Integrating the Relational Interval Tree into IBM's DB2 Universal Database Server

Christoph Brochhaus[1], Jost Enderle[1], Achim Schlosser[1], Thomas Seidl[1] and Knut Stolze[2]

[1]RWTH Aachen University, Data Management and Exploration Group
{brochhaus, enderle, schlosser, seidl}@informatik.rwth-aachen.de

[2]IBM Germany, Information Integration Development, and
University of Jena, Database and Information Systems Group
stolze@de.ibm.com

Abstract: User-defined data types such as intervals require specialized access methods to be efficiently searched and queried. As database implementors cannot provide appropriate index structures and query processing methods for each conceivable data type, present-day object-relational database systems offer extensible indexing frameworks that enable developers to extend the set of built-in index structures by custom access methods. Although these frameworks permit a seamless integration of user-defined indexing techniques into query processing they do not facilitate the actual implementation of the access method itself. In order to leverage the applicability of indexing frameworks, relational access methods such as the Relational Interval Tree (RI-tree), an efficient index structure to process interval intersection queries, mainly rely on the functionality, robustness and performance of built-in indexes, thus simplifying the index implementation significantly. To investigate the behavior and performance of the recently released IBM DB2 indexing framework we use this interface to integrate the RI-tree into the DB2 server. The standard implementation of the RI-tree, however, does not fit to the narrow corset of the DB2 framework which is restricted to the use of a single index only. We therefore present our adaptation of the originally two-tree technique to the single index constraint. As experimental results with interval intersection queries show, the plugged-in access method delivers excellent performance compared to other techniques.

1 Introduction

Users of database systems want to manage data of very different types, depending on the particular application area. While office applications, for example, mainly perform simple access and update operations on records of simple data types, data in technical and scientific domains usually have a complex structure and demand specialized operations. It is not a choice for vendors of database management systems to provide data types and management functions for each conceivable domain. So the design of extensible architectures allowing users to adapt systems to their special needs represents an important area in the development of database systems.

To make object-oriented and extensibility features also available in relational systems, database researchers and manufacturers proposed and implemented corresponding enhancements for the relational model during the last years. The resulting *object-relational database management systems* (ORDBMSs) retain all features of the relational model, especially the storage of data within tables and the powerful declarative query processing with the relational database language SQL. Beyond that, the object-relational model introduces abstract data types into relational database servers. So users are able to define application-specific complex types (by the nested application of type constructors) together with appropriate operations.

One of the simplest complex types is the interval data type. Intervals have a wide range of applications where they are commonly used as transaction time and valid time ranges [SA85]. In connection-oriented communication scenarios like phone calls or online banking sessions, for example, intervals may represent the starting and ending points of the connect times. A query like "Find all clients that were online between t_1 and t_2" can easily be expressed by an intersection query of intervals on a database storing the online periods of the clients. Further examples include the logging of passage times for vehicles driving on a certain road section, where the presence of vehicles during a time period may be evaluated for purposes of toll collection or analyses of the traffic flow. Intervals also appear in spatial and many other applications, e.g. for describing line segments on a space-filling curve [FR89] [KPS01] or managing the terms of contracts.

Because of their practical relevance, corresponding data types and operations have been introduced in recent SQL standards. SQL:2003 [ISO03a] differentiates between *intervals* and *chronological periods*. An interval represents the duration of a period in time (and thus is a scalar value), whereas a period is a compound object specified either as a pair of *datetimes* (starting and ending point of time) or as a starting datetime and an interval. The only operator currently defined on periods is *overlaps* which determines whether or not two periods overlap in time. SQL also provides corresponding types and operations for spatial applications within a separate application package [St03] [ISO03b]. To implement the above session scenario in an SQL-compliant database, we could write the following statements:

```
// create a table storing the periods (starting and ending points) of the sessions
CREATE TABLE sessions (
    theKey INT,
    thePeriod ROW (lower TIME, upper TIME))

// retrieve all sessions that proceeded during a certain period
SELECT theKey
FROM sessions
WHERE thePeriod OVERLAPS (TIME '10:45:00', TIME '11:45:00')
```

Figure 1: Period example

In order to provide efficient searching and querying on user-defined objects, special access methods for the new data types and their predicates are required. Unfortunately, most commercially relevant database systems do not provide any built-in access methods for temporal and spatial data types. B+-trees often serve as the solely available indexing method supporting only simple (built-in) data types and predicates. For indexing intervals, B+-trees are not a good choice as long as there is no appropriate sort order on the interval values. In the presence of intervals with very different lengths, using a simple composite index on the two bounding points does not improve access times significantly. Provided only with a functional implementation of the OVERLAPS predicate, the optimizer has to perform a full table scan to perform the query in Figure 1. Applications like the aforementioned ones, however, often have to handle millions of objects with interval-valued attributes and so would cause unacceptable response times.

Extensible indexing frameworks, as already proposed by Stonebraker [St86], enable developers to extend the set of built-in index structures by custom access methods in order to support user-defined data types and predicates. By opening up the hard-wired index maintenance and exploitation logic of the built-in optimizer, the user gets the chance to implement its own application-specific index components and plug them into the extensible optimizer. Having registered the appropriate index functions for a new data type, the database server automatically triggers the maintenance and scan of custom indexes on instances of this type, thereby preserving the declarative paradigm of SQL. With a cost model registered at the optimizer, the server is able to generate efficient execution plans for user-defined types.

Although these indexing frameworks provide a gateway to seamlessly integrate user-defined access methods into the standard process of query optimization, they do not facilitate the actual implementation of the access method itself. Modifying or enhancing the database kernel is usually not an option for database application developers, as the embedding of block-oriented access methods into concurrency control, recovery services and buffer management causes extensive implementation efforts and maintenance cost [Ko99], at the risk of weakening the reliability of the entire system.

With the Relational Interval Tree (RI-tree) [KPS00][KPS01], an efficient access method has been proposed to process interval intersection queries on top of any existing relational database system. Instead of accessing raw disk blocks directly, data objects are managed by common built-in relational indexes following the paradigm of relational indexing [Kr03] [Kr04]. Its implementation is restricted to (procedural) SQL, and no intrusive augmentations or modifications of the database kernel are required. On top of its pure relational implementation, the RI-tree is ready for immediate object-relational wrapping. More and more object-relational database systems implement extensible indexing frameworks [Bl99] [IBM03] [Sr00] [Ora04], and the RI-tree successfully has been integrated in one of those systems already [KPS00]. Starting with Version 7.1, the IBM DB2 Universal Database Server has been extended by an extensible index interface [Ch99] [IBM02]. Like other interfaces, it provides some “hooks” where the user can implement its own functions to describe a new access method for a user-defined type.

In this paper, we present the integration of the Relational Interval Tree into the DB2 server using its new indexing interface. Because of the characteristics of the DB2 interface, we have to do some redesign of the original implementation of the RI-tree: Instead of using two internal B+-trees for managing the *lower* and *upper* values of the intervals, we have to map all values onto a single internal B+-tree. As experimental results on a DB2 8.1 server with interval intersection queries show, the plugged-in RI-tree outperforms other approaches including a plain built-in composite index or the DB2 Spatial Extender [Ad01] significantly.

