., VanderMeer, D.E., Ramamritham, K.: Parallel Star Join + DataIndexes: Efficient Query Processing in Data Warehouses and OLAP. J. IEEE TKDE. 14(6), 1299-1316 (2002)
2. Dieu, N., Dragusanu, A., Fabret, F., Llirbat, F., Simon, E: 1,000 Tables Inside the From. J. ACM VLDB. 2(2), 1450-1461 (2009)
3. Garcia-Molina, H., Halevy, A.Y., Jensen, C.S., Langen, A., Madhavan, J., Shapley, R.,Shen, W.: Google fusion tables: data management, integration and collaboration in the cloud. In: SoCC, pp. 175-180 (2010)
4. Haas, L., Kossmann, D., Wimmers, E., Yang, J: Optimizing queries across diverse data source. In: VLDB, pp. 276-285 (1997)
5. Halevy, A., Rajaraman, A., Ordille, J.: Data Integration: The Teenage Years. In: VLDB, pp. 9-16 (2006)
6. Ives, G., Halevy, A., Weld, D.: Adapting to Source Properties in Processing Data Integration Queries. In: SIGMOD, pp. 395-406 (2004)
7. Josifovski, V., Schwarz, P., Haas, L., Lin, E.: Garlic: A new flavor of federated query pro-cessing for DB2. In: SIGMOD, pp. 524-532 (2002)

8. Kossmann, D.: The State of the Art in Distributed Query Processing. J. ACM Computing Surveys. 32(4), 422-469 (2000)
9. Pentaris, F., Ioannidis, Y.: Query Optimization in Distributed Networks of Autonomous Da-tabase Systems. J. ACM Transactions on Database Systems. 31(2), 537-583 (2006)
10. Risch, T., Josifovski, V.: Distributed Data Integration by Object-Oriented Mediator Servers. J. Concurrency and Computation: Practice and Experience. 13(11), 933-953 (2001)
11. Smart Vortex Project, http://www.smartvortex.eu/
12. Truong, T., Risch, T.: Scalable Numerical Queries by Algebraic Inequality Transformations. In: DASFAA, pp. 95-109 (2014)
13. Zhu, M., Risch, T.: Querying combined cloud-based and relational databases. In: CSC, pp. 330-335 (2011)
14. Zhu, M., Stefanova, S., Truong, T., Risch, T.: Scalable Numerical SPARQL Queries over Relational Databases. In: LWDM workshop, pp. 257-262 (2014)

Paper IV

Khalid Mahmood, Tore Risch, and Minpeng Zhu.
2015. Utilizing a NoSQL Data Store for Scalable Log Analysis. In Proceedings of the 19th International Database Engineering & Applications Symposium (IDEAS '15).
ACM, New York, NY, USA, 49-55.
DOI=10.1145/2790755.2790772
http://dx.doi.org/10.1145/2790755.2790772

IDEAS '15, July 13 - 15, 2015, Yokohama, Japan

Utilizing a NoSQL Data Store for Scalable Log Analysis

Khalid Mahmood Tore Risch Minpeng Zhu
Dept. of Information Technology Uppsala University, Sweden
khalid.mahmood@it.uu.se tore.risch@it.uu.se minpeng.zhu@it.uu.se

ABSTRACT
A potential problem for persisting large volume of data logs with a conventional relational database is that loading massive logs produced at high rates is not fast enough due to the strong consistency model and high cost of indexing. As a possible alternative, a modern NoSQL data store, which sacrifices transactional consistency to achieve higher performance and scalability, can be utilized. In this paper, we investigate to what degree a state-of-the-art NoSQL database can achieve high performance persisting and fundamental analyses of large-scale data logs from real world applications. For the evaluation, a state-of-the-art NoSQL database, MongoDB, is compared with a relational DBMS from a major commercial vendor and with a popular open source relational DBMS. MongoDB is chosen as it provides both primary and secondary indexing compared to other popular NoSQL systems. These indexing techniques are essential for scalable processing of queries over large scale data logs. To explore the impact of parallelism on query execution, sharding was investigated for MongoDB. Our results revealed that relaxing the consistency did not provide substantial performance enhancement in persisting large-scale data logs for any of the systems. However, for high-performance loading and analysis of data logs, MongoDB is shown to be a viable alternative compared to relational databases for queries where the choice of an optimal execution plan is not critical.

Categories and Subject Descriptors
H.2.m [Database Management]: Miscellaneous

General Terms
Measurement, Performance, Experimentation

Keywords
NoSQL data stores, large-scale log analysis, log archival, bulk loading, sharding

1 INTRODUCTION

Relational databases can be used for large-scale analysis of data logs from industrial applications such as sensor readings [21] [25] [29] and stream logs [27] [28]. Persisting large volume of data logs produced at high rate requires high performance bulk loading of data into a database before analysis. However, the loading time for relational databases may be time consuming due to full transactional consistency [9] and high cost of indexing [23]. In contrast to relational DBMSs, NoSQL databases are designed to perform simple tasks with high scalability [5]. For providing high performance updates, NoSQL databases generally sacrifice strong consistency by providing so called eventual consistency compared with the ACID transactions of regular DBMSs. NoSQL databases can be utilized for typical historical analysis of log data or numerical log analytics where transactional consistency conforming ACID compliance is not required.

