UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1352

Main-Memory Query Processing Utilizing External Indexes

THANH TRUONG

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2016

ISSN 1651-6214
ISBN 978-91-554-9509-1
urn:nbn:se:uu:diva-280374

Dissertation presented at Uppsala University to be publicly examined in 2446, ITC, Lägerhyddsvägen 2, Uppsala, Uppsala, Wednesday, 4 May 2016 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Martin Kersten (National research institute for mathematics and computer science in the Netherlands (CWI)).

Abstract
Truong, T. 2016. Main-Memory Query Processing Utilizing External Indexes. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1352. 45 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-554-9509-1.

Many applications require storage and indexing of new kinds of data in mainmemory, e.g. color histograms, textures, shape features, gene sequences, sensor readings, or financial time series. Even though, many domain index structures were developed, very a few of them are implemented in any database management system (DBMS), usually only B-trees and hash indexes. A major reason is that the manual effort to include a new index implementation in a regular DBMS is very costly and time-consuming because it requires integration with all components of the DBMS kernel. To alleviate this there are some extensible indexing frameworks. However, they all require re-engineering the index implementations, which is a problem when the index has third-party ownership, when only binary code is available, or simply when the index implementation is complex to re-engineer. Therefore, the DBMS should allow including new index implementations without code changes and performance degradation. Furthermore, for high performance the query processor needs knowledge of how to process queries to utilize the plugged-in index. Moreover, it is important that all functionalities of a plugged-in index implementation are correct. The extensible main memory database system (MMDB) Mexima (Mainmemory External Index Manager) addresses these challenges. It enablestransparent plugging in main-memory index implementations without code changes. Index specific rewrite rules transform complex queries to utilize the indexes. Automatic test procedures validate the correctness of them based on user provided index meta-data. Moreover, the same optimization framework can also optimize complex queries sent to a back-end DBMS by exposing hidden indexes for its query optimizer. Altogether, Mexima is a complete and extensible platform for transparently index integration, utilization, and evaluation

Keywords: Database indexing, query processing, index structures, main-memory, index validation

Thanh Truong, Department of Information Technology, Division of Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-554-9509-1
urn:nbn:se:uu:diva-280374 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-280374)

Dedication

This humble thesis work is dedicated to:

My parents ba Minh and má Thảo

For all your supports and encouragements

My wife Diễm

For being with me through the hardest times

My lovely daughters Anna and Lisa

List of Papers

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

I T. Truong and T. Risch: Transparent inclusion, utilization, and validation of main-memory domain indexes, *27th International Conference on Scientific and Statistical Database Management (SSDBM)*, San Diego, United States, June 29-Juli 1, 2015.
I am the primary author of this paper.

II T. Truong, T. Risch: Scalable Numerical Queries by Algebraic Inequality Transformations, *19th International Conference on Database Systems for Advanced Applications (DASFAA*), Bali, Indonesia, April 21-24, 2014.
I am the primary author of this paper.

III M.Zhu, S.Stefanova, T.Truong, and T.Risch: Scalable Numerical SPARQL Queries over Relational Databases, *4th international workshop on linked web data management (LWDM 2014)*, Athens, Greece, March 28, 2014.
I am one of the co-authors of this paper.

IV S.Badiozamany, L.Melander, T.Truong, C.Xu, and T.Risch: Grand Challenge: Implementation by Frequently Emitting Parallel Windows and User-Defined Aggregate Functions, *Proc. The 7th ACM International Conference on Distributed Event-Based Systems*, DEBS 2013, Arlington, Texas, USA, June 29 - July 3, 2013
I am one of the co-authors of this paper.

V K.Mahmood, T.Truong, and T.Risch: NoSQL Approach to Large Scale Analysis of Persisted Streams, *30th British International Conference on Databases, Edinburgh (BICOD)*, Scotland, July 6-8, 2015.
I am one of the co-authors of this paper.

Contents

Abbreviations

DBMS	Database Management System
Mexima	Main-memory External Index Manager
BAO	Basic access operator
SSF	Special search function
ISF	Index sensitive function
MMDB	Main-memory database system

Acknowledgements

Many people helped to make this PhD possible. First and foremost, my supervisor Prof. Tore Risch. I appreciate your contributions of time, ideas, and funding to my Ph.D. Your enthusiasm for database research inspired and motivated me. It was countless time you caught my sloppy code, or pointed out that my arguments were either weak, or wrong. Honestly, several times I walked out of your office door and thought that I would never come back. However, I did bounce back thanks to your encouragements. It does not matter how much I was struggling along the way. It is important that I am finishing strong.

I would like to say many thanks to Anders Berglund for accepting me to MSc studies at Uppsala University, where I have met so many interesting people and friends. Your acceptance brought me on a very long journey that lasts several cold and dark winters and bright summers in Uppsala. There is no way I can pick up Rikssvenska but at least I think that surströmming is not that horrible.

To my fellow members in UDBL group, you guys made my PhD time here more memorable: Kjell Orsborn, thank you for assisting me in teaching assignments. Erik Zeilter, and Manivasakan Sabesan, thank you for literally alerting me to some obstacle I might encounter along the way. I am also thankful to Lars Melander, and Andrej Andrejev, whom I did not talk much with but I enjoyed our technical discussions during these years. Cheng Xu, I am grateful for your contribution to the accepted paper that we co-authored in 2013. Sobhan Badiozamany, you deserve my thanks for sharing your opinions in various things including research ideas, life in Sweden, and various things. Thanks you for giving me rides from Ultuna to Flogsta after playing futsal very late in 2014. Silvia Stefanova, I am thankful for your friendship and talks whenever we had break or lunch together. Minpengzhu, we have talked a lot regarding research, life, and other non-serious matters. I have learnt something from your determination, focus, and hard-works. As a side note, you have not ever said yes to a fika or beer somewhere in town. I will keep on asking you then.

To Matteo Magnani, thanks you for your friendship, your humor, and your Italian dishes that I gladly tasted every time. I also want to thank Yunyun Zhu for coordinating our language meetings. Yes, Lillemor Arronson, jag är tacksam för ditt hjälp varje gång vi träffas. I would like to thank Seif Alwan for your hospitality every time I come by and for listening to me complaining of

work and study. Ulrika Andersson, I want to thank you for your excellent support regarding working matters.

In regards to Vietnamese friends, I thank you for dinners, gatherings, in addition, soccer that we played together.

To all, whom I had the privilege of making friends with, we might not be able to talk or see each other often but you made my life more colorful and enjoyable. I am thankful for that.

I would like to thank my family for all their love and encouragement. For my parents ba Minh and má Thảo who raised me with curiosity for science and supported me in all decisions I made. You are my lifetime friends, sources of inspiration, and the ones whom I seek words of wisdoms from. To my sisters Thuỷ, Thư and my brother Thái, you might not interest in what I was doing these years but I am indebted to your caring and concern.

And Diễm. I have taken a long pause to think before I write this but I still do not know where I should begin. You are just amazing as my best friend, and the love of my life. We share not only dreams, goals, but also disappointments that life has thrown to us. Without your love, support, and encouragement, I would not have done my PhD. I am sure that our life will turn to a new chapter soon after this. Thank you for always being there for me.

Finally, Anna and Lisa, you girls are God's blessing to us and I thank him for it every day.

1 Introduction

Main-memory databases (MMDBs) [12] [30] [46] [80] [64][49] are common approaches for many applications such as financial analyses, real-time operating systems, industrial machine sensors, and scientific applications that require fast data access, storage, and manipulation. The emergence of such domain applications put new requirements on main-memory database systems to support new kinds of data, e.g.; color histograms, textures, shape descriptions in image databases, gene sequences in biology databases, sensor readings in machine-log databases, time series data in finance, etc. To efficiently access and manipulate such domain data, MMDBs need to include new kinds of domain indexes that provide scalable facilities to query and update domain-oriented data representations.

Even though many index structures were developed, very a few of them are implemented in database systems in practice; most database systems [29] [32] provide only B-trees and hash indexes. The reason is that it is very difficult to extend a DBMS with new index structures. The manual integration effort is very costly and time-consuming since the new index needs to interface with most subcomponent of the DBMS kernel. Therefore, for new emerging domain applications that need novel index structures, it may not be feasible to develop indexes from scratch and integrate them into the DBMS. The DBMS index manager should be extensible to enable including new index structures without changing the DBMS kernel.

Existing extensible indexing frameworks [36] [51] [53] [91] support adding new indexes but they require re-engineering the index code strictly following each particular framework's coding conventions and API primitives. This may still be a daunting task because the index may have third-party ownership, perhaps only binary code is available, or simply it is very complex to re-engineer the code. Therefore, one challenge is how to add new index implementations to a DBMS without code modifications.

When plugging-in a new index implementation to a DBMS, it is important to test that all index functionalities are correct. Correctness means that all index operations should return exactly the expected results and leave the database in a consistent state after updates or bulk loading. Therefore, another challenge is how to automate the test procedures for a new plugged-in index implementation.

Adding a new index to a DBMS requires teaching the query processor properties of the new index, e.g. its supported access methods and how to rewrite

queries to make the index utilized by the query optimizer. Furthermore, complex expression in queries, e.g. for advanced analytics, often hinder the query optimizer to utilize its indexes. Therefore, an extensible indexing system has a need for *extensible query processing* in which new index specific rewrite rules can be plugged in. Such index specific rewrites improve query performance. Extensible query rewrite mechanisms could also improve the performance of advanced queries sent to a regular DBMS by transforming queries involving complex expressions. Therefore, another challenge is how to plug-in index specific rewrite-rules in an extensible query processor for utilizing the new indexes in queries.

This Thesis addresses the above challenges of extensible indexing, index testing, and index-specific query transformations in a main-memory DBMS.

The following research questions are investigated:

1. The overall research question is: How should an extensible indexing system be designed to enable transparent inclusion of index implementations without code changes in neither the DBMS kernel nor the index?
2. How should the query processor of the extensible indexing system be provided with knowledge of the access methods of a new plugged-in index to utilize transparently the index in queries? In particular: How should the query processor be extensible with new index-specific rewrite rules so that the new plugged-in index is transparently utilized in queries?
3. How should the correctness of a plugged-in index be automatically validated? In particular, what are the functionalities to test and how can the testing be automated?
4. When data is stored in a back-end database, how should the query processor transform complex queries so indexes in the back-end DBMS are utilized?

To answer these questions, we developed an extensible MMDB system called *Mexima* (*Main-memory External Index Manager)*.

To answer Research Question one, Mexima enables plugging-in different kinds of main-memory index implementations without altering or re-engineering their original source code (Paper I). An *index extension developer* takes an existing *index implementation*, writes a simple *extension driver* (interface code), and then compiles the whole module as a dynamic library or shared object called an *index extension*. Mexima loads these index extensions at runtime. This inclusion requires little development efforts and no detailed knowledge about the DBMS kernel.

To answer Research Question two, for data operations on a plugged-in index, Mexima invokes corresponding index access and update operators (Paper I). The index operators that are common for all kinds of indexes are called *basic access operators (BAOs),* i.e. methods for creating, dropping, updating,

accessing, and mapping over indexed elements respectively. Moreover, each kind of index often has *special search functions (SSFs)* that utilize index specific properties for efficient search, e.g., interval search on B-trees [67], and K-nearest neighbor and proximity search on R-trees [1] and X-trees [5]. To utilize SSFs transparently in queries the system contains an *SSF translator* that transforms query conditions into calls to SSFs.

Furthermore, when queries involve complex inequality conditions, they may hinder the query optimizer from utilizing the presence of indexes. This causes expensive scans of entire tables rather than direct index access calls. The *Algebraic Inequality Query Transformations (AQIT)* (Paper II) transforms complex queries involving inequalities into equivalent ones, which are more efficient by exposing hidden indexes.

To answer Research Question three, for each index type, the *Mexima tester* generates a number of queries that automatically test the correctness of the index implementation. The test queries use *data generators*, which are queries specified by the index extension developer to generate relevant data for testing BAOs and SSFs (Paper I).

To answer Research Question four, Mexima allows optimizing complex numerical queries sent to a back-end relational database (Paper II). The same AQIT rules used to utilize plugged-in main-memory indexes can also be used for exposing indexes in back-end relational DBMSs. Furthermore, scalable execution of rewritten complex numerical queries sent to the back-end relational database requires translating the numerical expressions into SQL (Paper III). This avoids data transfer from the backend database and enables the back-end query optimizer to utilize indexes.

This Thesis overview is organized as follows. Chapter 2 presents technology background and related work. Chapter 3 describes Mexima's architecture, including its query processing based on rewrite rules. Chapter 4 concludes the Thesis and gives some future work discussions. Finally, Chapter 5 summarizes papers I-V and states my technical contributions to each of them.

2 Background and Related Work

A *database* is a collection of information that is organized by a general-purpose software system called a *database management system (DBMS)* [60]. The DBMS enables creation, construction, manipulation, and maintenance of databases. A *data model* is the paradigm used by a DBMS for representing the structure of its databases. For example, in the relational data model [18] databases are represented as a set of tables having columns and rows. The database is manipulated by *queries*. In a DBMS, the *query processor* is responsible for translating queries into a *query plan*, which is a sequence of database operations executed by the DBMS kernel. The *query optimizer*, a component of the query processor, determines for a given query an optimized query plan likely to be the most efficient way to execute it.

In this Thesis, we focus on two important aspects: database indexing and query processing in main-memory database systems.

2.1 Main-memory database system

Most major DBMSs are designed to store data on disk and bring disk pages into main-memory as needed for processing. This model assumes computers with main-memory smaller than the databases. Nowadays, a normal computer has enough main-memory to fit most databases entirely. Therefore, to take full advantage of modern hardware, a new class of database systems was introduced: *main-memory database systems (MMDBs)* [4] [12], [29], [30]. MMDBs are most commonly used in applications that demand fast data access, storage, and manipulation, For example: enterprise applications [89], sensor networks [76], industrial data [78], and telecom applications [56]. In addition, for applications running on traditional disk-based DBMSs, when data is skewed some frequently accessed “hot” data can be kept in a main-memory database. For this, major DBMS vendors have developed their own main-memory database processing integrated with their disk-based DBMSs, e.g., Oracle TimesTen [82], Heckaton in SQL Server [10], and MySQL in-memory tables [58]. Furthermore, in recent years parallel MMDBSs have been in focus [37][86][83][62], with commercial products such as SAP HANA [77] [89], MySQL Cluster [62], and VoltDB [48].

Colum-store systems such as MonetDB [80], VectorWise [64][49], C-Store [54] (or its commercial version Vertica [2]), SybaseIQ [73], etc., are designed

to exploit the large main memories of modern computer systems efficiently when processing analytical and aggregate queries overs database in memory-mapped files. In particular, they partition a database into a collection of individual columns that are compressed and stored separately. It enables processing only the needed columns, rather than entire rows and discards other unneeded columns.

Even though main-memory database systems have been extensively studied in the past, the area of extensible indexing and query processing in a main-memory DBMS is little studied [29] and is the focus of this Thesis.

The next section discusses why indexing is needed in the context of main-memory database systems.

2.2 Indexing

A DBMS uses *indexes* to speed up the retrieval of records in response to certain search conditions [60]. An index associates a given key with a collection of addresses of matching records. To avoid physically scanning all records in a table for a given search condition on some search key, the DBMS can use the index to access only the relevant records. In a relational database, there can many indexes created for each table and an index can be associated with a single column or several columns. An index is *utilized* by a query when the query optimizer is able to use the index in the execution plan to speed up the execution of the query.

2.2.1 Indexing in MMDBs

Figure 1 illustrates the memory hierarchy layers when accessing a regular disk-based database. The widths of the triangles indicate the storage capacity of the layers, while the thicknesses of the arrows indicate the data volume transferred between the layers.

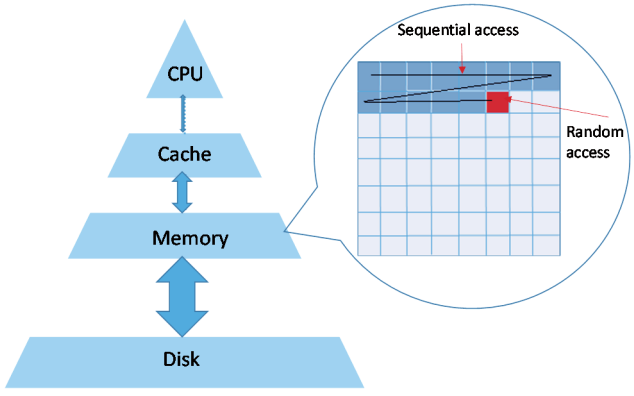

Figure 1 Memory hierarchy

The main optimization objective of a disk-based DBMS is to reduce the number of I/O accesses, which is the primary performance bottleneck. In an MMDB, all data is assumed to fit in main-memory so therefore the goals of an MMDB are to reduce memory consumption, to optimize memory cache usage, and to minimize the number of CPU cycles for maintaining data in memory [29].

In an MMDB, before data can be processed by the CPU it needs to be transferred into the memory cache as a *data block*; this is called a *cache miss*. If the same data block is referenced a second time, the access time becomes substantially faster since no transfer is needed; this is called a *cache hit*. When a cache miss happens, the computer must wait for other CPU cycle(s) before transferring another data block from memory to the cache. Therefore, sequentially reading data blocks is much faster than randomly accessing main-memory. If there is much data to access, there will be many costly cache misses so indexing will improve performance substantially for large main-memory databases.

A cache miss is analogous to a buffer pool miss in a disk-based database. The difference is that the block size in a disk-based database is substantially larger than the memory cache in an MMDB and the performance difference between a cache hit and a cache miss is substantially higher for disk-based databases. Therefore, the improvement by indexing is higher for disk-based databases.

Even though column-store systems [80] [64] [49] [54] [2] [73] can achieve high cache utilization and CPU efficiency when scanning columns, indexes on top of column-stores improve the performance with orders of magnitude for queries that do not need to scan entire columns [81][14].

An index entry in an MMDB is a data structure containing pairs *(index key, list of handles),* where the handles refer the records having the same index key. A handle can be a physical memory address, or it can be an indirect reference to physical memory in order to allow flexible memory management

[72]. Different index data structures organize the index entries in different ways, e.g. as B-trees [67], hashing [63][93], R-trees [1], TV-trees [42], etc.

The next section addresses the necessity of extending DBMSs with new domain index structures.

2.2.2 Extensible indexing

Figure 2 shows an application matrix proposed by Stonebraker in 1990s [55], which categories database applications into four quadrants. It motivates the need for DBMSs to support queries over complex data in the upper-right quadrant, which is required by many of today's applications such as multi-media, time series, and gene sequences. A key approach to support such new complex data types is the ability to include in a DBMS new data structures to represent domain data.

Query (SQL)	**Business** **Personal db** **Many applications**	**Multimedia retrieval** **Temporal data** **Measurements** **Customized search**
No Query (No SQL)	**VOD** **Text Editor** **Simple computations**	**CAD system** **Course planning** **Complex computations**
	Simple data	**Complex data**

Figure 2 Classifications of database applications

Adding a domain data type to a DBMS also requires supporting new domain operators in queries. For example, if an image data type is introduced, there is also need for query functions that compute image similarities. Importantly, inclusion of new data types requires new kinds of indexes to access the data efficiently.

Figure 3 summaries most index structures invented during 1960-1996 [90]. However, very few of them were actually implemented in a DBMS [32].

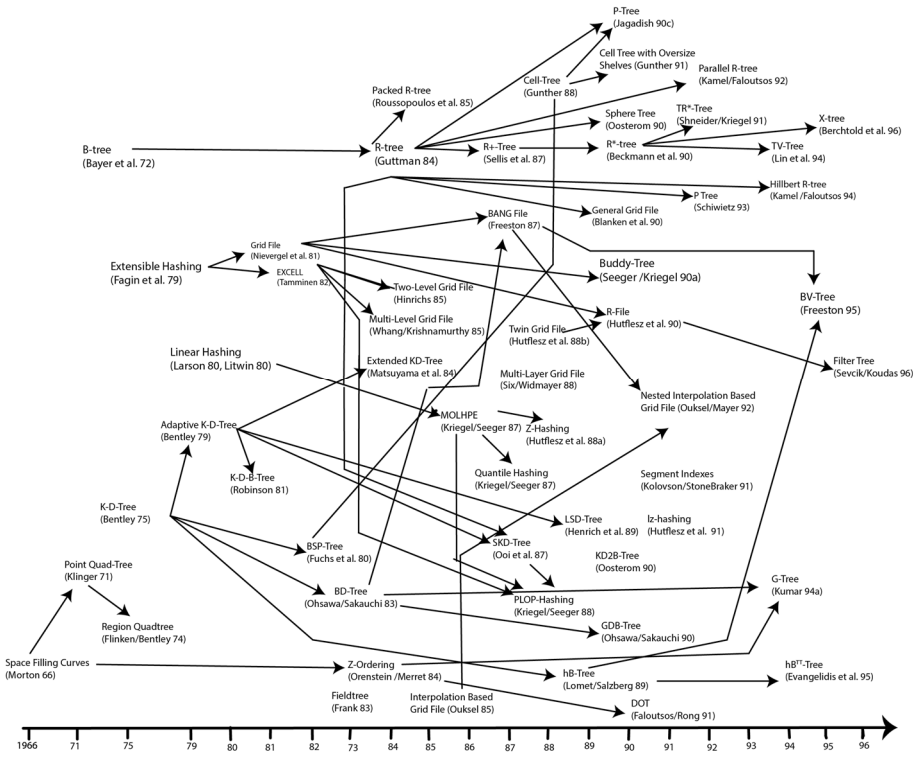

Figure 3 Summary of indexes invented during 1966 - 1996

Figure 4 shows what indexes are available today in well-known DBMSs. For example, both Oracle and MySQL employ R-trees up to 4D for geometry objects. In addition, Oracle Spatial engine supports also Linear Quad-trees [71], which uses a space-filling-curve technique [94] to decompose spatial data into linear-order data suitable for B-trees. Similarly, Microsoft SQL Server Spatial [94] and DB2 Spatial Extender [35] also use a space-filling curve to index spatial data. Alternatively, some DBMSs support function indexes [40][13], which are indexes on the result of a function. The DBMSs compute the result of the function for every update and materialize it in a B-tree or a hash table. Function indexes thus require computation for every single database update or insertion. That makes database updates more expensive and it is inapplicable for ad-hoc queries.

Index	MySQL 5.6	PostgreSQL 9.5.1	MS SQL 2014	Oracle 12c Release 1
B-tree	Yes	Yes	Yes	Yes
Hash	Yes*	No	Yes*	No
Bit-map	No	Yes	No	Yes
Spatial index	Quadratic R-tree	R-tree (GiST)	No	Quadratic R-tree Quad-tree
Function-based-index	No	Yes	Yes	Yes

Notes
- Versions
  - Oracle 12c Release 1
  - SQL Server 2014
  - MySQL 5.6
- Hash index is only available for in-memory tables.

Figure 4 Summary of indexes in some DBMSs

Another approach to improve performance of querying high-dimensional data is to use dimensionality reduction techniques such as DFT (Discrete Fourier Transform) [70], FFT (Fast Fourier Transform), or DWT (Discrete Wavelet Transform) that map from a high-dimensional space to lower dimensions. Then DBMSs can apply some low-dimensional index structures (R-trees or B-trees) to index the data. However, such reductions are lossy and thus they are applicable only on applications where loss of information is acceptable, e.g., financial databases [8] and images [7] [21].

MMDBs have even fewer index structures implemented than disk-based databases. For example, the following systems have only hash-based indexing: Memcached [4], Redis [74], RamCloud [38], Yahoo! S4 [43], and Piccolo [69]. SAP HANA has both CSB+ trees [77] [89] and hash indexes. H-store [68] and its commercial version VoltDB [48] have B-trees and hashing. For in-memory tables, MySQL 5.6 and MS SQL 2014 support hashing only.

The question is why most DBMSs implemented so few indexes, even though there is a demand for new kinds of indexing and there are many domain indexes. The answer is because it is very challenging to include new indexes into a DBMS kernel.

To simplify adding new indexes to DBMSs, several extensible indexing frameworks have been proposed. Generalized Search Trees [36] is a generalized template index structure, which provides a single code base for commonly invariant properties of B-tree-like search trees while leaving other characteristics to be specified by the user. GiST was realized in some prototype systems, e.g., Predator [66] and PostgreSQL [53]. Informix Dynamic Server with Universal Data Option (IDS/UDO) [51] simplified GiST's design while SP-GiST [91] extended GiST to include space-partitioning trees. A problem with GiST based approaches is that they require implementing the indexes completely following the coding conventions of the frameworks. This is still a challenging task because the index may have third-party ownership, or perhaps only binary code is available, or it is very complex to re-implement the code. It would be better if one could re-use an existing implementation of an

index without any code changes. This Thesis presents an extensible indexing framework to include new main-memory indexes without changing their implementations.

In order to utilize fully an index implementation it is very important that the query optimizer is aware of the presence of the index, its supported access methods, and how to process queries so that it is utilized in execution plans. The next section discusses this.

2.3 Query Processing

Query processing in general is first overviewed and it is then followed by a discussion of extensible query processing.

2.3.1 Overview

Figure 5 illustrates the steps of a database query processor.

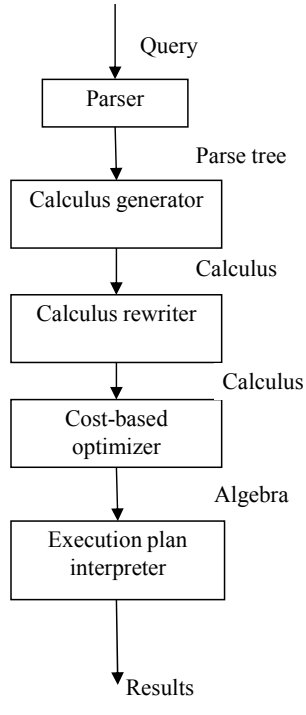

Figure 5 Query processing in a DBMS

The query processing consists of the five following steps:

1. A *parser* constructs a *parse tree* for the input query doing type checking and some semantic checks for the validity of the objects being referred in the query.
2. A *calculus generator* converts the parse tree into a predicate *calculus* representation, e.g. relational tuple calculus [26] [9] where variables are bound to tuples (rows) in tables, or alternatively domain calculus such as Datalog [26] where variables are bound to atomic values. Mexima uses the domain calculus ObjectLog [92], which is a dialect of Datalog allowing user-defined objects, types, overloading, and *foreign functions* that allow accessing external algorithms and data structures.
3. A *calculus rewriter* transforms the calculus expression into an equivalent expression to improve performance and enable further optimization. One very important rewrite is to expand views to expose indexed columns hidden inside views. Mexima enables calculus rewrite rules for transparent utilization of new indexes by user-provided index specific rewrite rules and algebraic inequality transformations. This contrasts with other extensible indexing frameworks that recommend manually reformulating queries involving complex expressions [61] [11].
4. A *cost-based optimizer* [28] applies some optimization algorithm on the transformed calculus expression to produces an *execution plan,* which is a program in *physical algebra* accessing the database. The optimizer 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. In Mexima, for given arguments, a function accessing a plugged-in index implementation can be associated with a *cost* and a *fanout*. The cost is an estimate of how expensive it is to retrieve the accessed tuples and the fanout estimates the expected number of emitted tuples in a results stream. If there is no specified cost, Mexima assumes a default cost and fanout based on the signature of the function, available index definitions, and some other heuristics.
5. Finally, the *execution plan interpreter* executes the plan and iteratively emits the result. In Mexima, special search functions supported by a plugged-in index are defined as foreign functions called from the execution plan.

2.3.2 Extensible query processing

In order to utilize a new index in queries, Oracle ODCIIndex framework [39] allows associating an index operator *op* with an index access path. Only conjunctive predicates where terms have the following forms are supported:

op(...) relop <value expression>, where *relop* is one of the relational operators: ≤, ≥, <,or >.

op(...) LIKE <value expression>

Oracle provides guidance [61] [11] on how to reformulate a query to utilize indexes when it is not exactly matching the above forms. In contrast, Mexima automatically transforms a wide range of query forms containing numerical expressions into queries that use index specific *special search functions (SSFs)* to utilize index properties.

In Starburst and DB2 [31], query transformations are driven by a rule engine. Internally, they represent queries by a *Query Graph Model (QGM)* in C++ structures. A rule table stores all rewrite rules and classifies them into different query classes. Each query class has different rewrite heuristics. The rule engine selects what rules to apply to transform the queries. Similarly, Volcano [24], Cascades [25], and Exodus [44] also use rules to transform relational algebra expression into physical operators.

In contrast, Mexima does not rely on procedural code since rewrite rules for SSFs are specified as declarative meta-data stored in *index property tables*. This is possible since the SSF rewriter is designed particularly for index utilization rather than for general relational algebra transformations as [31] [24] [25] [44]. Thus, a challenge is how to specify the rewrite rules as meta-data on a high-level. In Mexima each rewrite rule specified per index type describes a mapping from some terms of a *query fragment form* into an index-supported SSF function. Unlike Oracle, Mexima supports several query fragment forms. Altogether, the calculus rewriter in Mexima takes into account the existence of certain indexes, an extensible set of algebraic rules, and user-provided index rewrite rules to transform queries.

2.4 Database testing

It is critical that all functionalities of an index implementation are correct, meaning that query results and the database states after updates are the same, regardless of whether the index is used or not. Mexima includes a tester for automatic testing of the correctness of the functionalities provided by a plugged-in index implementation.

Testing of DBMS functionality in general has been studied in [23][45][59]. The database generator QAGen [23] provides general purpose testing of DBMS components. It generates test databases and test queries based on symbolic execution of queries. In [59] an inverse relational algebra generates query inputs for given query results. To implement unit testing for the query optimizer, the framework in [45] generates test queries based on user-defined transformation rules specified as trees of relational algebra operators. The JOB benchmark [88] tests the impact of a cost model and compares exhaustive dynamic programming with heuristic algorithms when enumerating sub-plans. It takes user-provided meta-data or DBMS statistics to generate dataset for given input queries.

QuEval [47] is a benchmark for testing spatial index implementations separate from a DBMS. QuEval produces test data sets based on user-provided parameters and built-in data generators. New index implementations can be developed and added to QuEval following its coding conventions.

Unlike QuEval, new complex indexes in C/C++ can be plugged into Mexima without any code changes. Furthermore, the Mexima tester generates and executes correctness tests of the plugged-in indexes, while the purpose of QuEval is to analyze performance of spatial index algorithms implementations in QuEval under different configurations.

2.5 Amos

Mexima extends the main-memory DBMS Amos [87]. Amos provides a functional and object oriented data model in which *objects, types,* and *functions* are the essential concepts. Functions are used to define properties, relationships, and computation over objects, which are classified by a type hierarchy stored in the database. The functional query language AmosQL supports queries in terms of functions over typed objects, where a *signature* and an *implementation* define a function. The signature declares the input and result parameter types and names, whereas the implementation defines how to compute outputs from inputs and vice versa. There are different kinds of functions. For example, a table is called a *stored* function and a *derived* function is a parameterized view defined by a single query. Furthermore, *foreign* functions can be defined in some external programming language, such as C/C++, Java, Python, or Lisp. Mexima uses the object-oriented data-model of Amos to represent index meta-data.

AmosQL queries are internally represented as ObjectLog [92], which is an extension of Datalog with objects, types, overloading, and foreign functions. A calculus rewriter transforms ObjectLog expressions to improve performance. After the rewrites, a cost-based optimizer produces an execution plan sent to the execution plan interpreter.

Mexima extends the query processor of Amos with calculus rewrite rules for transparent utilization of new indexes. It utilizes foreign functions to define SSFs to enable query transformation of queries into equivalent queries calling SSFs.

Amos represents all data objects internally as *physical objects* managed by a main-memory storage manager. Physical objects allocated inside a main-memory *database image* are persistent, which means that they can be saved on disk and later restored. A physical object, *po*, is accessed through an object *handle*, *hdl*, which is the offset to *po* from the start of the database image. Mexima provides a mechanism to access specialized external index storage managers for each index type so that index entries can be stored outside the database image.

