Efficiently Processing Queries on Interval-and-Value Tuples in Relational Databases

Jost Enderle Nicole Schneider Thomas Seidl

RWTH Aachen University, Germany
Department of Computer Science 9
{enderle, schneider, seidl}@informatik.rwth-aachen.de

Abstract

With the increasing occurrence of temporal and spatial data in present-day database applications, the interval data type is adopted by more and more database systems. For an efficient support of queries that contain selections on interval attributes as well as simple-valued attributes (e. g. numbers, strings) at the same time, special index structures are required supporting both types of predicates in combination. Based on the Relational Interval Tree, we present various indexing schemes that support such combined queries and can be integrated in relational database systems with minimum effort. Experiments on different query types show superior performance for the new techniques in comparison to competing access methods.

1. Introduction

In recent years, many special-purpose data types got available in relational databases. Instead of providing application-specific systems such as spatial, multimedia, or XML databases for each conceivable domain, database implementors more and more introduce new data types and corresponding management functions directly into relational systems, either by a direct integration into the database kernel or – much more common – by providing extensible interfaces allowing users to define the required complex types together with appropriate operations.

One of the simplest complex types is the interval data type. While intervals occur in various application areas, e. g. as tolerance ranges for imprecisely measured values in scientific databases, as line segments on a space-filling curve in spatial applications [1] [11], or as finite domain constraints in declarative systems [18] [26], the typical domain for intervals are temporal applications where they are used as transaction time and valid time ranges [30].

In a relational database system adopting the SQL:2003 [16] *period* data type (defined as a compound *row* object with starting and ending *datetimes*), we could define, e. g., a *contracts* table storing the contract identifier *c_no*, the available budget *c_budget* and the interval-valued period *c_period* of a contract tuple by the following DDL statement:

```
CREATE TABLE contracts (
      c_no       VARCHAR(10),
      c_budget   DECIMAL(10,2),
      c_period   ROW (c_start DATE, c_end DATE))
```

Figure 1: Statement to create a table with an interval attribute

In practice, queries on such tables often contain selections on the interval-valued as well as on the simple attributes at the same time, as in the following query:

```
SELECT c_no
FROM   contracts
WHERE  c_budget BETWEEN :v1 AND :v2
       AND c_period OVERLAPS
       (DATE '2005-01-01', DATE '2005-01-31')
```

Figure 2: Query selecting single- and interval-valued attributes in combination

Present-day relational database systems still provide a very limited selection of access structures, normally just indexes for simple data types (e. g. B-trees, hash tables) and sometimes also for spatial data types (e. g. R-trees). To support combined queries as in Figure 2, one could create a B-tree index on the simple attribute and an R-tree or composite index on the interval attribute. Unfortunately, interval queries are not very well supported by spatial access structures or composite indexes, especially

**Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005**

in the presence of intervals with long durations and high overlaps. Furthermore, the combined query has to be evaluated by an intersection of the inverted lists provided by the single indexes which results in additional overhead. Using a single spatial or composite index on the simple attribute and the interval bounds avoids the final intersection step, but disregards the individual characteristics of the different data types.

Several different approaches to support overlap queries on interval data exist in the literature, some of them also admitting the combined evaluation of selections on simple attributes (cf. Section 2). However, these algorithms usually are based either on the augmentation of existing indexes or on the definition of new access structures, providing hardly any support for an integration into existing database systems.

In [22], an efficient access structure has been introduced to process interval intersection queries on top of any existing relational database system. Instead of accessing raw disc blocks directly, the *Relational Interval Tree* (RI-tree) manages data objects by common built-in relational indexes following the paradigm of relational indexing [20] [9] [5].

In order to support combined queries on single- and interval-valued attributes, we propose several new techniques based on the structure of the Relational Interval Tree. Inheriting the easy implementation of the RI-tree and the superior performance of built-in database indexes, the new techniques have major advantages over former approaches. As experimental results on an Oracle10g server show, the new algorithms outperform existing relational methods for combined queries on interval-and-value tuples significantly.

The remainder of the paper is organized as follows: Section 2 surveys related work on interval intersection techniques that support a combined evaluation of simple selections. After recalling the Relational Interval Tree in Section 3, we present our new approaches in Section 4. Section 5 describes the results of our experimental evaluations, and the paper is concluded in Section 6.

2. Related Work

A straight-forward solution for supporting combined selections on different attributes is to create separate indexes on each of the queried attributes. To evaluate the query, the optimizer commonly uses the indexes to determine the results of the single selections separately and calculates the final query result by intersecting these *inverted lists*. In cases where, e. g., index accesses are very cheap and/or combined selections are rarely used, this solution is efficient, in general. But if one or both index accesses are also rather expensive, as is usually the case when evaluating specific predicates on complex data types, this approach may be suboptimal. Using a pipelined technique, i. e. first evaluating one (usually the more selective) predicate and then checking the filtered result on the second predicate, may help in some scenarios, e. g. if one predicate is very selective (i. e. a good filter) and the used access structures admit this pipelined fashion of evaluation.

In general, to support combined selections efficiently, one has to find an access structure that allows also a combined evaluation of several predicates. In our case, where comparison predicates on simple types and intersection predicates on intervals have to be calculated, an adequate access structure should integrate the different evaluation approaches of the separate indexes and use the (presumably cheap) selection of the simple type to prevent expensive intersection calculations. Various different access structures to support intersection queries on intervals exist in the literature [28], and some of them also allow a combined evaluation of simple selections.

The *Time Index* of Elmasri, Wuu, and Kim [8] is an access structure for storing intervals that can be combined with a conventional attribute indexing scheme. A set of linearly ordered indexing points is maintained by a B+-tree, and for each point, a bucket of pointers refers to the associated set of intervals. To support additional attributes, a two-level indexing scheme is proposed: The top-level index is a B+-tree for a (simple) search attribute, and each leaf node entry of the top-level index tree includes a value of the search attribute and a pointer to a time index, i. e. there is a time index tree for each attribute value! Answering a combined query involves traversing the first B+-tree to identify the leaf entry corresponding to the searched attribute value, followed by an interval search on the time index found there. In general, this approach is reasonably applicable on attributes with very narrow domains only.

Gunadhi and Segev [14] propose a related approach, the *ST-index* (*surrogate and time index*), that facilitates a two-level method whose top level indexes the key attribute of the interval objects (using a B+-tree), while the second level indexes the intervals that share the same key attribute. For the second level, a so-called *AP-tree* (*append-only tree*) is used, an extension of the B+-tree for indexing append-only periods. The AP-tree indexes the start times of all intervals sharing a distinct key attribute value. The problem with this approach is that pure interval queries will have to check all stored intervals to see whether they intersect the query interval. Besides that, this augmentation of the B-tree structure is not supported by commercial relational databases.

Blankenagel and Güting present the *external segment tree* (*ES-tree*) [3] which is a paginated version of the main-memory segment tree [2]. By embedding B-trees, the ES-tree can also be modified to address queries with additional predicates. The original ES-tree structure guides the search to a subset of intervals that intersect the query interval while an embedded B-tree allows searching this subset for whether the simple predicate is also satisfied. Unfortunately, this augmentation also isn't supported by present-day relational databases.

