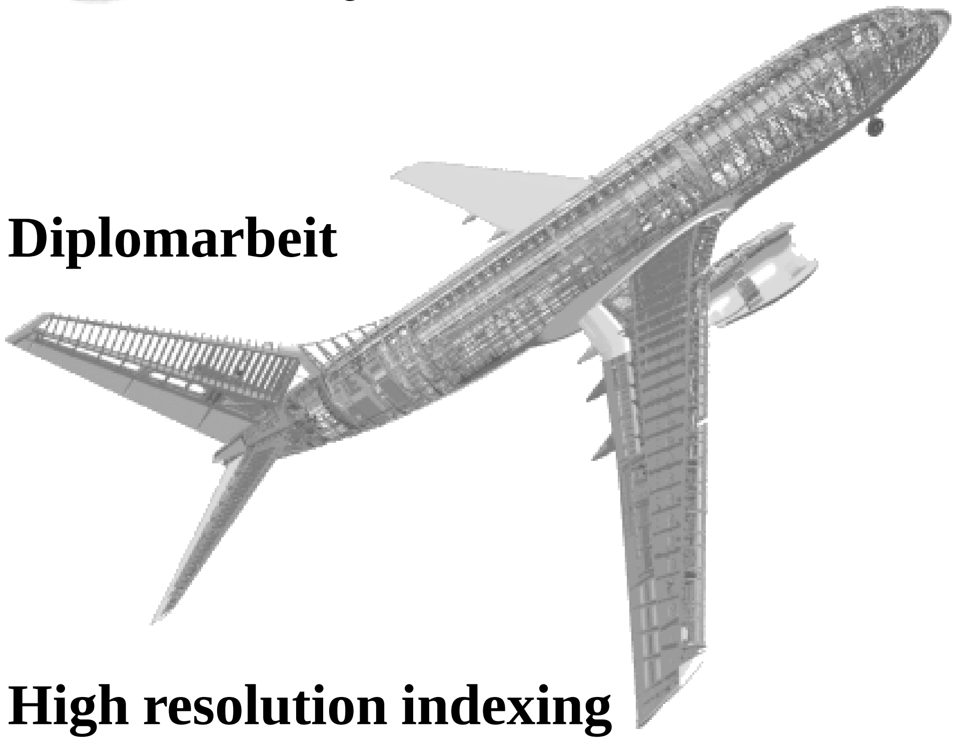
FernUniversität
Gesamthochschule in Hagen



Diplomarbeit

High resolution indexing for CAD databases

Aufgabensteller:	Prof. Dr. Ralf Hartmut Güting (Lehrgebiet Praktische Informatik IV) Prof. Dr. Hans-Peter Kriegel
Betreuer:	Dr. Marco Pötke Dr. habil. Thomas Seidl
Bearbeiter:	Martin Pfeifle
Abgabedatum:	31. Oktober 2001

Acknowledgments

I would like to thank Prof. Dr. Ralf Hartmut Güting for his alacrity to allow me to accomplish my diploma in computer science at the University of Munich. I was working there under the leadership of Prof. Dr. Hans Peter Kriegel to whom I am very grateful. Without his confidence in me, and without the productive and inspiring working atmosphere he created, this work could never have come into existence.

My diploma thesis was greatly inspired by the work of my two tutors Dr. Marco Pötke and Dr. habil. Thomas Seidl, the inventors of the RI-tree. I want to thank them for the many fruitful discussions concerning scientific questions and their readiness to help me with technical problems. I am also grateful to Bob Abarbanel, M.D., Ph.D., at the Boeing Company, who placed real-world test data at my proposal.

I want to thank all my other colleagues for their enjoyable collaboration and their friendliness.

Furthermore, I would like to express my deepest thanks to my parents, who constantly supported me. Last but not least, my special thanks go to my wife Valerie for her encouragement and unceasing love. Without her considerateness, this work could never have been accomplished.

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Hagen, den 31. Oktober 2001 ....

Martin Pfeifle

Abstract

The management of complex spatial objects in many non-standard database applications, such as computer-aided design (CAD), imposes new requirements on spatial database systems, in particular on efficient query processing. In the past two decades various index structures have been proposed to support this process. Recently, there has been an increasing awareness that it is indispensable to integrate these index structures into fully-fledged database systems.

In this master thesis a new spatial index is introduced, which supports the *intersect* predicate on a data type *TSpatialObject*. This new access method, called *X-RI-tree*, can easily be implemented on top of an object-relational database management system, exploiting its extensible indexing framework. Thus, fundamental services of underlying commercial database systems can be fully reused, including transactions, concurrency control, and recovery.

The X-RI-tree is a multi step index for *grey interval sequences*, which can be generated out of spatial objects via space filling curves. Each of these grey intervals consists of an interval hull, aggregated information of the grey interval, and a detailed attached black interval sequence. This structure is reflected in the query process, which is based on three consecutive filter steps. In a first filter step, all overlapping pairs of grey database and query intervals are determined, by means of a slightly altered *RI-tree*. In a second filter step a so called *fast grey test* is used to determine intersecting intervals without examining the attached interval sequences. In a last third filter step, an expensive *BLOB test* is carried out, scrutinizing the attached interval sequences.

Both the RI-tree and the X-RI-tree were implemented on top of an Oracle Server Release 8.1.7, using PL/SQL for the computational main memory based programming. The experimental evaluation, which was based on two real world test data sets, has pointed up that a well parameterized X-RI-tree outperforms the optimized version of the RI-tree approximately by an order of magnitude with respect to use of secondary storage, main memory session footprint, and overall query response time.

Abstract (in Deutsch)

Die Verwaltung komplexer räumlicher Objekte, wie sie beispielsweise im CAD-Bereich auftreten, stellt neue Anforderungen an räumliche Datenbanksysteme und besonders an die räumliche Anfragebearbeitung. In den letzten zwei Jahrzehnten wurden verschiedenartige Indexstrukturen zur Unterstützung solcher Anfragen entwickelt. In jüngster Zeit, rückte die Wichtigkeit der Integration dieser Indexstrukturen in vorhandene Datenbanksysteme immer mehr ins Bewusstsein.

In dieser Diplomarbeit wird eine neue Indexstruktur vorgestellt, die das *intersect*-Prädikat auf einem räumlichen Datentyp *TSpatialObject* unterstützt. Diese neue Zugriffsmethode, die wir *X-RI-Baum* nennen, kann sehr leicht in gängige objekt-relationale Datenbanksysteme integriert werden, so dass zentrale Datenbankdienste wie Transaktionen, kontrollierte Nebenläufigkeit und Wiederanlauf in Einklang mit der Benutzung des X-RI-Baum stehen.

Der X-RI-Baum ist ein mehrstufiger Index für *graue Intervallsequenzen*, die durch raumfüllende Kurven aus räumlichen Objekten gewonnen werden können. Jedes dieser grauen Intervalle besteht aus einer Intervallhülle, aggregierter Information über das graue Intervall und einer detaillierten schwarzen Intervallsequenz. Die Struktur der grauen Intervalle wird in der Anfragebearbeitung widergespiegelt, die auf drei Filterschritten basiert. In einem ersten Filterschritt werden alle Paare sich überlappender grauer Intervalle mit Hilfe des *RI-Baumes* ermittelt. Im zweiten Schritt wird ein *Schnelltest* durchgeführt, der ohne zusätzliche I/O-Zugriffe auskommt. In einem teuren dritten Filterschritt werden die schwarzen Intervallsequenzen ausgewertet.

Sowohl der RI-Baum als auch der X-RI-Baum wurden in PL/SQL implementiert und in einen Oracle Server Release 8.1.7 integriert. Die experimentelle Auswertung basierte auf zwei Echttestdatenmengen und hat aufgezeigt, dass das Sekundärspeicherplatzverhalten, der Hauptspeicherbedarf und die gesamte Anfragezeit ungefähr um eine Größenordnung beim X-RI-Baum besser sind als beim RI-Baum.

Table of Contents

Chapter 1
Introduction

1.1 Virtual Engineering

For a long time industrial practice in automotive, aerospace, and other manufacturing industries involves the creation of a number of physical product models. During the product development phase, both the product and the process are verified on the basis of *physical mock-up* (PMU). However, hardware checks cause tremendous time delays. Moreover, a late hardware verification very often leads to respectively late design changes, which are cost intensive (cf. figure 1).

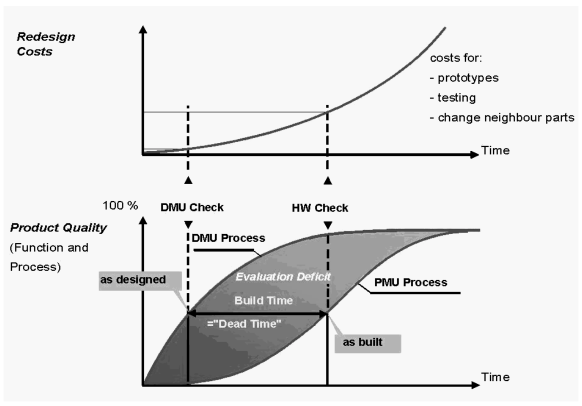


Figure 1: Digital mock-up (DMU) [IWB 01]

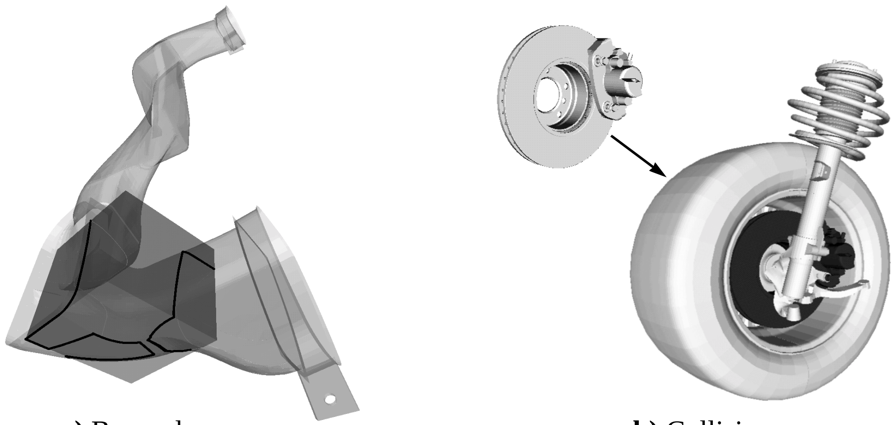
Nowadays, the different industries do not have to build real test models anymore. The real models currently required for automobile development, for instance, are being dispensed with new *digital mock-up* (DMU) methods, which combine all digital data from CAD, CAE and CAM, including the results from simulations and animations. This new approach, called virtual engineering, means that the development period in automotive construction can sharply be reduced. Shorter product cycles and a greater diversity of models are becoming decisive competitive factors in the hard-fought automobile market. These demands can only be met by digitalization of the manufacturer's whole development process and within the entire component supplier chain. The thereby incurred huge amounts of data, have to be managed somehow.

1.2 Spatial Databases

Database systems (DBS) are designed to manage and analyze huge amounts of persistent data, offering important advantages compared to a file-based organization. DBSs provide logical and physical data independence, transactions, concurrency control, integrity checking, recovery, security, standardization, and distribution [Dat 99].

One of the most promising data models for DBSs is the *object-relational* one. It provides two substantial advantages. First, the practical impact of ORDBMSs (object-relational database management systems) is very strong, as object-relational functionality has been added to most commercially available relational database servers, including Oracle [Doh 98], IBM DB2 [CCN+ 99], and Informix IDS/UDO [Bro 01]. Secondly, its extensibility is a necessary prerequisite for the seamless embedding of spatial data types and geometric predicates, which is vital for virtual engineering. Defining spatial data types and spatial predicates on top of any off-the-shelf ORDBMS enables us to ask queries like (cf. figure 2):

- Determine all spatial objects intersecting a given rectilinear box volume (*box volume query*).
- Find all objects that intersect an arbitrary query object (*collision query*).



a) Box volume query **b)** Collision query

Figure 2: Spatial queries on a Spatial Database System

Furthermore, integrating these spatial features into an ORDBMS, allows us to combine structural queries as, for instance, “retrieve all documents, that refer to the current version of the jet engine”, with the evaluation of geometric predicates. To put it another way, ORDBMSs allow us to combine easily Engineering Data Management systems (EDM) with spatial database systems.

According to Güting [Güt 94][1]spatial database systems, could be defined in the following way:

- A spatial database system is a database system.
- It offers spatial data types (SDTs) in its data model and query language.
- It supports spatial data types in its implementation, providing at least spatial indexing and efficient algorithms for spatial joins.

This definition points up, that a spatial database system, is a fully-fledged database system, with additional modules for handling spatial data. The extensibility interfaces of most ORDBMSs, including Oracle [Ora 99a] [SMS+ 00], IBM DB2 [IBM 99] [CCF+ 99], or Informix IDS/UDO [Inf 98] [BSSJ 99], enable us to integrate spatial requirements into off-the-shelf object-relational database systems.

In this work we focus on building a *spatial index*, supporting the *intersect*[2] predicate on a data type *TSpatialObject*[3]. This enables us to efficiently process collision and box volume queries, as depicted in figure 2. We do not claim, having build a new spatial database system, knowing that we fall short of presenting efficient algorithms for other spatial relationships and for spatial joins. Nevertheless, this diploma thesis can be used as a starting point and a guideline to build an efficient spatial database system on top of an ORDBMS.

[1] [Güt 94] can be used as an introductory paper into the area of spatial database systems. Furthermore, we recommend [GG 98] to the reader, in which an overview of multidimensional access methods can be found. Finally, [Sam 90a] and [Sam90 b] can serve as a starting point for the general area of spatial data structures.

[2] In [GG 98] it is mentioned that the intersection operator is one of the most important operators, and that it plays a crucial role in virtually all the other cases [GR 94].

[3] This data type is equal to a more general data type *TOIS* (*T*ype of *O*bject *I*nterval *S*equence), indicating that our approach is suitable for all domain objects, which can be represented by an interval sequence. Throughout this thesis the two data types are used interchangeable.

1.3 Spatially Extended CAD Objects

Engineering products can be regarded as a collection of individual, three-dimensional parts. Each of these parts may consist of a complex and an intricate geometric shape with a very high precision. In order to cope with the demands of accurate geometric modeling, highly specialized CAD applications are employed. In subsection 1.3.1, we talk about *triangle meshes*, a very accurate and widely spread representation form of CAD objects. In subsection 1.3.2 a coarser, conservative approximation of the parts, by means of *voxels,* will be discussed. These voxels can be grouped together to *Object Interval Sequences*[1], which build the foundation of this work.

1.3.1 Triangle meshes

Accurate representations of CAD surfaces are typically implemented by parametric bicubic surfaces, including Hermite, Bézier, and B-spline patches. For many operations, such as graphical display or the efficient computation of surface intersections, these parametric representations are too complex [MH 99]. As a solution, approximative polygon (e.g. triangle) meshes can be derived from the accurate surface representation. These triangle meshes allow for an efficient and interactive display of complex objects, for instance by means of VRML encoded files, and serve as an ideal input for the computation of spatial interference.

1.3.2 Object Interval Sequences

In order to employ spatial indexing on CAD databases, the geometry of each single CAD part can be transformed into an interval sequence by means of voxelization.

In [Pöt 01] different ways are described to achieve this voxelization, suitable for both solid and surface modeling. In this work we only consider representations of solid objects, which are already voxelized. Therefore, we do not have to take a closer look at the algorithms creating voxelized sets, but rather take their result as a starting point.

[1] In [Pöt 01], Pötke described in detail how these three different object representations are linked together and how they can be used to build a well functioning system for the Database Integration of Virtual Engineering (DIVE) for existing Engineering Data Management systems (EDM).

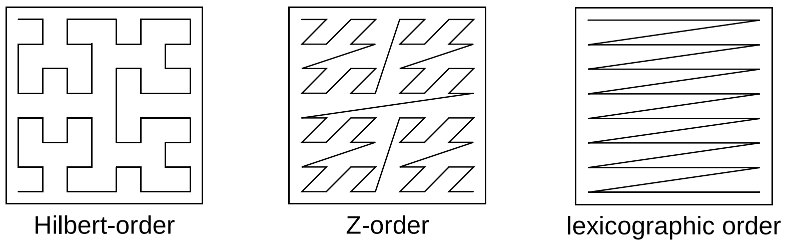


Figure 3: Examples of space-filling curves in the two-dimensional case

1.3.2.1 Mapping Extended Objects to Interval Sequences

Voxels coarsely approximate spatial objects. The voxels correspond to cells of a grid, covering the complete data space. The grid resolution determines the finest possible granularity for the approximation of the objects. By means of space filling curves, each cell of the grid can be encoded by a single integer number, and thus an extended object is represented by a set of integers. A lot of these space filling curves achieve good spatial clustering properties. Therefore, cells in close spatial proximity are encoded by similar integers or, putting it another way, contiguous integers encode cells in close spatial neighborhood. Examples for space filling curves include Hilbert-, Z-, and the lexicographic-order, depicted in figure 3. The Hilbert-order generates the last intervals per object [Jag 90] [FR 89] but unfortunately, it is the most complex linear order. Taking redundancy and complexity into consideration, the Z-order seems to be the best solution. Therefore, it will be used throughout the rest of this thesis.

Voxels can be grouped together to *Object Interval Sequences*, so that an extended object can be represented by some continuous ranges of numbers. The three shortly discussed representation forms of objects are depicted in figure 4.

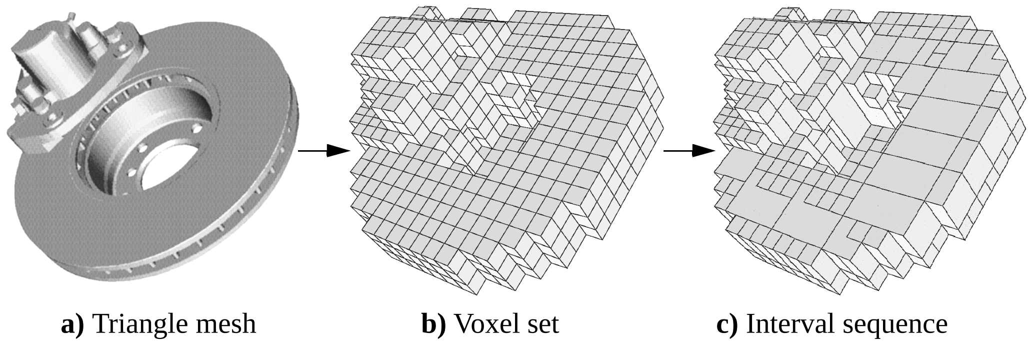


Figure 4: Conversion pipeline from triangulated surfaces to interval sequences

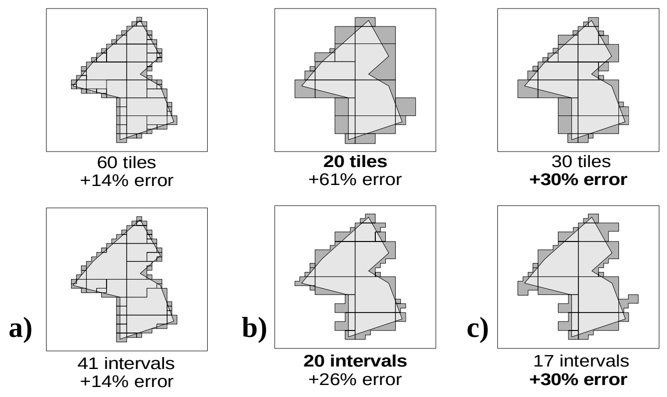


Figure 5: Decomposition of a spatial object
Top row: into Z-tiles, bottom row: into Z-ordered interval sequences;
a) granularity-bound, **b)** size-bound, and **c)** error-bound approach

1.3.2.2 Controlling Accuracy and Redundancy

The resolution of the underlying grid of the data space, i.e. the granularity, is decisive for the mapping of spatial objects to their corresponding interval sequences. On the one hand, refining the resolution approximates the object more accurate, but on the other hand, the redundancy increases. Gaede pointed out in [Gae 95] that the number of tiles and intervals exponentially depends on the granularity.

Replicating techniques based on the Linear Quadtree [TH 81] [OM 88] [Ora 99c] [RS 99] [FFS 00] decompose spatial objects into tiles which correspond to constraint segments on a space filling curve. In contrast, intervals are not confined to these tile boundaries, and therefore, yield a significantly lower redundancy, as shown in figure 5a.

Apart from the sketched granularity bound approximation [Gae 95], we need other concepts, allowing us to vary the resolution between insertion and query time as well as between different objects. Orenstein [Ore 89] introduced the *error-bound* and *size-bound* approximation approaches, embracing these problems.

These approaches are based on a recursive subdivision procedure, that stops if the desired redundancy (size-bound) or the desired maximum approximation error (error-bound) is reached. Figure 5b and 5c illustrate the size and error-bound approximation of a polygon into variable-sized tiles (top row) and into Z-ordered interval sequences (bottom row). Interval sequences yield either about half of the approximation error or half of the redundancy dependent on the desired approach compared to Quadtree tiling.

