Proc. 7th Int. Symposium on Spatial and Temporal Databases (**SSTD 2001**).
Lecture Notes in Computer Science, © Springer Verlag 2001.

Interval Sequences: An Object-Relational Approach to Manage Spatial Data

Hans-Peter Kriegel, Marco Pötke, Thomas Seidl

University of Munich, Institute for Computer Science
Oettingenstr. 67, 80538 Munich, Germany
Tel. +49-89-2178-2191, Fax: +49-89-2178-2192
{kriegel, poetke, seidl}@dbs.informatik.uni-muenchen.de

Abstract. The design of external index structures for one- and multidimensional extended objects is a long and well studied subject in basic database research. Today, more and more commercial applications rely on spatial datatypes and require a robust and seamless integration of appropriate access methods into reliable database servers. This paper proposes an efficient, dynamic and scalable approach to manage one-dimensional interval sequences within off-the-shelf object-relational database systems. The presented technique perfectly fits to the concept of space-filling curves and, thus, generalizes to spatially extended objects in multidimensional data spaces. Based on the Relational Interval Tree, the method is easily embedded in modern extensible indexing frameworks and significantly outmatches Linear Quadtrees and Relational R-trees with respect to usability, concurrency, and performance. As demonstrated by our experimental evaluation on an Oracle server with real GIS and CAD data, the competing methods are outperformed by factors of up to 4.6 (Linear Quadtree) and 58.3 (Relational R-tree) for query response time.

1 Introduction

After two decades of temporal and spatial index research, the efficient management of one- and multidimensional extended objects has become an enabling technology for many novel database applications. The interval, or, more generally, the sequence of intervals, is a basic datatype for temporal and spatial data. Interval sequences are used to handle finite domain constraints [Ram 97] or to represent periods on transaction or valid time dimensions [TCG+ 93]. Typical applications of one-dimensional interval sequences include the temporal tracing of user activity for service providers: a query like “Find all customers who were online last month between 5 and 6 pm” maps to an intersection query of interval sequences on a database storing online periods of all registered users. When applied to space-filling curves, interval sequences naturally represent spatially extended objects with even intricate shapes. By expressing spatial region queries as interval sequence intersections, vital operations for two-dimensional GIS and environmental information systems [MP 94] can be supported. Efficient and scalable database solutions are also required for two- and three-dimensional CAD applications to cope with rapidly growing amounts of dynamic data and highly concurrent workflows. Such applications include the digital mock-up of vehicles and airplanes [BKP 98], haptic simulations [MPT 99], or general engineering data management [KMPS 01].

For commercial use, a seamless and capable integration of temporal and spatial indexing into industrial-strength databases is essential. Unfortunately, most commercially relevant database systems provide no built-in access method for temporal and spatial datatypes [CCF+ 99], nor do they offer a generic framework to facilitate the integration of user-defined search trees based on disk-blocks, as proposed by Hellerstein, Naughton and Pfeffer [HNP 95]. In this paper, we therefore follow the paradigm of using *relational access methods* [KPS 00] to integrate index support for temporal and spatial datatypes. As access methods of this class are designed on top of the pure SQL layer, they can be easily implemented on virtually any available relational database server. Replicating techniques based on the *Linear Quadtree* [TH 81] [OM 88] [Sam 90] [IBM 98] [Ora 99b] [RS 99] [FFS 00] decompose spatial objects into tiles which correspond to constrained segments on a space-filling curve. In contrast, our new technique supports arbitrary intervals across tile boundaries, and therefore, yields a significantly lower redundancy. It is based on the *Relational Interval Tree* (RI-tree) [KPS 00], a relational adaption of the main-memory Interval Tree [Ede 80].

The remainder of the paper is organized as follows: Section 2 reviews the benefits and limitations of available extensible indexing frameworks. Section 3 surveys the related work on relational access methods for spatial data. Section 4 describes the application of the RI-tree to store and retrieve interval sequences. Section 5 generalizes our technique to multidimensional applications by mapping spatially extended objects to interval sequences on a space-filling curve. Following an experimental evaluation in Section 6 on 2D-GIS and 3D-CAD databases, the paper is concluded in Section 7.

2 Extensible Indexing

Extensible indexing frameworks, as already proposed by Stonebraker [Sto 86], enable developers to extend the set of built-in index structures by custom access methods in order to support user-defined datatypes and predicates. This section discusses the main properties of extensible indexing within object-relational database systems, including Oracle8i Server [Ora 99a] [SMS+ 00], IBM DB2 Universal Database [IBM 99] [CCF+ 99] or Informix Universal Server [Inf 98] [BSSJ 99].

2.1 Declarative Integration

An object-relational indextype encapsulates stored functions for opening and closing an index scan, iterating over resulting records, and performing insert, delete and replace operations on the indexed table. It is complementary to the functional implementation of user-defined predicates. The indextype also implements functions for the estimation of query selectivity and processing cost. The custom computation of persistent statistics and histograms is triggered by the usual administrative SQL statements. The query optimizer considers a custom index among alternative access paths and may actually choose it for the execution plan. This approach preserves the declarative paradigm of SQL, as it requires no manual query rewriting in order to handle user-defined predicates efficiently. As an example, we can now write:

```
SELECT * FROM db WHERE intersects(db.sequence, query_sequence);
```

2.2 Relational Implementation

Although available extensible indexing frameworks provide a gateway to seamlessly integrate user-defined access methods into the standard process of query optimization, they do not facilitate the actual implementation of the access method itself. Modifying or enhancing the database kernel is usually not an option for database developers, as the embedding of block-oriented access methods into concurrency control, recovery services and buffer management causes extensive implementation efforts and maintenance cost [Kor 99], at the risk of weakening the reliability of the entire system. We therefore focus on relational storage structures to implement the object-relational indextype for intersections of interval sequences. By following this approach, derived index data is stored in one or more *index tables* besides the original *data table*. An index table is organized by a built-in index structure, e.g. a B+-tree. Queries and updates on relational storage structures are processed by pure SQL statements. The robust transaction semantics of the database server is therefore fully preserved. According to the semantics of the index tables, we have identified two generic schemes for relational storage structures:

Positional scheme. The index tables contain transformed user data rows. Each row in the data table is directly mapped to a set of linear positions, i.e. rows in the index tables. Inversely, each row in an index table exclusively belongs to a single row in the data table. In order to support queries, the linear positions are indexed by a built-in index, e.g. a B+-tree. Examples for the positional scheme include the Linear Quadtree, the one-dimensional RI-tree and our optimization for interval sequence and multidimensional queries (cf. Sections 4 and 5).

Navigational scheme. The index tables contain data that is recursively traversed at query time in order to determine the resulting tuples. Therefore, a row in the index table is logically shared by many rows in the data table. Examples for the navigational scheme are the *Relational R-tree* [RRSB 99] and the *Relational X-tree* [BBKM 99], which map the nodes of a hierarchical directory to a flat relational schema. To support the navigation through the directory table, a built-in index is created on artificial node identifiers. To execute a navigational query by a single SQL statement, a recursive version of SQL like SQL:1999 [EM 99] is required.

Although the navigational scheme offers a straightforward way to simulate any block-based index structure on top of a relational data model, it suffers from the fact that navigational data is locked like user data. As two-phase locking on index tables is too restrictive, the possible level of concurrency is unnecessarily decreased. For example, uncommitted node splits in a hierarchical directory may lock entire subtrees against concurrent updates. Built-in indexes solve this problem by committing structural modifications separately from content changes [KB 95]. This approach is not feasible on the SQL layer without braking up the user transaction. A similar overhead exists with logging.