The remainder of the paper is organized as follows: Section 2 explains the DB2 extensible indexing framework. In Sections 3 and 4, we describe the structural adaptation of the Relational Interval Tree to the DB2 framework and the necessary changes in query processing. Section 5 presents the results of our experimental evaluation, and Section 6 considers related work. The paper is concluded in Section 7.

2 DB2 Extensible Indexing Framework

There are several tasks to consider when integrating an index structure into a database system. Whereas the DB2 UDB Server takes care of index creation, index deletion and other index administration tasks automatically, two obligations are left to the index implementor. The first task is index maintenance which includes insertion into, deletion from and updates on the index structure, whenever the indexed table is modified. The second task is applying the index to process a query.

In order to provide advanced index support for user-defined types, IBM introduced the extensible indexing framework [Ch99] for the DB2 UDB. It provides four hooks in the described tasks, namely the *key generator*, *predicate specification*, *range producer* and *index filter* marked in bold italics in Figure 2 and Figure 3. Whereas the first three methods are required to be implemented for advanced index support, the *index filter* is facultative.

2.1 Index Maintenance

Figure 2 illustrates the processing of insertions, deletions or updates on the indexed table and the resulting updates on the corresponding index. An SQL query may cause a new

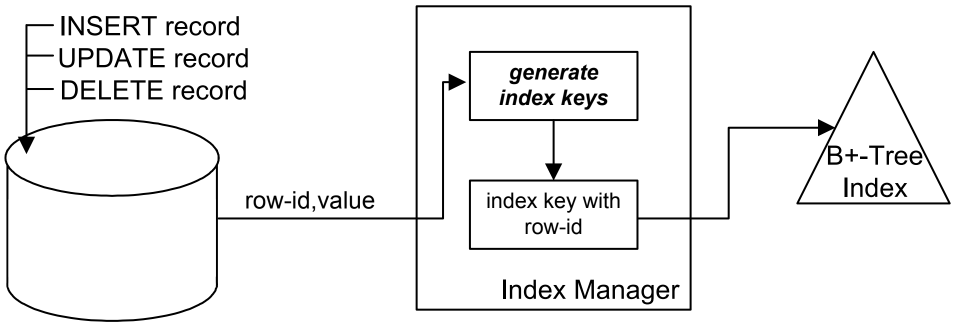


Figure 2: Index maintenance (cf. [SS03])

row to be produced, which internally is associated with a row identifier. The value of the indexed attribute and the row identifier are given to the Index Manager, which generates the corresponding index keys using the *key generator* that has to be provided by the index implementor. This *key generator*, a user-defined table function, has to be provided since DB2 does not know the semantics of the user-defined type. Each of the generated keys is stored, together with the row-identifier, in the corresponding B+-tree. Updates on rows and deletions will be handled accordingly. Note that it is possible to include multiple index entries for a single object.

2.2 Index Exploitation

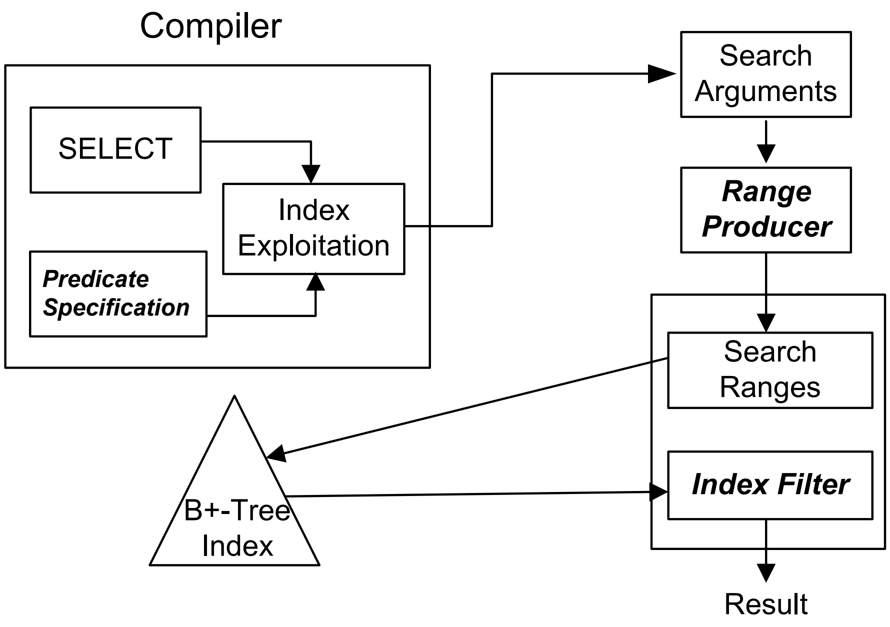


Figure 3: Index exploitation [SS03]

Index exploitation to answer queries is depicted in Figure 3. After issuing an SQL query containing a predicate, which is defined to be evaluated using index exploitation, the optimizer may choose an execution plan that accesses the index. The system does not know which predicates could benefit from index exploitation. The information about these predicates is to be attached to user-defined functions.

The search arguments are extracted from the query and passed to the *range producer* which is the second procedure the index implementor has to provide to the system. The *range producer*, a user-defined table function, generates the appropriate search ranges for the index scan, and passes them to the index manager. The index manager performs the index scan using these ranges. In cases where the index scan might not give the exact result, but only a super-set of all qualifying rows, an *index filter* can be defined to discard non-qualifying rows.

2.3 Characteristics of the Framework

There are two characteristic restrictions of the DB2 UDB extensible indexing framework a user defined index implementor is faced with, namely the usage of a single B+-tree only, and the static character of the index entries.

The first restriction of the framework is that it uses a single B+-tree to store the index keys, meaning that the indexing scheme to be implemented must not only be a relational storage structure, but also be applicable using a single B+-tree. Index structures that cannot map their index entries into a single B+-tree are excluded by the framework. The RI-tree, at first sight, shares this limitation but, in this paper, we extend the relational mapping to overcome this problem.

The second restriction is that once the entries are stored, they cannot be altered, which means that index entries representing an interval cannot be changed during index usage. All indexing structures which need to perform any kind of reorganization during index usage are therefore excluded by the framework. The SRPS-tree [AC03] for example maintains an ordering on the *lower* values of the stored intervals and is therefore not applicable. Another thing worth mentioning is that, since all user-defined functions may not access the database system, but only static parameters passed during index creation, the scheme may only use static metadata. The index implementor chooses which parameters must be set when creating the index, those parameters are passed to the user-defined functions by each invocation, but may not be altered by them.

3 Relational Mapping of the Interval Tree Structure

The RI-tree introduced in [KPS00] is a relational storage structure for interval data (*lower*, *upper*) which can be built on top of the SQL layer of any RDMBS. By design, it follows the concept of Edelsbrunner's main-memory interval tree [Ed80] and guarantees the optimal complexity of $O(n/b)$ for storage space and $O(\log_b n)$ for update operations in large sets of n intervals stored in blocks of size b. In case of a fixed data space and interval resolution (i.e. a fixed number h of significant bits for interval boundaries, corresponding to the height of the binary backbone), intersection query processing is performed in optimal $O(h \cdot \log_b n + t/b)$ I/O operations, where t is the number of returned intervals.

