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 [16][26][1]. 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

SSDBM '15, June 29 - July 01, 2015, La Jolla, CA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3709-0/15/06…$15.00
DOI: http://dx.doi.org/10.1145/2791347.2791375

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++ implementations[1]. 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 details. Section 5 presents queries used to illustrate Mexima's query processor in Section 6. 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

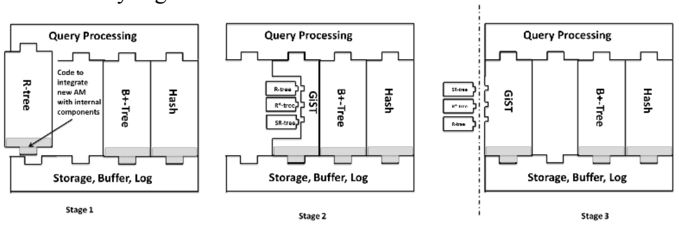


Figure 1 History of extensible indexing frameworks

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 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.

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

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.

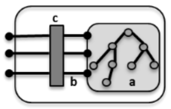


Figure 2 Index extension components

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*.

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.

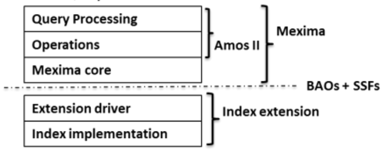


Figure 3 Mexima architecture

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.

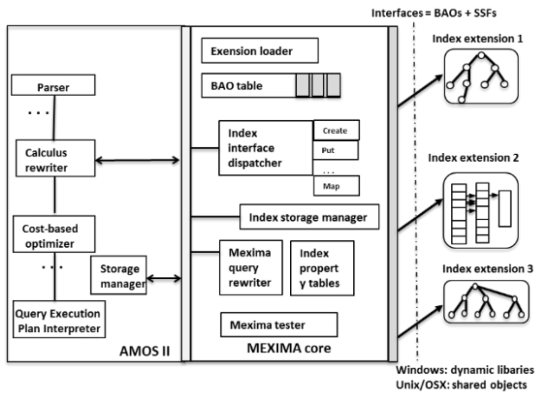


Figure 4 Mexima details

4.2 Mexima core components

We now discuss in details 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).

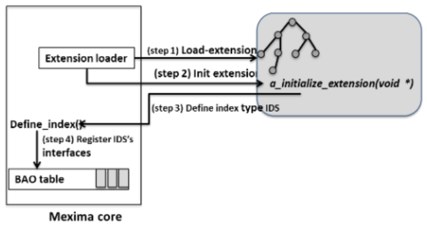


Figure 5 Extension loader's steps

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 might 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.

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')
```

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

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 r_1 expression) AND
(iv r_2 expression) AND
. . .
(iv r_n expression)

Here, ***iv*** is a variable bound to an indexed column of table *P(...)*. We say ***iv*** is an *indexed variable*. r_i are comparison operators in the set ***relop***, $r_i \in relop$, 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, …) r_1 expression AND
isf(…,iv, …) r_2 expression AND
. . .
isf(…,iv, …) 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 rewriter 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)
```

The first argument of the SSF *btree_select_range()* is the table *images* and the 2nd argument is position 0 of the indexed column *id*. We say that the corresponding B-trees index on column *id* is *exposed* by the SSF *btree_select_range()*.

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 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(              AND
  |               'image', 1, hist_p, (1/0.9 - 1))
3 | 1/(1+distance (hist_p, hist_q)) >= 0.9
```

7. THE 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 1). 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 1), 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_i and VQ_r are generated. VQ_i 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:

<u>Case 1</u>: 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 $p_1,\ldots,p_m$, and that r_i are the *relops* in Form (i). Then the validation query VQ_i has the following format:

```
select iv, v
from IT iv, Number v,
         $T_1$ $p_1$, $T_2$ $p_2$,.., $T_m$ $p_m$
where I_Table(iv, v)                and
        ($p_1$, $p_2$, …,$p_m$) in (SPG)   and
        (iv $r_1$ $p_1$)                   and
        (iv $r_2$ $p_2$)                   and
        . . .
        (iv $r_m$ $p_m$);
```

For example, the automatically generated validation query VQ_i 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;
```

VQ_r is the same query with *I_Table* replaced with *R_Table*.

Case 2: TR matches Form (ii).

VQ_i 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*.

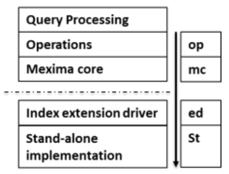


Figure 6 Execution layers

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 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 (*µ*s)

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. Analogously, Figures 8, 9, and 10 show lookup, delete, and

