Scientific and Statistical Database Management (SSDBM 2004), Santorini Island, Greece, 2004.

Efficient Query Processing on Relational Data-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

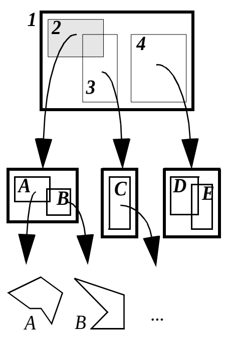
Abstract

In contrast to space-partitioning index structures, data-partitioning index structures naturally adapt to the actual data distribution which results in a very good query response behavior. Besides efficient query processing, modern database applications including computer-aided design, medical imaging, or molecular biology require fully-fledged database management systems in order to guarantee industrial-strength. In this paper, we show how we can achieve efficient query processing on data-partitioning index structures within general purpose database systems. We reduce the navigational index traversal cost by using "extended index range scans". If a directory node is "largely" covered by the actual query, the recursive tree traversal for this node can beneficially be replaced by a scan on the leaf level of the index instead of navigating through the directory any longer. On the other hand, for highly selective queries, the index is used as usual. In this paper, we demonstrate the benefits of this idea for spatial collision queries on the Relational R-tree. Our experiments with an Oracle9i database system show that our new approach outperforms common index structures and the sequential scan considerably.

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 contrast to a query optimizer of an ORDBMS which has to decide "once and for all" whether to include a specific access method into the execution plan, the approach of this paper is much more fine-grained. At each directory node of a hierarchical index structure it is individually decided whether it is beneficial to switch to a range scan on the leaf level of the index or whether it is beneficial to take further advantage of the index-directory. The experiments show that our new approach always adapts to the best of the two worlds "index" and "sequential scan". Therefore, the optimizer can under all circumstances include our new approach into the query execution plan.

a)



b) *polygons_rtree*

page id	page_lev	son_id	son_data (MBR)
ROOT	2	1	BOX((0,0),(200,120))
1	1	2	BOX((0,0),(80,60))
1	1	3	BOX((60,20),(100,120))
1	1	4	BOX((140,20),(200,120))
2	0	A	…
2	0	B	…
…	…	…	…

Figure 1: Relational mapping of an R-tree directory
a) Hierarchical directory, **b)** Index table

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 [4] must know the actual position of the pages on the storage media.

In this paper, we introduce a new approach based on index inherent statistics on top of an ORDBMS exemplarily for spatial intersection queries performed on the Relational R-tree. In [2] it is also shown how our approach can be adapted to similarity range queries and *k*-nn queries on the Relational M-tree. For more details about relational indexing, we refer the reader to [1].

The Relational R-tree. In this paragraph, we shortly introduce *Relational R-trees*, like they have been used by the Oracle developers Ravi Kanth et al. [3]. Figure 1 depicts a hierarchical R-tree along with a possible relational mapping (*page_id*, *page_lev*, *son_id*, *son_data*). The column *page_id* contains the logical page identifier, while *page_lev* denotes its level in the tree. Thereby, 0 marks the leaf level of the directory. The attribute *son_id* contains the *page_id* of the connected entry, while *son_data* stores its minimum bounding rectangle. To support the navigation through the R-tree table at query time, a built-in index can be created on the *page_id* column.

The remainder of this paper is organized as follows. In Section 2, we present our new indexing approach which combines the advantages of a recursive tree traversal and a sequential scan. In Section 3, we present experimental results and conclude the paper with a few remarks on future work in Section 4.