3 Mexima

This chapter gives an overview of Mexima's architecture with references to the papers on which this Thesis is based. Followed by the general system architecture, the second section describes more in details the query processor, while the Mexima tester is described in the last sub-section.

3.1 Architecture

Figure 6 illustrates the overall architecture of Mexima. The system can used either as an extensible main-memory database or as a front-end query processor to a back-end relational DBMS, or a combination of the two.

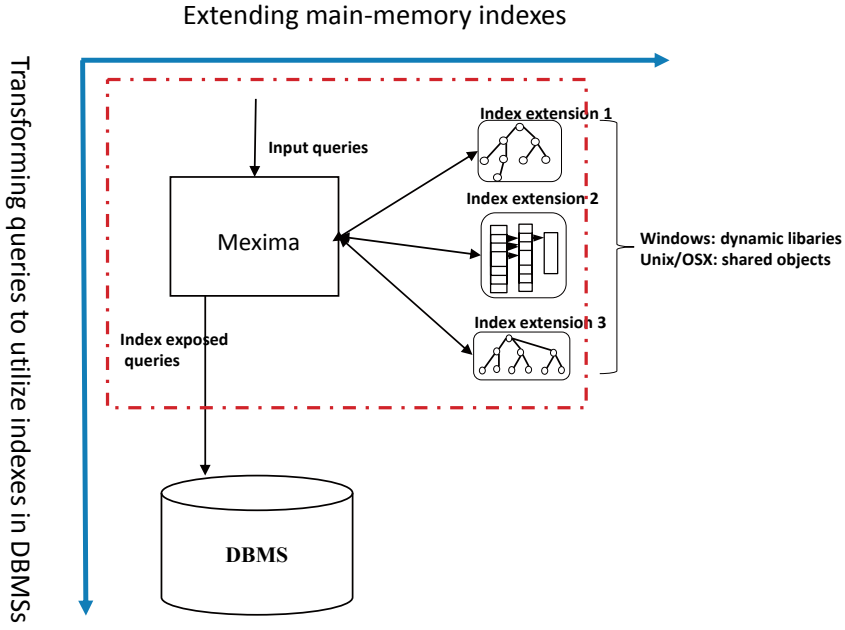

Figure 6 Mexima's architecture

Mexima enables plugging-in different kinds of main-memory index implementations in C or other programming languages without altering or re-engineering their original source code. An index extension developer can specify

meta-data about the plugged-in indexes. The meta-data can be index specific rewrite rules to hint the query processor on how to process queries to utilize indexes. The meta-data can also be data generators used to verify the correctness of the index. The details are in Paper I.

In addition, Mexima's query processor can transform complex queries over a back-end relational database so that indexes hidden inside complex expressions are exposed and utilized there, by applying algebraic transformation rules. In general, Mexima is a query processor with focus on index utilization. The details are mainly in Paper II and partly in Paper III.

Figure 7 illustrates the software layers of main-memory query processing in Mexima.

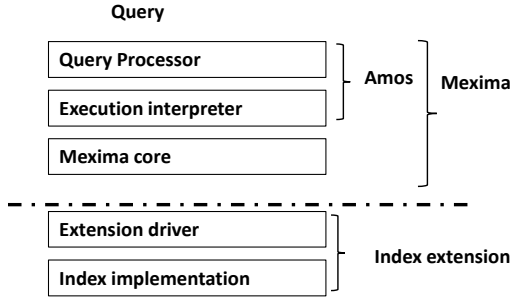

Figure 7 Mexima software layers

Queries are compiled by the query processor and executed by the execution interpreter. The execution plans call the *Mexima core* to execute the basic index operations (BAOs) such as *create(), drop(), put(), get()*, and *map()*, as well as special search functions (SSFs) for each kind of plugged-in index. The *extension driver* is a plugged-in interface between Mexima and the unchanged *index implementation*.

Figure 8 shows the details of the Mexima core component, with the Amos engine as the gray box. The *extension loader* loads at run-time the index extension as a dynamic library or a shared object. The extension loader registers the BAO index interfaces as C functions. The index name and its registered C functions are stored as meta-data in the *BAO table*.

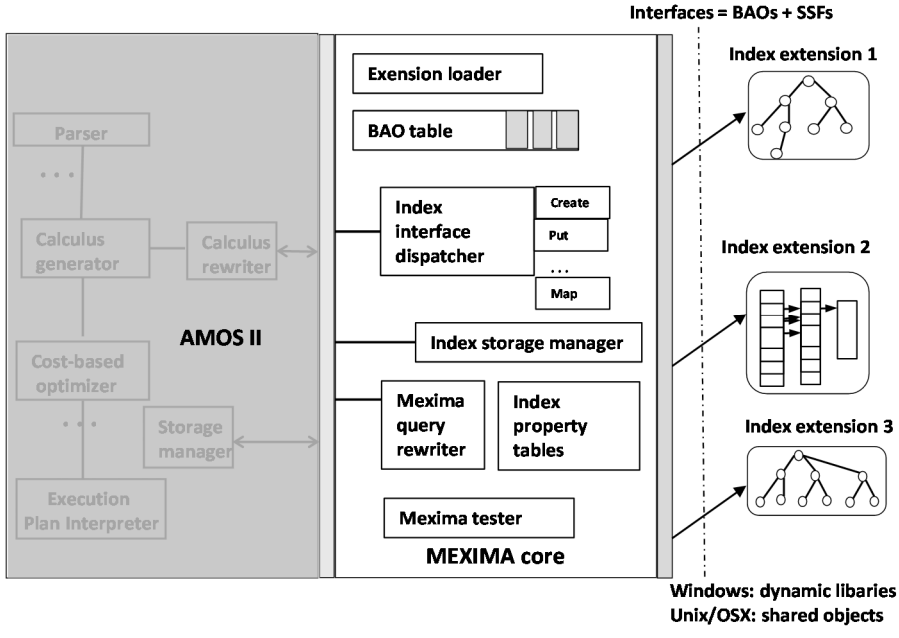

Figure 8 Mexima core

When an instance of a plugged-in index is associated with an attribute of a main-memory table, for every data update the *index interface dispatcher* accesses the index by invoking the corresponding registered BAOs (*create(), put(), get()*,etc.) in the BAO table. Importantly, the index interface dispatcher maintains reference counters of index keys and values. This frees the extension developer from handling garbage collection issues manually.

Mexima includes an *index storage manager* for saving the index structures on disk and reloading them later when the system starts. This is important for main-memory index implementations that do not support persistency. In addition, if an index implementation has persistency implemented, Mexima provides *save()* and *restore()* hooks to call index persistency functions registered in the BAO table. The details of Mexima's core are presented in Paper I.

The next section discusses the query processor of Mexima followed by the Mexima tester.

3.2 Query processor

Figure 9 illustrates Mexima's query processor. The calculus rewriter of Amos calls the *Mexima query rewriter* for transparent utilization of new indexes. The Mexima query rewriter applies rules to produce an *index exposed calculus*

expression, which is a calculus expression containing query fragments supported by an index. Without such rewrites, the query optimizer will not detect index access paths hidden inside complex expressions. The index extension developer populates the *index property tables* (*Figure 8*) containing *SSF translation rules*, which are index specific rewrite rules that describe how query fragments are translated to a new format that utilizes the index. Complex queries involving numerical expressions over indexed attributes are reformulated transparently so that the Mexima query rewriter can apply SSF translation rules to expose main-memory index implementation. The details are in Paper I and Paper II.

When the query plan is rewritten to expose a plugged-in index, the cost-based optimizer generates an execution plan that contains calls to the *Mexima core* to access the index. Paper I describes this in detail.

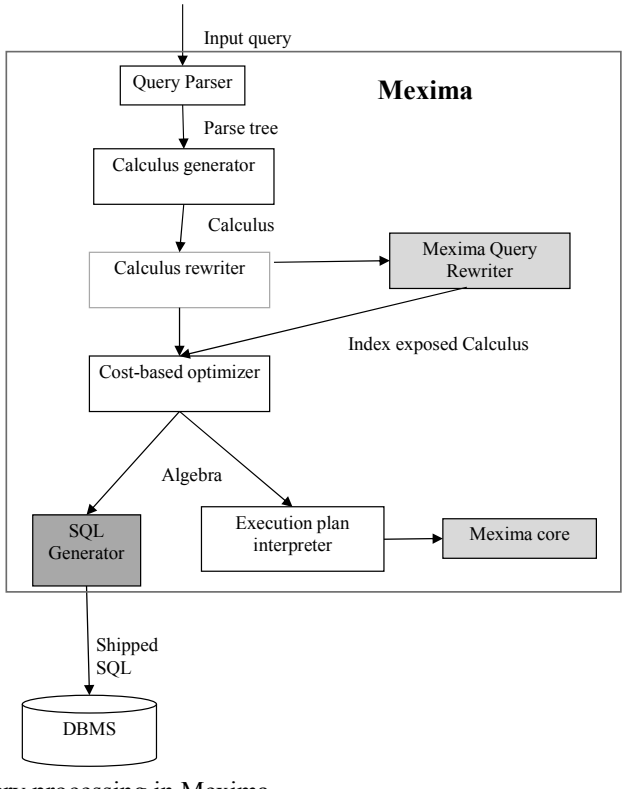

Figure 9 Query processing in Mexima

When Mexima is used as a query processor in front of a back-end DBMS, it enables transparent query transformation to exploit the presence of indexes in the backend DBMS. In this case, the *SQL Generator* (Paper III) translates the index exposed calculus expression into an equivalent *shipped SQL query* sent to the back-end DBSM through JDBC for optimization and evaluation.

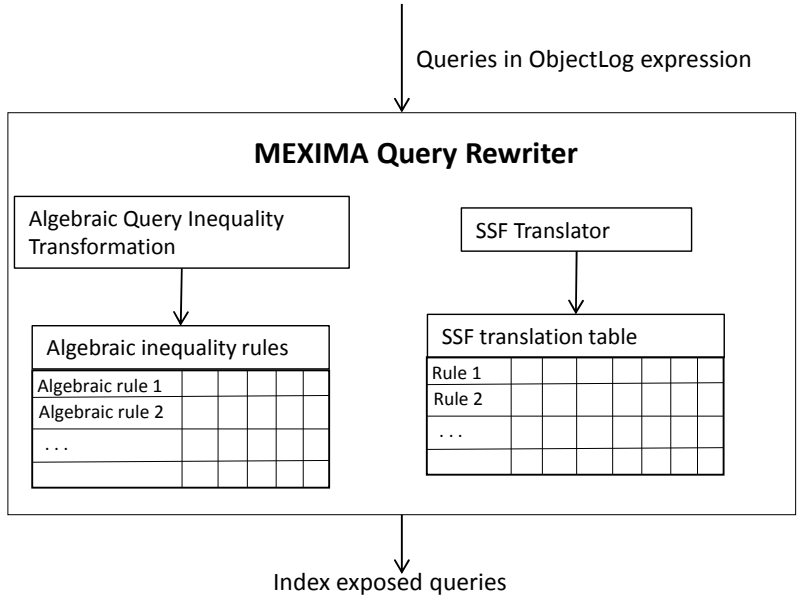

Figure 10 Mexima Query Rewriter

Figure 10 shows the Mexima query rewriter in details. For complex numerical expressions, the *Algebraic Inequality Query Transformations* (AQIT) component transforms a query into an equivalent one based on a set of *algebraic inequality rules*. AQIT transforms numerical complex inequality expressions so that query fragments supported by an index (B-trees in Paper II and high dimensional indexes in Paper I) are exposed.

The index extension developer populates the *SSF translation table* in which each row is an SSF translation rule for a particular index type. For plugged-in indexes, the *SSF translator* rewrites query fragments over indexed attributes into SSF calls (Paper I) based on these rules.

In summary, Mexima's query processor has the following features:

- SSF translation rules describe how query fragments are transformed to expose SSFs accessing plugged-in main-memory index implementations. Mexima currently supports six query fragment forms that can be transformed (Paper I).
- AQIT transformations enable transforming complex inequality expressions in queries to expose hidden indexes both for main-memory and back-end DBMS indexes (Paper I and Paper II).
- When data is stored in a back-end DBMS, queries having numerical expressions are translated to SQL queries sent to the back-end for execution (Paper III).

3.3 The Mexima tester

The *Mexima tester* illustrated by *Figure 11* automatically generates and runs test algorithms based on index meta-data provided by the index extension developer. The test algorithms require index specific random data generators. The system has some built-in random generators to generate keys as numbers and vectors of numbers respecting various distributions. User-defined data generators can easily be defined in terms of these built-in ones or as new kinds of data generators as queries. In addition, test keys stored in files can be declared as meta-data.

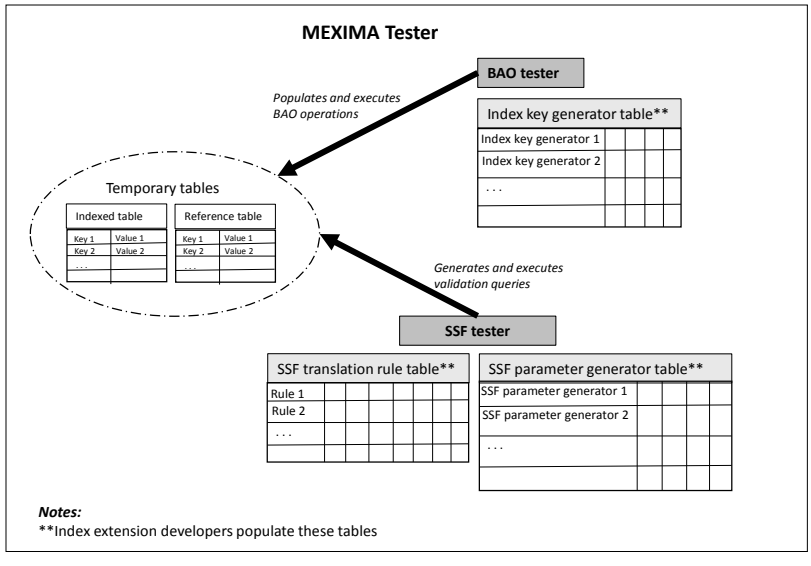

Figure 11 Mexima tester

To test the correctness of BAOs, the *BAO tester* generates two temporary tables per tested index implementation, an *indexed table*, and a *reference table*, where the indexed table has an index of the tested kind while the reference table has a hash index. The BAO tester populates these two tables by executing *index key generator* queries stored in an index meta-data table producing random index keys. The idea is that having the index or not should not change query results or table contents. In particular, randomly loading, accessing keys, mapping over, and deleting keys should produce the same results with or without the index.

For testing correctness of SSFs, the *SSF tester* generates and executes *validation queries*, which test the following:

- When an SSF is used,a query result has to be the same as with non-indexed naive scans.
- The SSF rewrite rules are correct.

The validation queries are generated based on SSF translation rules and *SSF parameter generators* specified as index meta-data. The Mexima tester validates that executing the same validation queries on the indexed and reference tables should return the same result when SSF translation rules are enabled or disabled.

The details about the Mexima tester are presented in Paper I.

4 Conclusions and Future Work

Table 1 positions this Thesis in comparison to existing state-of-the-art extensible indexing framework [36] [39] [51] [53] [91] regarding the following aspects of extensible indexing:

- Code reuse: None of them provides solutions to reuse existing index implementations without any code changes.
- Index utilization query rewrites: Oracle [39] provides limited support for rewriting queries to utilize new indexes without changing the DBMS core optimizer, while Mexima transparently transforms a large class of queries involving complex expressions to utilize plugged-in index implementations or indexes in a back-end DBMS.
- Index validation: Unlike Mexima, none of them has automated validation of plugged-in indexes.

Table 1. *Summary of extensible indexing framework*

		Requirements		
		Code reuse	Index utilization query rewrites	Index validation
Extensible indexing framework	GiST in PostgreSQL 9.3.5 [65]	No	Not in framework	No
	SP-GisT version 0.0.1 [79]	No	Not in framework	No
	Oracle [39]	No	Limited	No
	DB2 in DB2 Universal Database 7.1, [34][27]	No	Not in framework	No
	This Thesis	Yes	Yes	Yes

In summary, the Mexima framework allows transparent plugging-in of main-memory index implementations in a main-memory DBMS without code changes. The extension developer only writes a simple Mexima driver for the universal index operations (BAOs) and some index specific search functions (SSFs) that call the unchanged index implementation. Unmodified index implementations allow to easily utilizing highly optimized and complex index implementations such as Judy-tries [6]. For future work, more kinds of indexes should be plugged into Mexima than those evaluated so far B-trees [52], Linear-Hashing [52], Judy-Tries [6], X-trees [52], and R*-trees [20] (Paper I). This may add more requirements to the system.

To utilize plugged-in index implementations transparently in queries, the Mexima query processor uses SSF translation rules, and algebraic rewrite rules to rewrite queries. Future work should investigate how to extend the rewrite capabilities to support more algebraic rules and query fragment forms.

To validate the correctness of a new index, Mexima calls user-specified data generators, SSF translation rules, and query fragment forms that automate the correctness tests for a plugged-in index. As a future work, it should be investigated how to automate also performance tests of index operations.

Several experiments were made in Paper I: First, the penalty of calling an index implementation by plugging it into Mexima was compared with the stand-alone implementation showing that the overhead was less than one microsecond (μs) per index access. Furthermore, the experiments showed that rewrite rules provide substantial query performance improvements.

Moreover, when Mexima acts as a query processor in front of a DBMS, the experiments showed substantial query performance gains by Mexima's rewriting of queries to utilize indexes in the back-end DBMS.

For more future work, we shall investigate how extensible indexing and extensible query processing in Mexima can help to improve queries in NoSQL databases as discussed in Paper V. In addition, Mexima can be used to improve data access in distributed and parallel environments (Paper IV) to process massive stream in which each node is a Mexima node.

Altogether, Mexima is a complete and extensible platform for index integration, utilization, and evaluation.

5 Technical contributions

5.1 Paper I

T. Truong and T. Risch: Transparent inclusion, utilization, and validation of main-memory domain indexes, *27th International Conference on Scientific and Statistical Data-base Management (SSDBM)*, San Diego, United States, June 29-Juli 1, 2015.

Summary
In this paper, we presented the *Mexima (Main-memory External Index Manager)* system, which is an MMDB where new main-memory index structures can be plugged-in without modifying the index implementations or Mexima. To utilize plugged-in indexes in queries, the system transparently transforms query fragments into index operations based on user-provided *index property tables* containing index meta-data. The Mexima system includes a rule driven algebraic query transformation mechanism on complex numerical query expressions to expose potential utilization of a new index. To validate the correctness of an index implementation, Mexima generates and executes test queries based on general knowledge about indexing, the index meta-data, and the user-provided data generating queries. Several experiments were conducted to show that the index exposing rewrite mechanisms substantially improves performance and that the performance penalty of using an index plugged into Mexima is low compared to using the corresponding stand-alone C/C++ implementation. Finally, it is shown that the development effort of plugging in new indexes to Mexima is very small in comparison to other frameworks.

This paper partly answers Research Question one, two, and three.

Contributions
In 2011, Mexima allowed inclusion of new index structures and was used for research and education. Later in 2013, index utilization by transparently query rewrites was added whereas the index validation mechanism was designed and implemented during the autumn of 2014.

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

5.2 Paper II

T. Truong, T. Risch: Scalable Numerical Queries by Algebraic Inequality Transformations, *19th International Conference on Database Systems for Advanced Applications (DASFAA)*, Bali, Indonesia, April 21-24, 2014.

Summary

This paper is based on a real industrial application scenario where data streams derived from sensor readings are bulk-loaded into a relational database system [78]. The application was prototyped as the *Stream Log Analysis System (SLAS),* which enables historical analyses of logged data streams by SQL queries. These historical queries often contains complex numerical query inequalities e.g. to find suspected deviations from normal behavior of measured sensor values during a time-period. However, such queries are often slow to execute, because the query optimizer is unable to utilize ordered indexes on some attributes hidden inside complex numerical inequalities. In order to speed up the queries, they should be reformulated so that the indexes become exposed. Therefore, we introduced the query transformation algorithm *AQIT (Algebraic Query Inequality Transformation)* that automatically transforms SQL queries involving a class of algebraic inequalities into more scalable SQL queries utilizing ordered indexes.

AQIT was originally implemented as part of query rewriter in Mexima. AQIT is used both when plugging in new main-memory indexes and when transforming queries having complex numerical expressions to be sent to a back-end DBMS. The experimental results show that the queries execute substantially faster by a commercial DBMS when AQIT has been applied to preprocess them.

This paper partly answers Research Questions two and four.

Contributions

In 2011, I prototyped the first algorithm of AQIT that was part of the query rewriter in Mexima. Later, I wanted to investigate Mexima, in particular AQIT, in a real industrial application described in the paper. Thus, in 2012, I developed the SLAS system where Mexima acts as a query processor (also known as AQIT query processor) with a relational database back-end. The paper was written in the autumn of 2012, later accepted in the beginning of 2013.

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

5.3 Paper III

M.Zhu, S.Stefanova, T.Truong, and T.Risch: Scalable Nu-merical SPARQL Queries over Relational Databases, *4th international workshop on linked web data management (LWDM 2014)*, Athens, Greece, March 28, 2014.

Summary

In this paper, we investigated the problem of detecting past machine anomalies by querying historical sensor readings stored in a relational database. In this scenario, the main-memory database Mexima acts as query processor in front of the relational database. It takes anomaly detection queries containing numerical expressions, or inequality conditions, or string matching and produces equivalent SQL queries sent to the back-end database.

To enable scalable execution of such queries the numerical expressions should be translated into SQL rather than being post-processed in Mexima outside of the relational database. This is to avoid post-processing large data volumes, which must be transported back from the relational database server to Mexima. Furthermore, if the numerical expressions are post-processed, the indexes on the back-end database have no impact.

The paper presents the *NUMTranslator* algorithm, which translates numerical and other domain calculus operators into corresponding SQL expressions. The experiments showed that NUMTranslator substantially improves the query performance when the numerical expressions are highly selective.

This paper partly answers research question four.

Contributions

The NUMTranslator algorithm was first developed to enable scalable query execution of complex Mexima queries sent to a back-end relational database. Later, the NUMTranslator evolved to a part of a bigger system to harvest log databases [50]. The paper was written in autumn 2012, and then accepted in beginning of 2013.

I am one of the co-authors of the paper. In particular, I programmed the first limited version of the NUMTranslator, which later was fully developed as part of the FLOQ system [50]. I helped in paper writing and in data preparation for the experiments.

5.4 Paper IV – an application

Paper IV is an example application of Mexima.

S.Badiozamany, L.Melander, T.Truong, C.Xu, and T.Risch: Grand Challenge: Implementation by Frequently Emitting Parallel Windows and User-Defined Aggregate Functions, *Proc. The 7th ACM International Conference*

on Distributed Event-Based Systems, DEBS 2013, Arlington, Texas, USA, June 29 - July 3, 2013.

Summary
The paper describes an approach to monitor a soccer game that requires processing large volumes of data in real-time and delivers continuously physical summaries of the game as it is playing. The approach is based on an extensible DSMS in which high-volume data streams can be split and reduced into lower volume parallel streams by user-provided queries. Thus, expensive queries can be run in a parallel and distributed environment, in which each node is a main-memory Mexima database.

We experimented with plugging-in different indexes for indexing stream elements in a window. The application tested Mexima's performance and showed that Mexima can be used in a parallel and distributed environment.

Contributions
I am a co-author of the paper. I helped in prototyping the system and strategies.

5.5 Paper V – future development

Paper V will put requirements to the future work.

K.Mahmood, T.Truong, and T.Risch: NoSQL Approach to Large Scale Analysis of Persisted Streams, *30th British International Conference on Databases, Edinburgh (BICOD)*, Scotland, July 6-8, 2015.

Summary
In this paper, we first addressed some challenges in large scale persisting and analysis of numerical streaming logs. In order to investigate further these challenges, we propose to develop a benchmark that compares NoSQL stores with relational databases in storing and analyzing numerical logs. The benchmark is designed to serve as a base system for investigating query processing and indexing of large-scale numerical logs, in particular, how to reuse advanced indexing and query processing techniques in a scenario in which a main-memory is front-end while NoSQL stores data in the backend.

Contributions
I am a co-author of the paper.

Summary in Swedish

Primärminnesdatabaser (PDBSer) [12] [30] [46] används för ett växande antal applikationer som kräver snabb dataåtkomst, lagring och manipulation, exempelvis applikationer för finansiella analyser, realtidsoperativsystem, sensorsystem i industriella maskiner och mer allmänt i många tekniska och vetenskapliga tillämpningar. Framväxten av denna typ av databastillämpningar ställer nya krav på databashanteringssystemen för att stödja nya sorters data, såsom färghistogram, texturer, bildmönster, gensekvenser, sensorsekvenser, eller tidsserier. För att effektivt komma åt och manipulera sådana domändata måste ett databashanteringssystem inkludera nya typer av domänorienterade indexstrukturer.

Trots att en hel del domänorienterade indexstrukturer utvecklats har mycket få av dem använts i praktiken, de flesta system [29] [32] använder bara B-träd och hash-index. Anledningen är att det är mycket svårt att utvidga ett databashanteringssystem med nya indexstrukturer vilket normalt kräver omfattande ändringar i dess kärna. Den manuella insatsen för att göra en sådan integration är mycket kostsam och tidskrävande eftersom det nya indexet måste samverka med de flesta andra delkomponenter i kärnan. Därför skulle det vara önskvärt att man kunde göra databashanterarens indexering utbyggbar så att man kan implementera och lägga till nya indexstrukturer utan att ändra i kärnan.

Befintliga ramverk för utbyggbar indexering [36] [51] [53] [91] har stöd för att lägga till nya index, men de kräver dock omkodning av index-algoritmerna där man strikt följer varje enskilt ramverks kodningskonventioner och programmeringsgränssnitt. Detta kan vara en svår uppgift då eventuellt bara binärkod är tillgänglig, det kan vara mycket komplicerat att förändra koden eller äganderätten till indexet kan vara beroende av tredje part. Det skulle således vara mycket önskvärt att kunna lägga till nya indeximplementationer utan kodändringar.

När man lägger till en ny indeximplementation i en databashanterare är det vidare viktigt att testa att alla indexfunktioner fungerar korrekt. Korrekthet innebär här att alla funktioner ska returnera exakt förväntade resultat och lämna databasen i ett konsistent tillstånd efter uppdateringar eller databasladdningar. Det är även önskvärt att kunna automatisera testförfarandet för varje tillagd indeximplementering.

Att lägga till ett nytt index i en databashanterare kräver vidare att dess frågehanterare har kunskap om egenskaperna för det nya indexet, såsom dess

olika sätt att söka data och hur databasfrågor kan transformeras för att frågeprocessorn skall kunna använda indexet effektivt. Komplexa uttryck i frågor, t.ex. för exempelvis avancerad analys, hindrar ofta frågeprocessorn från att utnyttja indexet. Det är alltså önskvärt med utbyggbar transformering av databasfrågor så att indexspecifika omskrivningsregler kan kopplas in för att förbättra frågeprestanda. Frågeprocessorn behöver dessutom hantera omskrivningsregler för att förbättra prestanda för avancerade frågor som skickas för exekvering till en extern vanlig databashanterare för exekvering.

Denna avhandling behandlar ovanstående utmaningar av utbyggbar indexering, index testning och indexspecifika frågetransformationer i ett PDBS.

Följande frågeställningar undersöks:

1. Den övergripande forskningsfrågan är: Hur kan ett utbyggbart indexeringssystem utformas för att möjliggöra transparent inkludering av olika indeximplementationer utan kodändringar i vare sig databassystemet kärna eller i indeximplementationen
2. Hur kan frågeprocessorn förses med kunskap om ett nytt inkopplat index för att transparent kunna använda indexet i databasfrågor? I synnerhet:
   a. Hur kan frågeprocessorn göras utbyggbar med nya indexspecifika omskrivningsregler så att ett nytt inkopplat index transparent kan tillämpas i frågor?
3. Hur ska korrektheten av ett inkopplat index automatiskt valideras? I synnerhet, vilka är funktionerna att testa och hur kan testerna automatiseras?
4. När data lagrats i en extern databas, hur kan frågeprocessorn omvandla komplexa frågor så att indexen i den externa databasen kan utnyttjas och tillämpas där?

För att besvara dessa frågor har vi utvecklat ett utbyggbart PDBS som kallas Mexima (Main-memory Extern Index Manager).

För att besvara frågeställning 1 möjliggör Mexima inkoppling av olika typer av primärminnesindex utan att ändra deras ursprungliga källkoder (Paper I). En utvecklare tar en existerande implementation av ett index, skriver gränssnittkod och kompilerar hela modulen som ett dynamiskt bibliotek. Denna inkludering kräver lite utvecklingsarbete och ingen kunskap om databashanteringssystemets kärna.

För att besvara frågeställning 2 kan Mexima anropa relevanta indexoperationer för en given databasfråga (Paper I). De indexoperationer som finns för alla typer av index kallas grundläggande accessoperationer (BAOs). De är metoder för att skapa, hämta, uppdatera och traversera indexstrukturens element. Dessutom finns det ofta index-specifika sökfunktioner (SSFs) som utnyttjar speciella egenskaper hos en indexstruktur för effektiv sökning, exempelvis intervallsökning för B-träd [62] och områdessökning för R-träd [1] och and X-träd [5]. För att frågeprocessorn skall kunna tillämpa index-specifika sökfunktioner innehåller Mexima ett system för omskrivning (transformation) av

databasfrågor för att identifiera mönster i frågorna där index-specifika sökfunktioner kan användas.

När databasfrågor innehåller komplexa uttryck kan de hindra frågeprocessorn från att utnyttja förekomsten av index. Detta orsakar dyra genomsökningar av hela tabeller snarare än direkta indexanrop. AQIT omvandlar komplexa numeriska frågor till motsvarande frågor som är mer effektiva genom att exponera dolda index (Paper II).

För att besvara frågeställning 3 genererar Mexima för varje indextyp ett antal testfrågor som automatiskt testar korrektheten av indexets implementering. Testmodulen drivs av datageneratorer och beskrivningar av indexets egenskaper (Paper I).

För att besvara forskningsfråga 4 tillåter Mexima att komplexa frågor till en relationsdatabas först transformeras så att dolda index exponeras (Paper II). Samma regler som används för att transformera databasfrågor mot ett primärminnesindex används också för att exponera index i externa databaser. En skalbar mekanism för att hantera omskrivna numeriska frågor som skickas till en extern relationsdatabas kräver vidare generering av SQL-frågor för att representera numeriska uttryck (Paper III).

Bibliography

[1] A. Guttman: R-trees: A dynamic index structure for spatial searching, *Proc. SIGMOD Conf.*, pp 47–57, 1984.

[2] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. In *Proceedings of the Very Large Data Bases Endowment* (PVLDB), 5(12):1790–1801, 2012.

[3] Aravind Menon. 2012. Big data @ facebook. *In Proceedings of the 2012 workshop on Management of big data systems (MBDS '12)*. ACM, New York, NY, USA, 31-32.
DOI=http://dx.doi.org/10.1145/2378356.2378364

[4] B. Fitzpatrick and A. Vorobey. Memcached: a distributed memory object caching system. 2003. [Online]. Available: http://memcached.org/.

[5] B. Stefan, A.K. Daniel, H-P. Kriegel: The X-tree : An Index Structure for High-Dimensional Data, *Proc. 22nd of Very Large Databases Conference*., Bombay, India, pp. 28-39, 1996.

[6] Baskins, D. Judy home page. http://judy.sourceforge.net/ (Accessed at 2003).

[7] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. 1994. Efficient and effective querying by image content. *J. Intell. Inf. Syst. 3*, 3-4 (July 1994), 231-262.
DOI=http://dx.doi.org/10.1007/BF00962238

[8] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. 1994. Fast subsequence matching in time-series databases. *Proc. of the 1994 ACM SIGMOD international conference on Management of data (SIGMOD '94),* Richard Thomas Snodgrass and Marianne Winslett (Eds.). ACM, New York, NY, USA, 419-429.
DOI=http://dx.doi.org/10.1145/191839.191925

[9] Codd, E. F.: Relational completeness of database sublanguages. *J. Communications of the ACM. 13(6)*, pp 377-387, 1970

[10] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. *In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1243-1254.
DOI=http://dx.doi.org/10.1145/2463676.2463710

[11] D. Benoit, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin: Automatic SQL tuning in Oracle 10g, *Proc. of Thirtieth international conference on Very large data bases-Volume 30*, pp 1098-1109, 2004.