3.1 Relational Storage

The RI-tree strictly follows the paradigm of relational storage structures since its implementation is purely built on (procedural and declarative) SQL but does not require any lower level interfaces to the database system. In particular, built-in index structures are used as they are, and no intrusive augmentation or modification of the database kernel is required. On top of its pure relational implementation, the RI-tree is ready for immediate object-relational wrapping. It fits particularly well to extensible indexing frameworks as introduced above which enable developers to extend the set of built-in index structures by custom access methods in order to support user-defined data types and predicates. Some of the restrictions given by the DB2 extensible indexing framework mentioned above in Section 2 make modifications to the RI-tree index inevitable. With the RI-tree, all queries and updates on relational storage structures are processed by pure SQL statements. The robust transaction semantics of the database server is therefore fully preserved.

3.2 Dynamic Data Structure

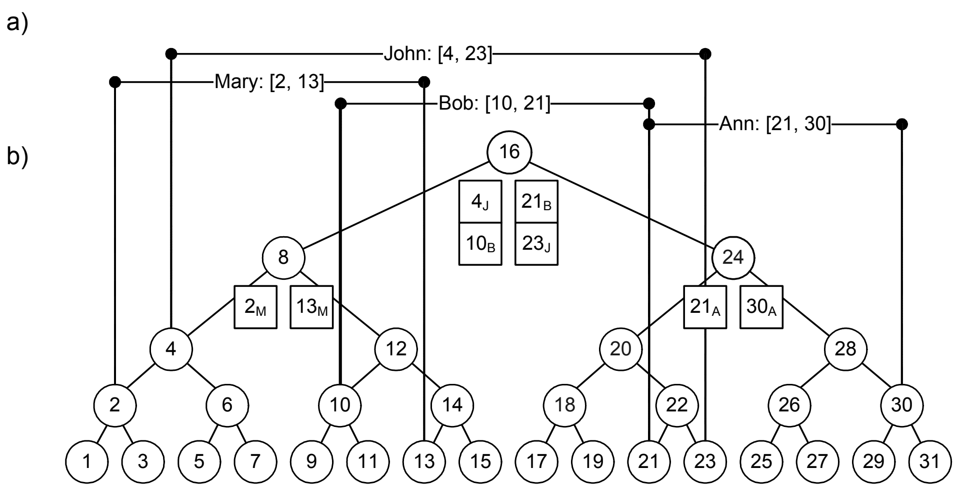
The structure of an RI-tree resembles a binary tree of height h which covers the range $[1, 2^h-1]$ of potential interval bounds. It is called the virtual backbone of the RI-tree since it is not materialized but only the root value 2^{h-1} is stored persistently in a metadata table. Traversals of the virtual backbone are performed purely arithmetically by starting at the root value and proceeding in positive or negative steps i of decreasing length 2^{h-i}, thus reaching any desired value of the data space in $O(h)$ CPU time and without causing any I/O operation. For the naïve relational storage of intervals, the node values of the tree are used as artificial keys: Upon insertion of an interval, the first node that hits the interval when descending the tree from the root node down to the interval location is assigned to that interval.

In the solution presented in [KPS00], an instance of the RI-tree consists of two relational indexes. The indexes obey the relational schema *lowerIndex* (*node*, *lower*, *id*) and *upperIndex* (*node*, *upper*, *id*) and store the artificial key value *node*, the bounds *lower* and *upper*, respectively, and the *id* of each interval. Both of these indexes are defined by using the same relational schema. An interval is represented by a single entry in each of the two indexes, and therefore, $O(n/b)$ disk blocks of size b suffice to store n intervals. For inserting or deleting intervals, the *node* values are determined arithmetically, and updating the indexes requires $O(\log_b n)$ I/O operations per interval.

The illustration in Figure 4 provides an example for the RI-tree. Let us assume the intervals (2,13) for Mary, (4,23) for John, (10,21) for Bob, and (21,30) for Ann (Figure 4a). The virtual backbone is rooted at 16 and covers the data space from 1 to 31 (Figure 4b). The intervals are registered at the nodes 8, 16, and 24, respectively. The interval (2,13) for Mary is represented by the entries (8, 2, Mary) in the *lowerIndex* and (8, 13, Mary) in the *upperIndex* since 8 is the registration node, and 2 and 13 are the lower and upper bound, respectively (Figure 4c).

3.3 New Relational Mapping for the DB2 Extensible Indexing Framework

This original mapping of the RI-tree defined in [KPS00] cannot be directly implemented using the DB2 extensible indexing framework. The framework only allows the handling of a single B+-tree index, but the original solution requires two index tables, namely



c) lowerIndex (node, lower, id):

8, 2, Mary	16, 4, John	16, 10, Bob	24, 21, Ann

upperIndex (node, upper, id):

8, 13, Mary	16, 21, Bob	16, 23, John	24, 30, Ann

Figure 4: Example of an RI-tree.
a) four sample intervals. b) virtual backbone and registration positions of the intervals. c) resulting relational indexes *lowerIndex* and *upperIndex*

lowerIndex and *upperIndex*. Therefore it is inevitable to apply fundamental structural changes to the index creation and query handling.

In order to achieve the above requirements, we developed a new relational mapping, which utilizes the idea of partitioned B+-trees proposed in [Gr03]. The original idea of partitioning a B+-tree by adding an additional flag to each index entry is slightly modified in order that we will not need the flag for partitioning and therefore prevent additional storage overhead. Since we must use a static version of the RI-tree, due to the restriction to static metadata, the primary structure covers the range $[1, 2^h-1]$ which means that all intervals will be registered at positive nodes. We will use the symbol *node* below, as an abbreviation for *forknode*(*lower*, *upper*). In the original RI-tree design each interval would result in one entry in each of the two indexes namely (*node*, *lower*) and (*node*, *upper*).

Our new approach is to partition the single B+-tree from the DB2 framework into a positive and negative side by assigning a sign to the fork node value. The goal is to use these partitions to manage the *lowerIndex* and the *upperIndex* simultaneously. To achieve this we use a composite index (*node*, *bound*), so each interval results in two index entries in the partitioned B+-tree, namely (−*node*, *lower*) and (*node*, −*upper*). Since the computed forknodes are positive, all entries for upper bounds are stored in the positive partition and accordingly all entries for lower bounds in the negative partition, hence the positive partition offers a composite index on (*node*, −*upper*) and the negative partition a composite index on (−*node*, *lower*). Figure 5 presents the SQL statements to be executed when inserting an interval, assuming the B+-tree is realized as index-organized table. Let us note that we switch the sign of the upper bound value too, for reasons explained below when discussing the simplification of query processing.

```
INSERT INTO boundIndex VALUES (−node, lower, id)
INSERT INTO boundIndex VALUES (node, −upper, id)
```

Figure 5: Insertion of an interval

4 Adaptation of Query Processing

We have seen above that the RI-tree in its original form uses two relational indexes to map its structure to the RDBMS. Using our new relational mapping we are now able to map all necessary information into a single B+-tree. In order to utilize our new approach we have to adapt the intersection query processing to the new mapping and, as a result of this, will be able to simplify it.

