Efficient Query Processing on the Relational Quadtree

HANS-PETER KRIEGEL, PETER KUNATH, MARTIN PFEIFLE, MATTHIAS RENZ
University of Munich, Germany, {kriegel, kunath, pfeifle, renz}@dbs.informatik.uni-muenchen.de

Abstract. Relational index structures, as for instance the Relational Interval Tree, the Relational R-Tree, or the Linear Quadtree, support efficient processing of queries on top of existing object-relational database systems. Furthermore, there exist effective and efficient models to estimate the selectivity and the I/O cost in order to guide the cost-based optimizer whether and how to include these index structures into the execution plan. By design, the models immediately fit to common extensible indexing/optimization frameworks, and their implementations exploit the built-in statistics facilities of the database server. In this paper, we show how these statistics can also be used for accelerating geo-spatial queries using the relational quadtree by reducing the number of generated join partners which results in less logical reads and consequently improves the overall runtime. We cut down on the number of join partners by grouping different join partners together according to a statistic driven grouping algorithm. Our experiments on an Oracle9i database yield an average speed-up between 30% and 300% for spatial selection queries on the Relational Quadtree.

1 Introduction

The efficient management of complex objects has become an enabling technology for geographical information systems (GIS) as well as for many novel database applications, including computer aided design (CAD), medical imaging, molecular biology. For commercial use, a seamless and capable integration of spatial indexing into industrial-strength databases is essential. In order to integrate these index structures into modern ORDBMSs, we need suitable cost models [7], which exploit the built-in statistics facilities of the database server. Based on these statistics it is possible to estimate the selectivity of a given query and to predict the cost of processing that query.

In an ORDBMS the user has no access to the exact information where the blocks are located on the disk. Former approaches which try to generate efficient read schedules for a given set of disk pages [14] must know the actual position of the pages on the storage media. As this information is not available in an ORDBMS, we pursue another idea which exploits already existing statistics in order to accelerate spatial query processing. We introduce our approach in general as well as exemplarily for spatial selection queries performed on the Relational Quadtree (RQ-tree).

The remainder of this paper is organized as follows. In Section 2, we discuss the paradigm of relational indexing. Exemplarily, we present the RQ-tree, for which we will demonstrate the advantages of our new approach throughout this paper. In Section 3, we show how we can use the already existing statistics to accelerate the query process. In Section 4, we present convincing experimental results based on both, a geographical 2D data set which corresponds to the SEQUOIA 2000 benchmark [13] and a 3D CAD data set from a German car manufacturer. Finally, we conclude the paper with a few remarks on future work.

2 Relational Access Methods

In this section, we will discuss the basic properties of relational access methods with respect to the storage of index data and query processing, by exemplarily introducing the RQ-tree. Furthermore, we discuss in Section 2.2 how we can integrate these index structures into modern ORDBMSs. We start with a definition common to all relational access methods:

Definition 1 (*Relational Access Method*)
An access method is called a *relational access* method, iff any index-related data are exclusively stored in and retrieved from relational tables. An instance of a relational access method is called a *relational index*. The following tables comprise the persistent data of a relational index:

(*i*) *User table*: a single table, storing the original user data being indexed.

(*ii*) *Index tables*: n tables, $n \geq 0$, storing index data derived from the user table.

(*iii*) *Meta table*: a single table for each database and each relational access method, storing $O(1)$ rows for each instance of an index.

The stored data are called *user data*, *index data*, and *meta data*.

To illustrate the concept of relational access methods, Figure 1 presents the minimum bounding rectangle list (MBR-List), a very simple example for indexing two-dimensional polygons. The user table is given by the object-relational table polygons (cf. Figure 1a), comprising attributes for the polygon data type (geom) and the object identifier (id). Any spatial query can already be evaluated by sequentially scanning this user table. In order to speed up spatial selections, we decide to define an MBR-List polygons_idx on the user table. Thereby, an index table is

polygons

id	geom
A	POLYGON((10,10), (25,15), ..., (10,10))
B	POLYGON((30,15), (30,45), ..., (30,15))
...	...

a) User table

polygons_mbr

id	mbr
A	BOX((5,10), (30,15))
B	BOX((30,5), (40,50))
...	...

b) Index table

mbr_index_metadata

index_name	user_table	index_table
'polygons_idx'	'polygons'	'polygons_mbr'
...	...	...

c) Meta table

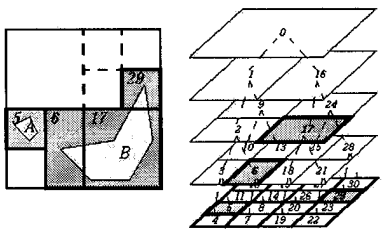
Figure 1: The MBR-List, a simple example for a relational access method

created and populated (cf. Figure 1b), assigning the minimum bounding rectangles (mbr) of each polygon to the foreign key id. Thus, the index table stores information purely derived from the user table. All schema objects belonging to the relational index, in particular the name of the index table, and other index parameters are stored in a global meta table (cf. Figure 1c).