[12] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. 1984. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of data*.

[13] D.J-H. Hwang. Function-Based Indexing for Object-Oriented Databases, *PhD Thesis, Massachusetts Institute of Technology*, 1994, 26-32.
[14] Daniel Abadi, Peter Boncz, and Stavros Harizopoulos. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Now Publishers Inc*., Hanover, MA, USA
[15] David B. Lomet, Sudipta Sengupta, and Justin J. Levandoski. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013*) (ICDE '13). IEEE Computer Society, Washington, DC, USA, 302-313. DOI=http://dx.doi.org/10.1109/ICDE.2013.6544834
[16] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.
[17] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv*. 11, 2 (June 1979), 121-137. DOI=http://dx.doi.org/10.1145/356770.356776.
[18] E. F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (June 1970), 377-387. DOI=http://dx.doi.org/10.1145/362384.362685
[19] E. G. Coffman, Jr. and J. Eve. 1970. File structures using hashing functions. *Commun. ACM 13*, 7 (July 1970), 427-432 DOI=http://dx.doi.org/10.1145/362686.362693
[20] Efficient and Lightweight In-Memory Implementation of R*-Tree: *http://www.ics.uci.edu/~salsubai/rstartree.html*.
[21] Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: applications to image and text data. *Proc. of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '01)*. ACM, New York, NY, USA, 245-250. DOI=http://dx.doi.org/10.1145/502512.502546
[22] F. Färber, N. May, W. Lehner, P. Große, I. Mller, H. Rauhe, and J. Dees, The sap hana database – an architecture overview. *IEEE Data Eng. Bull., 2012*.
[23] F. Haftmann, D. Kossmann and E. Lo: A framework for efficient regression tests on database applications, *The Very large data bases Journal, 16(1)*, pp. 145-164, 2007
[24] G. Goetz, W. J. McKenna: The Volcano optimizer generator: Extensibility and efficient search, *Proc. of IEEE Conference on Data Engineering*. pp. 209-218, 1993.
[25] G. Goetz: The cascades framework for query optimization, *IEEE Data Engineering Bulletin. 18(3)*, pp 19-29, 1995.
[26] Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall, Upper Saddle River, NJ, USA (2009).
[27] George Baklarz and Bill Wong. 2000. *The Db2 Universal Database V7.1 for Unix, Linux, Windows, and Os/2 with CD-ROM (4th ed.)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.
[28] Graefe, G.: Query evaluation techniques for large databases. *J. ACM Computing Surveys (CSUR)*. 25(2):73-169, 1993.
[29] H Zhang, G Chen, BC Ooi, KL Tan, M Zhang: "In-memory big data management and processing: A survey", *IEEE*, 2015
[30] H. Garcia-Molina and K. Salem. 1992. Main Memory Database Systems: An Overview. *IEEE Trans. on Knowl. and Data Eng. 4*, 6 (December 1992), 509-516. DOI=http://dx.doi.org/10.1109/69.180602

[31] H. Pirahesh, T.C. Leung, & W. Hasan: A rule engine for query transformation in Starburst and IBM DB2 C/S DBMS. *Proc. of IEEE Conference on Data Engineering*, pp. 391-400, 1997.
[32] Hans-Peter Kriegel and Martin Pfeifle and Marco Pötke and Thomas Seidl, The Paradigm of Relational Indexing: A Survey, The Paradigm of Relational Indexing: A Survey, 2003, *In BTW, volume 26 of LNI. GI*, pp 285—304, Springer.
[33] Henrich A., Six H.-W., Widmayer P.: 'The LSD-Tree: Spatial Access to Multidimensional Point and Non-Point Objects', *Proc. 15th Conf. on Very Large Data Bases Conference*, Amsterdam, The Netherlands, pp. 45-53, 1989.
[34] http://www.ibm.com/developerworks/data/library/techarticle/dm-0312stolze/, Accessed January, 2016.
[35] IBM DB2 Spatial Extender User's Guide and Reference, Version 7 (2001).
[36] J Hellerstein. M., J. F. Naughton, and A. Pfeffer: Generalized search trees for database systems*, Proc. of The Very large data bases Conference.*, pp 562-573, 1995.
[37] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar et al., Efficient implementation of sorting on multi-core simd cpu architecture, in *PVLDB '08*, 2008.
[38] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazi` eres, S. Mitra, A. Narayanan et al., The case for ramclouds: Scalable high-performance storage entirely in dram, OSR, 2010.
[39] J. Srinivasan, R. Murthy, S. Sundara, N. Agarwal, and S. DeFazio: Extensible indexing: a framework for integrating domain-specific indexing schemes into oracle8i. *Proc. of IEEE Conference on Data Engineering.*, pp 91–100, 2000.
[40] J.Gray, A. Szalay, and G. Fekete. Using Table Valued Functions in SQL Server 2005 to Implement a Spatial Data Library, *Technical Report, Microsoft Research Advanced Technology Division 2005*.
[41] K.Mahmood, T.Truong, and T.Risch: NoSQL Approach to Large Scale Analysis of Persisted Streams, *30th British International Conference on Databases, Edinburgh (BICOD)*, Scotland, July 6-8, 2015.
[42] King Ip Lin, H. V. Jagadish, and Christos Faloutsos. 1994. The TV-tree: an index structure for high-dimensional data. *The VLDB Journal 3*, 4 (October 1994), pp 517-542.
[43] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *ICDMW '10*, 2010.
[44] M. Carey, et al: The architecture of the EXODUS extensible DBMS, *Proc. 1986 international workshop on Object-oriented database systems, IEEE Computer Society Press,* 1986.
[45] M. Elhamali and L. Giakoumakis: Unit-testing Query Transformation Rules, *Proc. of 1st International Workshop on Testing Database Systems*, 2008
[46] M. H. Eich, "Mars: The design of a main memory database machine," in *Database Machines and Knowledge Base Machines, ser. The Kluwer International Series in Engineering and Computer Science. Springer US*, 1988.
[47] M. Schäler, A. Grebhahn, R. Schröter, S. Schulze, V. Köppen, and G. Saake. 2013: QuEval: beyond high-dimensional indexing à la carte, *Proc. of the Very large data bases Endowment, 6(14)*, pp 1654-1665, 2013.
[48] M. Stonebraker and A. Weisberg, "The voltdb main memory dbms," *IEEE Data Eng. Bull.*, 2013.
[49] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: dynamic bandwidth sharing in a DBMS. In *Proceedings of the International Conference on Very Large Data Bases* (VLDB), pages 723–734, 2007.

[50] M.Zhu, S.Stefanova, T.Truong, and T.Risch: Scalable Numerical SPARQL Queries over Relational Databases, *4th international workshop on linked web data management (LWDM 2014)*, Athens, Greece, March 28, 2014.
[51] Marcel Kornacker. 1999. High-Performance Extensible Indexing. *Proc. of the 25th International Conference on Very Large Data Bases (VLDB '99)*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 699-708.
[52] Mexima homepage, http://www.it.uu.se/research/group/udbl/mexima
[53] Michael Stonebraker and Greg Kemnitz. 1991. The POSTGRES next generation database management system. *Commun. ACM 34*, 10 (October 1991), 78-92. DOI=http://dx.doi.org/10.1145/125223.125262
[54] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel R. Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alexander Rasin, Nga Tran, and Stan B. Zdonik. C-Store: A Column-Oriented DBMS. *In Proceedings of the International Conference on Very Large Data Bases* (VLDB), pages 553–564, 2005.
[55] Michael Stonebraker, Dorothy Moore, and Paul Brown. 1998. Object-Relational Dbmss: Tracking the Next Great Wave (2nd ed.). *Morgan Kaufmann Publishers Inc*., San Francisco, CA, USA.
[56] Mikael Ronström: Design and Modelling of a Parallel Data Server for Telecom Applications, PhD Thesis, Linköping University Dissertation No 520, 1998
[57] MongoDB Inc., "Mongodb," 2009. [Online]. Available:http://www.mongodb.org/
[58] MSQL, The MEMORY storage engine. 2016. [Online]. Available: http://www.mysql.com/
[59] N. Bruno, S. Chaudhuri and D. Thomas: Generating Queries with Cardinality Constraints for DBMS Testing, *IEEE Transactions on Knowledge and Data Engineering, 18(12)*, pp 1721-1725, 2006.
[60] Navathe, Ramez Elmasri, Shamkant B. (2010). Fundamentals of database systems (6th ed.). *Upper Saddle River, N.J.: Pearson Education*. pp. 652–660.
[61] Oracle Inc: Query Optimization in Oracle Database 10g Release 2. *http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-general-query-optimization-10gr-130948.pdf*, 2005.
[62] Oracle, "Mysql cluster," 2016. [Online]. Available: http://www.mysql.com/
[63] Per-Ake Larson. 1988. Linear hashing with separators—a dynamic hashing scheme achieving one-access. *ACM Trans. Database Syst. 13*, 3 (September 1988), 366-388. DOI=http://dx.doi.org/10.1145/44498.44500
[64] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper- pipelining query execution. In *Proceedings of the biennial Conference on Innovative Data Systems Research* (CIDR), 2005.
[65] PostgreSQL version 9.3.5 http://www.postgresql.org/ftp/source/v9.3.5/
[66] Praveen Seshadri. 1998. PREDATOR: a resource for database research. *SIGMOD Rec. 27*, 1 (March 1998), 16-20. DOI=http://dx.doi.org/10.1145/273244.273251

[67] R. Bayer and E. McCreight. 1970. Organization and maintenance of large ordered indices. *In Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control (SIGFIDET '70)*. ACM, New York, NY, USA, 107-141. DOI=http://dx.doi.org/10.1145/1734663.1734671
[68] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden et al., "H-store: A high-performance, distributed main memory transaction processing system," in *PVLDB '08*, 2008
[69] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in *OSDI '10*, 2010
[70] Rakesh Agrawal, Christos Faloutsos, and Arun N. Swami. 1993. Efficient Similarity Search In Sequence Databases. *Proc. of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO '93),* David B. Lomet (Ed.). Springer-Verlag, London, UK, 69-84, 1993.
[71] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov. 2002. Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data (SIGMOD '02)*. ACM, New York, NY, USA, 546-557. DOI=http://dx.doi.org/10.1145/564691.564755
[72] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S. McKinley. 2013. Taking off the gloves with reference counting Immix. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications (OOPSLA '13*). ACM, New York, NY, USA, 93-110. DOI=http://dx.doi.org/10.1145/2509136.2509527
[73] Roger MacNicol and Blaine French. Sybase IQ multiplex - designed for analytics. In *Proceedings of the International Conference on Very Large Data Bases* (VLDB), pages 1227–1230, 2004.
[74] S. Sanfilippo and P. Noordhuis, "Redis," 2009. [Online]. Available: http://redis. io
[75] S.Badiozamany, L.Melander, T.Truong, C.Xu, and T.Risch: Grand Challenge: Implementation by Frequently Emitting Parallel Windows and User-Defined Aggregate Functions, *Proc. The 7th ACM International Conference on Distributed Event-Based Systems*, DEBS 2013, Arlington, Texas, USA, June 29 - July 3, 2013.
[76] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst. 30*, 1 (March 2005), 122-173. DOI=http://dx.doi.org/10.1145/1061318.1061322
[77] SAP, SAP HANA, 2010. [Online]. Available: http://www.saphana.com/
[78] Smart Vortex Project - http://www.smartvortex.eu/
[79] SP-GiST*: https://www.cs.purdue.edu/spgist/*
[80] Stratos Idreos, Fabian Gro!en, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin L Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.
[81] StratosIdreos.DatabaseCracking:Towards Auto-tuning Database Kernels. CWI, PhD Thesis, 2010.
[82] T. Lahiri, M. A. Neimat, and S. Folkman. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Engineering Bulletin 36(2): 6--13 (2013)*.
[83] T. Muhlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann, Instant loading for main memory databases, in *PVLDB'13*, 2013.

[84] T. Truong and T. Risch: Transparent inclusion, utilization, and validation of main memory domain indexes, *27th International Conference on Scientific and Statistical Database Management (SSDBM)*, San Diego, United States, June 29-Juli 1, 2015.
[85] T. Truong, T. Risch: Scalable Numerical Queries by Algebraic Inequality Transformations, *19th International Conference on Database Systems for Advanced Applications (DASFAA)*, Bali, Indonesia, April 21-24, 2014.
[86] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner, Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units, in *PVLDB '09*, 2009.
[87] T.Risch, V.Josifovski, and T.Katchaounov: Functional Data Integration in a Distributed Mediator System, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, *Springer, ISBN 3-540-00375-4*,2004.
[88] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? In *Proc. VLDB Endow. 9, 3* (November 2015), 204-215. DOI=http://dx.doi.org/10.14778/2850583.2850594
[89] Vishal Sikka, Franz Färber, Anil Goel, and Wolfgang Lehner. 2013. SAP HANA: the evolution from a modern main-memory data platform to an enterprise application platform. *Proc. VLDB Endow. 6, 11 (August 2013)*, 1184-1185. DOI=http://dx.doi.org/10.14778/2536222.2536251
[90] Volker Gaede and Oliver Günther. 1998. Multidimensional access methods. *ACM Comput. Surv. 30*, 2 (June 1998), 170-231. DOI=10.1145/280277.280279
[91] W. G. Aref and I. F. Ilyas: An extensible index for spatial databases, *Proc. of Statistical and Scientific Database Management*, pp 49–58, 2001.
[92] W. Litwin and T.Risch: Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering, 4(6)*, 1992.
[93] Witold Litwin. 1988. Linear Hashing: a new tool for file and table addressing.. *In Readings in database systems*, Michael Stonebraker (Ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA 570-581.
[94] Yi Fang, Marc Friedman, Giri Nair, Michael Rys, and Ana-Elisa Schmid. 2008. Spatial indexing in Microsoft SQL server 2008. *Proc. of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08)*. ACM, New York, NY, USA, 1207-1216. DOI=http://dx.doi.org/10.1145/1376616.1376737.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1352

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-280374

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2016

Paper I

Paper I

Thanh Truong and Tore Risch. 2015. Transparent inclusion, utilization, and validation of main memory domain indexes. In Proceedings of the 27th International Conference on Scientific and Statistical Database Management (SSDBM '15). ACM, New York, NY, USA, DOI=http://dx.doi.org/10.1145/2791347.2791375

Transparent inclusion, utilization, and validation of main memory domain indexes

Thanh Truong, Tore Risch
Department of Information Technology
Box 337, SE-751 05, Sweden
Uppsala University, Sweden
{thanh.truong,tore.risch}@it.uu.se

Abstract- Main-memory database systems (MMDBs) are viable solutions for many scientific applications. Scientific and engineering data often require special indexing methods, and there is a large number of domain specific main memory indexing implementations developed. However, adding an index structure into a database system can be challenging. *Mexima (Main-memory External Index Manager)* provides an MMDB where new main-memory index structures can be plugged-in without modifying the index implementations. This has allowed to plug-into Mexima complex and highly optimized index structures implemented in C/C++ without code changes. To utilize new user-defined indexes in queries transparently, Mexima automatically transforms query fragments into index operations based on *index property tables* containing index meta-data. For scalable processing of complex numerical query expressions, Mexima includes an algebraic query transformation mechanism that reasons on numerical expressions to expose potential utilization of indexes. The index property tables furthermore enable validating the correctness of an index implementation by executing automatically generated test queries based on index meta-data. Experiments show that the performance penalty of using an index plugged into Mexima is low compared to using the corresponding stand-alone C/C++ implementation. Substantial performance gains are shown by the index exposing rewrite mechanisms.

Keywords
Domain Indexing, Extensible Databases, Query Processing, Automatic Testing.

1 Introduction

Indexing is a key factor for scalable database query processing. Most DBMSs support one or several indexing structures, such as B-trees and hashing. It is

well recognized that many scientific applications involving, e.g., data mining, temporal queries, and spatial analyzes, require customized indexing to improve performance, which motivates the need for extensible indexing frameworks [1][16][26]. These frameworks allow implementing new indexing algorithms by strictly following framework specific coding conventions and primitives, which requires knowledge about DBMS internals. To include a new domain indexing structure into a DBMS can also be challenging because of third party ownership, having only binary code available, or simply being very challenging to re-engineer.

There are many domain-indexing algorithms developed for main-memory, for example, T-Trees [31], Cache Sensitive B+-Trees [34], Fast Architecture Sensitive Trees [32], and Adaptive Radix Trees [33]. The issue addressed in this paper is how to include a new main-memory domain indexing structure into a DBMS with minimal effort. The generalized extensible indexing framework *Mexima (Main-memory eXternal Index Manager)* enables plugging-in main-memory index implementations in an MMDB without changing their implementations.

When using Mexima the index extension developer needs not have knowledge about the DBMS internals, since there is a clean separation between the database kernel and a plugged-in domain index implementation. Only a simple interface that bridges Mexima with the untouched index implementation needs to be developed. Another important issue with domain indexing is how to extend the query processor so that the plugged-in index algorithms are utilized in a scalable and transparent way in queries. To utilize a new index without re-formulating queries, Mexima supports automatic query transformations based on user-provided *index property tables* populated by the index extension developer to specify meta-data about the index.

Basic access operators (BAOs) of an index are operators available for all kinds of indexes, i.e. methods for creating, dropping, updating, accessing, and mapping over indexed elements. In addition, each kind of index usually has *special search functions (SSFs)* to utilize index specific properties for efficient search, e.g., interval search on B-trees, and K-nearest neighbor and proximity search on R-trees and X-trees. To utilize SSFs transparently in queries the system must rewrite query conditions into calls to SSFs, for which Mexima allows the index extension developer to declare *SSF translation rules* that specify the rewrites.

For example, spatial proximity search can be expressed in queries using an *index sensitive function (ISF)*, such as *distance()*. The following query compares indexed color histograms with a given one. Here, *?* denotes query parameter:

```
SELECT name FROM Images i
WHERE distance(i.colorHistogram, ?) <= 0.11;
```

If there is a spatial index on *i.colorHistogram*, Mexima translates the query into an SSF call, rather than scanning all images to apply the ISF *distance()*.

If an indexed attribute is hidden inside expressions, the query processor cannot directly apply the SSF translation rules and fails to utilize the index. For example, in the following similarity query the index on *i.colorHistogram* is hidden inside a numerical expression, which prohibits a direct translation into an SSF call:

SELECT name FROM Images i
WHERE 1/ (distance(i.colorHistogram , ?) + 1 >= ?;

To expose indexes hidden inside numerical expressions Mexima transparently reformulates queries to call SSFs in order to utilize indexes in numerical query expressions.

An important aspect when plugging-in a new index implementation is to test that the index functionality is correct. Mexima has built-in automatic tests procedures for both BAOs and SSFs. Mexima utilizes index meta-data stored in the index property tables to generate test queries. This is a form of model-based testing [19] where a model of index properties stored in Mexima is used for automatically generating and executing test queries. For this, the index extension developer specifies as meta-data index-specific data generating queries expressed in terms of an extensible library of built-in data generating functions.

In summary, our contributions are:

1. The extensible indexing system, Mexima, allows inclusion of complex main-memory domain-specific index implementations in an MMDB without code changes. In addition, Mexima makes the plugged-in main-memory index data structures persistent.
2. In order to transparently utilize a new index in queries, the SSF translator rewrites query fragments over indexed attributes into SSF calls. The rewrites are driven by user populated index property tables containing SSF translation rules that describe the operations supported by the index.
3. Complex queries involving numerical expression over indexed attributes are automatically reformulated so that the SSF translator can rewrite them.
4. To validate correct functionality of a domain index, Mexima generates automatic test procedures driven by meta-data stored in the index property tables.
5. The experimental evaluation investigates the overhead of using main-memory index extensions in queries via Mexima compared

to directly executing hard-coded C/C++ implementations1. Furthermore, the substantial impact of the query rewrites is investigated.

The following main-memory index structures have been plugged-into Mexima: Main memory B-trees [30], Linear-Hashing [30], Judy-Tries [2], X-trees [30], and R*-trees [7].

The paper is organized as follows. Section 2 discusses related work. Section 3 defines some terminology. Section 4 presents the architecture of Mexima in detail. Section 5 presents queries used to illustrate Mexima's query processor in Section 5. Section 7 discusses Mexima's model-based test generators for both BAOs and SSFs. Section 8 shows our experimental results and evaluations. Finally, Section 9 concludes and outlines future work.

2 Related Work

Several index structures beyond B-trees and hash tables have been developed for domain-specific data, for example: R-trees [14], Quad-trees [10], KD-trees [23], and Tries [11]. Very few of them were implemented in DBMSs, even though the necessity of including new and domain-specific index structures as database indexes has been observed [1][16][26]. Some extensible indexing frameworks have been proposed for both commercial DBMSs and database research prototypes e.g, Oracle [27], Gist [16], and SP Gist [1]. Extensible indexing can be divided into three stages, as illustrated by *Figure 1*:

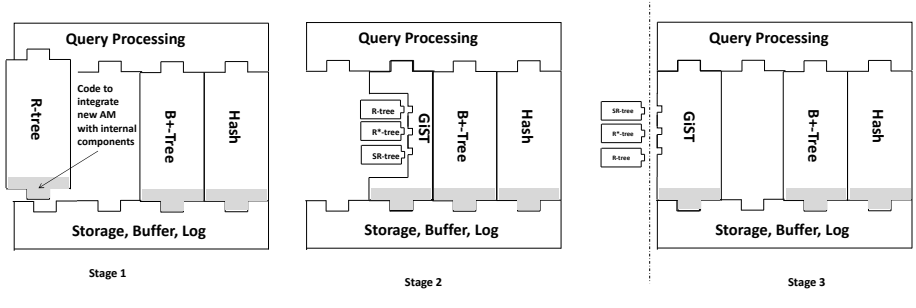

Figure 1. History of extensible indexing frameworks

Stage 1:

In DBMSs without support for extensible indexing all index structures have to be implemented and integrated with the DBMS kernel. This requires writing *access method (AM)* code and tightly integrating it with other components

[1] Even though MEXIMA supports Java as well, here we assume C/C++ as implementation languages.

in the kernel, such as the storage manager, the query optimizer, and the query executor.

Stage 2:
GiST (Generalized Search Trees) [16] is a template index structure for disk-based search trees, i.e., B-trees and R-tree-like indexes. GiST reduces the implementation effort by providing implementation code for commonly invariant properties of search trees and leaving other characteristics to be specified as user-defined index extensions. GiST itself is part of the DBMS kernel. The index extension developer writes extension code as user defined functions following GiST's conventions, without need to integrate the access method code with DBMS internals.

Stage 3:
To improve performance and simplify the index implementations, the GiST approach was generalized in IDS/UDO [17] and later in SP-GiST [1] to support spatial indexes. In IDS/UDO, the main idea is to redesign and separate the GiST implementation to reduce the number of calls to user-defined functions. Furthermore, unlike GiST, IDS/UDO and SP-GiST dynamically load the index implementation at runtime. The extended GiST system is divided into three sub-components [17]: *the GiST core*, *the access method extensions (AME)* for index-specific accesses, and the *data type adaptor (DTA)* for manipulating index keys. The GiST core is part of the DBMS kernel and provides interfaces to the AME for each new kind of index. The AME is written by the index extension developer following GiST's coding conventions. It interacts with the GiST core through a set of C interfaces and callback functions. The AME developer needs to supply 11 such callback functions. In addition, the developer must supply DTA code. SP-GiST *(Space Partitioning GiST)* is a framework for space-partitioning trees [1] supporting a wide range space partition algorithms.

Mexima:
While all Gist-based approaches require re-engineering the index code in terms of the Gist coding conventions, Mexima allows using existing main-memory index implementations or binary code without any code modifications. An index structure implemented by a third party without knowledge of DBMS kernel functionality can be integrated with the DBMS though Mexima by writing some simple interface code. For index implementations without support for persistence, Mexima provides transparent storage persistence. Thus, Mexima makes inclusion of main-memory index implementations possible with very limited implementation efforts.

Oracle's extensible indexing is an SQL-based framework for integrating domain-specific indexing schemes [26]. The index developer provides operations in C, C++, Java, or SQL/PSQL for index creation, index update, and

index-scans following the complex *Oracle Data Cartridge Interface (ODCIIndex)* interfaces and coding conventions [26]. By contrast, Mexima allows including new index implementations without changing any code.

While the approaches above address how to add index implementations to DBMS kernels, another critical issue is how to extend the query processor so that it can transparently utilize the new index structures without forcing users to reformulate queries. For example, in order to utilize a new index in queries, Oracle's ODCIIndex allows associating an ISF with an index access path [26]. Conjunctive predicates where terms have the following forms are supported:

isf(...) relop <value expression>, where *relop* is one of the relational operators: ≤, ≥, <,or >.

isf(...) LIKE <value expression>

Oracle provides guidance [3] [21] on how to reformulate a query to utilize indexes when it is not exactly matching the above forms.

Rather than manual query reformulations, Mexima transforms a wide range of query forms containing index sensitive functions and numerical expressions into queries that contain SSF calls utilizing domain index structures.

Starburst and DB2 [22] contains an internal rule engine for transformations of queries represented by a *Query Graph Model (QGM)* in C++ structures. Rewrite rules are stored in a rule table, and classified into different classes. Each class of rewrite rules has different rewrite heuristics. These rules rely heavily on a rich function library in C++ to exploit and manipulate queries representing QGMs. A rule engine is responsible for selecting rules to be executed along with controls how rules are fired. Similarly, Volcano [13], Cascades [12], and Exodus [5] also use rules to transform relational algebra expression into physical operators.

Rather than procedural code, in Mexima the SSF rewrites are specified as declarative index meta-data stored in the index property tables. This is possible since the SSF rewriter is designed particularly for index utilization rather than for general query transformations as [5] [12] [13] [22].

QuEval [20] is a framework for performance evaluating spatial index implementations. Based on parameters specified for each evaluated spatial index implementation, built-in data generators produce data sets for performance evaluations. By contrast, the purpose of Mexima's test generator is to automatically generate correctness tests based on index specific meta-data and queries. Furthermore, unlike QuEval, new complex indexes in C/C++ can be plugged into Mexima without code changes.

The database generator QAGen [15] provides general purpose testing of DBMS components. It generates test databases and test queries based on symbolic execution of queries. In [4] an inverse relational algebra generates query

inputs for given query results. To implement unit testing for the query optimizer, the framework in [8] generates test queries based on user-defined transformation rules specified as trees of relational algebra operators.

In conclusion, no other system provides inclusion, validation, and utilization of unchanged complex index implementations plugged into an extensible main-memory DBMS.

3 Preliminaries

The terminology used in the rest of the paper is defined along with requirements on an index implementation for being suitable to be plugged into Mexima.

3.1 Terminology

Figure 2 illustrates the components of an index extension:

- The *index implementation (a)* is the code implementing the index structure. It is left unchanged when plugged into Mexima.
- The *index API (b)* is the provided public interface to the index implementation.
- The *index driver (c)* is the implementation of the BAOs and SSFs of an index calling the index API. Properties of the index driver are stored as meta-data in the index property tables.

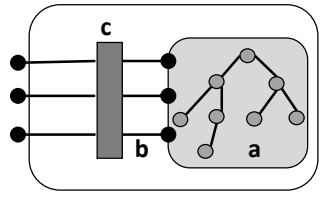

Figure 2. Index extension components

The above components are implemented by two kinds of developers:

- The *index developer*, who fully understands the algorithms and data structures used in the index implementation, develops the index code and API independent of Mexima.
- The *index extension developer*, who has sufficient understanding of the index and Mexima APIs but no knowledge of the index implementation and the DBMS kernel, develops the index driver.

Finally, the *end-user* defines indexes on tables and uses them in queries without concern for how they are implemented.

3.2 Prerequisites for index implementations

Mexima is designed bearing in mind the motto: *It should not be necessary to be a database kernel expert to introduce a new domain index*. An index implementation should thus meet the following two criterion:

- The candidate index implementation should be written in a regular programming language such as C, C++, or Java. In order to achieve high performance, C or C++ is preferable, for example to be able to plug in highly optimized C code such as the Judy-tries package [2].
- The candidate index implementation should provide APIs for the functionality of the BAOs and optional SSFs. Missing mandatory BAOs, e.g. mapping over indexed elements, may need to be implemented in the driver.

4 Mexima

Figure 3 shows the software layers of Mexima. *Query processing* uses the query processor of Amos II [29] to call *operations* that access the *Mexima core*. The Mexima core calls implementations of the BAOs and SSFs in the *extension driver* of an *index extension*.

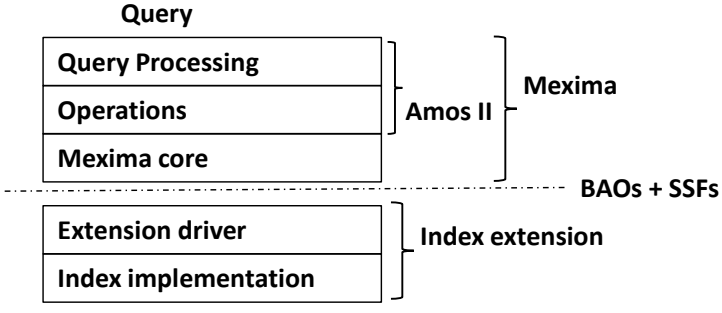

Figure 3. Mexima architecture

In the next section, we elaborate the implementation by first describing aspects of the query processing in Amos II followed by presentation of Mexima core.

4.1 Amos II

Figure 4 illustrates the details of Mexima, including how in utilizes the Amos II engine.

Amos II provides an object-oriented and functional query language, *AmosQL*. The parser translates a query into an object calculus representation [18] in *ObjectLog*, which is an extension of Datalog with objects, types, overloading, and foreign functions. Then the *calculus rewriter* transforms the un-

optimized object calculus expression to improve performance. After the rewrites, the *cost-based optimizer* produces an execution plan sent to the *execution plan interpreter*. Mexima extends the query processor with calculus rewrite rules for transparent utilization of new indexes.

AmosQL functions can be defined as *foreign functions* implemented in some regular programming language, e.g. C or Java. In Mexima SSFs are specified as foreign functions to enable query transformation of user queries into equivalent queries calling them. By contrast, BAOs are standard operations on domain indexes implemented as C functions called from the Mexima core when executing the operations.

In Amos II all data is stored in a continuous memory block called the *database image*. The *storage manager* is responsible for allocation and de-allocation of physical objects inside the database image. All data in a database are internally represented as physical objects managed by the storage manager. Physical objects allocated inside the image are persistent, which means that they can be saved on disk and later restored. A *physical object, po*, is accessed through an *object handle*, *hdl*, which is an indirect pointer to *po*. Amos II uses reference counting to manage memory allocation and automatic real-time garbage collection. When the reference counter of an object *po* in the image reaches zero, it is passed to the garbage collector and thereafter the memory occupied by *po* is marked as available for other memory allocation. Mexima extends the storage manager of Amos II with specialized external *index storage managers* for each index type. The garbage collector is called by the Mexima core when executing index updates.

