Proc. 9th International Conference on Database Systems for Advanced Applications (DASFAA 2004), Jeju, South Korea, pp. 169-183

Statistic Driven Acceleration of Object-Relational Space-Partitioning Index Structures

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 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 the access methods themselves by reducing the number of generated join partners which results in fewer 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 20% and 10,000% for spatial collision queries on the Relational Interval Tree and on the Relational Quadtree.

1 Introduction

The efficient management of complex objects has become an enabling technology for many novel database applications, including computer aided design (CAD), medical imaging or 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 [4], 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 [11] 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 intersection queries performed on the Relational Quadtree (RQ-tree) and the Relational Interval Tree (RI-tree). A rough sketch of the presented idea can be found in [3]. For a comprehensive overview about Relational Access Methods and how to integrate them into modern ORDBMs, we refer the reader to [6].

The remainder of this paper is organized as follows. In Section 2, we show how we can use the already existing statistics to accelerate the query process. In addition to the known error- and size-bound decomposition approaches, we present a new statis-

tic-bound decomposition approach for decomposing spatially extended objects. Exemplarily, we show how to adopt our new approach to the RQ-tree and the RI-tree. In Section 3, we present convincing experimental results and conclude the paper with a few remarks on future work.

2 Statistic based Acceleration of Relational Access Methods

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 [8]. 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 [8] 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 spatial relational index structures. Then we show how we can benefit from the statistics, used by the cost-models belonging to a relational access method. Finally, in Section 2.3 we introduce a new statistic-based decomposition approach in addition to the existing error- and size-bound decomposition approaches. In all three sections, we first introduce our ideas in general and then show how to adapt them to specific relational index structures.

2.1 Statistics Related to the Relational Access Method

We start with a definition common to all relational access methods:

Definition 1 (*Relational Access Method*) [6]

An access method is called a *relational access method*, iff any index-related data is 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 is called *user data*, *index data*, and *meta data*.

As already indicated in the above definition, 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 assume for instance that one half of the data space is completely empty, and we carry out a box volume query in this area, we can omit a lot of unnecessary I/O accesses if we take the actual data distribution into consideration. Consequently, it is beneficial if we store the variable data extension along with the fixed data space extension in the metadata table.

In Table 1, we summarized some optimizations which are suitable for the RI-tree and the RQ-tree. These simple statistics are especially useful for indexing extended spatial

Table 1. Simple Statistics for the RI-tree and the RQ-tree

denotation	explanation
*MaxNodeLeve*l MinNodeLevel (RI-tree)	These two parameters reflect the highest and lowest level of the *fork-nodes* of the intervals in the database. If we arithmetically traverse the primary structure for a given query interval q = (*l*,*u*), we only have to collect those nodes *n* as join partners, for which *MinNodeLevel* ≤ *Level* (*n*) ≤ *MaxNodeLevel* holds.
MaxLeftDist MaxRightDist (RI-tree)	These two parameters reflect the maximum distance of the boundary values of any database interval to its corresponding *fork-node*. If we arithmetically traverse the primary structure for a given query interval q = (*l*,*u*), we only have to collect those nodes *n* as join partners, for which *n-MaxLeftDist* ≤ *u* and *n+MaxRightDist* ≥ *l* holds.
*MaxTileLeve*l MinTileLevel (RQ-tree)	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* ≤ *Level* (*t*) ≤ *MaxTileLevel* holds.

objects. If we use the RI-tree or 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 or intervals [5] (cf. Section 3).

2.2 Statistics Related to the Built-in Index Structure

In [4], it was shown that using quantiles (‘equi-count histograms’) is more suitable for estimating the selectivity and the corresponding I/O cost than using 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 a relational index structure, e.g. RI-tree or 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.

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 1a) 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 elements r_i can be regarded as a linear ordered list $L(I) = \langle r_1, \ldots, r_N \rangle$ for which $r_1 \leq \ldots \leq r_N$ 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 $Seq_{Q,I} = (\langle s_1, \ldots, s_n \rangle)$ 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*) $C_n^{I/O}(s) = C_n^{I/O}(p(r'))$ (*navigational cost*)

(*ii*) $C_s^{I/O}(s) = C_s^{I/O}(\langle p(r'), \ldots, p(r'') \rangle)$ (*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 $Seq_{Q,I} = (\langle s_1, \ldots, s_n \rangle)$ are determined by $C^{I/O}(Seq_{Q,I}) = \sum_{i=1}^{n} C^{I/O}(s_i)$.

