Managing Intervals Efficiently in Object-Relational Databases

Hans-Peter Kriegel
University of Munich
Institute for Computer Science
kriegel@informatik.uni-muenchen.de

Marco Pötke
University of Munich
Institute for Computer Science
poetke@informatik.uni-muenchen.de

Thomas Seidl
University of Munich
Institute for Computer Science
seidl@informatik.uni-muenchen.de

Abstract

Modern database applications show a growing demand for efficient and dynamic management of intervals, particularly for temporal and spatial data or for constraint handling. Common approaches require the augmentation of index structures which, however, is not supported by existing relational database systems. By design, the new Relational Interval Tree[1] (*RI-tree*) employs built-in indexes on an as-they-are basis and is easy to implement. Whereas the functionality and efficiency of the RI-tree is supported by any off-the-shelf relational DBMS, it is perfectly encapsulated by the object-relational data model.

The RI-tree requires $O(n/b)$ disk blocks of size b to store n intervals, $O(\log_b n)$ I/O operations for insertion or deletion, and $O(h \cdot \log_b n + r/b)$ I/Os for an intersection query producing r results. The height h of the virtual backbone tree corresponds to the current expansion and granularity of the data space but does not depend on n. As demonstrated by our experimental evaluation on an Oracle8i server, competing dynamic interval access methods are outperformed by factors of up to 42 for disk accesses and 4.9 for query response time.

1 Introduction

There is a growing demand for database applications that handle temporal and spatial data. Intervals occur as transaction time and valid time ranges in temporal databases [SOL 94] [Ram 97] [BÖ 98], as line segments on a space-filling curve in spatial applications [FR 89] [BKK 99], as inaccurate measurements with tolerances in engineering databases, for hierarchical type systems in object-oriented databases [KRVV 93] [Ram 97], or for handling interval and finite domain constraints in declarative systems [KS 91] [KRVV 93] [HP 94]. Particularly for industrial or commercial applications, the integration into RDBMS or ORDBMS is essential.

The *Relational Interval Tree*[1] (*RI-tree*) is a new method to efficiently support intersection queries, i.e. reporting all intervals from the database that overlap a given query interval. Rather than being a typical external memory data structure, the RI-tree follows a new paradigm in being a *relational storage structure*. The basic idea is to manage the data objects by common relational indexes rather than to access raw disk blocks directly. While exploiting the availability, robustness and high performance of built-in index structures in existing systems, the advantages for the RI-tree are in detail:

- Built-in indexes are used on an *as-they-are* basis without any augmentation of the internal data structure. Thus, no interface below the SQL level is required, and any arbitrary off-the-shelf RDBMS immediately supports the technique.
- A proper integration with existing RDBMS is an essential aspect for most industrial or commercial applications. By using built-in relational index structures, their strong robustness, performance and integration into transaction management (including recovery services and concurrency control) is for free. Thus, a lot of implementation efforts and code maintenance is avoided by a relational storage structure in contrast to typical external memory solutions.
- The efficiency of the RI-tree is due to the logarithmic I/O complexity of the underlying relational system for one-dimensional range queries on point data. Almost all RDBMS qualify for this quite weak requirement since they typically have implemented the popular B+-tree. By virtualizing the backbone structure of the original main-memory method and storing the intervals in relational indexes, a high efficiency for the RI-tree is achieved.
- In addition to its efficient support by any off-the-shelf RDBMS, the RI-Tree perfectly fits to the object-relational facilities of modern DBMS including the Oracle8i Server [Ora 99a], the Informix Universal Server [Inf 98] or the IBM DB2 Universal Database [IBM 99]. These systems support integrating the RI-Tree with the declarative SQL level as well as with the relational query optimizer.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

[1] Patent pending [KPS 00]

Internally, the RI-tree manages intervals by two relational indexes. Storing n intervals occupies $O(n/b)$ disk pages, and inserting or deleting an interval requires $O(\log_b n)$ I/O operations where b denotes the disk block size as in [MTT 00]. For reporting the r intervals that intersect a given query interval, $O(h \cdot \log_b n + r/b)$ I/Os are required. The height h of the virtual backbone reflects the current expansion and granularity of the data space but does not dependend on the number n of intervals. On top of a good analytical complexity, also the empirical performance is superior to competitors.

The paper is organized as follows: Section 2 surveys related work for interval management in databases. In Section 3, we introduce the structure of the new Relational Interval Tree, whereas the algorithms for query processing are presented in Section 4. Section 5 discusses the integration into an ORDBMS. After an experimental evaluation in Section 6, the paper is concluded by Section 7.

2 Related Work

A variety of methods has been published concerning interval management in databases, most of them addressing temporal applications. The following sections intentionally survey interval handling in general. Specialized work e.g. on append-only structures for transaction time intervals is omitted due to lack of space.

2.1 Main Memory Structures

In the context of computational geometry, several data structures that support 1D interval data have been developed [PS 93] [Sam 90a]. Among them the *Segment Tree* of Bentley, the *Priority Search Tree* of McCreight and the *Interval Tree* of Edelsbrunner are the most popular. More recent developments include the *Interval Skip List* and the *IBS-Tree* of Hanson et al. [HJ 96].

As major limitation, the main memory resident data structures do not meet the characteristics of secondary storage. In a disk-oriented context, access is block-oriented and only small portions of a structure may reside in main memory at a given time. The concept of *Segment Indexes* [KS 91] is a way to overcome the problem by combining optimal interval structures with efficient disk-oriented indexing techniques. Our approach follows this paradigm and, moreover, uses existing index structures the way they are rather than to extend them what is typically required for custom secondary storage structures.

2.2 Secondary Storage Structures

A variety of secondary storage structures for intervals has been presented in the literature [TCG+ 93] [MTT 00]. Since they typically are based either on the augmentation of existing indexes or on the definition of new structures, most of them share the limited support for an integration into existing systems. When being committed to a commercial ORDBMS, the structures cannot be integrated as the built-in indexes are not extensible by the user.

The *Time Index* of Elmasri, Wuu and Kim [EWK 90] is an index structure for valid time intervals. 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. Since an interval may be registered with several indexing points, the space requirement is $O(n^2)$ for n stored intervals [HJ 96]. Due to this redundance, the time complexity is $O(n)$ for insertion and deletion and $O(n^2)$ for interval intersection query processing [AT 95].

The *Interval B-tree* (*IB-tree*) of Ang and Tan [AT 95] has been developed to overcome the weaknesses of the time index. It can be regarded as an implementation of Edelsbrunner's interval tree [Ede 80] using an augmented B+-tree rather than a binary tree. The original main memory model is thus transformed to an efficient secondary storage structure while preserving the optimal space and time complexity. As a disadvantage that we avoid in our approach, the complex three-fold structure of the interval tree is retained, and a dedicated structure of its own is used for each level. More seriously, the augmentation is not supported by commercial ORDBMS's.

The *Interval B+-tree* (*IB+-tree*) of Bozkaya and Özsoyoglu [BÖ 98] is a secondary storage model of the interval tree of [CLR 90] that differs from Edelsbrunner's interval tree by the fact that it uses the lower bounds of the intervals as primary keys. As a result, queries referring to the upper bounds of intervals such as *meets* or *after* are not supported well. The I/O complexity for insertions or deletions as well as for finding a single intersecting interval for a query is $O(\log_b n)$. Retrieving all r intersecting intervals, however, may result in a scan of the internal nodes covered by the query range. Thus, the worst case time complexity is $O(n)$ rather than the minimum $O(\log n + r)$ which Edelsbrunner's interval tree guarantees. The concept of time splits is introduced as a successful heuristics to avoid large fruitless scans. Again, the augmentation is an obstacle for the integration into commercial systems.