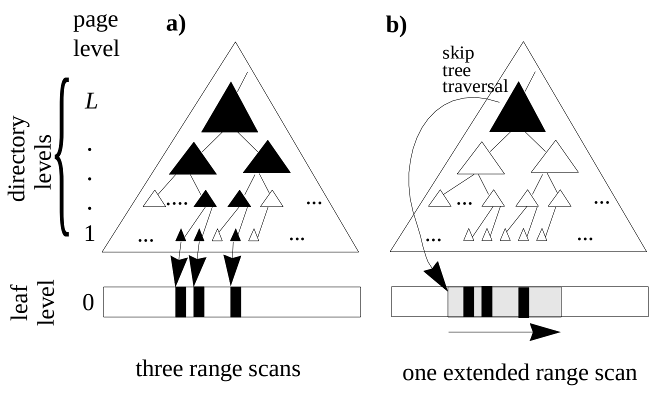


Figure 2: General Idea
a) Navigational approach, **b)** Scanning approach

2. Acceleration of Relational Access Methods

We assume that the *page_ids* are ordered according to a depth-first tree traversal and that we have a B$^+$-tree on this attribute. Furthermore, we assume that an additional B$^+$-tree exists on the attributes *page_level*, *page_id* so that we can easily scan over all data entries, i.e. all entries where the page_level is 0. The general idea is that we skip the recursive tree traversal at a certain point and perform an extended range scan on the leaf-level of our index. Thereby, we try to minimize the overall navigational cost on the hierarchical index while allowing to read false hits from the leaf-level of the index which are filtered out by a subsequent refinement step. Figure 2b depicts this general idea. The main advantages of our new approach is that we can reduce the navigational cost related to the hierarchical index structure (filled triangles in Figure 2) and to the built-in B$^+$-trees (arrows in Figure 2). On the other hand, we have higher cost related to the scanning of the leaf-level and higher CPU cost related to the additionally required refinement step.
In this section, we will discuss the cost related to a hierarchical tree traversal and the cost related to an extended range scan in general. Based on this reasoning, we will present a heuristic for intersection queries on the Relational R-tree which helps to answer the crucial question when to abort the recursive tree traversal and switch to an extended range scan. In Section 2.1, we will introduce the general I/O cost and CPU cost related to a certain directory node which presumably arise when continuing the tree traversal (cf. Figure 2a) and the cost related to an extended range scan starting at this directory node (cf. Figure 2b). These cost heavily depend on the "overlap-factor" $\sigma = \sigma(q, n)$ which denotes the percentage of accessed tuples during a query q in a certain subtree of a directory node n if the index structure is used as usual. In the following sections, we will show how we can estimate this "overlap-factor" σ for the intersect predicate on the Relational R-tree (cf. Section 2.2). We use the following notations:

symbol	meaning
m	average number of index entries per directory node
b	average number of index entries per disk page
$L(n)$	level of the current directory node n
h_B	height of the B$^+$-tree
k_{CPU}	CPU-cost for testing one index directory entry
$k_{I/O}$	I/O-cost for reading one page from the disk
$\sigma(q, n)$	value between 0 and 1 which denotes the percentage of accessed tuples in the subtree belonging to node n, if the index structure is used as usual for the query processing of a query q
$cost_{NAVI}(q, n)$	the navigational cost related to a node n and a query q when further using the hierarchical index
$cost_{SCAN}(q, n)$	the scanning cost related to a node n and a query q when applying an extended range for n

Our reasoning is based on the assumptions that we have a uniform data distribution and uniformly filled nodes. If this is not the case, we can improve the estimation by storing the data distribution, the actual number of directory nodes, and the number of leaf-nodes beneath a certain directory node along with this directory element. For the sake of clarity, we refrain from this more complex approach, and assume that we have a uniform data distribution and that all nodes are uniformly filled.

2.1. General Approach

We will first discuss the cost related to the navigational approach (cf. Section 2.1.1), before we look at the cost related to the scanning approach (cf. Section 2.1.2). In Section 2.1.3, we introduce our final combined approach which exploits the advantages of the navigational and the scanning approach.

2.1.1. Navigational Approach

The cost related to a directory node n when using the hierarchical index structure without further modifications for a query q (cf. Figure 2a) consist of an I/O- and a CPU-part and can be expressed as follows:

$$cost_{NAVI}(q, n) = cost_{NAVI}^{I/O}(q, n) + cost_{NAVI}^{CPU}(q, n)$$

In the following, we will discuss the detailed I/O- and CPU-cost of the navigational approach.

I/O-cost. We have to access *cnt_n* directory nodes:

$$cnt\_n_{NAVI}^{I/O}(q, n) = 1 + \sigma(q, n) \cdot \sum_{i=1}^{L(n)-1} m^i$$

Each of these nodes has m entries. For locating these nodes on the disk we use a built-in B$^+$-tree which has a height of h_B. Additional to the navigational cost on the B$^+$-tree, we have cost related to the reading of $cnt\_t_{NAVI} = m \cdot cnt\_n^{I/O}_{NAVI}$ index entries, i.e. tuples, distributed over cnt_t_{NAVI}/b disk pages. We penalize each page read with a factor $k_{I/O}$. To sum up, we have the following I/O-cost:

$$cost^{I/O}_{NAVI}(q,n) = k_{I/O} \cdot \left(h_B + \frac{m}{b}\right) \cdot \left(1 + \sigma(q,n) \cdot \sum_{i=1}^{L(n)-1} m^i\right)$$