The *Interval-Spatial Transformation* (*IST*) of Goh et al. [13] is based on encoding intervals by space-filling curves called *D*-, *V*- and *H-ordering* that map the boundary points into a linear space. The structure reveals a strong correspondence to relational composite indexes. Aside from quantization aspects, the D-ordering is equivalent to a composite index on the interval bounds (*end*, *start*), the V-ordering corresponds to an index on (*start*, *end*), and the H-ordering simulates an index on (*end–start*, *start*). To support combined selections, additional attributes can be included into the composite indexes. As mentioned above, this approach may result in poor query performance if the selectivity relies on the "wrong" bound or attribute.

The Window-List technique of Ramaswamy [26] is a static solution for the interval management problem and employs built-in B+-trees. The optimal space and I/O complexity for stabbing queries is achieved. For combined indexing, a composite B+-tree index is proposed, consisting of a window's starting point and the additional attribute (key). Unfortunately, updates do not seem to have non-trivial upper bounds, and adding as well as deleting arbitrary intervals can deteriorate the query efficiency of this structure.

As an alternative, multi-dimensional or spatial access structures may be used for combined indexing. The problem of this approach is that, in general, all dimensions stored in the index are handled in the same way, and the characteristics of the single attributes' types aren't respected. So, depending on the respective query, some of the dimensions/predicates may be supported very efficiently while other cannot profit by the index at all.

If only multidimensional points are supported, as in the k-d-B-tree [27], mapping an (interval, value) pair to a triplet consisting of lower bound, upper bound, and value allows the intervals to be represented by points in three-dimensional space. If intervals are represented more naturally, as line segments in a two-dimensional value-interval space, Guttman's *R-tr*ee [15] or one of its variants including *R+-tree* [29] and R*-tree [1] could be used. Such solutions sometimes provide good average-case performance, but overlapping still remains a severe problem, especially if the interval distribution is highly nonuniform.

The *Segment R-tree* (*SR-tree*) of Kolovson and Stonebraker [19] is a combination of the main memory-based segment tree with the secondary storage-oriented R-tree. The split algorithm cuts long intervals into spanning portions and remnant portions thus producing some redundance but being less sensitive to overlapping intervals. However, the method may suffer if there are many interval deletions, since all remnants (segments) of a deleted interval have to be found and physically deleted. Furthermore, implementing the SR-tree requires an adaption of the R-tree structure.

In [21], Kriegel, Pfeifle, Pötke, and Seidl present another spatial index structure. Similar to the approach presented here, it is based on the Relational Interval Tree and can be easily embedded in modern extensible indexing frameworks. Extended objects are mapped to interval sequences by means of space-filling curves, and the RI-tree is slightly modified to support sequences of intervals thus enabling the search for spatial objects. The (interval, value) pairs are regarded as line segments in a two-dimensional value-interval space and are mapped to corresponding interval sequences. The RI-Tree used to manage these interval sequences is twice as high as the RI-Tree that would have been used for storing the original intervals. Each interval is mapped on a sequence of intervals, thus increasing the required space and time for storing and accessing the data.

3. The Relational Interval Tree

For the paper to be self-contained, we give an overview of the Relational Interval Tree developed in our previous work [22]. The RI-tree is a relational storage structure based on Edelsbrunner's main-memory interval tree [7] and can be built on top of the SQL layer of any RDBMS [20] [5]. It guarantees an optimal complexity for storage space and I/O operations when updating or querying large sets of interval data (*start*, *end*).

The RI-tree strictly follows the paradigm of relational access structures since its implementation is restricted to (procedural and declarative) SQL but does not assume any lower-level interfaces to the database system. In particular, the built-in index structures of a DBMS are used as they are, and no intrusive augmentations or modifications of the database kernel are required. So this approach will directly profit by any enhancements of the underlying system's index implementation.

3.1 Dynamic Data Structure

The structure of the RI-tree consists of a binary tree of height h which makes the range $[1, 2^h-1]$ of potential interval bounds accessible. 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 without causing any I/O operation.

For the relational storage of the intervals, the node values of the tree are used as artificial keys: Each interval is assigned to a *fork node*, which is the first node that hits the interval when descending the tree from the root node down to the interval location.

In the context of temporal databases, the special values *now* [6] and *infinity* can occur as upper bounds of the intervals. Such temporal intervals can be managed by the RI-tree using artificial exclusive node values for the corresponding intervals. Now-ending intervals, for example, are all assigned to a special node $fork_{now}$ when being inserted.

Figure 4 illustrates an example of the RI-tree for the five sample intervals of the contracts table in Figure 3.

No.	Budget (m€)	Period	
		Start	End
C1	2	1	5
C2	5	2	9
C3	10	8	17
C4	6	14	19
C5	8	21	26

Figure 3: Sample contracts table

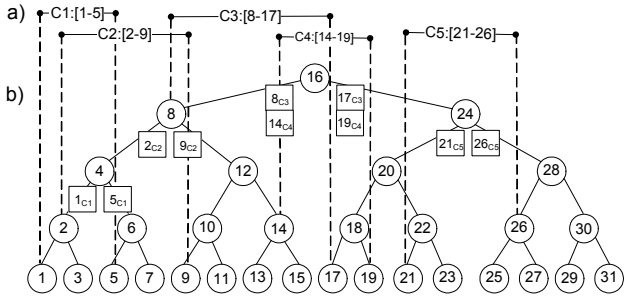

Figure 4: Example of an RI-tree. a) Five sample intervals. b) Virtual backbone and registration position. c) Relational indexes *lowerIndex* and *upperIndex*

The intervals (1,5) for contract 1, (2,9) for contract 2, (8,17) for contract 3, (14,19) for contract 4 and (21,26) for contract 5 are pictured in Figure 4a. The virtual backbone with root value 16 covers the data space from 1 to 31 (Figure 4b). The five intervals are registered at the nodes 4, 8, 16, and 24, respectively. The interval (1,5) for contract 1 is represented by the entries (4, 1, C1) in the *lowerIndex* and (4, 5, C1) in the *upperIndex* since 4 is the registration node, and 1 and 5 are the start and end points of the interval, respectively (Figure 4c).

```
SELECT id FROM upperIndex AS i
    JOIN :leftQueries USING (node)
    WHERE i.end >= :start
UNION ALL
SELECT id FROM lowerIndex AS i
    JOIN :rightQueries USING (node)
    WHERE i.start <= :end
UNION ALL
SELECT id FROM lowerIndex  // or upperIndex
    WHERE node BETWEEN :start AND :end
```

Figure 5: SQL statement for an intersection query with bind variables for *leftQueries*, *rightQueries*, *start* and *end*

