Stochastic Driven Relational R-Tree

HANS-PETER KRIEGEL, PETER KUNATH, MARTIN PFEIFLE, MARCO PÖTKE, MATTHIAS RENZ, PETRA-MARIA STRAUß
University of Munich, Germany, {kriegel, kunath, pfeifle, renz, strauss}@dbs.informatik.uni-muenchen.de
sd&m AG software design & management, marco.poetke@sdm.de

Abstract. Modern spatial database applications including computer-aided design (CAD), medical imaging, molecular biology, or geographical information systems (GIS) impose new requirements on spatial query processing. Particular problems arise from the design goal to use general purpose database management systems in order to guarantee industrial-strength. Recently, there has been an increasing awareness that it is indispensable to integrate stand-alone spatial index structures, e.g. R-trees or Quadtrees, into fully-fledged database systems resulting in relational index structures, e.g. Relational R-trees or Relational Quadtrees. In this paper, we introduce stochastic heuristics for the Relational R-tree which are based on the fact that the Relational R-tree allows an individual fanout for each node. This freedom from minimal and maximal fill factors of nodes, offers a wide range of potential improvements. We develop algorithms that consider the quality of the entries of a node rather than just the quantity. Our experiments clearly demonstrate the advantages of this new stochastic driven Relational R-tree compared to the Relational R*-tree.

1 Introduction

Modern geographical information systems have to manage huge amounts of data effectively and efficiently. According to a directive of the European Union, for example, almost all environmental data that is stored at public agencies have to be made available to any citizen on demand [4]. This requirement can only be met by providing systems which allow efficient and concurrent retrieval of spatial data. Therefore, we need suitable index structures which have to be integrated into fully-fledged database systems. In [3] it was shown that the R*-tree is especially useful for highly selective queries. Furthermore, there exist commercial systems which have already integrated R-trees into their kernels [12, 13].

For commercial use, a seamless and capable integration of temporal and spatial indexing into industrial-strength databases is essential. Fortunately, a lot of traditional database servers have evolved into Object-Relational Database Management Systems (ORDBMS). This means that in addition to the efficient and secure management of data ordered under the relational model, these systems now also provide support for data organized under the object model. Object types and other features, such as large objects (LOBs), external procedures, extensible indexing, user-defined aggregate functions and query optimization, can be used to build powerful, reusable server-based components.

Relational index structures are designed to operate on relations rather than on dedicated disk blocks. The persistent storage and block-oriented management of the relations is delegated to the underlying database server. Therefore, the robust functionality of the database kernel including concurrent transactions and recovery can potentially be reused.

In an ORDBMS, the user has no access to the exact information where the blocks are located on the disk. This information is hidden from the user, who can only access the data via SQL. In this environment, we are dealing with virtual pages which introduces a whole new abstraction level. This new abstraction level frees us from physical constraints, such as block sizes. In this paper, we show how this new concept helps to accelerate the Relational R-tree [12] which we call RR-tree throughout the rest of this paper.

The remainder of this paper is organized as follows. To establish the necessary fundamentals, we first provide a short introduction into the area of relational access methods [10]. In Section 3, we introduce the basic RR-tree. In Section 4, we discuss the theoretical foundation for a stochastic driven RR-tree. Based on the thereby derived formulas, we introduce algorithms for the insert, delete and update operations of the RR-tree in Section 5. In Section 6, we compare our new stochastic driven RR-tree to the Relational R*-tree which is outperformed considerably. We summarize our work in Section 7 and close by shortly sketching a few further potential improvements for the RR-tree.