CPU-cost. The CPU-cost related to the evaluation of cnt_t_{NAVI} index entries are:

$$cost^{CPU}_{NAVI}(q,n) = k_{CPU} \cdot m \cdot \left(1 + \sigma(q,n) \cdot \sum_{i=1}^{L(n)-1} m^i\right)$$

2.1.2. Scanning Approach

If we scan all data belonging to a directory node n on level $L(n)$, the following cost occur:

$$cost_{SCAN}(q,n) = cost^{I/O}_{SCAN}(q,n) + cost^{CPU}_{SCAN}(q,n)$$

The detailed I/O-and CPU-cost are as follows:

I/O-cost. We have to locate the starting point of the scanning area once by using a B$^+$-tree. Then, we read $m^{L(n)}$ index entries of the leaf level distributed over $m^{L(n)}/b$ disk pages. Again, we penalize each page read with a factor $k_{I/O}$. To sum up, we have the following I/O-cost for the scanning approach:

$$cost^{I/O}_{SCAN}(q,n) = \left(h_B + \frac{m^{L(n)}}{b}\right) \cdot k_{I/O} \approx \frac{m^{L(n)}}{b} \cdot k_{I/O}$$

CPU-cost. The cost related to the evaluation of $m^{L(n)}$ values on the leaf-level are $k_{CPU} \cdot m^{L(n)}$. Thus we have the following CPU-cost for the scanning approach:

$$cost^{CPU}_{SCAN}(q,n) = m^{L(n)} \cdot k_{CPU}$$

2.1.3. Combined Approach

Our approach starts with applying the navigational approach. For each visited node we estimate the navigational and the scanning cost. If $cost_{SCAN}(q,n) < cost_{NAVI}(q,n)$, we abort the recursive tree traversal and apply an extended range scan. This mixed approach is a kind of greedy approach, which tries to combine the advantages of the navigational and the scanning approach.

The main point in accurately estimating $cost_{SCAN}(q,n)$ and $cost_{NAVI}(q,n)$ is to forecast the overlap-factor σ as precise as possible. For each hierarchical index structure such a selectivity estimation function has to be provided for the optimizer anyway. When the execution plan for a given query

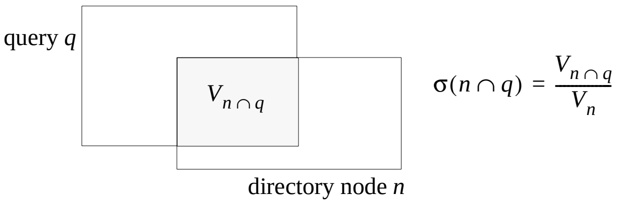


Figure 3: Determination of the overlap-factor σ for the intersect predicate on the Relational R-tree

q is determined, the optimizer evokes σ (q, n_{root}) in order to decide whether to include this index in the query execution plan or not. We propose to evoke this selectivity estimation function for each visited directory node in order to decide whether to use the tree directory further or to switch to an extended range scan. Let us note that our approach inherently benefits from a good selectivity estimator which can be used as black box by our new indexing method. Nevertheless, we will present a heuristic which aims at estimating the selectivity efficiently and effectively for the collision queries on the Relational R-tree. Needless to say that you can also use more sophisticated selectivity estimation functions to get better results. The main point of this paper is to show that already simple selectivity estimations suffice to accelerate the query processing considerably.

2.2. Accelerated Relational R-Tree

In this section, we adapt the concept presented in Section 2.1 to the intersect predicate on the Relational R-tree.