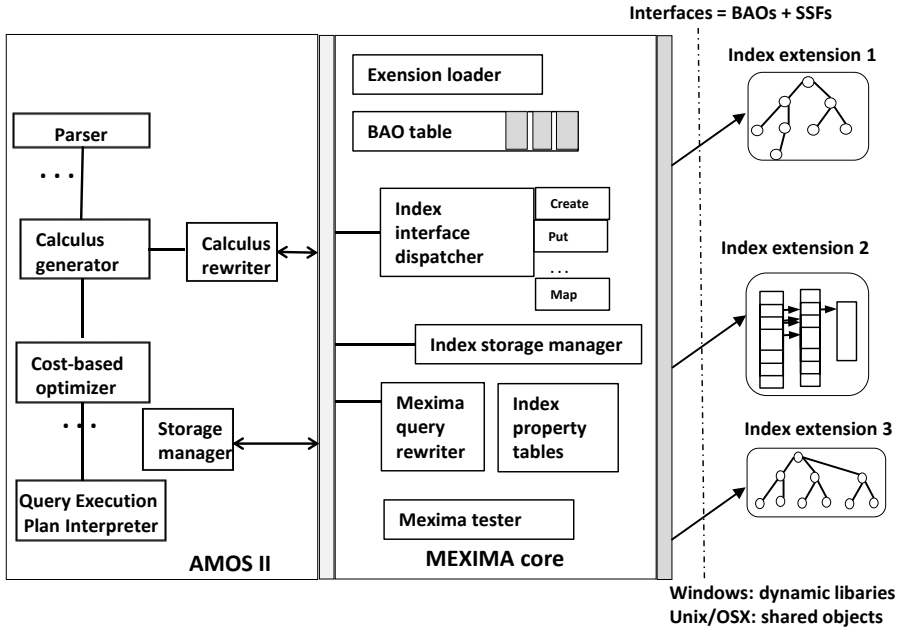

Figure 4. Mexima details

4.2 Mexima core components

We now discuss in detail the components of the Mexima core. To illustrate the functionality, we shall use an external index structure package named *IDS* through our discussion and examples. It indexes integers only.

The extension loader

The *extension loader* loads at run-time the index extension IDS as a dynamic library or shared object (step 1). It calls the initialization function (step 2) *a_initialize_extension()* of the index driver when the index extension has been loaded to register the index interfaces as C functions with Mexima (step 3). The index name IDS and its registered C functions are stored in Mexima's *BAO table* (step 4).

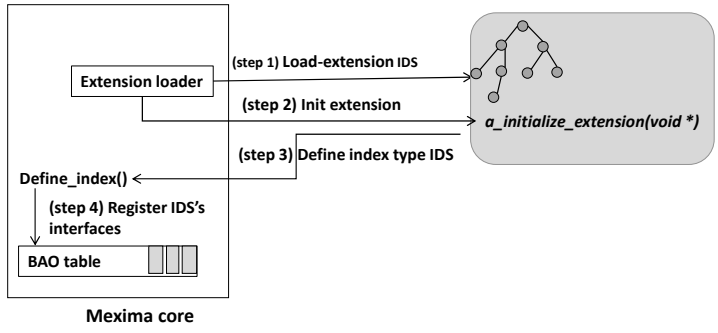

Figure 5. Extension loader's steps

The five mandatory BAOs registered in step 3 are: *create()*, *drop()*, *put()*, *delete()*, *get()*, and *map()*, where *create()* creates a new index while *drop()* removes it, *put()* inserts a key/value pair while *get()* retrieves it, and *delete()* removes it. The BAO *map()* scans the index by applying a specified mapper function on each index entry.

Some indexes require transforming the keys into integers used as actual keys, e.g. hashing or space filling curves. This is specified by the optional BAO *compute_key()* while the optional BAO *compare_key()* compares two computed keys for (in)equality.

For the representation of keys there are two variants supported:

- The index extension stores *boxed keys*, which are object handles managed by the storage manager. The data type of object handles is unsigned integer, so any index extension supporting integers can store boxed keys. In this case, the BAO *compare_key()* is not needed in Mexima, since comparisons of handles is built-in.
- If an index stores *unboxed keys,* i.e. the key values themselves, *compare_key()* compares keys, while *compute_key()* unboxes them.

Index interface dispatcher

When the end-user has placed an index of type IDS on an attribute of a table, the *index interface dispatcher* (*Figure 4*) accesses the index by invoking the corresponding registered BAOs (*create()*, *put()*, *get()*,etc.,) in the BAO table.

The index interface dispatcher is also responsible for maintaining reference counters of boxed keys and values so that the extension developer need not know about garbage collection.

Index storage manager

If the index implementation has storage facilities to persist index structures and has registered to Mexima the optional persistency BAOs *save()* and *restore()*, the *index storage manager* will invoke them upon saving and restoring the database.

If an index implementation is not persistent, i.e. it is all implemented in main-memory; Mexima automatically serializes and de-serializes the index entries. To save the index on disk, the index storage manager scans over the index entries using the BAO *map()* and streams them to disk. Only the primary index is made persistent, since, when restoring a table by streaming its rows from disk, the index storage manager also builds the secondary indexes. In case the index implementation does not balance the index structure on insertion, the restored index structure may become unbalanced, and the extension developer can then register a bulk loader and hook it to *restore()*.

Internally, the index storage manager relies on two system hooks executed at different states of the system: the *before-image-roll-out* hook is executed when a database is saved, and the *after-image-initialized* hook is executed when a database is restored. The index storage manager keeps track of all created indexes to save and restore them.

Mexima Query Rewriter
In order to utilize a new index in queries, the *Mexima query rewriter* transforms them to expose the SSFs of the new index. The *index property table* contains the necessary meta-data to do the transformations. This is further described in Section 6 below.

Mexima Tester
In order to validate that an index implementation is correct, the *Mexima tester* automatically generates and runs tests based on meta-data in the index property tables, as described in Section 7.

4.3 Implementation of an SSF

The index driver bridges Mexima and an index extension by implementing BAOs and SSFs. SSFs are defined as foreign functions that also can be used in queries. For example, if the index type IDS supports range search, it can be implemented by the SSF foreign function *IDS_select_range()* registered as follows in the initialization function of the index driver:

```
1  // Definition of the foreign function's signature:
2  a_amosql("create function IDS_select_range(Function tbl, Integer pos, Number lower, Number upper)-> Object as foreign 'IDS-range-search';");
3 // Bind C function IDS_range_search address to the symbol 'IDS-range-search':
a_extfunction("IDS-range-search",  IDS_range_search);
```

Here:

- The signature of the foreign function *IDS_select_range()* is defined by the *a_amosql()* call. In the signature the parameter *pos* is the indexed position

on the function *tbl* representing an indexed table, while *lower* and *upper* define the range in a search.

- *a_extfunction()* associates the address of the C-function implementing the SSF with a symbol used in the signature definition.

The first two arguments *tbl* and *pos* are bound when the SSF is called in a query. The remaining arguments, here *lower* and *upper*, are called *SSF parameters*. They are different for different SSFs and are bound in queries rewritten by the SSF translator based on meta-data in the index property tables. Even though the user can also call an SSF with explicit parameters specified in queries, this is not recommended since it makes the index access non-transparent.

The following snippet shows the C implementation of *IDS_range_search()* in the index driver of the IDS:

```
1    void IDS_range_search(m_context cxt){

2    a_handle tbl = a_arg(cxt,0);   // Table handle
3    int pos = a_int_arg(cxt,1);   // Indexed pos
4    int l = a_int_arg(cxt,2); // lower range
5    int u = a_int_arg(cxt,3); // upper range
6    IDShead *ind=(IDShead *)mexima_identifier(pos, tbl,ids_type);
7    IDScomparer cmp = mexima_get_comparer(pos, tbl, ids_type);
8    // call the map function of IDS-API:
9    IDSmap(ind->root, l, u,
10   (IDSmapper)rangemapper, cmp, cxt);}
     // the function rangemapper() is defined as:
11   int rangemapper(IDSitem *kv,m_context cxt){
12   a_bind(cxt, 4, kv->value);
13   a_emit(cxt);}
```

The *IDS_range_search()* accesses the first four function parameters from the binding context *cxt* on lines 2-5. Lines 2 and 4-5 dereference the handles to get integer values[2]. Line 6 assigns the pointer *ind* to the index structure on position *pos* of table *tbl*. Line 7 retrieves the compare function of the IDS registered in the BAO table. On line 9 the index API *IDSmap()* iterates over the index *ind* and calls the function *rangemapper(kv, cxt)*, defined on lines 11-13, on each index key/value pair *kv*. On line 12, the row (value part) of *kv* is bound to the result (5th parameter). Finally, the macro *a_emit()* emits a result tuple to Mexima.

[2] The system raises an error if the parameters are not integers.

5 Illustrative Query Examples

In this section, we present a database schema and queries to serve as examples when discussing Mexima's query processor.

In the table *images(id, hist)* each row represents an image identified by *id*. Search on table image often requires comparing images. However, it is expensive to compare images bit by bit. The most common technique is approximating an image with its features. Thus, a comparison between images becomes the cheaper comparison between the images' features. In our example, the features on an image are represented by its color histogram stored in the attribute *hist* as a vector of numbers.

To speed up search on table *images*, there is a B-tree index on column *id* and an X-tree index [27] on column *hist*. X-trees supports efficient proximity search of high-dimensional data. Main-memory implementations of B-trees and X-trees [30] are plugged-in to Mexima.

In the following example, we use the ObjectLog representation into which the queries are translated to illustrate the query processing.

Q1: find images *q* whose identifiers are between 30 and 100. In this case, there is no input parameter:

```
Q1(q):-
1 | images(q, hist_q)                    AND
2 | q >= 30                              AND
3 | q <= 100
```

Q2: For a given image *x* find the images q whose feature vectors are closer than epsilon *(eps = 0.11)*. In the query, the function *distance()* computes the Euclidean distance of two vectors.

```
Q2(x, q) :-
1 | images(x,  hist_x)                  AND
2 | images(q,  hist_q)                  AND
3 | distance (hist_x, hist_q) <= 0.11
```

Q3: find the *k = 10* closest images compared to a given image bound to *x*. We use the *'knn'* function to return the k nearest neighbors in table *'images'* to the input color histogram of *x*. *knn()* uses the table *'images'* that maps from an object identifier to its feature vectors.

Q3(x, q) :-

1	images(x, hist_x)	AND
2	images(q, hist_q)	AND
3	(q, hist_q) in knn(hist_x, 10, #'images')	

Q4: We note that the *distance()* function used in Q2 expresses the distance between vectors, but not similarity. To define similarity, we define query *Q4* using the following formula:

$$\frac{1}{1 + distance(p,q)} > threshold$$

Q4 finds images *q* that are 90 percent similar to a given image bound to *x*:
Q4(x, q):-

1	images(x, hist_x)	AND
2	images(q, q)	AND
3	1/(1+distance(hist_x, hist_q)) >= 0.90	

6 Mexima Query Rewriter

This section presents the SSF Translator. It transforms a query into an equivalent one where SSF calls are exposed to the query optimizer. If this transformation is not done, the optimizer is unable to utilize the index.

The system also does other rewrite tasks not related to indexing, e.g.: view expansion, elimination of common sub-expressions, and compile-time evaluation, which are not focus of this paper.

6.1 SSF translation rules

An SSF translation rule describes how query fragments are translated to a new format to expose SSFs. The translation rules can rewrite conjunctions in queries having terms of one the following *query fragment forms*:

Form (i): P(…**iv**,..) AND (iv $\mathbf{r_1}$ expression) AND

(iv $\mathbf{r_2}$ expression) AND

. . .

(iv $\mathbf{r_n}$ expression)

Here, ***iv*** is a variable bound to an indexed column of table *P(...)*. We say ***iv*** is an *indexed variable*. $\boldsymbol{r_i}$ are comparison operators in the set ***relop***, $\boldsymbol{r_i} \in$ *relop,* where ***relop*** *={=, <, >, >=, <=}*.

For example, the following fragment in Q1 is of Form (i):
images(q, hist_q) AND q >= 30 AND q <= 100.

Form (ii): P(…**iv**,..) AND **isf**(…,**iv**, …) $\mathbf{r_1}$ expression AND

isf(…,**iv**, …) $\mathbf{r_2}$ expression AND

. . .

isf(…,**iv**, …) $\mathbf{r_n}$ expression

Here, ***iv*** is an indexed variable occurring in parameter position of an index sensitive function ***isf()***.

For example, the following fragment in Q2 is of Form (ii):
images(q, hist_q) AND distance(hist_x, hist_q) <= 0.11

Form (iii): P(…,**iv**,…) AND (..,**iv**,..) in **isf**(…..,P,..)

Here the ***isf()*** is an index sensitive function that takes a table ***P*** as argument and emits a set of rows. For example, Form (iii) occurs in Q3:

images(q, hist_q) AND (q,hist_q) in knn(hist_x, 10, #'images')

If a query contains some fragment that matches any of Form (i), (ii), or (iii), the query has the potential of being supported by the index on ***iv***. If this is the case, the query fragment should be transformed into a format where the index is exposed through an SSF call. For each kind of index, the index developer can define SSF translation rules, which transform query fragments that match Form (i), (ii), or (iii), into the corresponding SSF call. The SSF translation rules are defined as rows in the *SSF translation table*. Table 1 is an example of translation rules for B-trees and X-trees indexes.

Table 1 SSF translation table

#	itype	pr	ISF	Relops	SSF	pf
1	B-tree	1	Nil	>=, <=	btree_select_range	F
2	B-tree	2	Nil	<=	btree_select_open	F
3	X-tree	1	distance	<=	xt_proximity_search	T
4	X-tree	2	Knn	nil	xt_knn_search	F

Each row represents an SSF translation rule. It has the attributes *itype*, *pr*, *isf*, *relops*, *ssf*, and *pf* where:

- *itype* is a user-defined index type.
- *pr* is the translation rule priority for a given *itype*.
- *isf()* is an index sensitive function. *isf* is nil in Form (i).
- *relops* is a set of allowed relational operators in *{=, <, >, >=, <=}*. *relops* is nil in Form (iii). The system knows how to infer open inequalities from closed ones.
- *ssf()* is a special search function supported by the index type.
- *pf* is the prune and filter flag. When it is true (T), the Mexima query re-writer applies the two-step paradigm [24], in which the *prune step* first prunes irrelevant data by calling the SSF to return a small set of candidates and then the *filter step* applies the original condition to carefully examine each candidate. Here it is important that pruning is done before the filtering.

For a given query fragment of Form (i), (ii), or (iii), the system finds the *matching* SSF translation rules. Form (i) matches SSF translation rules where *isf* is empty, Form (ii) matches rules where both *isf* and *relops* are non-empty, while Form (iii) matches rules where there is an *isf* but no *relops*. If more than one rule matches, the priority *pr* determines which one. If *pr* is nil and more than one rule applies, the system will pick one of the matching rules.

In Table 1 the translation rules TR1 – TR2 together define query fragments where B-trees interval search should be used, while TR3 define when X-trees proximity search should be used. The proximity search requires pruning so *pf* is true. Lastly, TR4 defines the translation from the ISF *knn()* to the SSF *xt_knn_search()*.

If an SSF translation rule for index type *itype* matches a query fragment of Form (i), (ii), or (iii) where *iv* the indexed variable, the SSF translator will replace *P(.., iv, ...), isf(...)*, and *relops* with the corresponding SSF defined by the rule. If the index translator finds no applicable translation rule, the query is kept intact.

For example, by applying rule TR1 on Q1, it is translated into calling the SSF *btree_select_range()*:

```
TQ1(q):-
1 | (q,_) in btree_select_range( #'images', 0, 30,100)
```

Analogously, applying rule TR3 on Q2 yields the transformed query TQ2:

TQ2(x, q):-

```
1 | image(x, hist_x)                                              AND
2 | (q, hist_q) in  xtree_proximity_search(#'images',
                          1, hist_x, 0.11)                        SAND
3 | distance (hist_x, hist_q) <= 0.11
```

Since TR3 has the prune and filter flag set, line 2 in TQ2 prunes away most images and then line 3 filters them with the full condition. The operator SAND is an order-preserving conjunction. TQ2 exposes the X-trees index on column *hist* by the SSF *xtree_proximity_search()*.

Finally, applying rule TR4 on Q3 yields the transformed query TQ3 that exposes the X-trees index by the SSF *xt_knn_search()*:

TQ3(x, q):-

```
1 | image(x,  hist_x)                                             AND
2 | (q,_) in xt_knn_search (#'images', 1, hist_x, 10)
```

For query Q4, neither of Form (i), (ii), or (iii) match since the ISF *distance()* is hidden inside the numerical expression. We next discuss our general solution for this case.

6.2 Extended Algebraic Query Inequality Transformation

The AQIT algorithm [28] translates a class of numerical expressions with inequalities over variables indexed by B-trees into query fragments of Form (i). The translations use a set of algebraic inequality transformations. AQIT can transform conjunctive query fragments having terms of Form (iv):

Form (iv)	P(…**iv**,..) AND **F(iv) relop** expression

Here ***iv*** is an indexed variable and ***F(iv)*** is an expression consisting of a combination of transformable functions *T*. Currently $T \in$ *{+, -, /, *, power, sqrt, abs}* and the set can be extended. AQIT tries to reformulate the query condition into an equivalent equivalent condition ***iv relop' F'(expression)*** of Form (i) where the index is exposed to the query optimizer. The algebraic inequality transformations in AQIT automatically determine *relop'* and *F'(expression)*. If AQIT fails to transform the condition, the original query is retained.

However, AQIT cannot translate numerical expressions as in Q4 because the ISF *distance()* is hidden inside the expression. Therefore, in Mexima, AQIT is generalized to translate inequalities over ISFs into query fragments of Form (ii). The extended AQIT automatically transforms conjunctive fragments with terms of Form (v):

Form (v)	P(…,iv,…) AND **F(isf(…,iv, …)) relop** expression

Here ***F(isf(…,iv,…))*** is an expression consisting of a combination of transformable functions *T*, and ***relop*** is an inequality comparison. The extended AQIT tries to reformulate the query fragment into ***isf(…,iv,…) relop' F'(expression)*** of Form (ii) where the index on iv is exposed.

For Q4, the system first applies the following algebraic inequality transformation:

$(A/x >= B \wedge A > 0 \wedge B > 0) \Leftrightarrow x <= A/B$

The query will be transformed to *TQ4-intermediate0:*

TQ4-intermediate0(x, q):-

1	images(x, hist_x)	AND
2	images(q, hist_q)	AND
3	(1+ distance (hist_x, hist_q)) <= 1/ 0.9	

Then, the system applies the transformation:

$x + A <= B \Leftrightarrow x <= B - A$

The query will be transformed to TQ4-intermediate1:

TQ4-intermediate1 (x, q):-

1	images(x, hist_x)	AND
2	images(q, hist_q)	AND
3	distance (hist_x, hist_q) <= 1/0.9 -1	

TQ4-intermediate1 matches Form (ii), which allows the SSF translator to apply translation rule TR3. This transformation produces the final TQ4:

TQ4 (x, q):-

1	images(x, hist_p)	AND
2	(q, hist_q) in xtree_proximity_search('image', 1, hist_p, (1/0.9 - 1))	AND
3	1/(1+distance (hist_p, hist_q)) >= 0.9	

7 Mexima Tester

To validate that a plugged-in index implementation is correct, Mexima provides automatic testing procedures of BAOs and SSFs. Both BAOs and SSFs are tested based on meta-data in the index property tables. For each index type, a number of test queries are automatically generated and executed. The test queries use *data generators*, which are queries specified by the extension developer that generate index keys for testing BAOs and SSFs.

The system has a library of predefined data generators implemented as foreign functions calling the C++ library *random.h* to support randomly generated numbers and vectors of numbers respecting various distributions. New data generators can easily be defined is terms of these as queries.

For example, the built-in data generator *uniform_int(n,l,u)* generates *n* integers in range *[l,u]*. For complete testing, the result set always includes the border values *l* and *u*. The data generator *uniform_vec_real(n,d,l,u)* generates a set of *n* vectors of dimension *d* where each element is a real number in the range *[l,u]*, including *l* and *u*.

7.1 The BAO Tester

The BAO tester automatically tests that the BAOs of an index implementation are correct, i.e. correct behavior of *put()*, *get()*, *delete()*, *map()*, and *drop()*. It also provides a function to produce a report of the execution times of each BAO.

The BAO tester is based on data generators specified as queries stored in the *index key generator table* (Table 2). The extension developer populates the table and specifies how index keys to be tested are generated. Based on the generated keys, the BAO tester runs a number of built-in algorithms described below to test basic index functionality.

Table 2 Index key generator table

#	Idxtype	Index key type	Index KeyGenerators
1	B-tree	Number	select uniform_int(1000,0,10000)
2	X-tree	Vector-Number	select uniform_vec_real(1000,5,0,1)
3	X-tree	Vector-Number	select CSV_file_rows("colorhistogram.csv")

In Table 2 the first row specifies a correctness test of B-tree indexes by generating 1000 uniformly distributed integer keys in range 0-10000. The 2nd row specifies a correctness test for X-trees by generating 1000 uniformly distributed vectors of real numbers of dimension 5 in range [0, 1]. The last row tests X-trees by reading index keys from a file *"colorHistogram.csv"*.

Based on the index key generator table, the BAO tester will run the following tests:

- *Lookup* tests that all inserted keys are also stored in the index.
- *Mapping* tests that the mapper iterates over all inserted key/values.
- *Deletion* tests that iteratively deleting one key at the time works.
- *Remaining* verifies that no keys are remaining after all keys have been deleted individually.
- *Dropping* tests that the *drop()* operation removes all key/values.

The result of the BAO tester is an error report that specifies for each test case, which BAO functionality failed.

The BAO tester does the following:

1. Create two tables, the indexed table: *I_Table(k, v)*, and the reference table: *R_Table(k, v)*. On column I_Table.k the system puts an index of type IDS, *idx(I_Table.k)*, while on column *R_Table.k* there is a hash index *idx(R_Table.k)*.
2. For each test case, the BAO tester first calls the key generator. For each generated key *k* and a corresponding random number *v*, it inserts a row *(k,v)* into both *I_Table* and *R_Table* using *put(k,v)*.
3. For lookup, the BAO-tester iterates though the R_Table to test correctness of *put()* and *get()*. For each key/value in *R_Table* it tests that the result of accessing the key in *I_Table* calling *get()* returns the same value.
4. For mapping the BAO tester iterates over each *(k,v)* in *I_Table* using *map()* and tests that the key/value pair is present in *R_Table*.
5. For deletion, the BAO tester uses *map()* to iterate over all *(k,v)* in *I_Table* calling *delete(k)* followed by *get(k)* to check that each value is actually deleted.
6. For remaining, the system verifies that the table is empty after step 5.
7. For dropping, the table is repopulated, then *drop()* is called, and eventual remaining keys are reported.

7.2 The SSF tester

The purpose of the SSF tester is to validate that the result from an SSF is correct. Based on user-defined generators of SSF parameters, the system automatically generates test queries for each SSF translation rule of an index type IDS. The tests are based on that the SFF translation rules provide transparent rewrites of a generated test query to utilize the index through the SSF. When an index is defined for some attribute and can be utilized by some SSF translation rules in a test query, the query should return the same result as when there is no index or no matching SSF translation rule.

In order to test an SSF, the user needs to specify data generators for SSF parameters as queries stored in the *SSF parameter generator table* (Table 3).

Table 3 SSF parameter generator table

#	Index type	SSF name	SSF parameter generator	SSF Parameter types
1	B-tree	btree_select_range	select l, u from Number l, Number u where l in uniform_int(100, 0,10000) and u in uniform_int(100, 0,10000)	(Number, Number)
2	B-tree	btree_select_open	select u from Number u where u in uniform_int(100, 0,10000)	(Number)
3	X-tree	xtree-proximity-search.	select x, d from Vector of Number x, Number d where x in uniform_vec_real(100,5,0,1) and d in uniform_real(100, 0, 1.4)	(Vector of Number, Number)
4	X-tree	xtree_knn-search	select x, k from Vector of Number x, Number k where x in uniform_vec_real(100, 5,0,1) and k in uniform_int(0,5)	(Vector of Number, Number)

In Table 3 the first row tests *btree_select_range()* by generating the two SSF parameters as 100 pairs of random integers in range *[0, 1000]*. The 2nd test case validates *btree_select_open()* by 100 random numbers in range *[0,1000]*. The 3rd test case validates X-trees proximity search by generating 100 pairs *(x,d)* where *x* is a 5D vector of random numbers in range *[0,1]* and *d* is a random number in range *[0,1.4]*. The fourth test case validates KNN search with an X-tree by generating 100 pairs *(x,k)* where *k* is the number of closest neighbors to be tested. There can be several test cases specified per SSF.

For each test case in the SSF parameter generator table (Table 3), the SSF tester generates one *SSF validation query VQ* for each SSF translation rule TR in the SSF translation table (Table). The generated validation query *VQ* contains a query fragment of form *Fm* matching the TR.

The SSF translator will rewrite the *VQ* using the translation rule *TR* when *VQ* contains query fragments of form *Fm* matching the *TR*. In order to guarantee that no other *TR* matches *VQ*, all other translation rules matching *Fm* are temporarily turned off when executing *VQ*. For each index type, this test procedure validates both the TRs and the SSFs.

Meta-data to generate each *VQ* is obtained by joining the SSF translation rule table (Table), the SSF parameter generator table (Table 3), and the index key generator table (Table 2), to get for each test case the index key type, the SSF name, the SSF parameter generator, and the SSF parameter types, respectively. For each test case and *TR*, two queries VQ_t and VQ_r are generated. *VQi* is a query over *I_Table*, which is rewritten by the chosen TR to call the SSF. VQ_r is the same query over the *R_Table*. If the SSF is correct, both queries should return the same result. Depending on which form *Fm* is matching *TR* the validation queries are generated as follows:

Case 1: TR matches Form (i).
Assume the SSF parameters types in the SSF parameter generator table are $T_1,.., T_m$ (Table 3), that *IT* is the index key type in the index key generator table (Table 2), that SPG is the SSF parameter generator (Table 3) for parameters *p1,…,pm*, and that ri are the *relops* in Form (i). Then the validation query VQ_i has the following format:

```
select iv, v
from IT iv, Number v,
          T1  p1, T2 p2,.., Tm  pm
where I_Table(iv, v)                  and
        (p1, p2, …,pm) in (SPG)   and
        (iv r1 p1)                    and
        (iv r2 p2)                    and
        . . .
        (iv rm pm);
```

For example, the automatically generated validation query *VQi* for test case 1 in Table 3 is:

```
select iv, v
from Number iv, Number v, Number  p1, Number p2
where I_Table(iv, v) and
      (p1, p2) in (select l, u from Number l, Number u
                   where l in uniform_int(100, 0,10000) and
                    u in uniform_int(100,0,10000)) and
       iv >= p1 and  iv <= p2;
```

Case 2: TR matches Form (ii).
VQi has the following format, assuming the *ISF()* has arity *j*.

```
select iv, v
from IT iv, Number v,
        T1 p1, T2 p2,.., Tm pm,
        Tj res
where I_table(iv, v)                  and
      (p1, p2, …,pm)  in (SPG)        and
      res = ISF (iv, p1,..,pj-1)      and
      (res r1 pj)                     and
       . . .
      (res rm pm);
```

For example, the generated validation query VQ_i for test case 3 in Table 3 is:

```
select iv, , v
from Vector of Number iv, Number v, Vector of Number p1,
       Number p2, Number res
where I_Table(iv, v) and
       (p1, p2) in (select x, d from Number x, Number d
                    where x in uniform_vec_real(100,5,0,1) and
                     d in uniform_real(100,0, 1.4)) and
        res = distance(iv, p1) and  res<= p2;
```

Case 3: When TR matches Form (iii) the generator validation query has the form:

```
select  iv, v
from IT iv, Number v,
        T1 p1, T2 p2,.., Tm pm,
where I_table(iv,v)             and
  (p1, p2, …,pm)  in (SPG)      and
   (iv,v) in ISF (p1,..,pm, I_Table)
```

For example, the generated validation query VQ_i for test case 4 in Table 3 is:

```
select iv, v
from Vector of Number iv, Number v, Vector of Number p1,
       Number p2
where I_Table(iv, v) and
       (p1, p2) in (select x, k from Number x, Number k
                  where x in uniform_vec_real(100,5,0,1) and k in
                   uniform_int(0,5)) and
        (iv,v) in knn(p1, p2, #'images');
```

The BAO and SSF testers are run on all chosen index implementations to validate that they were correct. One bug in the R* package [7] and two bugs in the X-tree implementation [30] were found by the SSF tester.

8 Experiments

We measured the performance of using Mexima for main memory implementations of B-trees [30], Linear-Hashing [30], Judy-Tries [2], X-trees [30], and R*-trees [7].

We conducted experiments in several perspectives. First, in Experiment A we compared the coding effort of the different index implementations based on disk-based GiST and SP-GiST with the corresponding main-memory index extensions in Mexima w.r.t. code size.

In Experiment B, we compared the execution times of calling a plugged-in index through the BAOs *put()*, *get()*, *map()*, and *delete()* with the execution times of the corresponding stand-alone implementations in C/C++. The absolute time difference was calculated as *overhead*. The overhead of both boxed and unboxed keys were investigated.

In Experiment C, the importance for scalability of using SSF translation rules is investigated. The queries were run with and without SSF translation enabled.

All performance experiments were repeated 10 times, from which the average figures were calculated after removing outlier results if any.

The experiments were run under Windows 7 on an Intel (R) Core(TM) i5 760 @2.80GHz 2.93 GHz CPU with 4GB RAM, using the Visual Studio 10 32 bits C compiler.

Experiment A – Code size

Table 4 shows the number of C/C++ code lines of different index interface implementations in PostgreSQL version 9.3.5 (http://www.postgresql.org/ftp/source/v9.3.5/) and SP-GiST version 0.0.1 [25], compared to the corresponding Mexima drivers. We excluded comments in the comparisons. The compared code is what an index extension developer needs to provide to interface the DBMS extensibility frameworks.

Table 4 Number of code lines'

	GiST	SP-GiST	Mexima	Factor
B-tree	5031	--	116	43
KD-tree	--	572	118	5
R-tree	1133	--	120	9.5
Trie	--	580	120	5

In PostgreSQL, the GiST-based B-tree was implemented as a fully separate module from the GiST core, while parts of the R-tree implementation are present in the GiST core. Thus, the number of code lines for R-trees with GiST is underestimated in the table.

Table 4 shows that the code size of including a main-memory index implementation in Mexima is 5–43 times smaller than the corresponding disk based index plug-in with GiST.

Notice that the Gist based index implementations are specially designed to follow the Gist coding conventions, while with the Mexima framework all used index implementation code is left unchanged, including memory allocation, which is particularly complex in Judy-tries.

To conclude, Mexima provides introduction of domain indexes with relatively little coding effort for the interface between the untouched domain index implementation and the Mexima kernel. This allows to plug-in very complex main-memory index implementations with small efforts.

Experiment B – Mexima BAO overhead

The purpose of this experiment is to investigate the performance overhead of plugging-in an existing index implementation in Mexima. *Figure 6* illustrates how the execution time is spent in different layers of a plugged-in index implementation.

Here:

- *op*: time spent to call algebra operations on an indexed table to add, delete, access, or map.
- *mc*: time spent to dispatch and call the BAO function in an algebra operation. This includes time spent for type checking and automatic garbage collection.
- *ed*: time spent in the index extension drivers for BAOs and SSFs.
- *st*: time spent on actually running the untouched index implementation code. This is the actual work to manipulate the index, i.e. the time to run the stand-alone C/C++ implementation.
- In the experiment, we measured the execution times for the different index implementations both when plugging-in the implementation into Mexima and when running the implementation as a stand-alone C/C++ program. The total execution time for using a plugged-in index implementation is $tot = op + mc + ed + st$. The Mexima overhead, o, of calling a plugged-in index implementation is calculated as $o = op + mc + ed$.

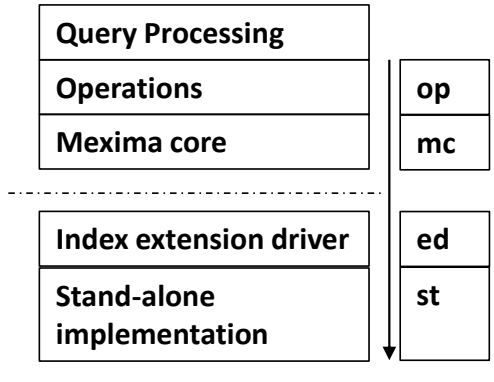

Figure 6. Execution layers

In the experiments, the performance of B-tree, Linear Hashing, and Judy-Trie implementations were measured for a database of size S with uniformly distributed random key/value pairs. The execution times of put(), get(), and delete() per call were measured by loading the database and then measuring the time of doing 1000 random inserts, lookups, and random deletes, respectively. The time to call map() was measured by iterating over the indexed table and dividing the total time with S. The time for generating data and populating the reference tables were excluded in all measurements.