In order to support queries on the index tables, a relational access method can employ any built-in secondary indexes, including hash indexes, B^+-trees, and bitmap indexes. Alternatively, index tables may be clustered by appropriate primary indexes. Consequently, the relational access method and the database system cooperate to maintain and retrieve the index data [2]. This basic approach of relational indexing has already been applied in many existing solutions, including Linear Quadtrees [16] [10] [3] and Relational R-trees [11] for spatial databases, Relational X-trees [1] for high-dimensional nearest-neighbor search, or inverted indexes for information retrieval on text documents [2].

2.1 The Relational Quadtree

A paradigmatic example for a spatial access method implementing the direct scheme is the Relational Quadtree [12]. This access method 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. In this subsection, we present the basic idea of the Relational Quadtree according to the in-depth discussion of Freytag, Flasza and Stillger [3].



a) Regular grid **b)** Recursive partitioning

polygons_quadtree

zval	id
5	A
6	B
17	B
29	B
...	...

c) Index table

Figure 2: Relational mapping of a Linear Quadtree

The Relational Quadtree organizes the multidimensional data space by a regular grid. Any spatial object is approximated by a set of tiles. Among the many possible one-dimensional embeddings of a grid approximation, the Z-order is one of the most popular [6]. The corresponding index representation of a spatial object comprises a set of Z-tiles which is computed by recursively bipartitioning the multidimensional grid. By numbering the Z-tiles of the data space according to a depth-first recursion into this partitioning, any set of Z-tiles can be represented by a set of linear values. Thereby an object is decomposed into several tiles which are stored independently within an index, i.e. redundancy is introduced because spatially extended data is referenced in an index more than once [4]. Figure 2 depicts some Z-tiles on a two-dimensional grid along with their linear values. The linear values of the Z-tiles of each spatial object can be stored in an index table obeying the schema (zval, id), where both columns comprise the primary key. This relational mapping implements the direct scheme, as each row in the index table exclusively belongs to a single data object. The linear ordering positions each Z-tile of an object on its own row in the index table. Thus, if a specific row in the user table polygons is updated, e.g. (B, ...) in Figure 1, only the rows (6, B), (17, B), and (29, B) in the index table are affected (cf. Figure 2), causing no problems with respect to the native two-phase locking.

In order to process spatial selection on the Relational Quadtree, the query region is also required to be decomposed to a set of Z-tiles. We call the corresponding function ZDecompose. For each resulting linear value zval, the intersecting tiles have to be extracted from the index table. Due to the Z-order, all intersecting tiles having the same or a smaller size than the tile represented by zval occupy the range ZLowerHull(zval) = [zval, ZHi(zval)] which can be easily computed [3]. In the example of Figure 2, we obtain ZLowerHull(17) = [17, 23]. In a similar way, we also compute ZUpperHull(zval), the set of all larger intersecting tiles. As in the case of ZUpperHull(17) = {0, 16} the corresponding linear values usually form no

```
SELECT DISTINCT idx.id        // select data object
FROM polygons_quadtree idx,
   TABLE(ZDecompose(BOX((0,0),(100,100)))) tiles,
   TABLE(ZUpperHull(tiles.zval)) uh
WHERE (idx.zval BETWEEN tiles.zval AND ZHi(tiles.zval))
   OR (idx.zval = uh.zval);
```

Figure 3: Window query on a Relational Quadtree

consecutive range. To find all intersecting tiles for a given zval, a range scan on the index table is performed with ZLowerHull(zval) and multiple exact match queries are executed for ZUpperHull(zval). These queries are optimally supported by a built-in B^+-tree on the zval column. Figure 3 depicts the complete window query on an instance of the Relational Quadtree using SQL:1999. Alternatively, the transient rowsets generated by the functions ZDecompose and ZUpperHull can be precomputed in the procedural phase for all Z-tiles of the query box and passed to the SQL layer in one step by using bind variables. This approach reduces the overhead of barrier crossings between the declarative and procedural environments to a minimum.

2.2 Extensible Indexing

The design of extensible architectures represents an important area in database research. The object-relational data model marked an evolutionary milestone by introducing abstract data types into relational database servers. Thereby, object-relational database systems may be used as a natural basis to design an integrated user-defined database solution. The ORDBMSs already support major aspects of the declarative embedding of user-defined data types and predicates. In order to achieve a seamless integration of custom object types and predicates within the declarative DDL and DML, ORDBMSs provide the database developer with extensibility interfaces. They enable the declarative embedding of abstract data types within the built-in optimizer and query processor. Corresponding frameworks are available for most object-relational database systems, including Oracle and DB2.