It has been argued in [23] that relational DBMSs can achieve the same performance as NoSQL database systems by specifying relaxed consistency to eliminate overhead. In [12] it is shown that this overhead is almost equally divided between four components for a typical relational DBMS: logging, locking, latching, and buffer management. However, we did not find any experimental benchmark that investigates how a weaker consistency model for relational DBMSs and NoSQL databases can be utilized to enhance performance for persisting and analysis of data logs. Although [10] compares the performance of SQL Server and MongoDB [15] for interactive data-as-a-service based on the YCSB benchmark [6], it does not investigate the performance of the systems for scalable log analysis. A more recent investigation [13] did not consider the state-of-the-art NoSQL database, MongoDB for performance evaluation. None of the papers consider the option for relaxing the consistency for both types of systems.

Unlike NoSQL data stores, relational databases provide advanced query languages and optimization technique for scalable analytics. Paper [19] demonstrates that indexing is a major factor for providing scalable performance, making relational databases having a performance advantage compared to a NoSQL data store without proper indexing to speed up analytical tasks. Like relational databases, MongoDB provides a query language as well as primary and secondary indexing, which should be well suited for analyzing persisted logs. Unlike relational databases and MongoDB, most other popular NoSQL data stores [22], Cassandra [2], Redis [20], HBase [1], Memcached [7], and CouchDB [3], do not provide full secondary indexing and query processing to transparently utilize indexes, which is essential for scalable performance of inequality queries. CouchDB has secondary indexes, but queries have to be written as map-reduce views [5], not transparently utilizing indexes.

In this paper we compare MongoDB with state-of-the-art relational DBMSs to investigate at what degree a state-of-the-art NoSQL database is suitable for persisting and analyzing large scale data logs compared with relational databases. The performance of MongoDB is compared with a commercial DBMS from a major relational vendor, called DB-C, and a popular open source relational DBMS, called DB-O.

The performance evaluation covers the bulk loading capacities of the systems w.r.t. indexing and relaxed consistency. We define three fundamental queries for accessing and analyzing persisted logs to investigate the efficiency of query processing and index utilization of the DBMSs. The properties of these queries are key selection, range search, and aggregation, which are fundamental to the analysis of persisted logs [25] [29]. We utilize data logs from a real world application [21] consisting of more than 1 billion sensor readings from industrial equipment.

Furthermore, the impact of MongoDB's auto-sharding (automatic partitioning) is investigated for persisted log analysis in order to explore whether its data partitioning can provide performance advantages for bulk loading and query execution.

In summary, the main contribution of the paper is a performance evaluation of persisting and analyzing data logs under different consistency configurations. The paper provides a comparison of the suitability of the two kinds of database systems for large-scale log analysis and reveals the trade-offs between bulk-loading and different levels of consistency. We discuss the cause of the performance differences influenced by how the systems choose different indexing strategies under relaxed consistency. The investigations provide insights in the issues that future systems should consider when utilizing weaker consistency of back-end storage for persisting and analyzing of data logs.

2 PERFORMANCE EVALUATIONS

In this section, we present bulk loading strategies and fundamental queries for persisted data logs. We first measure the performance in terms of execution time for different loading strategies by relaxing consistency overhead. Then we compare the performance of fundamental queries. For MongoDB, we also investigate the impact of auto-sharding on loading and querying. Based on inspecting the query execution plans, we discuss the causes of the performance differences.

2.1 Application Scenario

The Smart Vortex EU project [21] serves as a real world application context, which involves analyzing data logs from industrial equipment. In the scenario, a factory operates some machines and each machine has several sensors that measure various physical properties like pressure, power consumption, temperature, etc. For each machine, the sensors generate logs of measurements, where each log record has timestamp *ts*, machine identifier *m*, sensor identifier *s*, and a measured value *mv*. Each measured value *mv* on machine *m* is associated with a valid time interval *[bt, et)* indicating the begin time and end time for *mv*, computed from the log time stamp *ts*. The table measures *(m, s, bt, et, mv)* will contain a large volume of log data from many sensors on different machines. There is a composite key on *(m, s, bt)*.

In the performance measurements, the logs are bulk loaded into MongoDB and the two relational DBMSs. Since the incoming sensor streams can be very voluminous, it is important that the measurements are bulk-loaded fast. After data logs have been loaded into the *measures* table, the user can perform queries to detect anomalies of sensor readings by analyzing values of *mv*. The queries are used in the performance evaluation in order to understand the performance differences for both kinds of systems.