Kriegel, Pötke, and Seidl adapted in [KPS 01a] 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 (cf. figure 6). Starting with a single interval, encoding the entire data space, non-empty tiles are subdivided recursively, following the chosen space-filling curve. Depending on the desired approximation type, the recursion is terminated by a size-bound or an error-bound criterion.

```
function decompose (object, bound) → sequence of intervals;
begin
    Sequence result = ⟨ [0..$2^h$–1] ⟩;
    PriorityQueue tiles = ⟨(∞, entire_space)⟩;
    while not 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 6: Recursive decomposition of a spatial object into an interval sequence

Furthermore, there exists an alternative approach proceeding bottom-up, iteratively closing small gaps between intervals. For a size-bound approximation, this algorithm stops if the maximum redundancy has been reached. The error-bound approach minimizes the redundancy, by allowing only gaps larger than a *MAXGAP* parameter between the different intervals, grouping all other intervals together to "*grey intervals*".

1.4 Processing Spatial Queries

In this section we review some general ideas about the spatial query process as presented in the literature. In the first two subsections we point out the difference between *two*- and *multi-step query processing*. In subsection 1.4.3 we look at the topic of *complexity versus redundancy.* The new spatial access method, developed throughout this thesis, combines multi-step query processing with a balanced ratio between complexity and redundancy, which itself is essential for efficient query processing [SK 93].

1.4.1 Two-Step Query Processing

As already mentioned, we can transform spatially extended objects to object interval sequences. In the next section we will introduce an existing spatial access method, the *RI-tree*[1], enabling us to efficiently process interval intersection queries. As intervals, like minimum bounding rectangles[2], only approximate objects, a second step is needed to pinpoint whether two objects intersect or not.

Gaede describes this two-step strategy for the query process in the following way (cf. [Gae 95]):

- **Filter step:** By using a spatial access method (e.g. RI-tree, R-tree etc.) based on approximated geometries (e.g. intervals, MBRs...) one obtains a set of candidate objects. With this step one eliminates most objects that do not satisfy the query, however some false hits are usually included.

- **Refinement step:** To identify false hits in the candidate set, it is necessary to fetch the qualifying objects into main memory and perform a (computationally expensive) test on the accurate geometry.

In this thesis we only focus on the filter step. In order to refine collision and box volume queries, a fine-grained spatial interference detection of the candidate set can be implemented, as for instance done in the DIVE system (cf. [KMPS 01a], [KMPS 01b]) .

1 Patent pending [KPS 00b].

2 We could also approximate our spatial object with a minimum bounding rectangle (MBR), and use the R-tree (cf. [Gut 84]) or some variant of it as a spatial access method.

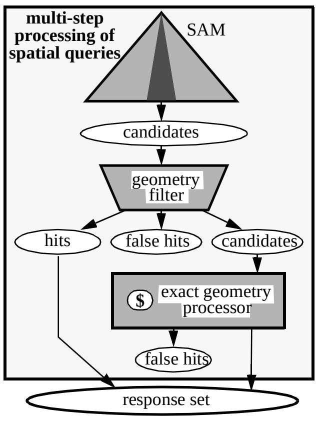


Figure 7: Multi-step query processing

1.4.2 Multi-Step Query Processing

In [BKKS 94][1] the filter step was replaced by two filter steps leading to a three step approach for the intersection problem (cf. figure 7).

- In a first filter step the minimum bounding rectangles of the objects are evaluated returning a set of candidates. This step can be efficiently supported by spatial access methods such as the R*-tree [BKSS 90].

- In the second filter step, more accurate approximations are exploited for filtering out elements (false hits) from the candidate set. Moreover, answers can already be identified using conservative as well as progressive approximations (e.g. minimum bounding 5-corners and maximum enclosed rectangles) without accessing the exact representation of the spatial objects. Two different techniques are presented in [BKKS 94]:

  The false area test: The difference between the area of an object *obj* and the area of its conservative approximation is called the *false area* (denoted by $fa_{Appr}(obj)$). For two intersecting polygonal objects obj_1 and obj_2 the following property holds:

  $$Appr(obj_1) \cap Appr(obj_2) > fa_{Appr}(obj_1) + fa_{Appr}(obj_2) \Rightarrow obj_1 \cap obj_2 \neq \emptyset$$

  To put it in words, if the area of the intersection of the approximations is larger than the sum of the false areas of the objects, it follows that the objects intersect.

[1] Jensen certifies this paper as being particularly beautiful and content-rich [Jen 99].

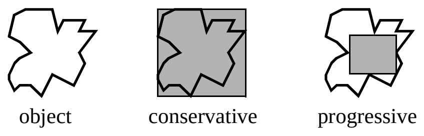


Figure 8: Conservative and progressive approximations

Progressive approximations: In addition to conservative approximations, progressive approximations (cf. figure 8) are adequate for identifying hits. A polygonal object is progressively approximated if the point set of the approximation is a subset of the point set of the object. If two progressive approximations intersect, it follows that the objects intersect.

- Eventually, in the third step, all remaining members of the candidate set are examined. This step requires access to the exact representation of the spatial objects.

In this thesis we introduce three filter steps based on object interval sequences, akin to the above steps. This yields a four step approach for query processing, i.e. three filter steps and one refinement step. Our second filter step is quite similar to the false area test, but can easily be enlarged to cope with progressive approximations as well (cf. subsection 6.1.3)[1].

We want to shortly discuss two more papers, taking advantage of an introduced second filter step. A variant of both is applied in the second filter step of our new access method.

In [ZS 98] a polygon approximation, called four-color raster signature (4CRS), is introduced. Each polygon contains $m \times n$ cells, each one having two bits of information to indicate whether a cell is empty, weak, strong or full (cf. figure 9). In the

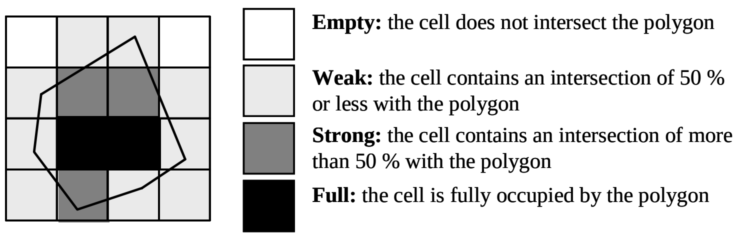

Empty: the cell does not intersect the polygon
Weak: the cell contains an intersection of 50 % or less with the polygon
Strong: the cell contains an intersection of more than 50 % with the polygon
Full: the cell is fully occupied by the polygon

Figure 9: Four-color raster signature

[1] Unfortunately, progressive approximations are very expensive to compute, especially for maximum enclosed approximations [BKKS 94].

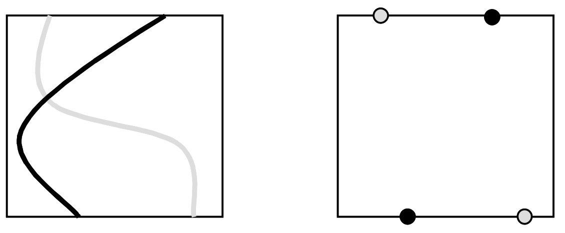

This combination of boundary points imply that the two polygons intersect

Figure 10: Polygon boundary test

second filter step the false area test is applied to each pair of superimposed cells. If they have each more than 50% of the polygon's area it is obvious that the polygons intersect each other.

In [HJR 97] it is deployed, that some configurations of boundaries imply that polygons must intersect. An example of such a situation is depicted in figure 10.

1.4.3 Complexity versus Redundancy

Most approaches for multi-step query processing are based on a first filter step, which itself uses minimum bounding rectangles to approximate the objects. This approach reveals strong disadvantages caused by the coarse approximation [SK 93]. These drawbacks are avoided by object decomposition techniques introduced in [KHS 91]. Object decomposition techniques use a set of simple components representing a complex spatial object. However, the number of components, called redundancy, results in a storage and query processing overhead. According to [SK 93] the point at issue is:

Which degree of redundancy is best suitable for efficient spatial query processing?

In the experiments, presented in chapter 4, it is shown that an answer to this question is crucial for the efficiency of our new spatial access method.

1.5 The Relational Interval Tree

The efficient management of interval data represents a core requirement for many spatial database applications[1]. The RI-tree, developed by Kriegel, Pötke, and Seidl, is a relational storage structure for interval data (*lower*, *upper*), built on top of the

SQL layer of any ORDBMS. With the Relational Interval Tree (RI-tree), an efficient access method has been proposed to process interval intersection queries. By design, it follows the concept of Edelsbrunner's main-memory interval tree [Ede 80] and guarantees the optimal complexity for storage space and for I/O operations when updating or querying large sets of intervals.

1.5.1 Relational Storage and Extensible Indexing

The RI-tree strictly follows the paradigm of relational storage structures since its implementation is purely built on procedural and declarative SQL but does not assume any lower level interfaces to the database system. In particular, built-in index structures are used as they are, and no intrusive augmentation or modification of the database kernel is required.

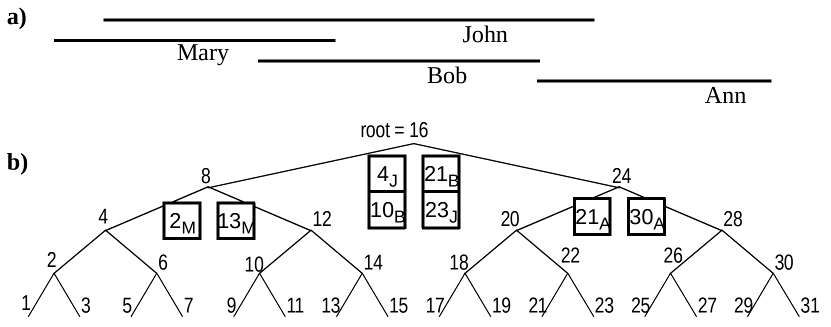
On top of its pure relational implementation, the RI-tree is ready for immediate object-relational wrapping. It fits particularly well to the extensible indexing frameworks, which were already proposed in [Sto 86]. These frameworks, which are provided by the latest object-relational database systems, enable developers to extend the set of built-in index structures by custom access methods in order to support user-defined data types and predicates without weakening the reliability of the entire system.

1.5.2 Dynamic Data Structure

The structure of an RI-tree consists of a binary tree of height h which covers the range $[1, 2^h–1]$ of potential interval bounds. It is called the virtual backbone of the RI-tree since it is not materialized but only the root value $2^{h–1}$ is stored persistently in a metadata table. Traversals of the virtual backbone are performed purely arithmetically by starting at the root value and proceeding in positive or negative steps of decreasing length $2^{h–i}$, thus reaching any desired value of the data space in $O(h)$ CPU time and without causing any I/O operation.

Upon insertion, an interval is registered at the highest node that is contained in the interval. For the relational storage of intervals, the value of that node is used as an artificial key. The resulting relational schema, called *intervals or Range* table, con-

[1] A variety of methods has been published concerning interval management in databases, most of them addressing temporal applications [TCG+ 93] [MTT 00]. In [KPS 00a] a survey of interval handling in general is given.



c)

intervals (*node*, *lower*,upper, *id*):

8, 2, 13, Mary	16, 4, 23, John	16, 10, 21, Bob	24, 21, 30, Ann

d)

lowerIndex (*node*, *lower*, *id*):

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

upperIndex (*node*, *upper*, *id*):

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

Figure 11: Example for an RI-tree
a) four sample intervals, **b)** virtual backbone and registration positions of the intervals, **c)** resulting table *intervals*, **d)** resulting relational indexes *lowerIndex* and *upperIndex*

tains the attributes (*node*, *lower*, *upper*, *id*) and is supported by two composite indexes (*node*, *lower*, *id*) and (*node*, *upper*, *id*), called *lower*- and *upperIndex*. An interval is represented by exactly one entry in the *Range* table and in each of the two indexes, and therefore, $O(n/b)$ disk blocks of size b suffice to store n intervals. For inserting or deleting intervals, the *node* values are determined arithmetically, and updating the indexes requires $O(\log_b n)$ I/O operations per interval.

The illustration in figure 11 provides an example for the RI-tree. Let us assume the intervals (2,13) for Mary, (4,23) for John, (10,21) for Bob, and (21,30) for Ann (cf. figure 11a). The virtual backbone is rooted at 16 and covers the data space from 1 to 31 (cf. figure 11b). The intervals are registered at the nodes 8, 16, and 24 which are the highest nodes hit by the intervals. The interval (2,13) for Mary is represented by one entry in the table *intervals* (8, 2, 13, Mary) and the entries (8, 2, Mary) in the *lowerIndex* and (8, 13, Mary) in the *upperIndex* since 8 is the registration node, and 2 and 13 are the lower and upper bound, respectively (cf. figure 11c and 11d).

1.5.3 Intersection 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. The procedural query preparation step descends the virtual backbone from the root node down to *lower* and to *upper*, respectively. The traversal is performed arithmetically, and the visited nodes are collected in two main-memory tables, *leftNodes* and *rightNodes*, as follows: nodes to the left of *lower* may contain intervals which overlap *lower* and are inserted into *leftNodes*. Analogously, nodes to the right of *upper* may contain intervals which overlap *upper* and are inserted into *rightNodes*. Whereas these nodes are taken from the paths, the set of all nodes between *lower* and *upper* belongs to the so-called *inner query* which is represented by a single range query on the node values. All intervals registered at nodes from the *inner query* are guaranteed to intersect the query and, therefore, will be reported without any further comparison. The query preparation step is purely based on main memory and requires no I/O operations.

In the subsequent declarative query processing step, the transient tables are joined with the *intervals* table by a single, three-fold SQL statement (cf. figure 12). The upper bound of each interval registered at nodes in *leftNodes* is compared to *lower*, and the lower bounds of intervals stemming from *rightNodes* are compared to *upper*. The *inner query* corresponds to a simple range scan over the intervals with nodes in (*lower*, *upper*). The SQL query requires $O(h \cdot \log_b n + r/b)$ I/Os to report *r* results from an RI-tree of height *h* since the output from the relational indexes is fully blocked for each join partner.

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

Figure 12: SQL statement for interval intersection queries

1.5.4 Optimizations

The naive approach of the RI-tree, produces a lot of unnecessary join partners. In this subsection we shortly introduce optimization rules for the RI-tree, which are especially useful for object interval sequences, e.g. spatial queries (for further elaborations cf. [KPS 01a] or [Pöt 01]).

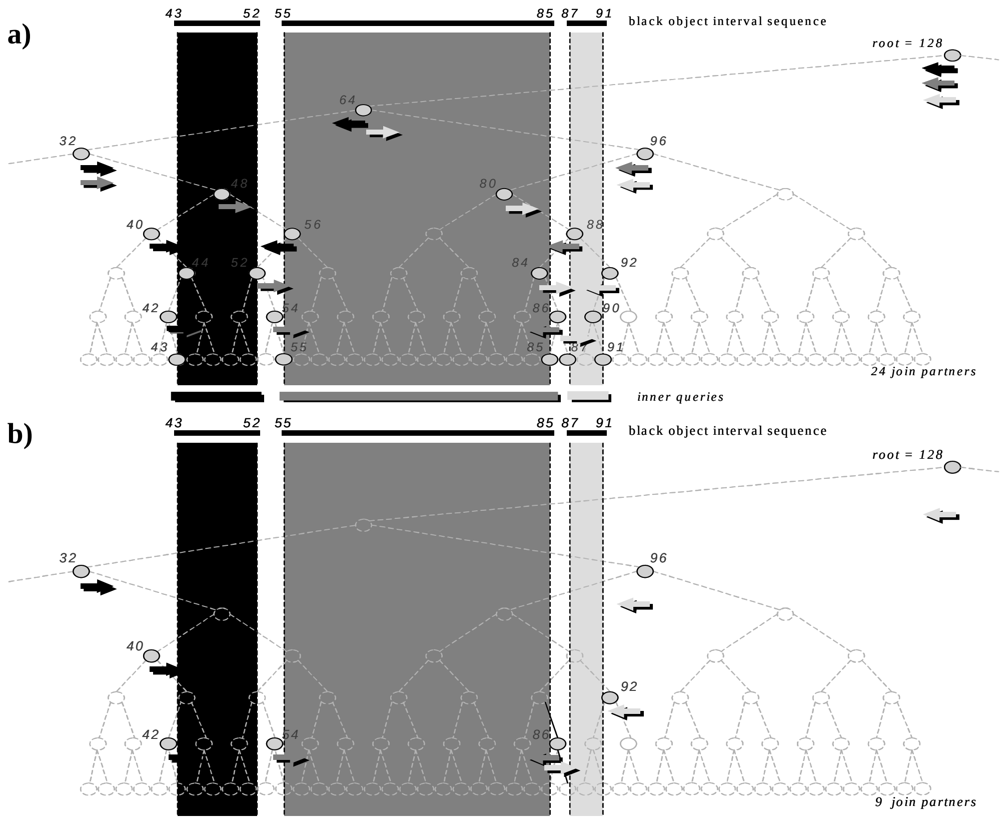


Figure 13: Optimization of the RI-tree
a) naive RI-tree, **b)** optimized RI-tree

1.5.4.1 Gap optimization

The naive approach of the RI-tree, 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. The basic idea is to avoid the generation of redundant queries, rather than to discard the respective queries after their generation.

In the example, depicted in figure 13, 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 interval suffices to report all resulting intervals from node 128, and discarding the other queries prevent the generation of duplicates without yielding false dismissals.

Node 64 is also queried three times. 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 inner query of the second interval suffices to produce the complete result. Analogously, the nodes 32, 48, 52, 56, 80, 84, 88, and 96 are queried twice and may produce duplicates.

In [KPS 01a] it is asserted and proved, that for a sorted 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 $lower_{n+1} = \infty$.

1.5.4.2 Integrating Inner Queries

As an example, consider the interval (43, 52) in figure 13 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. Generally spoken, if one interval bound is odd, the outer adjacent node is even and, thus, is reached earlier when descending the tree. The inner query may be merged with the closest corresponding left or right query. If both interval bounds are odd, the algorithm arbitrarily chooses the adjacent left node or right node. Only if both interval bounds are even, the inner query cannot be merged with an adjacent query.

The exploitation of this observation typically avoids the generation of 75% of the inner queries.

Figure 13 illustrates the effect of these two optimization rules to our example. Having originally generated 24 queries, now only 9 queries are produced.

1.5.5 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 the *upper* bound of the individual query interval. Query processing itself is performed by a single two-fold SQL statement as depicted in Figure 14.

```
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 14: Final SQL statement for interval sequence queries

1.5.6 Concluding remark

For a more detailed elaboration of the RI-tree have a closer look at [Pöt 01] or [KPS 01a]. We use this final version of the RI-tree as a starting point for a new access method, called *X-RI-tree*, which will be developed throughout this thesis. The X-RI-tree is an enhancement of the RI-tree, suitable for high resolution CAD databases. In order to understand the X-RI-tree, you have to be acquainted with the basics of the RI-tree, as introduced in this section.

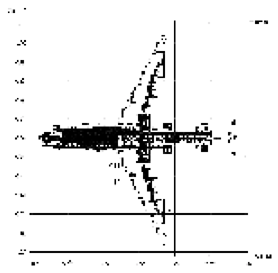
1.6 Problem Formulation

This work was mainly motivated by the need of two of our industrial partners, a German car manufacturer and an American plane producer, dealing both with high resolution CAD data. Gaede pointed out that the number of intervals, representing a spatially extended object, exponentially depends on the granularity of the grid approximation [Gae 95]. Furthermore, the extensive analysis given in [MJFS 96] and [FJM 97] shows, that the asymptotic redundancy of an interval-based decomposition is proportional to the surface of the approximated object. Thus, in the case of high resolution huge parts (e.g. wings of an airplane), the number of intervals can become very large. In order to support the process of virtual engineering, an efficient access

a)

approx. 200 parts
approx. 7 million intervals
resolution: 33 bit (0 .. 8.589.934.591)
data space: [0 m .. 6,144m]^3
voxel side length: 3 mm

b)



approx. 10.000 parts
approx. 10 million intervals
resolution: 42 bit (0 .. 4.398.046.511.103)
data space: [0 inch .. 3276,8 inch]^3
voxel side length: 0,2 inch

Figure 15: High resolution CAD test data sets
a) *CAR*, **b)** *PLANE*

method for spatial objects is needed. The best known access method, which is also suitable for a multi user environment, is the RI-tree (cf. [KPS 00a] [Pöt 01]). But unfortunately, it does not fully meet the industrial demands:

- First, processing collision and box volume queries in interactive time is no longer possible, because of the high redundancy (i.e. large amount of intervals for each object) and the high resolution (i.e. high virtual primary structure of the RI-tree).
- Secondly, a lot of secondary disk storage space is occupied, as for each interval a new entry in the *intervals* table and the corresponding *upper-* and *lowerIndexes* is spend.
- Third, the main memory session footprint may become very high because of the enormous amount of transient join partners, generated during the procedural query preparation step.

In this thesis a new access method for spatially extended objects is developed, which adopts the positive properties of the RI-tree, as for instance its easy implementation on top of any ORDBMS. Furthermore the main memory session footprint, the overall response time, and the amount of occupied secondary disk space are reduced. This new index, called *X-RI-tree*, is an enhancement of the RI-tree. The X-RI-tree supports the *intersect predicate* on a spatial data type *TOIS* (*T*ype of *O*bject *I*nterval *S*equence). Its evaluation is based on two real-world test data sets *CAR* and *PLANE* (cf. figure 15), demonstrating that this index is suitable for industrial use.

1.7 Method of Resolution

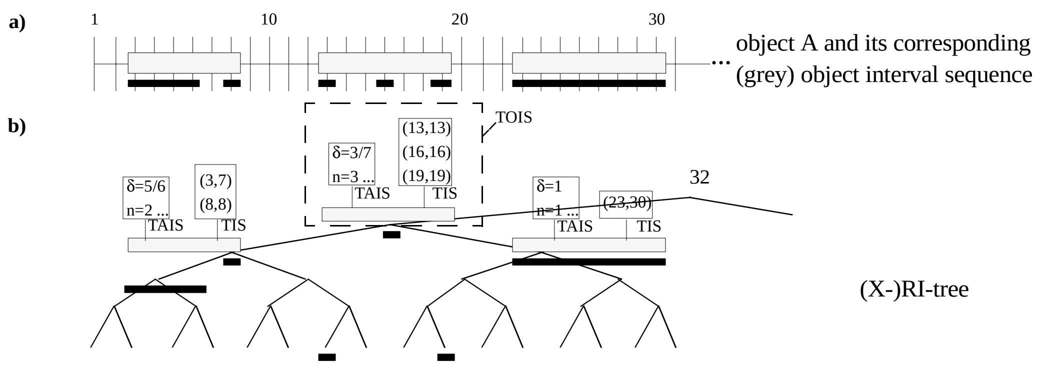
The basic idea of the X-RI-tree consists in grouping black intervals of the original RI-tree together to larger grey intervals and attach the corresponding black object interval sequence to the grey intervals. Furthermore, we enhance the *upper- and lowerIndexes* of the RI-tree with aggregated information of the grey intervals.

Figure 16 illustrates the query process of interval intersection queries. It consists of three consecutive filter steps, which are closely linked to the three parts of a grey interval.

different parts of a grey interval	**different filter steps of the query process**	**short explanation**
the hull (l_{grey}, u_{grey})	first filter step: *RI-tree*	In the first filter step all *interlacing* (i.e. intersecting) grey query and database intervals are detected by means of a slightly altered RI-tree, which evaluates the hulls of the intervals (l_{grey}, u_{grey}).
the ADT *TAIS* (Type of Aggregated Interval Sequence)	second filter step: *fast grey test*	By means of aggregated information of the grey intervals, the fast grey test tries to figure out, whether two interlacing intervals really *intersect*. It applies the "*false area test*" [BKK 94] and the "*boundary test*" [HJR 97] to grey intervals.
the ADT *TIS* *(T*ype of *I*nterval *S*equence)	third filter step: *BLOB test*	In the third filter step the attached interval sequences, stored in a *BLOB* (*B*inary *L*arge *OB*ject), are carefully investigated.

Figure 16: The different parts of a grey interval with the correspondent filter steps

Figure 17 shows how an object A is stored in the RI-tree and in the X-RI-tree. The grey interval with the hull *(13,19)* is once stored in the *intervals* table of the X-RI-tree whereas three entries in the *intervals* table of the RI-tree are necessary. Likewise, the X-RI-tree has less entries in the *upper*- and *lowerIndex*. Furthermore, the grey intervals are situated closer to the root of the tree.



c) *intervals* (node, lower, upper, id, AIS, IS): (X-RI-tree)

8,3,8,A,AIS$_1$,IS$_1$	16,13,19,A,AIS$_2$,IS$_2$	24,23,30,A,AIS$_3$,IS$_3$

intervals (node, lower, upper, id): (RI-tree)

4,3,7,A	8,8,8,A	13,13,13,A	16,16,16,A	19,19,19,A	24,23,30,A

d) *lowerIndex* (node, lower, id, AIS): (X-RI-tree)

8,3,A,AIS$_1$	16,13,A,AIS$_2$	24,23,A,AIS$_3$

lowerIndex (node, lower, id): (RI-tree)

4,3,A	8,8,A	13,13,A	16,16,A	19,19,A	24,23,A

upperIndex (node, upper, id, AIS): (X-RI-tree)

8,8,A,AIS$_1$	16,19,A,AIS$_2$	24,30,A,AIS$_3$

upperIndex (node, upper, id): (RI-tree)

4,7,A	8,8,A	13,13,A	16,16,A	19,19,A	24,30,A

Figure 17: Storing one object interval sequence in the (X-)RI-tree
a) black / grey object interval sequence, **b)** RI-tree / X-RI-tree,
c) *intervals* tables, **d)** *upper*- and *lowerIndexes*

1.8 Outline of this work

After this first introductory chapter, we will concentrate on the *X-RI-tree* in the following three chapters. The X-RI-tree is a new relational access method for object interval sequences. Chapter 2 introduces *grey intervals* and how they can be integrated into an ORDBMS. Chapter 3 discusses in great detail the process of intersection queries of grey object interval sequences, by means of the X-RI-tree. We will explain at full length the three different filter steps of the X-RI-tree and the way they are linked together. In chapter 4 the experimental results of our new relational access method are presented and compared to the RI-tree. This evaluation is based on the two test data sets, shown in figure 15. We will see that in all relevant areas the X-RI-tree outperforms the RI-tree approximately by an order of magnitude. Chapter 5 considers extensibility interfaces of ORDBMSs and shows, how the X-RI-tree can be integrated into an off-the-shelf ORDBMS, by means of these frameworks. In the last chapter we will resume our work. Furthermore, we will present a list of open problems and shortly sketch possible methods of resolution.

Chapter 2
Storage of Spatial Objects in an ORDBMS

The mapping of a spatial object by means of a space filling curve into a sequence of intervals, ends up with a huge number of very small intervals. Not only are the intervals very small, but also the gaps between them tend to be so. In order to minimize the number of intervals, which have to be stored in the RI-tree, we close these small gaps between intervals of the same (spatial) object. As we do not want to loose any information, we attach to each of these newly created *grey intervals* a black interval sequence.

The above indicated approach is introduced in this chapter. The presented technique perfectly fits into the well known error-bound approach and furthermore can easily be embedded into modern ORDBMSs by means of their extensible indexing frameworks (cf. chapter 5).

We call this new spatial indexing method *Extended Relational Interval Tree* (X-*RI-tree*), as it is based on the original *RI-tree*. Like the RI-tree, the X-RI-tree efficiently supports interval intersection queries, i.e. reporting all intervals from the database that intersect a given query interval. The X-RI-tree inherits most of the properties of its ancestor the RI-tree. Both implement the paradigm of relational access methods and exploit the availability, robustness and high performance of built-in index structures in existing systems.

In the next chapter we concentrate on the query process based on the X-RI-tree, whereas in this chapter we focus on the storage properties of our new approach. In section 2.1, we define the terms *grey* and *black intervals* as well as *attached black interval sequence*. Additionally to these definitions, in section 2.2 spatial objects and grey intervals are modelled with UML. In section 2.3 we discuss, how an interval sequence can be efficiently stored in a BLOB. In section 2.4 the model of section 2.2 is integrated in an off-the-shelf ORDBMS. We describe in detail the ADTs *TAIS* and *TIS*. Then, the structure of the necessary database tables and their corresponding indexes are presented. Section 2.5 and 2.6 might be skipped as they are not necessary for the basic understanding of the X-RI-tree, but have a rather accessory character. In section 2.5 the reader's attention is drawn to the way how *insert*, *delete* and *update statements* are handled, and in section 2.6 the term *transformed database* is introduced. This term is helpful for chapter 4, where we show by means of experiments in what way the *original databases* (RI-tree) differ from *transformed databases* (X-RI-tree).

2.1 Grey Intervals

As shown in [KPS 00a], [KPS 01a] and [Pöt 01] the RI-tree outperforms competing dynamic interval access methods by a factor of 5 for query response time and more than 40 for physical disk accesses. Furthermore, it needs only *O(n/b)* disk blocks of size *b* to store *n* intervals. Although this is the optimum analytical storage complexity, it seems rather wasteful to spend one row in the table *intervals* of the RI-tree for each short interval. As there are a lot of application problems where the corresponding objects can be modelled as interval sequences, consisting of very small intervals and small gaps, it seems worth investigating, whether it is not better to group small intervals, together to longer *grey intervals*. We can find such applications in a lot of different areas. They may occur as transaction time and valid time ranges in temporal databases [SOL 94] [Ram 97] [BÖ 98] or as line segments on a space-filling curve in spatial applications [FR 89] [BKK 99].

All the definitions and other basics necessary for the understanding of the X-RI-tree are introduced in this section. We start with the definition of a *black interval* and a *black interval sequence*.

Definition 1 (*black interval*)
Let $B \subseteq IN$ be a domain of *boundary points*. The closed interval $I_{black} = (l, u) \in B^2$ is called a *black interval*, iff $l \le u$. It represents all elements $x \in B$, where $l \le x \le u$. The number of represented elements $x \in B$ is called the length of the interval L_{black}, i.e. $L_{black} = u-l+1$. The values l and u are the *lower* and *upper bound* of I_{black}, respectively. The interval I_{black} is called *degenerated* or *point*, iff $l = u$.

Although it is possible to define the boundary points of the *black intervals* as real numbers [Pöt 01], we confine them to natural numbers as this answers our purpose. The definition of a *black interval* is straightforward and self-explaining. Note, the length of a point, according to our definition, equals one.

Definition 2 (*black interval sequence*)
Let $D = \{(l, u) \in B^2 | l \le u\}$ be a domain of *black intervals* with boundaries out of $B \subseteq IN$. A sequence $S_{black} = <s_1, \ldots, s_n>$ of *black intervals* $s_i \in D$ is called a *black interval sequence* with *cardinality n*, iff the following condition holds:

$$\forall i \in \{1, \ldots, n-1\}.(u_i + 1 < l_{i+1}).$$

Note that we define a *black*[1] *interval sequence* as an ordered set of intervals, where there exists at least one natural number m between two intervals (l_i,u_i) and (l_{i+1},u_{i+1}), with $u_i < m < l_{i+1}$. This is done because otherwise the two intervals could be joined to a longer one.

A *black interval sequence* naturally corresponds to *one* object. Although this is not included in definition 2, we silently assume it. Thus we could also speak of a *black object interval sequence.* Note that we allow one object consisting of several *black interval sequences*, but we do not allow, that a *black interval sequence* represents several application domain objects.

Definition 3 (*grey interval with an attached black interval sequence*)
Let $D = \{(l, u) \in B^2 | l \le u\}$ be a domain of *black intervals* with boundaries out of $B \subseteq IN$. Let $S_{black} = <s_1, \ldots, s_n>$ be a *black interval sequence* with $s_i \in D$, $s_1=(l_1,u_1)$ and $s_n=(l_n,u_n)$. Then, the tuple $I_{grey} = ((l_{grey}, u_{grey}), S_{black})$ is called a *grey interval with the attached black interval sequence* S_{black}, iff $l_{grey} = l_1$ and $u_{grey} = u_n$. The values l_{grey} and u_{grey} are the *lower* and *upper bound* of I_{grey} respectively.

[1] If it is clear from the context, we omit the word black.

$L_{grey} = u_{grey} - l_{grey}+1$ is called the length of the grey interval. Furthermore, we call $N_{Black} = \sum_{i=1}^{n} L_i > 0$ and $N_{white} = L_{grey} - \sum_{i=1}^{n} L_i \geq 0$ the number of *black* and *white cells* of the grey interval.

In a grey interval, not only black intervals are included but also the gaps between them. If we are only interested in the upper and lower bounds of I_{grey}, but not in the exact structure of S_{black}, we also write a little bit sloppy $I_{grey} = (l_{grey}, u_{grey})$. If the cardinality of S_{black} is 1, N_{white} is equal to 0. In this case $I_{grey} = ((l_{grey}, u_{grey}), <s_1>)$ is also called a black interval.

Note that at the beginning and at the end of a grey interval gaps are not allowed. Similar to Definition 2 we could easily define *grey interval sequences* and according to the remark following this definition, we could also speak of *grey object interval sequences*.

We are now defining the average density of a grey interval d_{grey}. It is defined as the ratio of the sum of the lengths of all black intervals attached to I_{grey} to the length of I_{grey}.

Definition 4 (*density of a grey interval*)
Let $S_{black} = <s_1, \ldots, s_n>$ be a *black interval sequence* with cardinality *n*. Let $I_{grey} = ((l_1,u_n), S_{black})$ be the corresponding grey interval. Then the density of a grey interval d_{grey} is defined as follows:

$$d_{grey} = \frac{\sum_{i=1\ldots n} L_i}{L_{grey}} = \frac{N_{Black}}{N_{Black} + N_{white}}$$

Correspondingly, to the *error-bound-approach* we now define a maximum gap parameter of a grey interval, called *MAXGAP*. This parameter assures that the maximum gap between two black intervals of I_{grey} is smaller or equal to the value of *MAXGAP*.

Definition 5 (maximum gap parameter)
Let $S_{black} = <s_1, \ldots, s_n>$ be a *black interval sequence* with *cardinality* $n > 1$. Let $I_{grey} = ((l_1,u_n), S_{black})$ be the corresponding grey interval. Then the maximum gap parameter *MAXGAP* of a grey interval is defined as follows:

$$MAXGAP = max\{l_{i+1} - u_i - 1 \mid (i \in 1\ldots n-1)\}$$

If $n = 1$, then *0* is assigned to *MAXGAP*.

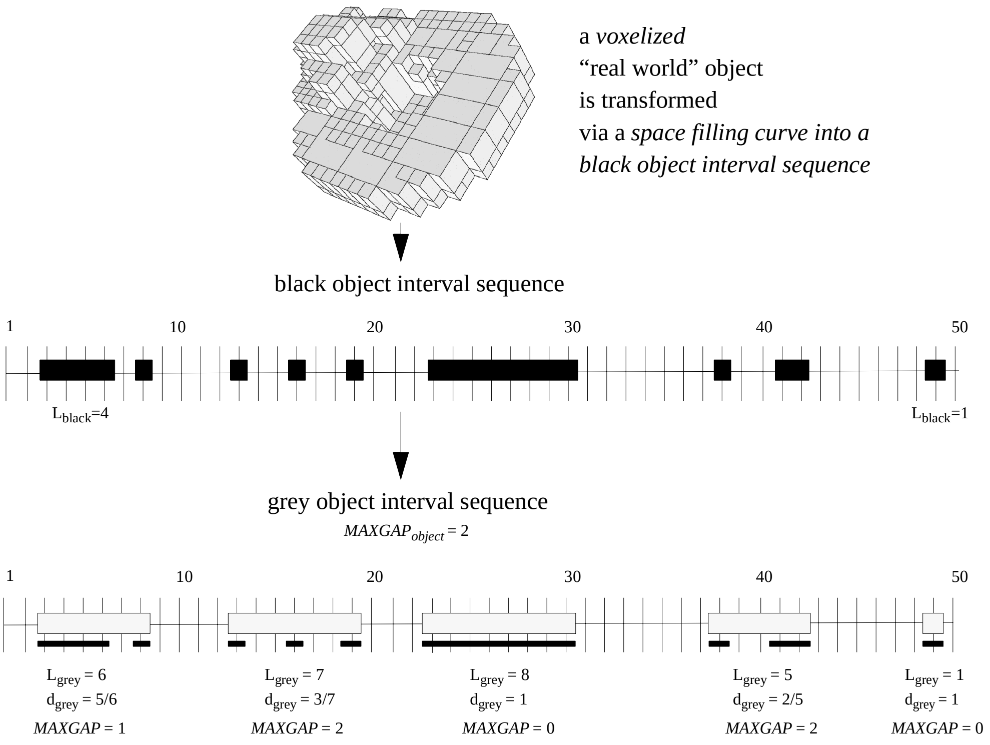


Figure 18: Grey object interval sequence

We also speak of a maximum gap parameter $MAXGAP_{object}$ of a (spatial) object, meaning that all the maximum gap parameters $MAXGAP_{(interval)}$ of the grey intervals, belonging to this object are smaller or equal to $MAXGAP_{object}$. Similarly, we speak of a $MAXGAP_{database}$ parameter, meaning that all the maximum gap parameters $MAXGAP_{objects}$ of all objects in the database are smaller or equal to *the* $MAXGAP_{database}$ parameter. If it is clear from the context, we omit the indices and use only the term *MAXGAP* parameter, or even shorter only *M*. In Figure 18 all the defined terms and their connections to each other are depicted for clarification.