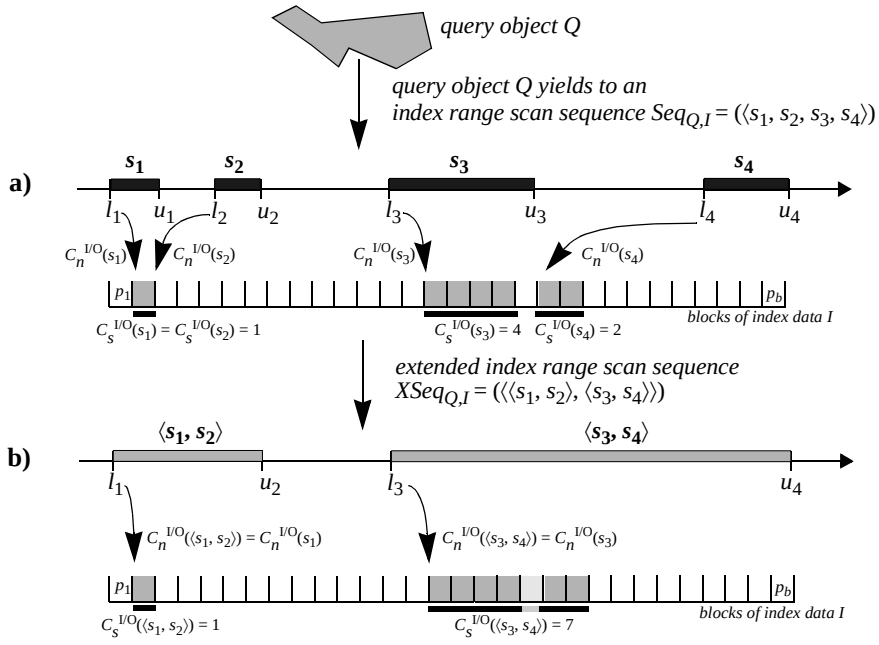

Figure 1. Accelerated query processing
a) Index range scan sequence **b)** Extended index range scan sequence

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 1b. 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*) $C_n^{I/O}(xs) = C_n^{I/O}(s_r)$ (*navigational cost*)

(*ii*) $C_s^{I/O}(xs) = C_s^{I/O}(l_r, u_s)$ (*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: $C^{CPU}(xs) = C^{CPU}(\langle r', .., r'' \rangle)$, 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 $C(XSeq_{Q,I}) << C(Seq_{Q,I})$ 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 $(q_0, \ldots, q_v)$ (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.

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

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 $s_1, ..., s_n$ of the index range scan sequence $Seq_{Q,I} = (\langle s_1, ..., s_n \rangle)$, whether we close this gap or skip it. Thus we obtain an *extended index range scan* sequence $XSeq_{Q,I} = (\langle\langle s_{i_0+1}, .., s_{i_1} \rangle, ..., \langle s_{i_{m-1}+1}, .., s_{i_m} \rangle\rangle)$, which satisfies the following property:

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

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 following, we will show how our approach can be applied to the intersect predicate for specific index structures.

Adoption to the RQ-tree. The classical example for a space partitioning relational access method is the *Relational Quadtree* [10]. In this section, we shortly introduce our approach based on the basic idea of the Linear Quadtree according to the in-depth discussion of Freytag, Flasza and Stillger [1].

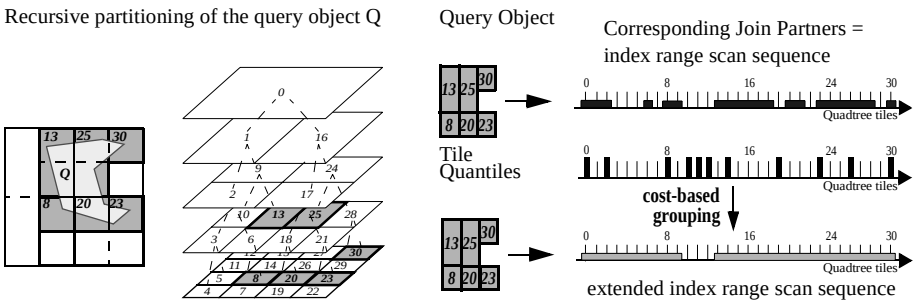

Figure 2. Cost-Based Tile Grouping

Assume object *Q* in Figure 2 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 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 2.