4.1 Original Intersection Query Processing

Conceptually, an interval intersection query (*lower*, *upper*) is processed in two steps: The procedural query preparation step descends the virtual backbone from the root node down to *lower* and to *upper*, respectively (Figure 6). The traversal is performed arithmetically without causing any I/O operations, and the visited nodes are collected in two tables, *leftQueries* and *rightQueries* both obeying the unary relational schema (*node*), as

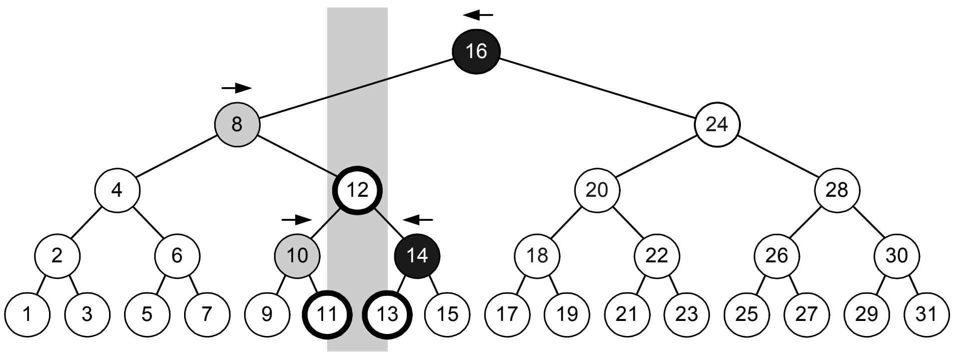


Figure 6: Query preparation step for the query interval [11,13] (shaded in light gray): *leftQueries* {8,10}; *rightQueries* {14, 16}; *innerQueries* {11-13}

follows: nodes to the left of *lower* may contain intervals which overlap *lower* and are inserted into *leftQueries*. These tables were handled as transient tables in [KPS00]. Analogously, nodes to the right of *upper* may contain intervals which overlap *upper* and are inserted into *rightQueries*. Whereas these nodes are taken from the paths, the set of all nodes between *lower* and *upper* belongs to the so-called *innerQueries* which are represented by a single range query on the node values. All intervals registered at nodes from the *innerQueries* are guaranteed to intersect the query since they have at least their node value in common with the query interval and, therefore, will be reported without any further comparison of lower or upper values. The query preparation step is purely based on main memory and requires no I/O operations.

```
SELECT i.id FROM boundIndex i, leftQueries left
        WHERE i.node=left.node AND i.bound >= :lower
UNION ALL
SELECT i.id FROM boundIndex i, rightQueries right
        WHERE i.node= right.node AND i.bound <= :upper
UNION ALL
SELECT i.id FROM boundIndex i
        WHERE i.node BETWEEN :lower AND :upper
```

Figure 7: SQL statement for an intersection query with bind variables for *leftQueries*, *rightQueries*, *lower* and *upper*

In the subsequent declarative query processing step, the transient tables are joined with the relational indexes *upperIndex* and *lowerIndex* by a single, originally three-fold SQL statement (Figure 7). The upper bound of each interval registered at nodes in *leftQueries* is compared to *lower*, and the lower bounds of intervals in *rightQueries* are compared to *upper*. The *innerQueries* correspond to a simple range scan over the intervals with nodes in (*lower*, *upper*). The SQL query requires $O(h \cdot \log_b n + r/b)$ I/Os to report *r* results from an RI-tree of height *h* since the output from the relational indexes is fully blocked for each join partner.

4.2 Simplified Query Processing

Since we are using a single index now, query processing has to be adapted accordingly; *rightQueries* are processed on the negative partition, *leftQueries* and *innerQueries* on the positive partition. To achieve this, all nodes in *rightQueries* are stored with negative sign, nodes in *leftQueries* and *innerQueries* simply positive. Figure 8 shows the adapted SQL query to retrieve all intersection intervals.

As a result of our new relational mapping to the RDBMS we are now able to simplify the processing of intersection queries. Note that all three subqueries above use the same single *boundIndex*, which allows us to simplify the three-fold SQL query to a single-fold one. Figure 9 presents the resulting single-fold SQL query to retrieve all intersecting intervals. The respective modifications are described in the following.

```
SELECT i.id FROM upperIndex i, leftQueries q
        WHERE i.node = q.node AND i.upper <= -(:lower)
UNION ALL
SELECT i.id FROM lowerIndex i, rightQueries q
        WHERE i.node = q.node AND i.lower <= :upper
UNION ALL
SELECT i.id FROM lowerIndex i   // or upperIndex
        WHERE i.node BETWEEN :lower AND :upper;
```

Figure 8: Intermediate version of transformed query statement

The first modification is the most obvious, for *leftQueries* we need a "larger than" comparison between the stored upper values and the lower value of the query. To flip this into a "smaller than" comparison we add a negative sign on both sides. We will do this in the following for the *lower* value of the query by adding *–lower* for nodes which belonged to *leftQueries*, and to do the same with the *upper* value of the stored intervals, we simple store them as (*node*, −*upper*) into the index. The remaining modifications are as follows.

```
SELECT id FROM boundIndex i, allQueries q
        WHERE i.node BETWEEN q.min AND q.max
                AND i.bound <= q.bound
```

Figure 9: Final SQL statement for intersection queries

The transient relations *leftQueries* and *rightQueries* are merged together forming a new transient relation *allQueries*, which obeys the ternary relational schema (*min*, *max*, *bound*). Nodes which formerly belonged to *leftQueries*, are inserted as triple (*w*, *w*, *-lower*) rather than a single value (*w*) as before. The correctness of this transformation is obvious since the condition '*i.node=left.node*' may be substituted by the equivalent condition '*i.node BETWEEN allQueries.min AND allQueries.max*' and setting *allQueries.min=allQueries.max*'. The resulting index scan searches the first hit by testing *allQueries.min* $\leq$ *i.node* and proceeds while testing the condition *i.node* $\leq$ *allQueries.max*. For nodes which belonged to *rightQueries*, we insert the pair (−*w*, −*w*, *upper*) rather than just the value (*w*). This transformation is obviously also correct following the same argumentation as for nodes which formerly belonged to *leftQueries*. Finally, to include the original BETWEEN subquery, the pair (*lower*, *upper*, −*lower*) is inserted into *allQueries*. This final transformation can also be shown to be correct since by definition, *i.node* $\leq$ *i.upper*, holds for any interval *i* in the tree, the condition :*lower* $\leq$ *i.node* implies :*lower* $\leq$ *i.upper* and thus '−(*i.upper*) $\leq$ −(:*lower*)'.

Speaking in terms of the DB2 Index Extension, the index entries (−*node*, *lower*) and (*node*, −*upper*) for each interval will be produced by the *key generator* and the transient relation *allQueries* will be the result of the table function *range producer*. The *key generator* delivers, for a given interval, the two tuples (−*node*, *lower*) and (*node*, −*upper*). As we have seen above, the *range producer* is a table function which returns a set of search ranges for the index, which means that we need to specify search ranges for each of the components of our index keys in order to use the index. The *q.min* and *q.max* components of *allQueries* already define a range, but *q.bound* does not. In order to replace the *q.bound* component with an equivalent range, we will use (*1*, *q.upper*) for *rightQueries* and ($-(2^h-1)$, $-q.lower$) for *leftQueries* and the BETWEEN subquery. Both ranges are obviously equivalent to the one-sided inequality.