The *TP-Index* of Shen, Ooi and Lu [SOL 94] is based on a transformation of intervals into a triangular 2D space. Duplicates are avoided and the index is well suited for appending intervals since the data space may grow dynamically at the upper bound. The access method is highly specialized to the suggested mapping, and an integration into existing ORDBMSs is not supported. A similar mapping organized by a grid file is presented in [LT 98].

The *External Memory Interval Tree* of Arge and Vitter [AV 96] is an externalization of Edelsbrunner's interval tree where the fan-out of the backbone tree is increased from 2 to $\sqrt{b}$ for disk blocks of size b. The intervals are stored in slab lists and multislab lists. The structure requires $O(n/b)$ pages for n intervals, supports insertions and deletions in $O(\log_b n)$ I/Os and requires $O(\log_b n + r/b)$ I/Os to answer a stabbing query reporting r results, which is the optimal complexity. Unfortunately, no experiments demonstrate the per-

formance and, again, the integration into existing systems is not supported.

Beside originally one-dimensional interval index structures even multi-dimensional index structures can be employed for the task of managing 1D intervals. In general, however, spatial access methods such as Guttman's *R-tree* [Gut 84] and its variants including R^+*-tree* [SRF 87] and *R*-tree* [BKSS 90] may not behave well for one-dimensional intervals. Particularly the long durations and high overlaps of intervals in many temporal applications induce severe performance problems [EWK 90] [GLOT 96]. Two particular solutions are sketched in the following.

The *Segment R-tree* (*SR-tree*) of Kolovson and Stonebraker [KS 91] 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. The authors recommend to combine the SR-tree with a *Skeleton Index* that performs a pre-partitioning of the data space in order to improve query processing performance. The SR-tree performs similar to the R-tree, and particularly the skeleton version yields an improvement. Just as the IB-tree and IB+-tree are augmentations of the B+-tree, implementing the SR-tree requires an adaption of the R-tree structure provided there exists any R-tree in the target DBMS at all. Another approach that supposes a specialized multi-dimensional index structure is suggested by Fenk et al. [FMB 00].

2.3 Relational Storage Structures

Very few methods immediately meet our core requirement to 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 *Window-List* technique of Ramaswamy [Ram 97] 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 non-trivial upper bounds, and adding as well as deleting arbitrary intervals can deteriorate the query efficiency of this structure to $O(n/b)$. Despite the practicability of the approach, no experimental results are demonstrated.

The *Tile Index* approach provided by the Oracle8i Spatial Product [RS 99] is a relational implementation of the multi-dimensional *Linear Quadtree* [Sam 90b]. Spatial objects are decomposed and indexed at a user-defined fixed quadtree level. Each resulting fixed-sized tile contains a set of variable-sized tiles as a fine-grained representation of the covered geometry. Intersection queries are performed by an equijoin on the indexed fixed-sized tiles, followed by a sequential scan on the corresponding variable-sized tiles. When applied to one-dimensional data, the *Tile Index* technique maps an interval to a set of fixed-sized segments to be stored in a built-in B+-tree. Finding a good fixed level for the expected data distribution is crucial, as with the fixed level set too high, too much redundancy emerges due to small fixed-sized tiles, whereas a low fixed level causes too much overhead for scanning the large variable-sized tiles. Therefore, an inappropriate setting causes the response time to degenerate vastly [Ora 97] [Ora 99b]. Unfortunately, the fixed level can only be set at index creation time, and adapting it to changing data and query distributions requires bulk-loading the whole dataset anew. This major drawback is not shared by our *RI-Tree*.

The *Interval-Spatial Transformation* (*IST*) of Goh et al. [GLOT 96] 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 $O(n/b)$. Whereas the expansion of the data space at the upper bound is an explicit feature of the method, the expansion at the lower bound which is supported in our solution remains unclear. Unfortunately, no experimental performance results are reported in the paper. 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. A dynamic refinement of the resolution is not supported by the *IST*. A closer look at 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 (*upper*, *lower*), and the V-ordering corresponds to an index on (*lower*, *upper*). For intersection queries, however, 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 H-ordering simulates an index on (*upper – lower*, *lower*), thus particularly supporting queries referring to the interval length. The *MAP21* approach of Nascimento and Dunham [ND 99] 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.

2.4 Custom Access Methods in ORDBMS

Modern commercial ORDBMS such as the Informix Universal Server [Inf 98], the Oracle8i Server [Ora 99a] or the IBM DB2 Universal Database [IBM 99] support the logical embedding of custom indextypes into the database system. Though the developer may use an extensibility framework to seamlessly bind a new access method to the query language, optimizer and query processor, there is no application program interface to the physical layer of the database engine, e.g. to the block manager. In the absence of any generalized search tree framework in the sense of [HNP 95], the developers have the option to store their custom index structure in external files. Of course, this technique allows excellent performance results, but as external files do not participate in the transaction management of the database server,

the developers have to implement and maintain their own block manager including "industrial strength" concurrency control and recovery services.

Alternatively storing the index as a single Large Object (LOB) in the database also requires extensive implementation and maintenance efforts, particularly because the built-in locking mechanism on entire LOBs is far too coarse in a multi-user environment [BSSJ 99]. A natural way to avoid these technical problems is to exploit as much functionality of the database server as possible by mapping the index structure to a fine granular relational schema organized by built-in access methods. We follow this approach in the present paper and propose an efficient index structure for interval data that is designed to operate as logical indextype on top of the relational query language of the DBMS. The code can be implemented and maintained with minimum effort. Nevertheless our technique provides "industrial strength" stability and transaction semantics, while still showing a logarithmic worst case I/O complexity for interval intersection queries and while demonstrating the best experimentally measured performance compared to previous approaches.

3 The Relational Interval Tree

In this section, we introduce the new *Relational Interval Tree*, which efficiently implements Edelsbrunner's interval tree on top of any relational database system.

3.1 Original Interval Tree Structure

Edelsbrunner's interval tree [Ede 80] [PS 93] is an optimal data structure for intervals. Since the registered intervals are not decomposed as in the segment tree, no redundancy is produced and the space complexity is $O(n)$. The three-fold structure is illustrated in Figure 1: The backbone tree or primary structure is a balanced binary search tree that organizes the values of all bounding points of the intervals. Each of the inner nodes w is associated with two lists $L(w)$ and $U(w)$ that form the secondary structure. $L(w)$ and $U(w)$ contain, respectively, sorted lists of the lower and upper bounds of the intervals that are associated to w. An interval (l, u) is registered at the highest node it overlaps, i.e. the first node w for which $l \le w \le u$ holds when descending the tree. The tertiary structure is an additional binary tree that supports fast range scans by linking the nodes w whose lists $L(w)$ and $U(w)$ are nonempty.

3.2 Structure of the Relational Interval Tree

The basic idea of our technique relies on the following observations:

- For many applications, the *primary structure* does not need to be materialized at all. First, the nonempty nodes are linked by the tertiary structure as well. Second, even dynamic data spaces can be managed without a physical tree structure as we will show below. Only a few system parameters occupying $O(1)$ space are required.

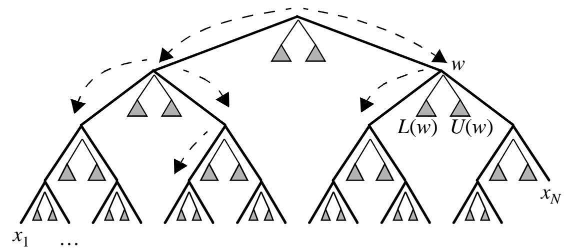


Figure 1: Three-fold structure of an interval tree.