An important requirement for applications is the availability of user-defined access methods. Extensible indexing frameworks proposed by Stonebraker [15] enable 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. The row-based processing of selections and update operations follows the iterator pattern [5]. Thereby, the indextype complements the functional implementation of user-defined predicates. If the optimizer decides to include this custom

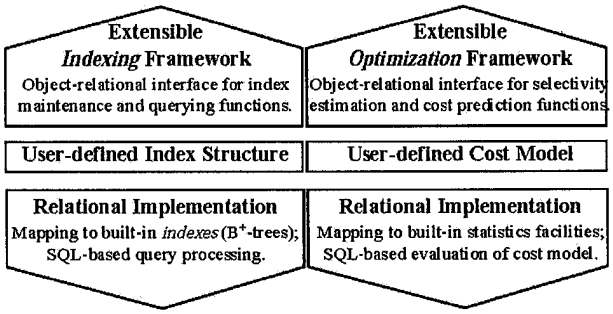


Figure 4: Analogous architectures for the object-relational embedding of user-defined index structures and cost models into extensible indexing and optimization frameworks, respectively

index into the execution plan for a declarative DML statement, the appropriate indextype functions are called by the built-in query processor of the database server. Thus, the maintenance and access of a custom index structure is completely hidden from the user, and the desired data independence is achieved. Furthermore, the framework guarantees any redundant index data to remain consistent with the user data.

The architecture of extensible optimization is analogous to extensible indexing as illustrated in Figure 4. Whereas the new methods are built on top of the relational SQL layer, they are object-relationally embedded by implementing the respective interfaces of the frameworks. Object-relational database systems typically support rule-based and cost-based query optimization. The extensible indexing framework comprises interfaces to tell the built-in optimizer about the characteristics of a custom indextype. Cost models particularly support methods to estimate the selectivity of a given range query on a database (function getSelectivity) and methods to predict the cost of processing that query (function getIndexCost).With a cost model registered at the built-in optimizer framework, the cost-based optimizer is able to rank the potential usage of a custom access method among alternative access paths. Thus, the system supports the generation of efficient execution plans for queries comprising user-defined predicates. This approach preserves the declarative paradigm of SQL, as it requires no manual query rewriting.

3 Statistic based Acceleration of Spatial Queries

In addition to the query optimizer of an ORDBMS, which uses statistics for rule-based optimizations such as push-selections, we use the statistics to minimize the overall navigational cost of a relational index structure. Our approach accelerates relational access methods by trying to reduce the total number of logical reads for a given query.

The relational access method can be any custom index structure mapped to a fine granular relational schema which is organized by built-in access methods, as for instance the B^+-tree. All statistic-based optimizations presented in this section can also be applied to variants of the basic relational index structures. For instance, there exist index structures which were especially tuned for coping efficiently with sequences. One example is the RI-tree as introduced in [9]. It supports the efficient detection of intersecting spatial objects, which are represented by interval sequences. The main idea of this index structure is to neglect such nodes as join partners which are already handled by the previous query interval or which will be handled by the following one. The main disadvantage of this approach is that only specific predicates are supported by this kind of index structures. For instance the RI-tree according to [9] only supports boolean intersection queries, but already fails to compute the intersection volume. Similar optimizations are possible for the RQ-tree by eliminating duplicates from the upper hulls resulting from different query tiles of a given query sequence.

In this section, we first look at very comprised statistic values, which can already be very useful for accelerating the relational Quadtree. Then we show how we can benefit from the statistics, used by the cost-models belonging to the relational access method.

3.1 Statistics Related to the Relational Quadtree

As already indicated in Definition 1 and Figure 1, the metadata table is a single table for each database and each relational access method, storing O(1) rows for each instance of an index. All schema objects belonging to the relational index, in particular the name of the index table, and other index parameters are stored in this global meta table.

Especially in the case of space partitioning index structures, often a few values, describing the actual data distribution, help to reduce the I/O cost dramatically. If we use the RQ-tree for indexing extended objects, very often only the lower levels of the virtual primary structure are engaged, as spatial objects tend to decompose into numerous small tiles (cf. Section 4). Consequently, we store two additional parameters *MaxTileLevel* and *MinTileLevel* within the metadata table of the Relational Quadtree. These two parameters reflect the highest and lowest level of stored tiles within the database. If we compute the upper hull of a given query tile q, we only have to consider those tiles t as join partners, for which $MinTileLevel \leq Level(t) \leq MaxTileLevel$ holds. In general such kind of simple statistics are especially useful for indexing extended spatial objects.