The overlap-factor σ (q, n) can easily be determined as shown in Figure 3. The overlap-factor σ (q, n) is equal to the ratio of the intersection volume $V_{n \cap q}$ between the query object q and the directory node n and the hyper-volume V_n of the directory node.

$$\sigma(q,n) = \frac{V_{n \cap q}}{V_n}$$

As the operation whether two boxes intersect or not can be performed very efficiently, we neglect the CPU cost and concentrate in this section on the accruing I/O cost. Thus we perform an extended index range scan for a directory node n on level L(n) and a query q if

$$cost^{I/0}_{SCAN} < cost^{I/0}_{NAVI}$$

i.e.

$$\left(h_B + \frac{m^{L(n)}}{b}\right) < \left(h_B + \frac{m}{b}\right) \cdot \left(1 + \sigma(q,n) \cdot \sum_{i=1}^{L(n)-1} m^i\right)$$

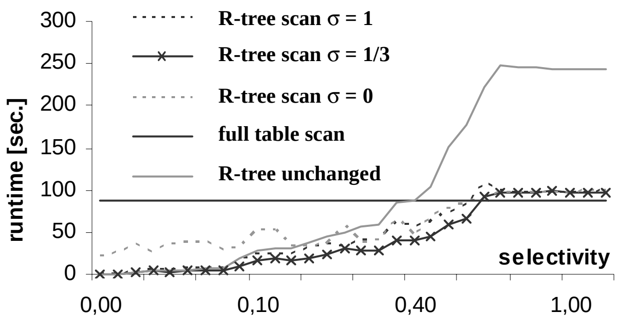


Figure 4: Relational R-tree for varying selectivity.

If we assume rather high values of m, a significant overlap factor $\sigma(q, n)$ and a directory level $L(n)$ higher than 2, we scan if the following simplified condition is fulfilled:

$$\frac{m^{L(n)}}{b} < \left(h_B + \frac{m}{b}\right) \cdot \sigma(q, n) \cdot m^{L(n)-1}$$

Or, slightly modified, we scan if:

$$1 < \sigma(q, n) \cdot \left(1 + \frac{b \cdot h_B}{m}\right)$$

If m is equal to b, i.e. we do not use the “supernode” concept of the X-tree [5] (cf. Section 3.1), and we assume that we have to perform two reads for navigating through the B^+-tree directory, it is beneficial to scan if the overlap-factor is higher than 1/3. Note that the resulting simplified formula is independent of the actual level of the directory nodes.

3. Experimental Evaluation

The tests are based on a test data set *CAR*, provided by our industrial partner, a German car manufacturer. It consists of approximately 1,400,000 high-resolution voxelized three-dimensional CAD parts managed by a Relational R-tree of height 5.

We have implemented our approach for the Relational R-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 the machine was used exclusively by one active session.

Intersection Queries on the Relational R-tree. We applied our new scanning approach proposed in Section 2 to the Relational R-tree for varying overlap-factors σ which were compared to a full table scan and an unchanged R-tree implementation. Figure 4 shows that the best results for a large range of selectivity parameters was obtained by using an overlap-factor of σ = 1/3 which is identical to the theoretically derived value (cf. Section 2.2). For σ = 0, an extended range scan is triggered as soon as the query box intersects a directory box. This results in rather high query response times for highly selective queries compared to the original R-tree. On the other hand, a parameter σ = 1 forces an extended range scan, if the directory node is completely covered by the query object resulting in rather good query response times over the complete range of selectivity parameters. Figure 4 shows that the decision whether to use the directory of the relational R-tree any longer or to switch to an extended range scan can be decided for each node with negligible overhead. Our combined approach can improve the overall query response time by more than 150% for queries of low selectivity compared to the navigational approach (R-tree). Furthermore, for highly selective queries our combined approach outperforms the sequential scan by more than 10,000%. To sum up, the combined approach naturally adapts to the best of the two worlds: “index” and “sequential scan”.

4. Conclusion

In this paper, we presented a new technique which uses the hierarchical directory of a relational index structure as long as it makes sense. At each directory node, it is individually decided whether it is beneficial to switch to a range scan on the leaf level of the index or whether it is beneficial to take further advantage of the index-directory. This new approach is contrary to the approach used by query optimizers which have to decide “once and for all” whether to include a specific access method into the execution plan. We introduced our approach in general as well as exemplarily for spatial intersection queries on the Relational R-tree. Our experimental evaluation showed that our new approach adapts to the best of the two worlds, “index” and “sequential scan”. Therefore, the optimizer can under all circumstances include this new fine-grained approach into any query execution plan.

References

[1] 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.

[2] Kriegel H.-P., Kunath P., Pfeifle M., Renz M.: *Efficient Query Processing on Relational Data Partitioning Index Structures*. Technical Report, University of Munich, 2004.

[3] 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.

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