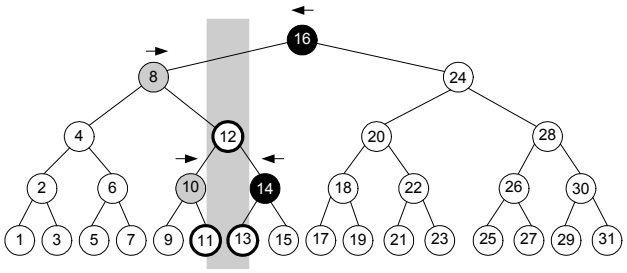

Figure 6: Query preparation step for the query interval [11,13] (shaded in light gray): *leftQueries* {8,10}; *rightQueries* {14, 16}; *innerQueries* {11-13}

3.2 Intersection Query Processing

For processing an interval intersection query (*start*, *end*) based on the RI-Tree, we distinguish two phases, the procedural query preparation phase and the declarative query processing phase. The first phase descends the virtual backbone from the root node down to *start* and to *end*, respectively (Figure 6). The traversal is performed arithmetically without causing any I/O operations, and the visited nodes are collected in two different main-memory tables *leftQueries* and *rightQueries* both obeying the unary relational schema (*node*). Nodes to the left of *start* may contain intervals which overlap *start* and are inserted into *leftQueries*. Analogously, nodes to the right of *end* may contain intervals which overlap *end* and are inserted into *rightQueries*. Whereas these nodes are taken from the paths, the set of all nodes between *start* and *end* belongs to the so-called *innerQuery* which is represented by a single range query on the node values. All intervals registered at nodes from the *innerQuery* are guaranteed to intersect the query and, therefore, will be reported without any further comparison. The query preparation phase is entirely performed in main memory and requires no I/O operations.

In the second phase, the transient tables are joined with the relational indexes *upperIndex* and *lowerIndex* by a single, three-fold SQL statement (Figure 5). The end point of each interval registered at nodes in *leftQueries* is compared to *start*, and the start point of intervals in *rightQueries* is compared to *end*. The *innerQuery* corresponds to a simple range scan over the intervals with nodes in (*start, end*).

Because intervals are organized by the node value, the significant intervals which intersect the query interval are stored in contiguous ranges on disk. For a tree height of h, there are at most $2 \cdot h$ different ranges which have to be considered when processing the query interval. Since the output from the relational indexes is fully blocked for each join partner, the SQL query requires $O(h \cdot \log_b n + r/b)$ I/Os to report r results from an RI-tree with n stored intervals (block size b). Therefore, the RI-tree has optimal I/O complexity for processing temporal stabbing and interval queries.

4. Managing Interval-and-Value Tuples Using the RI-Tree

In order to engage the RI-tree for managing and querying interval-and-value tuples in the presence of combined queries, we extend the structure in such a way that both the interval data as well as the simple attribute values are regarded in an equivalent way. In detail, our goal is to support also query mixes as variants of the query in Figure 2. In particular, we consider cases where the interval predicate only contains a single point (*stabbing query*) and the value predicate only queries a single value instead of a range, as important variants. Altogether, we regard the following types of queries:

- Value-Stabbing Queries
- Value-Interval Queries
- Range-Stabbing Queries
- Range-Interval Queries

In the next sections, we propose various new techniques by enhancing the RI-tree algorithms in such a way that selections on simple attributes are also supported. Here we restrict our considerations to the most important temporal operator *overlaps* which currently is the only defined predicate on the SQL:2003 *period* type. The approach can be easily extended to other interval relationships according to [23].

4.1 Extending the RI-tree Indexes by Simple Attributes

In order to integrate the evaluation of the additional value selection predicate into the RI-tree utilization, we start by extending the internal RI-tree query of Figure 5 by the according predicate (Figure 7). To support the new predicate also by the internal B-tree indexes *lowerIndex* and *upperIndex*, we investigate different ways to include the simple attribute in these composite indexes.

```
SELECT id                          //subquery (1)
FROM upperIndex AS i
   JOIN :leftQueries USING (node)
WHERE i.end >= :start
   AND i.value BETWEEN :Value1 AND :Value2
UNION ALL
SELECT id                          //subquery (2)
FROM lowerIndex AS i
   JOIN :rightQueries USING (node)
WHERE i.start <= :end
   AND i.value BETWEEN :Value1 AND :Value2
UNION ALL
SELECT id                          //subquery (3)
 FROM lowerIndex
   WHERE node BETWEEN :start AND :end
   AND value BETWEEN :Value1 AND :Value2
```

Figure 7: Adapted RI-tree SQL query supporting a simple attribute

Indexes	Value Node Bound		Node Value Bound		Node Bound Value	
Queries	(1)/(2)	(3)	(1)/(2)	(3)	(1)/(2)	(3)
Value-Stabbing	1 *q* 1	1 1 1	*q* 1 1	1 1 1	*q* 1 *n*	1 1 *b*
Value-Interval	1 *q* 1	1 1 1	*q* 1 1	1 *i* 1	*q* 1 *n*	1 1 *i·b*
Range-Stabbing	1 *r·q* 1	1 *r* 1	*q* 1 *r*	1 1 1	*q* 1 *n*	1 1 *b*
Range-Interval	1 *r·q* 1	1 *r* 1	*q* 1 r	1 *i* 1	*q* 1 *n*	1 1 *i·b*

Figure 8: Number of contiguous index ranges for different query types and index orders.

As mentioned in Section 3, the original index schema is (*node*, *start*, *id*) or (*node*, *end*, *id*), respectively. (In the following, we will subsume the interval bounds *start* and *end* by *bound*.) Now the question is where to insert the additional attribute to efficiently evaluate the query of Figure 7. To answer this question, we regard the different orders to integrate the *value* attribute into the composite index and the access cost caused by these combinations. We measure this access cost in contiguous ranges of index entries for the different query types of intent. Figure 8 lists an overview on the number of contiguous index areas that have to be accessed when evaluating a certain query type with a certain index order. In this table, we use the following variables:

- *q*: number of nodes in *leftQueries* or *rightQueries*
- *i:* length of the query interval, i. e. :*end* – :*start*
- *n:* number of nodes in the index that have to be checked on the query bound (i. e. those nodes in *upperIndex* whose *end* bound is greater or equal to the query interval's *start* bound, and vice versa for the *lowerIndex*)
- *b:* number of bounds in *lowerIndex* or *upperIndex*
- *r:* length of the query range, i. e. :*Value2* – :*Value1*

The table in Figure 8 has to be read in the following way: For a certain query type (e. g. a Value-Stabbing Query) and a certain index order (e. g. (*Value*, *Node*, *Bound*)) there are two columns describing the access cost for a subquery of type (1)/(2) or (3), respectively. According to the given index order, the product of the three numbers or variables indicate the number of contiguous index areas that have to be accessed when evaluating a subquery. From Figure 8, we can read the following results:

- A Value Query (i. e. a Value-Stabbing or a Value-Interval Query) certainly profits from a (*value*, *node*, *bound*, *id*) index: In this case, the simple attribute predicates of the single subqueries degenerate to point queries and the index can be used to first determine a