2.2 Data Set

The evaluation is made based on measurements from a real-world application in the Smart Vortex project [21]. A typical time series formed by a small piece of a larger numerical log from the application is plotted in Figure 1. In the performance evaluations more than 1 billion log measurements, which occupies 60GB of raw data from industrial sensors is used. It is important that data loading can keep up with increasing log volume. To investigate DBMS performance with growing data volume, increasing sections of the data logs were loaded into the databases.

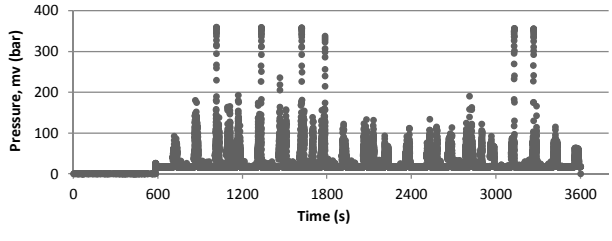


Figure 1. Pressure measurements of sensor for 1 hour

2.3 Consistency Configurations for Bulk Loading

In relational DBMSs, typical transactional overhead such as logging can be turned off and isolation level be relaxed to boost the performance. To investigate the impact of relaxed consistency levels, we configure the systems as in Table 1, which also defines the acronyms for the experiments. The lowest isolation level for the relational databases (dirty reads) corresponds to unacknowledged write concern in MongoDB, while serializable transactions correspond to *acknowledged write concern* [16]. For MongoDB, we also investigate the performance impact of autosharding, which can be combined with both consistency levels per shard. MongoDB does not have distributed transactions, therefore, synchronized updates of several shards is not supported.

Table 1. **Consistency configurations for the experiments**

Acronym	Name and Consistency Level	Properties
DB-C-S	DB-C & strong consistency	Logging, serializable isolation level
DB-C	DB-C, & weak consistency	Dirty reads, no logging
DB-O-S	DB-O & strong consistency	Logging, serializable isolation level
DB-O	DB-O & weak consistency	Dirty reads , no logging
Mongo-S	MongoDB & acknowledged write concern	Logging, serializable isolation level
Mongo	MongoDB & unacknowledged write concern	Dirty reads, no logging
Mongo-AS-S	MongoDB auto-shading & acknowledged write concern	Logging, no distributed transactions, serializable isolation
Mongo-AS	MongoDB auto-shading & unacknowledged write concern	No logging, no distributed transactions, dirty reads

We evaluated several alternatives of bulk loading by utilizing the different levels of consistency configurations. First, the impact of relaxed consistency is investigated and then the best consistency option for each system is used in all other experiments.

2.4 Fundamental Queries

The queries used in the performance evaluation are very fundamental to analytics over numerical logs and provide basic building blocks of analytics of persisted logs [21] [25] [29]. We made the experimental queries simplistic in nature, which is one of the four major criteria of domain specific benchmarks [11], to demonstrate the credibility of the performance trade-offs for the systems. The queries essentially explore the performance and scalability of primary and secondary index utilization for growing data logs. The first query, *key look-up query (Q1)*, gets a sensor reading for a given timestamp. The second query, *range query (Q2)* detects deviations of sensor readings from expected values. The third query, *aggregation query (Q3)*, performs aggregation of measurement deviations from persisted logs.

2.4.1 The Key Lookup Query, Q1

The task involves finding measured values mv for a given machine *m*, sensor *s*, and begin time, *bt*. The query expressed in SQL and MongoDB's query language [16], respectively, is specified as follows:

```
-- SQL                                    //MongoDB
SELECT m, s, mv                           db.measures.find(
FROM measures                             { m:?, s:?, bt:? }, { m: 1, s: 1 mv: 1})
WHERE m =? AND s =? AND bt =?
```

In order to provide scalable performance of the query, we need an index on the composite key. In all systems we index by defining a composite B-tree primary key index on *(m, s, bt)*. This query demonstrates the performance of primary key index utilization of the three systems.

2.4.2 Range Query, Q2

This query involves finding unexpected sensor readings by observing measured values *mv* that deviate from an expected value. Here, the sensor readings with the measured value *mv* higher than the unexpected value are retrieved. Such a query can be expressed in SQL and MongoDB as follows:

```
-- SQL                                    //MongoDB
SELECT *                                  db.measures.find(
FROM measures                             { mv: { $gt: ? } })
WHERE mv>?
```

In order to improve the performance of this query, we need a secondary ordered index on value, *mv*. Query Q2 shows the performance of secondary B-tree indexing and how well the query optimizer can utilize the index. Since the efficiency of a secondary index is highly dependent on the selectivity, the query was executed with different query selectivities by providing the appropriate ranges of *mv*. The correspondence between choice of *mv* and query selectivities for Q2 for the data set is plotted in Figure 2.