4.3 Preliminary Solution

Another attempt to implement the RI-tree using DB2 extensible indexing was presented in [SS03]. It uses a single index on the node values of the indexed intervals and also a static version of the RI-tree. This solution has the shortcoming that optimality cannot be achieved because information about the interval boundaries is not regarded in the index scan, and thus the scan delivers a superset of the exact result, which is filtered inside the extension.

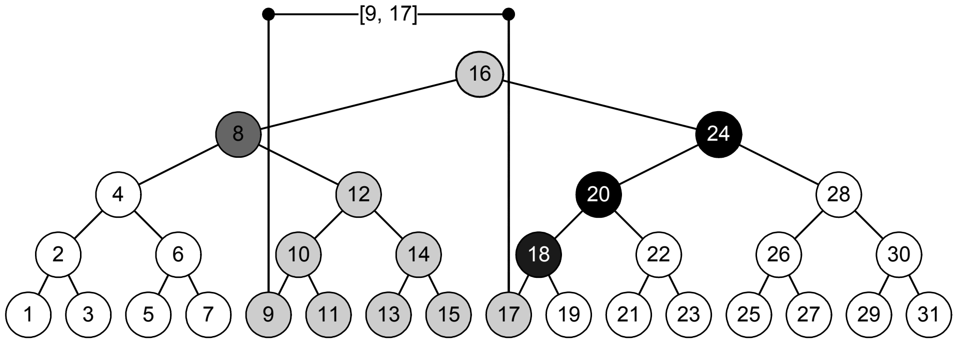


Figure 10: Worst case scenario for the preliminary implementation

In a worst case scenario depicted in Figure 10, the whole dataset is read without producing a single result. The effect in regular scenarios will be analyzed in the experimental evaluation. In our scenario all indexed intervals start at 18, the ending is arbitrary. This means all intervals will be registered at the nodes 18, 20 or 24. An intersect query using (*9*, *17*) yields *leftQueries* (dark grey), *innerQueries* (light grey) and *rightQueries* (black). Since no information about interval boundaries is regarded in the index scan, all intervals registered at these nodes are compared and the index pages must be loaded. Unfortunately, the nodes in *rightQueries* match those that contain all intervals and therefore the whole set of index pages is read. Let us in addition mention the observation that in general all intervals registered at the root node are accessed for any query posed to the RI-tree.

Our new optimal mapping will need slightly more space by using two index entries for each interval, but no additional queries to the relational index. One is therefore able to achieve an optimal behavior by providing slightly more storage space.

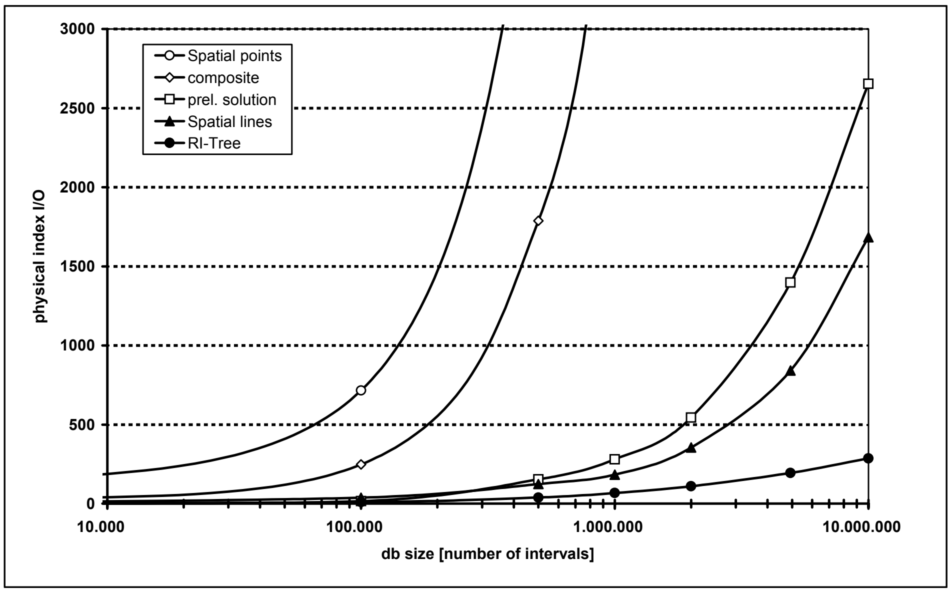


Figure 11: Disk accesses for varying database sizes

5 Experimental Evaluation

5.1 Experimental Setup

To evaluate the performance of our approach, we have integrated the adapted RI-tree into DB2 UDB Version 8.1 (Fixpak 6) and compared its performance to other indexing techniques available. Our code is publicly available on the web [RIT04]. All experiments have been executed on a dual Xeon/3 GHz Server having 4 GB main memory and a U-SCSI hard drive. The database cache was reduced to 1000 pages with a page size of 4 KB. In all experiments the bounding points of the intervals range over the domain [1, $2^{20}-1$]. Starting points were uniformly distributed in [1, $2^{20}-1$] while the interval length was uniformly distributed in [1, 4k]. The query intervals used for the comparisons were distributed analogously with varying ranges for length.

We compared the performance of the RI-tree to the following techniques: the preliminary solution presented in [SS03], a built-in composite index using (*lower*, *upper*) and the DB2 Spatial Extender using two different approaches to represent intervals in a 2-D space. For the first representation we stored intervals [*a*, *b*] as points (*a*, *b*) in the 2-D space, in the second one we stored intervals [*a*, *b*] as lines (*a 1*, *b 1*) with a fixed second dimension. The minimum, medium and maximal grid sizes, which were required as parameters for the DB2 Spatial Extender, were set to (*1*, *0*, *0*) for the point representation and to (*50*, *250*, *1350*) for the line representation.