These drawbacks are not shared by the positional scheme, as any row in the index tables exclusively belongs to one single data object. Therefore, relational storage structures following the positional scheme raise no overhead in combination with locking and logging. They do not only preserve the semantics of concurrent transactions and recovery services, but also inherit the high concurrency and efficient recovery of built-in access methods. We therefore follow the positional approach in this paper.

3 Related Work

A wide variety of access methods for one- and multidimensional extended objects has been published so far. Following the scope of this paper, we will focus our review on relational access methods and refer the reader to the surveys of Manolopoulos, Theodoridis and Tsotras [MTT 00] or Gaede and Günther [GG 98]. As an extensive comparison of index structures for one-dimensional intervals has been done by Kriegel, Pötke and Seidl [KPS 00], this section concentrates on multidimensional access methods for objects with a spatial (or temporal) extension. We classify these techniques with respect to inherent data replication, i.e. the need to produce redundancy for spatial objects and their identifiers.

3.1 Non-Replicating Spatial Access Methods

Non-replicating access methods use simple spatial primitives such as rectilinear hyperrectangles for one-value approximations of extended objects. The following relational techniques implement the positional scheme: The *hB-tree* of Lomet and Salzberg [LS 89] transforms spatially extended objects to points in a higher dimensional space and, thus, avoids the need to split data objects due to space partitioning. The relational *DOT* index of Faloutsos and Rong [FR 91] organizes this higher-dimensional space in a B+-tree by means of a space-filling curve. Unfortunately, query regions in the higher dimensional space are much more complex than in the original space. Furthermore, the transformation to points produces a highly nonuniform distribution even for originally uniform data [GG 98]. The *XZ-Ordering* of Böhm, Klump and Kriegel [BKK 99] is a space-filling curve specialized for extended objects. By encoding overlapping regions in the original data space, tight approximate representations of bounding boxes are mapped to a single region code and stored in a B+-tree. The *2dMAP21* method of Nascimento and Dunham [ND 97] indexes the one-dimensional projections of two-dimensional rectangles by separate B+-trees. Apart from the known drawbacks of this inverted list approach, only inclusion queries are supported efficiently by the proposed one-dimensional query processor.

R-trees as presented by Guttman [Gut 84], Beckmann et al. [BKSS 90], or Kamel and Faloutsos [KF 94] partition the original data space into overlapping regions. Thereby the spatial distribution and clustering of the data may be preserved. However, in the absence of a block-based interface to the database kernel, the pages of a hierarchical index structure have to be simulated on top of a relational model by following the navigational scheme. This approach has been adopted for the *Relational X-tree* of Berchtold et al. [BBKM 99] and the *Relational R-tree* of Ravi Kanth et al. [RRSB 99]. As a major drawback, concurrent updates are not supported well (cf. Section 2.2).

In many applications, GIS or CAD objects feature a very complex and fine-grained geometry. The rectilinear bounding box of the brake line of a car, for example, would cover the whole bottom of the indexed data space. A non-replicating storage of such data causes region queries to produce too many false hits that have to be eliminated by subsequent filter steps. For such applications, the accuracy can be improved by decomposing spatial objects independently from the index partitions, or, alternatively, by using a replicating index structure, which is inherently tuned for redundancy.

3.2 Replicating Spatial Access Methods

Replicating access methods partition the data space into mutually disjoint bucket regions. Thereby, spatially extended objects spanning more than one bucket region are decomposed. Typical index structures of this kind are the *R+-tree* of Sellis, Roussopoulos and Faloutsos [SRF 87] or the *Cell Tree* of Günther [Gün 89]. Their corresponding navigational mapping, again, is not suited for a high level of concurrency.

Tile-based techniques map spatial partitions on a regular grid to one-dimensional region codes indexed by built-in B+-trees. They can be classified as positional. The general concept of managing spatial objects by linear region codes has already been proposed by Tropf and Herzog [TH 81] and has been termed *Linear Quadtree* by Samet [Sam 90]. In the following, a *cell* denotes an element of a regular cubic grid. A cell corresponds to a *pixel* in 2D or to a *voxel* in 3D spaces. The number of cells along each dimension is called the *resolution* of the grid. A *tile* is a set of cells that is generated by recursive binary partitioning of the grid, e.g. according to the Z-order. A tile can be represented by two integers (*value* and *level*). A highly tuned sort-merge join algorithm for processing queries on tiled objects has been proposed by Orenstein and Manola [OM 88].

To eliminate the need of intrusive modifications to the query processor, Freytag, Flasza, and Stillger [FFS 00] have presented an adaption to object-relational database systems. The relational Linear Quadtree of the Oracle Spatial Cartridge [Ora 99b] [RS 99] and the IBM DB2/ESRI Spatial Extender [IBM 98] further refines this concept of tile-based indexing: spatial objects are decomposed at a user-defined fixed quadtree level (*fixlev*), and the resulting ordered Z-tiles are indexed by a built-in B+-tree. Each resulting fixed-sized tile contains a set of variable-sized tiles as a fine-grained representation of the covered geometry. Multidimensional query regions are also decomposed according to the fixed level. The technique then combines an equijoin on the fixed-sized tiles in the index with a sequential scan over the corresponding variable-sized tiles. Note that due to possible fruitless scans below the indexed fixed level, this popular variant of the Linear Quadtree does not guarantee blocked output. Finding an adequate fixed level for the expected data distribution is crucial. With *fixlev* set too high, too much redundancy and effort for duplicate elimination emerges due to small fixed-sized tiles, whereas a low *fixlev* causes too much approximation error and possible overhead for scanning many variable-sized tiles. Therefore, the setting has to be carefully tuned to achieve optimal cost for query processing and update operations [Ora 99b]. By introducing two additional fixed-sized tiling levels [IBM 98], huge objects are indexed at a lower accuracy and tiny objects at a higher accuracy.

If we employ space-filling curves, spatial objects are naturally represented by interval sequences [Jag 90] [Gae 95] [MJFS 96] [FJM 97]. The replicating Linear Quadtree technique can be regarded as a Segment Tree [PS 93] managing the resulting one-dimensional intervals: each interval is decomposed into segments, where each segment corresponds to a multidimensional tile. Therefore, the original redundancy of a spatial interval sequence is destroyed. In this paper, we preserve this redundancy by replacing the segment-based storage of the Linear Quadtree by the interval-based storage of the Relational Interval Tree.

4 Management of Interval Sequences

The RI-tree of Kriegel, Pötke and Seidl [KPS 00] is an application of extensible indexing for interval data. Based on relational storage, single intervals are stored, updated and queried with an optimal complexity. After discussing a naive algorithm that simply considers an interval sequence as a set of independent entities, we present an optimized version that exploits the connection between the elements of an interval sequence.

4.1 The Relational Interval Tree[1]

The RI-tree strictly follows the paradigm of relational access methods since its implementation is restricted to (procedural) SQL and does not assume any lower level interfaces. 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.

The conceptual structure of the RI-tree is based on a virtual binary tree of height h which acts as a backbone over the range $[0 \ldots 2^h-1]$ of potential interval bounds. Traversals are performed purely arithmetically by starting at the root value 2^h and proceeding in positive or negative steps of decreasing length 2^{h-i}, thus reaching any desired value of the data space in $O(h)$ time. This backbone structure is not materialized, and only the root value 2^h is stored persistently in a meta data tuple. For the relational storage of intervals, the nodes of the tree are used as artificial key values: Each interval is assigned a fork node, i.e. the first intersected node when descending the tree from the root node down to the interval location.