- The *secondary* and *tertiary structure* can be combined to a relational representation that highly fits to the strength of built-in composite indexes as provided already by an RDBMS. As desired, the space complexity is $O(n/b)$ for n intervals.

The secondary structure is mapped to a relational schema as follows: Let $L(w) = \{l_1, \ldots, l_{n_w}\}$ denote the list of lower bounds of the n_w intervals that are registered at node w. The same information is represented by the set of tuples $\{(w, l_1), \ldots, (w, l_{n_w})\}$. The union over all nodes w yields a relation (*node*, *lower*). Analogously, the lists $U(w) = \{u_1, \ldots, u_{n_w}\}$ of upper bounds correspond to $\{(w, u_1), \ldots, (w, u_{n_w})\}$ and yield a relation (*node*, *upper*). Together, the relations exactly reflect the information of the secondary structure.

In an RDBMS, the two relations (*node*, *lower*) and (*node*, *upper*) are efficiently organized by built-in composite indexes. These indexes typically own a robust and highly tuned implementation, e.g. a B+-tree; they already obey the transaction semantics and are hardly outperformed by user-defined structures. Key compression techniques avoid redundancy for equal node values w. Since the indexes only manage the nonempty nodes, they already comprise the tertiary structure.

The resulting relational schema contains the attributes (*node*, *lower*, *upper*, *id*) and is supported by two composite indexes (*node*, *lower*) and (*node*, *upper*). Thus, a given interval relation is prepared for the RI-tree by adding a single attribute *node* and two indexes. Figure 2 presents the respective DDL statements in SQL. Alternatively, the artificial attribute *node* may be omitted from the base table and encapsulated by index-organized tables for the two indexes.

```
CREATE TABLE Intervals (node int, lower int, upper int, id int);
CREATE INDEX lowerIndex ON Intervals (node, lower);
CREATE INDEX upperIndex ON Intervals (node, upper);
```

Figure 2: SQL statements to instantiate an RI-Tree.

3.3 Updates in Relational Interval Trees

Whereas the registered intervals are completely managed by the relational schema, the remaining task of the primary structure is to organize the data space in order to manage insertions and query processing. The original interval tree is

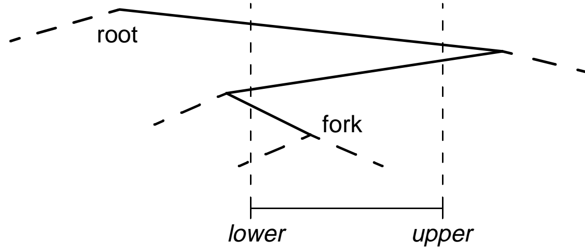


Figure 3: Fork node of an interval in the tree.

built on a static set of bounding points for the intervals. In a dynamic context, however, intervals are inserted and deleted whose actual bounding points are not known in advance. Moreover, temporal applications require an ongoing expansion of the data space. For this reason, a general and adaptable technique is required.

Our solution is as simple as effective: Rather than materializing any set of nodes, the primary structure is managed purely virtually. Thus, the bounding points of the intervals are not restricted to given values but the entire range $[1, 2^h–1]$ is supported for some $h \geq 0$. Moreover, no reorganization of any structure is necessary when inserting or deleting intervals.

In the basic version, the root node is set to $2^{h–1}$, and the tree is traversed recursively via bisection, i.e. using simple integer arithmetics but consuming no I/O operations. As already mentioned, an interval (l, u) is registered at the topmost node w for which $l \leq w \leq u$ holds, called the *fork node* (Figure 3). As an extension of the original interval tree, intervals may begin and end also at inner nodes rather than only at leaves. Points p are represented by degenerate intervals (p, p). A procedure to determine the fork node is provided in Figure 4. For computational reasons, the recursion is controlled by a decreasing step width rather than the depth in the tree.

```
FUNCTION int forkNode (int lower, int upper) {
  int node = root;
  for (int step = node/2; step >= 1; step /= 2)
    if (upper < node) node –= step;
    elsif (node < lower) node += step;
    else break;
  return node;
}
```

Figure 4: Computing the fork node of an interval.

Once the fork node is computed, inserting the interval into the relational indexes is efficiently performed by the DBMS itself. Only a single SQL statement needs to be executed (Figure 5) which also holds for the deletion of an interval. Todays RDBMS typically perform both operations by $O(\log_b n)$ I/Os on a database containing n intervals.

```
INSERT INTO Intervals
  VALUES (forkNode(:lower, :upper), :lower, :upper, :id);
```

Figure 5: Insertion of an interval (lower, upper, id).

3.4 Dynamic Expansion of the Data Space

In the basic version, the data space is fixed to a range of $2^h–1$ values yielding a tree of height h. Whereas the I/O complexity for updates is $O(\log_b n)$ and thus independent of h, the CPU time complexity linearly grows with h.

We suggest a solution that combines various aspects: First, the tree height is adjusted to the actual data distribution. Second, the data space may be expanded dynamically at the upper bound; this requirement is typical for temporal applications. On top of this, even expansions of the data space at the lower bound are supported.

The tree height is affected by two parameters: The value of the root node at which searches in the tree start, and the depth down to which algorithms have to descend in the tree. In order to control the minimum tree height, we introduce the system parameters *root*, *offset*, *leftRoot, rightRoot* and *minstep*.

Root. Dynamically adapting the parameter *root* yields two advantages: The tree height is kept minimal, and the data space may be expanded at its upper bound as new intervals arrive. A root value of 2^h is sufficient to manage intervals with $0 < lower$ and $upper < 2^{h+1}$, and $h = \lfloor \log_2(\max\{upper\}) \rfloor$ is adjusted at every insertion without affecting the existing entries, i.e. in $O(1)$.

Offset. The optimality of the root height clearly holds for an actual data space starting at 1. The intervals, however, may be located in a range $[x_1, x_N]$ with $x_1 >> 1$, i.e. far away from the origin. The resulting tree height is $\lfloor \log_2(x_N) \rfloor$ whereas a height of $\lfloor \log_2(x_N - x_1) \rfloor$ would be sufficient for a data range of length $x_N - x_1$. By shifting the intervals such that 1 becomes the lower bound of the data space, the optimal root height $h_{opt} = \lfloor \log_2(\max\{upper\} - \min\{lower\}) \rfloor$ is obtained. The amount of shift is stored in the parameter *offset*.

LeftRoot and RightRoot. Changing the *offset* parameter would cause a recalculation of all node values stored in the tree. To avoid such an unnecessary $O(n/b)$ I/O effort, *offset* is fixed after having inserted the first interval. The interval that leftmost begins in the data space, however, is not guaranteed to arrive at first to be inserted. Therefore, the space needs to be expanded at the lower bound as well as at the upper bound.

In our solution, we use 0 as global root value and manage a left and a right subtree for negative and positive node values, respectively. Instead of the single parameter *root*, two parameters *leftRoot* and *rightRoot* are maintained that manage the expansion of the data space at the lower bound and at the upper bound independently.

Minstep. The parameter *minstep* traces the lowest level i_{min} at which insertions of intervals have taken place with level 0 as the leaf level. Obviously, a query algorithm does not need to descend deeper than to level i_{min} since the sec-

ondary structures of all nodes in lower levels are empty. An estimation of i_{min} is obtained from the interval lengths:

Lemma. An interval (l, u) is not registered below the level $i_{min} = \lfloor \log_2(u - l) \rfloor$, i.e. the largest cardinal i with $2^i \le u - l$.