2 Relational Access Methods

In this section, we discuss the basic properties of relational access methods with respect to the storage of index data and query processing. For more detail, we refer the interested reader to [10]. We start with a definition common to all relational access methods:

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

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 a simple example for indexing two-dimensional polygons by using minimum bounding rectangles (*MBRs*). 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 *polygons_mbr* is 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 *mbr_index_metadata* (cf. Figure 1c).

a) *polygons_rtree*

page id	page_lev	son_id	son_mbr
ROOT	3	1	BOX((0,0),(200,120))
1	2	2	BOX((0,0),(80,60))
1	2	3	BOX((60,20),(100,120))
1	2	4	BOX((140,20),(200,120))
2	1	5	…
2	1	6	…
5	0	A	…
6	0	B	…
…	…	…	…

b)

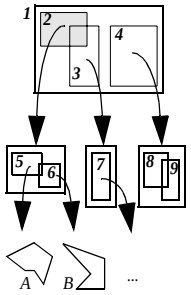

Figure 2: Relational mapping of a R-tree directory **a)** index table **b)** Hierarchical directory

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 [5]. This basic approach of relational indexing has already been applied in many existing solutions, such as Linear Quadtrees and the Relational R-trees for GIS databases or more generally for all kinds of spatial databases [6, 12, 13]. Furthermore, the concept was applied to the Relational X-tree [1] for high-dimensional nearest-neighbor search, or inverted indexes for information retrieval on text documents [5].

3 The Relational R-tree

The *Relational R-tree* was first introduced by the Oracle developers Ravi Kanth et al [12]. Figure 2 depicts a hierarchical R-tree along with a possible relational mapping (*page_id*, *page_lev*, *son_id*, *son_mbr*). The column *page_id* contains the logical page identifier, while *page_lev* denotes its level in the tree. Thereby, 0 marks the level of the data objects and 1 marks the leaf level of the directory. The attribute *son_id* contains the *page_id* of the connected entry, while *son_mbr* stores its minimum bounding rectangle. Thus, *page_id* and *son_id* together comprise the primary key. In our example, the logical page *2* represents a partition of the data space which con-

```
SELECT son_id AS id FROM polygons_rtree
WHERE page_lev = 0
START WITH page_id = ROOT
CONNECT BY
    PRIOR son_mbr INTERSECTS BOX((0,0),(100,100))
    AND PRIOR son_id = page_id;
```

Figure 3: Recursive window query on a Relational R-tree using Oracle SQL

tains the polygons *A* and *B*. The corresponding index row (*1*,*2*,*2*,..) is therefore logically associated with rows (*A*,...) and (*B*,..) in the *polygons* user table (cf. Figure 1).

To support efficient navigation through the relational R-tree table *polygons_rtree* at query time, a built-in index can be created on the *page_id* column. Alternatively, the schema can be transformed to NF^2 (non-first normal form), where *page_id* alone represents the primary key, and a collection of (*son_id*, *son_mbr*) pairs is stored with each row. In this case, the static storage location of each tuple can be used as *page_id*, avoiding the necessity of a built-in index. A primary filter for a window query using SQL is shown in Figure 3.

The relational mapping of the R-tree offers a new freedom, as it does not have to take into account physical block sizes. All data are stored in relations and accessed via standard SQL. At this abstraction level, pages are no longer physical entities but virtual objects. Thus, it is no longer necessary to look out for minimal or maximal fill factors of the pages. Even if there are sparsely filled pages, we are not wasting disk space. Furthermore, it is no longer mandatory to split a page only because the number of its entries exceeds a certain number. In fact, we are basically free to allow an individual fanout for each tree node.

Similar to the concept of supernodes for high-dimensional indexing [2], larger nodes could be easily allocated, e.g. if the contained geometries show a very high overlap or are almost equal. Splitting such pages would not improve the spatial clustering, it would rather result in high overlapping nodes. Thus, we suggest page splits triggered by measuring the clustering quality with a similarity measure similar to the one presented in [9].

To motivate our idea, we assume the most extreme case of overlap by inserting only one object repeatedly into the tree. We use stochastic criteria for the insert, update and delete operations which avoid splitting highly clustered data into different nodes by allowing nodes with a variable fan-out. Our basic idea consists of the following probability assumption. First, we assume that any given query q hits a page p with a probability P_{father}. Furthermore, we assume that this query q hits also all entries in the page p with a probability P_{all_child}. Then, we carry out a split iff P_{all_child}/P_{father} falls below a certain threshold $t_{similiarity}$.