An instance of the RI-tree consists of two relational indexes which in an extensible indexing environment are at best managed as index-organized tables. These indexes then obey the relational schema *lowerIndex* (*node, lower, id*) and *upperIndex* (*node, upper, id*) and store the artificial fork node value *node*, the bounds *lower* and *upper*, and the *id* of each interval. The RI-tree therefore implements the positional scheme of relational access methods and, thus, fully preserves the effectivity of the underlying concurrency control. As any interval is represented by exactly one entry for each the lower and the upper bound, $O(n/b)$ disk blocks of size b suffice to store n intervals. For inserting or deleting intervals, the *node* values are determined arithmetically, and updating the indexes requires $O(\log_b n)$ I/O operations per interval. We store an interval sequence by simply labelling each associated interval with the sequence identifier. Figure 1 illustrates the RI-Tree by an example.

4.2 Interval Query Processing

To minimize barrier crossings between the procedural runtime environment and the declarative SQL layer, an interval intersection query (*lower, upper*) is processed in two steps. In the procedural query preparation step, range queries are collected in two transient tables, *leftNodes* and *rightNodes*, which are obtained by a purely arithmetic traversal of the virtual backbone from the root node down to *lower* and to *upper*, respectively. The visited nodes fall into three classes: Nodes left of *lower* are collected in

[1] Patent pending

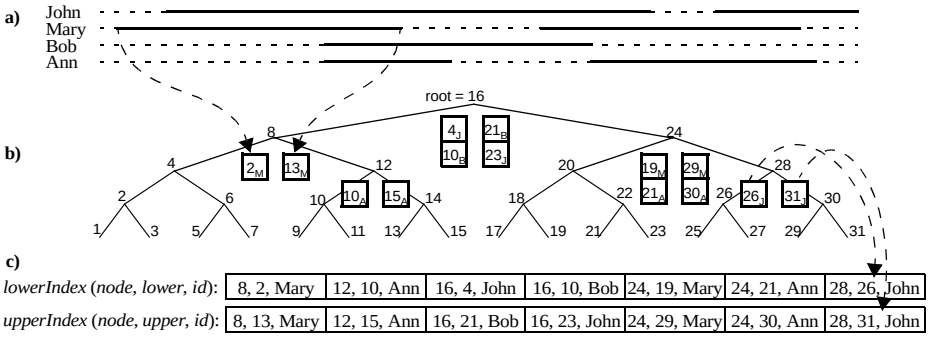

Figure 1. a) Four sample interval sequences. **b)** The virtual backbone positions the intervals. **c)** Resulting relational indexes.

leftNodes since they may contain intervals who overlap *lower*. Analogously, nodes right of *upper* are collected in *rightNodes* since their intervals may contain *upper*. The intervals registered at nodes between *lower* and *upper* are guaranteed to overlap the query and, therefore, will be reported without any further comparison by a so-called *inner query*. The query preparation procedure is purely main memory-based and, thus, yields no I/O operations.

In the second step, the declarative query processing, the transient tables are joined with the relational indexes *upperIndex* and *lowerIndex* by a single, three-fold SQL statement (Figure 2). The upper bound of each interval registered at nodes in *leftNodes* is checked against *lower*, and the lower bounds of intervals from *rightNodes* are checked against *upper*. We call the corresponding queries *left queries* and *right queries*, respectively. The *inner query* corresponds to a simple range scan over the nodes in (*lower*, *upper*). The SQL query yields $O(h \cdot \log_b n + r/b)$ I/Os to report r results from an RI-tree of height h. The height h of the backbone tree depends on the expansion and resolution of the data space, but is independent of the number n of intervals. Furthermore, output from the relational indexes is fully blocked for each join partner.

```
SELECT id FROM upperIndex i, :leftNodes left
   WHERE i.node = left.node AND i.upper >= :lower    // left queries
UNION ALL
SELECT id FROM lowerIndex i, :rightNodes right
   WHERE i.node = right.node AND i.lower <= :upper // right queries
UNION ALL
SELECT id FROM lowerIndex i   /* or upperIndex i */
   WHERE i.node BETWEEN :lower AND :upper;      // inner queries
```

Figure 2. SQL statement for a single query interval with bind variables *leftNodes*, *rightNodes*, *lower*, *upper*.

The naive way to process interval sequence intersections on an RI-tree is to perform independent queries for each of the intervals. As an example, let us consider the interval

sequence ⟨(43, 52), (55, 85), (87, 91)⟩. Figure 3 illustrates the resulting 24 queries for the query intervals. The traversed paths of the target RI-tree of height 8 are depicted, and the numbers denote the node values, e.g. 128 for the root of the virtual backbone. In the example, the 7 gray queries are generated for the first interval (43, 52), the 9 white queries for (55, 85), and the 8 black queries for (87, 91). From the total of 24 queries, 11 are left queries, 10 are right queries, and 3 are inner queries.

4.3 Gap Optimization for Interval Sequences

The naive approach disregards the important fact that the intervals of an interval sequence represent the same object. As a major disadvantage, many overlapping queries are generated. This redundancy causes an unnecessary high main memory footprint for the transient query tables, an overhead of query time, and lots of duplicates in the result set which have to be eliminated. Our basic idea is to avoid the generation of redundant queries, rather than to discard the respective queries after their generation. A related concept is known as *streaming* in the context of data space decomposition for decomposable searching problems [EO 85].

In the example, the root node (128) is queried by three right queries. An interval registered at the root node is reported three times if its lower bound is less or equal to 52, and twice if its lower bound is greater than 52 but not greater than 85. The right query of the rightmost (black) interval suffices to report all resulting intervals from node 128, and discarding the gray and the white query prevents the generation of duplicates without yielding false dismissals. Node 64 is also queried three times, i.e. by a gray right query, a black left query, and a white inner query. A registered interval is reported at least once – due to the inner query – and up to three times if its lower bound is less or equal to 52 and its upper bound is greater or equal to 87. Here, the white inner query suffices to produce the complete result at node 64, and the left and right queries yield only duplicates. Analogously, the nodes 32, 48, 52, 56, 80, 84, 88, and 96 are queried twice and may produce duplicates.

A particular case occurs at node 86 to which a white right query and a black left query are assigned. Though resulting intervals may be reported twice, both queries are necessary for a complete result. If discarding the white right query, an interval with a lower

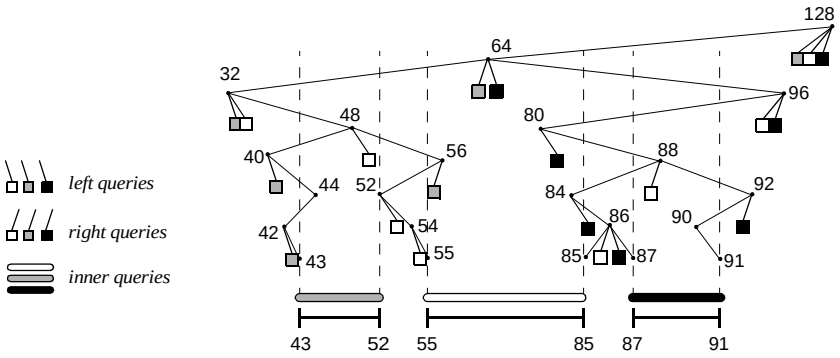

Figure 3. The 24 naive queries for an interval sequence.

bound less or equal to 85 and an upper bound less than 87 is missing in the output. Analogously, an interval with a lower bound greater than 85 and an upper bound greater or equal to 87 is a false dismissal if the black left query is omitted.