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 3). 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*().

Adoption to the RI-tree. Similarly to the RQ-tree, we can integrate the cost-based grouping algorithm into the procedural query preparation step of the RI-tree. This

```
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 3. Accelerated window query on a Relational Quadtree

grouping algorithm is independent of the high-level relational index-structure. It is only based on a B^+-tree and on a quantile vector. The quantile vector in the case of the RI-tree, is formed by the *fork-nodes* of the intervals stored in the database. This node quantile was also used for an effective and efficient cost-model for intersection queries on RI-trees [4].

2.3 Statistics Related to the Object Decomposition

Both the error- and size-bound decomposition approach for spatially extended objects lead to a sequence of simple query objects, e.g. a sequence of tiles, intervals or boxes. In this section, we introduce an additional decomposition approach which decomposes the object based on the expected I/O cost. The expected I/O cost can be estimated in the same way the optimizer estimates the cost for a given query. Like the approach of the last section, the decomposition of the query object is controlled by the statistics which are available for free and maintained by the cost model.

Figure 4 depicts this top down grouping algorithm which is beneficial for all of the discussed index structures. The algorithm starts with a query object comprising the complete object. In each step, we determine the maximum included gap and split along this gap resulting in a sequence of query objects. Then we estimate the I/O cost related to the original query object and the cost related to the sequence. If the cost of the original query object is smaller than the cost of the sequence, we terminate the algorithm. The query object now consists of a sequence of query objects. In an additional refinement step, we eliminate the false hits, which result from the fact that we have not decomposed the spatial object with the maximum possible accuracy.

Box Volume queries. The introduced approach is especially useful for highly selective box volume queries on the RI-tree or the RQ-tree. The traditional error-and size bound decomposition approaches [9] decompose a large query object into smaller query objects optimizing the trade off between accuracy and redundancy. In contrast, the idea of