3.2 Statistics Related to the Build-In Index Structure

In [7] it was shown that using quantiles ('equi-count histograms') is more suitable for estimating the selectivity and the corresponding I/O cost than using common histograms ('equi-width histograms'). In addition, the runtime required for the histogram computation is increased by the cost of barrier-crossings between the declarative environment of the SQL layer and our stored procedure. Fortunately, most ORDBMS comprise efficient built-in functions to compute single-column statistics, particularly for cost-based query optimization. Available optimizer statistics are accessible to the user by the relational data dictionary. The basic idea of our quantile-based selectivity estimation is to exploit these built-in index statistics rather than to add and maintain user-defined histograms. We start with the definition of a quantile vector, the typical statistics type supported by relational database kernels.

Definition 2 (*Quantile Vector*).

Let $(M, \leq)$ be a totally ordered multi-set. Without loss of generality, let $M = \{m_1, m_2, \ldots, m_N\}$ with $m_j \leq m_{j+1}$, $1 \leq j < N$. Then $Q(M, \nu) = (q_0, \ldots, q_\nu) \in M^\nu$ is called a *quantile vector* for M and a *resolution* $\nu \in IN$, iff the following conditions hold:

(*i*) $q_0 = m_1$

(*ii*) $\forall i \in 1, \ldots, \nu: \exists j \in 1, \ldots, N: \; q_i = m_j \wedge \frac{j-1}{N} < \frac{i}{\nu} \leq \frac{j}{N}$

We will now discuss how we can use this information to accelerate the query process itself. Any query for the RQ-tree leads to several index range scans on the built-in index structures, e.g. B^+-tree. The general idea of our approach is to minimize the overall navigational cost of the built-in index by applying extended index range scans. Thereby, we read false hits from the index, which are filtered out by a subsequent refinement step. Our approach closes the gaps between the index scan ranges if and only if the number of additional read data is comparably small, more precisely the cost related to these false hits is smaller than the navigational cost related to an additional range scan. This decision whether to close a gap is based on the built-in statistics. We will now formally introduce this idea.

3.2.1 Index Range Scan Sequences

For spatial intersection queries, the query object Q leads to many disjoint range queries $s_i = (l_i, u_i)$ on the built-in index I, e.g. the B^+-tree. We consider them as a sequence $Seq_{Q,I} = (\langle s_1, \ldots, s_n \rangle)$ of index range scans (cf. Figure 5a) for which the following assumptions hold:

- The elements r_i stored in the index are of the same type as l_i, u_i. Furthermore, we assume that the

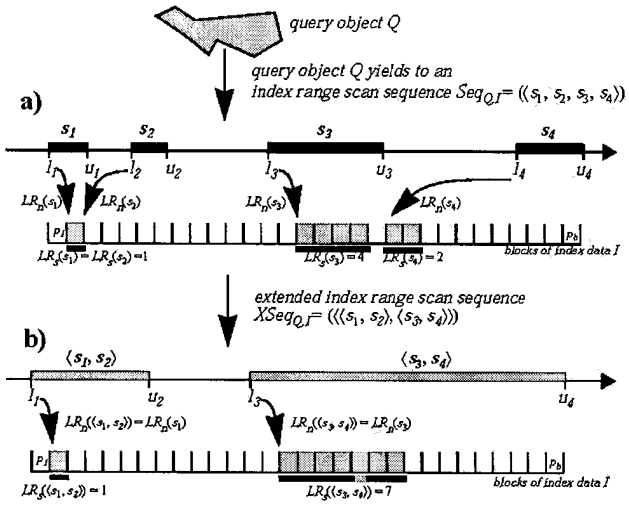


Figure 5: Accelerated query processing
a) Index range scan sequence **b)** Extended index range scan sequence

elements r_i can be regarded as a linear ordered list $L(I) = \langle r_1,...,r_N \rangle$ for which $r_1 \leq ... \leq r_N$ holds.

- We assume that the data pages p_i of the index obey a linear ordering $\leq$ and fulfill the following property: $r' \leq r'' \Leftrightarrow p(r') \leq p(r'')$, where $p(r)$ denotes the disk page of the index I, which contains the entry r.

I/0 cost. The I/O cost $C^{I/O}(s)$ associated with one index range scan $s = (l, u)$ of $Seq_{Q,I} = (\langle s_1,..., s_n \rangle)$ are composed from two parts: $C_n^{I/O}(s)$ the navigational I/O cost for finding the first page of the result set, and $C_s^{I/O}(s)$ the cost for scanning the remaining pages containing the complete result set. Formally, $C^{I/O}(s) = C_n^{I/O}(s) + C_s^{I/O}(s)$, with the following two properties:

$$(i) \quad C_n^{I/O}(s) = C_n^{I/O}(p(r')) \qquad (navigational\ cost)$$

$$(ii) \quad C_s^{I/O}(s) = C_s^{I/O}(\langle p(r'),...,p(r'') \rangle) \qquad (scan\ cost)$$

where $r', r'' \in L(I)$ and $\forall r \in L(I) : (r' \leq r \leq r'') \Leftrightarrow (l \leq r \leq u)$ holds. The I/O cost $C^{I/O}(Seq_{Q,I})$ associated with $Seq_{Q,I} = (\langle s_1,..., s_n \rangle)$ are determined by the sum of all individual I/O cost: $C^{I/O}(Seq_{Q,I}) = \sum_{i=1}^{n} C^{I/O}(s_i)$.