We now present the fundamental rule for optimizing the queries for a sorted interval sequence $q = \langle q_1, \ldots, q_n \rangle$, $q_i = (lower_i, upper_i)$. For the inexistent intervals q_0 and q_{n+1}, we assume the bounds $upper_0 = -\infty$ and $lower_{n+1} = \infty$.

Theorem. For a sorted query interval sequence $q = \langle q_1, \ldots, q_n \rangle$ with intervals $q_i = (lower_i, upper_i)$, the result of an intersection query is complete if for each q_i, query generation is restricted to nodes n with $upper_{i-1} < n < lower_{i+1}$ where $upper_0 = -\infty$ and $lower_{n+1} = \infty$.

Proof. For any interval q_i of an interval sequence q, the following queries are redundant to queries for the neighboring intervals q_{i-1} or q_{i+1} and, thus, may be discarded without affecting the completeness:

(*i*) Left queries at nodes $n \le upper_{i-1}$ (n lies to the left of the gap between q_{i-1} and q_i)

(*ii*) Right queries at nodes $n \ge lower_{i+1}$ (n lies to the right of the gap between q_i and q_{i+1})

We now show these propositions in detail.

(*i*) A left query for q_i at a node n retrieves the intervals r registered at n for which $r.upper \ge lower_i$. Let us now focus to the interval q_{i-1} immediately to the left of q_i. If n lies within q_{i-1}, i.e. $lower_{i-1} \le n \le upper_{i-1}$, the inner query for q_{i-1} reports all intervals from n and, thus, even all results for q_i at n independent of their individual bounds. If n lies even to the left of q_{i-1}, i.e. $n < lower_{i-1}$, a left query for q_{i-1} is generated at n since n belongs to the path from the root to $lower_{i-1}$ as well as to the path to $lower_i$. This left query reports intervals r with $r.upper \ge lower_{i-1}$ from n and, as a subset, contains the results r for q_i at n fulfilling $r.upper \ge lower_i$. Summarizing, the results of left queries for q_i at nodes $n \le upper_{i-1}$ are already reported by queries for the preceding interval q_{i-1}.

(*ii*) Analogously, the results of right queries for q_i at nodes $n \ge lower_{i+1}$ are reported by inner queries or right queries for the subsequent interval q_{i+1}. q.e.d.

4.4 Integrating Inner Queries

The proposed optimization in [KPS 00] which integrates the inner queries into the set of left queries is a purely syntactic rewriting that does not affect the number of queries. Contrary to rewriting, the exploitation of the following observation typically avoids the generation of 75% of the inner queries.

As an example, consider the interval (43, 52) in Figure 3 which yields the inner query ‘node BETWEEN 43 AND 52’ or, rewritten, ‘node BETWEEN 43 AND 52 AND upper ≥ 43’. The left query at node 42 translates to ‘node = 42 AND upper ≥ 43’ or, rewritten, ‘node BETWEEN 42 AND 42 AND upper ≥ 43’. The left query range (42, 42) is immediately adjacent to the inner query range (43, 52). Thus, merging both queries to the single range query ‘node BETWEEN 42 and 53 AND upper ≥ 43’ saves one (cached) B+-tree lookup without producing any redundancy.

Figure 4 illustrates the frequent applicability of the inner query optimization. For an odd interval bound, the outer adjacent node is even and, thus, is reached earlier when descending the tree (cases a, b, c). The inner query may be merged with the closest left

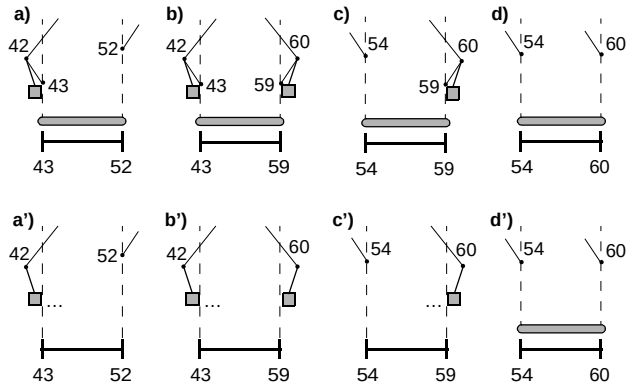

Figure 4. Top row: The four cases of interval bounds, **a)** odd–even, **b)** odd–odd, **c)** even–odd, **d)** even–even. Bottom row: Integration of the inner queries into *left* (**a'**, **b'**) or *right* (**c'**) *queries*, indicated by '…'. For **d'**, no integration is possible.

query (cases a, b) or right query (cases b, c). If both interval bounds are odd (43 and 59 in case b), the algorithm arbitrarily chooses the adjacent left node (42) or right node (60). Only if both interval bounds are even (case d), the inner query cannot be merged with an adjacent query. The descending algorithm stops at even interval bounds which reside higher in the tree than odd nodes which are located at the leaf level. Therefore, no left or right query is immediately adjacent and, though still syntactically rewritable, the inner query cannot be merged with a neighboring query. For uniformly distributed interval bounds, this situation applies to 25% of all cases. The optimization is thus highly effective.

4.5 Final Optimized Algorithm

The presented optimizations are orthogonal and may be integrated into the naive algorithm independent from each other. When descending from the root to the interval bounds, single queries beyond the adjacent gaps are suppressed, and inner queries may be combined with adjacent left or right queries. The resulting left and right queries are collected in two transient tables, *leftNodes* (*from*, *to*, *lower*) and *rightNodes* (*from*, *to*, *upper*), indicating the single nodes (if *from* = *to*) or the range of nodes (if *from* < *to*) to be scanned, and the lower or upper bound of the individual query intervals. Query processing itself is performed by a single two-fold SQL statement that merges the join of the transient *leftNodes* and the persistent *upperIndex* with the join of the transient *rightNodes* and the persistent *lowerIndex* (Figure 5).

Figure 6 illustrates the effect of the optimization to our prior example. Having originally generated 24 queries, now only 9 queries are produced: Three gray left queries to the left of the first interval (left boundary queries), a single white left query in the first gap, one white right query and one black left query in the second gap, and three black right queries to the right of the last interval (right boundary queries). All inner queries have been integrated into adjacent left queries as indicated by dots (…).

```
SELECT id FROM intervals i, :leftNodes left
   WHERE i.node BETWEEN left.from AND left.to
     AND i.upper >= left.lower   // using upperIndex
UNION
SELECT id FROM intervals i, :rightNodes right
   WHERE i.node BETWEEN right.from AND right.to
     AND i.lower <= right.upper   // using lowerIndex
```

Figure 5. SQL statement for interval sequence queries.

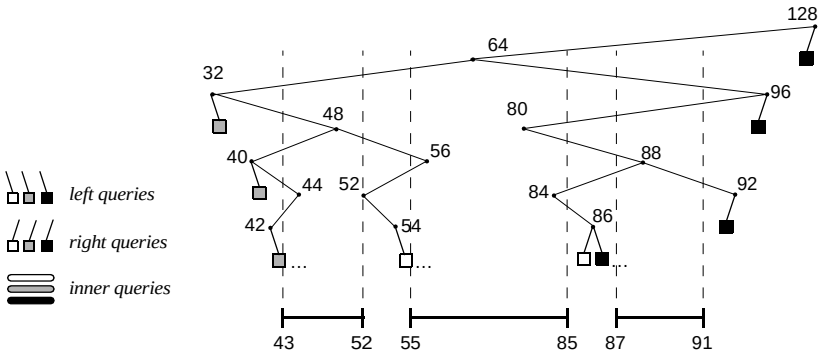

Figure 6. The 9 reduced queries for the interval sequence from Figure 3.