Table 5 shows the average Mexima overheads o in microseconds for the BAOs *put()*, *get()*, *delete()*, and *map()*. The database size S was 5 million key/value pairs. The total overhead was measured with both boxed keys *bo* and unboxed keys *o*. The standard deviations in all cases were less than 0.03 µs. The overhead of Mexima is well below one µs per call and particularly low for unboxed keys, so unboxed keys are used in all remaining experiments. Table 5 furthermore breaks down the percentages of how the overhead *o* is spent in the different layers *op*, *mc*, and *ed*.

Table 5 Mexima overhead for different BAO calls (µs)

BAO	Index	bo	o	%op	%mc	%ed
Put	LH	0.89	0.56	51.7%	36.2%	12.1%
	B-tree	0.89	0.53	52.3%	35.8%	11.9%
	Judy-trie	0.87	0.54	52%	35.3%	11.7%
Get	LH	0.57	0.26	37.2%	47.1%	15.7%
	B-tree	0.59	0.23	36.6%	47.6%	15.7%
	Judy-trie	0.57	0.22	36%	48%	16%
Map	LH	0.21	0.07	32.1%	50.9%	17%
	B-tree	0.19	0.07	34.4%	49.2%	16.4
	Judy-trie	0.23	0.07	33.7%	49.7%	16.6%
Delete	LH	0.65	0.42	45%	41.3%	13.7%
	B-tree	0.64	0.42	43.3%	42.5%	14.2%
	Judy-trie	0.63	0.41	43.4%	42.5%	14.1%

Figure 7 shows insert times in microseconds of different index implementations with unboxed keys compared with the corresponding stand-alone implementations for different database sizes.

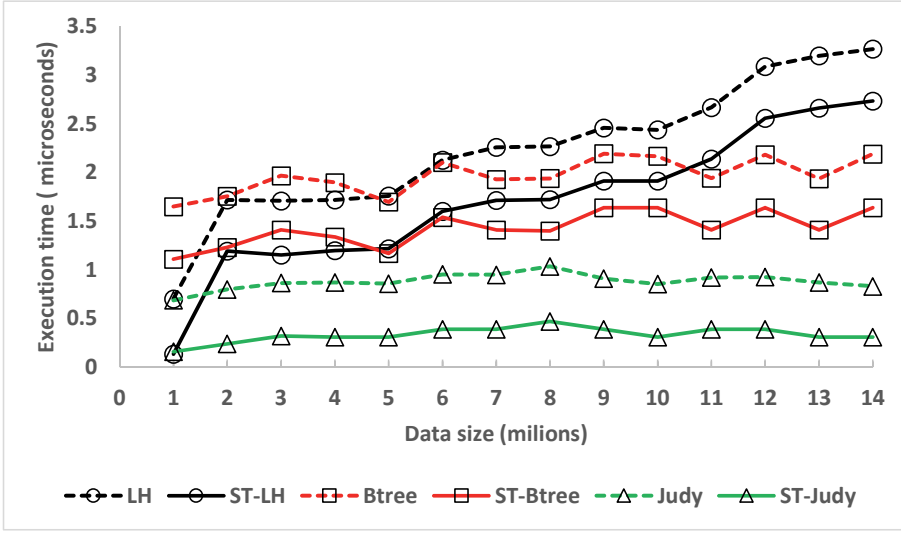

Figure 7. Put () overhead

Analogously, *Figure 8*, *Figure 9*, and *Figure 10* show lookup, delete, and map time per call with unboxed keys.

As expected, stand-alone index implementations were faster than their corresponding plug-in indexes using the same implementation because of the Mexima overhead. The overhead is not dependent on the database size for any of the methods as shown in *Figure 7*, *Figure 8*, *Figure 9*, and *Figure 10*. The system carefully makes sure that an index is not accessed more than once in an operation, as that would make the overhead larger as the database grows. For unboxed keys, the overhead is less than 0.6 µs and depends on the index driver implementation of the BAO, not the database size.

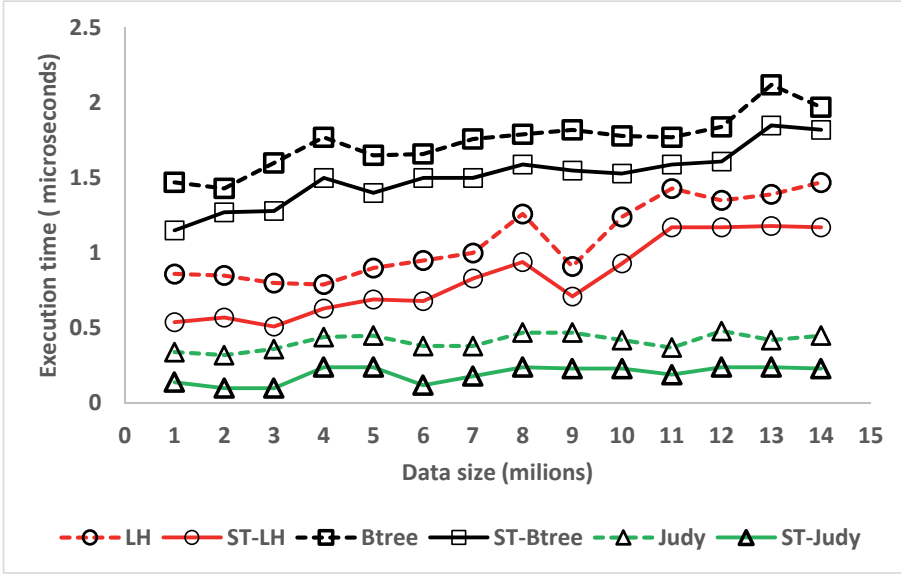

Figure 8. Get() overhead

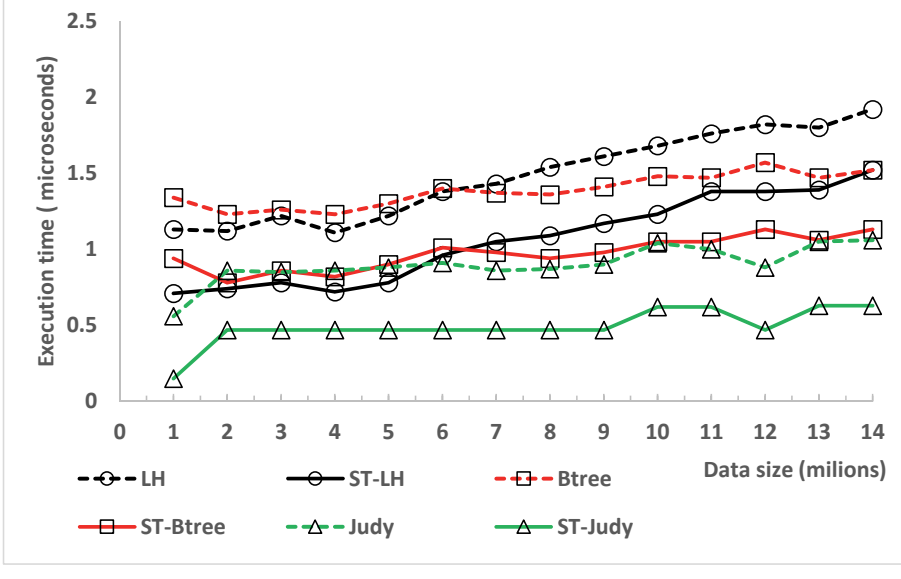

Figure 9. Delete()

The ease of plugging-in index implementations in Mexima without code changes with very low overhead shows that Mexima is an excellent tool for comparing domain index implementations. In particular not changing the index implementation allows to easily comparing highly optimized and complex domain-index implementations such as Judy-tries with other implementations.

For example, *Figure 7*, *Figure 8*, and *Figure 10* show that Judy-tries are better than B-trees for inserts and lookups, but not for mapping.

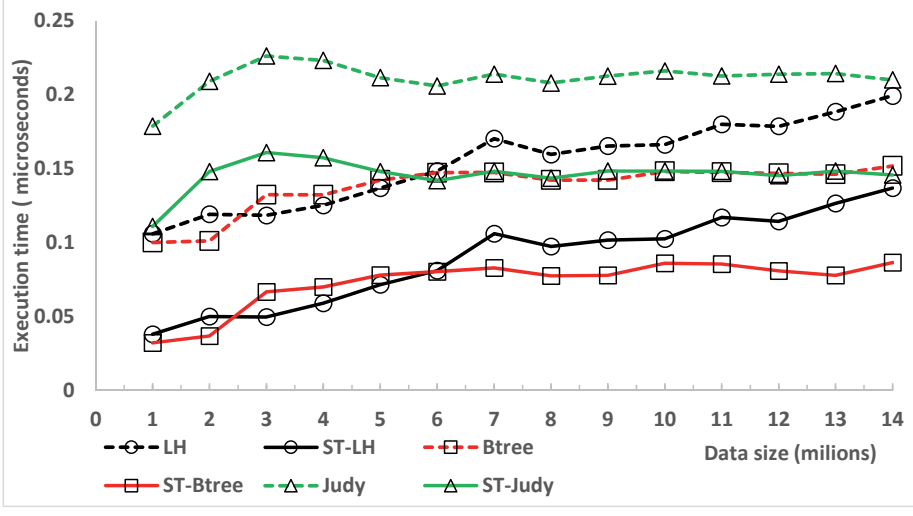

Figure 10. Map()

Experiment C – Impact of SSF translation rules

The purpose of this experiment is to show the importance of Mexima rewrite rules for scalable processing of user queries utilizing plugged-in domain indexes. In *Figure 11* the performance of range search query Q1 (Form (i)) with and without the SSF translation rules is investigated for B-trees and Judy-tries. Without the SSF translation rules the index is not utilized, so the system has to use the *map()* function to iterate over the index and then do post-filtering on every row. *Figure 11* also shows that B-trees are better than Judy-tries for interval search.

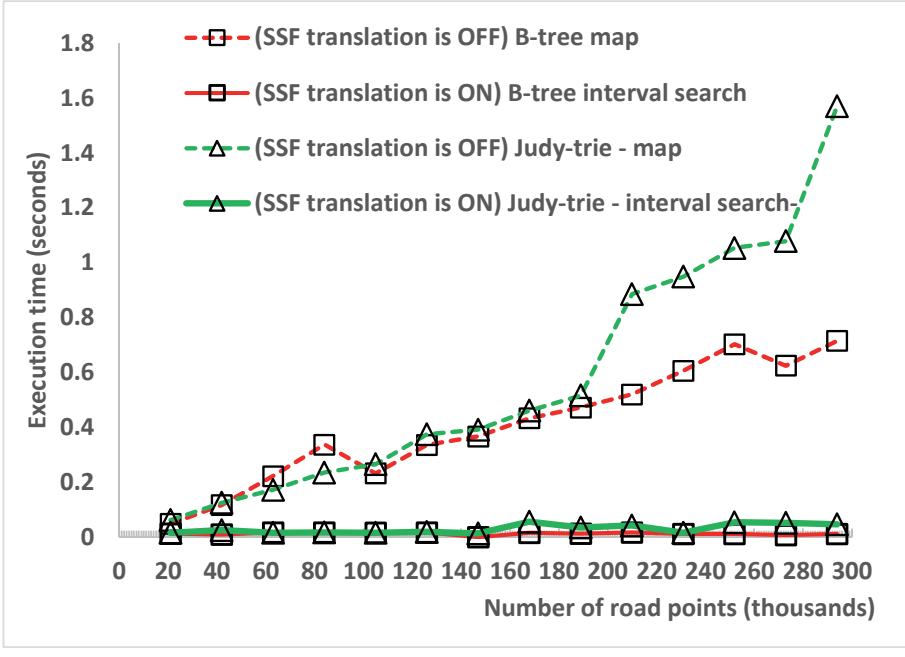

Figure 11. Q1 Range search

In *Figure 12* the performance of 2D KNN-search is investigated for X-trees and R*-trees. Query Q3 (Form (ii)) is used with the relation *Images* populated with 2D point vectors from a real data set [9], which is a collection of California road points. We enlarged the original data set to different data sizes by randomly generating points from 1.S to 14.S with S=210480 with the same range distribution as the origin. When SSF translation is enabled, Q3 with *k =10* scaled substantially better since the index was utilized. We also notice that the X-tree implementation performed as good as the R*-tree implementations for the given 2D database.

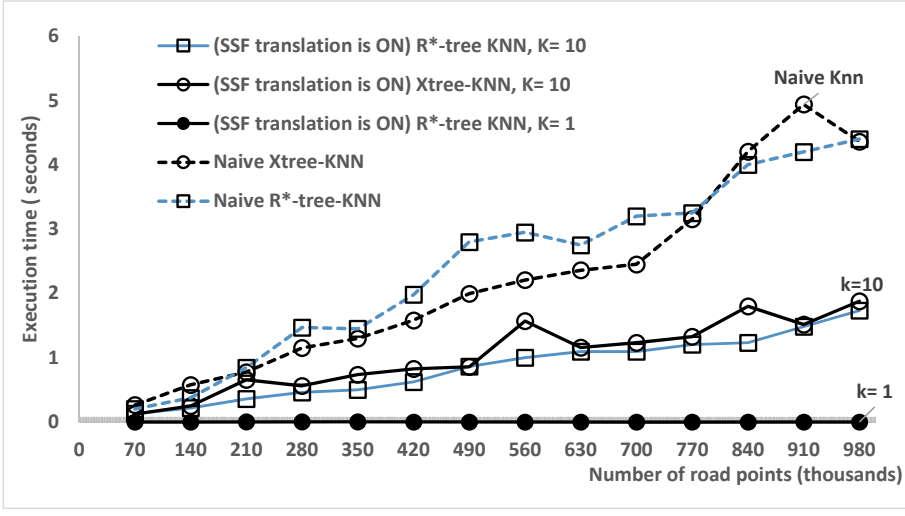

Figure 12. Q3 Knn

In *Figure 13* the performance of high dimension 9D proximity search for Q2 (Form (iii)) is measured with the X-tree implementation, with and without SSF translation rules enabled. In this experiment, we used the ColorHist database [6]. The database comprises of 9D (3 x 3) - color histograms extracted from *S=70000* images provided by the Corel Image Database. As for the road points, we enlarged the size of the database from 1.S to 14.S.

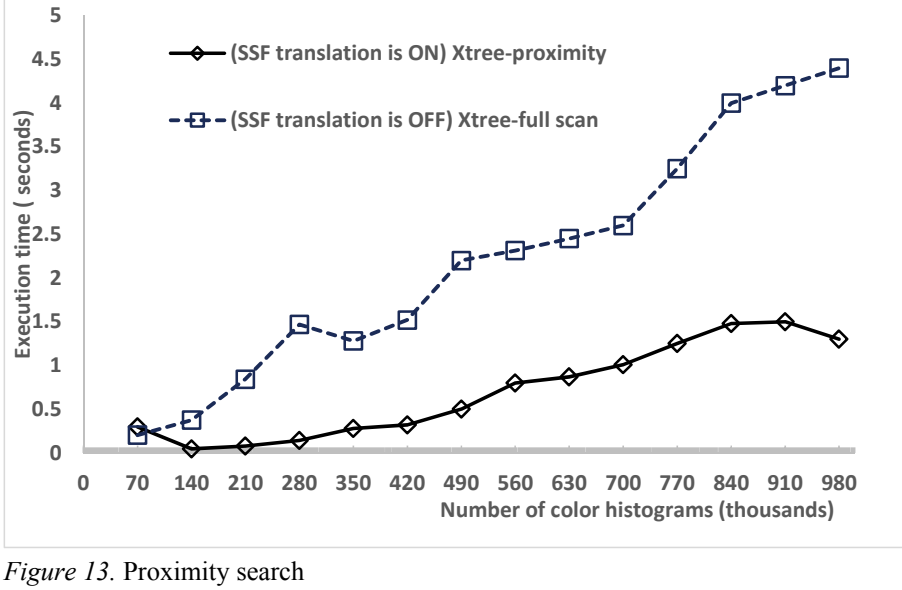

Figure 13. Proximity search

Figure 14 shows the scalability improvement by rewriting similarity queries for Q4 (Form (v)) using the X-tree implementation and the ColorHist database.

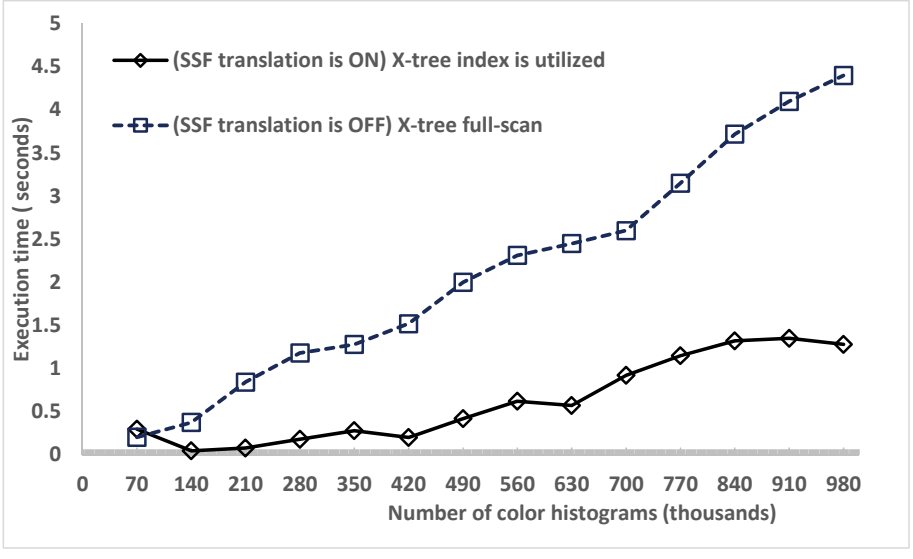

Figure 14. Similarity search

From experiment C, we conclude that SSF translation rules are critical for scalability of Mexima's extensible indexing, because they make the indexes be utilized in queries. To evaluate the quality of domain index implementations, plugging-in and comparing with ease proposed domain index implementations, as in Experiment C, are critical.

9 Conclusions & Future Work

The Mexima framework allows transparent plugging-in of main-memory domain index implementations into a main-memory DBMS without code changes. To plug-in a domain index implementation, the extension developer writes a simple Mexima driver for the universal index operations (BAOs) and the domain index specific search functions (SSFs).

To provide transparent utilization of SSFs in queries the extension developer populates an *SSF translation table* in which each row is an *SSF translation rule* that describes parameters for the query processor matching different query fragment forms. Combined with algebraic rewrites, the SSF translation rules provide the necessary meta-data for the query processor to generate scalable execution plans that utilize the index by calling its SSF operators.

To validate the correctness of new indexes, Mexima generates test queries based on index-specific data generators specified as queries by the extension

developer. The *index key generator table* contains queries that generate index keys to be tested for correct BAO behavior. The *SSF parameter generator table* contains queries that generate arguments of the SSF operators to be tested. Based on these two tables and the SSF translation table, Mexima automatically generates and executes test queries for the new index.

To show that existing index implementations can be transparently plugged into Mexima, five different main-memory index implementations were evaluated without changing their source code. In particular, the very complex Judy-trie index implementation [2] was included and compared with a textbook B-tree implementation.

The overhead of Mexima for BAOs of the different plugged-in index implementations was evaluated, showing that the current Mexima implementation has overhead in the sub-µs range per BAO call.

The importance of SSF translations was investigated for chosen index implementations showing the SSF translation rules provide scalable performance of declarative queries over tables indexed by plugged-in domain indexes.

The ease of plugging-in index implementations in Mexima without code changes and with very low overhead shows that Mexima is an excellent tool for comparing domain index implementations. In particular not changing the index implementation allows to easily utilizing highly optimized and complex domain-index implementations such as Judy-tries.

For future work, other kind's indexes will be plugged into Mexima to meet the specific requirements from other application domains. This is likely to put additional requirements on Mexima's query processor. Furthermore, also index performance measurements can be automated by extending the Mexima's tester to include performance tests.

Altogether, Mexima provides a complete and extensible platform for domain index integration and evaluation, as required in many scientific applications.

10 Acknowledgements

This work was supported by the Swedish Foundation for Strategic Research, grant RIT08-0041 and by the EU FP7 project Smart Vortex.

11 References

[1] W. G. Aref and I. F. Ilyas: An extensible index for spatial databases, *Proc. of Statistical and Scientific Database Management*, pp 49–58, 2001.

[2] D. Baskins: Judy home page [*http://judy.sourceforge.net/*], 2003.

[3] D. Benoit, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin: Automatic SQL tuning in Oracle 10g, *Proc. of Thirtieth international conference on Very large data bases-Volume 30*, pp 1098-1109, 2004.
[4] N. Bruno, S. Chaudhuri and D. Thomas: Generating Queries with Cardinality Constraints for DBMS Testing, *IEEE Transactions on Knowledge and Data Engineering, 18(12)*, pp 1721-1725, 2006.
[5] M. Carey, et al: The architecture of the EXODUS extensible DBMS, *Proc. 1986 international workshop on Object-oriented database systems, IEEE Computer Society Press,* 1986.
[6] Color Histogram Data Set: *http://archive.ics.uci.edu/ml/datasets/Corel+Image+Features*
[7] Efficient and Lightweight In-Memory Implementation of R*-Tree: *http://www.ics.uci.edu/~salsubai/rstartree.html*.
[8] M. Elhamali and L. Giakoumakis: Unit-testing Query Transformation Rules, *Proc. of 1st International Workshop on Testing Database Systems*, 2008
[9] L. Feifei, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.H. Teng: On trip planning queries in spatial databases, *Proc. of Symposium on Spatial and Temporal Databases*, pp. 273 - 290, 2005.
[10] R. A. Finkel, and J. L. Bentley: Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, vol. 4, pp 1–9, 1974.
[11] E. Fredkin: Trie memory, *Communications of the ACM, 3(9)*, pp 490–499, 1960.
[12] G. Goetz: The cascades framework for query optimization, *IEEE Data Engineering Bulletin. 18(3)*, pp 19-29, 1995.
[13] G. Goetz, W. J. McKenna: The Volcano optimizer generator: Extensibility and efficient search, *Proc. of IEEE Conference on Data Engineering*. pp. 209-218, 1993.
[14] A. Guttman: R-trees: A dynamic index structure for spatial searching, *Proc. SIGMOD Conf.*, pp 47–57, 1984.
[15] F. Haftmann, D. Kossmann and E. Lo: A framework for efficient regression tests on database applications, *The Very large data bases Journal, 16(1)*, pp. 145-164, 2007
[16] J Hellerstein. M., J. F. Naughton, and A. Pfeffer: Generalized search trees for database systems*, Proc. of The Very large data bases Conference.*, pp 562–573, 1995.
[17] M. Kornacker: High-performance extensible indexing, *Proc. of Very large data bases Conference*. pp 699–708, 1999.
[18] W. Litwin and T.Risch: Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering, 4(6)*, 1992.
[19] A. C. D. Neto, R. Subramanyan, M. Vieira, and G. H. Travassos: A survey on model-based testing approaches: a systematic review. *Proc. of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies, in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE), ACM*, New York, NY, USA, pp 31-36, 2007.
[20] M. Schäler, A. Grebhahn, R. Schröter, S. Schulze, V. Köppen, and G. Saake. 2013: QuEval: beyond high-dimensional indexing à la carte, *Proc. of the Very large data bases Endowment, 6(14)*, pp 1654-1665, 2013.
[21] Oracle Inc: Query Optimization in Oracle Database 10g Release 2. *http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-general-query-optimization-10gr-130948.pdf*, 2005.

[22] H. Pirahesh, T.C. Leung, & W. Hasan: A rule engine for query transformation in Starburst and IBM DB2 C/S DBMS. *Proc. of IEEE Conference on Data Engineering*, pp. 391-400, 1997.
[23] J. T. Robinson: The KDB-tree: a search structure for large multidimensional dynamic indexes, *Proc. of SIGMOD Conf.,* pp 10-18, 1981.
[24] S. Shekhar and S. Chawla: Spatial Databases: *A Tour, Prentice Hall, ISBN:013-017480-7*, 2003
[25] SP-GiST*: https://www.cs.purdue.edu/spgist/*
[26] J. Srinivasan, R. Murthy, S. Sundara, N. Agarwal, and S. DeFazio: Extensible indexing: a framework for integrating domain-specific indexing schemes into oracle8i. *Proc. of IEEE Conference on Data Engineering.*, pp 91–100, 2000.
[27] B. Stefan, A.K. Daniel, H-P. Kriegel: The X-tree : An Index Structure for High-Dimensional Data, *Proc. of Very Large Databases Conference*., pp 28-39, 1996.
[28] T. Truong, T. Risch: Scalable Numerical Queries by Algebraic Inequality Transformations, *Proc. Database Systems for Advanced Applications (DASFAA)*, pp 95-109, 2014.
[29] T.Risch, V.Josifovski, and T.Katchaounov: Functional Data Integration in a Distributed Mediator System, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): Functional Approach to Data Management - Modeling, *Analyzing and Integrating Heterogeneous Data, Springer, ISBN 3-540-00375-4*, 2004.
[30] http://www.it.uu.se/research/group/udbl/mexima
[31] T. J. Lehman and M. J. Carey: A study of index structures for main memory database management systems," in *PVLDB '86*, 1986.
[32] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt et al.: Fast: Fast architecture sensitive tree search on modern cpus and gpus, *Proc. of SIGMOD*, 2010
[33] V. Leis, A. Kemper, and T. Neumann: The adaptive radix tree: Artful indexing for main-memory databases, *Proc. of IEEE Conference on Data Engineering*, 2013
[34] J. Rao, and K. A. Ross: Making B+-trees cache conscious in main memory. *ACM SIGMOD Record. Vol. 29. No. 2. ACM*, 2000.

Paper II

Paper II

Truong, Thanh and Risch, Tore.
Scalable Numerical Queries by Algebraic Inequality Transformations,
In Proceedings of 19th International Conference: Database Systems for Advanced Applications (DASFAA 2014),
Springer International Publishing, pages 95-109.
Bali, Indonesia, April 21-24, 2014.
ISBN: 978-3-319-05810-8,
DOI=10.1007/978-3-319-05810-8_7,
http://dx.doi.org/10.1007/978-3-319-05810-8_7

The paper is reformatted for typographic consistency.

Scalable Numerical Queries by Algebraic Inequality Transformations

Thanh Truong, Tore Risch

Department of Information Technology
Box 337, SE-751 05, Sweden
Uppsala University, Sweden
thanh.truong@it.uu.se, tore.risch@it.uu.se

Abstract
To enable historical analyses of logged data streams by SQL queries, the *Stream Log Analysis System (SLAS)* bulk loads data streams derived from sensor readings into a relational database system. SQL queries over such log data often involve numerical conditions containing inequalities, e.g. to find suspected deviations from normal behavior based on some function over measured sensor values. However, such queries are often slow to execute, because the query optimizer is unable to utilize ordered indexed attributes inside numerical conditions. In order to speed up the queries they need to be reformulated to utilize available indexes. In SLAS the query transformation algorithm *AQIT (Algebraic Query Inequality Transformation)* automatically transforms SQL queries involving a class of algebraic inequalities into more scalable SQL queries utilizing ordered indexes. The experimental results show that the queries execute substantially faster by a commercial DBMS when AQIT has been applied to preprocess them.

1 Introduction

We first introduce a real-world scenario application under investigation in the Smart Vortex project [15], which requires queries involving numerical expressions. A factory operates some machines. On each machine, there are a number of sensors to measure different physical properties, e.g. power consumption, pressure, temperature, etc. The sensors generate logs of measurements per machine that carry a time stamp *ts*, a machine identifier *m*, a sensor identifier *s*, a measured value *mv*, and a measurement class *mc* for the kind of measurements made by the sensor. Examples of measurement classes are oil pressures of hydraulic filters and pressures of gear pumps. The logs are analyzed by bulk loading them into a relational DBMS. To speed up performance

when analyzing sensors of the same kind on many different machines, there is one table for each *measurement class* of each kind of physical property. To avoid repetition of unchanged sensor readings, each measured value *mv* on machine *m* is associated with a valid time interval *bt* and *et* indicating the begin time and end time for *mv*, computed from the log time stamp *ts* when the data is bulk loaded. Hence, the measurement of class *mc=MC* on machines *m* will be stored in the table *measuresMC(m, s, bt, et, mv)*. These tables will contain large volumes of log data from many sensors of the same kind on different machines.

After the data streams have been loaded into *measuresMC()*, the user can issue offline historical queries to find errors on machines in the past by looking for abnormal values of *mv*. This often requires search conditions containing inequalities inside numerical expression. In our scenario, in order to improve the performance of inequality queries over *mv*, a B-tree index is added on each *measuresMC.mv*, denoted *idx(measuresMC.mv)*. The following are typical numerical query conditions on tables *measuresA*, and *measuresB* to identify faulty behaviors of machines:

- *C1*: Were the measurements of class A higher than a threshold $v_0 = 15.6$? We ex-press the condition as $C1(mv): mv > v_0$.
- *C2*: Were the measurements of class A higher than $r_1 = 300$ above the expected value $v_1 = 15.6$? We express the condition a $C2(mv): mv - v_1 > r_1$.
- *C3*: Were the measurements of class B outside the range $r_2 = 11$ from the ideal value $v_1 = 20$? We express the condition as $C3(mv): |mv - v_1| > r_2$.
- *C4*: Were the measurements of class B outside the range r_3=20% from $v_1 = 20$? We express the condition as $C4(mv): \left|\frac{1}{mv - v_1}\right| > r_3$.

The above conditions can be expressed in SQL. Relational databases can handle SQL query conditions of type *C1* efficiently, since there is an ordered index *idx(measuresA.mv)*. However, in *C2-C4* the inequalities are not defined directly over the attribute mv but through some numerical expressions, which makes the query optimizer not utilizing the indexes and hence the queries will execute slowly. We say that the indexes *idx(measuresA.mv)* and *idx(measuresB.mv)* are not exposed in *C2-C4*. To speed up such queries, the DBMS vendors recommend that the user reformulate them [11], which often requires rather deep knowledge of low-level index operations.

To transform automatically a class of queries involving inequality expressions into more efficient queries where indexes are exposed, we have developed the query transformation algorithm *AQIT (Algebraic Query Inequality Transformation)*. We show that AQIT substantially improves performance for

queries with conditions of type *C2-C4*, exemplified by analyzing logged abnormal behavior in our scenario. Without the proposed query transformations, the DBMS will do a full scan, not utilizing any index.

AQIT transforms queries with inequality conditions on single indexed attributes to utilize range search operations over B-tree indexes. In general, AQIT can transform inequality conditions of form $F(mv)\ \psi\ \varepsilon$, where *mv* is a variable bound to an indexed attribute *A*, *F(mv)* is an expression consisting of a combination of transformable functions T, currently $T \in \{+, -, /, *, power, sqrt, abs\}$, and ψ is an inequality comparison $\psi \in \{\leq, \geq, <, >\}$. AQIT tries to reformulate inequality conditions into equivalent conditions, $mv\ \psi'\ F'(\varepsilon)$ that makes the index on attribute A, *idx(A)* exposed to the query optimizer. AQIT has a strategy to automatically determine ψ' and $F'(\varepsilon)$. If AQIT fails to transform the condition, the original query is retained. For example, AQIT is currently not applicable on multivariable inequalities, which are subjects for future work.

In summary, our contributions are:

- We introduce the algebraic query transformation strategy AQIT on a class of numerical SQL queries. AQIT is transparent to the user and does not require manual reformulation of queries. We show that it substantially improves query performance.
- The prototype system SLAS (Stream Log Analysis System) implements AQIT as a SQL pre-processor to a relational DBMS. Thus, it can be used on top of any relational DBMS. Using SLAS we have evaluated the performance improvements of AQIT on log data from industrial equipment in use.

This paper is organized as follows. Section 2 discusses related work. Section 3 presents some typical SQL queries where AQIT improves performance. Section 4 gives an overview of SLAS and its functionality. Section 5 presents the AQIT algebraic transformation algorithm on inequality expressions. Section 6 evaluates the scalability of applying AQIT for a set of benchmark queries based on the scenario database, along with a discussion of the results. Section 7 gives conclusions and follow-up future work.