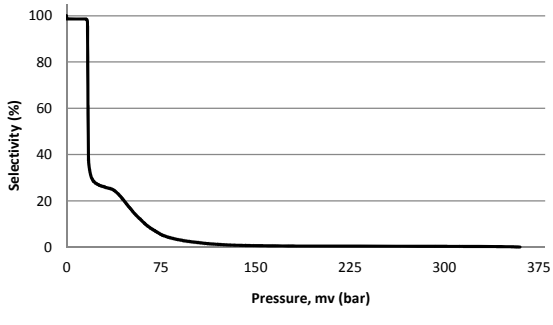


Figure 2. Measured value to selectivity mapping

Based on the data illustrated by Figure 2, we execute Q2 for value of *mv* resulting in the selectivities 0.002%, 0.02%, 0.2%, 0.25 %, and 1%, resulting around 0.02, .2, 2, 2.5,5 and 10 millions of log records.

Query Q2 is an example of a very fundamental analytics query that involves inequality comparisons. Complex analytics queries usually involve such inequalities and can often be rewritten into inequality queries like Q2, as automated in [25].

2.4.3 Aggregate Query, Q3

This query counts the total number of sensor readings having a measurement anomaly, using the same inequality as in Q2. Such a query is expressed in SQL and MongoDB as follows:

```
-- SQL                                  //MongoDB
SELECT COUNT(*)                 db.measures.count( { mv: {$gt: ?}})
FROM measures
WHERE mv > ?
```

Similar to Q2, this query was executed for different selectivities utilizing a secondary index on *mv*. In difference to Q2, which returns a large volume of abnormal sensor readings, Q3 returns a single aggregated value. The query has insignificant network communication overhead compared to Q2.

2.5 Indexing Strategy

To speed up lookups of sensor readings for a given timestamp, we define a composite index on machine id *m*, sensor id *s* and begin time *bt*. For query Q2 and Q3, we define an extra secondary index on *mv* in addition to the composite key index. The data is then bulk loaded and the three fundamental queries were executed.

2.6 Benchmark Configuration

The non-sharding experiments are performed on a computer running Intel® CoreTM i7, 3.0GHz CPU with Windows Server 2013 operating system. The computer has 16GB of physical memory.

2.6.1 MongoDB Configuration

MongoDB version 2.4.8 is used for the performance evaluation. Relational tables are represented as collections of binary-JSON (BSON) objects [14] in MongoDB. Since the attribute names are stored inside each BSON object, short attribute names are used to make the database compact.

The sharding experiments were run on a cluster of five nodes connected by a one Gbit Ethernet switch. Each node had the same hardware configuration as the non-sharding experimental setup. We ran one *mongod* process

managing each shard, one *config_db* process managing meta-data, and one *mongos* process for the MongoDB coordinator. The client, *mongos,* and *config_db* were run on the same node separate from the shards *mongod*.

2.6.2 Relational DBMS Configurations

The query result cache was turned off for both relational DBMSs. Also MongoDB does not utilize any query result cache when executing queries.

2.6.3 Benchmark Execution

For each system we measured the bulk-load time for 167, 333, 667, and 1000 million sensor readings consisting of approximately 10GB, 20GB, 40GB, and 60GB of data, respectively. The raw data files were stored in CSV format where each individual row represents a sensor reading for machine *m*, sensor *s*, begin time *bt*, end time *et*, and the measured value *mv*.

The bulk loading into the relational DBMSs and MongoDB were performed utilizing their batch commands for bulk loading CSV files.

The scalabilities of all fundamental queries were evaluated with the largest data set of 1 billion sensor records (60 GB) for all the systems.

To enable incremental bulk loading of new data into existing tables, the indexes should always be predefined in all experiments, rather than building them after the bulk loading. Although one might consider the option of bulk loading first and then building the index, this will contradict the notion in our application scenario, where bulk loading and analyzing streaming logs is a continuous process that demands incremental loading of the data into pre-existing tables. Nevertheless, we also made performance evaluations of building the indexes after bulk loading, which is faster compared to incremental bulk loading (Figure 4).

To provide stable results for bulk loading, we made all the experiment starting with empty databases. For each query, we measured the average time of three executions. The standard deviations of the measurements were less than 1%.

2.7 Experimental Results

For investigating the performance of bulk loading and queries for different consistency configurations, the following experiments were conducted.

2.7.1 Performance of Bulk Loading

In Figure 3, we observe that all systems offer scalable loading performance, except DB-O (*DB-O and DB-O-S*) and sharded MongoDB (*Mongo-AS and Mongo-AS-S*). DB-O scales significantly worse than DB-C and MongoDB for bulk-loading, whereas Mongo-AS is faster than DB-O. Both Mongo-AS and Mongo-AS-S are much slower compare to DB-C and non-sharded Mon-

goDB. We speculate that this performance degradation is due to MongoDB's internal re-balancing of data among the shards during bulk-loading.