5 Spatially Extended Objects

This section addresses the management of multidimensional extended objects by interval sequences in the Relational Interval Tree. Spatial region queries are transformed to interval sequence intersection queries.

5.1 Mapping Extended Objects to Interval Sequences

Encoding extended objects by means of space-filling curves is a common approach for many applications. First, an appropriate grid resolution is defined that determines the finest possible granularity for the approximation of spatial objects. Each cell of the grid is then encoded by a single integer number and, thus, an extended object is represented by a set of integers. Typically, cells in close spatial proximity are encoded by similar integers or, conversely, contiguous integers encode cells in close spatial neighborhood. Many space-filling curves have been evaluated with respect to their spatial clustering properties. Examples include the lexicographic-, Z- or Hilbert-order, with the Hilbert-order generating the least intervals per object [Jag 90] [FR 89] but being also the most complex linear ordering. As a good trade-off between redundancy and complexity, we use the Z-order throughout the following examples.

Since the numbers representing an extended object form some continuous ranges, we immediately obtain interval sequences from this encoding. Managing the resulting interval sequences by a dedicated interval storage structure now exploits the spatial clustering properties of space-filling curves in a very explicit and immediate way. Aside its

advantages for relationally storing intervals as described above, the RI-tree qualifies very well for interval sequences on space-filling curves. First, the extension of the data space is known in advance and does not dynamically expand while inserting new intervals. Thus, the *root* value is constant, and the height of the tree does not depend on the number of managed objects. Moreover, the interval bounds are integer values and, therefore, perfectly fit to the basic variant of the RI-tree.

5.2 Controlling Accuracy and Redundancy

A basic parameter for the mapping of extended objects to interval sequences is the granularity, i.e. the resolution of the underlying grid. When refining the resolution, the approximations become more accurate, but redundancy increases. Figure 7a illustrates the granularity-bound decomposition into variable-sized Z-tiles (top row) and into Z-ordered interval sequences (bottom row). The approximation error is the ratio of the dead space to the object area. According to the extensive analysis given in [MJFS 96] and [FJM 97], the asymptotic redundancy of a tile- or interval-based decomposition is in both cases proportional to the surface of the approximated object. Nevertheless, as intervals on a Z- or Hilbert-curve may span many tiles, their average number is significantly lower than the average number of tiles.

On top of the resolution of the data space and the clustering properties of the space-filling curve, a more fine-grained control of the trade-off between redundancy and accuracy is desired for many applications. First, the granularity may have to be adjustable for each individual object rather than to generally apply to all stored objects. Second, the resolution is fixed at database creation time whereas an object may have to be approximated differently at insertion time and at query time. An approach to control this trade-off is the concept of size-bound and error-bound approximation [Ore 89] beyond the mentioned granularity-bound approximation [Gae 95]. A recursive subdivision procedure stops if the desired redundancy (size-bound) or the desired maximum approximation error (error-bound) is reached. Figure 7b illustrates the size-bound approximation

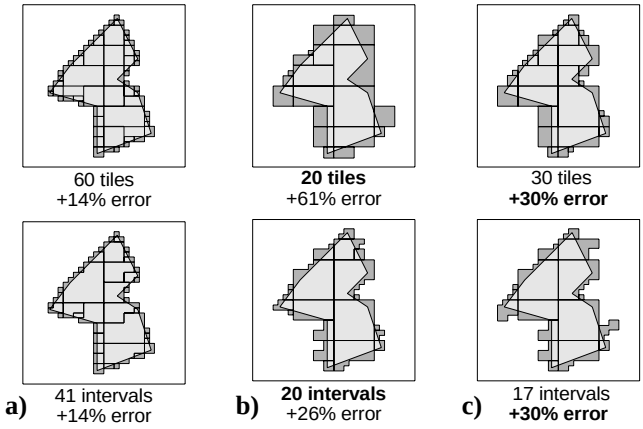

Figure 7. a) Granularity-bound, **b)** size-bound, and **c)** error-bound decomposition into Z-tiles (top row) and Z-ordered interval sequences (bottom row).

of a polygon into variable-sized tiles (top row) and into Z-ordered interval sequences (bottom row). Examples for an error-bound approximation are depicted in Figure 7c. These examples already illustrate the superiority of unconstrained interval sequences over Quadtree tiling: interval sequences yield about half the approximation error for the size-bound approach (cf. Figure 7b) and half the redundancy for the error-bound approach (cf. Figure 7c). The redundancy of the tiling approach may further increase, if one or more fixed-sized tiling levels are introduced (cf. Section 3.2).

We adapt the algorithms of [Ore 89] by integrating the management of generated intervals into the recursive spatial decomposition. The algorithm returns the sorted interval sequence for a given *d*-dimensional spatial object (Figure 8). Starting with a single interval encoding the entire data space, non-empty tiles are subdivided recursively following the chosen space-filling curve. Three cases may occur: first, if an interval starts or ends with an empty tile, the range of numbers encoding the empty tile is removed from the interval. The approximation error is thus decreased without affecting redundancy. Second, if a tile is empty but does not contain an interval bound, the interval is split into two by removing the range of numbers encoding the empty tile. Whereas the approximation error again is decreased, redundancy is increased by one in this case. Third, if none of the tiles is empty, the encoding interval is neither shrunk nor split. Depending on the desired approximation type, the recursion is terminated by a size-bound or an error-bound criterion.

```
fun decompose (object, bound) → sequence of intervals;
begin
   Sequence result = ⟨ [0..2^h–1] ⟩;
   PriorityQueue tiles = ⟨(∞, entire_space)⟩;
   while bound exceeded     // size bound or error bound
         and not tiles.empty() do     // granularity bound
      tile = tiles.dequeueGreatest ();
      if tile ∩ object is empty then
         remove the cell codes of tile from result;
      elsif | tile | > 1 then
         split tile into {tile_1, …, tile_n};
         for i = 1..n do tiles.enqueue(|tile_i – object|, tile_i);
      end if;
   end do;
   return result;
end decompose;
```

Figure 8. Recursive decomposition of a spatial object into an interval sequence.

An alternative approach proceeds bottom-up and iteratively closes the smallest gaps between the intervals. For a size-bound approximation, this algorithm stops if the maximal redundancy has been reached. For the error-bound case, the approximation error is controlled by a minimum gap length (*mingap*) for the resulting interval sequence, and the redundancy is minimized.