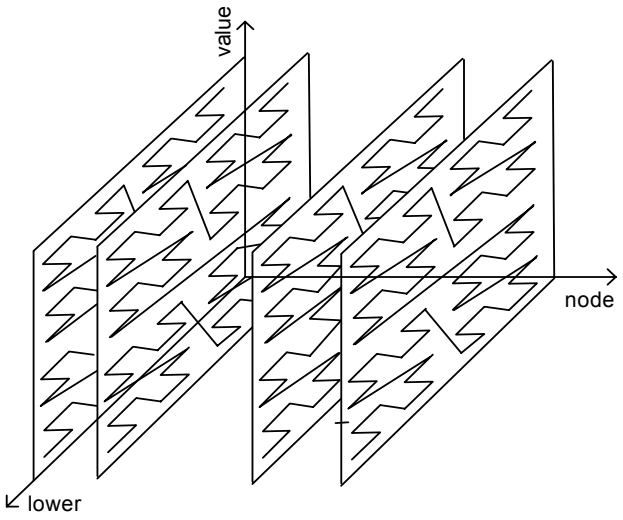

Figure 9: Sort order for a (*node*, {*value*, *bound*}) index

single contiguous tuple range sorted by (*node*, *bound*) that can be evaluated as in the original proceeding. Like there, we have to access q index ranges for evaluating subquery (1) or (2), and one for subquery (3).

- A (*node*, *value*, *bound*, *id*) index will not cause more disk accesses for subqueries (1) and (2) but the accessed disk areas are more dispersed hence reducing the positive effect of potential disk caches. Furthermore, for the Value-Interval Query, subquery (3) will cause additional overhead (factor i) as the interval predicate has to be evaluated before the value predicate (for each point in the interval, the right value has to be found).
- A (*node*, *bound*, *value*, *id*) index would be the worst choice: the simple selections are performed after processing the basic RI-tree predicates, so each result tuple of the interval evaluation has to be checked on the simple value.
- For Range Queries, the indexes have to access $q \cdot r$ or $q \cdot n$ areas to evaluate subqueries (1) and (2). For subquery (3), the (*node*, *value*, *bound*, *id*) index behaves best for Range-Stabbing queries, having to access only a single index area. Subquery (3) of Range-Interval Queries can be evaluated by r or i accesses.

As a first conclusion, Value Queries can be evaluated very efficiently by a (*value*, *node*, *bound*, *id*) index, requiring the same number of disk accesses as in the original proceeding without an additional *value* predicate. Now the question is if Range Queries can be further enhanced. We will address this issue in the next section.

4.2 Improving Range Query Processing

When using composite indexes for multiple attributes, a problem arises if a query contains range predicates on two or more of the indexed attributes. As the index is sorted first by one attribute and then by the other, all tuples satisfying the first predicate can be found in a contiguous disk area. However, all tuples also satisfying the second predicate are scattered within this area and for each value of the first predicate's range there is an area containing final result tuples.

Indexes	{Node,Value} Bound		Node {Value, Bound}	
Queries	(1)/(2)	(3)	(1)/(2)	(3)
Value-Stabbing	q 1	1 1	q n	1 b
Value-Interval	q 1	i 1	q n	i b
Range-Stabbing	$r \cdot q$ 1	r 1	q $< r \cdot n$	1 $< r \cdot b$
Range-Interval	$r \cdot q$ 1	$< r \cdot i$ 1	q $< r \cdot n$	i $< r \cdot i$

Figure 10: Number of contiguous index ranges for different query types and indexes based on space-filling curves.

To handle this sort of problem, space-filling curves as Z-order or Hilbert curves, for instance, have been successfully engaged for multi-dimensional indexing in recent years [11][17][10][4][25][24]. By mapping multi-dimensional data to one-dimensional values, a one-dimensional indexing method can be applied. If space-filling curves are used, the mapping is distance-preserving, i. e. similar values of the original data are mapped on similar index data, and that for all dimensions. Thus, if the index data are clustered on disk, ranges of an indexed attribute will also be found in contiguous disk areas.

According to Figure 7, there are different cases where a subquery contains two range predicates and thus could eventually profit by a space-filling curve. When evaluating a Range Query, Subqueries (1) and (2) contain range predicates for the *bound* and *value* attributes. In the special case of Range-Interval Queries, subquery (3) contains range predicates on the *node* and *value* attributes. Thus, in order to reduce the access cost, we consider space-filling curves on the attributes *bound* and *value* and on the attributes *node* and *value*, respectively (without jeopardizing the natural order on *node* and *bound*). In doing so, we receive two new composite indexes:

- (*node*, {*value*, *bound*})
- ({*node*, *value*}, *bound*)

where the attributes in curly braces denote the combined value that is received by applying the space-filling curve function on these attributes. The sort order of the single attributes of the (*node*, {*value*, *bound*}) index is depicted in Figure 9.

Figure 10 shows the expected number of contiguous index ranges that have to be accessed when evaluating a

Queries	Subqueries (1)/(2)	Subquery (3)
Value-Stabbing	(value, node, bound)	(value, node, bound)
Value-Interval	(value, node ,bound)	(value, node, bound)
Range-Stabbing	(node, {value, bound})	(node, value, bound)
Range Interval	(node, {value, bound})	({node, value})

Figure 11: Best expected indexes for each subquery

certain query type with the new indexes. It provides the following results:

- Range Queries may profit by a (*node*, {*value*, *bound*}) index that can be used to save disk accesses for subqueries (1) and (2). On a *value*/*bound*-plane, the combined values calculated by the space-filling curve are clustered in an $r \cdot n$ rectangle. Depending on the clustering behavior of the space-filling curve, the number of contiguous disk areas required to store the queried data is much below the upper limit of $r \cdot n$. However, Subquery (3) will not profit by the space-filling index as there is no predicate on the *bound* attribute.
- Subquery (3) of Range-Interval Queries may profit by a ({*node*, *value*}, *bound*) index. On a *node*/*value*-plane, the combined values calculated by the space-filling curve are clustered in an $r \cdot i$ rectangle. Depending on the clustering behavior of the space-filling curve, the number of contiguous disk areas required to store the queried data is much below the upper limit of $r \cdot i$. However, Subqueries (1) and (2) will not profit by the space-filling index as there is no predicate on the *node* attribute.
- Value Queries do not profit by the new indexes as there appear no subqueries containing two range predicates.

The results show that space-filling curves may reduce disk accesses in certain cases but there does not exist a "universal" index that supports all queries to the same extent. Indeed, even different subqueries would require different indexes. We will focus to this aspect in the next section.

4.3 Employing Index Mixes

As pointed out in the previous sections, the desire to find a single index that is suited for all query types is hard to fulfil. If we abandon this attempt, we can try to find the best index for each query type. In most applications, the occurring query types are predictable and the best index type can be chosen accordingly. If the application requires different query types, we may create even several indexes to support each query in the best possible way. Using a common index interface of an object-relational database system to implement the proposed extensions, we can give hints to the optimizer on how to choose the appropriate index.