We can assess d_{grey} by means of the following theorem.

Theorem 1 (*density of a grey interval*)
The density d_{grey} of a grey interval I_{grey} with a defined maximum gap parameter M can be estimated in the following way:

$$\frac{1}{1+M} \leq d_{grey} \leq 1$$

Proof. According to definition 4, d_{grey} is smaller or equal to one. On the other hand, the minimum density is achieved if the *n* intervals of the attached black interval sequence are points and all gaps between them are of length *M*.
In this case, $d_{\text{grey}} = n/(n+M\cdot(n-1)) = 1/(1+M\cdot(n-1)/n) \geq 1/(1+M)$ holds as well.

2.2 Modelling Grey Intervals

In this section the grey object interval sequences, defined in the last section, are modelled with UML.

As depicted in figure19, a spatial object consists of an ordered collection of grey intervals. The grey intervals themselves are made up of three different objects and offer two methods for testing intersection and interlacing, which are scrutinized in

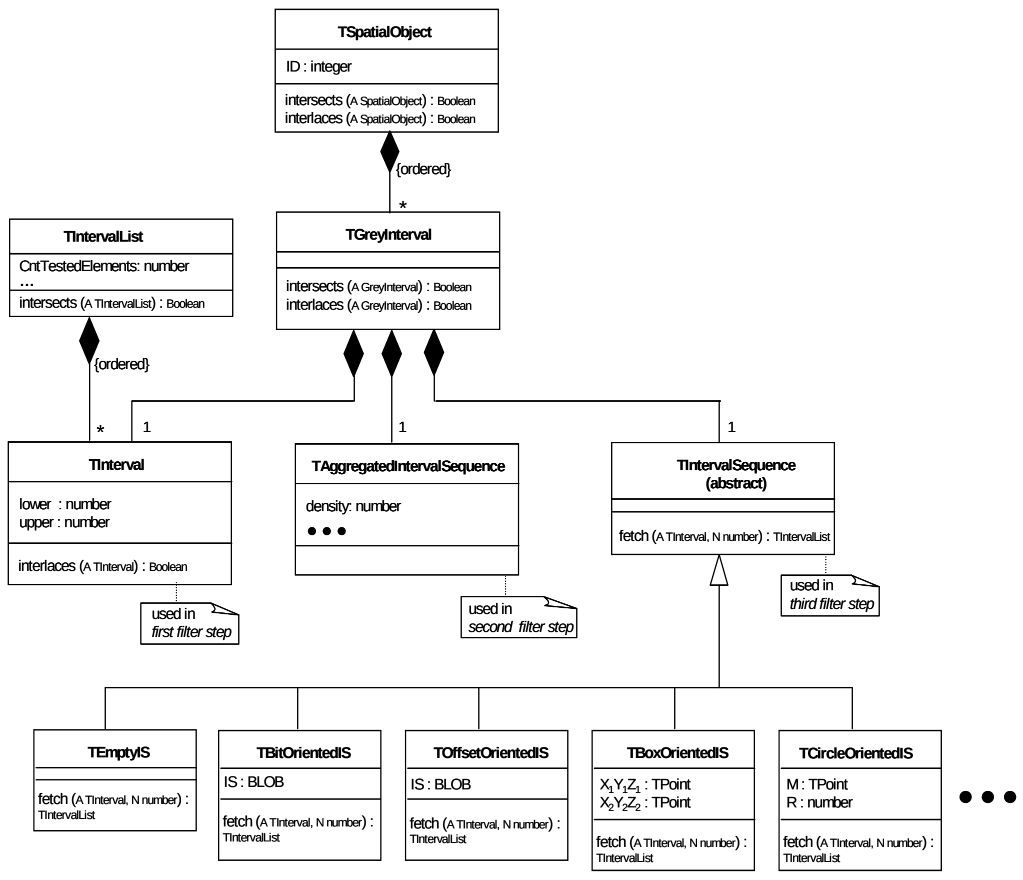

TSpatialObject
ID : integer
intersects (A SpatialObject) : Boolean
interlaces (A SpatialObject) : Boolean
{ordered}
*
TIntervalList
CntTestedElements: number
...
intersects (A TIntervalList) : Boolean
TGreyInterval
intersects (A GreyInterval) : Boolean
interlaces (A GreyInterval) : Boolean
{ordered}
*
1
1
1
TInterval
lower : number
upper : number
interlaces (A TInterval) : Boolean
used in first filter step
TAggregatedIntervalSequence
density: number
• • •
used in second filter step
TIntervalSequence (abstract)
fetch (A TInterval, N number) : TIntervalList
used in third filter step
TEmptyIS
fetch (A TInterval, N number) : TIntervalList
TBitOrientedIS
IS : BLOB
fetch (A TInterval, N number) : TIntervalList
TOffsetOrientedIS
IS : BLOB
fetch (A TInterval, N number) : TIntervalList
TBoxOrientedIS
X1Y1Z1 : TPoint
X2Y2Z2 : TPoint
fetch (A TInterval, N number) : TIntervalList
TCircleOrientedIS
M : TPoint
R : number
fetch (A TInterval, N number) : TIntervalList
• • •

Figure 19: Modelling of a Spatial Object

chapter 3. Each of the three objects of a grey interval is linked directly to one of the three filter steps in the query process (cf. figure 19).

- The ADT *TInterval* is the hull of the grey interval, used in the first filter step.
- The ADT *TAggregatedIntervalSequence* (*TAIS*) contains aggregated information of the attached interval sequence. It has to comprise the *density* of the attached interval sequence. Other information like the *cardinality*, the actual $MAXGAP_{interval}$ value, the coordinates of the bounding box of the grey interval, etc. might be useful, but can be omitted. The ADT *TAIS* could easily be enlarged by such information and may thereby improve the effectiveness of the second filter step.
- The ADT *TIntervalSequence* (*TIS*) provides only a method *fetch (A TInterval, N number): TIntervalList*, purveying an ordered list of black intervals. This list contains maximum *N* elements, starting with the black interval closest to *A.lower*. If less than *N* black intervals are included in the range (*A.lower, A.upper*) the cardinality of the list is smaller than *N*, otherwise it is equal to N. There are no additional attributes assigned to this abstract data type, because different implementations need different attributes.

  We can distinguish three main kinds of implementations.

  First, if the grey interval is actually a black interval no additional attributes have to be stored (cf. class *TEmptyIS*).

  In the second case, represented by the classes *TBitOrientedIS* and *TOffsetOrientedIS*, the attached black interval sequence is stored in a BLOB. This approach is useful for intricate spatial objects, and is discussed in detail in the next section.

  In case of dynamically created query objects or simple database objects, like boxes (*TBoxOrientedIS*) or circles (*TCircleOrientedIS*), we can use another approach, which does not materialize the attached interval sequence but rather creates it on demand during the fetch-operations. In order to do that, only little information, like box coordinates or the central point plus radius are necessary. In this work, we pursue this idea only in case of dynamically created box volume queries (cf. section 3.7), but as already mentioned it would be useful for simple database objects as well.

2.3 Storing an Interval Sequence in a BLOB

In this section, we discuss how the attached interval sequence of a grey interval can be efficiently materialized and stored in a BLOB. We introduce two different approaches the bit-oriented and the offset-oriented one, which are applied in the two classes *TOffsetOrientedIS* and *TBitOrientedIS* (cf. figure 19). Furthermore, it is shown in which cases each of them uses less secondary disk space, leading us to central theorems for the efficient storage of attached black interval sequences.

2.3.1 Introduction

As it is the main task of the RI-tree to store black intervals and to efficiently support their access, it is the main task of the X-RI-tree to do the same with grey intervals, while using a minimum of secondary storage space (cf. design goals of section 1.6). The RI-tree spends for each (small) interval of a black *object interval sequence* one record in the *intervals*[1] table. Additionally, two index entries in the *lower*- and *upperIndex* are created for each interval.

Grouping *n* black intervals together to a grey interval saves *(n-1)* entries in the *intervals* table plus *2·(n-1)* entries in the indexes. But what we have to do additionally, is storing the attached interval sequence of the grey interval.

As already mentioned *O(n/b)* is the optimum complexity class for storing *n* intervals. Furthermore, there is no redundancy included in a *black object interval sequence,* so that we have to store all the information included in this sequence in order to deliver error-free results, with respect to the voxel representation (cf. section 1.3).

We now present an approach of organizing the attached interval sequence, that saves disk space without loosing any information.

2.3.2 Bit-Oriented Approach

A very important observation is, that an *object interval sequence* consists of a large number of very short intervals (e.g. points) which are connected by short gaps.

[1] In the case of the RI-tree, this table is also called *Range* table and in the case of the X-RI-tree it is called *XRange* table.

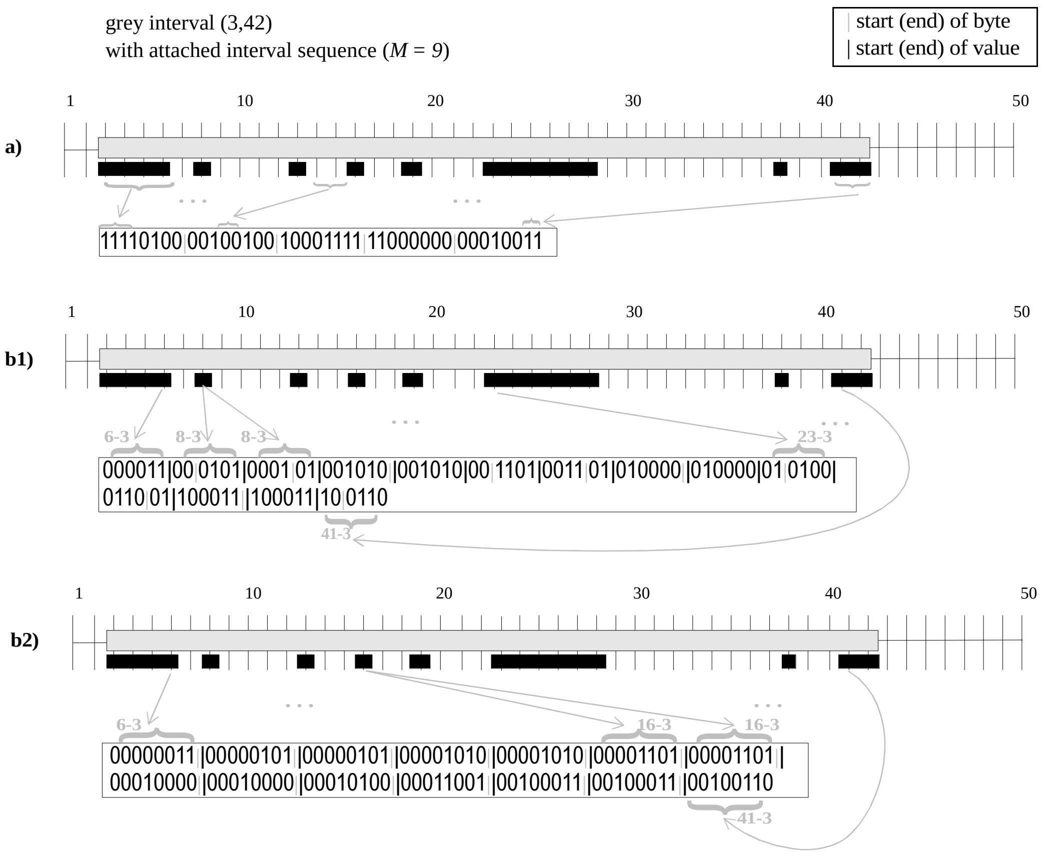


Figure 20: Storing the attached interval sequence in a BLOB
a) bit-oriented, **b)** offset-oriented (bit (**1**) and byte (**2**))

Experiments suggest that both the number of intervals and the number of gaps of a specific length x are exponentially distributed (cf. section 4.3).

This motivates the bit-oriented approach illustrated in figure 20a. We represent each voxel of the data space, which is covered by a grey interval by one bit in its BLOB. Obviously this approach works well for short intervals with short gaps. Instead of storing e.g. 5 black intervals (10,10) (12,12) (14,14) (16,16) (18,18) of one object, we store only the hull (10,18) and the sequence (101010101) in the attached BLOB. On the other hand, this approach is extremely bad, if the grey intervals include very long intervals or gaps. For example, grouping the two intervals (10,200) and (400,450) together, results in the hull (10,450) and in an attached BLOB, in which 441 bits are stored. Thus the size of the attached BLOB is always equal to $(u_{grey} - l_{grey}+1)$ bits, i.e. $O(L_{grey})$.

2.3.3 Offset-Oriented Approach

In the offset-oriented approach (cf. figure 20b) we process the boundary values of the black intervals belonging to an attached black interval sequence. If the attached interval sequence S_{black} of a grey interval I_{grey} is of cardinality *n*, we store the values u_1-l_{grey}, l_2-l_{grey}, u_2-l_{grey}, ..., $l_{(n-1)}$-l_{grey}, $u_{(n-1)}$-l_{grey}, l_n-l_{grey} sequentially in a BLOB. Each of these $1+2\cdot(n-2)\ +1\ =\ 2\cdot(n-1)$ values is smaller than L_{grey}. Thus only $\lceil \log_2 L_{grey} \rceil$ bits are needed for storing one value ($\lceil \log_2(42-3) \rceil = 6$ bits in the example of figure 20).

In the offset-oriented approach the size of the BLOB does not depend linearly *on* L_{grey} but logarithmic and additionally it depends linearly on the cardinality *n* of the attached interval sequence, i.e. we have a space complexity of $O(n \times \log_2 L_{grey})$.

The *offset-oriented-bit* and the *offset-oriented-byte* approach differ in the fact that the *offset-oriented-bit* approach only uses the calculated number $\lceil \log_2 L_{grey} \rceil$ of bits for expressing one boundary value, whereas the *offset-oriented-byte* approach always uses full bytes, i.e. $8 \times \lceil (\log_2 L_{grey}) \div 8 \rceil$ bits. Thus the *offset-oriented-byte* approach allows a more comfortable access to the different values in the BLOB.

A final remark: instead of using the *offset-oriented approach*, we could have organized the attached interval sequence by means of run-length-coding. The result is in the same storage space complexity class but the process of accessing the BLOB at a desired offset is only possible by scanning the BLOB from the beginning, whereas our approach allows bipartitioning the BLOB and so a logarithmic access time is guaranteed (cf. chapter 3).

2.3.4 Discussion

In this subsection, we first discuss the differences between the two offset-oriented approaches and then compare both to the bit-oriented approach.

Theorem 2 (offset-oriented approach (storage))
Let $S_{black} = \langle s_1, \ldots, s_n \rangle$ be a black interval sequence of cardinality *n*, which belongs to a grey interval $I_{grey} = ((l_{grey}\ ,\ u_{grey})\ ,\ S_{black})$ of length L_{grey}. Then, the *offset-oriented-bit* approach always needs less or equal secondary disk space compared to the *offset-oriented-byte approach* for storing S_{black}.

Proof. As outlined in subsection 2.3.3, the *offset-oriented* approaches need $2 \times (n-1) \times X$ bits for storing S_{black} where X denotes the number of bits needed for storing one boundary value. In the case of the *offset-oriented-byte* approach X equals $8 \times \lceil (\log_2 L_{grey}) \div 8 \rceil$ and in the case of the *offset-oriented-bit* approach X equals $\lceil \log_2 L_{grey} \rceil$.

Let $(\log_2 L_{grey}) \div 8$ be equal to $n+q$ with $n \in IN$, $0 \le q < 1$.

As $\lceil \log_2 L_{grey} \rceil = \lceil 8(n+q) \rceil = 8n + \lceil 8q \rceil \le 8n + 8\lceil q \rceil = 8\lceil n+q \rceil = 8\lceil (\log_2 L_{grey}) \div 8 \rceil$ holds, the theorem is true.

Obviously it is sometimes better to use the *bit-oriented approach* which needs *O(L_{grey})* bits and sometimes the *offset-oriented* one is better, needing $O(n \times \log_2 L_{grey})$ bits. Fortunately, it can be decided which one is preferable for each grey interval, dependent on the length of the grey interval and the cardinality of the attached interval sequence.

Theorem 3 (bit-oriented approach versus offset-oriented approach (storage))
Let $S_{black} = \langle s_1, \dots, s_n \rangle$ be a black interval sequence of cardinality n, which belongs to a grey interval $I_{grey} = ((l_{grey}, u_{grey}), S_{black})$ of length L_{grey}. Let X be equal to $8 \times \lceil (\log_2 L_{grey}) \div 8 \rceil$ in the *offset-oriented-byte* approach and equal to $\lceil \log_2 L_{grey} \rceil$ in the *offset-oriented-bit* approach. Then, the *bit-oriented* approach needs less secondary disk space compared to the *offset-oriented approaches* for storing S_{black} if the following formula holds:

$$L_{grey} < 2 \times (n-1) \times (X)$$

Proof. The *bit -oriented* approach needs L_{grey} bits for storing the attached interval sequence S_{black} in a BLOB. As outlined in the theorem above X bits are needed for storing one interval boundary in the case of the offset-oriented approaches. As we have to store $1 + 2 \cdot (n-2) + 1 = 2 \cdot (n-1)$ of these boundary values, the theorem holds.

Based on this last theorem, we can decide for each grey interval whether the *offset-* or the *bit-oriented* approach needs less secondary storage. We call this the *final approach*. In section 4.3 the experimental results of the *bit-oriented*, *offset-oriented* and the *final-approach* are compared to each other and to the RI-tree.

2.4 Implementation of Grey Intervals

As in a lot of CAD-databases the data space is fixed, we are assuming a constant data space, which is known in advance. Nevertheless, all methods which are introduced in [Pöt 01] for handling dynamic data spaces can be applied to the X-RI-tree as well.

The RI-tree is an efficient implementation of Edelsbrunner's interval tree [Ede 80], [PS 93] on top of any relational database system. In this section it is shown how the model of grey intervals (cf. section 2.2) can be integrated into an ORDBMS, thus meeting one of the design goals of section 1.6. In the first two subsections, we discuss the implementation of the abstract data types *TIS* and *TAIS*. Finally, we introduce the database schema, which uses these two abstract data types.

2.4.1 The Abstract Data Type *TIS*

As polymorphism is an inherent part of most ORDBMSs, we can implement the abstract data type *TIS* straightforward (cf. figure 21). After having defined the abstract supertype *TIS*, the specific subtypes can be derived.

```
CREATE TYPE TIS AS OBJECT
( NOT INSTANTIABLE FUNCTION
  Fetch (Area TInterval, N number) RETURN TIntervalList;
) NOT INSTANTIABLE NOT FINAL;
```

```
CREATE TYPE TEmptyIS  UNDER TIS
( FUNCTION Fetch (Area TInterval, N number) RETURN TIntervalList );

CREATE TYPE TBitOrientedIS  UNDER TIS
( IS BLOB,
  FUNCTION Fetch (Area TInterval, N number) RETURN TIntervalList );

CREATE TYPE TOffsetOrientedIS  UNDER TIS
( IS BLOB,
  FUNCTION Fetch (Area TInterval, N number) RETURN TIntervalList );

CREATE TYPE TBoxOrientedIS  UNDER TIS
( x1y1z1 TPoint,
  x2y2z2 TPoint,
  FUNCTION Fetch (Area TInterval, N number) RETURN TIntervalList );

•••
```

Figure 21: Implementation of the ADT *TIS*

2.4.2 The Abstract Data Type *TAIS*

The ADT *TAIS* is kept rather simple. It consists only of one attribute *Density* and no additional methods. As already mentioned, it could be enlarged by other values, if required.

```
CREATE TYPE TAIS AS OBJECT(
   // attributes
   Density           number
   // if necessary further attributes like MAXGAP_interval, cardinality etc. could be included as well
 );
```

Figure 22: Implementation oriented ADT *TAIS*

2.4.3 Database Schema

We can now include the two abstract data types *TAIS* and *TIS* into the table *intervals*. Figure 23 compares the necessary table and index creation statements for the X-RI-tree and the RI-tree to each other. In the case of the X-RI-tree, the table *intervals* is augmented by the two ADTs *TIS* and *TAIS*.

The ADT *TAIS* is also included into the *upper*- and *lowerIndex* so that until the third filter step, we do not have to access the table *intervals* at all, but can confine ourselves to the corresponding indexes. For the same reason both boundary values of the grey interval, and the attribute *ID* are incorporated into the indexes[1].

```
RI-tree
CREATE TABLE intervals (node int, lower int, upper int, id int);
CREATE INDEX lowerIndex ON intervals (node, lower, id);
CREATE INDEX upperIndex ON intervals (node, upper, id);

X-RI-tree
CREATE TABLE intervals (node int, lower int, upper int, id int, AIS TAIS, IS TIS);
CREATE INDEX lowerIndex ON intervals (node, lower, upper, id, AIS);
CREATE INDEX upperIndex ON intervals (node, upper, lower, id, AIS);
```

Figure 23: SQL statements to instantiate an (X-)RI-tree with secondary indexes

[1] We resigned from introducing the *TInterval*-object for the sake of comparability to the RI-tree.

2.5 Insert-, Delete- and Update-Statements

2.5.1 Determination of the Fork Node

As already mentioned the primary structure of the X-RI-tree is managed purely virtually. Let $[1, 2^{h}–1]$ be the entire range of the data space. In this case, the root node is set to $2^{h–1}$ [KPS 01a] . Let (l, u) be an interval, which has to be inserted. First, we have to determine the fork node of the interval. The fork node is the topmost node w for which $l \le w \le u$ holds (cf. figure 24). The computation of the fork node can be done by recursive traversing the virtual backbone via bisection. No I/O access is necessary, but only simple integer arithmetic, i.e. bit-shift-operations.

2.5.2 Insert and Delete

The X-RI-tree has basically the same behavior as the RI-tree with respect to the DML-statements *insert* and *delete*. The I/O complexity for the search in the index (B^+-directory) is $O(log_b n)$, where n denotes the number of (grey) intervals stored in the database.

Keep in mind, that inserting or deleting one interval in the X-RI-tree is equivalent with inserting or deleting several intervals in the RI-tree. For example, if you group in average M black intervals together to one grey interval of length L_{grey}, you get an I/O complexity of $O(Mlog_b n+M)$ for inserting these M intervals in the RI-tree, whereas you get a complexity of $O(log_b(n/M)+(min(L_{grey} , 2(M-1)log_2 L_{grey})/b))$ in the X-RI-Tree. The first factor in this sum is due to the fact that you have to search once in the B^+-tree, the second factor describes the number of disk accesses necessary to store the M intervals in the BLOB. Not only do you always need less space to

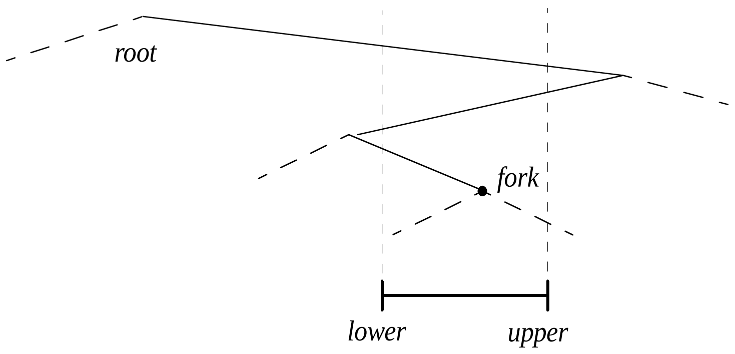


Figure 24: Fork node of an interval in the virtual backbone

store an attached interval sequence of cardinality M in the X-RI-tree compared to storing these M intervals in the RI-tree (cf. subsection 4.3.3), but you also need less disk accesses for index lookups.

2.5.3 Update

The X-RI-tree has basically the same behavior as the RI-tree with respect to the update-statement. The I/O complexity for the search in the index (B^+-tree) is $O\ (log_b n)$, where n denotes the number of (grey) intervals stored in the database.

Note that updating one grey interval in the X-RI-tree, can be equivalent to *i* insert, *j* delete and *k* update statements in the RI-tree. Thus it is rather difficult to compare the update operations in both trees to each other. Nonetheless, you can assert, that the number of I/0 accesses for searching in the B^+-tree, which is the underlying operation of all the DML-statements discussed in this section, is smaller in the case of the X-RI-tree. This is due to the fact, that less intervals are stored in the X-RI-tree. Therefore the height of the corresponding B^+-directory can at least be as high as the one of the RI-tree (cf. section 4.3).

2.6 Transformed Database

2.6.1 Transformation Function

In the previous sections we have introduced the X-RI-tree. We have seen, that an instance of it can be described like an instance of the RI-tree by a five tuple (*intervals*, *lowerIndex*, *upperIndex*, *root*, $MAXGAP_{database}$). We will call an instance of the RI-tree an *original database* D_{ori}. Grouping black intervals of an *original database* D_{ori} together to grey intervals, leads to a *transformed database* $D_{trans} = f\ (D_{ori})$. The *function f* (i.e. algorithm) is called the *transformation function*[1].

A few obvious remarks concerning $D_{trans} = f\ (\ D_{ori}\ )$:

- Of course, only intervals of the same object (same *id* value in the table *intervals*) can be grouped together.
- The number of records for each object stored in the *intervals* table is smaller in D_{trans} than in D_{ori}.

[1] We use $D_{ori(trans)}$ interchangeable for an instance of a database as well as for the domain of the *function f*.

- We use the X-RI-tree for storing D_{trans} and the RI-tree for storing D_{ori}.
- There exist a lot of *transformation functions*, some of them are introduced in figure 25.

F_{MAXGAP} is the function, we implicitly applied up to now and the only one we want to investigate in more detail in this thesis. Nevertheless, the other functions might be useful, especially because some of them seamlessly fit into the *size-bound* approach [Ore 89]. As already mentioned F_{MAXGAP} can be regarded as a generalization of the *error-bound* approach.

Function	**Explanation**
F_{ident}: $D_{ori} \to D_{trans}$	The two database D_{ori} and D_{trans} are identical. Nevertheless, D_{ori} is connected to the RI-tree and D_{trans} to the X-RI-tree.
F_{N1Step}: $D_{ori} \times N \to D_{trans}$	We allow only N (grey) intervals for each object. This is a kind of s*ize-bound approach* on the *first* filter step. Of course the maximum number N of intervals is not enough for a deterministic behavior of *f*. Additionally we ask for a maximum average density of each *grey object interval sequence.*
F_{N2Step}: $D_{ori} \times N \to D_{trans}$	We allow only grey intervals with a maximum cardinality of N. This is a kind of *size-bound approach* on the *second* filter step. Additionally we ask for a maximum average density of each *grey object interval sequence.*
F_{MAXGAP}: $D_{ori} \times M \to D_{trans}$	We allow only grey intervals with a maximum gap of M between two black intervals of the attached black interval sequence. Furthermore, we demand that the grey intervals start and end with a black interval. Additionaly, the cardinality of an attached black interval-sequence is maximum with respect to the above mentioned restriction. Note that if M is *zero* this function is equal to F_{ident}.

Figure 25: Transformation functions

2.6.2 Characteristica of a Database

The term *characteristica of a database* denotes all the properties of an original or transformed database, including the distribution, length, density and number of all (grey) intervals and their corresponding fork nodes.

This informal description can hardly be investigated, as it is not measurable and includes a lot of different aspects. In order to receive a quantitative measure, the following definition is introduced, although it only partly comprises the informal description.