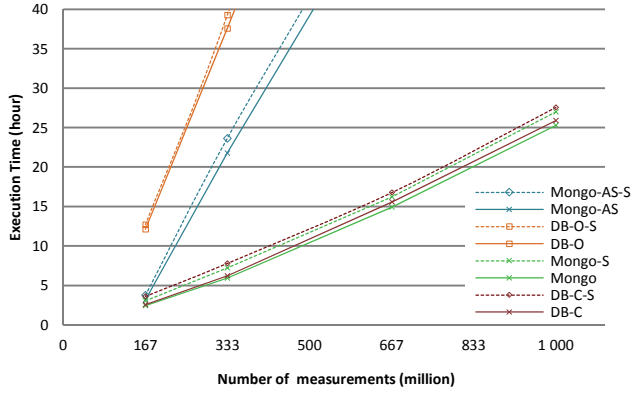


Figure 3. The performance of bulk loading with different consistency configurations

For bulk loading of large databases, the improvement of weak consistency is around 24.8% for DB-C (*DB-C vs. DB-C-S*), while it is around 26% for MongoDB (*Mongo vs. Mongo-S*). MongoDB with weak consistency performs best compared to other systems. In summary, relaxing transactional overhead did not provide substantial performance improvement for any system. From now on we always use the faster weak consistency levels in the experiments.

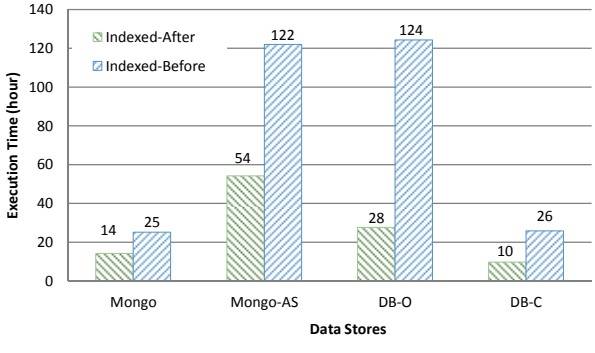


Figure 4. The performance of building indexes before and after bulk loading

Figure 4 illustrates the performance degradation for bulk loading one billion records incrementally with predefined indexes (*Indexed-Before*) compared with building the indexes after all data are loaded (*Indexed-After*). For *Indexed-After*, we show the total time of bulk loading and building the indexes. Here for the *Indexed-After* experiment, DB-C performs best. Although all systems demonstrate better performance with *Indexed-After*, this option pre-

vents incremental bulk loading and is not suitable for incrementally persisting logs.

2.7.2 Performance of Key Lookup Query (Q1)

Figure 5 shows the performance of key lookup query Q1 to retrieve a particular sensor record. As expected, indexing the key provides scalability of Q1 in all systems, with DB-C being fastest.

Figure 5. Performance of key lookup query (Q1)

2.7.3 Performance of Range Query (Q2)

Figure 6 shows the performance of query Q2 with both primary and secondary indexes defined. The selectivities are varied from 0.002% up to 1.0% for 1 billion sensor records. Clearly there is a problem with secondary indexes for inequality queries in DB-O. Both sharded and non-sharded MongoDB and DB-C scale substantially better and is therefore investigated further in Figure 7.

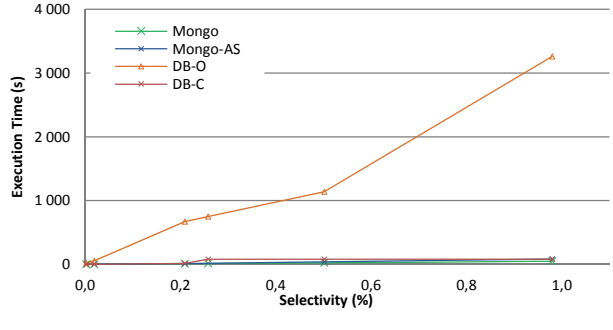


Figure 6. The performance of range query (Q2)

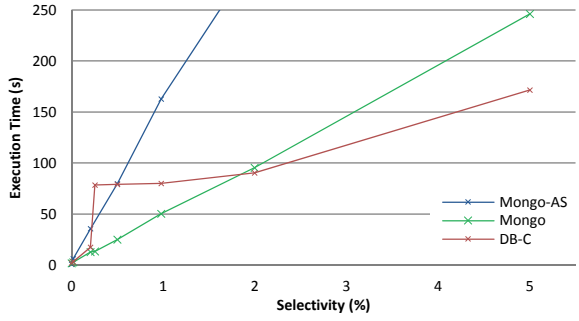


Figure 7. The performance of Q2 for DB-C and MongoDB

Figure 7 compares Q2 for DB-C and both sharded and nonsharded MongoDB while varying the selectivities from 0.002% up to 5.0%. The figure shows that DB-C switches from scanning the secondary index to full table scan when around 0.25% of the rows are selected. This makes DB-C faster than all configurations of MongoDB for non-selective queries (selecting more than 2.0%), because MongoDB does not switch the execution strategy and continues with an index scan for growing selectivities. Mongo-AS is clearly slowest for non-selective queries, while the query optimizer of DB-C makes it the system with the most stable performance. In the Figure, no performance differences can be observed for selectivities less than 0.2% and therefore Table 2 details the performance differences for selectivities up to 0.2%.