Figure 8 and Figure 10 indicate which index types may perform best for a certain query type. Value Queries are certainly best supported by a (*value*, *node*, *bound*, *id*) index, requiring the same access cost as in the original approach without an additional value predicate. For Range Queries, the best index cannot be definitely determined as the number of block accesses required to answer a query depends on the data and the behavior of the space-filling curve (if used). In particular, the different subqueries of Range Queries are best supported by different indexes. As these subqueries are independent of one another, the optimizer is also able to use different indexes.

Based on that observation, we compile a list of best expected indexes for each subquery (Figure 11). For Range Queries, the results are based on the assumption that an application of space-filling curves provides positive effects regarding the disk accesses. This assumption gets confirmed by the experiments in the next section.

Creating several different indexes for the same attributes may enhance response times for the different query types but also results in higher costs for index updates and storage requirements. As standard B+-trees are used for all indexes, estimating these costs is quite straightforward: Insertions and deletions for each tree are performed by $O(\log_b n)$ I/Os on a database with block size b and n entries. Storage costs depend on the number of key components for each index. We will further investigate various index mixes for different types of queries in Section 5.

4.4 Adapting the RI-tree Algorithms

In this section, we explain how to adapt the original RI-tree algorithms to support the indexing of an additional simple attribute.

When using an index without a space-filling curve, only minor changes of the original procedures are required. The composite index is extended by the additional attribute and the query is adapted according to Figure 7. The preparation step for a query, i.e. collecting *left*- and *rightQueries*, remains unchanged.

For indexes based on space-filling curves, the according attributes of the original tables are mapped on a single value by the space-filling function first. Then the index is created on this calculated value and the remaining attribute. For evaluating the query, the transient tables *leftQueries* and *rightQueries* have to be determined as before and then have to be recalculated according to the space-filling curve. The query has also to be transformed accordingly. In the following, we will explain these steps in more detail for the general case of a Range-Interval Query using the indexes according to Figure 11.

```
SELECT id
FROM index1
    JOIN :leftQueries USING (node)
    JOIN :range1 ON (sfc BETWEEN from AND to)
UNION ALL
SELECT id
FROM index2
    JOIN :rightQueries USING (node)
    JOIN :range2 ON (sfc BETWEEN from AND to)
UNION ALL
SELECT id
FROM index3
    JOIN :range3 ON (sfc BETWEEN from AND to)
```

Figure 12: Modified SQL query using space-filling curves.

Using a space-filling function *sfc*: $T^2 \rightarrow T$ that maps two values of type T to a single value of type T, we first have to create the following composite indexes (cf. Figure 11):

- index1: (*node*, *sfc*(*value*, *end*), *id*)
- index2: (*node*, *sfc*(*value*, *start*), *id*)
- index3: (*sfc*(*node*, *value*), *id*)

The space-filling values can be calculated in advance by adding a calculated column *sfc* to the original tables. In this case, the space-filling values are calculated on the insert of new tuples (after performing the necessary type mappings).

Furthermore, we assume the existence of a function *sfcRect*: $T^4 \rightarrow 2\text{^}T^2$ that maps a rectangle defined by its coordinates (x_1, x_2, y_1, y_2) on the space-filling plane to a set of pairs (*from*, *to*) that represent the bounds of contiguous sequences of the space-filling curve within the specified rectangle. Using this function, we can calculate the following transient tables with schema (*from*, *to*) for a given query with a specified range (*val1*, *val2*) and interval bounds (*start*, *end*):

- range1 := *sfcRect*(*val1*, *val2*, *start,* (2^*h*)-1)
- range2 := *sfcRect*(*val1*, *val2*, 1, *end*)
- range3 := *sfcRect*(*start*, *end*, *val1*, *val2*)

No.	Budget (m€)	Period		Node	*Sfc*(budget, start)
		Start	End		
C1	2	1	5	4	4
C2	5	2	9	8	50
C3	10	8	17	16	221
C4	6	14	19	16	149
C5	8	21	26	24	186

Figure 13: Sample contracts table with additional columns for *node* and *sfc*-value

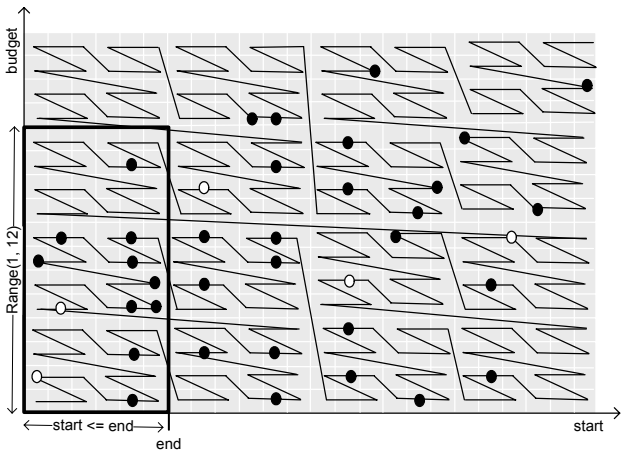

Figure 14: Query rectangle for the sample contracts table

Having also calculated the transient tables *leftQueries* and *rightQueries* in the usual way, we can compile the final query as presented in Figure 12.

As an example, we present the evaluation of a Range-Interval Query by means of *index2*. In Figure 13, the sample contracts table of Figure 3 is extended by two additional columns, one for the *node* and one for the *sfc* value calculated from the columns *budget* and *start* via z-order. *Index2* is created by building a composite B-tree on (*node*, *sfc*, *id*).

Given a Range-Interval Query with value range (1, 12) and a query interval (3, 6), *range2* can be determined by applying the function *sfcRect* on the according rectangle (1, 12, 1, 6) that is depicted in Figure 14. There, a space-filling curve on {*budget*, *start*} clusters the entries of the corresponding index. Each point in the figure represents the (*budget*, *start*) pair of a tuple in the database (with the white points showing the values of Figure 13), while the rectangle represents the specified ranges for *budget* and *start* within a query. As one can see, all qualifying tuples are found on three contiguous sequences on the space-filling curve. In contrast, for an ordinary composite index on (*budget*, *start*) or (*start*, *budget*), we would have to read twelve or six sequences (i. e. the rectangle's side lengths), respectively. So, in this case, using a space-filling curve for the index would be favorable.

5. Experimental Evaluation

To analyze the performance of our approach, we implemented the Relational Interval Tree and extended it by our new algorithms for the evaluation of Range-Interval Queries. All experiments were executed on a dual Xeon/3 GHz Server with 4GB main memory and U-SCSI hard drive (standard block size of 8KB). The new algorithms were implemented in Oracle10g using PL/SQL.

For generating data, we used the TimeIT [12] software, a package for testing and evaluating the effectiveness of temporal query evaluation algorithms. The ex-

Table size	*n* tuples
Interval domain	$[1, 2^{25}-1]$
Starting point distribution	uniform in $[1, 2^{25}-1]$
Length distribution	uniform in $[1, 2^{25}-1]$
Attribute domain	$[1, 2^{25}-1]$

Figure 15: Parameters of the table/queries

periments were performed on various relations with different sizes and interval lengths (cf. Figure 15). In all experiments, the bounding points of the intervals and the values lie in the domain of $[1, 2^{25}-1]$. Starting points and lengths of the intervals are uniformly distributed.