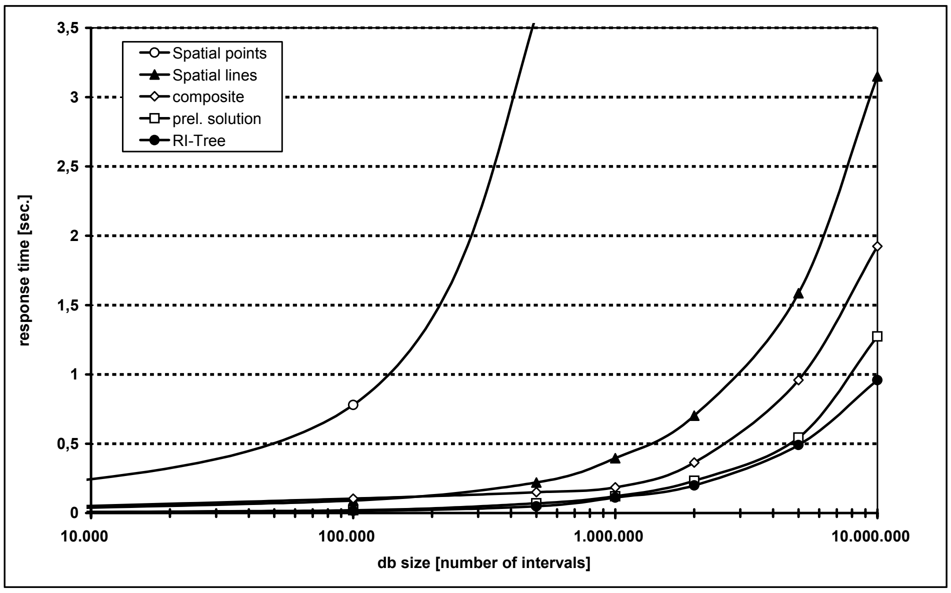


Figure 12: Response time for varying database sizes

5.2 Performance with varying database sizes

In our first set of experiments we compared the performance of the different solutions according to varying sizes of the database. Figure 11 depicts the number of physical accesses to index pages on datasets growing from 10.000 to 10.000.000 intervals. For each database size the average physical index page accesses of 20 intersection queries is presented, the average selectivity of the queries was 0.4%.

The RI-tree clearly outperforms all the other techniques. For a database size of 10.000.000 intervals, the speedup factor from the second best result, the Spatial Extender using the line representation, to the RI-tree is 5.9. The speedup to the preliminary solution, by a factor of 9.3, clearly shows the benefit of including information about the interval boundaries into the index scan. The benefit is not only important for ensuring optimality in theoretical worst case scenarios, but clearly significant under regular circumstances. The composite index and the Spatial Extender using the point representation perform poor with respect to the physical index accesses.

Figure 12 depicts the corresponding response time for the different sizes of the database. The results are somewhat different, while the RI-tree still performs best and the Spatial Extender using the point representation performs worst, the preliminary solution performs nearly as good as the RI-tree. This is due to the fact that the response time is dominated by the effort of subsequently fetching the result rows from the base table, since after the index scan both will fetch the same rows and the I/O needed to fetch these objective rows is much larger then the index I/O, even for a selectivity as high as 0.4%,

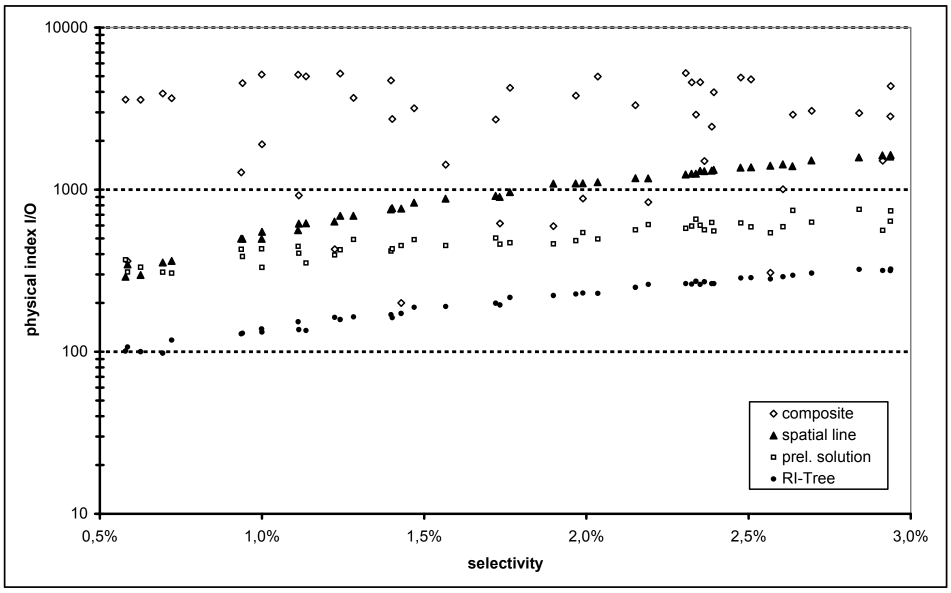


Figure 13: Disk accesses for varying query selectivity

the overall response time is more similar. The same argument applies for the speedup factor between RI-tree and the Spatial Extender using the line representation, whereas the enhancement is not as big as with the preliminary solution, the speedup factor to the RI-tree has still a value of 3 at a database size of 10.000.000 intervals.

The built-in composite index also performs better when regarding response time. Since it is integrated into the database core and uses a highly tuned implementation, it can compensate the high number of physical index page accesses. All other techniques are implemented through the extensible index interface which causes overhead. Nonetheless, the speedup factor from the composite index to the RI-tree has a value of 2 at a database size of 10.000.000 intervals.

Since the line representation of intervals has shown to be superior to the point representation, we will not include the point representation in all our further experiments.

5.3 Performance with varying query selectivity

The next set of experiments investigates the influence of the query selectivity on the query performance. Figure 13 depicts the number of physical accesses to index pages for a selectivity varying between 0.5% and 3%. The database size was fixed at 1.000.000 intervals, for each indexing technique the results of 50 range queries are presented. Due to the performance of the composite index we chose a logarithmic scale for the accesses.

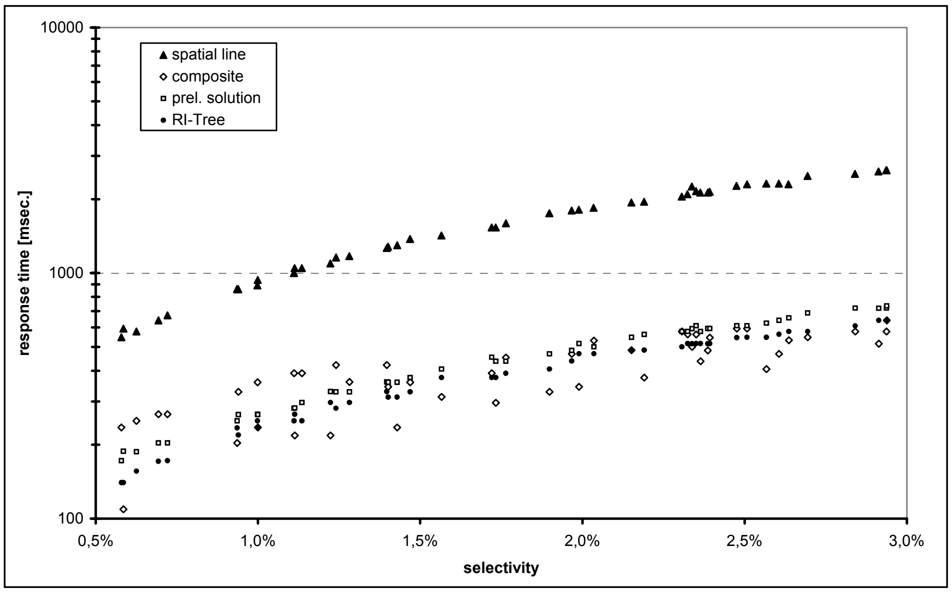


Figure 14: Response time for varying query selectivity

Once again the RI-tree outperforms all other techniques, the number of physical accesses is highly predictable and grows sublinear with respect to the query selectivity, with about 100 accesses for 0.6% selectivity and about 320 accesses for 2.9% selectivity. The Spatial Extender shows a linear behavior, with page accesses growing from about 300 for 0.6% selectivity to about 1600 accesses for 2.9% selectivity. The preliminary solution outperforms the Spatial Extender at selectivities larger than 1% and shows a similar behavior as the RI-tree at a level of accesses about 100% higher. The performance of the composite index is highly unpredictable, with the number of page accesses ranging between 200 up to about 5000. It is independent of the selectivity of the query, since the range scanned during query processing depends only on the location of the query, meaning a query with very high selectivity which is located at the end of the dataspace will nevertheless result in a full scan of the composite index.