Table 2. **The performance of very selective Q2 for DB-C and MongoDB**

Selectivity	Records	Performance of Q2 (second)		
%	*n*	Mongo-AS	Mongo	DB-C
0.002	237,360	0.56	1.6	1.8
0.02	2,256,240	5.89	2.7	3.3
0.20	22,261,280	35.5	12.7	17.4

Table 2 shows that for very selective queries (selectivity from 0.002% to 0.2%) Mongo-AS is the fastest, since a relatively small number of records have to be transferred, which results in less network communication overhead. Sharded MongoDB is fastest only for highly selective queries.

2.7.4 Performance of Aggregate Query (Q3)

Figure 8 shows the performance of the aggregate query Q3 where a single value is returned. We use the same selectivities of the condition inside the aggregate as for Q2. Here it turns out that Mongo-AS performs much better compared to the corresponding performance of Q2 in Figure 7, since each shard performs a parallel scan and then sends a single result object to the coordinator.

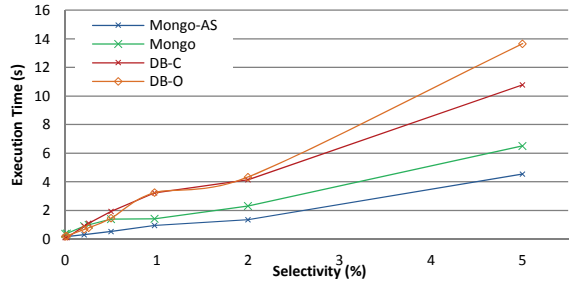


Figure 8. The performance of the aggregate query (Q3)

The performance of DB-O for Q3 is much better than for Q2 in Figure 6, but it still scales worse than the other systems. For aggregates over non-selective conditions (5%), Mongo-AS scales best, being 1.4 times faster than non-sharded MongoDB and DBC, respectively. However, five shards provides only 40% speedup in our settings.

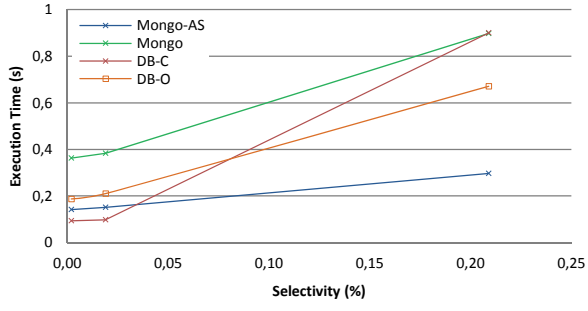


Figure 9. The performance of Q3 for smaller selectivities

Figure 9 further highlights the performance of Q3 with highly selective conditions, where DB-C is fastest for selective queries, while the performance of Mongo-AS is slightly slower due to overhead of coordination among shards.

Table 3. **Qualitative summary of the experimental results**

Task\System		DB-O	DB-C	Mongo	Mongo-AS
Bulk Loading (Figure 3)		very bad	good	very good	bad
Key lookup, Q1 (Figure 5)		good	very good	good	good
Range query, Q2	Selective (Table 2, Fig. 7)	very bad	good	good	good
	Non-selective (Figure 6,7)	very bad	very good	good	bad
Aggregate query, Q3	Selective (Figure 9)	good	good	good	good
	Non-selective (Figure 8)	bad	bad	good	very good

The overall results of the performance evaluation are summarized in Table 3, where MongoDB is shown to have comparable performance as the state-of-the-art relational database from a major commercial vendor (DB-C).

3 RELATED WORK

Typical TPC benchmarks [24] such as TPC-C, TPC-DS, and TPC-H are targeted towards either OLTP or decision support, not for large scale log analysis, which often requires scalable persisting and querying over persisted logs, the focus of this paper.

Floratou et al. [10] compared the performance of SQL Server and MongoDB for interactive data-as-a-service queries based on the YCSB benchmark [6], showing that SQL Server has significant performance advantages over MongoDB. However, the work neither explored the options of relaxing consistency overheads nor investigated indexing and query optimization issues for scalable execution of persisted data logs. Dede et al. [8] evaluated the performance of MongoDB and Hadoop for scientific data analysis, but not for scalable log analysis and there was no comparison with relational DBMSs.

Barahmand et al. [4] compared the performance of an SQL solution with MongoDB for interactive social networking actions and sessions, which does not fit into the context of persisting and analyzing logs.

Wei et al. [26] utilized MongoDB for storing and analyzing network logs. Although they provide queries that analyze network logs, they did not compare the performance with other systems.

Finally, the performance of online incremental bulk loading with a main-memory DBMS was investigated in [17] [18]. By contrast, our focus is on comparing disk-based NoSQL and relational databases for persisting large-scale data logs.