6 Empirical Evaluation

6.1 Experimental Setup

To evaluate the performance of our approach, we have implemented the naive and optimized intersection queries of Section 4 on top of an RI-tree for the Oracle Server Release 8.1.7. We have used PL/SQL for the computation of the transient query tables *leftNodes* and *rightNodes* and executed the single SQL statement of Figure 5 for the actual interval sequence query. All experiments have been performed on an Athlon/750 machine with IDE hard drives. The database block cache was set to 50 disk blocks with a block size of 8 KB and was used exclusively by one active session. In our experiments, we have examined two-dimensional GIS polygon data (*2D-GIS*) from the SEQUOIA 2000 benchmark [SFGM 93]. In addition, we used different datasets of voxelized three-dimensional CAD parts supplied by two European car manufacturers. As the results for these two CAD databases were comparable, only the evaluation on one car project is reported here (*3D-CAD*). In the next subsections, we evaluate storage and performance characteristics for the 2D-GIS and 3D-CAD databases by comparing the following three relational access methods:

RI-tree (naive and optimized). We used the proposed mapping of Section 5 to transform the spatial objects to interval sequences on different space-filling curves. Unless otherwise noted, all experiments are based on the Z-order. We have set the granularity to 30 bits (2D-GIS) and 27 bits (3D-CAD), i.e. a grid resolution of 2^{15} cells per dimension in 2D and 2^9 cells per dimension in 3D. The grid resolution has been chosen to match the finest possible granularity that users should be able to materialize in the spatial index. An error bound for the decomposition of spatial objects was defined by closing all gaps in an interval sequence that are smaller than *mingap*.

Linear Quadtree. We took the variant that is used in many commercial spatial data engines [Ora 99b] [RS 99] [IBM 98]. We implemented a two- and a three-dimensional version with one fixed level, parameterized by *fixlev*. As described in Section 3.2, finding a good setting for *fixlev* is crucial for the performance and redundancy of the Linear Quadtree. As all spatial objects are decomposed to the fixed-sized tiles, the *fixlev* has to be tuned for the desired approximation error. By design, the Linear Quadtree only supports a static fixed-level optimization. Thus, the performance and redundancy might degenerate due to changing data distributions. Below the fixed level, variable-sized tiles refine the approximation as long as the resulting redundancy does not deteriorate. The granularity bound for variable-sized tiles was set to 2^{32} cells per dimension in 2D and to 2^9 cells per dimension in 3D.

Relational R-tree. By following the navigational scheme of relational access methods, a relational implementation of the well-known R-tree structure can be easily achieved. For the following empirical evaluation, we have transformed the hierarchical block-based R-tree directory to a relational scheme (cf. Section 3.1). This approach has already been adopted for commercial products [RRSB 99]. In order to adjust the approximation error in the R-tree to the RI-tree and Linear Quadtree, the complex geometries of the 2D-GIS and 3D-CAD databases have been decomposed by using standard algorithms [KHS 91] [SK 93] [BKP 98]. The resulting sets of rectilinear boxes have been bulk-loaded with the popular *VAMSplit* approach [WJ 96] [RRSB 99].

6.2 Redundancy, Accuracy, and Storage

In the first set of experiments, we evaluated the approximation error, redundancy, and storage occupancy for the competing techniques. The approximation error is defined by the average ratio of dead space to the polygon area. Redundancy for the RI-tree is measured by the average number of intervals per polygon, i.e. the cardinality of the corresponding interval sequence. The approximation error was controlled by the *mingap* parameter. For the Linear Quadtree, redundancy is equal to the average number of variable-sized tiles. We determined an optimal *fixlev* of 10, 11, 12, and 13 for different approximation errors. Figure 9 depicts the resulting redundancy for the replicating techniques, where each sampling point is labelled with the chosen setting of *mingap* and *fixlev*, respectively.

To achieve a redundancy of 10, the Linear Quadtree requires 3.6 times more approximation error than the RI-tree, because it is restricted to tiles and generates redundancy to populate the fixed level rather than to improve the overall approximation. Inversely, at an average approximation error of 33%, the Linear Quadtree generates an 8 times higher redundancy than the RI-tree. At an approximation error of 64%, the RI-tree and the Linear Quadtree occupy about 24 MB and 51 MB of index space, respectively. The average redundancy for the corresponding interval sequences is about five times higher than for the box sets indexed by the R-tree. Nevertheless, the total storage occupancy of the R-tree (17 MB) is comparable to the RI-tree. This is due to the higher storage complexity of multidimensional boxes compared to one-dimensional intervals and the storage overhead for the relational R-tree directory. For the following query experiments, the databases have been indexed with an average approximation error of 64% (2D-GIS) and 0% (3D-CAD).

6.3 Query Processing

This subsection compares the query performance of the techniques by averaging the results of 200 region queries. The region queries consist of window/polygon queries (2D) and box/collision queries (3D) following a distribution which is compatible to the respective spatial database. To reduce the overhead of barrier crossings and value passing between the procedural runtime environment (PL/SQL, in our case) and the declarative SQL layer, both the Linear Quadtree and the RI-tree precompute the transient que-

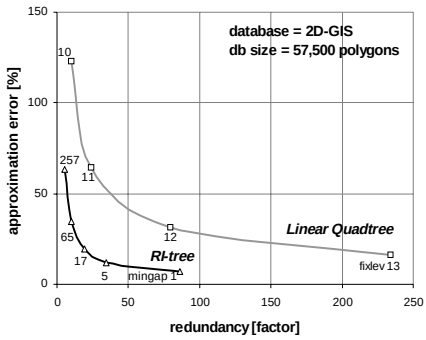

Figure 9. Redundancy vs. accuracy for the Linear Quadtree and the RI-tree.

ry tables and bind them at once to the SQL statement. Thus, the total number of rows in the query tables should be minimized in order to keep a low memory profile of concurrent sessions. Figure 10 presents the average number of one-dimensional range queries generated for a region query. For the RI-tree, we distinguish three types of join partners in the *leftNodes* and *rightNodes* tables: *Boundary queries* are left or right queries assigned to nodes outside the span of the interval sequence. *Gap queries* stem from nodes within the gaps, and *inner queries* from nodes within the actual intervals. The naive RI-tree query preparation generates 7.7 times more range queries than our optimized approach. The Linear Quadtree still requires 5.2 times more transient join partners, as query regions are decomposed to tiles rather than to unrestricted interval sequences. Therefore, the optimized RI-tree scales well in a multi-user environment, as it drastically reduces the main memory footprint per session. Furthermore, redundant queries are avoided, and, thus, the cost for duplicate elimination on the result set are minimized.

Figure 11a compares the average physical disk block accesses for region queries on the 2D-GIS database. Both the naive and optimized RI-tree queries clearly outperform the Linear Quadtree by a factor of up to 15. Opposed to the RI-tree, the Linear Quadtree suffers from its non-blocked output (cf. Section 4.2). Moreover, the join partners in the transient query tables of the naive RI-tree are redundant to a large extent, whereas the queries for the Linear Quadtree are not (cf. Figure 10). Therefore, the LRU database

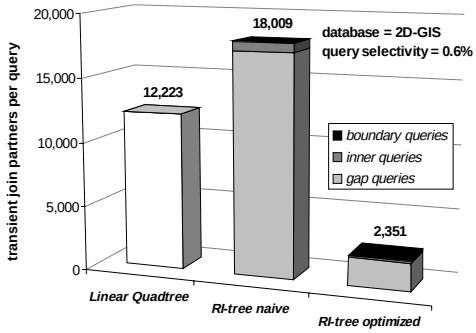

Figure 10. Average number of transient join partners for region queries.