2 Related Work

The recommended solution to utilize an index in SQL queries involving arithmetic expressions is to manually reformulate the queries so that index access paths are exposed to the optimizer [5] [11] [13]. However, it may be difficult for the database user to do such reformulations since it requires knowledge about indexing, the internal structure of execution plans, and how query optimization works. There are a number of tools [11] [16], which point out inefficient SQL statements but do not automatically rewrite them. In contrast, AQIT

provides a transparent transformation strategy, which automatically transforms queries to expose indexes, when possible. If this is not possible, the query is kept intact.

Modern DBMSs such as Oracle, PostgreSQL, DB2, and SQL Server support *function indexes* [8] [10], which are indexes on the result of a complex function applied on row-attribute values. When an insertion or update happens, the DBMS computes the result of the function and stores the result in an index. The disadvantage of function indexes compared to the AQIT approach is that they are infeasible for ad hoc queries, since the function indexes have to be defined beforehand. In particular, function indexes are very expensive to build in a populated database, since the result of the expression must be computed for every row in the database. By contrast, AQIT does not require any pre-computations when data is loaded or inserted into the database. Therefore, AQIT makes the database updates more efficient, and simplifies database maintenance.

Computer algebra systems like Mathematica [1] and Maple [4] and constraints database systems [7] [9] also transform inequalities. However, those systems do not have knowledge about database indexes as AQIT. The current implementation is a DBMS independent SQL pre-processor that provides the index specific query rewritings.

FunctionDB [2] also uses an algebraic query processor. However, the purpose of FunctionDB is to enable queries to continuous functions represented in databases, and it provides no facilities to expose database indexes.

Extensible indexing [6] aims at providing scalable query execution for new kinds of data by introducing new kinds of indexes. However, it is up to the user to reformulate the queries to utilize a new index. By contrast, our approach provides a general mechanism for utilizing indexes in algebraic expressions, which complements extensible indexing. In the paper, we have shown how to expose B-tree indexes by algebraic rewrites. Other kinds of indexes would require other algebraic rules, which is a subject of future work.

3 Example Queries

A relational database that stores both meta-data and logged data from machines has the following three tables:

- *machine(m, mm)* represents meta-data about each machine installation identified by *m* where *mm* identifies the machine model. There is a secondary B-tree index on *mm*.
- *sensor(m, s, mc, ev, ad, rd)* stores meta-data about each sensor installation s on each machine *m*. To identify different kinds of measurements, e.g oil pressure, filter temperature etc., the sensors are classified by their *measurement class*, *mc*. Each sensor has some tolerance thresholds, which can

be an absolute or relative error deviation, *ad* or *rd*, from the expected value *ev*. There are secondary B-tree indexes on *ev*, *ad*, and *rd*.

- *measuresMC(m, s, bt, et, mv)* enables efficient analysis of the behavior of different kinds of measurements over many machine installations over time. The table stores measurements *mv* of class *MC* for sensor installations identified by machine *m* and sensor s in valid time interval *[bt,et)*. By storing *bt* and *et* temporal interval overlaps can be easily expressed in SQL [3] [14]. There are B-tree indexes on *bt*, *et*, and *mv*.

We use the abnormality thresholds *@thA* for queries determining deviations in table *measuresA*, *@thB* for queries determining absolute deviation in table *measuresB*, and *@thRB* for queries determining relative deviation in table *measuresB*. We shall discuss these thresholds in Section 6 in details.

The following queries *Q1, Q2*, and *Q3* identify abnormalities:

Query Q1 finds when and on what machines, the pressure reading of class A was higher than *@thA* from its expected value:

```
1 SELECT    va.m, va.bt, va.et
2 FROM      measures A va, sensor s
3 WHERE     va.m = s.m AND va.s = s.s AND va.mv > s.ev + @thA.
```

AQIT has no impact for query *Q1* since the index *idx(measuresA.mv)* is already exposed.

Query Q2 identifies abnormal behaviors based on absolute deviations: When and for what machines did the pressure reading of class B deviate more than *@thB* from its expected value? AQIT translates the query into the following SQL query *T2*:

```
 Q2:
1 SELECT  vb.m, vb.bt, vb.et
2 FROM    measuresB vb, sensor s
3 WHERE vb.m =s.m              AND
          vb.s=s.s             AND
4         abs(vb.mv - s.ev) > @thB
5
```

```
T2:
SELECT vb.m, vb.bt, vb.et
FROM   measuresB vb, sensor s
WHERE vb.m=vb.m              AND
        vb.s=s.s             AND
       ((vb.mv > @thB + s.ev) OR
        (vb.mv < - @thB + s.ev))
```

In *T2* lines 4-5 expose the ordered index *idx(measuresB.mv)*.

Query Q3 identifies two different abnormal behaviors of the same machine at the same time based on two different measurement classes and relative deviations: When and for which machines were the pressure readings of class A higher than *@thA* from its expected value at the same time as the pressure reading of class B were deviating *@thRB* % from its expected value? After the AQIT transformation *Q3* becomes *T3*:

```
Q3:
1 SELECT   va.m,greaest(va.bt,vb.bt)
2                 least(va.et, vb.et)
 3 FROM measuresA va,measuresB vb,
4            sensor sa, sensor sb
5 WHERE   va.m=sa.m                AND
6               va.s=sa.s                AND
7               vb.m=sb.m               AND
8               vb.s=sb.s                AND
9               va.m=vb.m               AND
10.             va.bt<=vb.et             AND
11               va.et>=vb.bt            AND
12     va.mv - sa.ev > @thA          AND
13.    abs((vb.mvsb.ev)/sb.ev)>@thRB
14
15
16
```

```
T3:
SELECT  va.m,greatest(va.bt, vb.bt),
              least(va.et, vb.et)
FROM measuresA va, measuresB vb,
          sensor sa, sensor sb
WHERE  .va.m=sa.m                AND
              va.s=sa.s                  AND
              vb.m=sb.m                 AND
              vb.s=sb.s                  AND
             va.m =vb.m                 AND
             va.bt<=vb.et               AND
             va.et>=vb.bt               AND
             va.mv >@thA + sa.ev     AND
          ((vb.mv>(1+@thRB)*sb.ev
                AND  sb.ev >0)
           OR  (vb.mv<(1+@thRB)*sb.ev
                AND  sb.ev<0)
           OR  (vb.mv<(-@thRB+1)*sb.ev
                AND  sb.ev>0)
           OR  (vb.mv>(-@thRB+1)*sb.ev
                AND  sb.ev<0)
```

Lines 10-11 in *Q3* selects temporal overlap of the time interval *[va.bt, va.et]* with *[vb.bt, vb.et]*. The functions *greatest(va.bt, vb.bt)* and *least(va.et, vb.et)* return the maximum and minimum values of their two arguments, respectively. These functions are supported by Oracle, MySQL, DB2 and PostgreSQL but not by SQL Server [14]. Therefore, we defined *greatest(x, y)* and *least(x, y)* as user defined functions for SQL Server.

In *T3* line 13 exposes *idx(measuresA.mv)* and lines 14-16 expose *idx(measuresB.mv)*.

4 Stream log analysis system (SLAS)

Figure 1 illustrates the architecture of SLAS. It uses a datastream management system, DSMS, to process raw streams of measurements from different *machines*. The log writer receives from the DSMS a stream of tuples with format *(mc, m, s, ts, mv)* specified as a continuous query. The *log writer* produces once per system determined time interval a CSV file of tuples *(m, s, bt, et, mv)* for each measurement class *mc* to be loaded into the corresponding table *measuresMC*. Here, *[bt,et)* is the valid time interval for *mv*, computed from *ts*. When the log writer has written a CSV file it notifies the *log loader* for measurement class *mc*, which bulk loads the new log file rows into the corresponding measurement log table *measuresMC*.

In order to limit and customize the amount of log data stored in the DBMS the *log deleter* continuously deletes log data from the DBMS according to user specified configuration parameters.

The user can analyze the stored data streams by issuing historical SQL queries over loaded log data through the *AQIT processor*. The strategy used by AQIT to improve numerical SQL queries is the focus of this paper.

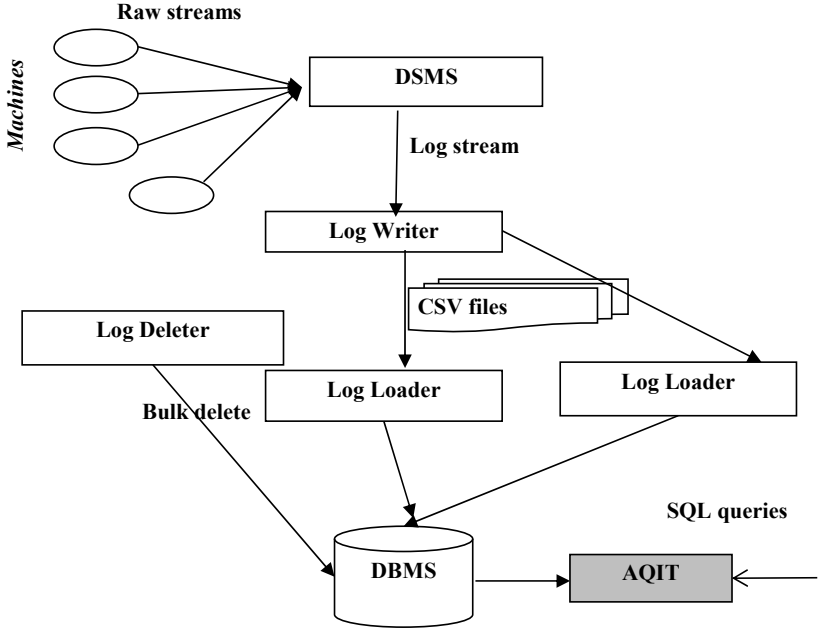

Figure 1. Stream Log Analyse System

Figure 2 illustrates the query processing of AQIT. An SQL query is first parsed into an internal query in a Datalog dialect [12]. The *AQIT rewriter* transforms the Datalog query into an equivalent *index exposed query*. The *SQL Generator* transforms the index exposed Datalog query into an equivalent *shipped SQL* query sent to the back-end DBSM through JDBC for optimization and evaluation.

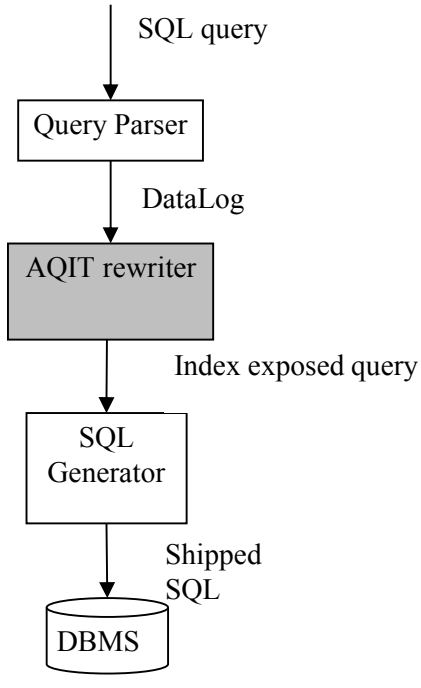

Figure 2. AQIT Preprocessor

5 Algebraic Query Inequality Transformation

To explain the AQIT transformations we need the following definitions:

Definition 1: A source predicate *r(…)* of a query is a predicate that represents access to a relation named *r*.
Definition 2: If there is a B-tree index *idx(r.a)* on some attribute *a* of a source predicate *r(…a…)*, we say that *r* is an indexed predicate.
Definition 3: If there is an occurrence of a variable *v* representing *idx(r.a)* in an indexed predicate *r(…v…)* of a query, we say that *v* is an indexed variable in the query.
Definition 4: If there is an inequality ψ(v,x) where *v* is an indexed variable, we say that the indexed variable *v* is exposed by the inequality predicate ψ.

In this section, we use *Q1* and *Q2* to show how AQIT works. First the parser translates *Q1*, and *Q2* into the following Datalog queries *DQ1* and *DQ2*:

```
DQ1(m,bt,et)←
  measuresA(m,s,bt,et,mv)      AND
  sensor(m,s,_,_,ev,_,_)       AND
```

```
DQ2(m,bt,et)←
  measuresA(m,s,bt,et,mv)      AND
  sensor(m,s,_,_,ev,_,_)       AND
  v1 = mv – ev AND
```

v1 = ev + @thA mv > v1	AND	v2 = abs(v1) AND v2 > @thA

Here, the source predicates *measuresA(m,s,bt,et,mv)* and *measuresB(m,s,bt.et,mv)* represent relational tables for two different measurement classes. For both tables there is a B-tree index on *mv* to speed up comparison and proximity queries, and therefore *measuresA()* and *measuresB()* are indexed predicates and the variable *mv* is an indexed variable. In *Q1*, the index *idx(measuresA.mv)* is already exposed because there is a comparison between *measuresA.mv* and variable *v1*, so AQIT will have no effect.

In *Q2*, the index *idx(measuresB.mv)* is not exposed by the inequality predicate *v2 > @thB* since the inequality is defined over a variable *v2*, which is not bound to the indexed attribute *measuresB.mv*. Here AQIT transforms the predicates to expose the index *idx(measuresB.mv)* so in *T2 idx(measuresB.mv)* is exposed in both OR branches.

5.1 AQIT Overview

The AQIT algorithm takes a Datalog predicate as input and returns another semantically equivalent predicate that exposes one or several indexes, if possible. AQIT is a *fixpoint* algorithm that iteratively transforms the predicate to expose hidden indexes until no further indexes can be exposed. The full pseudo code can be found in [17].

The transformations are made iteratively by the function *transform_pred()* in Listing 1. At each iteration, it invokes three functions, called *chain()*, *expose()*, and *substitute()*. *chain()* finds some path between an indexed variable and an inequality predicate that can be exposed, *expose()* transforms the found path so that the index becomes exposed, and *substitute()* replaces the terms in the original predicate with the new path.

Listing 1 Transform predicate

```
function transform_pred(pred):
input:   A predicate pred
output:  A transformed predicate or the original
pred
begin
  if pred is disjunctive then
    set failure = false
    /*result list of transformed branches*/
    set resl = null
    do /*transform each branch*/
      set b = the first not transformed branch in
      pred
      set nb = transform_pred(b)/*new branch*/
      if  nb not null then add nb to resl
      else  set failure = true
```

```
    until failure or no more branch of pred to try
     if not failure then
        /*return a disjunction from resl*/
         return  orify(resl)
     end if
  else if pred is conjunctive then
    set path = chain(pred)
    if path not null then
      set exposedpath = expose(path)
      if  exposedpath not null then
           return substitute(pred, path, exposed-
      path)
      end if
    end if
  end if
  return pred
end
```

Chain: The *chain()* algorithm tries to produce a path of predicates that links one indexed variable with one inequality predicate. If there are multiple indexed variables a simple heuristic is applied. It sorts the indexed variables decreasingly based on selectivities of the indexed attributes, which can be obtained first from the backend DBMS. The path must be a conjunction of *transformable terms* that represent expressions transformable by AQIT. Each transformable term in a path has a single common variable with adjacent terms. Such a chain of connected predicates is called an *index inequality path (IIP)*. Query *DQ2* has the following IIP called *Q2-IIP* from the indexed variable *mv* to the inequality *v2 > @thB*, where the functions '–' and *'abs'* are transformable:

Q2-IIP: measuresB(m, s, bt, et, mv) → v1=mv - ev → v2=abs(v1) → v2>@thB.

In this case *Q2-IIP* is the only possible IIP, since there are no other unexposed index variables in the query after *Q2-IIP* has been formed. The following graph illustrates *Q2-IIP*, where nodes represent predicates and arcs represent the common variable of adjacent nodes:

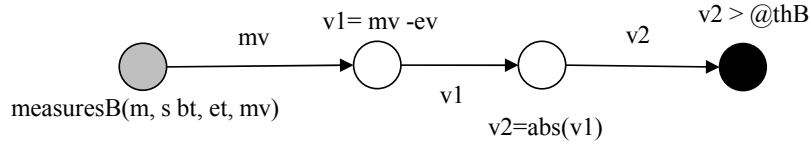

Figure 3. Q2-IIP

An IIP starts with an indexed *origin predicate* and ends with an inequality *destination predicate*. The origin node in an IIP is always an indexed predicate where the outgoing arc represents one of the indexed variables.

chain() is a backtracking algorithm trying to extend partial IIPs consisting of transformable predicates from an indexed variable until some inequality predicate is reached, in which case the IIP is *complete*. The algorithm will try to find one IIP per indexed variable. If there are several common variables between transformable terms, *chain()* will try each of them until a complete IIP is found. If there are other not yet exposed ordered indexes for some source predicates, the other IIPs may be discovered later in the top level fixpoint iteration.

The *chain()* procedure successively extends the IIP by choosing new transformable predicates *q* not on the partial IIP such that one of *q*'s arguments is the variable of the right-most outgoing arc (*mv* in our case) of the partial IIP. For *DQ2* only the predicate *v1=mv-ev* can be chosen, since *mv* is the outgoing arc variable and '–' is the only transformable predicate in *DQ2* where *mv* is an argument. When there are several transformable predicates, *chain()* will try each of them in turn until the IIP is complete or the transformation fails.

An IIP through a disjunction is treated as a disjunction of IIPs with one partial IIP per disjunct in Listing 1. In this case, the index is considered utilized if all partial IIPs are complete.

Expose: The *expose()* procedure is applied on each complete IIP in order to expose the indexed variable. The indexed variable is already exposed if there are no intermediate nodes between the origin node and the destination node in the IIP. For example, the IIP for Q1 is *Q1-IIP: measuresA(m, s, bt, et, mv)* → *mv>v1*. Here the indexed variable *mv* is already exposed to the inequality. Therefore, in this case *expose()* returns the input predicate unchanged.

The idea of *expose()* is to shorten the IIP until the index variable is exposed by iteratively combining the two last nodes through the algebraic rules in Table 4 into larger destination nodes while keeping the IIP complete. To keep the IIP complete the incoming variable of the last node must participate in some inequality predicate. As an example, the two last nodes in *Q2-IIP* in *Figure 3* are combined into a disjunction in *Figure 4*. Here the following algebraic rule is applied: *R10:* $|x| > y \Rightarrow (x > y \vee x < -y)$.

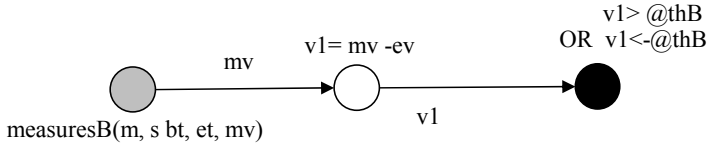

Figure 4. Q2-IIP after the first reduction

The algebraic rule *R10* exposes a variable *x* hidden inside *abs()* of an inequality. The following table shows how *R10* is applied on the two last nodes in *Figure 3* to form the new predicate in *Figure 4*.

Table 1 Applying R10

Before	After
v2 = abs(v1) AND v2 > @thB	(v1 > @thB OR v1 < -@thB)

By iteratively exposing each variable on the IIP, the indexed variable (and the index) will possibly be exposed. For example, *Q2-IIP* in *Figure 4* is reduced into *Figure 5* by applying the algebraic rules *R3:* $x - y > z \Rightarrow x > y + z$ and *R4:* $x - y < z \Rightarrow x < y + z$.

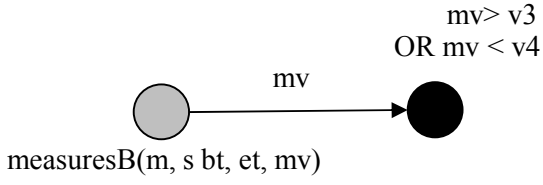

Figure 5. Q2-IIP after the second reduction

The following two tables show how rules R3 and R4 have been applied:

Table 2 Applying R3

Before	After
v1 = mv –ev AND v1 > @thB	v3 = ev + @thB AND mv > v3

Table 3 Applying R4

Before	After
v1 = mv –ev AND v1 < -@thB	v4 = ev -@thB AND mv < v4

The new variables *v3* and *v4* are created when applying the rewrite rules to hold intermediate values.

In *Figure 5* there are no more intermediate nodes and the index *idx(measuresB.mv)* is exposed, so *expose()* succeeds.

expose() may fail if there is no applicable algebraic rule when trying to combine some two last nodes, in which case the *chain()* procedure will be run again to find a next possible IIP until as many indexed variables as possible are exposed.

Substitute: When *expose()* has succeeded, *substitute()* updates the original predicate by replacing all predicates in the original IIP, except its origin, with

the new destination predicate in the transformed IIP [17]. For *Q2* this will produce the final transformed Datalog query:

```
DQ2(m,bt,et) ← measuresB(m,s,bt,et,mv)  AND
        sensor(m, s, _,_,ev,_,_)          AND
        v3  =  ev + @thB                  AND
        v4 = ev -@thB                     AND
       (mv < v4 OR mv >  v3)
```

The Datalog query is the translated by the SQL Generator into SQL query *T2*.

5.2 Inequality Transformation Rules

Table 4 the algebraic rewrite rules currently used by AQIT are listed. The list can be extended for new kinds of algebraic index exposures. In the rules, x, y, and z are variables and ψ denotes any of the inequality comparisons $\geq, \leq, <$, or $>$, while ψ^{-1} denotes the inverse of ψ. CP denotes a positive constant ($CP > 0$), while CN denotes a negative constant ($CN < 0$). Each rule shows how to expose the variable x hidden inside an algebraic expression to some inequality expression.

Table 4 Algebraic inequality transformations

R1	$(x + y)\ \psi z$	$\Leftrightarrow$	$x\ \psi (z - y)$
R2	$(y + x)\ \psi z$	$\Leftrightarrow$	$x\ \psi (z - y)$
R3	$(x - y)\ \psi z$	$\Leftrightarrow$	$x\ \psi (z + y)$
R4	$(y - x)\ \psi z$	$\Leftrightarrow$	$x\ \psi^{-1} (y - z)$
R5	$(x * CP)\ \psi z$	$\Leftrightarrow$	$(x\ \psi z/CP)$
R6	$(x * CN)\ \psi z$	$\Leftrightarrow$	$(x\ \psi^{-1}\ z/CN)$
R7	$x/y\ \psi z \wedge y != 0$	$\Leftrightarrow$	$(x\ \psi y*z \wedge y > 0) \vee (x\ \psi^{-1} z*y \wedge y < 0)$
R8	$y/x\ \psi z$	$\Leftrightarrow$	$(y/z\ \psi x \wedge x*z > 0)$ $\vee (y/z\ \psi^{1} x \wedge x*z < 0)$ $\vee\ (y = 0 \wedge 0\ \psi z)$
R9	$\lvert x\rvert \leq y$	$\Leftrightarrow$	$(x \leq y \wedge x \geq -y)$
R10	$\lvert x\rvert \geq y$	$\Leftrightarrow$	$(x \geq y \vee x \leq -y)$

R11	$\sqrt[x]{}\ \psi y$	$\Leftrightarrow$	$x\ \psi y^2$
R12	$x^y\ \psi z$	$\Leftrightarrow$	$(x\ \psi\ \sqrt[y]{z} \wedge y > 0)$ $\vee\ (x\ \psi^{-1}\ \sqrt[y]{z} \wedge y < 0)$ $\vee\ (x\ \psi z \wedge y = 0)$
R13	$(x+y)/x\ \psi\ z$	$\Leftrightarrow$	$(1+y/x)\ \psi\ z$
R14	$\lvert(x - y) / y\rvert > z$	$\Leftrightarrow$	$(x > (z + 1) * y \wedge y > 0)\ \vee (x < (z + 1) * y \wedge y < 0)$ $\vee (x < (- z + 1) * y \wedge y > 0) \vee (x > (- z + 1) * y \wedge y < 0)$

6 Experiment Evaluation

We experimentally compared the performance of a number of typical queries finding different kinds of abnormalities based on 16000 real log files from two industrial machines. To simulate data streams from a large number of machines, 8000 log files were constructed by pairing the real log files two-by-two and then time-stamping their events based on off-sets from their first time-stamps. This produces realistic data logs and enables scaling the data volume by using an increasing number of log files.

6.1 Setup

To investigate the impact of AQIT on the query execution time, we run the SLAS system with SQL Server™ 2008 R2 as DBMS on a separate server node. The DBMS was running under Windows Server 2008 R2 Enterprise on 8 processors of AMD Opteron ™ Processor 6128, 2.00 GHz CPU and 16GB RAM. The experiments were con-ducted with and without AQIT preprocessing.

6.2 Data

Figure 6 is a scatter plot from a small sampled time interval of pressure readings of class A. This is an example of an asymmetric measurement series with an initial warm-up period of 581.1 seconds.

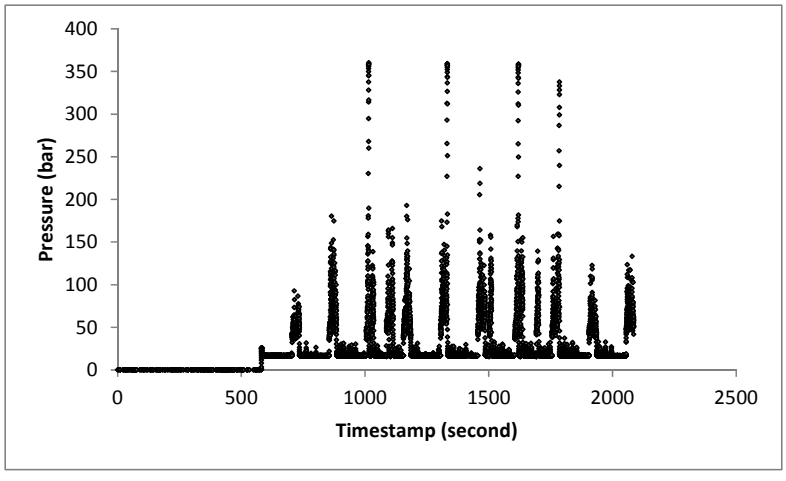

Figure 6. Pressure measured of class A

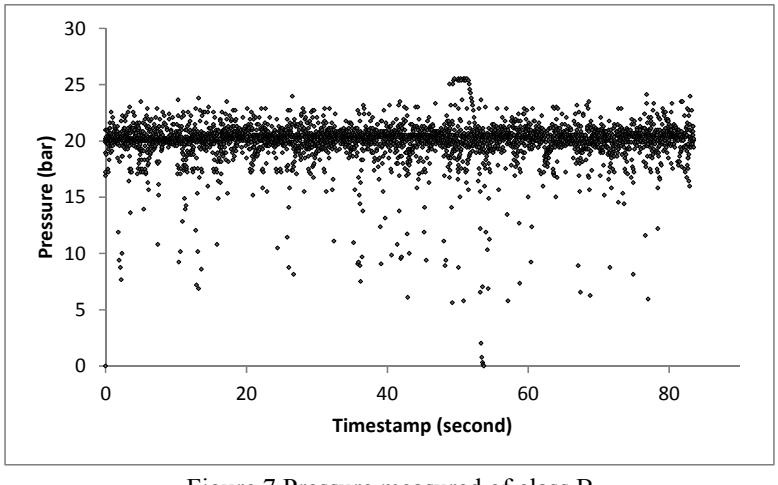

Figure 7 Pressure measured of class B

The abnormal behavior in this case is that the measured values are larger than the expected value (17.02) within a threshold. When the deviation threshold is 0 all measurements are abnormal, while when the threshold is 359.44 no measurements are abnormal. For example, *Q1* finds when a sensor reading of class A is abnormal based on threshold *@thA* that can be varied.

Figure 7 plots pressure readings of measurements of class B over a small sampled time interval. Here the abnormality is determined by threshold *@thB*, indicating absolute differences between a reading and the expected value (20.0), as specified in *Q2*. When the threshold is 0 all measurements are abnormal, while when the threshold is 20.0 no measurements are abnormal.

In addition, the abnormality of measurements of class B is determined by threshold *@thRB* as in *Q3*, indicating relative difference between a reading

and the expected value. When the relative deviation threshold is 0%, no measurements are abnormal, while when the threshold is 100% all measurements are abnormal.

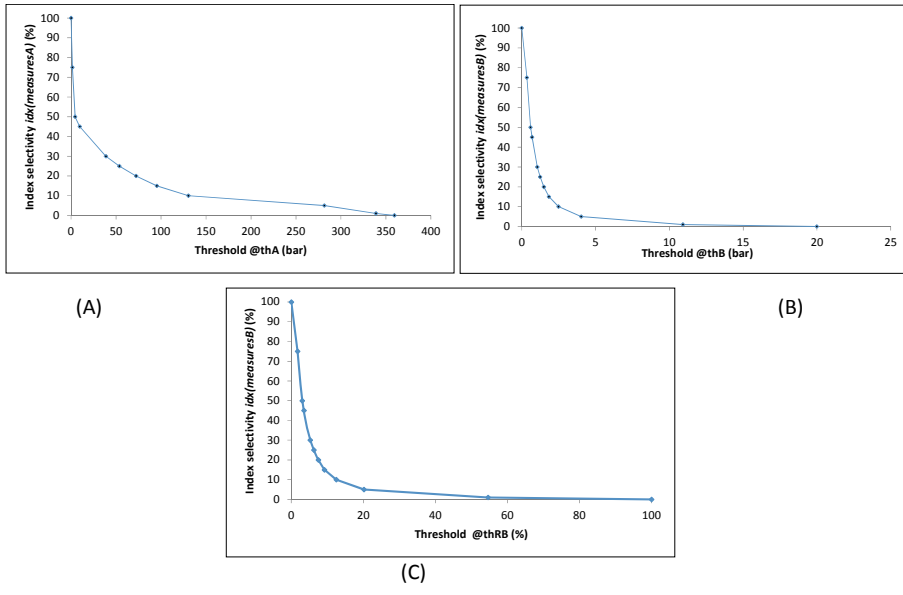

Figure 8. Thresholds and selectivity mappings

6.3 Benchmark queries

We measured the impact of index utilization exposed by AQIT by varying the abnormality thresholds *@thA* for queries determining deviations in *measuresA*, and the thresholds *@thB* and *@thRB* for queries determining deviations in *measuresB*. The larger the threshold values the fewer abnormalities will be detected. We also defined three other benchmark queries *Q4*, *Q5*, and *Q6*. All the detailed SQL and Datalog formulations before and after AQIT for the benchmark queries are listed in [18].

- *Q4* identifies when the pressure readings of class B deviates more than *@thB* for the machines in a list *machine-models* of varying length. Here, if a query spans many machine models the impact of AQIT should decrease since many different index keys are accessed.

```
Q4:
SELECT   vb.m, vb.bt, vb.et
FROM     measuresB vb, sensor s,
         machine ma
WHERE  vb.m = s.m                      AND
       va.s=s.s                        AND
       vb.m = ma.m                     AND
       ma.mm in@machine-models AND
       abs(vb.mv - s.ev) > @thB
```

```
T4:
SELECT vb.m, vb.bt, vb.et
FROM   measuresB vb, sensor s,
       machine ma
WHERE vb.m = s.m                AND
       va.s=s.s                 AND
       vb.m = ma.m              AND
ma.mm in @machine-models        AND
       (vb.mv > @thB + s.ev
       OR vb.mv < - @thB + s.ev)
```

- *Q5* identifies when the pressure reading of class B deviates more than *@thB* for two specific machine models using a temporal join. The query involves numerical expressions over two indexed variables, which are both exposed by AQIT. See [18] for details.
- Query *Q6* is a complex query that identifies a sequence of two different abnormal behaviors of the same machine happening within a given time interval, based on two different measurement classes. On what machines the pressure readings of class B were out-of-bounds more than *@thB* within 5 seconds after the pressure readings of class A were higher than *@thA* from the expected value. Here, both *idx(measuresA.mv)* and *idx(measuresB.mv)* are exposed by AQIT. See [18] for de-tails.

6.4 Performance measurements

To measure performance based on different selectivities of indexed attributes, in *Figure 8* we map the threshold values to the corresponding measured index selectivities of *idx(measuresA.mv)* and *idx(measuresB.mv)*. 100% of the abnormalities are detected when any of the thresholds is 0 and thresholds above the maximum threshold values *(@thA=359.44*, *@thB=20.0*, and *@thRB=100%*) detect 0% abnormalities.