```
QV: quantile vector
Q:  query object

Decompose(Q, QV)
{ query_sequence_list := split_at_maximum_gap(Q);
  cost0 := statistic_look_up(Q, QV);
  costdec := 0;
  for each q in query_sequence_list do
    costdec := costdec + statistic_look_up(q, QV);
  if cost0 > costdec then
    for each q in query_sequence_list do
      Decompose(q, QV);
  else
    report(Q); }
```

Figure 4. Grouping Algorithm *Decompose*

taking the actual data distribution into account in order to decompose the query object, leads to a new *selectivity-bound decomposition* approach, which tries to minimize the overall number of logical reads. We decompose a query box dependent on the stored data. If there are not many data stored in the query area, the box is decomposed into comparable few simple query objects, i.e. tiles or intervals. On the other hand, if the query returns a lot of results, we decompose the query into comparably many simple query objects.

A box can be described by a few parameters, e.g. by two points. The few parameters which are necessary to describe the box are attached to each incompletely decomposed interval or tile. In the refinement step, we further decompose the query intervals or tiles on demand from the compact geometric information.

3 Experimental Evaluation

The tests are based on two test data sets *CAR* and *PLANE*. These test data sets were provided by our industrial partners, a German car manufacturer and an American plane producer, in form of high-resolution voxelized three-dimensional CAD parts. The CAR dataset consists of approximate 14 million voxels and 200 parts, whereas the PLANE dataset consists of about 18 million voxels and 10,000 parts. The CAR data space is of size 2^{33} and the PLANE data space is of size 2^{42}. In both cases, the Z-curve was used as a space filling curve to enumerate the voxels.

We have implemented our approach for the RI-tree and the RQ-tree on top of the Oracle9*i* 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.

3.1 Histograms of the Test Data Sets

Figure 5 depicts the interval and gap histograms for our two test data sets. Both test data sets consist of many short intervals and short gaps and only a few longer ones. Consequently, mainly the lowest levels of the RQ-tree and RI-tree contain index entries. Figure 6a shows that in the case of the RQ-tree on the CAR data set only the seven lowest

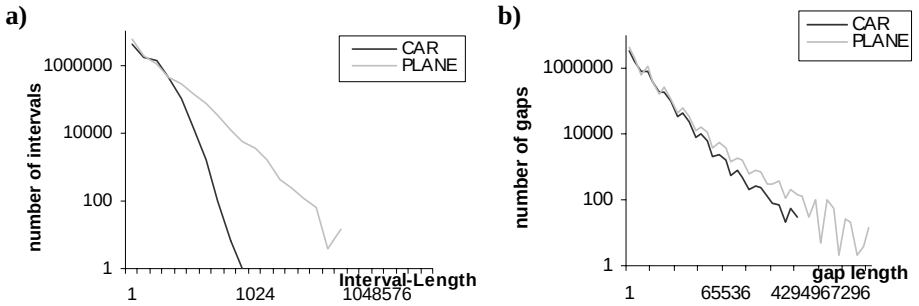

Figure 5. RI-tree histograms **a)** Intervals **b)** Gaps

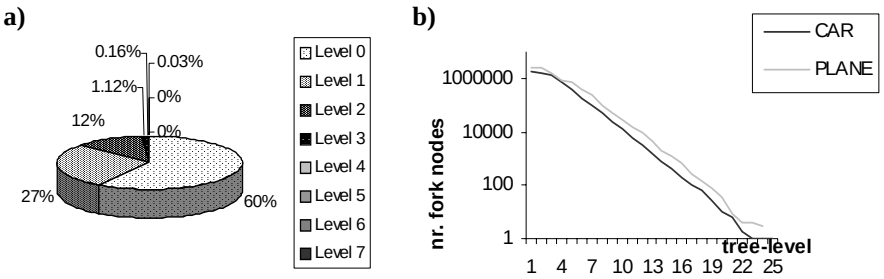

Figure 6. Used index levels
a) tile levels (RQ-tree (CAR)) **b)** *fork-node* levels ((RI-tree) (CAR & PLANE))

of 33 levels are occupied. Similar observations hold for the RI-tree (cf. Figure 6b) where most intervals are registered at very low *fork-node* levels. The observation that spatial objects are decomposed into many small intervals and tiles are not confined to our two test data sets but hold for spatial objects in general [2] [5]. Therefore, the statistics presented in Section 2.1 are very beneficial for efficient query processing on spatially extended objects in general.

3.2 Query Processing

In this section, we examine the benefits of using extended index range scans. For the RI-tree and the RQ-tree, we used 10% of the database objects as query objects and report the average results from these queries.