Proof. Assume an interval (l, u) registered at a level $j < \lfloor \log_2(u - l) \rfloor$. Then there are two successive multiples $k \cdot 2^j$ and $(k+1) \cdot 2^j$ for which $l \le k \cdot 2^j < (k+1) \cdot 2^j \le u$. Since one of the multiples is also a multiple of 2^{j+1}, the interval (l, u) had to be registered not lower than level $j+1$ which contradicts the assumption.

Figure 6 presents the final insertion procedure including the update of the persistent tree parameters. Only the artificial node value is shifted by *offset*; the lower and upper bounds of the intervals are stored without modification. The parameters *leftRoot* and *rightRoot* are initially set to 0, and *minstep* is initialized by infinity. The minimum value of 0.5 for minstep will not be stored and, thus, the implementation by an integer works well.

```
PROCEDURE insertInterval (int lower, int upper, int id) {
  // initialize offset and shift interval
  if (offset = NULL) offset = lower;
  int l = lower – offset, u = upper – offset;

  // update leftRoot and rightRoot
  if (u < 0 and l <= 2*leftRoot)   leftRoot = –2^⌊log2(–l)⌋;
  if (0 < l and u >= 2*rightRoot) rightRoot = 2^⌊log2(u)⌋;

  // descend the tree down to the fork node
  int node, step;
  if (u < 0) node = leftRoot;
  elsif (0 < l) node = rightRoot;
  else /* 0 is fork node */ node = 0;

  for (step = abs(node/2); step >= 1; step /= 2) {
    if (u < node) node –= step;
    elsif (node < l) node += step;
    else /* fork reached */ break;
  } // now node is fork node

  if (node != 0 and step < minstep) minstep = step;

  INSERT INTO Intervals VALUES (:node, :lower, :upper, :id);
}
```

Figure 6: Insertion of an interval and update of the tree parameters *offset*, *leftRoot*, *rightRoot* and *minstep*.

3.5 Analysis of the Tree Height

The parameters *offset*, *leftRoot*, *rightRoot* and *minstep* form an $O(1)$ representation of the primary structure that is dynamically adjusted to the cardinality m of the current data space. Including the global root 0, the resulting tree height is $\log_2(m) + 1$ with m given by the following formula where the minimum value of 0.5 for minstep may occur:

$$m = \max\{-leftRoot, rightRoot\} / minstep$$

In terms of data characteristics, the tree height is determined as follows: The range from *leftRoot* to *rightRoot* reflects the expansion of the data space from min{*lower*} to max{*upper*} over all currently registered intervals, and *minstep* indicates the granularity of the data space, i.e. the smallest interval length, min{*upper* – *lower*}. We increase this value by 1 to proper handle points which are represented by degenerate intervals. Nevertheless, *minstep* could be greater than min{*upper* – *lower* + 1} since even small intervals can be registered at high nodes, e.g. at the root node. In any case, the tree height does not depend on the number of intervals. In terms of the interval bounds, the tree height is $O(\log_2 m)$ where m obeys the following complexity:

$$m = O\left(\frac{\max\{upper\} - \min\{lower\}}{\min\{upper - lower + 1\}}\right)$$

4 Query Processing

Having presented the internal structure of the relational interval tree in the preceding section, we now introduce the algorithms for query processing.

4.1 Original Intersection Search

Let us shortly review the algorithm for intersection query processing in the original interval tree. For any query interval (*lower*, *upper*), the primary structure is descended as follows:

(1) Descend from the root node down to the node preceding the fork node of the query interval. Each node w on this path lies either to the left or to the right of the query interval. Suppose $w < lower$, then intervals (l, u) registered at w intersect the query interval exactly if $lower < u$. To report these r_w intervals, the sorted list $U(w)$ of upper bounds is scanned in $O(r_w)$ time. Analogously, $L(w)$ is scanned for intervals fulfilling $l < upper$ in the symmetric case $upper < w$.

(2) Descend from the fork node down to the node that is closest to *lower*. For each node w on this path, two cases are distinguished: If $w < lower$, $U(w)$ has to be scanned as before to report the intersecting intervals registered at w. Otherwise, if $lower \le w$, the query interval is known to intersect all intervals registered at the node w. In addition, all intervals from the right subtree of w are reported except if w is the fork node.

(3) Descend from the fork node down to the node closest to *upper*. Analogously to step (2), the lists $L(w)$ have to be scanned, and all registered intervals from the respective nodes are reported.

Note that the algorithm even works for degenerate intervals, i.e. $lower = upper$, thus supporting point queries as efficient as interval queries. Figure 7 provides an illustration of the algorithm. Only the nodes of the tree which are affected by the search are depicted. The symbols indicate the

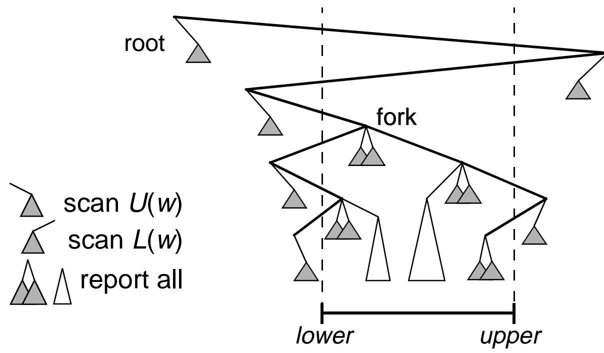


Figure 7: Query processing in the interval tree.

nodes for which $U(w)$ or $L(w)$ are scanned, and the nodes for which all entries have to be reported. Note that the latter are exactly the nodes w that are covered by the query interval, i.e. $lower \le w \le upper$.

4.2 Translation into a Single SQL Query

The basic idea of our approach is to exploit the efficiency of built-in relational indexes. Scanning the lists $U(w)$ and $L(w)$ immediately translates to an index range scan over the attributes (*node*, *upper*) and (*node*, *lower*), respectively. These attribute combinations are managed by the *upperIndex* and *lowerIndex* as defined above. Scanning the nodes w between *lower* and *upper* is supported by any of the two indexes.

Rather than immediately scanning the lists $U(w)$ and $L(w)$ while descending the tree, in our algorithm the respective nodes are collected in transient lists *leftNodes* and *rightNodes* both obeying the unary relational schema (*node*). These transient relations are managed in the transient session state thus causing no I/O effort. As for interval insertion (Figure 6), the virtual primary structure is descended by integer arithmetics without any I/O operation. Finally, a single SQL query suffices to retrieve all intersecting intervals from the database. A basic version of the query is shown in Figure 8.

```
SELECT id FROM Intervals i, leftNodes left, rightNodes right
WHERE (i.node = left.node AND i.upper >= :lower)
   OR (i.node = right.node AND i.lower <= :upper)
   OR (i.node BETWEEN :lower – offset AND :upper – offset);
```

Figure 8: Prelim. SQL query to retrieve intersecting intervals.

As illustrated in Figure 7, the nodes from *leftNodes*, from *rightNodes*, and the nodes between *lower* and *upper* are distinct from each other. The three OR-connected conditions in the WHERE clause therefore specify disjoint interval sets, and the DISTINCT option is omitted from the SELECT clause since no duplicates have to be eliminated.

4.3 Simplified SQL Query

The first transformation typically performed by relational optimizers is to split the complex OR-query into a set of three simpler queries connected by UNION ALL. The subqueries concerning *leftNodes* and *rightNodes* are efficiently supported by the respective indexes *upperIndex* and *lowerIndex* and cannot be intermixed. The third subquery that only addresses the attribute node, however, is supported by any of the two indexes. Hence, in order to reduce the cost for internal query management, we combine this subquery with the *leftNodes* subquery according to the following lemma which analogously holds for the *rightNodes* subquery.