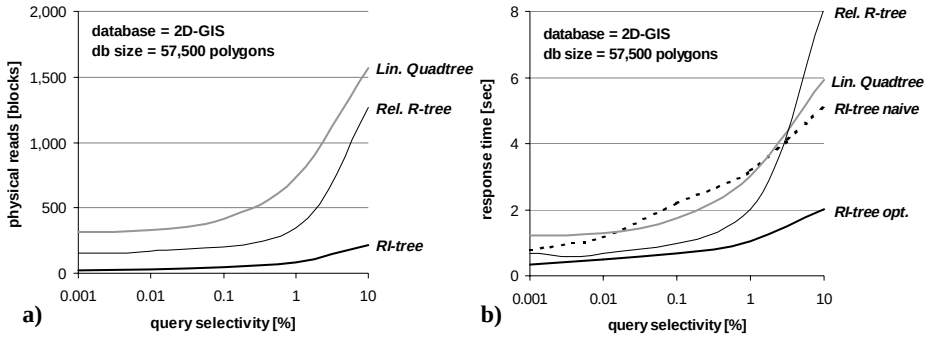

Figure 11. Averages for region queries on 2D data: **a)** physical disk accesses, **b)** response time.

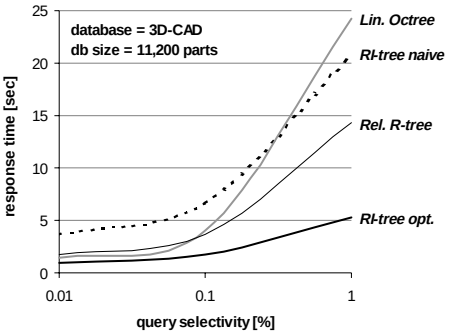

Figure 12. Average response time for region queries on 3D data.

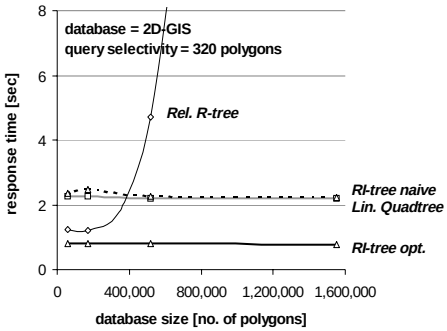

Figure 13. Scale-up of the average response time for region queries on 2D data.

cache eliminates the difference between the naive and the optimized RI-tree, whereas much more B+-tree lookups remain to be processed for the Linear Quadtree. The Relational R-tree requires up to 7.4 times more disk accesses than the RI-tree.

Figure 11b clearly reveals the effect of our query optimization over the naive RI-tree. The speed-up in the response time of the optimized RI-tree to the Linear Quadtree ranges from 2.9 to 3.6. On the whole, the speed-up of our technique to the Linear Quadtree is achieved by physical I/O optimization, whereas the speed-up to the naive RI-tree stems from logical I/O and CPU cost reduction. The speed-up from 2.0 to 4.0 of the RI-tree to the Relational R-tree is also caused by much lower physical I/O cost. Figure 12 presents the results for region queries on the 3D-CAD database. The optimized RI-tree outperforms the Linear Quadtree by a factor of up to 4.6, and the Relational R-tree by a factor of up to 2.7.

The next experiment in Figure 13 shows the scale-up of the competing approaches for two-dimensional databases growing from 57,500 to 1.55 million polygons. The larger databases have been created by replicating polygon data outside of the regions touched by the sample queries. The queries therefore retrieve a constant number of 320 results on the average. As the height of the underlying B+-trees remains constant, the average speed-up factor of 2.8 of the optimized RI-tree over the Linear Quadtree and of 2.9 over the naive RI-tree are almost unaffected. In contrast, the spatial partitioning of the Relational R-tree deteriorates significantly with increasing database size, mainly due to highly overlapping data objects. In consequence, the average speed-up from the RI-tree to the Relational R-tree increases from 1.6 to 58.3. Note that the Relational R-tree has been created by a bulk-load operation, whereas our RI-tree is fully dynamic. Moreover, the Relational R-tree suffers from its navigational implementation (cf. Subsection 2.2), and, thus, does not support concurrent updates of directory entries.

6.4 Impact of Space-Filling Curves

The number of intervals generated by various space-filling curves has already been thoroughly studied [Jag 90] [Gae 95] [MJFS 96] [FJM 97]. Extensive evaluations identified the Hilbert curve as one of the best mappings. Unfortunately, existing index structures as the Linear Quadtree replicate intervals on tile bounds and therefore destroy the

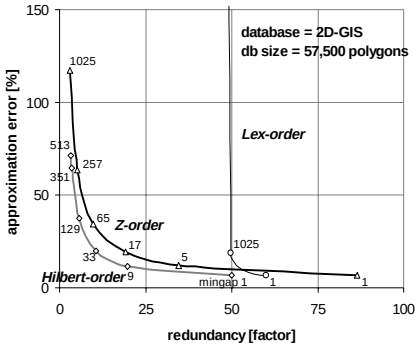

Figure 14. Redundancy vs. accuracy on the RI-tree for different space-filling curves.

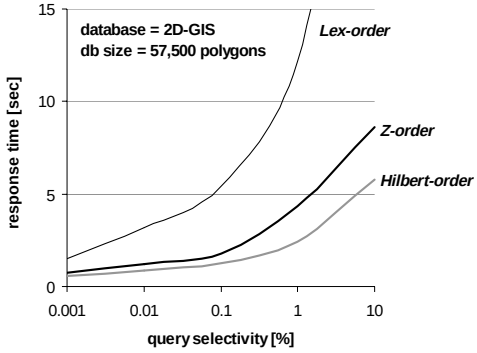

Figure 15. Average response time for different space-filling curves (optimized RI-Tree).

original redundancy of the resulting interval sequences (cf. Figure 9). According to a performance study on Linear Quadtrees, the Hilbert curve or other fractal space-filling curves do not yield a substantial performance improvement over the Z-order [BKK 99].

One of the most important features of our RI-tree application is the ability to handle arbitrary interval sequences at their original redundancy. We therefore conclude this experimental evaluation with a comparison of different space-filling curves for the optimized RI-tree. Figure 14 compares the redundancy of interval sequences for different approximation errors. As expected, the Hilbert-order generates the smallest sequences, whereas the lexicographic order is hardly adjustable to desired approximation errors. According to Figure 15, the achieved clustering for database and query objects largely affects the corresponding response times. The polygons have been indexed with an approximation error of 6.8% (*mingap* 1). Although at this accuracy, the lexicographic order generated a lower redundancy than the Z-order, its poor clustering deteriorates the query performance. The Hilbert-order does not only deliver the least redundancy, but also the best clustering.

7 Conclusions

In this paper, we presented a relational access method for one-dimensional interval sequences based on the Relational Interval Tree. We introduced algorithms for mapping multidimensional extended objects to interval sequences and thereby extended its applicability to spatial region queries on multidimensional databases. In contrast to existing tile-based access methods, our proposal manages arbitrary interval sequences at a minimal redundancy. Our experimental evaluation on two- and three-dimensional data demonstrates that our technique significantly outperforms the Linear Quadtree and the Relational R-tree. Due to a much lower memory footprint, it also scales better for a multi-user environment. The optimized RI-tree is easily integrated into commercial database systems by using an extensible indexing framework. By encapsulating our method into an object-relational indextype for interval sequence objects or spatial entities, the declarative paradigm of SQL is fully preserved.

Another contribution of this work is the classification of relational access methods into positional and navigational approaches. As the RI-tree follows the positional scheme, high concurrency and efficient logging are naturally achieved when managing interval sequences. This is not the case for navigational techniques, including the Relational R-tree. Therefore, interval sequences on the RI-tree provide an efficient, dynamic, scalable and yet simple solution to temporal and spatial indexing in off-the-shelf object-relational databases.