Extended range scans without statistics. In a first experiment, which does not use any statistical information, we point out the benefits of using our extended index range scans (cf. Section). For a given query object, we did not collect all possible join partners, but omitted the last levels and used an extended index range scan instead. Figure 7a shows that the number of join partners decreases with an increasing number of scanned tree levels. At the beginning, the number of logical reads also decreases, but if we neglect too many tree levels of the RI-tree the number of logical reads increases again along with the increasing number of physical reads. The number of physical reads stays almost constant if we scan over only a small number of levels. On the other hand, the number of physical reads dramatically increases if the number of scanned tree levels

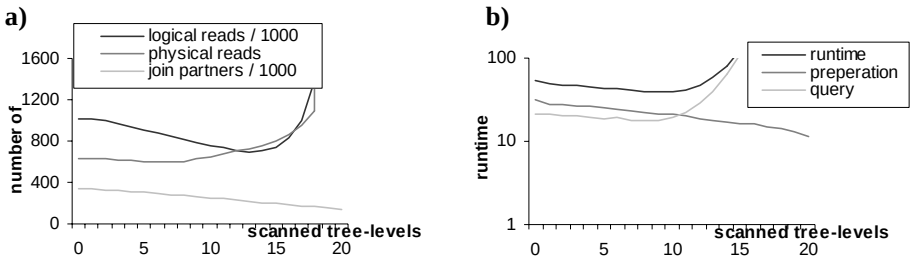

Figure 7. RI-tree optimizations without using statistics

exceeds 18 because of the increasing number of false hits which are filtered out in a consecutive filter step. In Figure 7b it is shown that the preparation time decreases with an increasing number of scanned tree levels. Due to the reduced number of join partners and the decreasing preparation time, the overall runtime reaches a minimum if we neglect the last 10 levels of the RI-tree and apply an extended range scan instead. By using a fixed scan level, we can already improve the query response time by 30%. In the following sections, we will see that if we use statistics to form our extended range scans, we can further improve the overall query response behavior.

Extended range scans with statistics. In Figure 8, it is shown in detail that our new statistic-based approach accelerates both the basic variant of the RI-tree [7] and the variant which is optimized for efficient handling of sequences [8]. Figure 8 depicts that we can reduce the number of logical reads approximately by an order of magnitude if we exploit the available statistics. This reduction is achieved without increasing the number of physical reads so that the overall runtime decreases. If we use the statistics we outperform the simple scanning approach even for the optimum scanning level (cf. Figure 7). In all our tests, we accelerate the query process by 20% to 150% if we form the extended range scans according to the available statistics.

In the next experiments, we applied the statistic based approach to the RQ-tree (cf. Figure 9). Figure 9a shows that the use of our quantile statistics (cf. Section 2.2) accelerates the RQ-tree by about 200%. A further improvement can be achieved by using the information of the highest and lowest level of stored tiles within the database (cf. Section 2.1), leading to a speed-up of almost 300%. Figure 9b depicts the acceleration of the

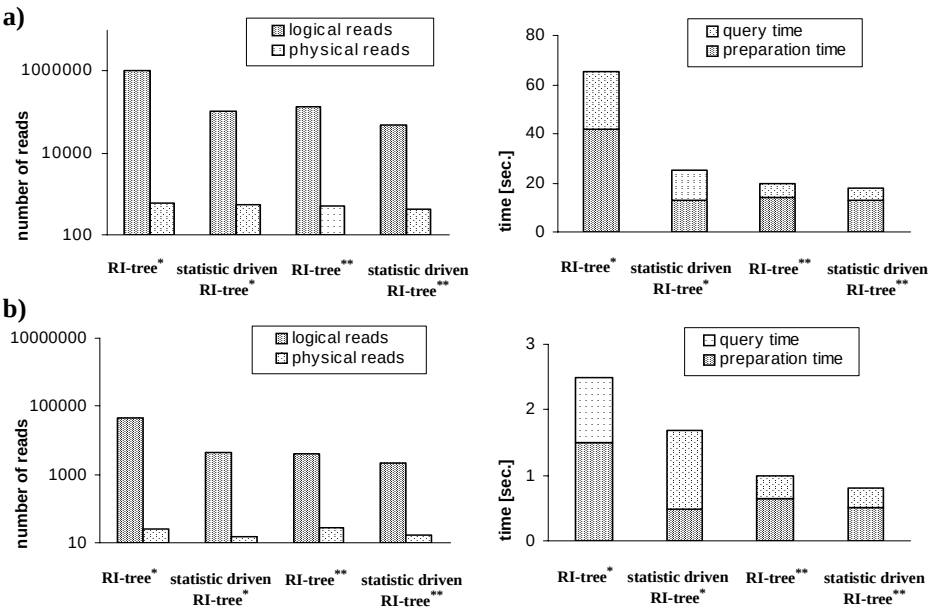

Figure 8. Acceleration for two variants of the RI-tree ([7]*, [8]**)
a) *CAR* dataset **b)** *PLANE* dataset

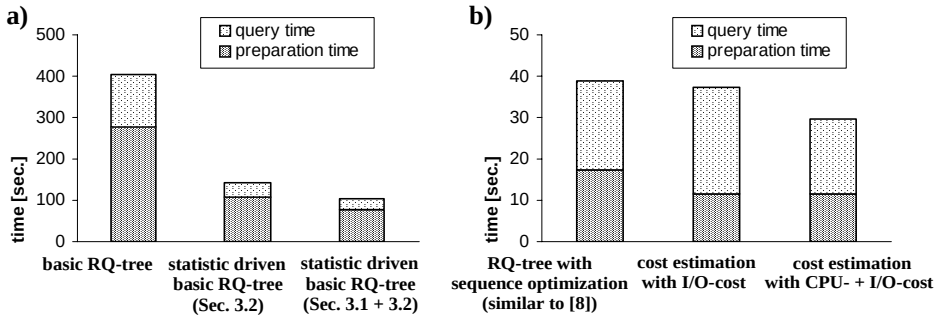

Figure 9. Statistic based accelerated RQ-tree on the CAR dataset
a) Runtime for the basic RQ-tree, the RQ-tree optimized according to Section 3.2, and the RQ-tree optimized according to Section 3.1 and 3.2
b) Runtime for the RQ-tree optimized for sequences (similar to [8])