We compared the performance of our new algorithms to the following techniques (Figure 16):

Composite indexes. For our experiments we used compound indexes on the attributes *value*, *start* and *end*. We differentiated between two indexation sequences: (*start*, *end*, *value*) and (*value*, *start*, *end*).

Intersection of interval lists. Separate evaluation of Interval and Value Queries by creating an Oracle B-tree on the simple attribute and two RI-tree indexes *lowerIndex*, *upperIndex* on the intervals.

Pipelining. Sequential evaluation of Value-Interval Queries by either preprocessing the Value Queries or the Interval Queries.

R-Tree. We used the Oracle built-in R-tree, so no code had to be implemented here. The data, i.e. the intervals combined with the attribute values are represented as lines in a 2-D space.

Spatial RI-tree approach. We evaluated Value-Interval Queries queries by means of the approach presented in [21].

no	description	abbr.
1	Comp. index: (lower, upper, value)	(LUV)
2	Comp. index: (value, lower, upper)	(VLU)
3	B-tree and RI-tree: intersection	B∩RI
4	B-tree and RI-tree: pipelining	B→RI
5	RI-tree and B-tree: pipelining	RI→B
6	R-tree: 2D lines	R-tree
7	Spatial RI-tree: 2D lines	SpatRI
8	RI-tree: (value, node, bound, id)	RI(VNB)
9	RI-tree: (node, value, bound, id)	RI(NVB)
10	RI-tree: (node, bound, value, id)	RI(NBV)
11	RI-tree: ({node, value}, bound, id) H	RI({NV}B)h
12	RI-tree: ({node, value}, bound, id) Z	RI({NV}B)z
13	RI-tree: (node, {value, bound}, id) H	RI(N{VB})h
14	RI-tree: (node, {value, bound}, id) Z	RI(N{VB})z

Figure 16: Different types of indexes used for experiments (H: Hilbert curve, Z: Z-order curve)

To investigate the behavior of our newly proposed index techniques, we firstly performed experiments measuring the access cost of the four different query types. Figure 17 to Figure 20 show the average results of 100 queries for varying table sizes.

As suspected in Section 4.1, for Value-Stabbing Queries (Figure 17) the indexes (*value*, *node*, *bound*) and (*node*, *value*, *bound*) perform best, providing an optimal sort order for this kind of query. Value-Interval Queries (Figure 18) also profit most by a (*value*, *node*, *bound*) index, as expected.

When considering Range Queries, other indexes come out on top. While the (*node*, *value*, *bound*) index performs well for Range-Stabbing Queries (Figure 19), the space-filling version (*node*, {*value*, *bound*}) delivers slightly better results, though not significantly. To check our assumption that a Range-Stabbing Query may profit from using both of these indexes (for different subqueries), we