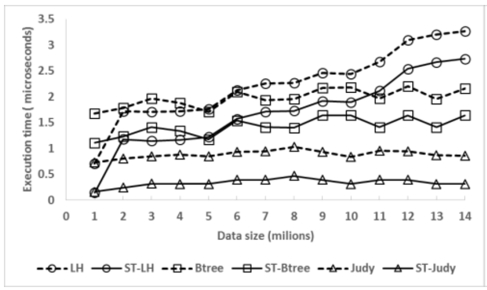


Figure 7 Put() overhead

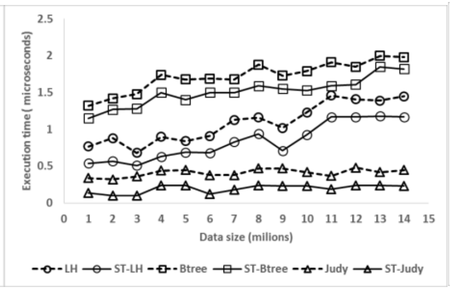


Figure 8 Get() overhead

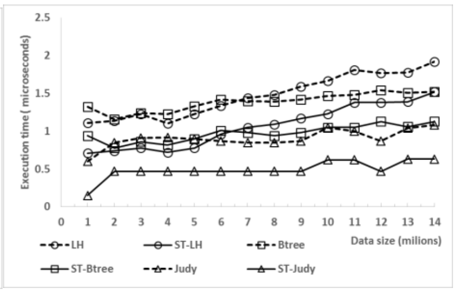


Figure 9 Delete() overhead

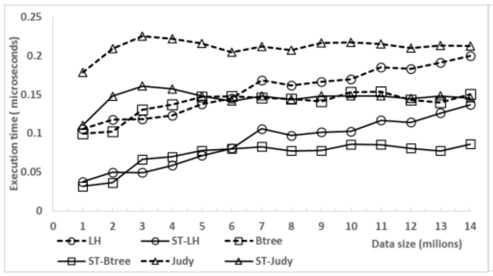


Figure 10 Map() overhead

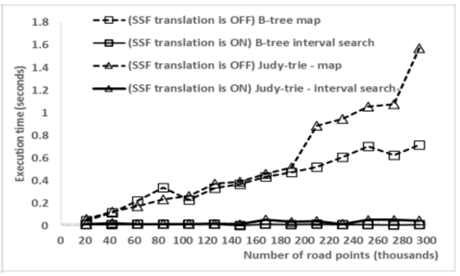
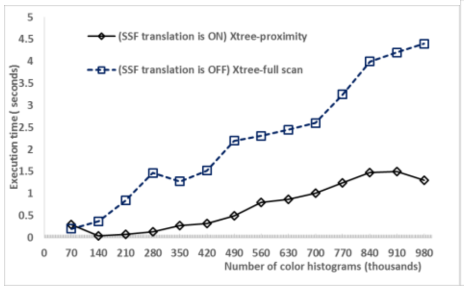


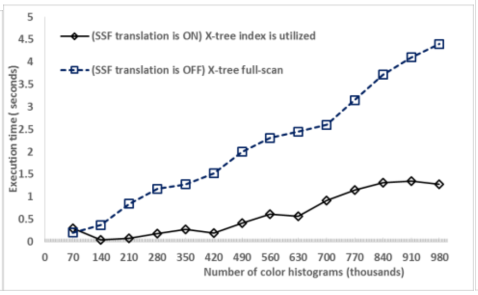
Figure 11 Q1 - Range search

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.

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 Figures 7, 8, 9, and 10. The system carefully makes sure that an index is not accessed more than once in an operation, as





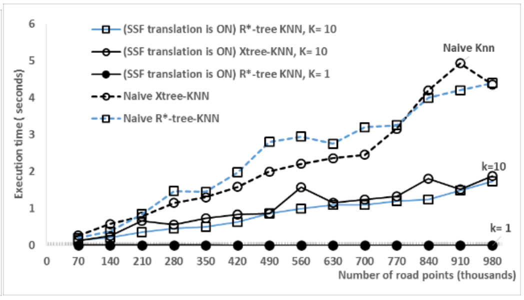


Figure 12 Q2 – Proximity search **Figure 13 Q2 - Similarity search** **Figure 14 Q3-KNN search**

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.

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

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.

In Figure 14, 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 $1 \cdot S$ to $14 \cdot S$ 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.

In Figure 12, 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 $1 \cdot S$ to $14 \cdot S$.

Figure 13 shows the scalability improvement by rewriting similarity queries for Q4 (Form (v)) using the X-tree implementation and the ColorHist database.

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. ACKNOWLEDGMENTS

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. SSDBM*, 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. VLDB Conf*, 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: h*ttp://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. 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. 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 Eng. Bull.* 18(3), pp 19-29, 1995.

[13] G. Goetz, W. J. McKenna: The Volcano optimizer generator: Extensibility and efficient search, *Proc. ICDE Conf.*, IEEE, 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 VLDB Journal*, 16(1), pp. 145-164, 2007

[16] J Hellerstein. M., J. F. Naughton, and A. Pfeffer: Generalized search trees for database systems, Proc. *VLDB Conf.*, pp 562–573, 1995.

[17] M. Kornacker: High-performance extensible indexing, *Proc. VLDB Conf.*, 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, *Proceedings of the VLDB 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. ICDE Conf.*, pp. 391-400, 1997.

[23] J. T. Robinson: The KDB-tree: a search structure for large multidimensional dynamic indexes, *Proc 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. ICDE Conf.*, pp 91–100, 2000.

[27] B. Stefan, A.K. Daniel, H-P. Kriegel: The X-tree : An Index Structure for High-Dimensional Data, *Proc. VLDB Conf.*, 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 system*s," 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*, in SIGMOD '10, 2010

[33] V. Leis, A. Kemper, and T. Neumann: *The adaptive radix tree: Artful indexing for main-memory databases*, in ICDE '13, 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