Definition 6 (*characteristica of a database*).
Let $D_{ori(trans)}$ be an original (or transformed) database. Let N denote the number of intervals in the database. Let L denote the average interval length of a grey interval. Let D denote the average density of a grey interval. Then $C_{D_{ori(trans)}}$ denotes the characteristica of a database $D_{ori(trans)}$ and is defined as follows:

$$C_{D_{ori(trans)}} = N \times D \times L$$

Chapter 3
Intersection of Spatial Objects in an ORDBMS

In the last chapter we introduced *grey intervals* and how we can store them in an ORDBMS. In this chapter we focus on the process of interval intersection queries. In section 3.1 we define the terms interval *interlacing* and interval *intersection*. We then point out, in what cases we can tell, whether two interlacing grey intervals intersect each other or not, without accessing the *attached interval sequence*, stored in the BLOB. This test is only based on information, which are integrated in the *upper*- and *lowerIndexes*. We further introduce two different probability models, enabling us to predict the probability, whether *interlacing* intervals *intersect*. In section 3.2 we shortly survey the complete query process which consists of three consecutive filter steps. In section 3.3 we discuss the first step, which is based on the original RI-tree. In section 3.4 we introduce the so called *fast grey test*, which is a pure cpu test, yielding no additional I/O accesses. In section 3.5 it is shown how this second filter step is linked to the third one, which is discussed in section 3.6. In section 3.7 we point out, that the concept of grey intervals is especially useful for dynamically created query objects. We close this chapter by applying the optimization rules of the RI-tree of subsection 1.5.4 to the X-RI-tree.

3.1 Intersection of Interlacing Intervals

3.1.1 Introduction

In this section we introduce the two terms *intersect* and *interlace* and try to figure out, when two *interlacing* intervals *intersect* each other. Of course we could perform this task by examining the BLOBs, but we will confine ourselves to the following information, available in the *upper*- and *lowerIndexes*:

- *upper* and *lower bound* of the interval
- *density* of the interval d_{grey} (i.e. number of black (N_{black}) and white (N_{white}) cells of the interval)

What is discussed in this section forms a necessary prerequisite for the understanding of the query behavior of the X-RI-tree, which will be explained in detail in the following sections of this chapter.

3.1.2 Definitions

We will use the term *interlace* for the intersection of the hull of two grey intervals, whereas we will use *intersect*, if the attached interval sequences of two interlacing grey intervals intersect. For the intersection of two grey intervals it is a necessity that they interlace. Note that in SQL:1999, which provides the *Period* as basic interval data type, the corresponding operator for intersection is called "*overlap*" [Sno 00].

Definition 7 (*interval interlacing*)
Let $l_\tau, u_\tau, l_\kappa, u_\kappa \in IN$, let $\tau = (l_\tau, u_\tau)$ and $\kappa = (l_\kappa, u_\kappa)$ be two intervals. In the following, we say that τ and κ *interlace* (or, alternatively, τ *interlace*s κ), iff $(l_\tau \le u_\kappa) \wedge (l_\kappa \le u_\tau)$.

We call $l_{interlace} = \max(l_\tau, l_\kappa)$ the lower bound of the interlacing area and $u_{interlace} = \min(u_\tau, u_\kappa)$ the upper bound of the interlacing area.

Note that we defined *interval interlacing* in such a way that we can use it for the *hulls of grey intervals* as well as for *black intervals* of attached black interval sequences.

Definition 8 (*interval intersection*)
Let $\tau = ((l_\tau, u_\tau), S_\tau)$ and $\kappa = ((l_\kappa, u_\kappa), S_\kappa)$ be two (grey) intervals of cardinality n_τ and n_κ, which *interlace* each other. These two intervals *intersect* each other, iff $(\exists i \in 1 \ldots n_\tau)(\exists j \in 1 \ldots n_\kappa)$ so that s_i *interlaces* s_j.

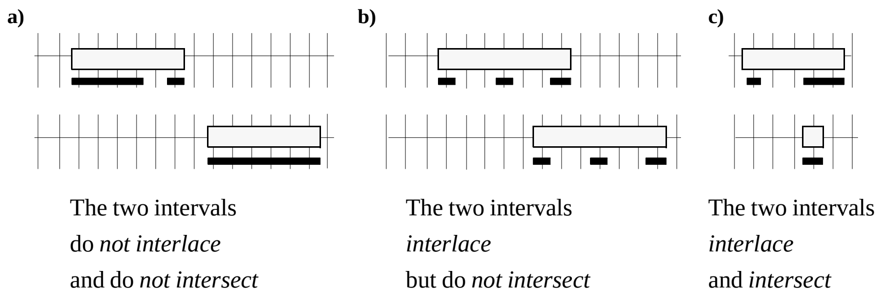


Figure 26: Interval *interlacing* and *intersection*
a) no interlacing, **b)** interlacing but no intersection, **c)** intersection

In figure 26 the relation between the two above definitions is depicted.

3.1.3 Intersection

3.1.3.1 Introduction

As we will see later on, it is very advantageous, not having to access the attached interval sequence of an interval, in order to decide whether two interlacing intervals intersect each other or not. If we can decide this, only based on the information introduced in subsection 3.1.1, we can omit the third filter step. In this subsection we will discuss what grey and black[1] intervals have to look like, so that we can test them successfully in a fast second filter step.

3.1.3.2 Two Black Intervals

If two black intervals interlace, they necessarily intersect as well. This is the standard case in the RI-tree, where we do not have any attached interval sequences. But we will find this situation in the X-RI-tree as well, although less frequently.

3.1.3.3 Black and Grey Intervals

In this case, the situation is a little bit more complicated than in the last subsubsection. But we will still see that in almost any cases where a black interval $I_{black} = (l_{black}, u_{black})$ interlaces a grey interval $I_{grey} = (l_{grey}, u_{grey})$, it intersects it as well. If any of the three conditions depicted in table 1 holds, then I_{black} and I_{grey} intersect each other.

[1] Recall that we call grey intervals $I_{grey} = ((l_{grey}, u_{grey}), <s_1>)$ also black intervals.

Note that the third case in table 1 is the generalization of the intersection test between two black intervals. Furthermore, it is a special case of the situation discussed in the next subsubsection.

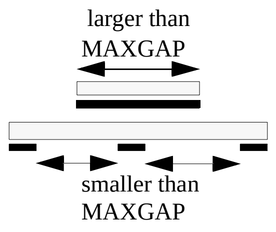
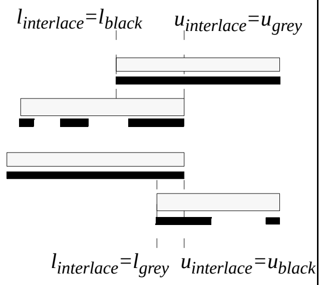
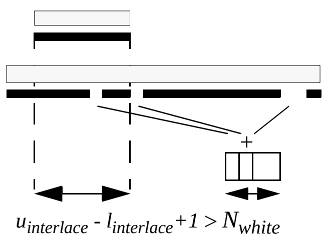
condition	explanation	figure
$L_{black} > MAXGAP$	If the black interval is longer than the *MAXGAP*$_{database}$ parameter, the two intervals intersect because the maximum gap between two black intervals of the attached interval sequence of I_{grey} is smaller than *MAXGAP*.	
$l_{interlace} = l_{black}$ and $u_{interlace} = u_{grey}$ or $l_{interlace} = l_{grey}$ and $u_{interlace} = u_{black}$	If one of the two conditions depicted in the cell on the left holds, the black and grey interval intersect. This is due to the fact, that the grey intervals end and start with black intervals. We do not allow grey intervals starting or ending with a "gap" (cf. definition 3)	 $l_{interlace}=l_{black}$ $u_{interlace}=u_{grey}$ $l_{interlace}=l_{grey}$ $u_{interlace}=u_{black}$
$N_{white} < L_{interlace}$	If the number of the white cells N_{white} of a grey interval is smaller than the length of the interlacing area, then the grey and the black interval necessarily intersect. Note, this is the reason why two black intervals which interlace each other also intersect each other (N_{white} of a black interval is 0)	

Table 1: Intersection between an interlacing black and grey interval

3.1.3.4 Two Grey Intervals

Of course, two grey intervals $I_{grey} = (l_{grey}, u_{grey})$ and $I'_{grey} = (l'_{grey}, u'_{grey})$ which interlace do not have to intersect. Fortunately, there are two cases where we can assert that they intersect without examining in detail the attached interval sequences of the *XRange* table. The two cases are illustrated in table 2.

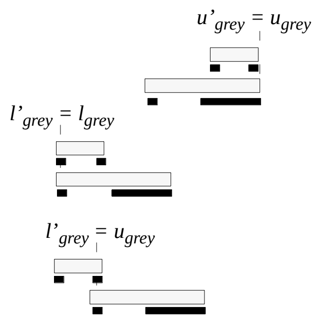
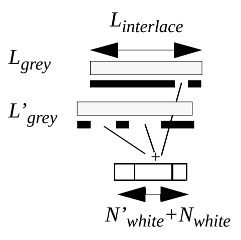
condition	explanation	figure
$u_{grey} = u'_{grey}$ or $l_{grey} = l'_{grey}$ or $l_{grey} = u'_{grey}$	If one of the three conditions depicted in the cell on the left holds, the two grey intervals intersect (grey intervals start and end with black intervals). This test is similar to the *polygon boundary test* in [HJR 97].	 $u'_{grey} = u_{grey}$ $l'_{grey} = l_{grey}$ $l'_{grey} = u_{grey}$
$N'_{white} + N_{white}$ $<$ $L_{interlace}$	If the sum of the number of the “white cells” of two grey intervals $N'_{white} + N_{white}$ is smaller than the length of the interlacing area, then the two intervals necessarily intersect. This is the generalization of the third case of table 1. This test is similar to the *false area test* in [BKSS 94].	 $L_{interlace}$ L_{grey} L'_{grey} $N'_{white}+N_{white}$

Table 2: Intersection between two interlacing grey intervals

3.1.4 No intersection

There are only two situations, depicted in table 3, where we can assert that two interlacing intervals do *not* intersect.

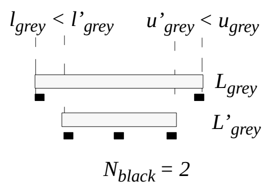
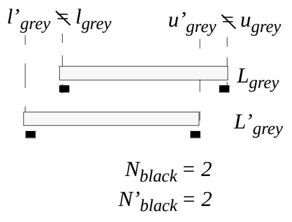
condition	explanation	figure
$N_{black} = 2$ and $l_{grey} < l'_{grey}$ and $u_{grey} > u'_{grey}$	If I_{grey} consists only of two black cells, and I'_{grey} is totally “included” in I_{grey} then we know that the two intervals cannot intersect each other, although they interlace.	 $l_{grey} < l'_{grey}$ $u'_{grey} < u_{grey}$ L_{grey} L'_{grey} $N_{black} = 2$
$N_{black} = 2$ and $N'_{black} = 2$ and $l_{grey} \neq l'_{grey} \neq u_{grey} \neq u'_{grey}$	If both grey intervals consist only of two black cells and, furthermore, have distinct interval bounds, then the two intervals certainly do not intersect.	 $l'_{grey} \neq l_{grey}$ $u'_{grey} \neq u_{grey}$ L_{grey} L'_{grey} $N_{black} = 2$ $N'_{black} = 2$

Table 3: No intersection between two interlacing grey intervals

3.1.5 Probability models

3.1.5.1 Introduction

As we have seen in the foregoing subsection we can pinpoint from time to time, based on a few information, whether two interlacing intervals intersect or not. Nevertheless, there will be a lot of cases where we cannot do that. But it is still helpful, if we can predicate how probable an intersection might be.

Since we want to integrate this probability model into the SQL statement of the first filter step, its computation should be rather cheap. This is the reason why we introduce two models. The first one models the problem a little bit better, but unfortunately it is very expensive to compute. On the other hand, the second one is easy to compute and also meets our needs.

The two probability models differ in whether *"drawn elements are put back into the bucket or not"*. The first one, which assumes, that they are not put back, equals the *sweepstake-model*, whereas the second one is equal to the *coin-toss experiment*, i.e. it is a *Bernoulli experiment*.

Both models assume that the black and white cells are equally distributed[1].

3.1.5.2 First Probability Model

In this subsection we introduce our first probability model and mention a few reasons, why it is not suitable for our needs. The probability for a successful intersection test is computed in the same way as you compute for instance the probability, that you tip *no* number right out of seven in the very popular sweepstake game *"7 out of 49"*.

- Consider two grey intervals I_{grey} (with a density d_{grey}) and I'_{grey} (with a density d'_{grey}) which interlace at a length L. As we assume that the black cells of both grey intervals are equally distributed, we can conclude that $N = d_{grey} \times L$ black cells of I_{grey} are included in the interlacing area and likewise $N' = \mathrm{d'}_{grey} \times L$ black cells from I'_{grey}. We now compute the probability P for a successful intersection in the following way:

[1] We neglect the fact, that the grey intervals represent (parts of) spatial objects and that in this case the white and black cells of the grey intervals tend to form groups.

$$P = 1 - \frac{\binom{L-N}{N'}}{\binom{L}{N'}}$$

$\binom{L-N}{N'} =$ number of different possibilities, how *N’* black cells of I'_{grey} can be placed over the remaining *L-N* white cells, assuming that all *N* black cells of I_{grey} are already positioned.

$\binom{L}{N'} =$ number of all different possibilities, how *N’* black cells of l'_{grey} can be placed over the *L* cells of the interlacing area.

$$P = 1 - \frac{\binom{L-N}{N'}}{\binom{L}{N'}} = \left(1 - \frac{(L-N)!(L-N')!}{L!(L-N-N')!}\right)$$

Figure 27: Computation of P (first probability model)

- Computing factorials *fac(n)* is a rather expensive operation. Straightforward algorithms are based on the following definition *fac(n)=n · fac(n-1)*, taking *O(n)* time[1].
  Furthermore, a lot of database systems do not provide the *fac-operator* as a *built-in function*. Therefore, we have to simulate it with a *loop-Operator*, executing the multiplication inside the loop. This loop-operator has to be embedded into a user-defined function, which has to be called within the SQL statement of the first filter step. As shown by M. Kornacker in [Kor 99], the calling of a user defined function out of an SQL statement is a very expensive operation, which should be avoided.

  To avoid high computational costs for the determination of *n!*, you could use approximations of the factorials, like stirlings formula: $n! \approx n^n e^{-n} \sqrt{2\pi n}$.
  This expression can be computed more efficiently, as you can compute a^n in *O(log n)* time, based on the following definition $a^n = (a^{n/2})^2$.

  Another way to reduce the runtime for the computation of P is to simplify the model, rather than approximating the result of a complex model. We pursue this approach in our second probability model.

1 We assume that the multiplication is a basic operation.

3.1.5.3 Second Probability Model

Our second model is based on the following steps:

- Consider two grey intervals I_{grey} and I'_{grey} with their corresponding densities d_{grey} and d'_{grey}, which interlace at a length L.
- Let x be one of the cells in the interlacing area. The probability that this cell is covered by a black cell of I_{grey} and a black cell of I'_{grey} is $P_x = d'_{grey} \times d_{grey}$. The probability, that either x or an other cell y is covered by black intervals from I_{grey} and I'_{grey} is $P = d'_{grey} \times d_{grey} + (1 - d'_{grey} \times d_{grey}) \times d'_{grey} \times d_{grey}$. Thus the probability that I_{grey} and I'_{grey} share at least one black cell can be computed as follows[1]:

$$P = \sum_{\nu=0}^{L-1} d_{grey} d'_{grey} (1 - d_{grey} d'_{grey})^{\nu}$$

$$P = d_{grey} d'_{grey} \frac{1 - (1 - d_{grey} d'_{grey})^{L}}{1 - (1 - d_{grey} d'_{grey})}$$

$$P = 1 - (1 - d_{grey} d'_{grey})^{L}$$

Figure 28: Computation of P (second probability model)

- The last representation of P enables us, to tell something about the intersection probability of two intervals without putting too much effort in its computation. P can be computed efficiently without calling a *user-defined function*, using itself a *loop-operator.* Instead, we can confine ourselves to *built-in functions*, which are provided by off-the-shelf DBMSs:

```
algorithm ComputeP
begin
    return := floor (1 - power ( 1 - ($d_{grey} \cdot d'_{grey}$) ,
                  least ($u_{grey}$ , $u'_{grey}$) - greatest ($l_{grey}$ , $l'_{grey}$) + 1 )) ;
end ComputeP;
```

Figure 29: Algorithm for the computation of P (second probability model)

We will use this model and not the first one for further considerations.

[1] This formula can easily be proved by induction on the interlacing length L.

3.2 General Survey of the Query Process

The general query processing flow of the *X-RI-tree* consists of three major steps.

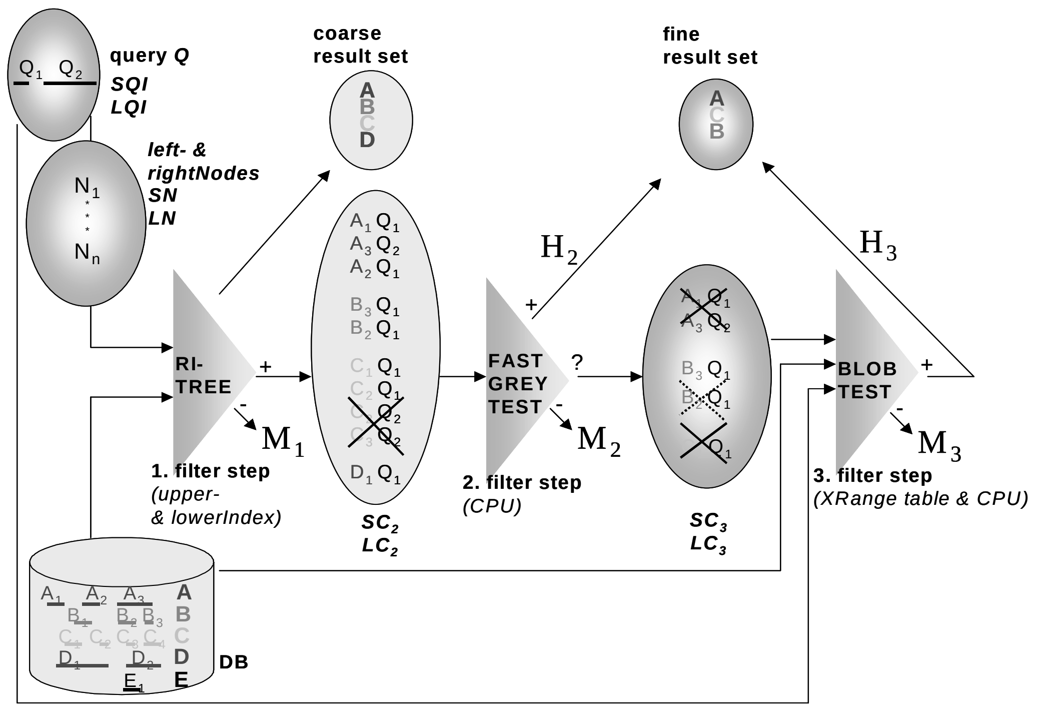
- In a first step we use the slightly modified *RI-tree* to determine all interlacing pairs of grey database and query object intervals. These interval pairs are ordered by database *ID* and probability *P*, introduced in the last section.
- In a second step we perform the so called *fast grey test* to determine intersecting intervals without examining the attached interval sequences. This test is based on the cases discussed in table 1 to 3.
- Finally, we carry out the expensive *BLOB test*, scrutinizing the attached interval sequences.

We could stop the query process after each of these three steps.

- Stopping after the first step yields an error-bound result, according to the $MAXGAP_{database}$ parameter.
- Stopping after the second step results in a result set, which might not be complete, but nonetheless correct. If you are only interested in questions like "*Does my query object intersect at least x database objects*" you might be able to leave out the third filter step completely.
- Stopping after the third step delivers an error-free complete result, with respect to the voxel set representation of the objects.

Figure 30 depicts the complete query processing flow, including a short example for clarification. In this example our database consists out of five objects *A, B, C, D,* and *E*, and a query object *Q* consisting of two grey intervals. Each of the database objects is composed out of one to four grey intervals with their corresponding attached interval sequences.

In a first step we determine all pairs of interlacing grey database and query intervals (e.g. A_1Q_1, A_3Q_2, A_2Q_1, B_3Q_1, ...). These pairs are ordered by database *ID* and decreasing probability *P* for a further successful intersection test. This first filter step is entirely based on the *upper*- and *lowerIndex* of the *XRange* table. At this point we could stop the query process, telling that object *A,B,C,* and *D* intersect our query object *Q* with a maximum error of *MAXGAP/2*.



H_i: database objects intersecting the query object (**H**it).

M_i: database objects not intersecting the query object (**M**iss).

C_i: pairs of query and database intervals, which might be tested in step i (**C**andidates).

T_i: pairs of query and database intervals, which have actually been tested in step i (**T**ests).

$\$_i$: costs for testing one interval pair in step i.

SQI (*LQI*) : ***S***tructure (***L***ist) of ***Q***uery ***I***ntervals.

SN (*LN*) : ***S***tructure (***L***ist) of *left-* and *right**N**odes*.

SC*2* (LC*2*): ***S***tructure (***L***ist) of ***C***andidates for the ***2***nd filter step.

SC*3* (LC*3*): ***S***tructure (***L***ist) of ***C***andidates for the ***3***rd filter step.

● transient tables

✕ deleted in the second filter step

⤫ deleted in the third filter step

Our goal is to receive the result set $H_2 \cup H_3$ with minimal costs: $\sum_{= 1..3} \$_i \times |T_i|$

Figure 30: General Survey of the Query Process

In a second, main memory based filter step, we consider all these pairs (i.e. candidate set C_2) grouped by database *ID*. We test them until the first successful test occurs or until there are no more pairs of this *ID* to test. This test is based on the reflections discussed in section 3.1. If the test yields neither a positive nor a negative result, we subjoin this pair to C_3, the candidate set of the third filter step (e.g. A_1Q_1, A_3Q_2, B_3Q_1, B_2Q_1,C_1Q_1). On the other hand, if the test is successful we add the database *ID* to the final result set (e.g. *A*, *C*) and delete all pairs belonging to the same ID out of C_2 (e.g. C_2Q_2,C_3Q_2) and C_3 (e.g. A_1Q_1, A_3Q_2, C_1Q_1). In the case of the X-RI-tree with a *MAXGAP* parameter equal to zero, the hits in this step (i.e. H_2) form the complete result set.

In a third step we pinpoint whether the remaining interval pairs intersect or not. This is done by accessing the BLOBs stored in the *XRange* table. If a test is successful (e.g. B_3Q_1), we stop examining other pairs belonging to the same *ID* (e.g. B_2Q_1) and add the corresponding object ID to the result set (e.g. *B*).

As you can see the number of tested pairs T_2 and T_3 is always equal or less than the number of pairs in C_2 and C_3. For a good runtime behavior it is essential that T_2 and T_3 are small. Filter step two as well as filter step three obey the algorithm of figure 31, aiming at the decreasing of the ratio between T_2/C_2 and T_3/C_3.

```
algorithm TestCandidates (C Candidates, R ResultSet)
begin
    TestSet T := C;
    while not T.IsEmpty() do
        Element e := T.first();
        if SuccessfulTest (e) then
            R := R+{e};
            T := T-{e'|e.id=e'.id};
        else
            T := T-{e};
        end if;
    end while;
end TestCandidates.
```

Figure 31: Algorithm *TestCandidates*

We will see in the following sections, dealing with filter step two and three, how this algorithm is put into practice.

3.3 First Filter Step

3.3.1 Introduction

Basically, the first filter step can be seen as the application of the *RI-tree* to the transformed database. Note, that this step can be done without accessing the *XRange* table. It is entirely based on the *upper*-and *lowerIndex*.

In a preliminary step, the virtual backbone has to be traversed in order to collect all possible fork nodes of those intervals, which might interlace the query interval. Concerning this, the RI-tree and the X-RI-tree do not differ. After this preliminary step, which is implemented in a procedural runtime environment such as PL/SQL, we pass one single SQL query to the SQL engine. Thus, the first filter step is a *cursor-driven-operation* [Pöt 01].

3.3.2 Ranking

In the steps succeeding this first filter step, we have to examine whether two *interlacing* intervals *intersect* or not. In order to determine whether an object in the database intersects the query object, we may have to accomplish several interval intersection tests. As we can stop after the first succesful one (with respect to the same database *ID*) (cf. algorithm of figure 31), it is beneficial to do those tests first, which have the highest probability of being successful. In the example, depicted in figure 32, the database object with the ID *n* interlaces the query interval *q* q_n times.

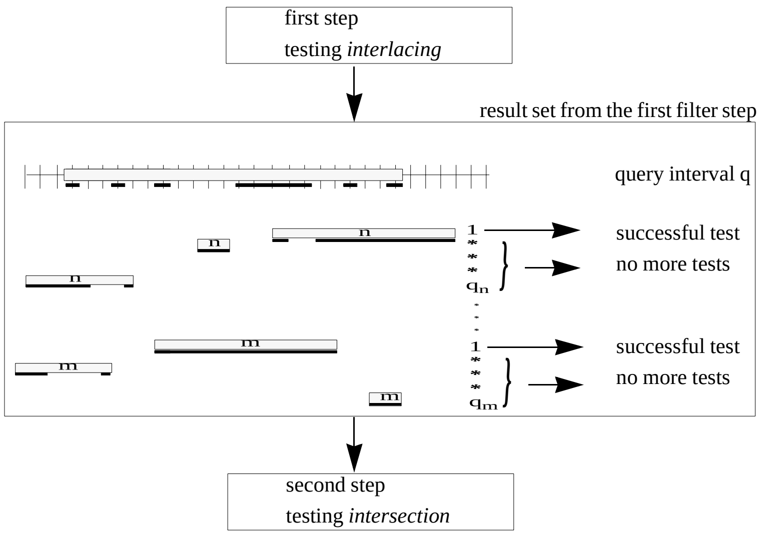


Figure 32: Ranking of the first filter step

But already the first test is successful, so the q_n*-1* other intersection tests have not to be carried out. The same holds for the ID *m* where we also have to test only once. Note, that if the candidate set for the second filter step C_2 had been in a different order, we would have carried out more intersection tests. Therefore, it is important, that C_2 is ordered in an *advantageous* way. What this exactly means, has already been discussed in section 3.1. We order the candidate set C_2 according to the second probability model introduced in 3.1.5. In order to put this idea into practice, we enlarge the transient tables *left- and rightNodes.*

3.3.3 Structure of the transient tables *left*- and *rightNodes*

In the final approach for the *RI-tree*, both transient tables *leftNodes (from, to, lower)* and *rightNodes (from, to, upper)* contain information about the nodes of the virtual backbone (*from* and *to*) and about one boundary value of the query interval *(lower, upper).* As we need more information belonging to the query interval in order to rank the results of the first filter step in a proper way, we augment both transient tables so that they obey the relational schema:

```
SN= {
          N_from          number,
          N_to            number,
          Q_ref           integer,
          Q_lower         number,
          Q_upper         number,
          Q_AIS           TAIS
     }
```

Figure 33: *S*tructure of the transient table *left*- & *rightNodes*

Note, that the four last entries of the transient table *left*- and *rightNodes* are equal to the entries in the *transient input query table LQI* introduced in section 3.6. We introduced redundancy in the *left*- and *rightNodes* table in order to omit an additional join with *LQI* in the first filter step. This join would be based on the *Q_ref* attribute, which points to exactly one grey interval in *LQI*, i.e. refers to one grey interval of the complete spatial query object.

a)

```
SELECT id FROM intervals i, :leftNodes left
WHERE i.node BETWEEN left.n_from AND left.n_to
        AND i.upper >= left.q_lower                      // inner query and left queries
UNION ALL
SELECT id FROM intervals i, :rightNodes right
WHERE i.node BETWEEN right.n_from AND right.n_to
        AND i.lower <= right.q_upper;                    // right queries
```

b)

```
SELECT STATEMENT
        UNION-ALL
                NESTED LOOPS
                        COLLECTION ITERATOR
                        INDEX RANGE SCAN lowerIndex
                NESTED LOOPS
                        COLLECTION ITERATOR
                        INDEX RANGE SCAN upperIndex
```

Figure 34: Interval intersection query (first step only)
a) SQL statement, and **b)** execution plan

3.3.4 One step only

Obviously, we could stop the query process after the first filter step, accepting an error corresponding to the *MAXGAP* parameter. Therefore, it might be wise to choose the *MAXGAP* parameter not only based on storage space requirement reasoning , but also on facts like "*A lot of application-queries are satisfied, if the maximum error, does not exceed MAXGAP*/2". Thus the set C_2 forms an error bound result set, which meets the needs of many application queries.