Lemma. (*i*) The condition '*i.node = left.node*' may be substituted by the equivalent condition '*i.node BETWEEN left.min AND left.max*' if *left.node = left.min = left.max* without loss of efficiency for an index scan.

(*ii*) The condition '*i.node BETWEEN :lower – offset AND :upper – offset*' is not restricted by adding the constraint '*i.upper >= :lower*'.

Proof. (*i*) The equivalence is obvious. An index scan searches the first hit by testing *left.min* ≤ *i.node* and proceeds while testing the condition *i.node* ≤ *left.max*.

(*ii*) Since by definition, *i.node* ≤ *i.upper – offset* holds for any interval i in the tree, the condition *:lower – offset* ≤ *i.node* implies *:lower* ≤ *i.upper*.

In detail, the modifications of the query are as follows: The transient relation *leftNodes* now obeys the binary relational schema (*min*, *max*) instead of the unary schema (*node*). When descending the tree, a node w is inserted into *leftNodes* as a pair (w, w) rather than as a single value (w) as before. Finally, to include the original BETWEEN subquery, the pair (*lower – offset*, *upper – offset*) is inserted into *leftNodes*. The lemma guarantees that no intervals are missing after the transformation. Figure 9 presents the resulting two-fold SQL query for intersection search still producing no duplicates.

```
SELECT id FROM Intervals i, leftNodes left
  WHERE i.node BETWEEN left.min AND left.max
    AND i.upper >= :lower
UNION ALL
SELECT id FROM Intervals i, rightNodes right
  WHERE i.node = right.node AND i.lower <= :upper;
```

Figure 9: Final SQL statement for intersection queries.

Figure 10 shows the execution plan for the query as generated by an Oracle8i server. Most RDBMS provide an easy way ('hints') to guarantee this query plan to be chosen by the optimizer. For this example the attribute *id* was included in the indexes.

4.4 Analysis of the Algorithm

In Section 3.5, we already observed that the tree height of $h = O(\log m)$ only depends on two parameters that determine the quotient m, i.e. the extension of the intervals in the data space and the minimal interval length. It does not depend on the number n of intervals registered in the tree. The tree height is an upper bound for the number of entries in the transient relations *leftNodes* and *rightNodes*. For each of the

```
SELECT STATEMENT
  UNION-ALL
    NESTED LOOPS
      COLLECTION ITERATOR
      INDEX RANGE SCAN UPPER_INDEX
    NESTED LOOPS
      COLLECTION ITERATOR
      INDEX RANGE SCAN LOWER_INDEX
```

Figure 10: Execution plan for an intersection query in Oracle.

$O(\log m)$ entries in the transient relations, an index range scan on *upperIndex* or *lowerIndex* is performed. Such an index range scan consists of two phases. In a search phase, the beginning of the range ρ is located, and in a scan phase, the r_ρ resulting objects from the range are reported. Typical index structures such as the B+-tree in relational database systems require $O(\log_b n)$ I/O operations for the search phase on a database containing n objects, and $O(r_\rho/b)$ I/Os in the scan phase to report the r_ρ results for the range ρ.

Theorem (*Complexity of Query Processing*).
An intersection query on a Relational Interval Tree of height h that returns r results from the n intervals in the tree has an I/O complexity of

$$O(h \cdot \log_b n + r/b)$$

Proof. For each of the $O(h)$ entries in the transient relations *leftNodes* and *rightNodes*, an index search of $O(\log_b n)$ I/O complexity is performed. Scanning and reporting the total of r results requires $O(r/b)$ operations.

We conjecture that this complexity is optimal for managing intervals by relational storage structures.

4.5 General Topological Queries

In addition to the intersection query predicate, there are 13 more fine-grained temporal relationships between intervals [BÖ 98]. Obviously, also queries based on these specialized predicates are efficiently supported by the Relational Interval Tree. For some of them, there is an additional potential for optimization since they only refer to the lower bound as in *meets* or in *before*, or they only refer to the upper bound as in *met-by* or in *after*. Competing methods such as the IB+-tree [BÖ 98] or the IST [GLOT 96] efficiently support only queries referring to one of the two interval bounds, i.e. *lower* for the IB+-tree or the V-ordering and *upper* for the D-ordering. Using these techniques, queries referring to the opposite bound are processed with a poor performance since $O(n)$ comparisons are required in the worst case.

4.6 Handling Temporal Intervals

In the context of temporal databases, the special values *now* and *infinity* occur as upper values of valid time intervals [BÖ 98]. The straightforward solution to manage these intervals in separate indexes, however, yields the major disadvantage that additional SQL (sub-)queries have to be executed. This overhead is avoided by managing appropriate values for the fork nodes thus achieving a very natural integration into the Relational Interval Tree.

Infinity. In a first attempt, we set the fork node of an infinite interval to MAXINT but do not further modify the algorithms. Thus, the tree becomes very high but it is almost empty close to the root. A slight but very effective extension avoids the resulting overhead for query processing: An artificial exclusive value $fork_\infty$ is assigned to the attribute *node* of an infinite interval. At query processing time, $fork_\infty$ is inserted into the transient list *rightNodes*. Thus, the lower bounds of intervals ending at *infinity* are tested against the upper bound of the query interval as desired. Note that if choosing $fork_\infty$ = NULL, the condition '*i.node = right.node*' in Figure 9 is not evaluated correctly whereas our choice to set $fork_\infty$ = MAXINT avoids any modification of the SQL statement thus yielding a perfect integration.

Now. Whereas *infinity* is constant over time, intervals ending at *now* continuously change their upper bound. Aiming at a correct positioning of now-relative intervals within the tree at any time requires permanent modifications of the node values and, therefore, of the tree. Our solution completely avoids such an overhead and, again, uses an artificial exclusive node value, e.g. $fork_{now}$ = MAXINT – 1, which is assigned to *now*-ending intervals when being inserted. At query processing time, $fork_{now}$ is inserted into the transient table *rightNodes* exactly if $lower \leq now$, i.e. if the query interval begins in the past. As desired, the SQL query then tests the lower bounds of the *now*-ending intervals against the upper bound of the query interval.

5 Object-Relational Wrapping

The Relational Interval Tree may be easily implemented on top of *any* relational DBMS featuring a procedural query language like the Oracle8i Server with PL/SQL or the Informix Universal Server with SPL. A persistent data dictionary provides a convenient way to store index specific system parameters such as *root* or *minstep*, whereas the *leftNodes* and *rightNodes* query tables can be efficiently managed in the transient user session state. As mentioned in Section 3.3, the insertion and deletion of a new interval requires only a single SQL statement. The computation and storage of the fork node and the update of the index parameters can be performed automatically by database triggers. Whereas the complete index maintenance therefore may be managed by a trigger mechanism, query processing has to be started manually by invoking the appropriate stored procedure.

Modern object-relational DBMS provide a solution to preserve the declarative paradigm of SQL even at query time, because all maintenance and access procedures of a custom index structure are completely hidden from the user. An extensible indexing framework allows the developer to package the implementation of the access method and the corresponding index data into a user-defined indextype [Inf 98] [Ora 99a] [IBM 99]. As the object-relational data-

base server automatically triggers the maintenance and scan of custom indexes, end users can use the Relational Interval Tree just like a built-in index. With a cost model registered at the optimizer, the server is able to generate efficient execution plans for queries on interval data types.