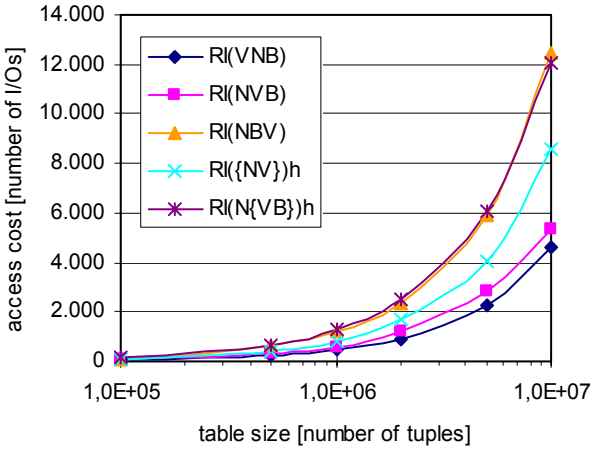

Figure 17: Value-Stabbing Queries: Access cost with varying table sizes

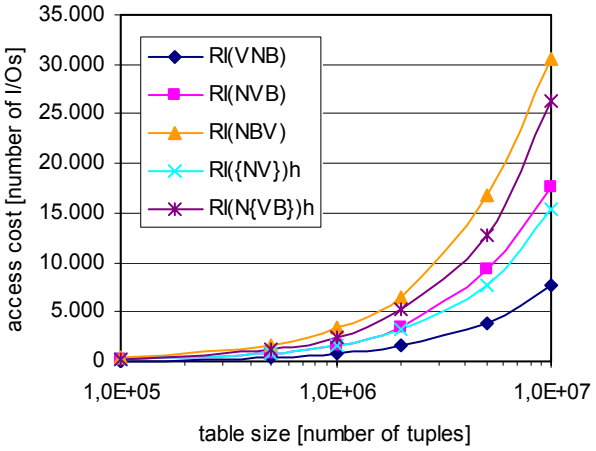

Figure 18: Value-Interval Queries: Access cost with varying table sizes

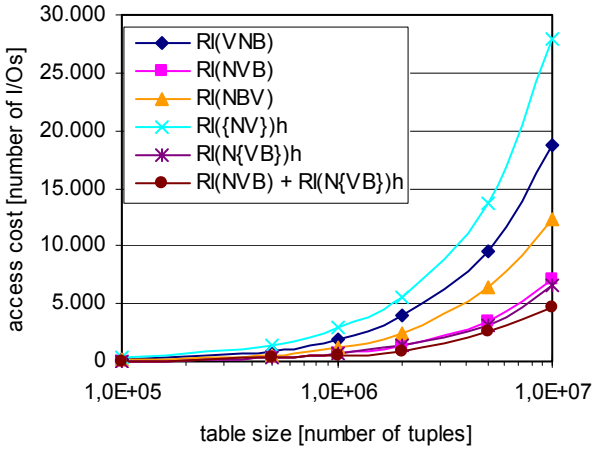

Figure 19: Range-Stabbing Queries: Access cost with varying table sizes

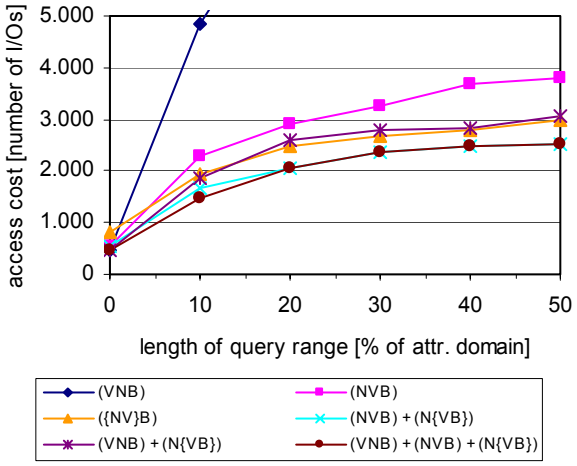

Figure 21: Stabbing Queries: Access cost for varying length of ranges

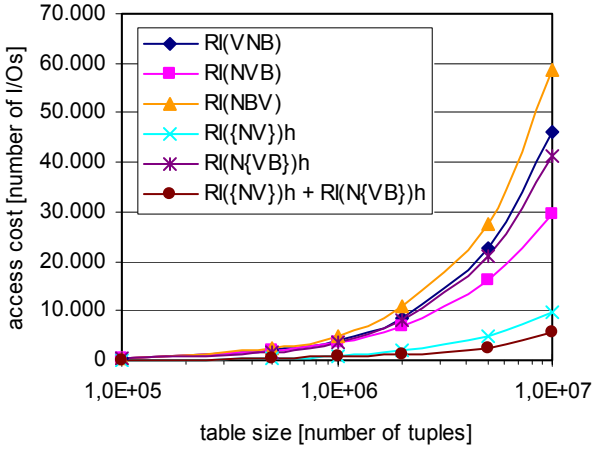

Figure 20: Range-Interval Queries: Access cost with varying table sizes

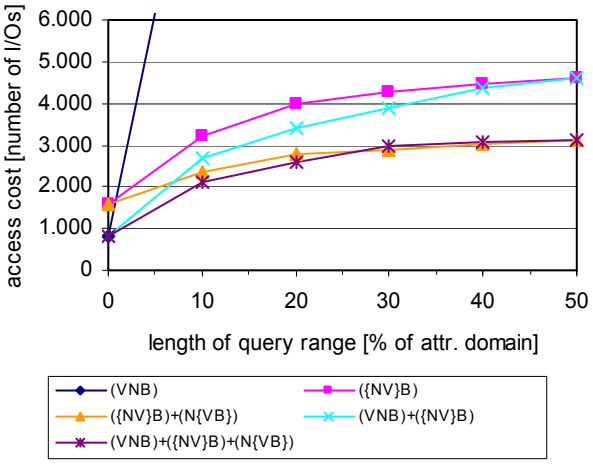

Figure 22: Interval Queries: Access cost for varying length of ranges

also performed measurements for this case. As Figure 19 shows, we indeed receive clearly better results with a two-index solution. Subqueries (1) and (2) are able to use the space-filling index, while subquery (3) adopts the normal (*node*, *value*, *bound*) index that is best suited for this case.

For the general case of Range-Interval Queries (Figure 20), the space-filling indexes show the best performance. The index ({*node*, *value*}, *bound*) achieves significantly better results than all other single-index solutions. When combining this index with a (*node*, {*value*, *bound*}) index, the number of I/Os can be further decreased. This corresponds to the considerations of Section 4.3.

We also checked the case when one of the query parameters (range or interval) is undefined. As one may expect, best results are achieved in this case for indexes where the unspecified parameter has a low "sort priority".

Moreover, we performed extensive experiments to investigate how access cost is influenced by varying the query parameters. In the following, all measurements refer to a table size of 10^6.

In the first two experiments, we examined Stabbing and Interval Queries by varying the length of the query range from 0 up to 50 percent of the attribute domain. Figure 21 shows the results for Stabbing Queries. While we performed experiments for each conceivable index combination of the newly proposed indexes, only the most interesting data, i. e. the most efficient index combinations are presented. The question is, how many indexes are required to expect low access cost. In Figure 21, a combination of the three indexes (*value*, *node*, *bound*), (*node*, *value*, *bound*) and (*node*, {*value*, *bound*}) performs best, requiring the least number of I/Os. But Figure 21

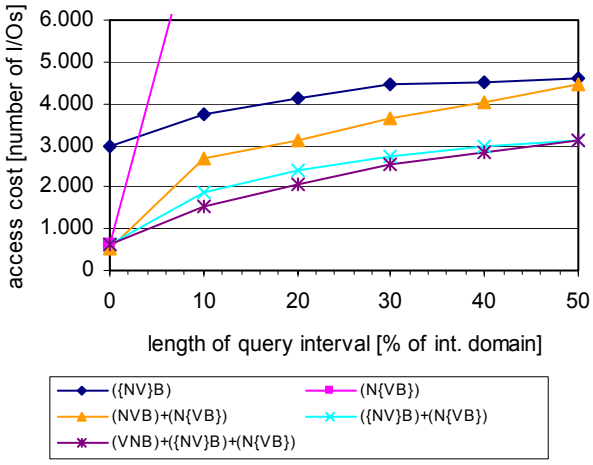

Figure 23: Range Queries: Access cost for varying length of intervals

also reveals that the access cost doesn't increase significantly if one of the first two indexes is omitted. Even a single index ({*node*, *value*}, *bound*) achieves good average behavior and may suffice if storage and update performance are critical. In contrast, the (*value*, *node*, *bound*) index is not competitive for this case.

Figure 22 shows similar results for Interval queries. Again, the threefold index combination (*value*, *node*, *bound*), ({*node*, *value*}, *bound*) and (*node*, {*value*, *bound*}) delivers best performance, but omitting the index (*value*, *node*, *bound*) doesn't harm really. The best one-index solution for this case is provided by the ({*node*, *value*}, *bound*) index, again.

In the next experiment, we investigated how Range Queries behave for varying lengths of the query interval. Figure 23 depicts the results for interval lengths varying from 0 up to 50 percent of the interval domain. As in the previous cases, many indexes help to reduce access cost, but also smaller index sets deliver acceptable performance. As an exception, Value Queries are best supported by (*value*, *node*, *bound*) queries for any length of the query interval.

For an overall comparison, we measured the access cost for a uniform distribution of the four query types. From Figure 24, we can read the following results:

- The newly proposed techniques perform clearly better than the competing approaches.
- The indexes enhanced by space-filling curves achieve significant improvements in performance.
- Using index combinations also improves performance, especially when applied to a corresponding query mix.

6. Conclusions

Following the principle of relational indexing, the Relational Interval Tree (RI-tree) is an approved access method for interval data that is entirely built on top of the SQL interface of a relational database system. Based on the RI-tree, we propose various indexing schemes to support queries that contain selections on simple (e. g. numbers, strings) as well as interval-valued attributes. By integrating the simple attribute into the internal B-trees of the RI-tree and sorting the multi-dimensional index entries according to space-filling curves, the simple attributes can be managed and queried in combination with the interval data. Our experimental comparisons with competing methods demonstrate the efficiency of this technique depending on the characteristics of the data and queries. When using several of the proposed indexes in combination, additional performance improvements are achieved. In our future work, we plan to extend the proposed techniques to more complex queries and to develop cost models to predict the benefits of indexes for an evolving query workload.

References

[1] Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: *The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles.* SIGMOD Conf. 1990: 322-331.

[2] Bentley J. L.: *Algorithms for Klee's Rectangle Problems.* Computer Science Department, Carnegie-Mellon University, Pittsburgh (1977).

[3] Blankenagel G., Güting R. H.: *External Segment Trees.* Algorithmica 12(6): 498-532 (1994).