In the case all entries are identical, P_{all_child}/P_{father} is always equal to 1 and never falls below a certain threshold $t_{similiarity}$. Therefore, a split is never performed. In this extreme case, we would obtain a tree with only one single big node comprising all the data of the tree. Any query would result in a range scan on this node.

On the other hand, filling the tree in a normal R*-tree [3] manner, i.e. splitting nodes due to the quantity of the entries, we would have to perform a lot of page splits during the index creation process. Furthermore, during the query process, a given query would have to go through the whole tree which is easily understood to be far more expensive in terms of the navigational cost.

In our experimental evaluation, we show that our new stochastic driven Relational R-tree outperforms the relational version of the R*-tree with respect to insert and select operations by far.

4 Stochastic Criteria for Organisation of the R-tree

In this section, we formally compute the probability $P(R_1, .. ,R_L | R_R)$ that any given hyper rectangle which intersects a directory node R_R, intersects also all children R_1 .. R_L within this directory node. Based on this probability, we can trigger the splitting algorithm of the stochastic driven RR-tree.

We start with normalizing the coordinates of our hyper rectangles of dimension d to assure that all data lies within the unit hyper rectangle $\prod_{i=1}^{d}[0, 1]$. For clarity, we first examine the one-dimensional case looking at intervals and their point transformation into the upper triangle $D := \{(x, y) \in [0, 1]^2 | x \le y\}$ of the unit hyper rectangle with dimension $2 \cdot d$ (i.e. $2 \cdot d = 2$ in this case). An interval $[x, y]$ therefore corresponds to the point (x, y) with $x \le y$ (cf. Figure 4a).

Let $I = [a, b]$ be an interval. All intervals that intersect I are visualized by the shaded area in Figure 4a. The area displays all intervals whose lower bounds are smaller

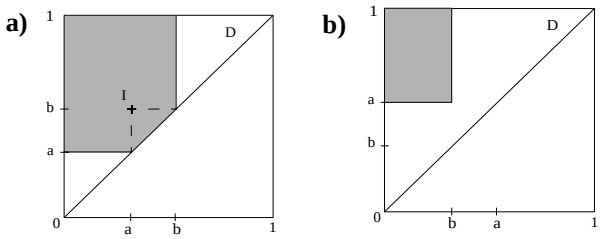

Figure 4: Point transformation of the interval I = [a,b] with **a)** a < b and **b)** a > b

than or equal to b and whose upper bounds are bigger than or equal to a. These intervals are exactly the ones that have a non-empty intersection with I.

The upper triangle D has the area $\Delta = \frac{1}{2}$, the shaded area spans $b \cdot (1-a) - \frac{1}{2} \cdot (b-a)^2$ if $a \le b$. A more general formula is defined in the following using the *Iverson Notation* [7]. A statement which can either be true or false is placed in square brackets, like this $[a \le b]$. If the statement is true, the whole expression is 1, otherwise it is 0. Let $a, b \in [0, 1]$ be arbitrary numbers. For the interval $I = [a, b]$ we use from now on the abbreviation $A(I) = A(a, b)$ representing all intervals intersecting *I*. Formally, we define $A(I)$ as follows:

$$A(a, b) := b \cdot (1-a) - [a \le b] \cdot \frac{1}{2} \cdot (b-a)^2 . \qquad (01)$$

Presuming equal distribution of the data, the probability that interval $I = [a, b]$ is struck by any query interval is:

$$P(I) = \frac{A(a, b)}{\Delta} . \qquad (02)$$

The probability $P(I_1, I_2)$ of a query hitting two intervals $I_1 = [a_1, b_1]$ and $I_2 = [a_2, b_2]$ with $a_i \le b_i$ and $a_i, b_i \in [0, 1]$ for all $i \in \{1, 2\}$, and $I_0 = I_1 \cap I_2$ is thus:

$$P(I_1, I_2) = P(I_0) = \frac{A(max\{a_1, a_2\}, min\{b_1, b_2\})}{\Delta} . \qquad (03)$$

The case where I_1 and I_2 intersect each other, i.e. $I_0 = I_1 \cap I_2 \neq \emptyset$, is illustrated in Figure 5a.