6 Experimental Evaluation

6.1 Experimental Setup

To evaluate the performance of our approach, we have integrated the Relational Interval Tree into the Oracle Server Release 8.1.5 using PL/SQL and packaged stored procedures. All experiments have been executed on a Pentium Pro/180 server having 128 MB main memory and an U-SCSI hard drive. The database block cache was set to the default value of 200 database blocks with a block size of 2 KB. We have evaluated the performance of interval intersection queries on various interval databases with different data distributions and cardinalities (cf. Table 1). The bounding points of all intervals lie in the domain of $[0, 2^{20}-1]$. For the distributions D_3 and D_4, we assume transaction time or valid time intervals where the arrival of temporal tuples follows a Poisson process. Thus the inter-arrival time is distributed exponentially.

Name	Starting point distribution	Duration distribution
$D_1(n,d)$	uniform in $[0, 2^{20}-1]$	uniform in $[0, 2d]$
$D_2(n,d)$		exponential in $[0, \infty]$, mean = d
$D_3(n,d)$	poisson process in $[0, 2^{20}-1]$	uniform in $[0, 2d]$
$D_4(n,d)$		exponential in $[0, \infty]$, mean = d

Table 1: Sample interval databases with cardinality *n*.

As mentioned in Section 2.3, among the wide range of existing interval access methods only the static Window-List approach [Ram 97], the Tile Index [RS 99] and the Interval-Spatial Transformation technique [GLOT 96] are designed to use existing B+-trees on an as-they-are basis, i.e. without any internal modifications or augmentations. Therefore, we restrict our performance comparison to these techniques.

Window-List. In our experiments, queries on Window-Lists produced twice as many I/O operations than on the dynamic RI-tree. As the Window-List technique is a static storage structure, we do not further investigate it in the following evaluation of dynamic structures.

Tile Index (*T-index*). In our experiments, we have used the recommended hybrid indexing method of fixed- and variable-sized tiling as it is documented in [Ora 97] and [Ora 99b]. To ensure comparability to the other techniques, we have reimplemented the hybrid indexing package for one-dimensional data spaces. Our version is less complex and shows a significant performance gain over the original two-

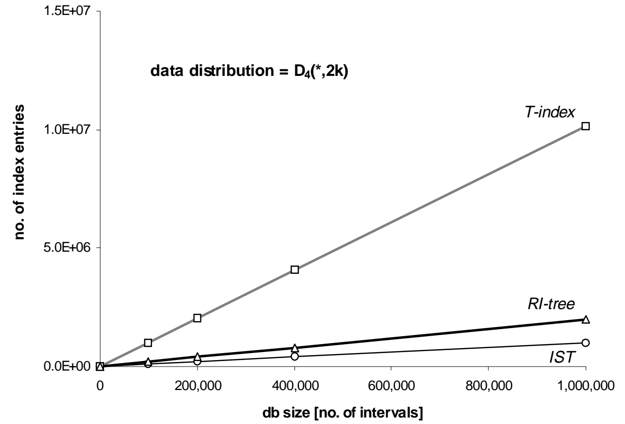


Figure 12: Number of index entries for varying database size.

dimensional implementation. When we use the Tile Index for the interval domain of $[0, 2^{20}-1]$, the fixed level parameter may be set to a value between 0 and 20. For our experiments, we took a representative sample of 1,000 intervals from each individual data distribution and determined the optimal setting for the fixed level. In most cases, the optimum for the query performance was found at the level 7, 8 or 9.

Interval-Spatial Transformation (*IST*). For the following experiments we have implemented the Interval-Spatial Transformation with *D-order* as proposed by [GLOT 96]. For integer interval bounds [*lower*, *upper*], the D-order index is equivalent to a composite index on the attributes (*upper*, *lower*) and therefore has identical performance characteristics. Range queries on D-ordered intervals can be expressed in a simple SQL statement by just testing the upper and lower bounds for intersection with the query range, as presented in Figure 11.

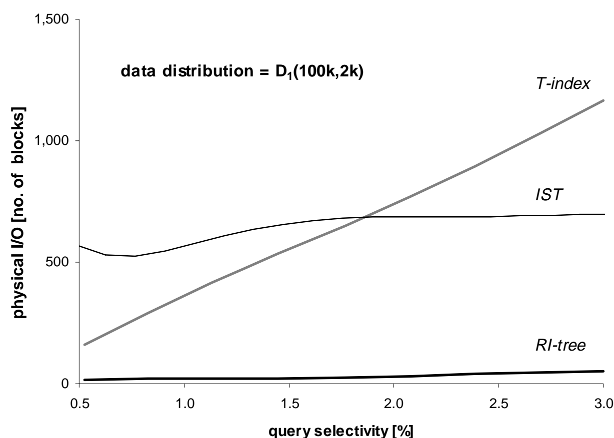
```
SELECT id FROM Intervals i
  WHERE (i.upper >= :lower AND i.lower <= :upper);
```

Figure 11: A range query for the Interval-Spatial Transformation (*IST*) on a D-ordered index.

Relational Interval Tree (*RI-tree*). We have implemented the Relational Interval Tree as it is described in the previous sections. As each data distribution of Table 1 contains intervals with length 0 (i.e. points), the granularity of the respective data space is maximal. Therefore, the *minstep* system parameter always reaches its minimum value of 1 upon index creation and the virtual backbone tree is expanded to a height of 20, unless noted otherwise.

6.2 Storage Occupation

We performed several experiments to compare the *RI-tree* with the *IST* and the *T-index*. An illustration of the storage occupation of the three techniques is given in Figure 12 for a $D_4(*,2k)$ distribution. As the *IST* technique produces no redundancy, the number of index entries is equal to the num-



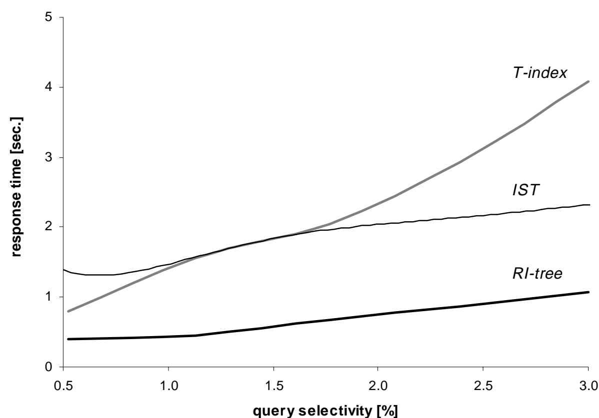
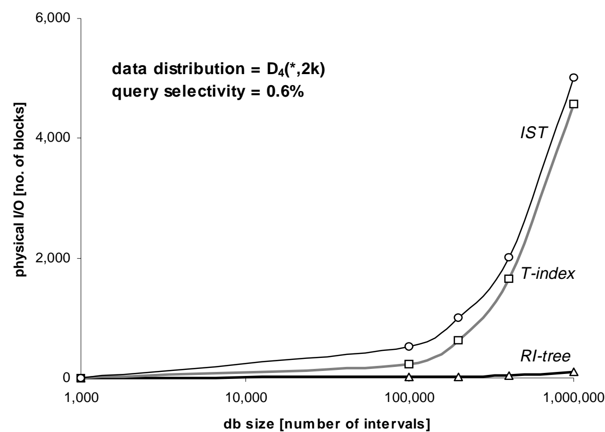


Figure 13: Disk accesses and response time for range queries on a D_1 data distribution (depending on query selectivity).