In figure 34 the final SQL statement for the first filter step and the corresponding execution-plan are shown, provided we omit the second filter step.

3.3.5 Final SQL command for the first filter step

If we continue the query process after the first step, the corresponding SQL statement has to be enlarged. The necessary changes are depicted in figure 35. Note, that we do not call a stored procedure to do the ranking but only use built-in functions of ordinary ORDBMS. The execution plan belonging to this SQL statement is the same as the one shown in figure 34, as we do not select fields from the *XRange* table, which are not included in the corresponding indexes.

```
SELECT
        DB_id, DB_row, Q_ref,
        DB_lower, DB_upper, DB_density, Q_lower, Q_upper, Q_density
FROM
(
        SELECT
        floor (1-power(1-(i.AIS.density · left.Q_AIS.density ),
               least(i.upper,left.Q_upper) -greatest (i.lower,left.Q_lower) +1 )) as Q_DB_rank,
        i.id as DB_id , i.rowID as DB_row, left.Q_ref as Q_ref,
        i.lower as DB_lower, i.upper as DB_upper, i.AIS.density as DB_density,
        left.Q_lower as Q_lower,  left.Q_upper as Q_upper, left.Q_AIS.density as Q_density
        FROM intervals i, :leftNodes left
        WHERE i.node BETWEEN left.N_from AND left.N_to
        AND i.upper >= left.Q_lower                          // left and inner queries
        UNION ALL
        SELECT
        floor (1-power(1-(i.AIS.density · right.Q_AIS.density ),
                least(i.upper,right.Q_upper) -greatest (i.lower,right.Q_lower) +1 )) as Q_DB_rank,
        i.id as DB_id , i.rowID as DB_row,right.Q_ref as Q_ref ,
        i.lower as DB_lower, i.upper as DB_upper, i.AIS.density as DB_density,
        right.Q_lower as Q_lower, right.Q_upper as Q_upper, right.Q_AIS.density as Q_density
        FROM intervals i, :rightNodes right
        WHERE i.node BETWEEN right.N_from AND right.N_to
        AND i.lower <= right.Q_upper                         //right queries
)
ORDER BY DB_id, Q_DB_rank desc
```

Figure 35: Final SQL statement of the first filter step

3.4 Second Filter Step

In this section we put into practice what we discussed in section 3.1. But first, we shortly address ourselves to the structure of the result set of the first filter step, i.e. the candidate set of the second step.

3.4.1 Structure of the candidate set of the second filter step SC_2

As already mentioned, it is not enough that the first filter step delivers the object *ID*s of those objects interlacing the query interval. We need additional information for the second step, which has to be provided by the first one. This information is collected in SC_2 (*S*tructure of *C*andidates for step *2*), shown in figure 36. The first two fields *DB_id* and D*B_row* reference ROWs in the *XRange* table, i.e. in the database. The next field *Q_ref* points to an entry in *LQI*, so that we can join both the *XRANGE* table and *LQI* in the third filter step. The next six fields *DB_lower*, *DB_upper*, *DB_density*, *Q_lower*, *Q_upper* and *Q_density* are used to determine in the second filter step, whether the query and database interval intersect without examining the

```
SC_2= {
        DB_id          integer,    // necessary for 3rd filter step
        DB_row         varchar,    // necessary for 3rd filter step
        Q_ref          integer,    // necessary for 3rd filter step
        DB_lower       number,     // necessary for 2nd filter step
        DB_upper       number,     // necessary for 2nd filter step
        DB_density     number,     // necessary for 2nd filter step
        Q_lower        number,     // necessary for 2nd filter step
        Q_upper        number,     // necessary for 2nd filter step
        Q_density      number      // necessary for 2nd filter step
     }
```

Figure 36: Structure of the candidate set for the second filter step SC_2

corresponding BLOBs. Note, that we do not collect the results from the first filter step in a transient table, but process them right away. In the remainder of this work, we will use the notion LC_2 (*L*ist of *C*andidates for the *2nd* filter step) for both the “entire result set of the first step” and for a “cursor running through this set”.

3.4.2 Algorithm

In the second filter step we perform the *fast grey test* for each entry in LC_2, i.e. for each pair of interlacing intervals, until no more intervals are available or a pair of the same database ID has already been tested positive. This is done, by calling the function *SecondFilterStep* (cf. figure 37), which tests, whether the database interval and the query interval intersect. This test is based only on the upper and lower bounds of

```
function SecondFilterStep (DB_lower, DB_upper, DB_density,
                           Q_lower, Q_upper, Q_density) : integer;
begin
    if the query and database interval intersect (cf. subsection 3.1.3) then
        return 1
    else if the query and database interval do not intersect (cf. subsection 3.1.4) then
        return 2
    else
        return 3
    end if;
end SecondFilterStep;
```

Figure 37: Procedure *SecondFilterStep*

the intervals and their density. Note, that based on this information you can calculate the number of black cells and decide whether the interval is black or grey.

3.5 Connection between the Second and the Third Filter Step

3.5.1 Introduction

In this section we explain the connection between the second and the third filter step. The two steps do not strictly follow each other in the temporal flow of the query process, but take turns. They alternate because otherwise the transient table LC_3 could become very large and in its aftermath the main memory footprint could not be controlled. In this section we will introduce a concept which solves this problem.

3.5.2 The LC_3MAX parameter

The algorithm depicted in figure 38 illustrates the general connection between the second and the third filter step. We introduce a user defined boundary LC_3MAX, which allows us, to control the main memory footprint. If the number of records in the transient table LC_3 exceeds this value, we perform the third filter step. The disadvantage of this construction is, that we cannot predict in advance, how many SQL statements have to be executed in the third step because this depends on the number of generated candidates by the two preceding steps. On the other hand, we do control the main memory footprint and make thereby the X-RI-tree fit for a multi-user-environment.

If we detect, that a database object intersects our query object, we can leave out all following tests belonging to this database object. Owing to the fact that the result set of the first filter step is ordered by *DB_id* we only have to keep in mind the last successfully tested database *ID* and not a list of already successfully tested objects in order to skip tests. Furthermore, LC_2 is ordered by Q_DB_rank (=probability value for the intersection of query and database interval). This ordering was mainly designed to meet the needs of the third filter step, but it is also beneficial for the second step.

```
algorithm SecondAndThirdFilterStep
begin
    LastFoundDBID := -1;          // variable used for skipping tests
    ResultSet     := {};          // result set
    LC_3          := {};          // Candidates for the third step
    execute FirstFilterStep ;
    while LC_2 not empty do
        // all information of the current candidate in LC_2 is assigned to local variables, needed for filter step two and three
        LC_2.fetch into   (DB_id, DB_row, Q_ref,
                           DB_lower, DB_upper, DB_density,
                           Q_lower, Q_upper, Q_density)
        if DB_id <> LastFoundDBID then
            result_of_2_filter_step := SecondFilterStep ( DB_lower, DB_upper, DB_density,
                                                          Q_lower, Q_upper, Q_density) ;
            if result_of_2_filter_step =1 then
                ResultSet := ResultSet + {DB_id};
                LastFoundDBID := DB_id;
                Delete all records in LC_3 with ID = DB_id; // we do not have to examine these records in
                                                            // filterstep 3, because we already determined that DB_ID
                                                            // intersects our query object.
            else if result_of_2_filter_step =2 then
                do nothing;
            else // add it to the candidate list of the third step
                LC_3 := LC_3 + {(DB_id, DB_row, Q_ref)};
                if LC_3.count > LC_3MAX then              // we control the main memory footprint
                    execute ThirdFilterStep;
                    ResultSet := ResultSet +{results of third step};
                    LC_3:= {};
                end if;
            end if;
    end while;
    // all results from the first step are processed
    // we have to check whether there are some more candidates for the third step and if necessary execute the corresponding statement
    if not LC_3.IsEmpty() then
        execute ThirdFilterStep;
        ResultSet := ResultSet +{results of third step}
    end if;
end SecondAndThirdFilterStep;
```

Figure 38: Connection between second and third step

3.6 Third Filter Step

3.6.1 Introduction

In the second filter step there might be a lot of interval pairs for which we cannot decide by means of the *fast grey test*, whether they intersect or not. In contrast, in the third filter step we can do that for all interval pairs, by scrutinizing the attached interval sequences. These sequences might be materialized and stored in a BLOB, or being generated during the fetch-calls of the ADT *TIS*.

In this section, we will first shortly discuss the structure of *LQI* and LC_3, which are used in the SQL statement, executed in the third step. Furthermore, we will emphasize how the algorithm *TestCandidates* of figure 31 is implemented. Additionally, we will talk about how we can efficiently access materialized black interval sequences which are stored in a BLOB.

3.6.2 Structure of the transient table of query intervals *LQI*

The original RI-tree does not have to store any information about the query interval. The entire information consists of the values of the lower and upper bound of the interval. This information is completely used in the preliminary procedural step by filling the transient tables *left*- and r*ightNodes*.

In the case of the X-RI-tree the query interval consists also of an attached interval sequence, stored in a BLOB. In the *third filter step* we have to access this information. We therefore store a grey query interval in *SQI* (*S*tructure of *Q*uery *I*nterval), depicted in figure 39.

```
SQI = {
            Q_ref integer,
            Q_lower number,
            Q_upper number,
            Q_AIS TAIS,
            Q_IS TIS
          }
```

Figure 39: *S*tructure of *Q*uery *I*nterval *SQI*

Note, that in the case of query objects consisting of several intervals, we just use the corresponding transient table *LQI* (*L*ist of *Q*uery *I*ntervals), where the field *Q_ref* is used as an identifier of the different grey query intervals. The boundary values *Q_lower* and *Q_upper* allow us to compute the exact interlacing area, so that we can concisely examine the BLOB.

3.6.3 Structure of the candidate set of the third filter step SC_3

This structure is rather simple. It consists of the first three fields of SC_2, which are just passed through in the second filter step (cf. figure 40).

We need these fields in the third step in order to join the transient *query input table LQI* and the *XRange* table of the database.

```
SC3 = {
          DB_id          integer,
          DB_row         varchar,
          Q_ref          integer
        }
```

Figure 40: *S*tructure of the candidate set of the third filter step SC_3

3.6.4 SQL statement

As already mentioned, there can be several records of LC_3, belonging to the same (database) ID. For each of these IDs we have to decide whether there exists one record in LC_3 intersecting the corresponding query interval. If we have found one, we do not have to examine the other records belonging to this ID. As the records of LC_3 are ordered by ID plus additional criteria, we can keep in mind the last ID intersecting the query interval and skip all the other tests belonging to this database ID. Thus we can adjust the "*TestCandidates*" algorithm of figure 31 ending up with an algorithm depicted in figure 41. Note, the adjusted algorithm does not delete the candidates

a)

```
algorithm ThirdFilterStep;
begin
    LastFoundDBID := - 1;
    LC3.first;
    Result := {};
    for i :=1 to LC3.count() do
        if LC3[i].DB_id <> LastFoundDBID then
            if IntersectionTest(query interval of LC3[i],database interval of LC3[i]) then
                Result := Result + LC3[i].DB_id;
                LastFoundDBID := LC3[i].DB_id;
            end if;
        end if;
    end for;
end ThirdFilterStep;
```

b)

```
select distinct (id) from intervals i,: LC3 c, :LQI q
where   skipID(c.DB_id) = 0 //optional, can be omitted as it is included in the IntersectionTest procedure as well
        and i.rowID = c.DB_row and q.Q_ref=c.Q_ref
        and IntersectionTest (i.lower, i.upper, i.IS, q.lower, q.upper, q.IS, i.ID)=1
```

c)

```
SELECT STATEMENT
        NESTED LOOPS
            NESTED LOOPS
                COLLECTION ITERATOR PICKLER FETCH
                TABLE ACCESS BY USER ROWID
            COLLECTION ITERATOR PICKLER FETCH
```

Figure 41: Third filter step
a) algorithm, **b)** SQL statement, and **c)** execution plan

which are not tested but just ignores them. The SQL statement putting this algorithm into practice and the corresponding execution plan are depicted together in the same figure. Note, the SQL statement only works according to the described algorithm, if the three different predicates of the where-clause are evaluated in the order they are written down. The function *skipID* corresponds to the first if-statement of the algorithm. If a former record of LC_3, with the same database ID as the actual one, has already intersected one of the query intervals, *skipID* delivers 1 and the other two predicates of the where statement are not evaluated. If skipID = 0, the intersection test is carried out. If the *IntersectionTest* is successful, *LastFoundDBID* is set to the current ID of LC_3. This is done in the *IntersectionTest* procedure. As we cannot assume, that the query optimizer evaluates the predicates of the where-clause in the order they have been written down, we include the skipping also into the *IntersectionTest* procedure. In this case the BLOBs are accessed, as they are passed as parameters to the stored procedure, but their content is not scrutinized, because the first thing done in the stored procedure *IntersectionTest*, is to test whether the parameter value *ID*, is equal to the latest successfully tested database ID (cf. figure 42). This version is of course slightly slower, but the declarativity of SQL is maintained.

3.6.5 Stored procedure *IntersectionTest*

In order to decide whether two interlacing intervals $I_1=((l_1,u_1),S_1)$ and $I_2=((l_2,u_2),S_2)$ intersect each other or not, we have to carefully examine their attached interval sequences S_1 and S_2. This is done in the procedure *IntersectionTest* as shown in figure 42. First, the fetch-methods of the abstract data types *DB_IS* and *Q_IS* are invoked. The resulting two interval lists are compared to each other in order to find out whether they contain interlacing black intervals. If this is the case, the two grey intervals intersect and the procedure stops. If one interval list has been tested until its end, the fetch-method of the corresponding *TIS* object is called again and the test proceeds. If an interval list is empty the procedure stops and the two grey interlacing intervals do not intersect.

Note, that if an intersection is detected, the *IntersectionTest* procedure does not have to test all intervals of the interlacing area but can stop as soon as an intersection is detected. This is the reason, why the loop-operator has been included into the *IntersectionTest* procedure. It allows us to fetch small portions of intervals of the interlacing area. Thus unnecessary disk accesses can be avoided. In the experiments,

```
algorithm IntesectionTest (  DB_lower number, DB_upper number , DB_IS TIS ,
                             Q_lower number, Q_upper number, Q_IS TIS, DB_ID integer);
const N = MAXINT; // number of intervals per fetch;
begin
    if LastFoundDBID = DB_ID then
            result := 0
    else
            l_interlace := max (DB_lower, Q_lower);
            u_interlace := min (DB_upper, Q_upper);
            InterlaceArea := TInterval(l_interlace,u_interlace);
            // we fetch maximum N intervals from the interlacing area.
            DB_IntervalList := DB_IS.fetch (InterlaceArea, N);
            Q_IntervalList := Q_IS.fetch (InterlaceArea, N);
            TestFinshed := false;
            while not TestFinished do
                result := DB_IntervalList.intersects(Q_IntervalList); // parallel run through the lists
                if result = 0 then // either DB_IntervalList or Q_IntervalList was empty => no intersection. Test finished
                    TestFinished := true
                else if result = 1 then // intersection detected. Test finished
                    TestFinished := true
                else if result = 2 then // DB_IntervalList tested until the end => fetch new one
                    DB_IntervalList := DB_IS.fetch (InterlaceArea, N);
                else if result = 3 then // Q_IntervalList tested until the end => fetch new one
                    Q_IntervalList := Q_IS.fetch (InterlaceArea, N);
                else if result = 4 then // DB_IntervalList and Q_IntervalList tested until the end => fetch new ones
                    DB_IntervalList := DB_IS.fetch (InterlaceArea, N);
                    Q_IntervalList := Q_IS.fetch (InterlaceArea, N);
                end if;
            end while;
            if result =1 then
                 LastFoundDBID := DB_ID;
            end if;
    end if;
    return result;
end IntesectionTest ;
```

Figure 42: IntersectionTest

presented in the next chapter, we did not exploit this feature. The *fetch*-method was always invoked with a second parameter equal to *MAXINT.*

The algorithm of figure 42 can be applied to all subtypes of the ADT *TIS*. In the next subsubsection we survey the subtypes *TBitOrientedIS* and *TOffsetOrientedIS* which materialize the black interval sequences and store it in a BLOB.

3.6.5.1 Accessing an Interval Sequence in a BLOB

As shown in theorem 3, we can pinpoint for each black interval sequence whether the offset-oriented approach or the bit-oriented approach is preferable.

Obviously, the number of bytes needed for the storage of a black interval sequence correlates with the number of disk accesses which are needed for reading the complete interval sequence.

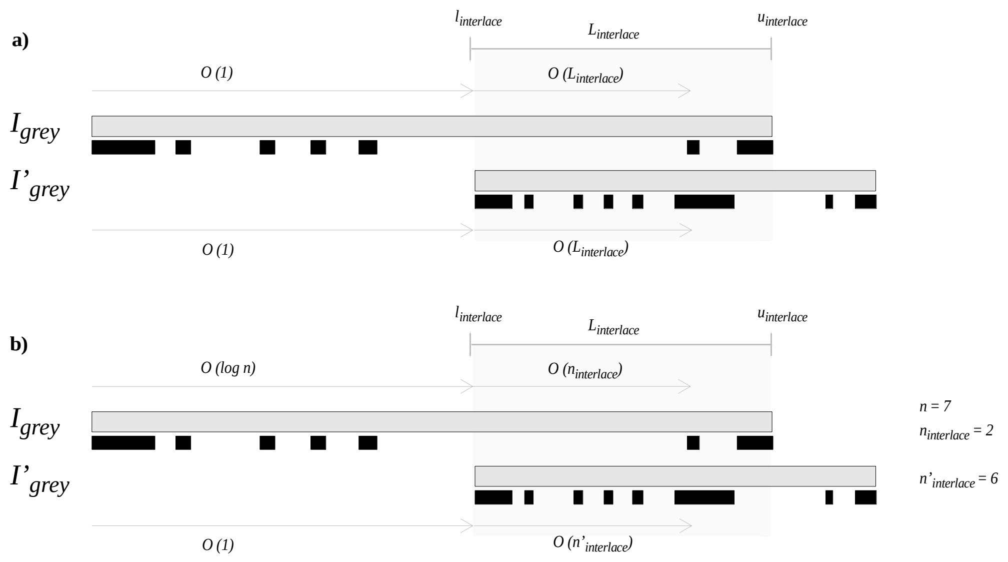


Figure 43: Accessing the attached interval sequences stored in a BLOB
a) bit-oriented, **b)** offset-oriented

As shown in figure 43 it is not always necessary to access the whole interval sequence. It is enough to test those parts of the interval sequence falling into the interlacing area. This imposes a twofold problem:

- Finding the starting point of the interlacing area efficiently.
- Accessing those parts of the black interval sequence efficiently which fall into the interlacing area.

The following two theorems deal with these questions in case both intervals follow the same storage approach. The theorems can easily be extended to those cases where one interval is *bit-oriented* and the other *offset-oriented* organized.

Theorem 4 (bit-oriented approach (access))
Let $I_{grey} = ((l_{grey}, u_{grey}), S_{black})$ and $I'_{grey} = ((l'_{grey}, u'_{grey}), S'_{black})$ be two grey intervals which interlace. Let $L_{interlace}$ be the length of the interlacing area and b the disk block size. Then $O(L_{interlace} / b)$ disk accesses are needed for testing these two intervals for intersection.

Proof. We can find the starting point of the interlacing area $l_{interlace}$ in both intervals in $O(1)$ time by using $l_{interlace} - l_{1/2}$ as offset. We have to test at most $L_{interlace}$ bits to pinpoint whether the two intervals intersect. This worst case occurs if there is no intersection. Thus we need $2 \cdot O(1) + 2 \cdot O(L_{interlace} / b) = 0(L_{interlace} / b)$ disk accesses for the intersection test, as the bits are consecutive organized in the BLOB.

Theorem 5 (offset-oriented approach (access))
Let $I_{grey} = ((l_{grey}, u_{grey}), S_{black})$ and $I'_{grey} = ((l'_{grey}, u'_{grey}), S'_{black})$ be two interlacing grey intervals of length L_{grey} and L'_{grey}. Let b be the disk block size and $l_{interlace} = l'_{grey}$. Let n be the cardinality of S_{black}, and let $n_{inlerlace}$ and $n'_{inlerlace}$ be the number of black intervals of S_{black} and S'_{black} interlacing the interval $I_{interlace}=(l_{interlace}, u_{interlace})$. Then testing these two intervals for intersection needs the following number of disk accesses:

$$O(\log n + (n_{interlace} \log L_{grey})/b + (n'_{interlace} \log L'_{grey})/b)$$

Proof. The starting point of the interlacing area $l_{interlace}$ of I' can be accessed in $O(1)$ time. In I we can find this point by bipartitioning. We first access the value in the middle of the BLOB and compare it to $l_{interlace}$. If it is smaller we only have to consider the upper half of the BLOB. If higher, we take the lower half. This test can be done in $0(1)$ time and has to be done at most $O(\log n)$ times. Thus finding $l_{interlace}$ in both intervals needs $O(1+\log n) = O(\log n)$ disk accesses. Then we have to access all black intervals in the interlacing area until we detect an intersection. These are at most $n_{inlerlace}+n'_{inlerlace}$ intervals which are consecutive organized in the BLOB. Each of these boundary values can be expressed by $O(\log L)$ respective $O(\log L')$ bits as outlined in section 2.3.
Thus $O(\log n + (n_{interlace} \log L_{grey})/b + (n'_{interlace} \log L'_{grey})/b)$ disk accesses are needed in order to decide whether the two intervals intersect.

3.7 Dynamically created query objects

3.7.1 Introduction

As already mentioned in section 1.3, we can encode a spatial object by an interval sequence while recursively decomposing the space into an *error-bound* or a *size-bound* approximation of the object. What we did up to now, was using the *error-bound* approach by closing small gaps between the intervals from *bottom-up*, leading us to grey intervals. As we wanted to work error-free, we added the attached interval sequences to the grey intervals, so that in the third filter step we could determine exactly whether two grey intervals *intersect* or only *interlace*. If we want to know, which parts in our database are intersected by a query object stemming from the database as well, we can revert to the attached interval sequences, available in the database. Thus, we do not have to process the costly step of creating the attached interval sequences during the collision query process.

On the other hand, if a user wants to know which parts are in an area of his interest, our system has to decompose this spatial query object into grey intervals and their corresponding interval sequences before the actual query process can start. This decomposition, especially of large objects, could cost us minutes or even hours (cf. section 4.4). Therefore, it is necessary to provide a concept for the decomposition of spatial objects, which are not known in advance. We shortly explain this concept, using *boxes* as an example for dynamically created query objects.

3.7.2 Basic idea

The basic idea is to use the *error-bound top-down* approach with a big *MAXGAP* parameter, so that we can quickly receive grey intervals. As most of these spatial objects can be described with a few parameters (e.g. a box can be described by two points), the corresponding abstract data types consist only of a few attributes (cf. the type *TBoxOrientedIS* of figure 19).

Note, that the algorithm of figure 6 has to be adjusted to the needs of the X-RI-tree. During the top-down approach we have to ensure, that our grey intervals start and end with black voxels. Furthermore, we have to compute the density of these intervals during this process in order to use the X-RI-tree without further changes.

If the query objects are boxes, these restrictions and computations can be easily integrated in the recursively decompositioning algorithm of figure 6.

3.8 Optimizations

In this section, we want to discuss, how far the optimization rules of the RI-tree, introduced in subsection 1.5.4 , can be applied to the X-RI-tree.

3.8.1 Gap optimization

Unfortunately, we cannot apply the gap-optimization of the RI-tree straightforward to the X-RI-tree, owing to the gaps included in the grey intervals itself.

But if the grey object interval sequence includes black intervals with a length longer than $MAXGAP_{database}$ we can apply the gap optimization to the X-RI-tree as

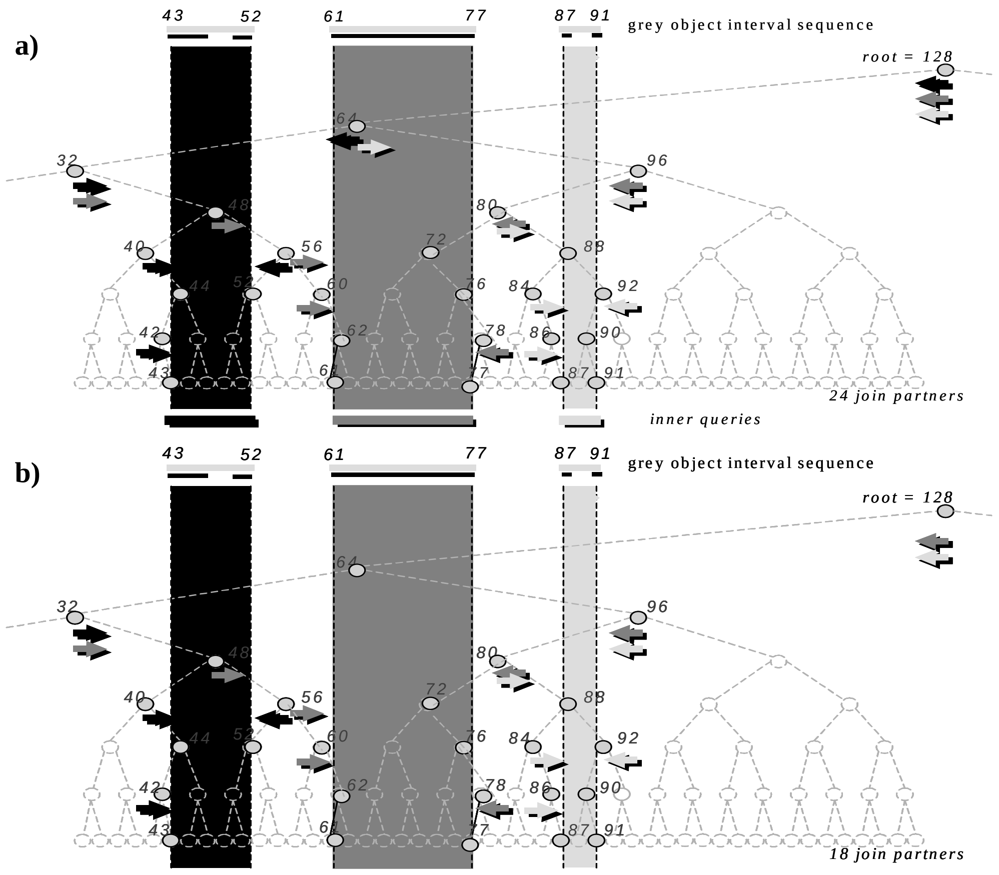


Figure 44: Optimization of the X-RI-tree
a) naive X-RI-tree **b)** optimized X-RI-tree

well. The example in figure 44 shows, that in the case, that the middle interval of the query sequence is such a long black interval, we can save a right query of the leftmost interval from node 128 and two queries of node 64. Unfortunately, the number of long black intervals decreases very fast with an increasing *MAXGAP* parameter (cf. figure 51). Therefore, this optimization is not as beneficial to the X-RI-tree as it is to the RI-tree.

3.8.2 Integrating Inner Queries

Integrating inner queries does not yield any problems. This optimization can be applied to the X-RI-tree without further changes. In the example of figure 44 we save three more join partners by integrating the inner queries.

Note, that in the experiments we always compare the optimized variant of the RI-tree to the optimized variant of the X-RI-tree.

Chapter 4
Experimental Evaluation

4.1 Introduction

In this chapter, we want to evaluate the performance of the X-RI-tree based on two test data sets *CAR* and *PLANE* (cf. figure 15). These test data sets were provided from our industrial partners, a German car manufacturer and an American plane producer, in form of high resolution voxelized three-dimensional CAD parts. In both cases, the Z-curve was used as a space filling curve to enumerate the voxels. To express one of these z-values we had to spend 33 bits in the case of the *CAR* data and 42 bits in the case of the *PLANE* data. To put it another way, the *CAR* data space consists of 8.589.934.592 voxels and the *PLANE* data space of 4.398.046.511.102 voxels. The voxels were grouped together to black intervals, so that we could use these data as test sets for the RI-tree. Furthermore, we used different $MAXGAP_{database}$[1] parameters in order to evaluate the X-RI-tree. Note, that the X-RI-tree and the RI-tree coincide if the *MAXGAP* parameter is 0. In this case we always used the original optimized version of the RI-tree and not the X-RI-tree.

As the RI-tree outperforms competitive techniques like the *Linear Segment Tree* and the *Composite Index* by factors of up to 4.9 for the query response time and the *Linear Quadtree (Octree)* and the *Relational R-tree* by factors of up to 4.6 and 58.3 [Pöt 01] [KPS 01a], the X-RI-tree was only compared to the RI-tree[2].