3.2.2 Extended Index Range Scan Sequences

The main purpose of our approach is to minimize the overall cost for the navigational part of the built-in index. Therefore, we try to reduce the number of generated range queries on the index I, while only allowing a small increase in the output cost. This can be achieved by merging two suitable adjacent range scans $s' = (l', u')$ and $s'' = (l'', u'')$ together to one *extended range scan* $xs = (l', u'')$.

Intuitively, an *extended range scan* $xs = \langle s_r,..,s_s \rangle$ is an ordered list of index range scans. When carrying it out, we traverse the index directory only once and perform a range scan (l_r, u_s), as for example (l_3, u_4) in Figure 5b. Performing the *extended range scan* we read false hits from the index I, which have to be filtered out in a subsequent refinement step. The overall cost $C(xs)$ of an *extended range scan* xs are composed from the sum of the I/O cost of the *extended range scan* and the CPU cost related to the refinement step: $C(xs) = C^{I/O}(xs) + C^{CPU}(xs)$.

I/O cost. The I/O cost $C^{I/O}(xs)$ associated with one extended range scan $xs = \langle s_r,..,s_s \rangle$ are composed from two parts $C^{I/O}(xs) = C_n^{I/O}(xs) + C_s^{I/O}(xs)$, with the following properties:

$$(i) \quad C_n^{I/O}(xs) = C_n^{I/O}(s_r) \qquad (navigational\ cost)$$

$$(ii) \quad C_s^{I/O}(xs) = C_s^{I/O}(l_r, u_s) \qquad (scan\ cost)$$

CPU cost. The CPU cost $C^{CPU}(xs)$ associated with one extended range scan $xs = \langle s_r,..,s_s \rangle$ denote the cost which are required to perform the filter operation for all tuples resulting from the extended range scan:

$$C^{CPU}(xs) = C^{CPU}(\langle r',...,r'' \rangle),$$
$$\text{where } \forall r \in L(I) : (r' \leq r \leq r'') \Leftrightarrow (l_r \leq r \leq u_s).$$

The total cost $C(XSeq_{Q,I})$ associated with an *extended index range scan sequence* $XSeq_{Q,I} = (\langle xs_1,..., xs_m \rangle)$ can be computed as follows:

$$C(XSeq_{Q,I}) = \sum_{j=1}^{m} C(xs_j).$$

Obviously, there might exist extended index range scan sequences $XSeq_{Q,I}$ for which the following relation $C(XSeq_{Q,I}) << C(Seq_{Q,I})$ holds. For each gap g between two adjacent range queries s' and s'' we decide, whether the cost of scanning over the gap g are lower than the navigational I/O cost related to s''. The decision whether to merge range scan s' and s'' to one extended range scan and apply an additional refinement step afterwards in order to filter out false hits is based on statistics, which are necessary for the cost models anyway.

The multi-set M of our quantile vector $(q_0, ..., q_v)$ (cf. Definition 2) is formed by the values of the first attribute A_1 of the domain values of our index I. By means of these statistics we can estimate the I/O cost $C_s^{I/O}(s)$ associated with one range scan $s = (l, u)$. In the following formula, b denotes the number of disk blocks at the leaf level of I, v denotes the resolution of the quantile vector, N denotes the overall number of entries stored in the index I and *overlap*

returns the intersection length of two intersecting intervals.

$$C_s^{I/O}((l, u)) \approx C_s^{est}((l, u)), \text{ and}$$

$$C_s^{est}((l, u)) = \frac{\sum_{i=1}^{v} \left(\frac{overlap((l, u), (q_{i-1}, q_i))}{q_i - q_{i-1}} \cdot \frac{N}{v} \right)}{(N/b)}$$

We can also apply the above formula to estimate the total cost $C_s(g) = C_s^{I/O}(g + C^{CPU}(g)$ related to scanning over a gap $g =]u', l'[$ between two adjacent range queries s' and s''. The CPU cost can be estimated by $C^{CPU}(g) = k \cdot C_s^{I/O}(g)$, with a parameter $k > 0$, since both the I/O cost and the CPU cost are directly proportional to the size of the result set of the range scan. If $C_s(g)$ are lower than $C_n(s'')$, we close the gap g.