Figure 14 depicts the corresponding response times for the 50 intersection queries. The RI-tree shows a similar behavior as for disc accesses, with the difference that the response time grows linearly which is no surprise since the response time is, as mentioned before, dominated by the effort to fetch result rows from the referenced base table. The preliminary solution also performs well, the gap is not as big as with index page accesses for the same reason. The composite index shows the same unpredictable behavior as with physical page accesses, while it can compensate for the high number of accesses because of the above mentioned reasons.

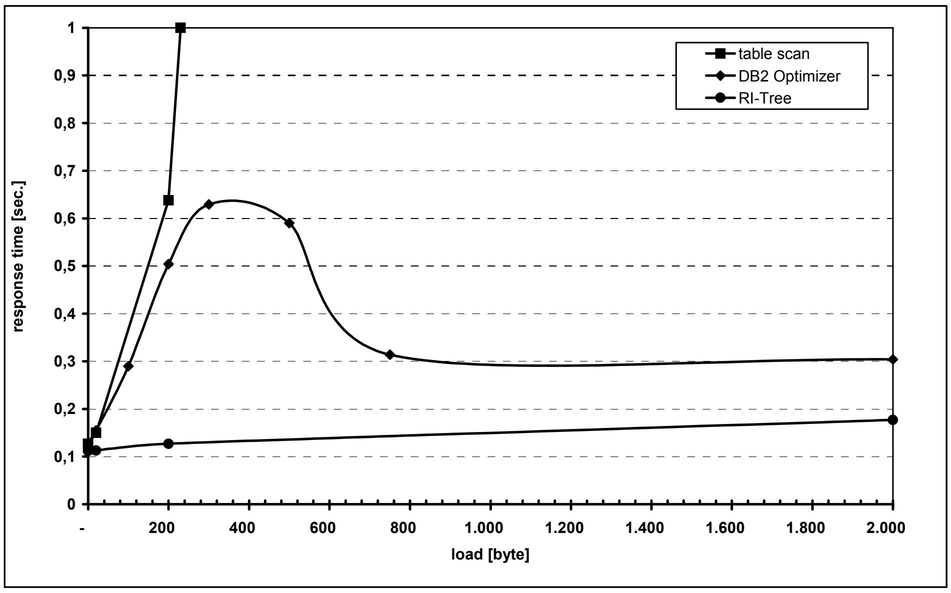


Figure 15: Response time using different amounts of load attached to each tuple

5.4 Simulating real life data

In the last experiment we evaluated the performance while simulating real life circumstances. Since in a real life database, each tuple of the indexed relation will hold additional information besides an interval, we evaluated the performance using different amounts of load attached to each tuple. Figure 15 depicts the mean response time for loads ranging from 0 byte to 2 kbytes. For each load the mean value of 20 queries is presented, the database size was fixed at 1.000.000 intervals, the average query selectivity was 0.4%. We compared the performance of the best performing indexing structure the RI-tree to the built-in possibilities, the table scan and the composite index whereas we let the DB2 Optimizer choose whether to use it or not.

The performance of the RI-tree is not affected by the amount of attached load since each result is fetched using a single random access to the disk no matter how large the tuple is. In contrast to that, the table scan is heavily affected by the attached load and is unusable for higher values. The DB2 Optimizer starts using the composite index at about 200 byte load, which means the 20 queries are partially answered by a table scan and partially by using the composite index. The percentage of queries answered by the composite index grows with higher loads, until at about 750 byte all queries will be answered by the composite index, even then the RI-tree will remain about 40% faster.

6 Related work

Several different approaches to provide efficient interval intersection query processing already exist in the literature, especially in the field of temporal database applications. Beside main memory structures that are inappropriate for use in persistent databases, many secondary storage structures have been proposed: the Time Index of Elmasri, Wuu and Kim [EWK90], the Interval B-tree of Ang and Tan [AT95], the Interval B+-tree of Bozkaya and Özsoyoglu [BÖ98], and the TP-Index of Shen, Ooi and Lu [SOL94], for example. Unfortunately, as these access methods typically are based either on the augmentation of existing indexes or on the definition of new structures, they share the limited support for an integration into existing systems. When being committed to a an industrial-strength ORDBMS like DB2, the structures cannot be integrated as the built-in indexes are not extensible by the user. We therefore restrict our considerations to relational storage structures that use built-in index structures the way they are rather than to augment indexes or to introduce new structures whose integration is typically not supported by existing RDBMS.

The SRPS-tree of Arge and Chatham [AC03] is an efficient index structure capable of efficiently answering all general interval relationship queries. Intervals $[a, b]$ are stored as points (a, b) in 2-D. Similar to the RI-tree, the SRPS-tree uses a virtual backbone, namely a binary tree on the *lower* values of the stored intervals. In order to achieve optimality, an ordering regarding the *upper* values is employed. Therefore the index key representing a certain interval may change during insertions into, deletions from or updates on the referenced table which leads to continuous reorganization during usage. This fact prevents the SRPS-tree from being applicable using DB2 extensible indexing although it uses a single index.

The *Interval-Spatial Transformation* (*IST*) of Goh et al. [Go96] is based on encoding intervals by space-filling curves called *D-*, *V-* and *H-ordering* that map the boundary points into a linear space. No redundancy is produced, and space complexity is $O(n/b)$. The I/O complexity of the query algorithm linearly depends on the resolution of the space whereas our method guarantees a logarithmic dependency on the resolution. The structure shows a strong correspondence to relational composite indexes. Aside from quantization aspects, the D-ordering is equivalent to a composite index on the interval bounds (*upper*, *lower*), and the V-ordering corresponds to an index on (*lower*, *upper*) what we have used for our experiments. For intersection queries, these indexes reveal a poor query performance if the selectivity relies on the “wrong” bound, i. e. the secondary attribute in the index. Thus, intersection queries have a worst case I/O complexity of $O(n/b)$.

The *MAP21* approach of Nascimento and Dunham [ND99] behaves very similar to the IST while the composite index (*lower*, *upper*) is implemented by a single-column index. A static partitioning by the interval lengths is introduced, but intersection query processing still requires $O(n/b)$ I/Os if the database contains many long intervals.

The *Window-List* technique of Ramaswamy [Ra97] is a static solution for the interval management problem and employs built-in B+-trees. The optimal complexity of $O(n/b)$

space and $O(\log_b n + r/b)$ I/Os for stabbing queries is achieved. Unfortunately, updates do not seem to have upper bounds, and adding as well as deleting arbitrary intervals can deteriorate the query efficiency of this structure to $O(n/b)$.