Experiment A varies the database size from 5GB to 25GB while keeping the selectivities (abnormality percentages) at 5% and a list of three different machine models in *Q4*.

Figure 9 shows the performance of example queries *Q2*, *Q3*, *Q4*, *Q5*, and *Q6* (without AQIT) and their corresponding transformed queries *T2*, *T3*, *T4*, *T5*, and *T6* (with AQIT) when varying the database size from 5 to 25 GB. The original queries without AQIT are substantially slower since no indexes are

exposed and the DBMS will do full scans, while for transformed queries the DBMS backend can utilize the exposed indexes.

Experiment B varies index selectivities of *idx(measuresA.mv)* and *idx(measuresB.mv)* while keeping the database size at 25 GB and selecting three different machine models in *Q4*. We varied the index selectivities from 0% to 100%. *Figure 10* presents execution times of the all benchmark queries with and without AQIT.

Without AQIT, the execution times for *Q2* - *Q6* stay constant when varying the selectivity since no index is utilized and the database tables are fully scanned.

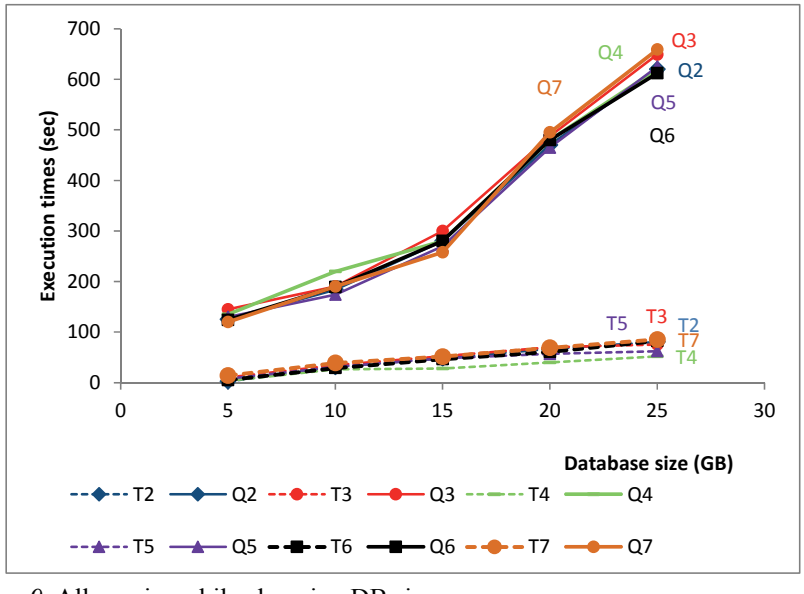

Figure 9. All queries while changing DB size

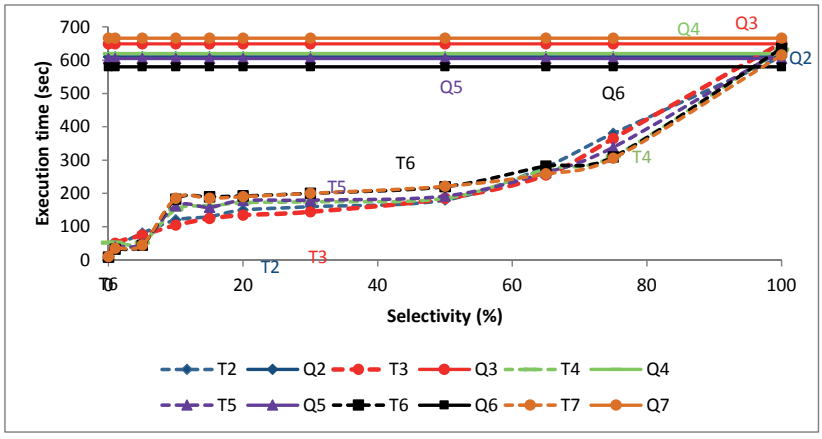

Figure 10. All queries while changing selectivities

Figure 10 shows that AQIT has more effect the lower the selectivity, since index scans are more effective for selective queries. For non-selective queries the indexes are not useful. When all rows are selected the AQIT transformed queries are slightly slower than original ones; the reason being that they are more complex. In general AQIT does not make the queries significantly slower.

Experiment C varies the number machine models in *Q4* from 0 to 25 while keeping the database size at 25 GB and the selectivity at 5%, as illustrated by *Figure 11*. It shows that when the list is small the transformed query *T4* scales much better than the original query *Q4*. However, when the list of machine increases, *T4* is getting slower. The reason is that the index *idx(measuresB.mv)* is accessed once per machine model, which is faster for fewer models.

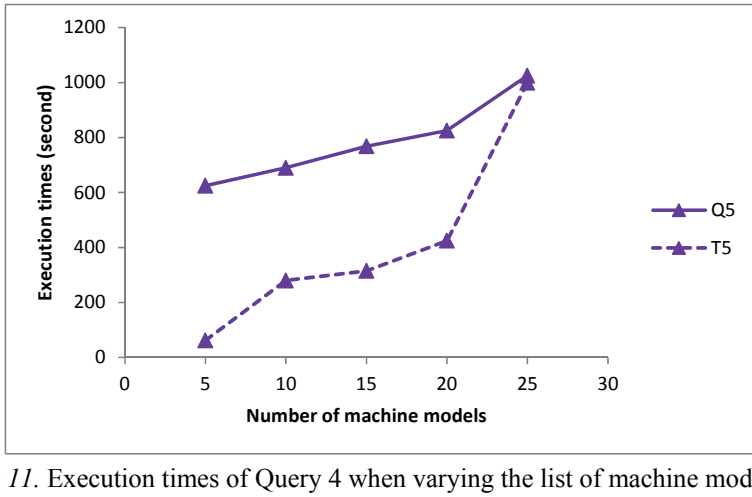

Figure 11. Execution times of Query 4 when varying the list of machine models

The experiments A, B, and C show that AQIT improves the performance of the benchmark queries substantially and will never make the queries significantly slower. In general AQIT exposes hidden indexes while the backend DBMS decides whether to utilize them or not.

7 Conclusion & Future work

In order to improve the performance of queries involving complex inequality expression, we investigated and introduced the general algebraic query transformation algorithm AQIT. It transforms a class of SQL queries so that indexes hidden inside numerical expressions are exposed to the back-end query optimizer.

From experiments, which were made on a benchmark consisting of real log data streams from industrial machines, we showed that the AQIT query transformation substantially improves query execution performance.

We presented our general system architecture for analyzing logged data streams, based on bulk loading data streams into a relational database. Importantly, looking for abnormal behavior of logged data streams often requires inequality search conditions and AQIT was shown to improve the performance of such queries.

We conclude that AQIT improves substantially the query performance by exposing indexes without making the queries significantly slower.

Since inequality conditions also appear in spatial queries we plan to extend AQIT to support transforming spatial query conditions as well user defined indexing. We also acknowledge that the inequality conditions could be more complex with multiple variables and complex mathematical expression, which will require other algebraic rules.

8 Acknowledgements

The work was supported by the Smart Vortex EU project [15].

9 References

[1] Andrew, G.L. Cain, S. Crum, T.D. Morley, Calculus Projects Using Mathematica. *McGraw Hill (1996).*

[2] T.Arvind and M. Samuel. Querying continuous functions in a database system, *Proc. of SIGMOD 2008*, Vancouver, Canada, 791 – 804.

[3] J.Celko. SQL for Smarties (Fourth Edition): Advanced SQL Programming, ISBN: 978-0-12-382022-8, 2011.

[4] C.M. Chang. Mathematical Analysis in Engineering, *Cambridge University Press* (1994)
[5] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zaït, and M. Ziauddin. Automatic SQL Tuning in Oracle 10g, *Proc. of Very Large Database 2004*, Toronto, Canada, 1098-1109
[6] M.Y. Eltabakh, R. Eltarras, W.G. Aref, Space-Partitioning Trees in PostgreSQL: Realization and Performance, *Proc. of ICDE 2006*, Atlanta, Georgia, USA, 100 – 112.
[7] K.Gabriel, L.Leonid, and P.Jan: Constraint Databases. ISBN 978-3-642-08542-0, *Springer Berlin Heidelberg*, 21-54.
[8] J.Gray, A. Szalay, and G. Fekete. Using Table Valued Functions in SQL Server 2005 to Implement a Spatial Data Library, *Technical Report, Microsoft Research Advanced Technology Division 2005*.
[9] S. Grumbach, P. Rigaux, and L. Segoufin. The DEDALE system for complex spatial queries, *Proc. of SIGMOD 1998*, Seattle, Washington, 213-224.
[10] D.J-H. Hwang. Function-Based Indexing for Object-Oriented Databases, PhD Thesis, Massachusetts Institute of Technology, 1994, 26-32.
[11] Leccotech. LECCOTECH Performance Optimization Solution for Oracle, *White Paper, http://www.leccotech.com/, 2003*.
[12] W. Litwin and T. Risch. Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 6*, December 1992.
[13] Oracle Inc. Query Optimization in Oracle Database 10g Release 2. *An Oracle White Paper, June 2005*.
[14] R.T. Snodgrass. Developing Time-Oriented Database Applications in SQL , *Morgan Kaufmann Publishers, Inc., San Francisco*, ISBN 1-55860-436-7, 1999
[15] Smart Vortex Project - *http://www.smartvortex.eu/*
[16] Quest Software. Quest Central for Oracle: SQLab Vision, *http://www.quest.com, 2003*.
[17] http://www.it.uu.se/research/group/udbl/aqit/PseudoCode.pdf
[18] http://www.it.uu.se/research/group/udbl/aqit/Benchmark_queries.pdf

Paper III

Paper III

Minpeng Zhu, Silvia Stefanova, Thanh Truong, and Tore Risch.
Scalable Numerical SPARQL Queries over Relational Databases,
In Proceedings of the Workshops of the EDBT/ICDT, 2014 Joint Conference, pages 257—262.
Athens, Greece, March 28, 2014.
http://ceur-ws.org/Vol-1133/paper-41.pdf,

Scalable Numerical SPARQL Queries over Relational Databases

Minpeng Zhu, Silvia Stefanova, Thanh Truong, Tore Risch
Department of Information Technology, Uppsala University, Sweden
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 back-end 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:

1. We propose a table driven approach to translate numerical domain calculus operators into numerical SQL tuple calculus operators.
2. We present the NUMTranslator algorithm that extracts numerical ObjectLog expressions and translates them into corresponding numerical SQL expressions.
3. 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.

4. 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.

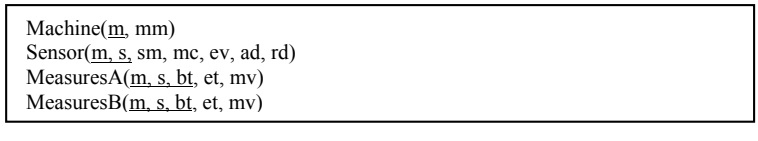

Figure 1 RDB schema for log data

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

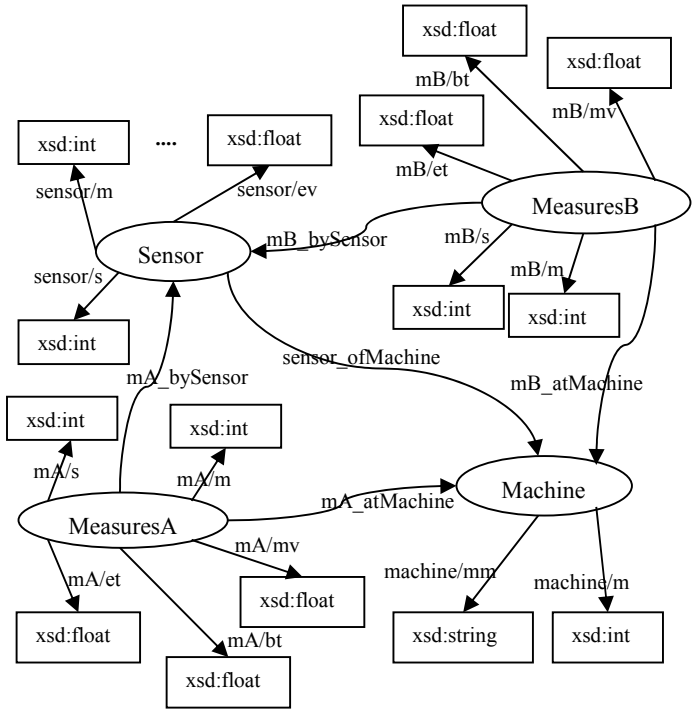

Figure 2. RDF grahp of the RDF view for the example RDB

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.

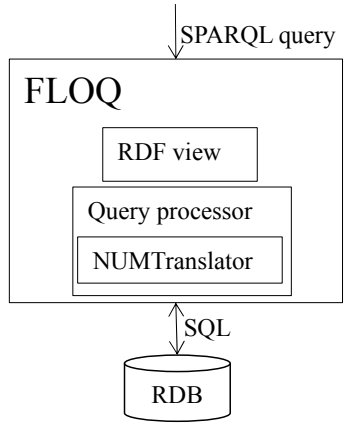

Figure 3. FLOQ query processor

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* $R(a_1, a_2, \ldots, a_n)$ that represent the content of each referenced relational database table R where the tuple $(a_1, \ldots, a_n)$ 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*.

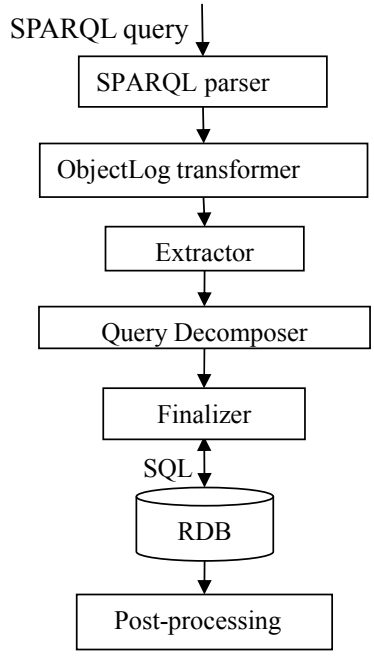

Figure 4. Query processing steps

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:

```
                    (m, bt, et)
                         ↑
γ sql(ds, "SELECT m.m, bt, et FROM MeasuresA
m, SENSOR s WHERE m.mv > s.ev + @thA AND
m.m=s.m AND m.s=s.s", (m, bt, et))
```

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:

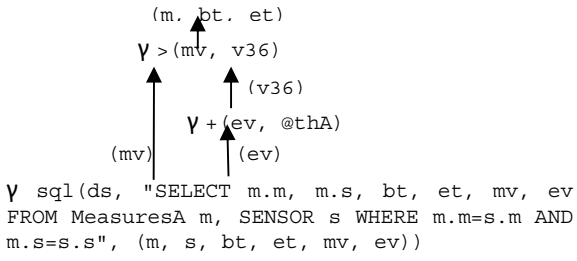

Figure 7. Execution plan *P2* without NUMTranslator

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, *Figure 9*, and *Figure 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.

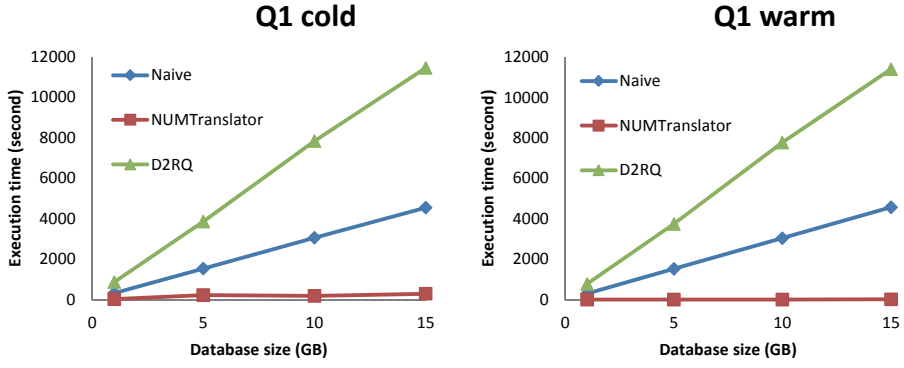

Figure 8. Execution times for Q1

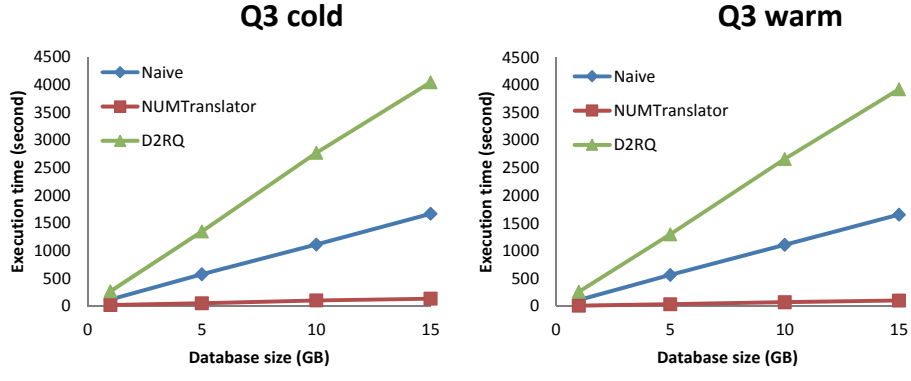

Figure 9. Execution times for Q3

Figure 8 and *Figure 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.

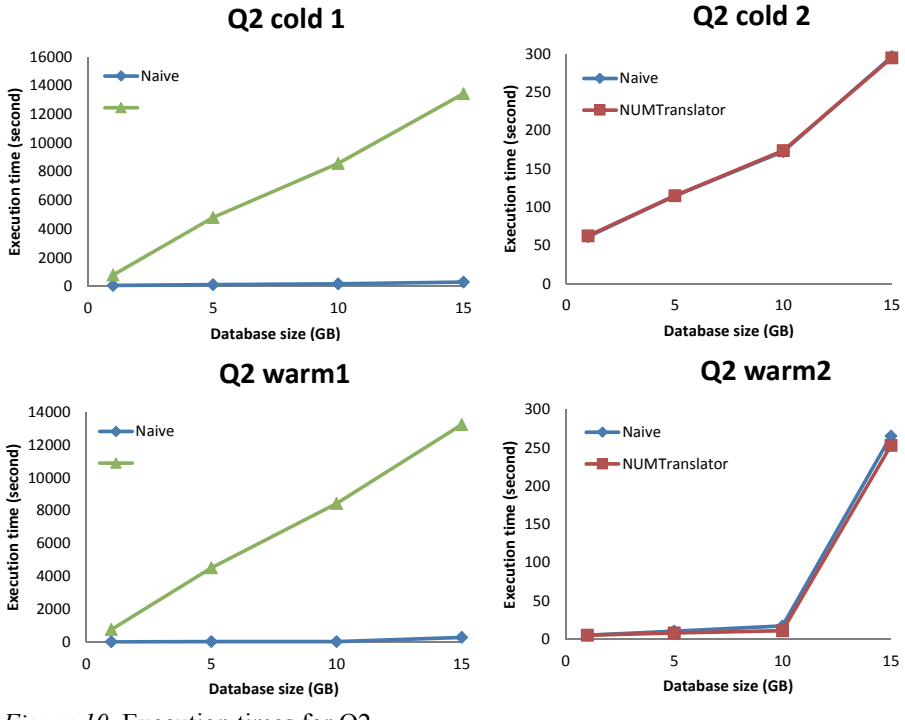

Figure 10. Execution times for Q2

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.

ACKNOWLEDGMENT

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

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. *7th 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 IV

Paper IV

Sobhan Badiozamany, Lars Melander, Thanh Truong, Xu Cheng, and Tore Risch. 2013. Grand challenge: implementation by frequently emitting parallel windows and user-defined aggregate functions. In Proceedings of the 7th ACM international conference on Distributed event-based systems (DEBS '13).
ACM, New York, NY, USA, 325-330.
DOI=10.1145/2488222.2488284
http://dx.doi.org/10.1145/2488222.2488284

Grand Challenge: Implementation by Frequently Emitting Parallel Windows and User-Defined Aggregate Functions

Sobhan Badiozamany, Lars Melander, Thanh Truong, Cheng Xu, Tore Risch
Department of Information Technology
Box 337, SE-751 05, Sweden
Uppsala University, Sweden
{firstname.lastname }@it.uu.se

ABSTRACT
Our implementation of the DEBS 2013 Challenge is based on a scalable, parallel, and extensible DSMS, which is capable of processing general continuous queries over high volume data streams with low delays. A mechanism to provide user defined incremental aggregate functions over sliding windows of data streams provide real-time processing by emitting results continuously with low delays. To further eliminate delays caused by time critical operations, the system is extensible so that functions can be easily written in some external programming language. The query language provides user defined parallelization primitives where the user can express queries specifying how high volume data streams are split and reduced into lower volume parallel data streams. This enables expensive queries over data streams to be executed in parallel based on application knowledge. Our OS-independent implementation was tested on several computers and achieves the real-time requirement of the challenge on a regular PC.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – *Parallel databases*, *Query processing*

Keywords
Parallel data stream processing; continuous queries; spatial-temporal window operators.

1 Introduction

Monitoring a soccer game requires a system than can process, in real-time, large volumes of data to dynamically determine physical properties as they appear. This requires a system having the following properties:

- To keep up with the very high data flow the system must deliver high throughput while processing expensive computations over high volume data.
- Response in real-time requires continuous delivery of query results with low latency.
- Continuous identification of physical phenomena, such as moving balls and players, requires complex spatio-temporal algebraic computations over windows.

Our EPIC (Extensible, Parallel, Incremental, and Continuous) DSMS provides very high throughput and low latency through parallelization, extensibility, and user defined incremental aggregation of windowed data streams. The high level query language provides numerical data representations and data stream windows as first class objects, which simplifies complex numerical computations over streaming data and enables automatic query optimization. To provide very high performance of low level numerical and byte processing functions the system is easily extensible with user defined functions over streams and numerical data, which allows accessing external systems and plugging in time-critical user algorithms.

EPIC extends the SCSQ system [9] with several kinds of data stream windows and incremental evaluation of user-defined aggregate functions over the windows. In particular the window operator *FEW* (Frequently Emitting Windowizer) decouples the frequency of emitted tuples from a window's slide.

To process expensive queries with high-throughput and low latency the system provides application specific stream parallelization functions where general *distribution queries* specify how to parallelize and reduce outgoing data streams.

2 The EPIC Approach

First FEW and its incremental user-define aggregation are presented in sections 2.1 and 2.2, and then the solution is outlined in section 2.3.

Figure 1 shows the overall data stream flow of the implementation. The thickness of the arrows in all data flow diagrams in this paper correspond to the relative volume of the data streams. Each node in the dataflow diagram is a separate OS process, called a *query processing node*, in which a partial continuous execution plan is running. The topology of the dataflow diagram is completely expressed in the query language where it is possible to specify continuous sub-queries running in parallel [9]. The system automatically creates OS processes running the execution plans of the sub-queries and the communication channels between them (local TCP). In the Grand Challenge implementation, the query processing nodes all run on the same computer and the OS is responsible for assigning CPUs to the processes. The system can

also distribute query processing nodes over several computers but those features are not used here.

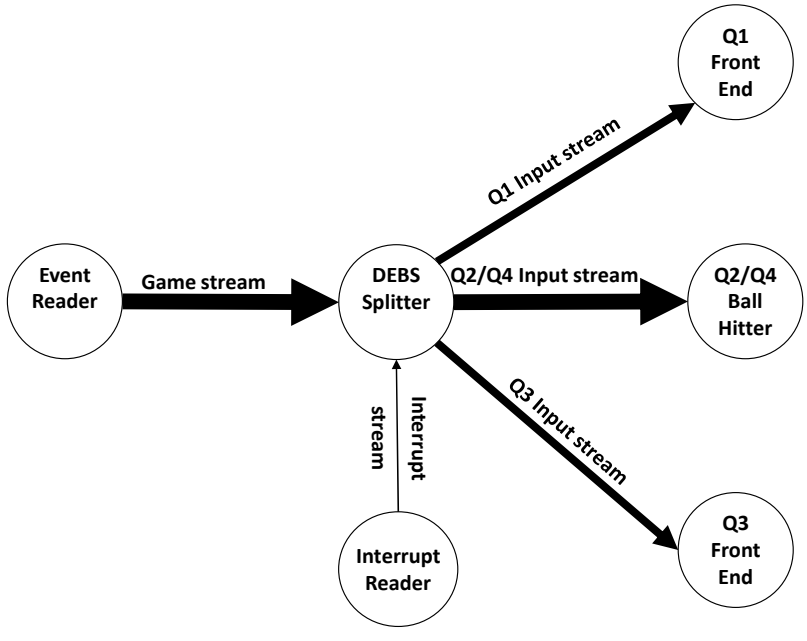

Figure 1. High-level data stream flow.

2.1 Frequently Emitting Windowizer, FEW

EPIC provides window forming operators that support several kinds of windows, including time, count, and predicate windows [5][2][7]. The windows are formed by window functions mapping streams to streams of objects of type Window. For example, the window function
tWindowize(Stream s, Number length, Number stride) -> Stream of Window ws
forms a stream *ws* of timed windows over a stream *s* where windows of length time units (seconds) slide every *stride* time units. To avoid copying, the windows are represented by pointers to their first and last elements. When a window slides the pointers are updated.

A naive implementation of *tWindowize()* would emit tuples only when the formed windows slide. This causes substantial delays, in particular for large windows. For example, when forming a 10 minutes window, it is not practical to wait 10 minutes for the aggregation to be emitted. To be able to emit aggregation results before a complete window is formed, we have introduced a window function having a parameter *ef*, the emit frequency:
fewtWindowize(Stream s, Number length, Number stride, Number ef) -> Stream of Window pw

The window forming function *fewtWindowize()* forms partial time windows, pw, every *ef* time units. The emitted partial windows are landmark subwindows of the elements of the window being formed. When the formed window is complete it is emitted as well before it slides, and then the landmark is reset to the start time of the newly slided window.

The FEW windows are required when:

- The results must be emitted before the window is formed.
- The results must be emitted more often than the slide (not used in this application).

2.2 User-defined incremental window aggregate functions

The windowing mechanism in EPIC supports incrementally evaluated user defined aggregate functions [1][8]. These are defined by associating *init()*, *add()*, and *remove()* functions with a user defined aggregate function:

- *init() -> Object o_new* creates a new aggregation object, *o_new*, which is used for accumulating changes in a window.
- *add(Object o_cur, Object e) -> Object o_nxt* takes the current aggregation object *o_cur* and the current stream element *e* and returns the updated aggregation object *o_nxt*.
- *remove(Object o_cur, Object e_exp) -> Object o_nxt* removes from the current aggregation object *o_cur* the contribution of an element *e_exp* that has expired from a window. It returns the updated *o_nxt*.

A user defined aggregate function is registered with the system function:
aggregate_function(Charstring agg_name, Charstring initfn, Charstring addfn, Charstring removefn) -> Object

For example, the following shows how to define the aggregate function *mysum()* over windows of numbers:

create function initsum() -> Number s as 0;

create function addsum(Number s_cur, Number e) -> Number s_nxt as res + e;

create function removesum(Number s_cur, Number e_exp) -> Number s_nxt as s_cur – e_exp;

These functions are registered to the system as the aggregate function *mysum()* by the function call:

aggregate_function('mysum','initsum','addsum','removesum');

After the registration *mysum()* can be used in CQs as:

select mysum(w) from Window w where w in fewtWindowize(s, 10, 2, 1);

In this simple example the aggregation object is a single number. It can also be arbitrary objects, including dictionaries (temporary tables) holding sets of rows, which is used in the Challenge implementation to incrementally maintain complex spatio-temporal aggregations.

2.3 Solution outline

In *Figure 1* the *Event Reader* node reads the full-game CSV file and produces the *Game* stream consisting of events for both balls and players. The *Event Reader* then scales the time stamps by subtracting the start time. It also transforms the position, velocity, and acceleration values to metric scales. To avoid the *Event Reader* becoming a bottleneck it is implemented as a foreign function in C. To speed up the communication we use binary representation of all events communicated between query processing nodes, while the input and output log files use the CSV format.

The *Interrupt Reader* node produces the *Interrupt* stream, which contains referee interruptions, by reading and transforming the provided game interruptions files.

The *DEBS Splitter* node merges the two input streams based on the time stamps in the streams and produces parallel input streams for the different queries. It also filters out those event stream tuples of the Game stream that are in-between game interruptions. The nodes *Q1 Front End*, *Q2/Q4 Ball Hitter*, and *Q3 Front End* receive parallel data streams required for the four Grand Challenge queries Q1-Q4. Q2 and Q4 share some downstream computations executed by *Q2/Q4 Ball Hitter* node.

In EPIC the *splitstream()* system function provides customizable distribution and transformation of stream tuples. The user can provide customizable splitting logic as a *distribution query* over an incoming tuple that specifies how a tuple is to be distributed, filtered and transformed.

The distribution query for the *DEBS Splitter* in Listing 1 is passed as an argument to *splitstream()*.

Listing 1 DEBS Splitter distribution query

```
select i, ev from Integer i
where (i = 0 and isPlayer(ev)) or
      (i = 1) or
      (i = 2 and isPlayer(ev));
```

The result of the query are pairs *(i, ev)* specifying that an incoming event *ev* is to be sent to output stream number *i*. In the DEBS splitter distribution query three output streams enumerated by *i* are specified. They produce the corresponding streams *Q1 Input*, *Q2/Q4 Input*, and *Q3 Input*. The Boolean function *isPlayer(v)* returns true if *v* is a player sensor reading.

To speed up the processing, shared computations are made in separate nodes. In *Figure 1* the *Q1 Front End* and the *Q3 Front End* nodes perform stream preprocessing and reduction for queries 1 and 3, respectively, while the *Q2/Q4 Ball Hitter* node detects hits to the ball needed by queries 2 and 4.

2.3.1 Query Q1: Running Analysis

Figure 2 shows the topology of Q1. The aggregated running statistics for different time windows are computed in parallel based on the common current running statistics produced by the *Q1 Front End* node. The stream containing player sensor readings is sent to the *Q1 Front End* node (see Listing 1), which produces the running statistics. The running statistics is then broadcasted to four other nodes to compute the aggregated running statistics of different time window lengths.

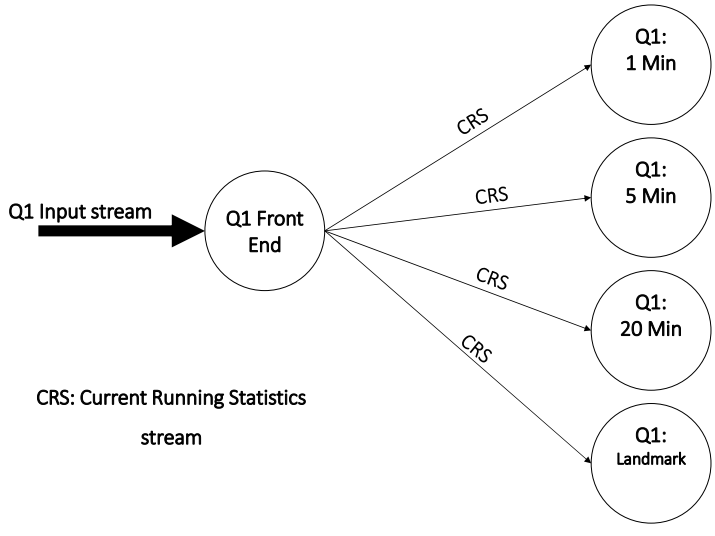

Figure 2. Query 1 data stream flow

2.3.1.1 Incremental maintenance of running statistics

In order to make the result more reliable for the current running statistics, we first create a *1 s* tumbling window and then calculate the statistics for each player over that window. The window length *1 s* was chosen experimentally to produce stable results. Both running and aggregate statistics utilize user defined aggregate functions to maintain arrays of the two types of statistics for each player.