We can find the extended range scan sequence $XSeq_{Q,I}$, trying to minimize $C(XSeq_{Q,I})$, by deciding for each of the n-1 gaps between the index range scans $s_1, \ldots, s_n$ of the index range scan sequence $Seq_{Q,I} = (\langle s_1, \ldots, s_n \rangle)$, whether we close this gap or skip it. Thus we obtain an *extended index range scan* sequence $XSeq_{Q,I} = (\langle \langle s_{i_0+1}, .., s_{i_1} \rangle, \ldots, \langle s_{i_{m-1}+1}, .., s_{i_m} \rangle \rangle)$, which satisfies the following property:

$$\forall i \in 1 \ldots n-1 : i \in i_1 \ldots i_{m-1} \Leftrightarrow C_n^{est}(s_{i+1}) < C_s^{est}(u_i, l_{i+1})$$

Usually, the actual navigational cost $C_n^{I/O}$ are independent of the actual range scan and can easily be estimated by C_n^{est}, e.g. by the height of the B$^+$-directory.

In the next sections, we will show how our approach can be applied to the intersect predicate for a specific index structures, namely the Relational Quadtree.

3.2.3 Adoption to the RQ-tree

In this section, we shortly introduce our approach based on the basic idea of the Relational Quadtree according to the in-depth discussion of Freytag, Flasza and Stillger [3].

Assume object Q in Figure 6 is used as query object. Then there are multiple exact match and range scan queries which have to be performed in order to detect all intersecting database objects. We can reduce the cost by closing small gaps on the leaf-level of the underlying B$^+$-tree. By using the information stored in the statistics, i.e. using the tile quantiles, the number of join partners, which correspond directly to the navigational cost $C_n^{I/O}$, can be reduced drastically. The quantile vector is built over the values stored in the leaf-level of the B$^+$-tree.

We investigate all gaps included in the sequence of our generated join partners and decide whether it is beneficial to close this gap. Assume the height of our B$^+$-directory is n. If we close the gap, we reduce the navigational cost as follows: $C_n^{I/O} = C_n^{I/O} - n$. On the other hand, we

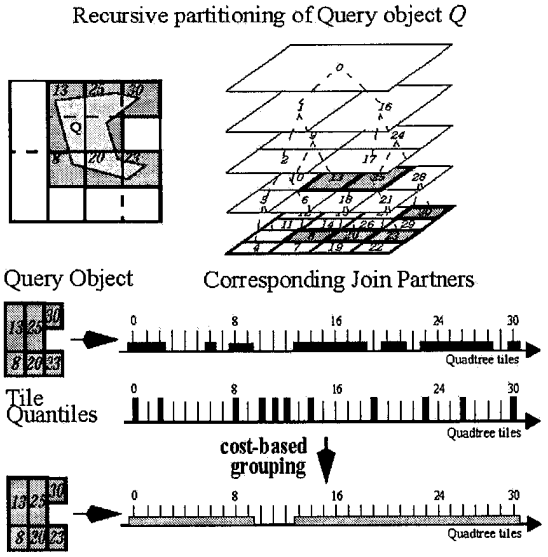


Figure 6: Cost-Based Tile Grouping

estimate the cost $C_s(g)$ required to read the leaf blocks on our index (*zval*), which are covered by the database tiles of the actual investigated gap g. If these estimated cost are lower than n, we close this gap. Thus we reduce the join cost $C_n^{I/O}$ by n, while not increasing the output cost C_s by more than n. This procedure is depicted in Figure 6.

The above mentioned cost-based grouping step can be carried out in a procedural preparation step JoinPartGen by using bind variables, leading to one single SQL-statement (cf. Figure 7). This approach reduces the overhead of barrier crossings between the declarative and procedural environments to a minimum. The resulting table tiles contains entries of a type which consists of three attributes *ZvalLow*, *ZvalHigh* and *ExactZvalList*. The attribute *ExactZvalList* is a collection of tile ranges, representing the accurate query information. It is needed for an additional refinement step to filter out false index hits, by calling *TestZval*().

4 Experimental Evaluation

The tests are based on two test data sets *CAR* (3D CAD data) and *GEO* (2D geographical data, derived from the SEQUOIA benchmark).