7 Conclusions

Following the principle of relational indexing, the Relational Interval Tree is an approved access method for interval data that is entirely built on top of the SQL interface of a relational database system. It fits particularly well to extensible indexing frameworks which enable developers to extend the set of built-in index structures by custom access methods in order to support user-defined data types and predicates. In this paper, we discuss our modifications of the RI-tree when integration it into the IBM DB2 server using the recently released DB2 indexing interface. By design, this framework holds a single B+-tree to store the index keys, meaning that the indexing scheme to be implemented must not only be a relational storage structure, but also be applicable using a single B+-tree. So we had to adapt the original implementation of the RI-tree that manages the lower and upper bounds of stored intervals separately using two B+-trees. By mapping the two trees on a single partitioned B+-tree and performing according changes to the RI-tree query statements, we were able to overcome this limitation of the indexing framework. Experimental comparisons with other techniques using built-in composite indexes and the DB2 Spatial Extender confirmed the excellent performance of the RI-tree for interval intersection queries.

References

[AC03] Arge, L.; Chatham, A.: *Efficient Object-Relational Interval Management and Beyond*. Proc. 8th Int. Symp. on Spatial and Temporal Databases, 2003; pp. 66-82.

[Ad01] Adler, D. W.: *DB2 Spatial Extender - Spatial data within the RDBMS*. Proc. 27th Int. Conf. on Very Large Databases, 2001; pp. 687-690.

[AT95] Ang, C.-H.; Tan K.-P.: *The Interval B-Tree*. Information Processing Letters 53(2), 1995; pp. 85-89.

[Bl99] Bliujute, R.; Saltenis, S.; Slivinskas, G.; Jensen, C. S.: *Developing a DataBlade for a New Index*. Proc. 15th Int. Conf. on Data Engineering, 1999; pp. 314-323.

[BÖ98] Bozkaya, T.; Özsoyoglu, Z. M.: *Indexing Valid Time Intervals*. Proc. 9th Int. Conf. on Database and Expert Systems Applications. LNCS 1460, 1998; pp. 541-550.

[Ch99] Chen, W.; Chow, J.-H.; Fuh, Y.-C., Grandbois, J.; Jou, M.; Mattos, N. M.; Tran, B. T.; Wang, Y.: *High Level Indexing of User-Defined Types*. Proc. 25th Int. Conf. on Very Large Data Bases, 1999; pp. 554-564.

[Ed80] Edelsbrunner, H.: *Dynamic Rectangle Intersection Searching*. Inst. for Information Processing Report 47, Technical University of Graz, Austria, 1980.

[EWK90] Elmasri, R.; Wuu, G. T. J.; Kim Y.-J.: *The Time Index: An Access Structure for Temporal Data*. Proc. 16th Int. Conf. on Very Large Databases, 1990; pp. 1-12.

[FR89] Faloutsos, C.; Roseman, S.: *Fractals for Secondary Key Retrieval*. Proc. 8th ACM Symp. on Principles of Database Systems, 1989; pp. 247-252.

[Go96] Goh, C. H.; Lu, H.; Ooi, B. C.; Tan, K.-L.: *Indexing Temporal Data Using Existing B+-trees*. Data & Knowledge Engineering, Elsevier, 18(2), 1996; pp. 147-165.

[Gr03] Graefe G.: *Partitioned B-trees - a user's guide*. Proc. 10th GI-Conf. on Database Systems for Business, Technology, and the Web (BTW), Februar 2003, Leipzig. LNI 26 GI 2003: pp. 668-671.

[IBM02] IBM Corp.: *IBM DB2 Universal Database Application Development Guide, Version 8*, Armonk, NY, 2002.

[IBM03] IBM Corp.: *IBM Informix Virtual-Index Interface Programmer's Guide*, *Ver. 9.4*. Armonk, NY, 2003.

[ISO03a] ISO/IEC 9075-2:2003: *Information Technology – Database Languages – SQL – Part 2: Foundation (SQL/Foundation)*, 2003.

[ISO03b] ISO/IEC 9075-3:2003: *Information Technology – Database Languages – SQL Multimedia and Application Packages – Part 3: Spatial*, 2003.

[Ko99] Kornacker, M.: *High-Performance Extensible Indexing*. Proc. 25th Int. Conf. on Very Large Databases, 1999; pp. 699-708.

[KPS00] Kriegel, H.-P.; Pötke, M.; Seidl, T.: *Managing Intervals Efficiently in Object-Relational Databases*. Proc. Int. Conf. on Very Large Data Bases 2000: 407-418.

[KPS01] Kriegel, H.-P.; Pötke, M.; Seidl, T.: *Interval Sequences: An Object-Relational Approach to Manage Spatial Data*. Proc. 7th Int. Symp. on Spatial and Temporal Databases, 2001; pp. 481-501.

[Kr03] Kriegel, H.-P.; Pfeifle, M.; Pötke, M.; Seidl, T.: *The Paradigm of Relational Indexing: a Survey*. Proc. 10th Conf. on Database Systems for Business, Technology and Web, LNI P-26, 2003; 285-304.

[Kr04] Kriegel, H.-P.; Pfeifle, M.; Pötke, M.; Seidl, T.; Enderle, J.: *Object-Relational Spatial Indexing*. To appear in: Manolopoulos, Y.; Papadopoulos A.; Vassilakopoulos, M. (eds.): *Spatial Databases: Technologies, Techniques and Trends*. Idea Group Inc., 2004

[ND99] Nascimento, M. A.; Dunham, M. H.: *Indexing Valid Time Databases via B+-Trees*. IEEE Trans. on Knowledge and Data Engineering 11(6), 1999; pp. 929-947.

[Ora04] Oracle Corp.: *Oracle Data Cartridge Developers Guide, 10g Release 1 (10.1.0.2.0)*. Redwood City, CA, 2004.

[Ra97] Ramaswamy, S.: *Efficient Indexing for Constraint and Temporal Databases*. Proc. 6th Int. Conf. on Database Theory, LNCS 1186, 1997; pp. 419-431.

[RIT04] http://www-i9.informatik.rwth-aachen.de/ritree

[SA85] Snodgrass, R. T.; Ahn, I.: *A Taxonomy of Time in Databases*. SIGMOD Conference 1985, pp. 236-246.

[SOL94] Shen, H.; Ooi, B. C.; Lu, H.: *The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases*. Proc. IEEE Int. Conf. on Data Engineering, 1994; pp. 274-281.

[Sr00] Srinivasan, J.; Murthy, R., Sundara, S.; Agarwal, N.; DeFazio, S.: *Extensible Indexing: A Framework for Integrating Domain-Specific Indexing Schemes into Oracle8i*. Proc. 16th Int. Conf. on Data Engineering, 2000; pp 91-100.

[SS03] Steinbach, T.; Stolze, K.: *DB2 Index Extensions by example and in detail*. DB2 Developer Domain, December 2003.

[St86] Stonebraker M.: *Inclusion of New Types in Relational Data Base Systems*. Proc. Int. Conf. on Data Engineering 1986, pp. 262-269.

[St03] Stolze, K.: *SQL/MM Spatial - The Standard to Manage Spatial Data in a Relational Database System*. Proc. 10th BTW Conf., LNI P-26, 2003; 247-264.