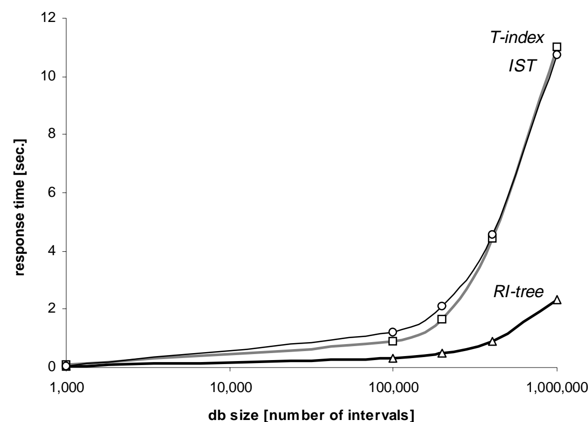


Figure 14: Disk accesses and response time for range queries on a D_4 data distribution (depending on the database size).

ber of indexed intervals. The *RI-tree* requires two index entries for each stored interval (for the lowerIndex and the upperIndex, cf. Figure 2). In our example, the *T-index* needs a redundancy factor of 10.1 to index the decomposed intervals accurately. As we have experienced in our evaluation, this causes major performance and storage problems for very large interval databases.

6.3 Query Processing

All query experiments given in this subsection have been performed with query intervals following a distribution which is compatible to the respective interval database. Our first experiment compares the number of physical disk block accesses and the response time of the three access methods depending on the selectivity of the range queries. Figure 13 depicts polynomially interpolated results of 100 range queries on a D_1(100k,2k) distribution. At a query selectivity of 0.5%, the *RI-tree* clearly outperforms the other techniques by a factor of 10.8 (*T-index*) and 46.3 (*IST*) for the disk accesses. At a 3.0% selectivity, the speedup factor is 22.8 (*T-index*) and 13.6 (*IST*). Thus the Relational Interval Tree shows a superior performance for both high and low query selectivities. The fast response times of *T-index* and *IST* (e.g. 500 I/Os in two seconds) are caused by the good clustering properties of the bulk loaded indexes and will deteriorate in a dynamic environment. For D_2(100k,2k), D_3(100k,2k), and D_4(100k,2k) datasets we measured similar results.

Figure 14 compares the scaleup of the three techniques for D_4(*,2k) datasets growing from 1,000 to 1,000,000 stored intervals. For each database size, the average number of disk accesses and the average response time of 20 range queries is presented. Both the *T-index* and the *IST* demonstrate their linear scaleup whereas the *RI-tree* scales sublinearly and shows a significant performance gain over the other access methods. The speedup factor from the *T-index* to the *RI-tree* increases from 2 to 42 (disk access) and from 2.0 to 4.9 (response time). We observed a similar improvement for the same experiments on D_1(*,2k), D_2(*,2k), and D_3(*,2k) data distributions.

The next set of experiments investigates the influence of the dataspace granularity on the query performance of the *RI-tree*. For this experiment, we restricted the domain for the interval lengths of a D_3 distribution from [0, 4k] to [500, 3.5k], [1k, 3k], and [1.5k, 2.5k], respectively. With increasing minimum interval length, fewer levels of the virtual backbone have to be traversed due to larger *minstep* values. As shown in Figure 15, the response time is almost indepen-

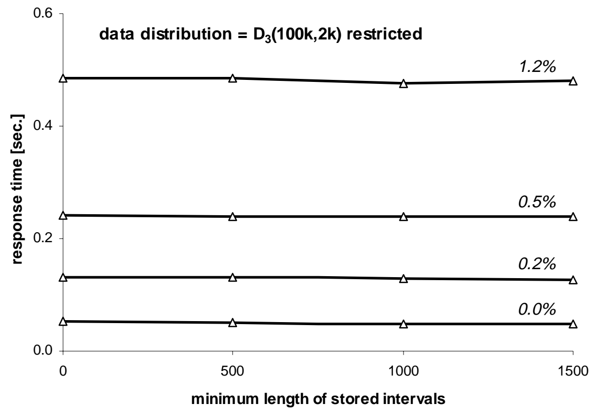


Figure 15: Response time for range queries with different selectivities on an *RI-tree* for restricted D_3 databases.

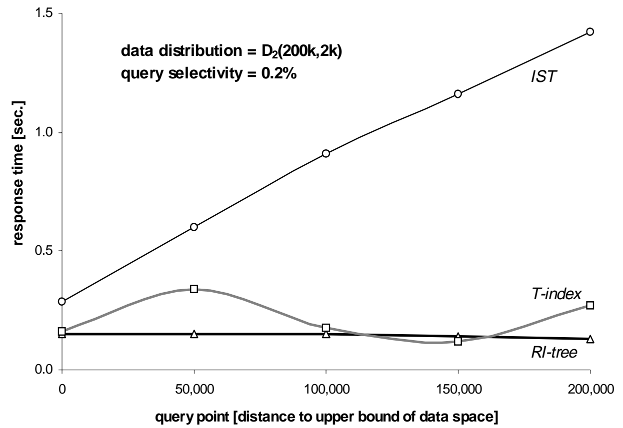


Figure 17: Response time for a "sweeping" point query on a D_2 data distribution. The *IST* degenerates with higher distance to the upper bound of the data space.

dent of the minimum length of the stored intervals. So the resulting height *h* of the virtual backbone has only little empirical significance. The response times for the different selectivities illustrate also the desired property that the performance of queries is largely bound to the number of results.

The next series of experiments compares the influence of the mean of interval duration on the query performance of the different techniques. Figure 16 depicts the average results for a sample of 20 range queries on various D_4(100k,*) datasets with increasing average length of intervals. The *T-index* and the *RI-tree* require about the same response time for range queries, if the average length of the indexed intervals is very low. As short intervals do not suffer from the spatial decomposition, the redundancy caused by the *T-index* tiling approach decreases from 10.1 to 1 when the mean value of interval duration is reduced from 2,000 to 0. Even for a pure point database, the *RI-tree* performs slightly better than the *T-index*. The benefit of the *RI-tree* becomes obvious for a higher mean of duration. Both the *RI-tree* and the *IST* perform better as longer intervals are stored in the database.

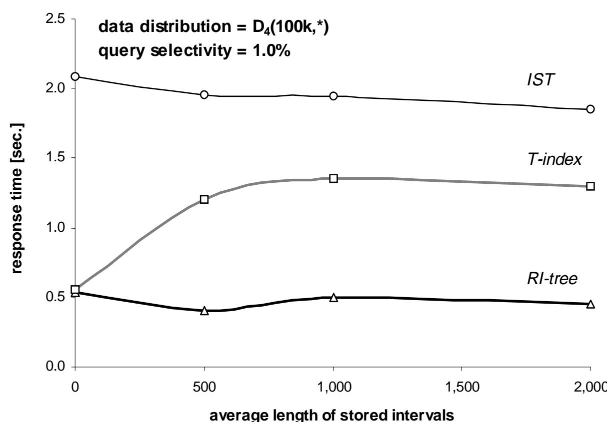


Figure 16: Response time on a D_4 data distribution with varying mean of interval length. Even for small intervals, the *RI-tree* outperforms the *T-index* approach.

As expected, the location of the query range with respect to the data domain exerts a strong influence on the performance of the *IST*. In Figure 17 we illustrate this effect by 'sweeping' a query point starting at the upper bound of the data space where the bound index on (*upper*, *lower*) benefits the most from the high selectivity in the first indexed column. The comparison between the *RI-tree* and *T-index* reveals another interesting aspect of this experiment: Although for point queries the *T-index* performs at its best as it retrieves no duplicates caused by redundancy, the *RI-tree* is still slightly better on the average. We obtained these results as well for the other interval data distributions D_1, D_3 and D_4.