In our future work, we plan to investigate other query types such as object ranking for nearest neighbor search or spatial join algorithms. Another interesting extension is the support of the 9-intersection model [ES 93] for topological predicates. In order to support cost-based optimization, we are working on selectivity estimation and cost models for interval sequence intersections and their spatial interpretation.

References

[BBKM 99] Berchtold S., Böhm C., Kriegel H.-P., Michel U.: *Implementation of Multidimensional Index Structures for Knowledge Discovery in Relational Databases*. Proc. 1st DaWaK, LNCS 1676, 261-270, 1999.

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

[BKP 98] Berchtold S., Kriegel H.-P., Pötke M.: *Database Support for Concurrent Digital Mock-Up*. Proc. IFIP Int. Conf. PROLAMAT, 499-509, 1998.

[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, 322-331, 1990.

[BSSJ 99] Bliujute R., Saltenis S., Slivinskas G., Jensen C.S.: *Developing a DataBlade for a New Index*. Proc. IEEE 15th ICDE, 314-323, 1999.

[CCF+ 99] Chen W., Chow J.-H., Fuh Y.-C., Grandbois J., Jou M., Mattos N., Tran B., Wang Y.: *High Level Indexing of User-Defined Types*. Proc. 25th VLD), 554-564, 1999.

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

[EO 85] Edelsbrunner H., Overmars M. H.: *Batched Dynamic Solutions to Decomposable Searching Problems*. Journal of Algorithms, 6(4), 515-542, 1985.

[EM 99] Eisenberg A., Melton J.: *SQL:1999, formerly known as SQL3*. ACM SIGMOD Record, 28(1): 131-138, 1999.

[ES 93] Egenhofer M., Sharma J.: *Topological Relations Between Regions in R^2 and Z^2*. Proc. 3rd SSD, LNCS 692, 316-336, 1993.

[FFS 00] Freytag J.-C., Flasza M., Stillger M.: *Implementing Geospatial Operations in an Object-Relational Database System*. Proc. 12th SSDBM, 2000.

[FJM 97] Faloutsos C., Jagadish H. V., Manolopoulos Y.: *Analysis of the n-Dimensional Quadtree Decomposition for Arbitrary Hyperrectangles*. IEEE TKDE 9(3): 373-383, 1997.

[FR 89] Faloutsos C., Roseman S.: *Fractals for Secondary Key Retrieval*. Proc. ACM PODS, 247-252, 1989.

[FR 91] Faloutsos C., Rong Y.: *DOT: A Spatial Access Method Using Fractals*. Proc. IEEE 7th ICDE, 152-159, 1991.

[Gae 95] Gaede V.: *Optimal Redundancy in Spatial Database Systems*. Proc. 4th SSD, LNCS 951, 96-116, 1995.

[GG 98] Gaede V., Günther O.: *Multidimensional Access Methods*. ACM Computing Surveys 30(2): 170-231, 1998.

[Gün 89] Günther O.: *The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases*. Proc. IEEE 5th ICDE, 598-605, 1989.

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

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

[IBM 98] IBM Corp.: *DB2 Spatial Extender Admin. Guide and Ref., 2.1.1*. Armonk, NY, 1998.

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

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

[Jag 90] Jagadish H. V.: *Linear Clustering of Objects with Multiple Attributes*. Proc. ACM SIGMOD, 332-342, 1990.

[KB 95] Kornacker M., Banks D.: *High-Concurrency Locking in R-Trees*. Proc. 21st VLDB, 134-145, 1995.

[KF 94] Kamel I., Faloutsos C.: *Hilbert R-tree: An Improved R-tree Using Fractals*. Proc. ACM SIGMOD, 500-509, 1994.

[KHS 91] Kriegel H.-P., Horn H., Schiwietz M.: *The Performance of Object Decomposition Techniques for Spatial Query Processing*. Proc. 2nd SSD, LNCS 525, 257-276, 1991.

[KMPS 01] Kriegel H.-P., Müller A., Pötke M., Seidl T.: *Spatial Data Management for Computer Aided Design*. Proc. ACM SIGMOD, 2001.

[Kor 99] Kornacker M.: *High-Performance Extensible Indexing*. Proc. 25th VLDB, 699-708, 1999.

[KPS 00] Kriegel H.-P., Pötke M., Seidl T.: *Managing Intervals Efficiently in Object-Relational Databases*. Proc. 26th VLDB, 407-418, 2000.

[MJFS 96] Moon B., Jagadish H. V., Faloutsos C., Saltz J. H.: *Analysis of the Clustering Properties of Hilbert Space-filling Curve*. Techn. Report CS-TR-3611, University of Maryland, 1996.

[MP 94] Medeiros C. B., Pires F.: *Databases for GIS*. ACM SIGMOD Record, 23(1): 107-115, 1994.

[MPT 99] McNeely W. A., Puterbaugh K. D., Troy J. J.: *Six Degree-of-Freedom Haptic Rendering Using Voxel Sampling*. Proc. ACM SIGGRAPH, 401-408, 1999.

[MTT 00] Manolopoulos Y., Theodoridis Y., Tsotras V. J.: *Advanced Database Indexing*. Boston, MA: Kluwer, 2000.

[ND 97] Nascimento M. A., Dunham M. H.: *Using Two B+-Trees to Efficiently Process Inclusion Spatial Queries*. Proc. 5th Int. Workshop on Advances in Geographic Information Systems (GIS), 5-8, 1997.

[OM 88] Orenstein J. A., Manola F. A.: *PROBE Spatial Data Modeling and Query Processing in an Image Database Application*. IEEE Trans. on Software Engineering, 14(5): 611-629, 1988.

[Ora 99a] Oracle Corp.: *Oracle8i Data Cartridge Developer's Guide, 8.1.6*. Redwood City, CA, 1999.

[Ora 99b] Oracle Corp.: *Oracle Spatial User's Guide and Reference, 8.1.6*. Redwood City, CA, 1999.

[Ore 89] Orenstein J. A.: *Redundancy in Spatial Databases*. Proc. ACM SIGMOD, 294-305, 1989.

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

[Ram 97] Ramaswamy S.: *Efficient Indexing for Constraint and Temporal Databases*. Proc. 6th ICDT, LNCS 1186, 419-413, 1997.

[RRSB 99] Ravi Kanth K. V., Ravada S., Sharma J., Banerjee J.: *Indexing Medium-dimensionality Data in Oracle*. Proc. ACM SIGMOD, 521-522, 1999.

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

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

[SFGM 93] Stonebraker M., Frew J., Gardels K., Meredith J.: *The SEQUOIA 2000 Storage Benchmark*. Proc. ACM SIGMOD, 2-11, 1993.

[SK 93] Schiwietz M., Kriegel H.-P.: *Query Processing of Spatial Objects: Complexity versus Redundancy*. Proc. 3rd SSD, LNCS 692, 377-396, 1993.

[SMS+ 00] Srinivasan J., Murthy R., Sundara S., Agarwal N., DeFazio S.: *Extensible Indexing: A Framework for Integrating Domain-Specific Indexing Schemes into Oracle8i*. Proc. IEEE 16th ICDE, 91-100, 2000.

[SRF 87] Sellis T., Roussopoulos N., Faloutsos C.: *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects*. Proc. 13th VLDB, 507-518, 1987.

[Sto 86] Stonebraker M.: *Inclusion of New Types in Relational Data Base Systems*. Proc. IEEE 2nd ICDE, 262-269, 1986.

[TH 81] Tropf H., Herzog H.: *Multidimensional Range Search in Dynamically Balanced Trees*. Angewandte Informatik, 81(2), 71-77, 1981