[4] Böhm C., Klump G., Kriegel H.-P.: *XZ-Ordering: A Space-Filling Curve for Objects with Spatial Extension.* SSD 1999: 75-90.

[5] Brochhaus C., Enderle J., Schlosser A., Seidl T., Stolze K.: *Integrating the Relational Interval Tree into IBM's DB2 Universal Database Server.* BTW 2005: 67-86.

[6] Clifford J., Dyreson C. E., Isakowitz T., Jensen C. S., Snodgrass R. T.: *On the Semantics of "Now" in Databases.* ACM Trans. Database Syst. 22(2): 171-214 (1997).

[7] Edelsbrunner H.: *A New Approach to Rectangle Intersections*, parts I and II. Internat. J. Comput. Math., 13:209-229, 1983.

[8] Elmasri R., Wuu G. T. J., Kim Y.-J.: *The Time Index: An Access Structure for Temporal Data.* VLDB 1990: 1-12.

[9] Enderle J., Hampel M., Seidl T.: *Joining Interval Data in Relational Databases.* SIGMOD Conf. 2004: 683-694.

[10] Faloutsos C., Rong Y.: *DOT: A Spatial Access Method Using Fractals.* ICDE 1991: 152-159.

[11] Faloutsos C., Roseman S.: *Fractals for Secondary Key Retrieval.* PODS 1989: 247-252.

[12] Gao D., Kline N., Soo M. D., Dunn J.: *TIME-IT: The Time Integrated Testbed, version 1.1.* Available on ftp.cs.arizona.edu, August 2002.

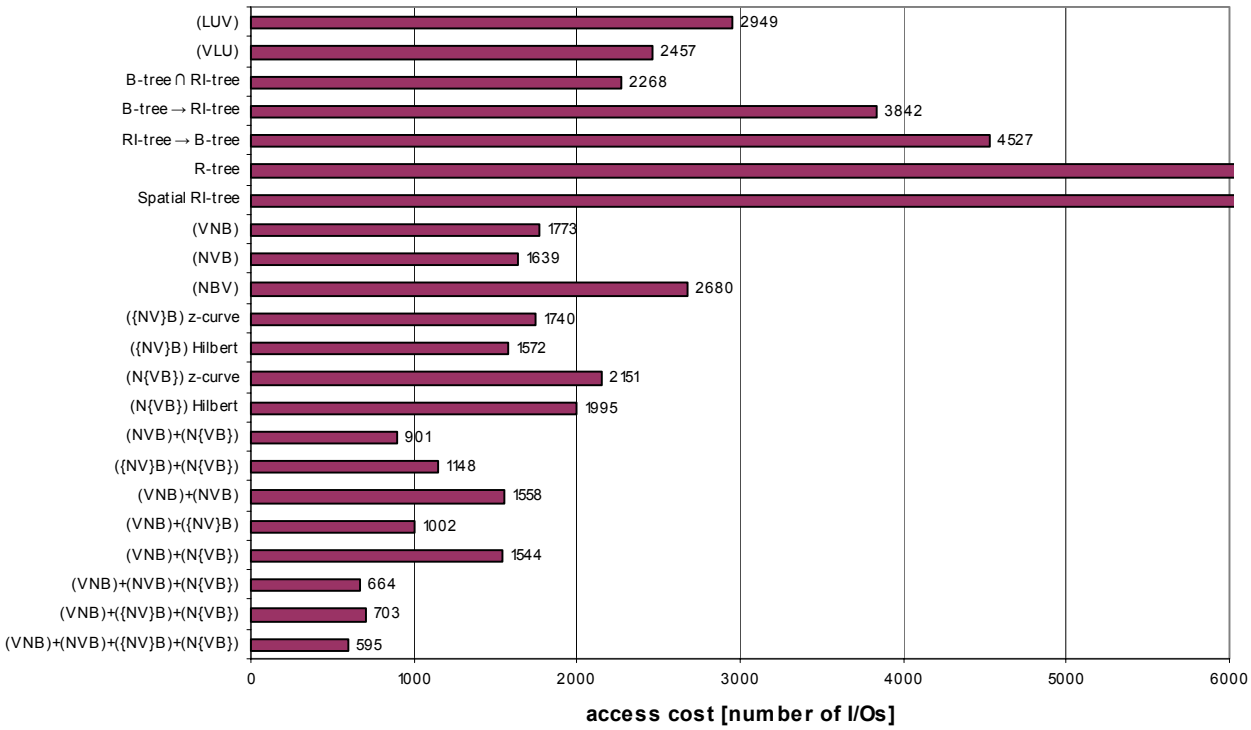

Figure 24: Comparison with competing techniques: access cost

[13] Goh C. H., Lu H., Ooi B. C., Tan K.-L.: *Indexing Temporal Data Using Existing B+-Trees.* Data Knowl. Eng. 18(2): 147-165 (1996).

[14] Gunadhi H., Segev A.: *Efficient Indexing Methods for Temporal Relations.* IEEE Trans. Knowl. Data Eng. 5(3): 496-509 (1993).

[15] Guttman A.: *R-Trees: A Dynamic Index Structure for Spatial Searching.* SIGMOD Conf. 1984: 47-57.

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

[17] Jagadish H. V.: *Linear Clustering of Objects with Multiple Atributes.* SIGMOD Conf. 1990: 332-342.

[18] Kanellakis P. C., Ramaswamy S., Vengroff D. E., Vitter J. S.: *Indexing for Data Models with Constraints and Classes.* PODS 1993: 233-243.

[19] Kolovson C. P., Stonebraker M.: *Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data.* SIGMOD Conf. 1991: 138-147.

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

[21] Kriegel H.-P., Pötke M., Seidl T.: *Interval Sequences: An Object-Relational Approach to Manage Spatial Data.* SSTD 2001: 481-501.

[22] Kriegel H.-P., Pötke M., Seidl T.: *Managing Intervals Efficiently in Object-Relational Databases.* VLDB 2000: 407-418.

[23] Kriegel H.-P., Pötke M., Seidl T.: *Object-Relational Indexing for General Interval Relationships.* SSTD 2001: 522-542.

[24] Lawder J. K., King P. J. H.: *Querying Multi-dimensional Data Indexed Using the Hilbert Space-filling Curve.* SIGMOD Record 30(1): 19-24 (2001).

[25] Lawder J. K., King P. J. H.: *Using Space-Filling Curves for Multi-dimensional Indexing.* BNCOD 2000: 20-35.

[26] Ramaswamy S.: *Efficient Indexing for Constraint and Temporal Databases.* ICDT 1997: 419-431.

[27] Robinson J. T.: *The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes.* SIGMOD Conf. 1981: 10-18.

[28] Salzberg B., Tsotras V. J.: *Comparison of Access Methods for Time-Evolving Data.* ACM Comput. Surv. 31(2): 158-221 (1999).

[29] Sellis T. K., Roussopoulos N., Faloutsos C.: *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects.* VLDB 1987: 507-518.

[30] Snodgrass R. T., Ahn I.: *A Taxonomy of Time in Databases.* SIGMOD Conf. 1985: 236-246.