CAD data set: The first test data set is provided by our industrial partner, a German car manufacturer, in form of

```
SELECT DISTINCT idx.id
FROM   DBTiles idx,
       TABLE(JoinPartGen(BOX((0,0),(10,10)))) tiles,
WHERE  (idx.zval BETWEEN tiles.ZvalLow AND tiles.ZvalHigh)
       AND TestZval(idx.zval, tiles.ExactZvalList);
```

Figure 7: Accelarated window query

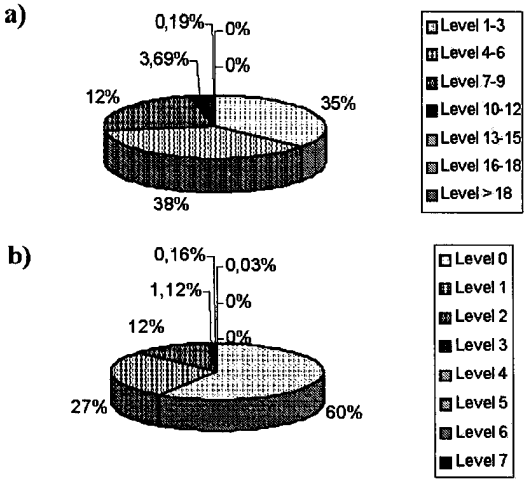


Figure 8: Used index levels: **a)** GEO **b)** CAD data set

high-resolution voxelized three-dimensional CAD parts. It consists of approximately 14 million voxels representing 200 parts. The *CAD* data space is of size 2^{33}.

GEO data set: The GEO data set consists of voxelized two-dimensional polygons derived from the *SEQUOIA 2000* Polygon Data Set. It contains approximate $1.1 \cdot 10^9$ voxels representing about 57,500 polygons. The *GEO* data space is of size 2^{30}.

In both cases, the Z-curve was used as a space filling curve to enumerate the voxels.

We have implemented our approach for the RQ-tree on top of the Oracle9i Server using PL/SQL for the computational main memory based programming. All experiments were 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.

4.1 Histograms of the Test Data Sets

Mainly the lowest levels of the RQ-tree contain index entries. Figure 8a shows that in the case of the GEO data set only the lower levels are occupied and in the case of the CAD data set (cf. Figure 8b) even only the seven lowest of 33 levels. The observation that spatial objects are decomposed into many small tiles are not confined to our two test data sets but hold for spatial objects in general [4] [8]. Therefore, the statistics presented in Section 3.1 are very beneficial for efficient query processing on spatially extended objects in general.

4.2 Query Processing

In this section, we examine the benefits of using extended index range scans. The experiments on the GEO data set are based on window queries, which comprise about 9500

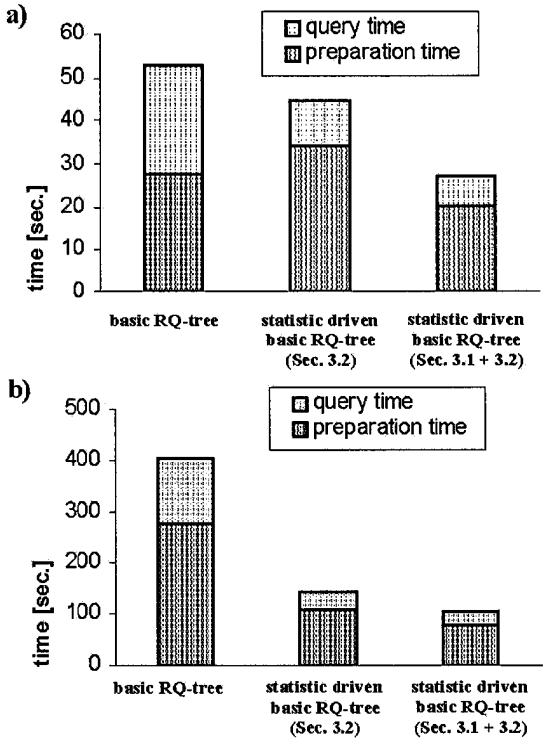


Figure 9: Statistic based accelerated RQ-tree on the GEO and CAR dataset: runtime for the basic RQ-tree and the optimized versions according to Section 3.1 and 3.2 using **a)** the GEO data set and **b)** the CAD data set.

tiles and have an average selectivity of about 2% of all stored objects. For the CAD data set we used 10% of the database objects as query objects. For all experiments we report the average results from these queries.

In the following experiments, we applied the statistic based approach to the RQ-tree for the GEO data set as well as for the CAD data set. Figure 9 shows that the use of our quantile statistics (cf. Section 3.2) accelerates the RQ-tree dramatically. A further improvement can be achieved by using the information of the highest and lowest level of stored tiles within the database (cf. Section 3.1), leading to a speed-up of almost 300% in the case of the CAD data set. Figure 10 depicts the acceleration of the sequence optimized RQ-tree, where we compare the variant without incorporating the CPU-cost of the refinement step with the variant including the CPU-cost (cf. Section 3). The first variant considers only the I/O-cost and neglects the CPU-cost for forming the extended range scan sequences: Figure 10b shows that this approach leads only to an acceleration in the preparation step, but the overall query time increases due to the expensive refinement process. On the other hand, if we incorporate the CPU-cost for the cost estimation, we can achieve an overall speed-