[1] If not otherwise stated, we use the term *MAXGAP* for denoting a global $MAXGAP_{database}$ parameter.

[2] According to the motto:"If you beat the best, you can beat them all."

We have implemented the optimized RI-tree and the X-RI-tree for the Oracle Server Release 8.1.7, using PL/SQL for the computational main memory based programming. All experiments have been performed on a Pentium III/700 machine with IDE hard drives. The database block cache was set to 500 disk blocks with a block size of 8 KB and was used exclusively by one active session.

As already mentioned, F_{MAXGAP} is the only transformation function we want to investigate in more detail. We apply this function to both example data sets *CAR* and *PLANE* and evaluate both the static and the dynamic behavior of the X-RI-tree. Before doing that, we shortly pinpoint in section 4.2, that accessing a BLOB in Oracle 8*i* is in accordance with our expectations. In section 4.3 we evaluate the static properties, including the *characteristica of the transformed database*s, *interval histograms* and *storage requirements*. In section 4.4 we concentrate on the dynamic behavior of the X-RI-tree, as for instance *response time*, *main memory footprint*, *tested candidates* and number of *disk accesses*. We conclude this chapter with section 4.5, where we summarize the main experimental results.

4.2 BLOB access

The only experiment presented in this section generally clarifies a few things about BLOB accessing in Oracle 8*i*.

In figure 45a it is shown, that only a few logical reads are necessary to open a BLOB no matter how large it is. Furthermore you can see, that if you want to read a

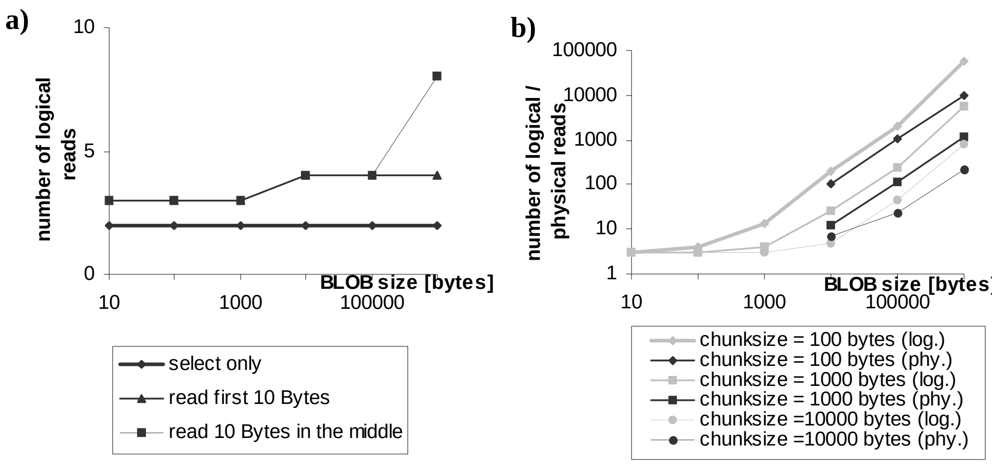


Figure 45: BLOB access
a) tiny parts, **b)** whole BLOB

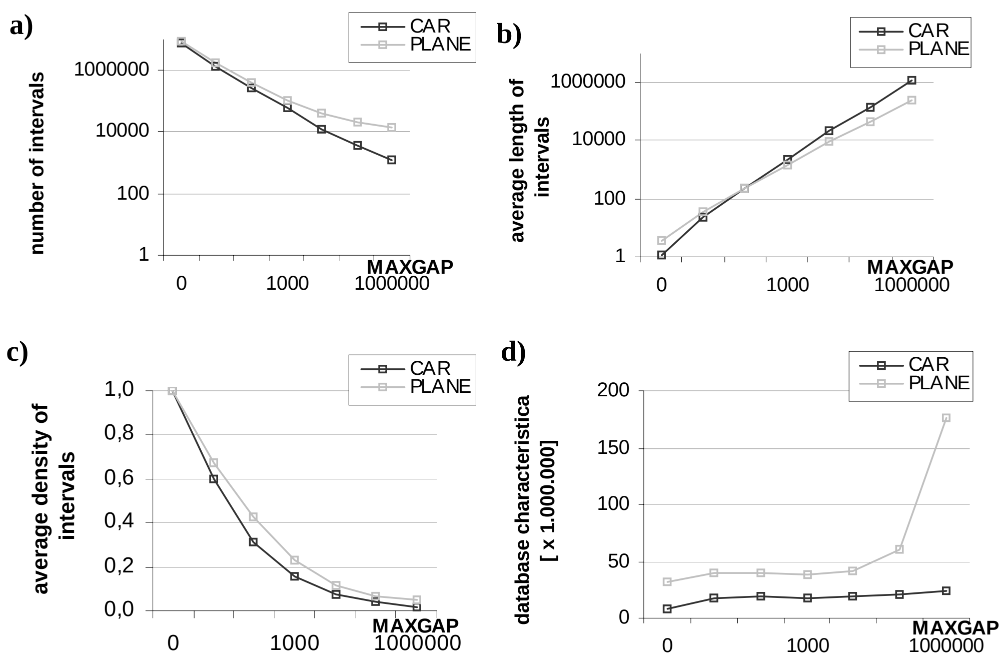


Figure 46: Transformed databases
a) N_{trans}, **b)** L_{trans}, **c)** D_{trans}, and **d)** $C_{trans}=N_{trans}*L_{trans}*D_{trans}$

tiny part out of the middle of a BLOB (e.g. 10 bytes), the system does not have to go through all the pages, starting from the beginning.

On the other hand, if we want to get all the information included in a BLOB the number of logical reads linearly increases with the BLOB size (cf. figure 45b) . This figure also depicts the fact, that if you have to read the whole BLOB, it is advantageous to read with a high chunksize (number of bytes per read). For instance, reading one million bytes of one BLOB with a chunksize of 100 needs 70 times more logical reads than doing it with a chunksize of 10.000 and still 50 times more physical reads.

All following experiments have been executed with a chunksize of 30.000.

4.3 Evaluation of the static properties of the X-RI-Tree

4.3.1 Characteristica of the transformed Databases

Obviously, the characteristica of a transformed database strongly depends on the *MAXGAP* parameter. As this parameter determines which small intervals are

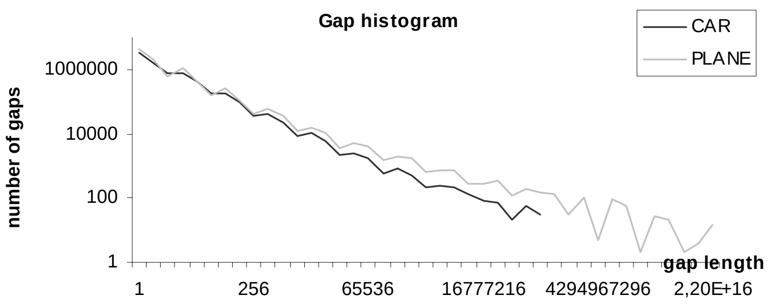


Figure 47: Gap histogram

grouped to large grey intervals, it is not surprising that with increasing *MAXGAP* parameter, the number of intervals N_{trans} in the transformed database decreases[1]. It is also obvious, that their average density D_{trans} decreases, owing to the fact, that you include gaps into the long grey intervals. This is also the reason for the increasing of the average length L_{trans} of the grey intervals. What you might expect is that the product $\mathrm{N}_{\mathrm{trans}} \cdot \mathrm{D}_{\mathrm{trans}} \cdot \mathrm{L}_{\mathrm{trans}}$ is independent of the *MAXGAP* parameter. But this is not true as the following unequation clarifies.

$$\sum_{i=1}^{N_{trans}} l_{I_i} d_{I_i} \neq \left(\frac{\sum_{i=1}^{N_{trans}} l_{I_i}}{N_{trans}} \times \frac{\sum_{i=1}^{N_{trans}} d_{I_i}}{N_{trans}} \times N_{trans} = L_{trans} \times D_{trans} \times N_{trans} \right)$$

The extreme rise in the C_{trans} *PLANE curve* at very high *MAXGAP* parameters (cf. figure 46) is owing to the fact, that we have a few extremely long parts (e.g. wings of the plane) but still a lot of short parts with high density.

4.3.2 Gap and interval histograms

As shown in [Gae95] the number of intervals generated via a space-filling curve out of a real-world-object, mainly depends on the surface, the shape of the object and the granularity of the underlying grid approximation. Unfortunately, there is nothing mentioned about the distribution of the intervals or the corresponding gap distribu-

[1] The *PLANE* curve becomes flatter because we have almost 10.000 objects in the database and we need at least one grey interval for each object. If we want to pinpoint the same effect on the *CAR* data set, which consists of less than 200 objects, we have to increase the *MAXGAP* parameter even more.

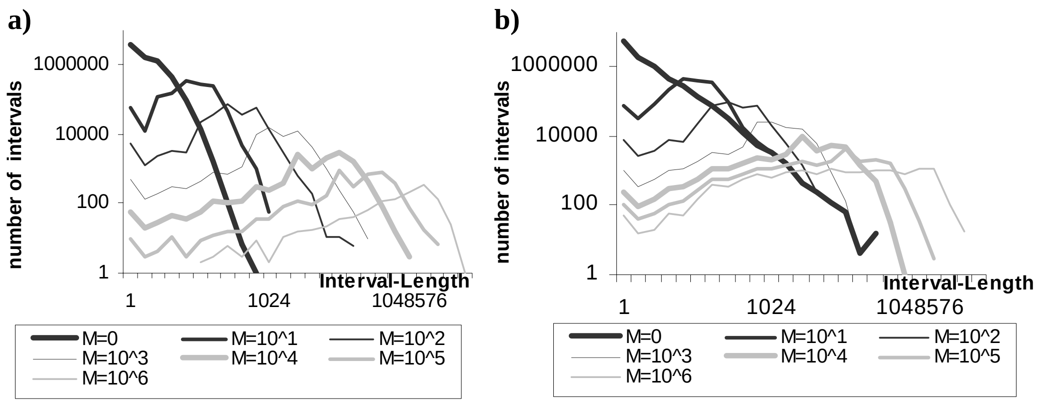


Figure 48: Interval length dependent on the *MAXGAP* parameter
a) CAR, **b)** PLANE

tion. In [Pöt 01] it is asserted, that the binary logarithms of fractal gaps typically obey an exponential distribution. Furthermore, histograms on fractal gaps show local peaks at whole multiples of three (=the original data dimension *d*), i.e. at gap lengths around $2^{3 \cdot k}$, $k \geq 0$. This behavior is caused by the fact that many gaps represent empty cube-like (3D) regions at the boundary of spatial objects. Figure 47 supports the assertion made in [Pöt 01]. Figure 48 depicts the interval distribution. It can be seen that the bucket which includes most intervals is regularly increasing with increasing *MAXGAP*. The gap histograms dependent on the *MAXGAP* parameter are obvious because all gaps smaller than *MAXGAP* were used to form the grey intervals. On the other side, the gaps larger than *MAXGAP* are unused.

4.3.3 Storage Requirements

Although the storage complexity O(n/b) of the native RI-tree is optimal, it seems rather wasteful to spend a whole row in the *Ranges* table for a small interval. Figure 49a shows the different storage requirements for the *XRange* table with respect to the different organization approaches of the BLOBs. These experiments were carried out based on subsets of the original test data sets, comprising approximately 10% of the original data. As you can see, the bit-oriented approach is very bad for high *MAXGAP* values, but it is better than the offset-oriented approach[1] when using small *MAXGAP* values. The final approach combines the advantages of both

[1] We used the *offset-oriented-byte* approach throughout the experiments presented in this chapter.

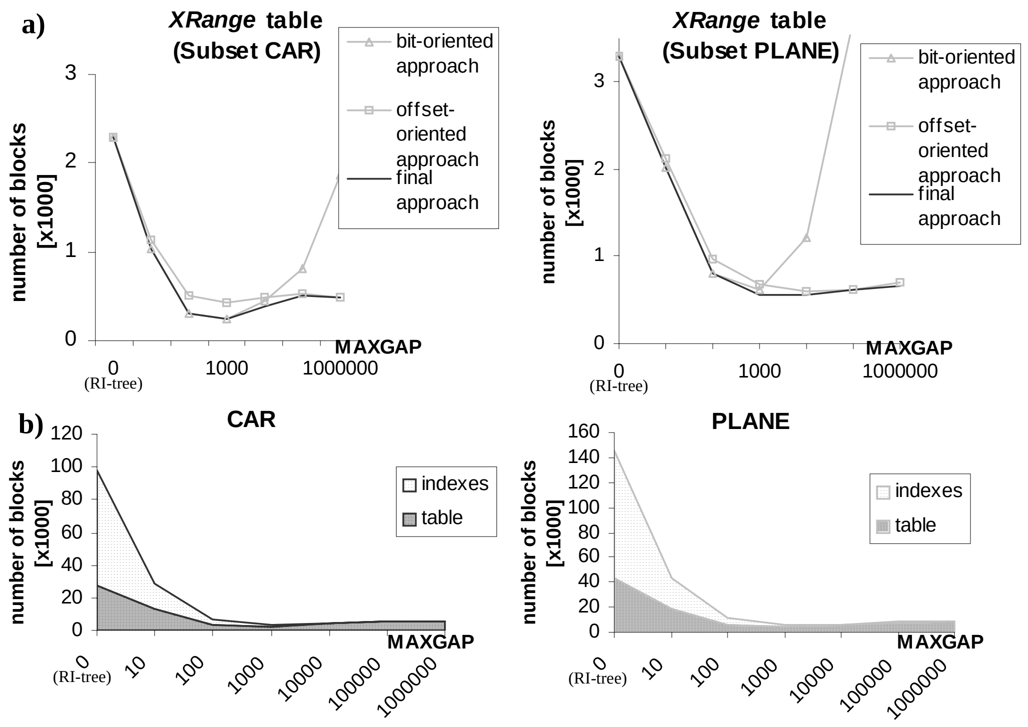


Figure 49: Storage Requirements for the *XRange-Table*
a) different approaches (subset), **b)** final approach (all objects)

and can never be worse than the original RI-tree. The curve of the final approach increases a little bit when the *MAXGAP* parameter exceeds values greater than 1.000. This is due to the fact, that in this case, most intervals follow the offset-oriented approach, whereas in the case of small *MAXGAP* parameters, they are organized according to the bit-oriented approach. In Figure 49b the storage requirements for the sum of the *upper*- and *lowerIndexes* as well as for the complete *XRange* table are depicted. In the case of small *MAXGAP* parameters, the number of disk blocks used by the *upper*- and *lowerIndexes* dominate the number of disk blocks for the *XRange* table. With increasing *MAXGAP* parameters the number of disk blocks for the indexes dramatically decreases (cf. figure 46a) and at high parameter values they yield no significant contribution any more to the overall sum of used disk blocks.

Lesson 1

With a well parametrized X-RI-tree you can improve the storage behavior at least by an order of magnitude compared to the RI-tree.

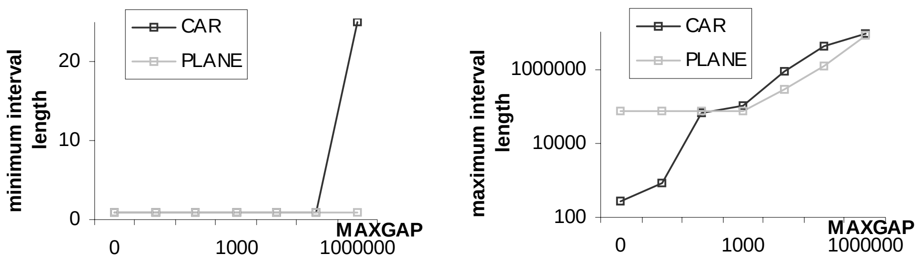


Figure 50: Minimum and maximum length of interval

4.3.4 Miscellaneous

4.3.4.1 Minimum and maximum length of intervals

As depicted in figure 50, the maximum length of the intervals increases with the *MAXGAP* parameter. Note, that the minimum length does not likewise. Although using great *MAXGAP* values, there are still intervals of length 1.

4.3.4.2 Black intervals

Figure 51 illustrates, that the number of grey intervals with maximum density[1] decreases faster than the number of all grey intervals, with increasing *MAXGAP* parameter. The number of long black intervals, meaning black intervals longer than *MAXGAP*, decreases even faster than the number of all black intervals. So with increasing *MAXGAP* parameter there remain only a very few long black intervals, which are useful for gap optimization (cf. section 3.8).

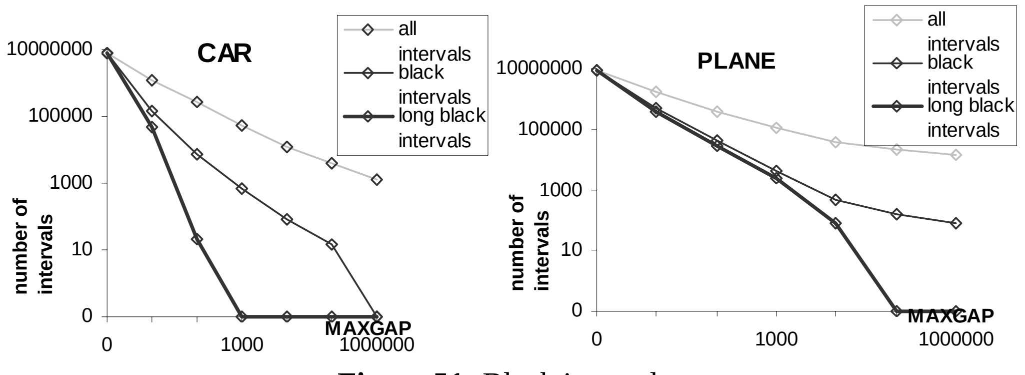


Figure 51: Black intervals

[1] Recall that we call $I_{grey} = ((l_{grey}, u_{grey}), <s_1>)$ also a black interval.

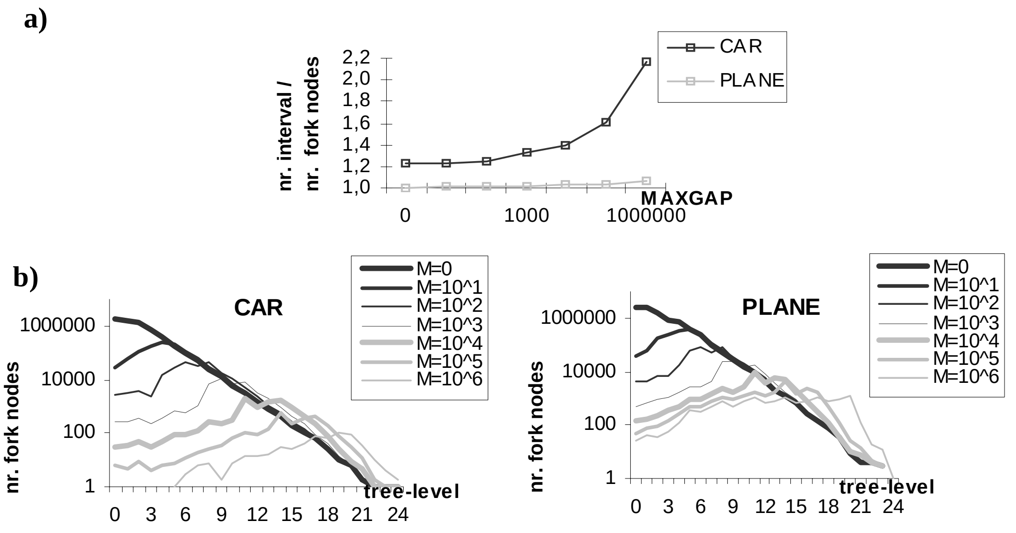


Figure 52: Fork nodes
a) number of different fork nodes / number of intervals, **b)** fork node level

4.3.4.3 Fork nodes

Figure 52 depicts in what way *fork node* properties change with changing *MAX-GAP* parameters.

In figure 52a it is shown, that the average number of intervals which share the same fork node is much higher in the case of the *CAR* data than it is in the case of the *PLANE* data. This is rather obvious, as the *PLANE* space is $2^{(42-33)}$ times larger than the *CAR* space, but comprising only marginally more intervals. Note that two grey intervals sharing the same fork node belong to two different objects and that these two different objects thus interlace each other. As we will see in the next section the average number of collisions for one part is much higher on the *CAR* data set than it is on the *PLANE* data set. Figure 52b is quite similar to figure 48, thus indicating the connection between fork node level and interval length. This is due to the fact, that an interval of length l cannot belong to a fork node n_f which resides on a level smaller than $log_2 l$. Note, that there still exist fork nodes on the leave-level, even if rather high *MAXGAP* values are applied.

4.3.4.4 Upper- and lowerIndexes

Figure 53 illustrates that we can save one level in the B$^+$-directory, if we increase the *MAXGAP* parameter. It is interesting that the *MAXGAP* parameter is always the

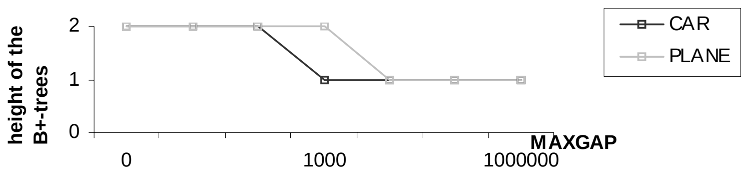


Figure 53: Height of the B$^+$-directory of the *upper*- and *lowerIndex*

same there, where the height of the B$^+$-directory decreases, where we need a minimum of secondary storage (cf. figure 49), and where we get the best response time (cf. figure 54), e.g. *1.000* on the *CAR* data and *10.000* on the *PLANE* data.

4.4 Evaluation of the dynamic properties of the X-RI-Tree

In this section, we want to turn our attention to the different facets related to the query response behavior of the X-RI-tree. In subsection 4.4.1 we concentrate on *collision queries*, whereas in the subsequent subsection, we consider *box volume queries,* or more generally spoken dynamically created query objects.

4.4.1 Collision Queries

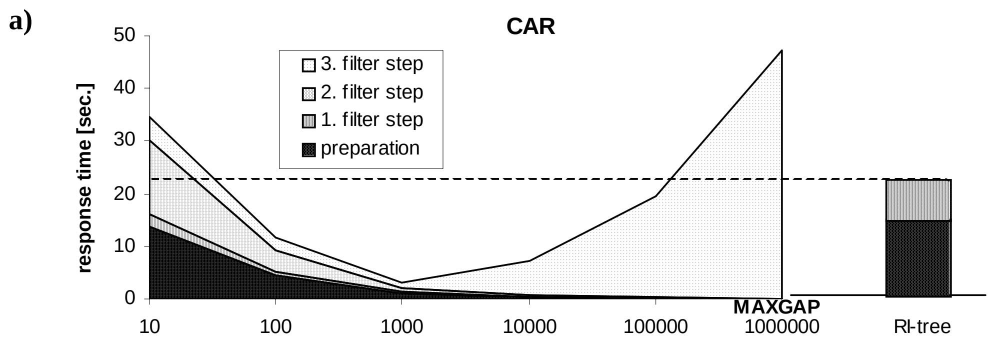
4.4.1.1 Introduction

All figures presented in this subsection depict the average result yielding from collision queries, where we have taken every part from both test data sets *CAR* and *PLANE* as query objects and asked, which parts in the associated database are colliding with them. We first discuss the overall runtime behavior and the correlated disk accesses. Furthermore, we address the issue of main memory session footprint and number of tested candidates. This subsection is closed with a few general remarks on miscellaneous facets[1].

4.4.1.2 Response time

In figure 54, it is shown in what way the overall response time depends on the $MAXGAP_{database}$ parameter. If we use small *MAXGAP* parameters, we still need a lot

[1] We use $MAXGAP_{query}$ to denote the MAXGAP parameter belonging to the query object and $MAXGAP_{database}$ to denote the MAXGAP parameter belonging to the database. If not explicitly stated, these two values are equal.



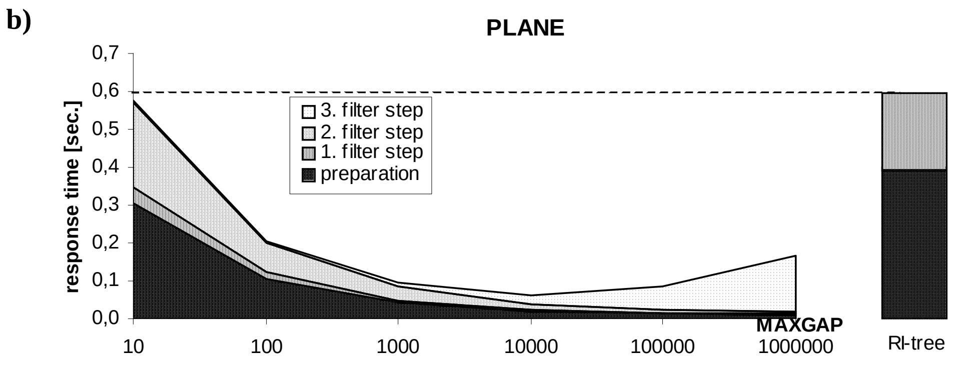


Figure 54: Response time on collision queries
a) CAR, **b)** PLANE

of time for the determination of all transient join partners, i.e. the preparation step. On the other hand, using big values leads to an expensive third filter step. Fortunately, using MAXGAP parameters in the middle leads to a good query response time.

Lesson 2

With a well parametrized X-RI-tree you can improve the response time of collision queries by an order of magnitude compared to the RI-tree.

4.4.1.3 Disk accesses

Analyzing the number of disk accesses (cf. figure 55), reveals, that the number of logical reads is smaller in the case of the RI-tree, than it is in the case of the X-RI-tree with a small *MAXGAP* parameter (e.g. *MAXGAP=10*). This is because the X-RI-tree does not benefit as much as the RI-tree from the *gap optimization* and that conse-

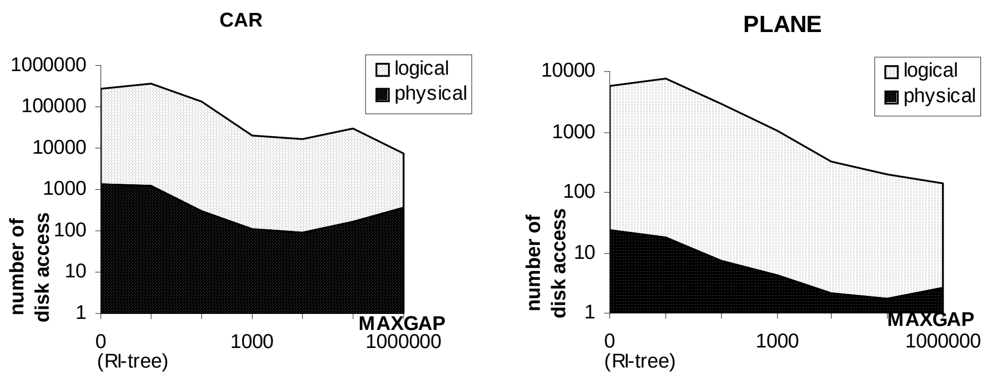


Figure 55: Logical and physical disk access

quently, we have more join partners in the transient tables *left*- and *rightNodes* (cf. figure 56) yielding to more cached B^+-directory lookups.

4.4.1.4 Main memory footprint

As mentioned in the last subsubsection, we have a lot of records in the transient tables *left*- and *rightNodes*. Fortunately, this number decreases hand in hand with the number of entries in *LQI* (*L*ist of *Q*uery *I*ntervals), when using large *MAXGAP* parameters. Figure 56 depicts, that the number of entries in LC_3 (*L*ist of *C*andidates for the *3*rd filter step) is neglectable and does not yield any significant contribution to the overall main memory footprint. Thus, we might have dispensed with the LC_3MAX parameter introduced in subsection 3.5.2.

Lesson 3

With a well parametrized X-RI-tree you can dramatically reduce the session footprint.

4.4.1.5 Tested candidates

Figure 57 illustrates the number of candidate pairs of query and database intervals and the number of the corresponding tests, which were actually carried out in the second and third filter step.