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 9b 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-up of approximately 30%, even for this highly specialized index structure.

To sum up, similar to the experiments related to the RI-tree, 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.

Statistic based decomposition. In a last experiment, we carried out different box volume queries on the RI-tree for the PLANE database. Figure 10 depicts the average runtime for three different boxes, where we moved each box to 10 different locations. As shown in Figure 10, our statistic-based decomposition approach can improve the query response behavior up to 10,000%, i.e. by two orders of magnitude, compared to the granularity-bound approach. This speed up is mainly due to the reduced decomposition time. On the other hand, the query response time does not suffer from the fact that we did not decompose the boxes with the maximum possible accuracy. The time we need for the additional refinement step to filter out false hits is compensated by the much smaller number of query intervals resulting from a coarser decomposition of the query box. To sum up, our statistic-based decomposition approach is especially useful for commonly used box volume queries.

4 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 ap-

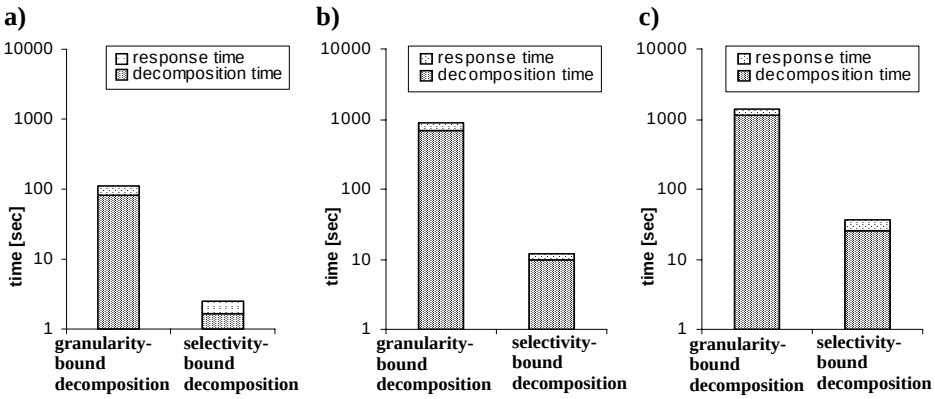

Figure 10. Box queries on the *PLANE* data (decomposition and response time)
a) box size equals 0.00002% of data space yielding 0.03% selectivity
b) box size equals 0.003% of data space yielding 0.1% selectivity
c) box size equals 0.008% of data space yielding 1.0% selectivity

proach for the Relational Interval Tree as well as for the Relational Quadtree on top of the Oracle9*i* database system. According to our experiments, we achieved speed-up factors of up to two orders of magnitude. 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 variable 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 with haptic and visual rendering of complex spatial environments. Furthermore, we plan to apply our statistic driven query processing for dynamic queries.

5 References

1. 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.
2. Gaede V.: Optimal Redundancy in Spatial Database Systems. Proc. 4th Int. Symp. on Large Spatial Databases (SSD), LNCS 951: 96-116, 1995.
3. Kriegel H.-P., Kunath P., Pfeifle M., Renz M.: Acceleration of Relational Index Structures Based on Statistics, Proc. 15th Int. Conf. on Scientific and Statistical Database Management (SSDBM), Cambridge, Massachusetts, USA, pp. 258-260, 2003.
4. 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.

5. 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, pp. 17-26, 2003.
6. Kriegel H.-P., Pfeifle M., Pötke M., Seidl T.: The Paradigm of Relational Indexing: A Survey. 10. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web, 2003.
7. 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.
8. 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.
9. Orenstein J. A.: Redundancy in Spatial Databases. Proc. ACM SIGMOD Int. Conf. on Management of Data, 294-305, 1989.
10. Samet H.: Applications of Spatial Data Structures. Addison Wesley Longman, Boston, MA, 1990.
11. Seeger B., Larson P., McFadyen R.: Reading a Set of Disk Pages. Proc. 19th Int. Conf. on Very Large Databases (VLDB): 592-603, 1993.