2.3.1.2 Current running statistics

For each incoming player sensor reading in the current 1 s window, the following statistics tuple for each player is incrementally maintained in an array: *(ts_start, ts_stop, pid, left_x_start, left_y_start, left_x_stop, left_y_stop, right_x_start, right_y_start, right_y_stop, right_y_stop, sum_speed, count)*

The time stamp *ts_start* stores the first time when a sensor reading of player *pid* arrives to the current window, while *ts_stop* stores the last sensor reading. The elements *left_x_start*, *left_y_start*, *right_x_start,* and *right_y_start* are the position readings of the left and right foot of the player at time *ts_start*, while *left_x_stop*, *left_y_stop, right_x_stop,* and *right_y_stop* are the corresponding foot position readings at time *ts_stop*. To incrementally calculate the average velocity the elements *sum_speed* and *count* are also included. *ts_start*, *left_x_start*, *left_y_start*, *right_x_start,* and *right_y_start* are updated only when the first sensor reading of the player *pid* arrives to the window, while all the other elements are updated every time a sensor reading of *pid* arrives. Here, no remove function is needed for the aggregation, since we are maintaining a stream of tumbling windows where the statistic will be re-initialized every time the window tumbles.

With the statistics above, the current running statistics for a given player is calculated as the Euclidian distance between the average position of the first and last update during the time window.

2.3.1.3 Aggregate running statistics

We have chosen to log the result tuple of Q1 in CSV format every *1 s* since the current running statistics are not emitted more often than once per second. Four FEW time windows were defined for aggregating running statistics with lengths 1 minute, 5 minutes, 20 minutes, and the entire game. All windows slide and emit results every *1 s*. FEW is critical for early emission while the first windows are being formed.

Aggregate running statistics over the window are incrementally maintained in an array similar to current running statistics.

The stream from the *Q1 Front End* node contains the elements *ts_start, ts_stop, player_id, intensity, distance,* and *speed*. The difference *ts_stop – ts_start* is used to incrementally maintain the duration of a player being in the corresponding running *intensity* class. Analogously, the moving distance is maintained for the corresponding intensity classes by incrementally associating the incoming distance with the right intensity.

2.3.2 Query Q2: Ball Possession

Figure 3 shows the data flow of queries Q2 and Q4 combined. The *Q2/Q4 input* stream consists of player, ball, and interrupt sensor readings. The *Q2/Q4 Ball Hitter* computes the *Ball Hitter* and the *Ball* streams. The *Ball Hitter* stream contains ball hitter events, which occur when a player *pid* at timestamp *ts* hits the ball. The *Ball* stream contains *Ball Hitter* events interleaved with ball sensor readings. The *Q2/Q4 Ball Hitter* node emits the *Ball* stream to the *Shot on Goals* query processing node, which executes the final stages of query Q4. The *Ball Hitter* stream contains only ball hitter events and is sent to the *Player Possession* node, which calculates and broadcasts the same *Player Ball Possession* stream to four *Team Possession* query processing nodes. The *Team*

Possession nodes log every *10 s* statistics of team ball possessions for the two teams with the different window lengths: 1 minute, 5 minutes, 20 minutes, and a landmark window of the entire game. As an alternative, we also measured reporting team possessions every *1 s* resulting in the same latency and throughput.

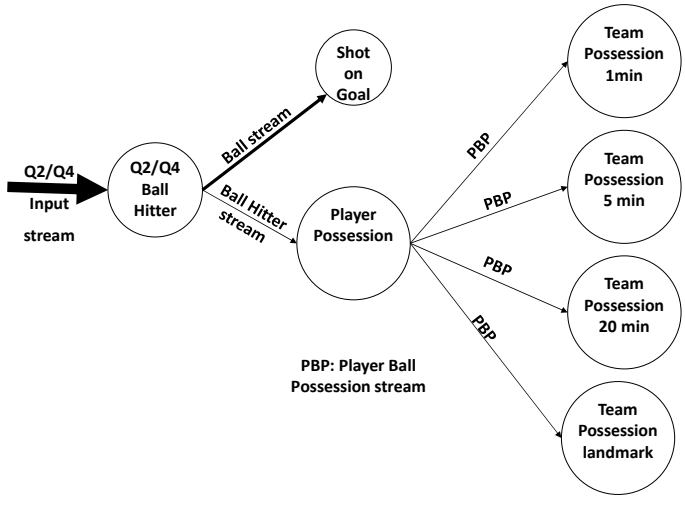

Figure 3. Query 2 and Query 4 data stream flow

2.3.2.1 The Q2/Q4 Ball Hitter query processing node

In order to compute a stream of ball hitters, we maintain acceleration of the ball *ballacc*, its position *bx, by, bz*, the shortest distance from a player to the ball *sdist*, and the player *pid*.

For every input ball sensor reading, the *Q2/Q4 Ball Hitter* node incrementally updates the ball acceleration and the ball position accordingly. When a player sensor reading arrives, it incrementally maintains *sdist.*

A ball hitter event is emitted when both the following criteria hold:

- C1: The ball acceleration reaches a predefined threshold: $ballacc > 55\ m / s^2$.
- C2: The shortest distance *sdist* is within the player's proximity: $sdist < 1\ m$.

There are 36*200 player sensor readings per second. In addition, after being hit, the ball acceleration remains high for a while, in particular before the ball leaves the player's proximity. Therefore, the two conditions C1 and C2 will hold for a short period of time within which several ball hitter events could be reported for the same actual ball hit by the player. To avoid generating false *ball hitter* events, we employ a *dropping policy* to drop player sensor readings occurring significantly later than the last report time. The dropping policy is expressed by the following query condition over a player sensor reading *v*:

ts(v)-lrts > epsilon;

Here, *lrts* is the latest timestamp when a ball hitter event was reported, and *epsilon* is the minimum time period between two reports. Because Q4 is more sensitive to the *ball hitter* events, we have empirically tuned this parameter to *0.2 s* to get the best possible accuracy of Q4.

2.3.2.2 The Player Possession query processing node

The *Player Possession* node emits the *Player Ball Possession (PBP)* stream consisting of the variables *fts*, *pid*, and *hits*, which state that the player *pid* possessed the ball *hits* times, starting from first time the player hits the ball, *fts*.

The *Player Possession* node increases the variable *hits* if a ball hitter event *bhe* is from the same player *pid*. Otherwise, it will emit ball possession events for player *pid* and then reset the variables. The total possession time is the interval between the timestamps *bhe* and *fts*.

2.3.2.3 The Team Possession query processing nodes

There are four *Team Possession* nodes, each with different window length: 1 minute, 5 minutes, 20 minutes, and a landmark of the whole game. For the received *Player Ball Possession* stream they compute team possession statistics as follows:

- Incrementally calculate the sum of the ball possessions of all players in each team when a corresponding player ball possession arrives.
- When a report is logged, the following two percentages are calculated:

$$P_A = \frac{sumTeamA}{sumTeam\ A + sum\ TeamB}$$

$$P_B = \frac{sumTeamB}{sumTeam\ A + sum\ TeamB}$$

Here FEW windows are used to frequently report while the first windows are being formed. For example, the results must be regularly delivered every *10 s* while the *team possession landmark* window is being formed.

2.3.3 Query 4: Shot on Goal

The *Shot on Goal* node receives three different kinds of events in the *Ball* stream:

- A ball hitter event marks a shot and contains a time stamp and the *pid* of the shooting player.
- A ball event contains the current ball sensor reading.
- An interrupt event indicates a game interruption. It is good practice to reset the shot detection when an interruption occurs.

Q4 shares detection of a ball hit with Q2. However, the logic for detecting a shot is slightly different for the two queries: Q2 is specified stricter than needed for Q4. To share computations this stricter logic is also used for Q4.

The operation of Q4 is straightforward; it is an iteration over the *Ball* stream to keep track of the state of a shot:

1. Wait for the next ball hitter event.
2. Check ball events until the ball has travelled one meter.
3. Return ball events as long as the ball is approaching the opposite team's goal.

The calculation of the ball direction uses basic linear algebra over the ball sensor readings.

Gravity is accounted for to an extent. The expected time for the ball to travel to the goal line is multiplied twice with the acceleration constant *g*, and added to the height of the goal bar. The actual ball trajectory is not considered, but the current calculation should be an adequate approximation.

Using the Q2 requirements for detecting a ball hit has the draw-back that some events are not detected, such as the header at 12:19 in the second half our example *Game* stream, since the ball is more than one meter away from any sensor. Whether that is technically a "shot" is questionable.

Curve balls need special attention. For example, at 26:07 in the first half there is a curve ball goal. In this case the direction of the ball is pointing outside the goal posts, while the ball later curves inwards and comes to rest inside the goal.

To handle curve balls we have introduced a state *pending*, indicating that a shot is not yet dismissed, but could later be become a shot on goal. The model adds two meters of margin on both sides of the goal posts and the shot is considered pending if it points in the direction of the margin area.

Bounces are considered as long as the direction of the bounce is within the negative distance of the goal bar plus gravity. While the instructions do not account for bounces at all, this limit should add some correctness to the algebra.

Shots that are bounces, which we detect, are not included in the provided list of shots on goal. In the second half of the game there are four shots on goal that are bounces. They are at 4:11, 19:39, 24:36 and 29:29. Setting the bounce threshold to zero, i.e. not considering bounces creates a result in accordance to the specification. Viewing the video makes it apparent that the specification is not correct in this regard.

2.3.4 Query 3: Heat Map

In Query 3 a grid on the field is formed where the cells are numbered in row order, for example from 0 to 6399 in a 64 X 100 grid. Given the position of a player *(x,y),* the function *cell_id(x,y,grid_size)* returns the corresponding cell

number for a given grid size. Query results for lower resolution grids are computed by aggregating the results for the higher resolution grids. Thus we incrementally maintain the results only for the highest resolution.

Note that the results of longer windows cannot be built on top of the results from a shorter window. This is due to the *1 s* stride parameter in all the queries. For example, the 5 minute window can't be built on top of the results produced by the 1 minute window, since the 5 minute window needs to remove the contributions made to the statistics by the expired elements, i.e. the elements with the time stamp *ts – 300 s*, where *ts* is the current time stamp. Those elements are too old to be in the 1 minute window. Nevertheless, the definition of longer windows in terms of shorter ones could have been utilized if the stride was one minute instead of the one second stride in the Challenge specification

2.3.4.1 Q3 Front End

Figure 4 shows the dataflow diagram for query Q3. As specified in Listing 1 the *Q3 Input Stream* contains all player sensor readings. The *Q3 Front End* node produces the *One Second HeatMap (OSHM)* stream by forming *1 s* tumbling windows over the incoming tuples. Thereby incremental user defined aggregate functions are used to maintain statistics per second in a table *heamap1s(pid, cell_id, ts, cnt)* local per window. Here *ts* is the latest time stamp player *pid* has been present in the cell identified by *cell_id* cell identifier in the highest resolution grid (64 X 100). *cnt* is the total number of sensor readings for player *pid* in the cell in the current window.

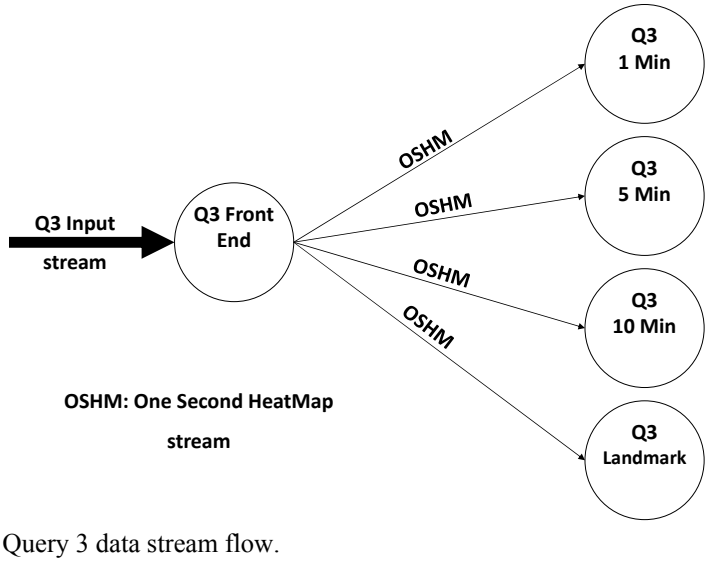

Figure 4. Query 3 data stream flow.

The *OSHM* stream is produced by emitting all the rows accumulated in the table during the past second.

The *Q3 Front End* significantly reduces the stream volume by summarizing it. It receives 200 tuples per second from 36 sensors, in total 7200 tuples/second. It emits at maximum the total number of cells all the players have been present in the highest grid resolution during one second, which is about 70 tuples per second, i.e. a factor 10 reduction in stream flow.

2.3.4.2 Q3 query nodes

The *OSHM* stream is broadcasted to four Q3 query nodes *Q3 1 Min*, *Q3 5 Min*, *Q3 10 Min*, and *Q3 Landmark*. These nodes run parallel CQs over time windows with lengths 1, 5, 10 minutes, and whole game, respectively. The windows are formed by the FEW window specification *fewtWindowize(oshm, length, 1, 1),* where *length* is 60s, 300s, 600s and the whole game duration, respectively. The stride and the emit frequency are both *1 s*. The emit frequency is needed so that sub-windows are emitted while the window is being formed the first time.

Similar to *Q3 Front End*, the Q3 query nodes incrementally maintain user defined aggregates by updating the following local tables inside each window as the input stream elements arrive:

heatmap(pid, cell_id, ts, cnt)
sensor_count(pid, total_cnt

In table *heatmap*, the attribute *cell_id* is the cell player *pid* has been present in, *ts* is the latest time player *pid* was in the cell, *cnt* is the number of times the player has been present in the cell. To enable translation of *cnt* into percentages per cell, the Q3 query nodes also maintain *total_cnt* per player, which stores the total number of position reports in all cells for a given player during the window in question.

Since Q3 query nodes only maintain the statistics for the highest resolution in a given window length, at reporting time they compute lower resolutions by aggregating grid cells per player to fill the bigger cells in the higher resolutions.

The Q3 query nodes log the output CSV streams to files. Since each Q3 query nodes cover all grid settings in a given window size, the produced log files contains output stream elements for more than one grid setting. We use the following grid identifiers to tag streams per grid: *6400* for 64 X 100, *1600* for 32 X 50, *400* for 16*25, and *104* for 8 X 13 grid setting.

The size of these log files is huge (ca 400,000 rows/s) since they cover all movements between grid cells over several very long windows. Here it becomes important to use SSD as storage medium, which is fast at writing big blocks in parallel, while disk arm movements for writing different log files has been observed to slow down the entire system throughput with a factor of around two.

3 Performance

We measured the performance of our implementation based on both throughput and delay. The throughput was measured as the total execution time per query and for all queries in parallel over the entire game. The latency was measured by propagating the system wall clock of the entry time of the latest event contributing to each result tuple. The delay was calculated by subtracting the propagated entry time from the wall time when a result tuple is delivered. The throughput is measured per query while the latency is measured per output stream.

We ran our experiments on a VMware virtual machine with Windows Server 2008 R2 x64, running on a laptop with the following specifications: Dell Latitude E6530, CPU: Intel Core i7-3720QM @2.60 GHz, RAM: 8 GB, Hard Disk Device: ST500LX003-1AC15G, OS: Windows 7 64-bit.

Figure 5 illustrates the throughput of the individual queries as well as all queries running together. Queries Q1, Q2, and Q4 take around 5 minutes to finish separately, while Q3 takes considerably longer time, which is mainly due to intensive report computations in the Q3 query nodes. To investigate the log writing time, *Q3* and the *all queries* columns have a watermark indicating how much time it takes to execute them without logging to disk, showing that this takes around 35 % of the Q3 alone time and 25 % of all queries together. We also investigated whether it would be favorable to parallelize the logging of the result stream for Q3 query nodes, but that turned out to be slower in our current environment.

Since all queries run in parallel according to the dataflow diagrams, running all of them together takes approximately the same time as running the slowest one, Q3.

Figure 6 shows the average delay per output stream while running all queries together. Notice that Q2 and Q4 are time critical queries since they immediately report real-time phenomena. By contrast Q1 and Q3 report delayed statistics aggregated over time.

The VMware virtual machine containing our implementation of the Grand Challenge can be downloaded from http://udbl2.it.uu.se/DEBS/. There is also a zip archive that can be run on any Windows machine.

4 Related Work

In the stream processing community, there has been a lot of work for developing query languages over data streams [5] [7] introduced a formal specification of different kinds of windows over data streams and provided a taxonomy of window variants. The notation of report (emit) frequency was proposed in SECRET [2] without any actual implementation. SECRET is a descriptive model to help users understand the result of window-based queries

from different stream processing engines. Esper [4] also allows a report frequency but does not have user defined window aggregate functions. Furthermore Esper's sliding window model is different from FEW because the slides are triggered by window content changes rather than explicitly specified time periods.

To efficiently calculate the aggregate result over long windows with small strides, [6] and [1] use delta computations to reduce the latency and the memory usage. The focus of [8] is to extend a DSMS with online data mining facilities by user defined aggregate functions over windows. The implementation described in this paper shows that EPIC is general enough to define very complicated user defined aggregations as functions while in [1] and [8] the aggregates are defined as updates.

5 Conclusions

We have addressed the Grand Challenge by expressing continuous queries in a high level language that supports incremental evaluation of aggregate functions over windows and frequently emitting windowing. We meet the real-time requirements of the real-time queries on a virtual machine running on a laptop. The extensibility of the query engine was used for supporting high throughput and low latency of time critical operations.

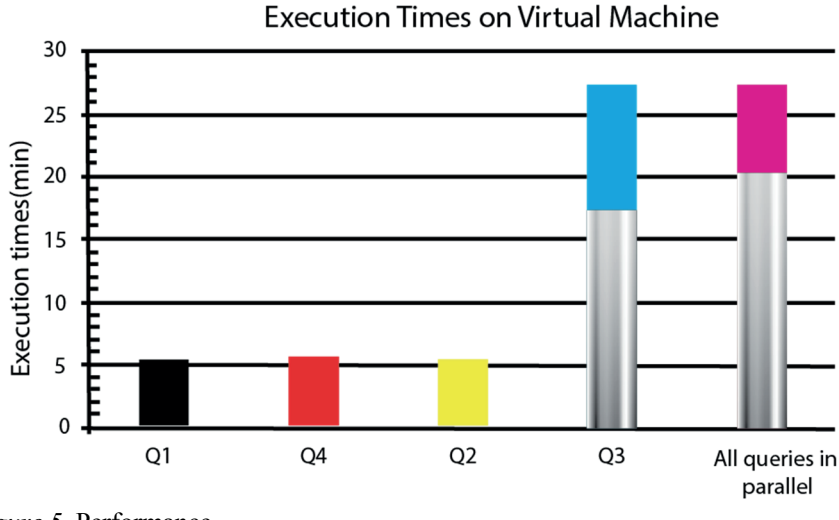

Figure 5. Performance.

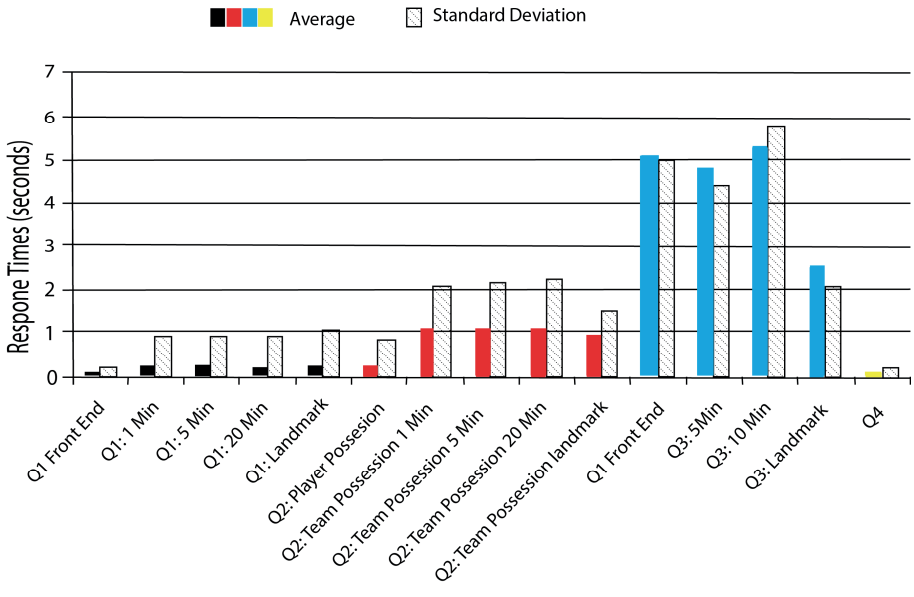

Figure 6. Delays.

Acknowledgements

This work was supported by the Swedish Foundation for Strategic Research, grant RIT08-0041 and by the EU FP7 project Smart Vortex.

References

[1] Bai, Y., Thakkar, H., Wang, H., Luo, C., and Zaniolo, C.: A Data Stream Language and System Designed for Power and Extensibility. *Proc. CIKM Conf.*, 2006.

[2] Botan, I., Derakhshan, R., Dindar, N., Haas, L., Miller, R. J. and Tatbul, N. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *Proc. VLDB Conf.*, 2010.

[3] Botan, I., Fischer, P. M., Florescu, D., Kossmann, D., Kraska, T., and Tamosevicius, R. Extending XQuery with Window Functions. *Proc. VLDB Conf.,* 2007.

[4] http://esper.codehaus.org/

[5] Law, Y-N, Wang, H., and Zaniolo, C.: Relational Languages and Data Models for Continuous Queries on Sequences and Data Streams. ACM TODS 36, 2, (May 2011).

[6] Li, J., Maier, D., Tufte, K., Papadimos,V., and Tucker, P. A. Semantics and evaluation techniques for window aggregates in data streams. *Proc. SIGMOD Conf.*, pp. 311 - 322, 2005.

[7] Patroumpas, K. and Sellis, T. Window specification over data streams. *Proc. EDBT Conf.,* 2006.

[8] Thakkar, H., Mozafari, B. and Zaniolo, C.: Designing an Inductive Data Stream Management System: the Stream Mill Experience. *Proc. 2nd International Workshop on Scalable Stream Processing Systems*, 2008.

[9] Zeitler, E. and Risch, T.: Massive scale-out of expensive continuous queries, *Proc. of the VLDB Endowment*, ISSN 2150-8097, Vol. 4, No. 11, pp.1181-1188, 2011

Paper V

Paper V

Mahmood, Khalid, Truong, Thanh and Risch, Tore.
NoSQL Approach to Large Scale Analysis of Persisted Streams.
In Proceedings of the 30th British International Conference on Databases,
BICOD 2015, pages 152—156.
Edinburgh, UK, July 6-8, 2015.
Springer International Publishing.
ISBN: 978-3-319-20424-6.
DOI=10.1007/978-3-319-20424-6_15
http://dx.doi.org/10.1007/978-3-319-20424-6_15

The paper is reformatted for typographic consistency.

NoSQL Approach to Large Scale Analysis of Persisted Streams

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

ABSTRACT
A potential problem for persisting large volume of streaming logs with conventional relational databases is that loading large volume of data logs produced at high rates is not fast enough due to the strong consistency model and high cost of indexing. As a possible alternative, state-of-the-art NoSQL data stores that sacrifice transactional consistency to achieve higher performance and scalability can be utilized. In this paper, we describe the challenges in large scale persisting and analysis of numerical streaming logs. We propose to develop a benchmark comparing relational databases with state-of-the-art NoSQL data stores to persist and analyze numerical logs. The benchmark will investigate to what degree a state-of-the-art NoSQL data store can achieve high performance persisting and large-scale analysis of data logs. The benchmark will serve as basis for investigating query processing and indexing of large-scale numerical logs.

Keywords. NoSQL data stores, numerical stream logs, data stream archival.

1 Introduction

The data rate and volume of streams of measurements can become very high. This becomes a bottleneck when using relational databases for large-scale analysis of streaming logs [1][2][3][4]. Persisting large volumes of streaming data at high rates requires high performance bulk-loading of data into a database before analysis. The loading time for relational databases may be time consuming due to full transaction-al consistency [5] and high cost of indexing [6]. In contrast to relational DBMSs, NoSQL data stores are designed to perform simple tasks with high scalability [7]. For providing high performance updates and bulk-loading, NoSQL data stores generally sacrifice strong consistency by providing so called eventual consistency compared with the ACID

transactions of regular DBMSs. Therefore, NoSQL data stores could be utilized for analysis of streams of numerical logs where full transactional consistency is not required.

Unlike NoSQL data stores, relational databases provide advanced query languages and optimization technique for scalable analytics. It has been demonstrated in [8] that indexing is a major factor for providing scalable performance, giving relational databases a performance advantage compared to a NoSQL data store to speed up the analytical task. Like relational databases, some state-of-the-art NoSQL data stores (e.g. MongoDB), also provide a query language and both primary and secondary indexing, which should be well suited for analyzing persisted streams.

To understand how well NoSQL data stores are suited for persisting and analyzing numerical stream logs, we propose to develop a benchmark comparing state-of-the-art relational databases with state-of-the-art NoSQL data stores. Using the bench-mark as test bed, we will then investigate techniques for scalable query processing and indexing of numerical streams persisted with NoSQL data stores.

2 Application Scenario

The Smart Vortex EU project [1] serves as a real world application context, which involves analyzing stream logs from industrial equipment. In the scenario, a factory operates some machines and each machine has several sensors that measure various physical properties like power consumption, pressure, 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*. Relational databases are used to analyze the logs by bulk-loading them in table measures *(m, s, ts, mv)* which contains a large volume of data logs from many sensors of different machines [3][4].

Since the incoming sensor streams can be very large in volume, it is important that the measurements are bulk-loaded fast. After stream logs have been loaded into the database, the user can perform queries to detect anomalies of sensor readings. The following query analyzes the values of *mv* from sensor logs for a given time interval and parameterized threshold.

```
SELECT *    FROM measures
WHERE       m = ? AND s = ?AND
            ts > ? AND
             ts < ? AND mv > @th
```

In order to provide scalable performance of the query, we need an index on the composite key of *m*, *s*, *ts* and a secondary B-tree index on *mv*.

3 Challenges in Analyzing Large Scale Persisted Streams

Analysis of large-scale stream logs in the above application scenario poses the follow-ing challenges (C1 to C6) in utilizing relational and NoSQL data stores.

C1. Bulk-loading: In relational DBMSs, the high cost of maintaining the indexes and full transactional consistency can degrade the bulk-loading performance of large volume of data logs. The loading performance of a relational DBMS from a major commercial vendor, called *DB-C* and a popular open source relational database, called *DB-O* for 6GB of data logs is shown in *Figure 1* It took more than 1 hour in a high performance commodity machine for the state-of-the-art commercial DBMS, DB-C to bulk-load data logs consisting of around 111 million sensor measurements. Some of the data logs consist of more than a billion sensor measurements, which require high-performance bulk-loading. To boost up the performance, weak consistency level of a NoSQL or relational database can be utilized.

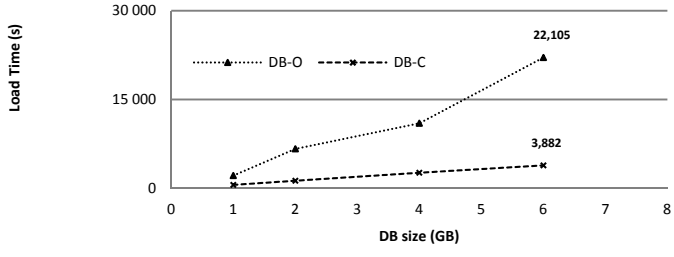

Figure 1. Bulk-loading performance of 6GB logs.

Figure 2. Index and database size of 6GB of logs.

C2. Index size: *Figure 2* shows the index and database sizes for 6GB of stream logs loaded into the two DBMSs. The size of the index created in both relational DBMSs was larger than the size of the original logs. For high performance and scalable analysis of typical stream logs, hundreds of gigabytes

of memory is required in our application. It is interesting to see whether the state-of-the-art NoSQL data store can provide memory efficient indexing strategies. Novel indexing techniques can also be incorporated in order to provide a memory efficient indexing for analyzing persisted streams.

C3. Indexing strategies: Unlike relational databases and MongoDB, most NoSQL data stores do not provide both primary and secondary indexing, which are essential to scalable processing of queries over data logs. Some NoSQL data stores such as Hbase, Cassandra, Memcached, Voldemort, and Riak do not provide full secondary indexing, which is needed for queries having inequalities over non-key attributes. CouchDB has secondary index, but queries have to be written as map-reduce view [7], not transparently utilizing indexes.

C4. Query processing: Unlike relational databases, most NoSQL data stores do not provide a query optimizer. Some NoSQL data stores, e.g. MongoDB, provide a query language that is able to transparently utilize indexes. However, the sophistication of query optimizer still needs to be investigated for scalable analysis of data logs.

C5. Advanced analytics: Relational DBMS features for advanced analytics such as joins or numerical expressions is limited in NoSQL data stores. Therefore, it needs to be investigated how advanced numerical analytics over large-scale data logs could be performed by NoSQL data stores.

C6. Parallelization of data: NoSQL data stores have the ability to distribute data over many machines, which can provide parallel query execution. However, typical queries for analyzing data logs can generate lots of intermediate results that need to be transferred over the network between nodes, which can be a performance bottleneck. Therefore, the performance of both horizontal and vertical partitioning of distributed NoSQL data stores can be investigated for query execution over numerical logs.

4 Proposed Work

There are several investigations that can be performed for large-scale analysis of numerical stream logs.

Stream log analysis benchmark: Typical TPC benchmarks [9] such as TPC-C, TPC-DS, and TPC-H are targeted towards OLTP or decision support, not for log analysis. To benchmark data stream management systems, the Linear Road Benchmark (LRB) [10] is typically used. However, LRB does not include the performance of persisted streams. Analysis of large-scale data logs often requires scalable queries (e.g. [3][4]) over persisted numerical logs, which should be the focus the benchmark. In the benchmark, several state-of-the-art NoSQL data stores should be compared with relational DBMSs to investigate at what degree NoSQL data stores are suitable for persisting and analyzing large scale numerical data streams. The performance of bulk-loading

capacities of the databases w.r.t. indexing and relaxed consistency should be investigated in the benchmark. The queries should be fundamental to log analyses and targeted to discover the efficiency of query processing and utilization of primary and secondary index of the data logs. The benchmark should analyze and compare the performance differences of loading with relaxed consistency, index utilization, and query execution for both NoSQL and relational databases, which can provide the important insights into challenges C1, C3, C4, and C6.

Query processing: Supporting advanced analytics using a complete query language with a NoSQL data store requires the development of query processing techniques to compensate for the limitation of the NoSQL query languages, for example lack of join and numerical operators. The push-down of query operators as generated parallel server side scripts should be investigated. Furthermore, it should be investigated how domain indexing strategies [11] in a main memory client-side database (e.g. Amos II [12] developed at UDBL of Uppsala University and [13]) can improve performance of numerical data log analyses of data retrieved from back-end NoSQL databases. These can provide the insights of the challenges C2 and C5.

References

[1] Smart Vortex Project, http://www.smartvortex.eu/

[2] Zeitler, E., Risch, T.: Massive Scale-out of Expensive Continuous Queries. In: VLDB (2011)

[3] Truong, T., Risch, T.: Scalable Numerical Queries by Algebraic Inequality Transformations. In: DASFAA (2014)

[4] Zhu, M., Stefanova, S., Truong, T., Risch, T.: Scalable Numerical SPARQL Queries over Relational Databases. In: LWDM Workshop (2014)

[5] Doppelhammer, J., Höppler, T., Kemper, A., Kossmann, D.: Database performance in the real world. In: SIGMOD (1997)

[6] Stonebraker, M.: SQL databases v. NoSQL databases. Comm. ACM. (2010)

[7] Cattell, R.: Scalable SQL and NoSQL data stores. ACM SIGMOD Rec. 39 (2011)

[8] 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 (2009)

[9] Council, T.P.P.: TPC Benchmarks, http://www.tpc.org/information/benchmarks.asp

[10] Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear road: a stream data management benchmark. In: VLDB (2004)

[11] Gaede, V., Günther, O.: Multidimensional Access Methods. ACM Comput. Surv. 30 (1998)

[12] 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 (2004)

[13] Freedman, C., Ismert, E., Larson, P.-Å.: Compilation in the Microsoft SQL Server Hekaton Engine. In: IEEE Data Eng. Bull. 37, 1 (2014)