In the *second filter step* the number of these candidate pairs rapidly decreases with increasing *MAXGAP*, although the number of candidate object IDs increases (cf. figure 58). Thus the redundancy reduction dominates the effect of falsely detected objects. At low *MAXGAP* values we have to test only a fractional amount of candidate

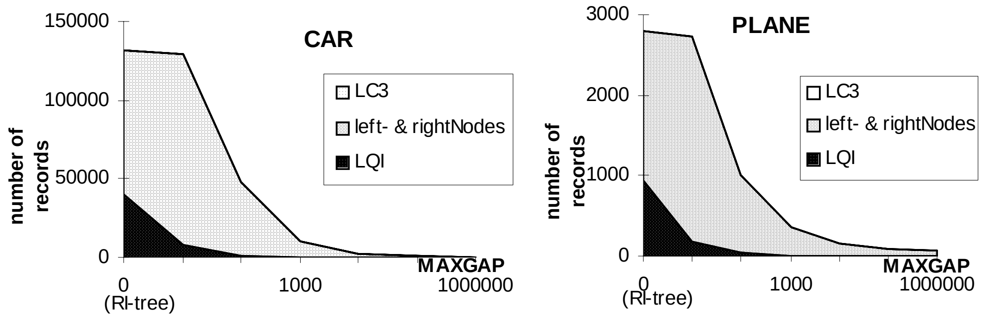


Figure 56: Number of entries in the transient tables

pairs, as the *fast grey test* works very successfully with this parametrization (see also figure 58). Consequently, there is only a relative small number of candidate pairs left for the third filter step. With increasing *MAXGAP* values this test looses effectiveness.

In the *third filter step* the number of both candidate pairs and corresponding tests do not vary as much as in the second step. It is difficult to make a profound statement

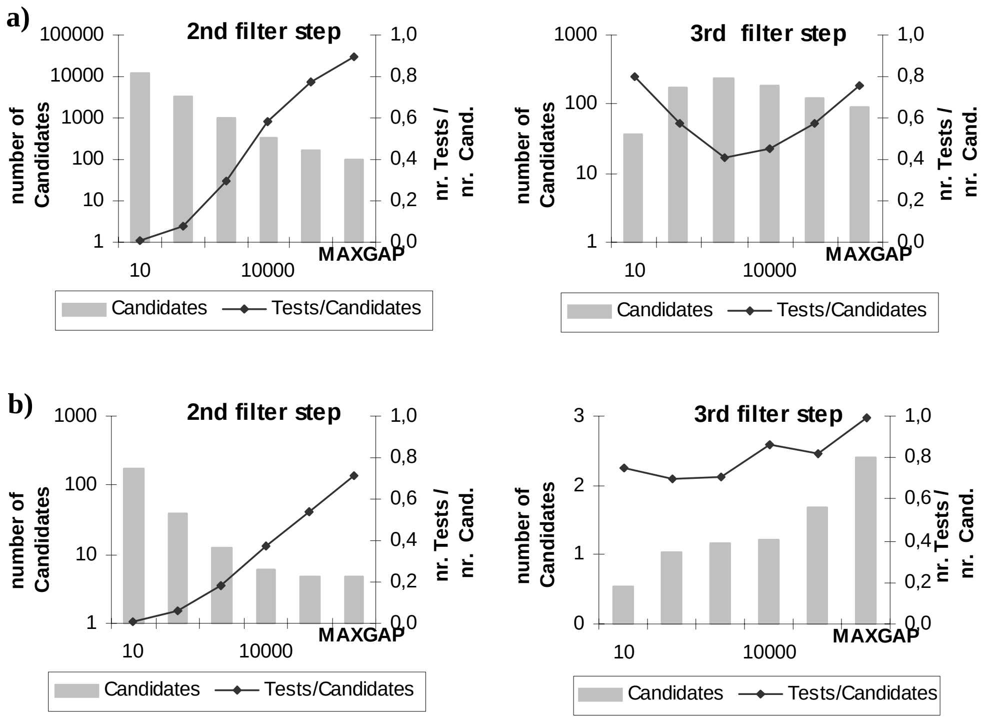


Figure 57: Tested candidate pairs of query and database intervals
a) CAR, **b)** PLANE

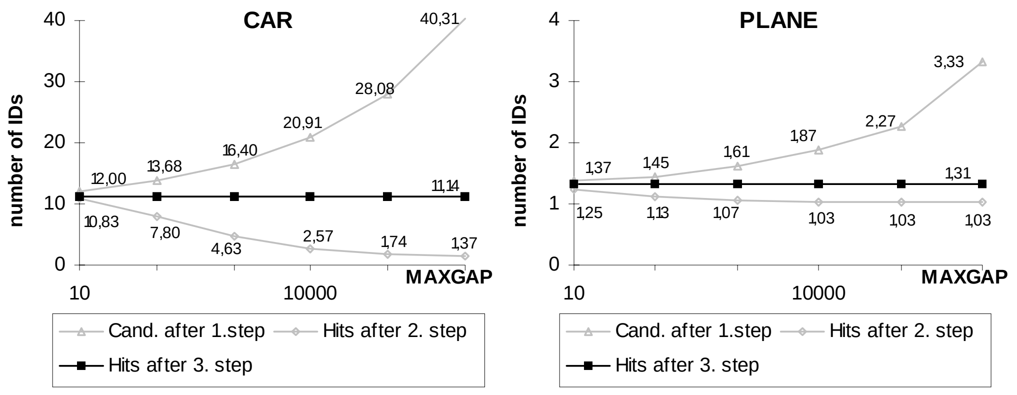


Figure 58: Candidate IDs and result sets

in this case, but we can still see, that in the case of the best response time on the *CAR* data (i.e. *MAXGAP* equals *1.000*), we only have to test 40% of all candidates and thus benefit in the third step as well from the *skipping principal* introduced in section 3.6.

In figure 58, it is illustrated, that at small *MAXGAP* values the number of different objects IDs resulting from the first filter step is only marginally higher than the number of different IDs in the final result set. Likewise, the number of detected hits in the second filter step is only marginally smaller. With increasing *MAXGAP* values the two curves disperse.

4.4.1.6 Miscellaneous

The size of the parts in the *PLANE* data set vary very much. We have a lot of small parts and only a few very large ones. In the case of the *CAR* data this peculiarity is far less distinctive. As large query parts produce a large number of query intervals, it is obvious that the size part correlates with the response time. In figure 59a it is shown that for most parts out of the *PLANE* data set, the X-RI-tree outperforms the RI-tree *“only”* by a factor of 2.9, whereas there are some parts, for which this factor is higher than 100. In figure 59b it is illustrated that the high response time of the RI-tree is mainly owing to the high preparation time, which naturally correlates with the number of intervals of the query object. In the case of the RI-tree we have to wait for more than five minutes for some collision queries in order to get the response. On the other hand, using the X-RI-tree yields almost interactive response time for all collision queries (cf. figure 59).

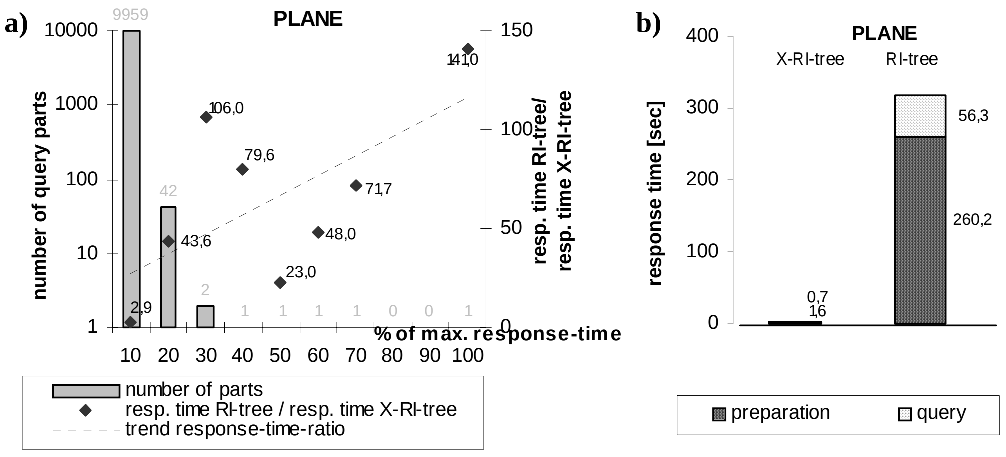


Figure 59: Response time
a) |resp. time RI-tree | / |resp. time X-RI-tree|, **b)** maximum response times

Lesson 4

If the overall response time of the RI-tree is extremely high, we particularly benefit from the X-RI-tree.

4.4.2 Box queries

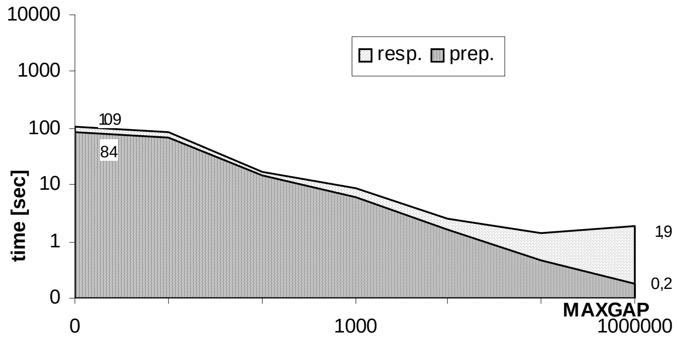
In this subsection we discuss *box volume queries*. All remarks in this subsection can be generalized to other dynamically created query objects. The tests have been carried out only on the *PLANE* data set because we focus on the decompositioning of the boxes rather than on the underlying database. The preparation step comprises two steps, the decompositioning of the boxes according to the algorithm of figure 6 and the creation and registration of the transient join partners.

In figure 60 it is shown that the decompositioning of the boxes into black interval sequences takes very long, even if the box size is relatively small. With increasing box size the preparation time increases as well[1]. Note, that with an X-RI-tree with a high *MAXGAP* value, the preparation time can be reduced by more than two orders of magnitude leading to a much better overall response time.

In a last experiment we carried out box queries on databases where the two values *MAXGAP*$_{query}$ *and MAXGAP*$_{database}$ might differ. The results in figure 61 show, that

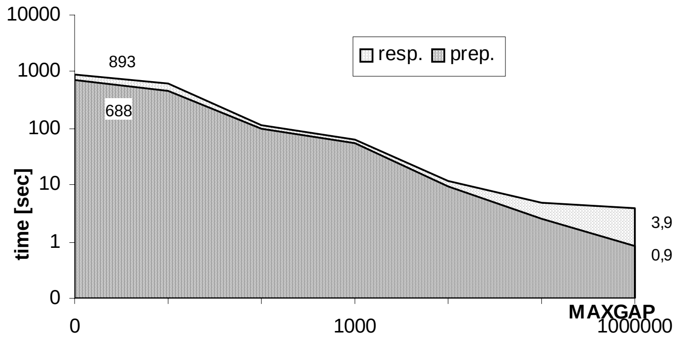
[1] The number of intervals linearly depend on the surface of the query object [Gae95].

a)



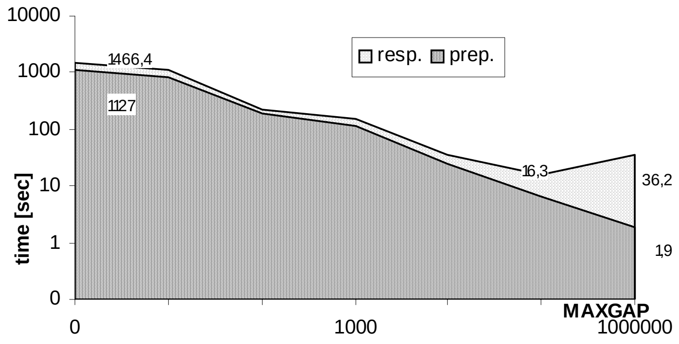
box size equals
0,00002% of data space
yielding to
0.03% selectivity

b)



box size equals
0,003% of data space
yielding to
0.1% selectivity

c)



box size equals
0,008% of data space
yielding to
1.0% selectivity

Figure 60: Box queries on the *PLANE* data (preparation and response time)

at $MAXGAP_{query}$ values of 1.000.000 and at a $MAXGAP_{database}$ value of 10.000, we get the best response time. The optimal value for $MAXGAP_{database}$ is the same as the one, we evaluated during the collision tests in the last subsection. As a rule-of-thumb we can say that it is good to use high $MAXGAP_{query}$ values for the dynamically created query objects, so that the preparation time is reduced, whereas for the database we can use $MAXGAP_{database}$ values which lead to an optimal storage exploitation and furthermore to an optimum response time for collision queries.

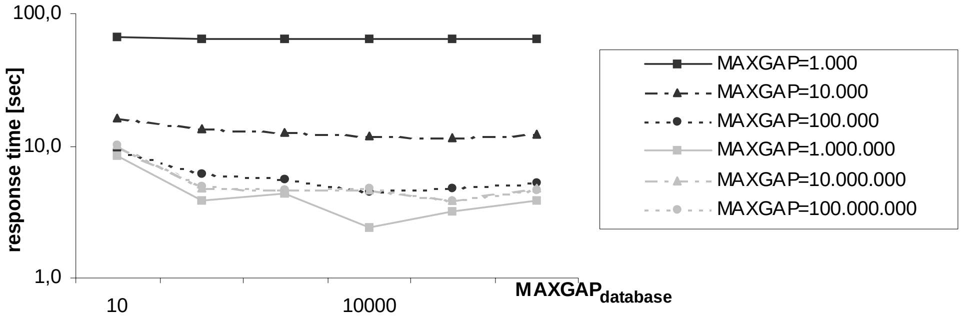


Figure 61: Box queries with different query and database *MAXGAP* parameters

Lesson 5

In the case of dynamically created query objects, we particularly benefit from the X-RI-tree compared to the RI-tree.

4.5 Summary

We have seen, that the X-RI-tree works well on our test data sets *CAR* and *PLANE*. With $MAXGAP_{CAR} = 1.000$ and $MAXGAP_{PLANE} = 10.000$ we outperform the optimized version of the RI-tree with respect to storage exploitation and response time by an order of magnitude. Using the X-RI-tree gains the following advantages over the RI-tree:

- Only a fractal of *secondary storage* is occupied.
- The *session footprint* is less. Therefore the X-RI-tree is better qualified for a multi-user environment.
- The query *response time* is much better.
- The concept of grey intervals is particularly useful for *dynamically created query objects*, which are resolved into grey interval sequences in a top-down approach.

Chapter 5
Incorporation of Spatial Objects in an ORDBMS

A lot of traditional database servers have evolved into an Object-Relational Database Management System (ORDBMS). This means that in addition to the efficient and secure management of data ordered under the relational model, these systems now also provide support for data organized under the object model. Object types and other features, such as large objects (LOBs), external procedures, extensible indexing and query optimization, can be used to build powerful, reusable server-based components.

In this chapter we pursue a twofold plan.

In the first two sections, which are more or less a reformulation and summary of what can be found in [Pöt 01], we want to talk generally about extensible OBRDMS. In section 5.1, we shortly introduce the object-relational data model among other approaches and talk about abstract data types. In section 5.2, we turn our attention to the extensible query language, enabling the declarative embedding of abstract data types within the built-in optimizer and query processor.

In section 5.3 we describe in detail how the X-RI-tree can be integrated into the extensible indexing framework of the Oracle8*i* server, using the concepts introduced in the foregoing sections.

This mix of Oracle specific and more general information, should enable the reader to implement the X-RI-tree on top of any other ORDBMS such as IBM DB2 or Informix IDS/UDO.

5.1 Extensible Data Model

In this section, we shortly point out, why the object-relational data model is preferable to other existing approaches. Furthermore, we show how abstract data types can be integrated into object-relational servers.

5.1.1 Classification of Data Models

Stonebraker and Brown [SB 98] considered the complexity of the stored data and submitted queries to classify some common data models. Four major groups were identified (cf. figure 62):

- File system
- Relational DBMS (RDBMS)
- Object-oriented DBMS (OODBMS)
- Object-relational DBMS (ORDBMS)

We focus on object-relational database management systems, as they combine the advantages of both, the object-oriented and the relational data model. Their extensible design enables us to integrate new access methods, e.g. the X-RI-tree. In addition, the practical impact of ORDBMSs is very strong as object-relational functionality has been added to most commercially available relational database servers, including Oracle [Doh 98], IBM DB2 [CCN+ 99], and Informix IDS/UDO [Bro 01].

	Simple data	**Complex data**
Simple queries	File system	Object-oriented DBMS
Complex queries	Relational DBMS	Object-relational DBMS

Figure 62: Classification of data models

5.1.2 Abstract Data Types

DDL statements like *CREATE*, *ALTER*, and *DROP* have been extended in SQL:1999 [SQL 99] to support the declaration and implementation of *abstract data types* [Bro 01] [CZ 01], often also referred to as *object types*. According to [Ora 99b] an object type is a schema object with three kinds of components:

- A *name*, which identifies the object type uniquely within that schema.
- *Attributes*, which model the structure and state of the real-world entity. Attributes can be built-in types or object types.
- *Methods* of an object type are functions or procedures that are called by the application to model the behavior of the objects. Methods can be stored in the database, which is preferable for data-intensive procedures and short procedures that are called frequently.

5.2 Extensible Query Language

Most ORDBMSs, including Oracle [Ora 99a] [SMS+ 00], IBM DB2 [IBM 99] [CCF+ 99], or Informix IDS/UDO [Inf 98] [BSSJ 99], provide extensibility interfaces in order to enable database developers to seamlessly integrate custom object types and predicates within the declarative DDL and DML. The resulting custom server components, built on these interfaces, are called *data cartridges*, *database extenders*, and *data blades*, respectively.

5.2.1 Extensible Indexing

In order to guarantee an efficient evaluation of user-defined predicates, the extensibility services of the ORDBMS offer a conceptual framework to supplement the functional evaluation of user-defined predicates with index-based lookups. This is in accordance with the extensible indexing frameworks proposed by Stonebraker [Sto 86], enabling developers to register custom secondary access methods at the database server in addition to the built-in index structures. An object-relational *indextype* encapsulates stored functions for creating and dropping a custom index and for opening and closing index scans. Figure 63 shows some basic indextype methods, invoked by extensible indexing frameworks. Furthermore, there exist additional functions to support query optimization. Most ORDBMSs support both rule-based and cost-based query optimization, whereby the cost-based approach is preferable to the rule-based approach when referencing user-defined methods as predicates (cf. [BO 99], [HS 93]). Therefore most ORDBMSs provide cost-based functions, which are invoked by the extensible indexing framework (for a list of such functions have a look at [Pöt 01] or [Ora 99a]).

Some main advantages of extensible indexing frameworks are:

- The maintenance and access of a custom index structure is completely hidden from the user, achieving thereby data independence.
- Any redundant index data remains consistent with the user data.
- The declarative paradigm of SQL is preserved.

5.3 Implementation of the X-RI-tree on top of Oracle 8*i*

In section 5.3 we describe how the X-RI-tree can be integrated into the extensible indexing framework of the Oracle8*i* server, using the concepts introduced in the foregoing sections.

5.3.1 Declarative integration of the abstract data type *TOIS*

As already mentioned, an interval sequence is a basic data type for temporal and spatial data, which can efficiently be managed with the X-RI-tree. In order to seam-

Function	**Task**
index_create(), index_drop()	Creates and drops a custom index.
index_open(), index_close()	Opens and closes a custom index.
index_start(), index_fetch()	Starts an index scan. Fetches the next record from the index that meets the query predicate.
index_insert(), index_delete(), index_update()	Adds, deletes, and updates a record of the index.

Figure 63: Methods for extensible index definition and manipulation

lessly embed it into an ORDBMS we need an abstract data type for such an interval sequence. We call this type *TSpatialObject*, or a bit more general *TOIS* (*T*ype of *O*bject *I*nterval *S*equence). Instances of this custom object type are stored as elements of relational tuples. Figure 64 depicts some of the required object-relational DDL statements in pseudo SQL. By using the functional binding of the user-defined predicate *INTERSECTS*, object-relational queries can be expressed in the usual declarative fashion (cf. figure 64).

5.3.2 Extensible Indexing

In subsection 5.2.1 we discussed from a general point of view the extensible indexing framework, whereas in this subsection we want to show exemplary, how the X-RI-tree can be embedded into the Oracle 8*i* extensible indexing framework. We

```
// Type declaration

CREATE TYPE INTERVAL AS OBJECT (lower NUMBER, upper NUMBER);
CREATE TYPE INTERVAL_TABLE AS TABLE OF INTERVAL;
CREATE TYPE TOIS AS OBJECT (
     intervals INTERVAL_TABLE,
     MEMBER FUNCTION intersects (Aois TOIS) RETURN BOOLEAN
);

// Type implementation
// …

// Functional predicate binding

CREATE OPERATOR INTERSECTS (Aois1 TOIS, Aois2 TOIS)
RETURN BOOLEAN
BEGIN RETURN Aois1.intersects(Aois2); END;

// Table SpObjs (SpatialObjects) definition

CREATE TABLE SpObjs (id NUMBER PRIMARY KEY, SpObj TOIS);
```

```
// Intersection query

SELECT id FROM SpObjs
WHERE INTERSECTS(SpObj, :q) = TRUE;
```

Figure 64: Object-relational DDL and DML statements for *TOIS*

```
CREATE TYPE XRItree_im AS OBJECT (
   // attributes
   crs1Step      NUMBER,           // cursor for the first filter step
   crs3Step      NUMBER,           // cursor for the third filter step
   XRIMetadata   TXRIMETADATA,     // metadata for this index object (resolution, DB_MAXGAP)
   LC3           TLC3,             // List of candidates for the third filter step
   LC3Cnt        number,           // counter for the above collection set
   LQI           TLQI              // List of the input query sequence
   F2ActID       number,           // ID of actual part in the second filter step
   F2ActIDStart  number,           // position of the first record of this part stored in F2ResTab
   LastHitID     number,           // Last ID, added to the result set
   XRangeTab     varchar2(100),    // Name of the XRange-Table
   // ODCII-Functions
   STATIC FUNCTION ODCIIndexCreate     ...,
   STATIC FUNCTION ODCIIndexStart      ...,
   MEMBER FUNCTION ODCIIndexFetch      ...,
   MEMBER FUNCTION ODCIIndexClose      ...,
  ....
   // Additional functions
  ....
);
```

```
CREATE INDEXTYPE XRItree
FOR intersects (TOIS, TOIS)
USING XRItree_im;
```

Figure 65: Indextype *XRItree*

will mention a few technical details in connection with index creation and intersection queries.

5.3.2.1 Indextype XRItree

The first thing we have to do, is to encapsulate the X-RI-tree within the custom indextype *XRItree*. Figure 65 gives a rough impression of what this means.

First, the *ODCIIndex interface* (*O*racle *D*ata *C*artridge *I*nterface *I*ndex), which is a set of index definition, maintenance and scan routine specifications, has to be implemented. This interface does not refer to a separate schema object but rather to a logical set of documented method specifications (for detailed documentation see [Ora 99a]). To accomplish this task, you can add different attributes and additional functions to your indextype.

// Index creation

CREATE INDEX *spatial_idx* ON *SpObjs* (SpObj) INDEXTYPE IS *XRItree* parameters ('33; 10000');

Figure 66: Creation of a custom index on spatial data

Secondly, a new indextype has to be created by specifying the list of operators supported by the indextype and referring to the type that implements the index interface. In figure 65 the DDL statement for defining the new indextype *XRItree*, which supports the *intersects* operator and whose implementation is provided by the type *XRItree_im*, is depicted.

5.3.2.2 Create index statement

In figure 66 it is shown that we can create an index *spatial_idx* on the *SpObj* attribute on the *SpObjs* table by submitting the usual DDL statement. We can append to this create-statement a parameter-clause, specifying the *resolution* and the $MAXGAP_{database}$ parameter. The *ODCIIndexCreate* method is called when a CREATE INDEX statement is issued. Upon invocation, any parameters specified in the parameter-clause are passed in along with a description of the index. Based on this information we can start grouping the black intervals together to grey intervals, which is the essential part of the implementation of the *ODCIIndexCreate* function in the case of the X-RI-tree.

5.3.2.3 Select statement

After having issued an intersection query, as illustrated in figure 64, an index scan is executed, which is specified through three routines, *ODCIIndexStart*, *ODCIIndexFetch*, and *ODCIIndexClose*. These routines perform initialization, fetch rows (essentially row identifiers) satisfying the predicate, and clean-up once all rows are returned.

ODCIIndexStart: ODCIIndexStart is invoked to initialize any data structures and start an index scan. Since the index and operator related information are passed in as arguments to ODCIIndexStart and not to the other index scan routines (ODCIIndexFetch and ODCIIndexClose), any information needed in the later routines must be saved. This is referred to as the *state* that has to be shared among the index scan routines. Oracle RDBMS will pass the *SELF* value to subsequent ODCIIndexFetch and ODCIIndexClose calls which can then be used to access the relevant context information.

In the case of the X-RI-tree, we use the ODCIIndexStart routine to fill the transient *left*- and *rightNodes* tables and then post the query statement of the first filter step. Furthermore, we save all necessary state attributes (cf. attribute list in figure 65).

ODCIIndexFetch: ODCIIndexFetch returns the *next* row identifier of the row that satisfies the operator predicate. The operator predicate is specified in terms of the operator expression (name and arguments) and a lower and upper bound on the operator return values. Thus, an ODCIIndexFetch call returns the row identifier of those rows for which the operator return value falls within the specified bounds. A NULL is returned to indicate the end of an index scan. The fetch-method supports returning a batch of rows in each call. The state returned by ODCIIndexStart or a previous call to ODCIIndexFetch is passed in as an argument.

In the case of the X-RI-tree, the second and the third filter step are executed interleaved as shown in figure 38. Note, that we might benefit from the *fast grey test* because we can deliver results without accessing the BLOB. Therefore, we might accomplish an ODCIIndexFetch call without executing the third filter step at all. As the introductory example of figure 30 shows, the result set has not to be ordered by the object ID attribute, as we return results as soon as they are available. Nevertheless, we have to save the transient table LC_3 (cf. figure 65) for further ODCIIndexFetch calls, where we might have to execute the SQL statement of the third step. If we have already executed this statement, we save the corresponding cursor for further ODCIIndexFetch calls.

ODCIIndexClose: ODCIIndexClose is invoked when the cursor is closed or reused. In this call the indextype can perform any clean-ups, etc. The current state is passed in as an argument.

In the case of the X-RI-tree, we just close open cursors.

5.4 Conclusions

In this chapter, we have shown, that we can seamlessly integrate the X-RI-tree into a modern extensible ORDBMS. It is beneficial, that the second filter step of the X-RI-tree, yields very fast first results, without knowing the complete result set.

In order to enable the optimizer of the database system to place the X-RI-tree at its optimal position in the query execution plan, we still have to develop a cost model for the X-RI-tree. This is deferred to future work (cf. chapter 6), but certainly will not pose any insuperable problems, as a cost model for the RI-tree has already been developed (cf. [Pöt 01]) and can serve as a guideline.

Chapter 6
Conclusions

In this chapter we first turn our attention to a list of open problems. For a lot of these problems, there already exist methods of resolution, which have not yet been implemented and evaluated. Nevertheless, we will sketch the rough ideas, so that others can take these thoughts as a starting point for future elaborations. At the end of the last chapter we have already pursued this approach, where we suggested that it is a good idea to develop a cost model for the X-RI-tree, based on the available cost model for the RI-tree.

In section 6.2 we summarize the work presented in the foregoing chapters, putting emphasize on the advantages of the X-RI-tree.

6.1 Future work

In this section, we list a few open problems and shortly outline solutions to the posted problems, but no detailed elaboration and verification.

In subsection 6.1.1 we introduce a new auspicious idea called, *self-adapting indexing*, where we try to combine the strengths of both, linear scan and (X-)RI-tree. We will see, that we already used this concept to some extent in the case of the X-RI-tree, without being aware of it. In subsection 6.1.2 we write down a few thoughts about a constructive computation method of an optimum *global MAXGAP* parameter, whereas in the following subsection we glance at optimum *local MAXGAP* parameters combined with the idea of integrating bounding boxes into the *upper*- and *lowerIndexes*. This aims at minimizing the candidates for the second and third filter step. In the last subsection, we shortly summarize our annotations about future work.

6.1.1 Self-adapting indexing

As having seen in the foregoing chapters, the transformed database contains much less intervals, and furthermore, the fork nodes of these intervals reside much closer to the root. To put it another way, the tree levels close to the leaves of the virtual backbone are only sparsely occupied, although there are still some leaves with intervals registered at them (cf. figure 52). This raises the question, whether it is beneficial to use the full path from the root of the virtual backbone to the leaves in order to generate the necessary join partners. A lot of them will not contribute to the result set. Therefore, it might be much better to stop the process of generating join partners at a higher level of the tree and scanning the index from this point onward.

This principal seems to be useful not only for transformed databases, but it can also be applied to the RI-tree, especially if the underlying data spaces are sparsely occupied. For example, imagine the data space of an airplane (e.g. 100 m in each dimension). If you put a box volume query returning no results (e.g. Z-value > 50 m), it is desirable that the system recognizes this at an early stage of the query process.