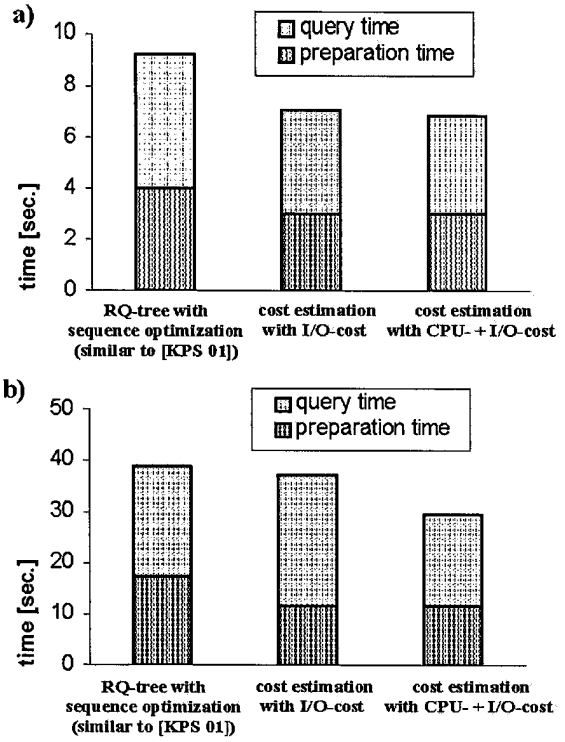


Figure 10: Statistic based accelerated RQ-tree: Runtime for the basic RQ-tree optimized for sequences (similar to [9]) using **a)** the GEO data set and **b)** the CAD data set.

up of approximately 30%, even for this highly specialized index structure.

To sum up, we achieve an acceleration of the query process by 30% to 300%, if we form the extended range scans according to the available statistics considering both expected I/O-cost and expected CPU-cost.

5 Conclusion

In this paper, we have shown how we can accelerate spatial query processing by means of statistics which are available for free, as they are maintained by the cost models belonging to the corresponding spatial index structures. We have implemented our approach for the Relational Quadtree on top of the Oracle9i database system. According to our experiments, we achieved speed-up factors of up to 300%. Our new statistic-driven approach accelerates the query processing considerably. This acceleration is due to the fact that we can dynamically switch between a further use of the index structure and a linear scan. Our statistic-driven approach adapts the access method continuously to the best of these two worlds.

In our future work, we want to show that our statistic-based acceleration approach can fruitfully be applied to time critical applications as for instance Virtual Reality applications Furthermore, we plan to apply our statistic driven query processing for dynamic spatial queries.

References

[1] Berchtold S., Böhm C., Kriegel H.-P., Michel U.: *Implementation of Multidimensional Index Structures for Knowledge Discovery in Relational Databases.* Proc. 1st Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK), LNCS 1676: 261-270, 1999.

[2] DeFazio S., Daoud A., Smith L. A., Srinivasan J.: *Integrating IR and RDBMS Using Cooperative Indexing.* Proc. 18th ACM SIGIR Conference on Research and Development in Information Retrieval: 84-92, 1995.

[3] Freytag J.-C., Flasza M., Stillger M.: *Implementing Geospatial Operations in an Object-Relational Database System.* Proc. 12th Int. Conf. on Scientific and Statistical Database Management (SSDBM): 209-219, 2000.

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

[5] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns.* Addison Wesley Longman, Boston, MA, 1995.

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

[7] Kriegel H.-P., Pfeifle M., Pötke M., Seidl T.: *A Cost Model for Interval Intersection Queries on RI-Trees.* Proc. 14th Int. Conf. on Scientific and Statistical Database Management (SSDBM), Edinburgh, Scotland, pp. 131-141, 2002

[8] Kriegel H.-P., Pfeifle M., Pötke M., Seidl S.: *Spatial Query Processing for High Resolutions.* Proc. 8th Int. Conf. on Database Systems for Advanced Applications (DASFAA'03), Kyoto, Japan, 2003, pp. 17-26

[9] 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 2121: 481-501, 2001.

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

[11] Ravi Kanth K. V., Ravada S., Sharma J., Banerjee J.: *Indexing Medium-dimensionality Data in Oracle.* Proc. ACM SIGMOD Int. Conf. on Management of Data: 521-522, 1999.

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

[13] Stonebraker M., Frew J., Gardels K., Meredith J.: *The SEQUOIA 2000 Sorage Benchmark.* Proc. ACM SIGMOD Int. Conf. on Management of Data: 1993.

[14] Seeger B., Larson P., McFadyen R.: *Reading a Set of Disk Pages.* Proc. 19th Int. Conf. on Very Large Databases (VLDB): 592-603, 1993.

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

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