7 Conclusions

In this paper, we presented the Relational Interval Tree which is a new access method for interval data. It can be created for any relational or object-relational table containing intervals. As we have shown, the main design goals for our new approach have been fulfilled:

- *Integration.* The RI-tree is not a stand-alone concept. It can easily be implemented on top of any relational DBMS. As much functionality as possible of built-in indexes is exploited and no changes or additions to the internal layer of the database server are made. Therefore the effort of code development and code maintenance is minimal. For modern database servers featuring an object-relational application program interface, a natural and seamless integration can be achieved while preserving the declarative paradigm of SQL.
- *Performance.* Our analytical and experimental evaluation of the RI-tree shows superior performance characteristics compared to previous approaches. This is achieved by introducing the virtual primary structure. Although the structure is space-oriented, the storage of intervals is object-driven and, thus, no storage space is wasted for empty regions in the data space.

- *Extensions.* Our basic concept supports a wide range of efficient application specific extensions. We have illustrated this by the dynamic expansion of the data space, by handling the special temporal variables *now* and *infinity*, and by discussing fine-grained topological query types.

The flexibility and extensibility of the RI-tree concept opens up a number of interesting research problems and applications. A promising extension is the application of the Skeleton Index technique to the RI-tree, because a partial materialization of the primary structure can be adapted to the expected data distribution and, for example, the management of string intervals is supported.

References

[AT 95] Ang C.-H., Tan K.-P.: *The Interval B-Tree*. Information Processing Letters 53(2): 85-89, 1995.

[AV 96] Arge L., Vitter J. S.: *Optimal Dynamic Interval Management in External Memory.* Proc. 37th Annual Symp. on Foundations of Computer Science, 560-569, 1996.

[BKK 99] Böhm C., Klump G., Kriegel H.-P.: *XZ-Ordering: A Space-Filling Curve for Objects with Spatial Extension*. Proc. 6th Int. Symp. on Large Spatial Databases, LNCS 1651, 75-90, 1999.

[BKSS 90] Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: *The R*-tree: An Efficient and Robust Access Method for Points and Rectangles*. Proc. ACM SIGMOD Int. Conf. on Management of Data, 322-331, 1990.

[BÖ 98] Bozkaya T., Özsoyoglu Z. M.: *Indexing Valid Time Intervals*. Proc. 9th Int. Conf. on Database and Expert Systems Applications, LNCS 1460, 541-550, 1998.

[BSSJ 99] Bliujute R., Saltenis S., Slivinskas G., Jensen C.S.: *Developing a DataBlade for a New Index*. Proc. IEEE Int. Conf. on Data Engineering, 314-323, 1999.

[CLR 90] Cormen T. H., Leiserson C. E., Rivest R. L.: *Introduction to Algorithms*. Cambridge, MA: MIT-Press, 1990.

[Ede 80] Edelsbrunner H.: *Dynamic Rectangle Intersection Searching*. Institute for Information Processing Report 47, Technical University of Graz, Austria, 1980.

[EWK 90] 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, 1-12, 1990.

[FMB 00] Fenk R., Markl V., Bayer R.: *Management and Query Processing of One-Dimensional Intervals with the UB-Tree*. PhD Workshop, 7th Conf. on Extending Database Technology, Konstanz, Germany, 7-10, 2000.

[FR 89] Faloutsos C., Roseman S.: *Fractals for Secondary Key Retrieval*. Proc. 8th ACM Symp. on Principles of Database Systems, 247-252, 1989.

[GLOT 96] Goh C. H., Lu H., Ooi B. C., Tan K.-L.: *Indexing Temporal Data Using Existing B+-Trees*. Data & Knowledge Engineering, Elsevier, 18(2): 147-165, 1996.

[Gut 84] Guttman A.: *R-trees: A Dynamic Index Structure for Spatial Searching.* Proc. ACM SIGMOD Int. Conf. on Management of Data, 47-57, 1984.

[HJ 96] Hanson E., Johnson T.: *Selection Predicate Indexing for Active Databases Using Interval Skip Lists*. Information Systems, 21(3): 269-298, 1996.

[HNP 95] Hellerstein J. M., Naughton J. F., Pfeffer A.: *Generalized Search Trees for Database Systems*. Proc. 21st Int. Conf. on Very Large Databases, 562-573, 1995.

[HP 94] Hellerstein J. M., Pfeffer A.: *The RD-Tree: An Index Structure for Sets*. Technical Report #1252, University of Wisconsin at Madison, Oct. 1994.

[IBM 99] IBM Corp.: *IBM DB2 Universal Database Application Development Guide, Version 6.* Armonk, NY, 1999.

[Inf 98] Informix Software, Inc.: *DataBlade Developers Kit User's Guide*. Menlo Park, CA, 1998.

[KRVV 93] Kanellakis P. C., Ramaswamy S., Vengroff D. E., Vitter J. S.: *Indexing for Data Models with Constraints and Classes*. Proc. 12th ACM Symp. on Principles of Database Systems, 233-243, 1993.

[KPS 00] Kriegel H.-P., Pötke M., Seidl T.: *Relational Interval Tree*. EPO patent application no. 00112031.0, 2000.

[KS 91] Kolovson C. P., Stonebraker M.: *Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data*. Proc. ACM SIGMOD Int. Conf. on Management of Data, 138-147, 1991.

[LT 98] Lee C., Tseng T.-M.: *Temporal Grid File: A File Structure for Interval Data*. Data & Knowledge Engineering, Elsevier, 26(1): 71-97, 1998.

[MTT 00] Manolopoulos Y., Theodoridis Y., Tsotras V. J.: *Advanced Database Indexing*. Chapter 4: *Access Methods for Intervals.* Boston, MA: Kluwer, 2000.

[ND 99] Nascimento M. A., Dunham M. H.: *Indexing Valid Time Databases via B+-Trees*. IEEE Trans. on Knowledge and Data Engineering 11(6): 929-947, 1999.

[Ora 97] Oracle Corp.: *Oracle8 Spatial Cartridge User's Guide and Reference, Rel. 8.0.4*. Redwood City, CA, 1997.

[Ora 99a] Oracle Corp.: *Oracle8i Data Cartridge Developer's Guide, Rel. 8.1.5*. Redwood City, CA, 1999.

[Ora 99b] Oracle Corp.: *Oracle8i Spatial User's Guide and Reference, Rel. 8.1.5.* Redwood City, CA, 1999.

[PS 93] Preparata F. P., Shamos M. I.: *Computational Geometry: An Introduction*. 5th ed., Springer, 1993.

[Ram 97] Ramaswamy S.: *Efficient Indexing for Constraint and Temporal Databases*. Proc. 6th Int. Conf. on Database Theory, LNCS 1186, 419-431, 1997.

[RS 99] Ravada S., Sharma J.: *Oracle8i Spatial: Experiences with Extensible Databases*. Proc. 6th Int. Symp. on Large Spatial Databases, LNCS 1651, 355-359, 1999.

[Sam 90a] Samet H.: *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley, 1990.

[Sam 90b] Samet H.: *Applications of Spatial Data Structures*. Reading, MA: Addison-Wesley, 1990.

[SOL 94] 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, 274-281, 1994.

[SRF 87] Sellis T., Roussopoulos N., Faloutsos C.: *The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects*. Proc. Int. Conf. on Very Large Databases, 507-518, 1987.

[TCG+ 93] Tansel A. U., Clifford J., Gadia S., Jajodia S., Segev A., Snodgrass R.: *Temporal Databases: Theory, Design and Implementation*. Redwood City, CA, 1993.