We will shortly introduce four closely linked approaches, where we stop generating join partners, somewhere between the root and the leaves of the virtual backbone.

We call this new index method *S-RI-tree* (*S*can-*R*elation-*I*nterval-*tree*), or *XS-RI-tree* respectively. The first two approaches totally disregard the data stored in the

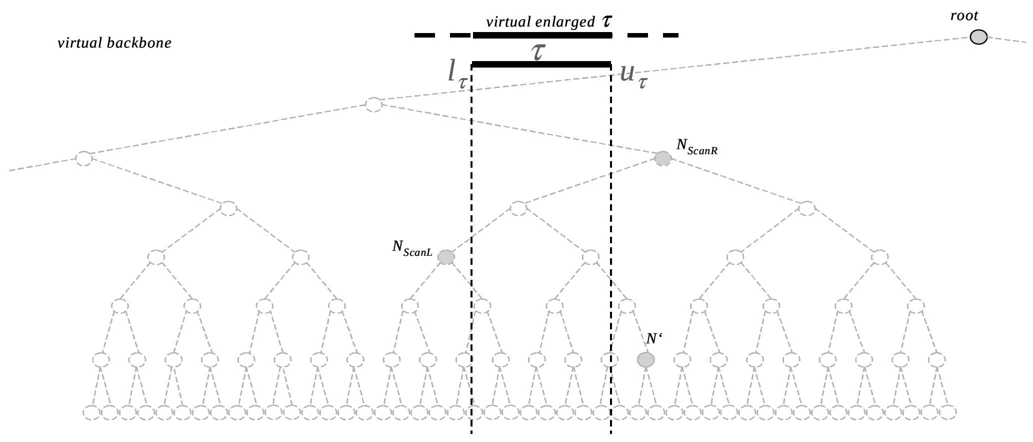


Figure 67: Determination of N_{SanL} and N_{SanR}, i.e. the truncation level

database for the determination of the scan nodes N_{SanL} and N_{SanR} (cf. figure 67), whereas the other two approaches take them into account.

6.1.1.1 S-RI-tree based on fixed truncation levels

As already described, we can stop collecting join partners if the nodes are beneath a fixed truncation level. In addition to the already collected join partners, we produce two more range queries ‘node BETWEEN N_{ScanL} AND l_τ AND upper $\geq l_\tau$’ and ‘node BETWEEN N_{ScanR} AND u_τ AND lower$\geq u_\tau$’. Note, that intervals registered at N' (cf. figure 67) can never contribute to the result set. Thus, we can no longer guarantee blocked output.

6.1.1.2 S-RI-tree based on query dependent truncation levels

Pötke presents in [Pöt 01] an architecture for the Database Integration of Virtual Engineering (DIVE) for existing Engineering Data Management systems (EDM). In this work, emphasis was put on the efficient embedding of the RI-tree into an off-the-shelf object-relational database system. A prototype of the DIVE system has been evaluated in cooperation with the Volkswagen AG, Wolfsburg [KMPS 01a] [KMPS 01b]. In these tests it emerged, that the RI-tree works more efficiently, if we virtually enlarge the query intervals by a constant factor (cf. figure 67). If we find nodes in this virtually enlarged area (e.g. N_{ScanL} and N_{ScanR}), we stop the collecting of further transient join partners and produce range queries as described in the last sub-subsection.

6.1.1.3 S-RI-tree based on data dependent truncation levels

We could compute the truncation level dependent on the histogram of the fork-node levels (cf. figure 52). If there are only few fork-nodes close to the leave level, we might use a higher truncation level. In this approach, we use (few) data in the database to assess the truncation level but neglect any information stemming from the query intervals (e.g. the position of the intervals).

6.1.1.4 S-RI-tree based on data and query dependent truncation levels

In [Pöt 01] effective and efficient methods to estimate the selectivity and the performance of interval intersection queries are presented. The developed appropriate I/O and CPU cost models can be invoked by common extensible indexing frameworks (cf. remarks at the end of chapter 5). These models immediately exploit the built-in statistics facilities of the database server, to cope with arbitrary interval distributions. For instance, histograms or quantiles can be employed to capture the data characteristics at any desired resolution.

In order to compute the optimum truncation level, we could pursue a similar approach. At each node, we could compute the number of remaining nodes, which still have to be visited in the preparation step. Each of these nodes is linked to at least one disk access of the leaves of the *upper*- and *lowerIndex*es.

On the other hand, we could estimate the number of blocks between the actual node and the interval boundary, based on built-in statistics. Comparing these two values helps us to decide, whether we should further randomly access the leaves of the *upper*- and *lowerIndex* by means of further collected transient join partners, or just scan contiguous leaf blocks of these relational indexes, taking into account, that we do not have blocked output.

Note, that random access to a leaf block is only beneficial with respect to I/O cost, if the preceding block gap is larger than the size of a disk block [Pöt 01].

6.1.1.5 Summary

We have shortly sketched four different approaches to determine the scan nodes N_{SanL} and N_{SanR}, i.e. the truncation levels, of the *S-RI-tree*. For the classification of the different approaches, we can use the information stored in the database and the information stemming from the query object. The result of this classification, applied to our four approaches is depicted in figure 68.

	no query information	**query information**
no DB information	*S-RI-tree based on fixed truncation levels*	*S-RI-tree based on query dependent truncation levels*
DB information	*S-RI-tree based on data dependent truncation levels*	*S-RI-tree based on data and query dependent truncation levels*

Figure 68: Classification of self-adapting indexing (2-dimensional)

Adjusting this *self-adapting indexing* approach to other access methods as for instance the *Linear Quadtree (Octree)* and the *Relational R-tree* could also be very beneficial and lead to a general concept.

Finally, we would like to stress, that the *X-RI-tree* and the *S-RI-tree*, have a few things in common, indicating, that the runtime behavior of the S-RI-tree is likely to be much better as the one of the *RI-tree*.

- In both cases the number of *transient join partners* can be dramatically reduced. In the case of the X-RI-tree, this is owing to the lower number of query intervals, whereas in the case of the S-RI-tree it is owing to the abbreviated virtual backbone.
- Note, that due to possible fruitless scans, both indexes do not guarantee *blocked output*, whereas the RI-tree does. As we have seen in the case of the X-RI-tree, this drawback does not seem to be grave.
- Both indexes use *linear scan*. In the case of the X-RI-tree this scan is confined to parts of one database object stored in a BLOB, whereas in the case of the S-RI-tree several database objects might be affected. Nevertheless, in both cases, we have to cope with the same problem: *What is the optimum level for switching between the RI-tree and a linear scan?* In this work we pursued the approach comparable to the one of the S-RI-tree based on fixed truncation levels. The variation of the *MAXGAP* parameter can be compared to the variation of the truncation level. Thus, a third dimension could be added to the classification of figure 68, dealing with the question, whether we use linear scan for one or all objects. The main difference between this classification of the X-RI-tree and the S-RI-tree is, that the *MAXGAP* parameter is fixed after the grey intervals have been inserted, whereas the truncation level has nothing to do with the insertion of intervals, but only with the query

response behavior. If we could access concurrently several databases, containing the same objects, but based on different *MAXGAP* parameters, we could also choose the optimum one, dependent on database and/or query data, at the beginning of the query process[1].

6.1.2 Mathematical approach for F_{MAXGAP}

In this work, we pursued a global empirical approach concerning the determination of an optimum $MAXGAP_{database}$ parameter. We applied the F_{MAXGAP} function to different *MAXGAP* parameters, yielding different transformed databases, which we empirically evaluated. To find an optimum *global MAXGAP* parameter, it could be helpful to investigate in more detail the interval histograms resulting from the different *MAXGAP* values. They could provide an indication of which parameter is the best one. We will now introduce a mathematical approach for F_{MAXGAP}, by means of which we could determine the resulting interval histograms, dependent on *MAXGAP*. Drawing conclusions from these interval histograms is a further unsolved task.

Note that the interval histograms correlate closely with the *distribution of the fork-node-levels* (cf. section 4.3). By intensifying this mathematical approach, you should be able to answer questions like: H*ow probable is it, to find all objects intersecting an interval query, if you neglect the last ten layers of the virtual backbone?* Furthermore, this approach could build a sound foundation for the process of *information retrieval* which itself is an interesting topic for future research.

The now presented computational model is deferred to the section dealing with *future work*, because the model has not yet been empirically analyzed. Nevertheless, we introduce the basic idea of it and a corresponding algorithm for the computation of F_{MAXGAP}.

6.1.2.1 Algorithm for the computation of F_{MAXGAP}

The basic idea is, to put all the intervals into a $bucket_{intervals}$ and all the gaps into a $bucket_{gaps}$. Then you have to draw by turns out of the two buckets, starting with the

[1] You could also think of a more sophisticated concept, where the different tables, containing grey intervals, are hierarchically linked. In this case only the one with the smallest *MAXGAP* parameter may contain an additional BLOB, whereas all the others only store the hulls of the grey intervals. Here we omit redundancy because we only store the complete information of the grey intervals with their attached interval sequences in one table. On the other hand we still introduce redundancy compared to the approach of this work because we additionally store the hulls of the grey intervals based on different *MAXGAP* parameters.

bucket of intervals. The drawn intervals and gaps are concatenated to a *long grey interval*. If a gap is drawn from *bucket*$_{gaps}$, which is longer than *MAXGAP*, the algorithm stops.

The now introduced model assumes that both the gap and the interval sequence form an *i.i.d. sequence* (*independent*, *i*dentically *d*istributed sequence). This means, that if you repeatedly draw intervals and gaps out of the two buckets, the probability distribution of both does not change. To put it an other way, if you have drawn an interval or gap, the model assumes, that you put it back. Of course, this is not really done, thus the model simplifies the real world (cf. subsection 3.1.5). If you do not assume that you put the intervals back into the buckets, you have to apply a *hyper geometric model* or use *transition probabilities*, which is much more complicated.

Figure 69 depicts the gap and interval histogram, both normalized to 1. The two distribution functions are called *X* and *Y*. Furthermore, let p_M be the probability that a gap is smaller than *MAXGA*P and q_M the probability that it is greater than *MAXGAP* ($p_M+q_M=1$). In a *first step*, we create the function Y^M, by omitting the gaps greater than *MAXGAP* from *Y* and normalize it to 1. In a *second step*, we create the function f^* by convoluting Y^M and *X: .*

$$f^*(x) = Y^M(x)*X(x) = \sum_y Y^M(y)*X(x-y)$$

This function f^* describes the probability distribution of grey intervals, consisting of one black interval and one gap (so they are “no real” grey intervals with respect to definition three, because they do not end with a black interval).

The now introduced algorithm (cf. figure 69) is based on this function f^* and on P_k, the probability of drawing *k-1* gaps with a length smaller than *MAXGAP* and then an interval with a length greater than *MAXGAP*. For P_k the two following statements hold:

$$P_k = p_M^{k-1} \times q_M \quad \text{and} \quad \sum_{=1}^{\infty} P_k = \sum_{k=1}^{\infty} p_M^{k-1} \times q_M = \frac{1}{1-p_M} \times q_M = 1 \cdot$$

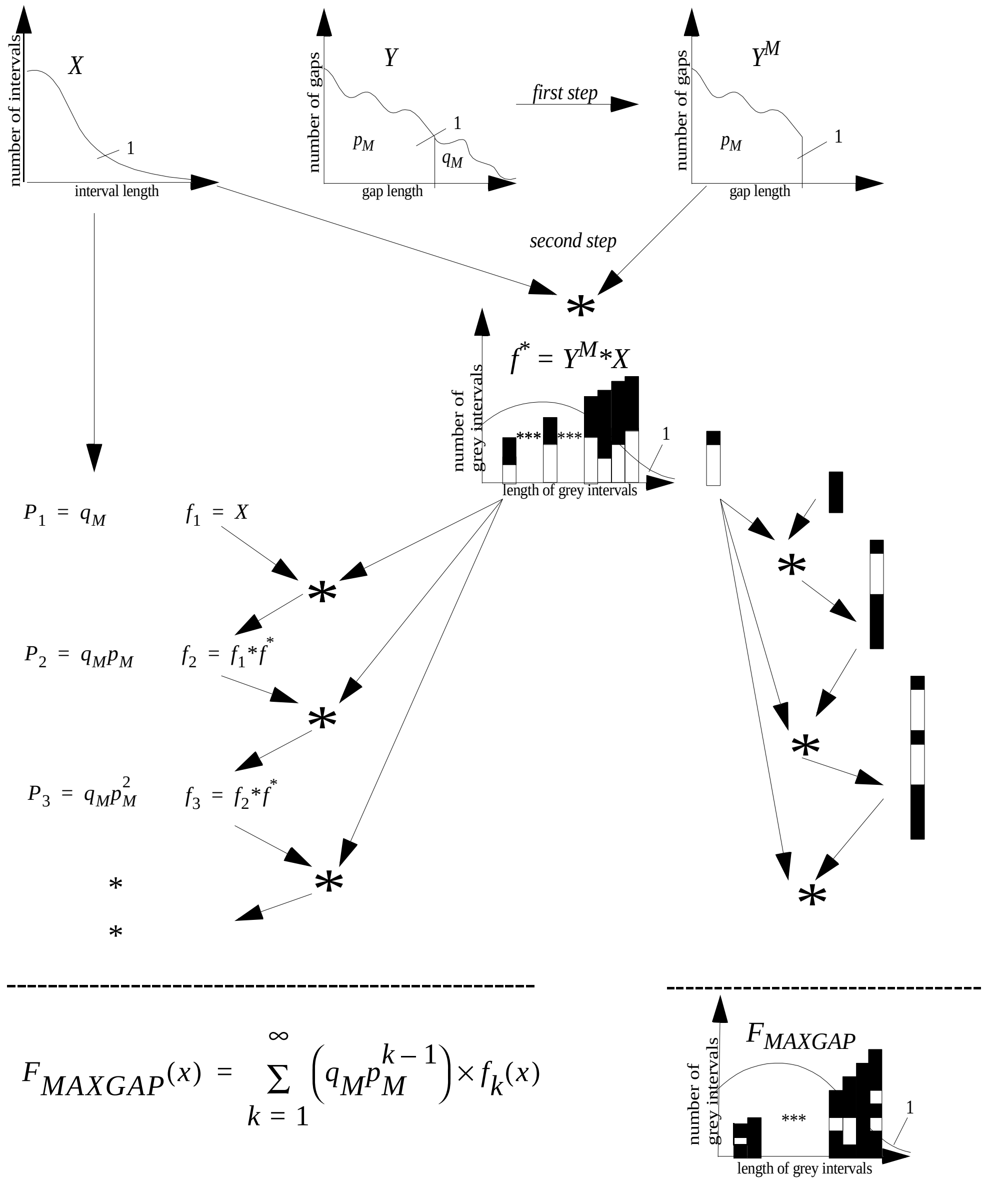


Figure 69: Algorithm for computing F_{MAXGAP}

The algorithm is based on *recursive convolution*. We start with $f_1 = X$ and form each f_k by convoluting f_{k-1} with f^*. Each of the so generated functions is weighted with P_k. Adding all these functions, leads to:

$$F_{MAXGAP}(x) = \sum_{k=1}^{\infty} P_k \times f_k(x)$$

You can stop the algorithm at any point *k* with a controlled error:

$$R^{(k)}(x) = \sum_{j=k+1}^{\infty} P_j \times f_j(x)$$

The error over all *x* can be estimated by:

$$\sum_x R^{(k)}(x) = \sum_{j=k+1}^{\infty} P_j$$

Of course, the more natural way is, to allow a maximum error ε and compute the number of necessary steps k, so that the following holds:

$$\sum_x R^{(k)}(x) < \varepsilon$$

The mathematical approach, indicated above, complements the empirical approach, which has been mainly pursued throughout this work. The future work consists in uniting the both approaches and verifying the mathematical approach by means of empirical results.

6.1.3 An optimum local *MAXGAP* parameter

6.1.3.1 Using different MAXGAP parameters

Up to now we have always used a global *MAXGAP* value for the whole database. In this subsection, we want to talk about an *intermediate* and a *local approach* for an optimum *MAXGAP* parameter (cf. figure 70). In a first step, we only allow different *MAXGAP* parameters for the different objects in the database, but not for their corresponding grey intervals, leading us to an *intermediate approach*. In a second step, we allow different parameters for each grey interval, leading us to a *local approach*.

Finding such optimal local or intermediate *MAXGAP* parameters, would certainly reduce the response time of intersection queries.

One very important task is to define quality criterions for the local, intermediate and global approach which for instance take into account the average density and the

level of different *MAXGAP* parameters	**name of approach**
database	*global approach*
database object	*intermediate approach*
grey interval	*local approach*

Figure 70: Global, intermediate and local *MAXGAP* parameters

overall number of intervals. This criterion should help us to assess whether a fragmentation of an object into grey intervals is good or not.

6.1.3.2 Bounding boxes for grey intervals

Not only could we allow different *MAXGAP* parameters for each grey interval, we could also add bounding boxes to the grey intervals. These bounding boxes could easily be integrated into the *TAIS*-structure which itself is a part of the *upper*- and *lowerIndexes* of the *XRange* table. This approach, depicted in figure 71, might contribute to an enormous reduction of the candidate pairs for the second and third filter step.

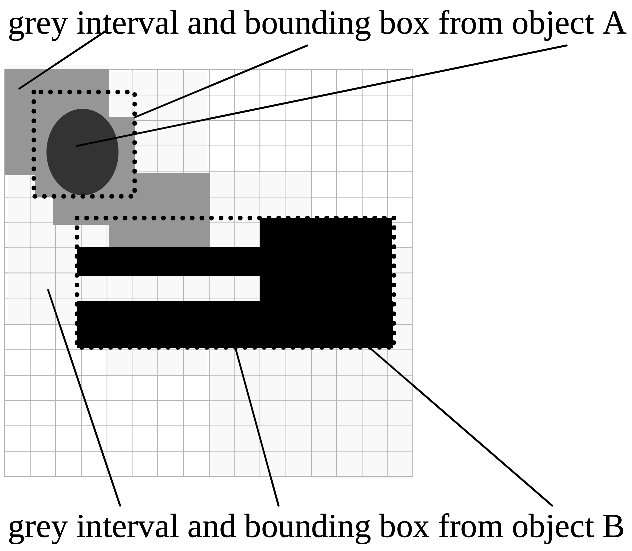

grey interval and bounding box from object A
grey interval and bounding box from object B

Figure 71: Bounding boxes for the grey intervals (2-dimensional)

Figure 71 illustrates that although the grey intervals of the objects A and B interlace, the corresponding bounding boxes do not. In this example, we do not have to test this pair of grey intervals in the expensive third filter step, but can exclude it already in the first or second step, which is still only based on the *upper*- and *lower-Indexes*.

Furthermore we could include progressive approximations into the *TAIS*-structure (cf. subsection 1.4.2) in order to detect more hits in the second filter step.

Thus the detection of an optimum local *MAXGAP* parameter for each grey interval should consider the bounding boxes as well as progressive approximations, as for instance minimum bounding 5-corners and maximum enclosed rectangles (cf. [BKKS 94]), in order to improve the efficiency of the second filter step.

6.1.4 Summary

In this section we presented a list of open problems. We do not claim that this list is exhaustive. It could be arbitrarily extended by topics like *information retrieval* or *general interval relationships*[1] based on the X-RI-tree. Furthermore, you could apply the X-RI-tree to object interval sequences, not stemming from CAD data, but from *temporal applications* or any other area, where objects can be expressed by means of interval sequences. Another interesting topic would be, to use other *transformation functions* than F_{MAXGAP} (cf. figure 25) to group the black intervals together to grey intervals.

A nice side-effect of our reflections about future work is, that we deepened our understanding of the *X-RI-tree*, by comparing it to the *S-RI-tree*. Both trees use the RI-tree index to some extent before switching to a linear scan.

[1] In addition to the intersection query predicate, there are 13 more fine-grained topological and directional relationships between intervals [Allen 83], which are of practical relevance, as a subset of them has been introduced into the new SQL:1999 standard [Sno 00]. We do have to investigate, whether queries based on these predicates are also efficiently supported by the X-RI-tree. The paper from Kriegel, Pötke and Seidl on “object relational indexing for general interval relationships” [KPS 01b] could serve as orientation.

6.2 Summary

In this master thesis, a new indexing method for *object interval sequences* was presented, called *X-RI-tree*. The X-RI-tree is a multi step index, which is based on the RI-tree. To minimize the number of intervals for each object, we *close small gaps*, with a maximum length of *MAXGAP*. As we do not want to loose any information, we connect to each of these newly created *grey intervals* an *attached interval sequence*, stored in a BLOB. This structure of the grey intervals is reflected in the query process, which is based on three major steps:

- In a first filter step, we use the slightly *modified RI-tree* to determine all interlacing pairs of grey database and query object intervals. These interval pairs are ordered by *database ID* and a *value P*, indicating how probable an intersection between the two intervals might be.
- In a second step we perform the so called *fast grey test* to determine intersecting intervals without examining the attached interval sequences. All necessary information for this step is provided by the first filter step, so that no additional I/O accesses are necessary.
- Finally, we carry out the expensive *BLOB test*, scrutinizing the attached interval sequences.

The X-RI-tree has been implemented on top of an Oracle8*i* Server, exploiting its extensible indexing framework. The experimental evaluation of a well parameterized X-RI-tree, compared to the optimized version of the RI-tree, can be summarized as follows:

- Using the X-RI-tree improves the secondary *storage behavior* at least by an order of magnitude.
- Using the X-RI-tree dramatically reduces the *session footprint*.
- Using the X-RI-tree improves the *response time* of collision queries by an order of magnitude.
- Using the X-RI-tree improves the *response time* of box queries by an order of several magnitudes, because the concept of grey intervals is especially beneficial for *top-down dynamically created query objects*.

List of Figures and Tables

List of Definitions

List of Theorems

List of Lessons

References

[Allen 83] Allen J. F.: *Maintaining Knowledge about Temporal Intervals*. Communications ACM 26(11), 832-843, 1983.

[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.

[BKSS 94] Brinkhoff T., Kriegel H.-P., Schneider R., Seeger B.: *Multi-Step Processing of Spatial Joins*. Proc. ACM SIGMOD Int. Conf. on Management of Data, 197-208, 1994.

[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.

[BO 99] Boulos J., Ono K.: *Cost Estimation of User-Defined Methods in Object-Relational Database Systems*. ACM SIGMOD Record, 28(3), 22-28, 1999.

[Bro 01] Brown P.: *Object-Relational Database Development – A Plumber's Guide*. Informix Press, Menlo Park, CA, 2001.

[BSSJ 99] Bliujute R., Saltenis S., Slivinskas G., Jensen C.S.: *Developing a DataBlade for a New Index*. Proc. 15th Int. Conf. on Data Engineering (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 Int. Conf. on Very Large Databases (VLDB), 554-564, 1999.

[CCN+ 99] Carey M. J., Chamberlin D. D., Narayanan S., Vance B., Doole D., Rielau S., Swagerman R., Mattos N.: O-O, What Have They Done to DB2? Proc. 25th Int. Conf. on Very Large Databases (VLDB), 542-553, 1999.

[CZ 01] Chaudhri A. B., Zicari R.: *Succeeding with Object Databases*. Wiley, New York, NY, 2001.

[Dat 99] Date C. J.: *An Introduction to Database Systems*. Addison Wesley Longman, Boston, MA, 1999.

[Doh 98] Doherty C. G.: *Database Systems Management and Oracle8*. Proc. ACM SIGMOD Int. Conf. on Management of Data, 510-511, 1998.

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

[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 Symposium on Principles of Database Systems (PODS), 247-252, 1989.

[Gae 95] Gaede V.: *Optimal Redundancy in Spatial Database Systems*. Proc. 4th Int. Symp. on Large Spatial Databases (SSD), LNCS 951, 96-116, 1995.

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

[GR 94] Gaede V., Riekert W.-F.: *Spatial Access Methods and Query Processing in the Object-Oriented GIS GODOT*. Proc. AGDM Workshop, Geodetic Commission, 1994.

[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.

[Güt 94] Güting R. H.: *An Introduction to Spatial Database Systems*. VLDB Journal , 3(4), 357-399, 1994.

[HJR 97] Huang Y.-W., Jones M.C., Rundensteiner E. A.: Improving Spatial Intersect Joins Using Symbolic Intersect Detection. Proc. 5th Int. Symp. on Large Spatial Databases (SSD), LNCS 1262, 165-177, 1997.

[HS 93] Hellerstein J., Stonebraker M.: *Predicate Migration: Optimizing Queries with Expensive Predicates*. Proc. ACM SIGMOD Int. Conf. on Management of Data, 267-276, 1993.

[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, Version 3.4*. Menlo Park, CA, 1998.

[IWB 01] Digital Mock-up Process Simulation For Product Conception and Downstream Processes (Brite-Euram Project BRPR-CT95-0066) http://ww.iwb.tum.de/projekte/dmu-ps

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

[Jen 99] Jensen C. S.: *Review - Multi-Step Processing of Spatial Joins*. ACM SIGMOD Digital Review 1, 1999.

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

[KMPS 01a] Kriegel H.-P., Müller A., Pötke M., Seidl T.: *DIVE: Database Integration for Virtual Engineering* (Demo). Demo Proc. 17th Int. Conf. on Data Engineering (ICDE), 15-16, 2001.

[KMPS 01b] Kriegel H.-P., Müller A., Pötke M., Seidl T.: *Spatial Data Management for Computer Aided Design* (Demo). Proc. ACM SIGMOD Int. Conf. on Management of Data, 2001.

[Kor 99] Kornacker M.: *High-Performance Extensible Indexing*. Proc. 25th Int. Conf. on Very Large Databases (VLDB), 699-708, 1999.

[KPS 00a] Kriegel H.-P., Pötke M., Seidl T.: *Managing Intervals Efficiently in Object-Relational Databases*. Proc. 26th Int. Conf. on Very Large Databases (VLDB), 407-418, 2000.

[KPS 00b] Kriegel H.-P., Pötke M., Seidl T.: *Relational Interval Tree*. European Patent Office, Patent Application No. 00112031.0, 2000.

[KPS 01a] Kriegel H.-P., Pötke M., Seidl T.: *Interval Sequences: An Object-Relational Approach to Manage Spatial Data*. Proc. 7th Int. Symposium on Spatial and Temporal Databases (SSTD), LNCS, 2001.

[KPS 01b] Kriegel H.-P., Pötke M., Seidl T.: *Object-Relational Indexing for General Interval Relationships*. Proc. 7th Int. Symposium on Spatial and Temporal Databases (SSTD), LNCS, 2001.

[MH 99] Möller T., Haines E.: *Real-Time Rendering*. A K Peters, Natick, MA, 1999.

[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.

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

[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, Release 2 (8.1.6)*. Redwood Shores, CA, 1999.

[Ora 99b] Oracle Corp.: *Oracle8i Object-Relational Features, Release 2 (8.1.6)*. Redwood Shores, CA, 1999.

[Ora 99c] 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 Int. Conf. on Management of Data, 294-305, 1989.

[Pöt 01] Pötke M.: *Spatial Indexing for Object-Relational Databases*, Ph.D. Thesis, Faculty for Mathematics and Computer Science, University of Munich, 2001.

[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 SSD, LNCS 1651, 355-359, 1999.

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

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

[SB 98] Stonebraker M., Brown P.: *Object-relational DBMSs – Tracking the Next Great Wave*. Morgan Kaufmann, San Francisco, CA, 1998.

[SK 93] Schiwietz M., Kriegel H.-P.: *Query Processing of Spatial Objects: Complexity versus Redundancy.* Proc. 3rd Int. Symp. on Large Spatial Databases (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. 16th Int. Conf. on Data Engineering (ICDE), 91-100, 2000.

[Sno 00] Snodgrass R. T.: *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, San Francisco, CA, 2000.

[SOL 94] Shen H., Ooi B. C., Lu H.: *The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases*. Proc. 10th Int. Conf. on Data Engineering (ICDE), 274-281, 1994.

[SQL 99] American National Standards Institute: *ANSI/ISO/IEC 9075-1999 (SQL:1999, Parts 1-5)*. New York, NY, 1999.

[Sto 86] Stonebraker M.: *Inclusion of New Types in Relational Data Base Systems*. Proc. 2nd Int. Conf. on Data Engineering (ICDE), 262-269, 1986.

[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.

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

[ZS 98] Zimbrao G., Moreira de Souza J.: *A Raster Approximation for the Processing of Spatial Joins*. Proc. 24th Int. Conf. on Very Large Databases (VLDB), 558-569, 1998.