To our best knowledge we did not find any performance evaluation that compares MongoDB with relational DBMS in the context of persisting and analyzing of numerical logs.

4 CONCLUSIONS AND DISCUSSIONS

The conclusions from the evaluation can be divided into three different factors influencing performance: (i) relaxing consistency, (ii) indexing and query processing, and (iii) sharding.

First, we discovered that relaxing the consistency does not provide any substantial performance enhancement in querying large scale data logs for neither SQL nor NoSQL databases. Although it is shown in [12] that remov-

ing transactional overhead can improve performance up to 20 times for updates, we discovered that both commercial and open source relational databases provide less than 25% performance improvement for bulk-loading with relaxed transaction consistency. In contrast to the aggressive modification of the database kernel in [12], a common user will not be able to modify the DBMS kernel but has to rely on the options provided by the system. For MongoDB, weak consistency configuration of bulk loading provides around 26% improvement.

For bulk loading in general, both MongoDB and DB-C scale substantially better than DB-O. For the largest data size, bulk loading with non-distributed MongoDB and DB-C are more than five times faster than DB-O. Distributing MongoDB by autosharding is about 4 times slower than non-distributed MongoDB and DB-C.

All systems perform well for looking up records matching the key (query Q1) by utilizing a primary key index. For the analytical task of range comparisons between a non-key attribute and a constant (query Q2), both MongoDB and DB-C scale substantially better than DB-O. A more careful comparison of DB-C and MongoDB revealed that DB-C scales better for nonselective queries, while MongoDB is faster for selective ones. The reason is that, unlike MongoDB and DB-O, DB-C switches from a non-clustered index scan to a full table scan when the selectivity is sufficiently low, while MongoDB (and DB-O) continues to use an index scan even for non-selective queries.

The aggregation query (query Q3) scales for all systems by utilizing the secondary index when computing an aggregated value. Here, sharded MongoDB scales best being around 1.4, 2.4, and 9.5 times faster than non-sharded MongoDB, DB-C, and DBO, respectively. The reason is that a parallel scan without sending lots of results among distributed shards speeds up query execution. Therefore, we conclude that, only when an analytics task is inherently parallel with insignificant communication/datatransfer among parallel nodes, distributed MongoDB (or similar NoSQL data store) is an alternative to vertical scaling to speed up the analytics.

To conclude, non-sharded MongoDB performs significantly better compared to DB-O and has comparable performance with DB-C, making it suitable for large scale persisting and analyzing logs. However, DB-C demonstrates that relational databases can have performance advantages compared to both distributed and non-distributed NoSQL databases by having a sophisticated query optimizer. NoSQL databases can also be equipped with more sophisticated query optimizers as in state-of-the-art relational DBMSs, which will improve query performance. However, some NoSQL databases such as MongoDB provide a flexible schemaless paradigm which makes query optimization challenging due to the absence of rigid schema and proper data statistics.

Finally, although we have discussed the issues of persisting and analysis of data logs, our results can be utilized also in other large-scale and data intensive application scenarios where bulk loading with relaxed consistency and scalable query execution are required.

For high performance loading and analysis of large-scale data logs, MongoDB is shown to be a viable alternative compared to relational databases.

5 ACKNOWLEDGMENTS

This work is supported by EU FP7 project Smart Vortex, the Swedish Foundation for Strategic Research under contract RIT08-0041, and eSSENCE.

6 REFERENCES

1 Apache Software Foundation. 2015. Apache HBase. (June 2015). Retrieved June 18, 2015 from http://hbase.apache.org/

2 Apache Software Foundation. 2015. Cassandra. (June 2015). Retrieved June 18, 2015 from http://cassandra.apache.org/

3 Apache Software Foundation. 2015. CouchDB. (June 2015). Retrieved June 18, 2015 from http://couchdb.apache.org/

4 Barahmand, S., Ghandeharizadeh, S. and Yap, J. 2013. A comparison of two physical data designs for interactive social networking actions. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management* (CIKM '13). ACM, New York, NY, USA, 949-958. DOI=http://doi.acm.org/10.1145/2505515.2505761

5 Cattell, R. 2011. Scalable SQL and NoSQL data stores. *ACM SIGMOD Rec.* 39, 4 (May 2011), 12-27. DOI=http://doi.acm.org/10.1145/1978915.1978919

6 Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R. and Sears, R. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing* (SoCC '10). ACM, New York, NY, USA, 143-154. DOI=http://doi.acm.org/10.1145/1807128.1807152

7 Danga Interactive. 2015. Memcached. (June 2015). Retrieved June 18, 2015 from http://www.memcached.org/

8 Dede, E., Govindaraju, M., Gunter, D., Canon, R.S. and Ramakrishnan, L. 2013. Performance evaluation of a MongoDB and hadoop platform for scientific data analysis. In *Proceedings of the 4th ACM workshop on Scientific cloud computing* (Science Cloud '13). ACM, New York, NY, USA, 13-20. DOI=http://doi.acm.org/10.1145/2465848.2465849

9 Doppelhammer, J., Höppler, T., Kemper, A. and Kossmann, D. 1997. Database performance in the real world: TPC-D and SAP R/3. In *Proceedings of the*

1997 ACM SIGMOD international conference on Management of data (SIGMOD '97), Joan M. Peckman, Sudha Ram, and Michael Franklin (Eds.). ACM, New York, NY, USA, 123-134. DOI=http://doi.acm.org/10.1145/253260.253280

10 Floratou, A., Teletia, N., DeWitt, D.J., Patel, J.M. and Zhang, D. 2012. Can the elephants handle the NoSQL onslaught?. *Proc. VLDB Endow.* 5, 12 (August 2012), 1712-1723. DOI=http://dx.doi.org/10.14778/2367502.2367511

11 Gray, J. 1992. *Benchmark Handbook: For Database and Transaction Processing Systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

12 Harizopoulos, S., Abadi, D.J., Madden, S. and Stonebraker, M. 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. DOI=http://doi.acm.org/10.1145/1376616.1376713

13 Kuhlenkamp, J., Klems, M. and Röss, O. 2014. Benchmarking scalability and elasticity of distributed database systems. *Proc. VLDB Endow.* 7, 12 (August 2014), 1219-1230. DOI=http://dx.doi.org/10.14778/2732977.2732995

14 MongoDB Inc. 2015. BSON Types. (June 2015). Retrieved June 18, 2015 from http://docs.mongodb.org/manual/reference/bson-types/

15 MongoDB Inc. 2015. MongoDB. (June 2015). Retrieved June 18, 2015 from http://www.mongodb.org/

16 MongoDB Inc. 2015. The MongoDB 2.4 Manual. (June 2015). Retrieved June 18, 2015 from http://docs.mongodb.org/v2.4/

17 Neumann, T. and Weikum, G. 2010. x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. *Proc. VLDB Endow.* 3, 1-2 (September 2010), 256-263. DOI=http://dx.doi.org/10.14778/1920841.1920877

18 Neumann, T. and Weikum, G. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal* 19, 1 (February 2010), 91-113. DOI=http://dx.doi.org/10.1007/s00778-009-0165-y

19 Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., Dewitt, D.J., Madden, S. and Stonebraker, M. 2009. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data* (SIGMOD '09), Carsten Binnig and Benoit Dageville (Eds.). ACM, New York, NY, USA, 165-178. DOI=http://doi.acm.org/10.1145/1559845.1559865

20 Salvatore Sanfilippo. 2015. Redis. (June 2015). Retrieved June 18, 2015 from http://redis.io/

21 Smart Vortex. 2015. Retrieved June 18, 2015 from http://www.smartvortex.eu/

22 solid IT. DB-Engines Ranking. (June 2015). Retrieved June 18, 2015 from http://db-engines.com/en/ranking

23 Stonebraker, M. 2010. SQL databases v. NoSQL databases. *Commun. ACM* 53, 4 (April 2010), 10-11. DOI=http://doi.acm.org/10.1145/1721654.1721659

24 Transaction Processing Performance Council (TPC). 2015. Active TPC Benchmarks. (June 2015). Retrieved June 18, 2015 from http://www.tpc.org/information/benchmarks.asp

25 Truong, T. and Risch, T. 2014. Scalable Numerical Queries by Algebraic Inequality Transformations. In *Proceedings of the 19th International Conference on Database Systems for Advanced Applications* (DASFAA '14) (Bali, Indonesia, 2014). Springer International Publishing, 95–109. DOI=http://dx.doi.org/10.1007/978-3-319-05810-8_7

26 Wei, J., Zhao, Y., Jiang, K., Xie, R. and Jin, Y. Analysis farm: A cloud-based scalable aggregation and query platform for network log analysis. In *Proceedings of the 2011 International Conference on Cloud and Service Computing* (CSC '11). IEEE Computer Society, Washington, DC, USA, 354-359. DOI=http://dx.doi.org/10.1109/CSC.2011.6138547

27 Zeitler, E. and Risch, T. 2011. Massive Scale-out of Expensive Continuous Queries. *Proc. VLDB Endow.* 4, 11 (2011), 1181–1188.

28 Zeitler, E. and Risch, T. 2010. Scalable splitting of massive data streams. In *Proceedings of the 15th International Conference on Database Systems for Advanced Applications,* (DASFAA '10) (Tsukuba, Japan, Apr. 2010). Springer International Publishing, 184–198. DOI=http://dx.doi.org/10.1007/978-3-642-12098-5_15

29 Zhu, M., Stefanova, S., Truong, T. and Risch, T. 2014. Scalable Numerical SPARQL Queries over Relational Databases. In *Proceedings of 4th international workshop on linked web data management* (Athens, Greece, 2014).