In Figure 5b the case is depicted that the two intervals do not intersect, i.e. $I_1 \cap I_2 = \emptyset$ (w.l.o.g. we assume $a_1 \le b_1 < a_2 \le b_2$). Based on formula (01) we can compute the shaded area as follows:

$$\begin{aligned} A(max\{a_1, a_2\}, min\{b_1, b_2\}) &= A(a_2, b_1) \\ &= b_1 \cdot (1-a_2) - [a_2 \le b_1] \cdot \frac{1}{2} \cdot (b_1 - a_2)^2 \\ &= b_1 \cdot (1-a_2) \end{aligned}$$

In general, the probability of a query intersecting the intervals $I_1, \ldots, I_n$ with $I_i = [a_i, b_i]$ and $i \in \{1, .., n\}$ is obtained by the following formula:

$$P(I_1, .., I_n) = \frac{A(max_{i=1..n}\{a_i\}, min_{i=1..n}\{b_i\})}{\Delta} . \qquad (04)$$

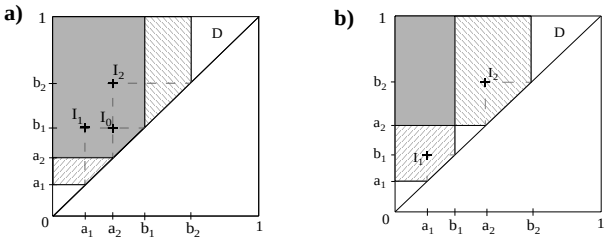

Figure 5: Point transformation of two intervals $I_1 = [a_1,b_1]$ and $I_2 = [a_2,b_2]$
a) with non-empty intersection **b)** with empty intersection

The conditional probability of interval I_1 being intersected by a query that already intersects I_2 is thus:

$$P(I_1 | I_2) = \frac{P(I_1, I_2)}{P(I_2)} = \frac{A(max\{a_1, a_2\}, min\{b_1, b_2\})}{A(I_2)} \qquad (05)$$

and analogously, let $L, R \in IN$:

$$P(I_1, .., I_L | I_R) = \frac{P(I_1, \ldots, I_L, I_R)}{P(I_R)} \qquad (06)$$

Assuming independence of the x and y sizes of the data, the formulas derived so far can be expanded to arbitrary dimension:

Let therefore $R_1 := \prod_{i=1..d} I_i$ and $R_2 := \prod_{i=1..d} J_i$ be two *d*-dimensional hyper rectangles. The probability that a query intersects R_1 is thus

$$P(R_1) := \prod_{i=1}^{d} P(I_i) . \qquad (07)$$

Analogously:

$$P(R_1, R_2) := \prod_{i=1}^{d} P(I_i, J_i) . \qquad (08)$$

For hyper rectangles with more than 2 entries, the conditional probabilities are derived in the same manner:

$$P(R_1, .., R_L | R_R) := \prod_{i=1}^{d} P(I_{1_i}, .., I_{L_i} | I_{R_i}) \qquad (09)$$

with $R_j := \prod_{i=1..d} I_i$, $j \in \{1, .., L, R\}$, and $L, R \in IN$.

5 Stochastic Driven Relational R-Tree

We now adopt the stochastic criteria, i.e. probabilities described in the foregoing section, to the Relational R-tree. In this section, we describe how and where the heuristics can be applied. We state the different update operations in detail and show their enhancement by presenting our experiments in the next section.

Note that the algorithms of our stochastic driven RR-tree are based upon the ones of the R*-tree [3] which we use without modification, apart from the changes described below. In the following, we do not quote the full algorithms and therefore refer the reader to the corresponding literature [3].

5.1 Insert

The stochastic criteria, i.e. probabilities, described above can be adopted by the RR-tree *insert*-algorithm in the following decision points:

- selection of insert position
- election of the point of time for a split
- partitioning of entries when splitting is performed

```
PROCEDURE Insert(R,Page)
   ActualSimilarity REAL;
   MaxSimilarity REAL;
   BestEntry PageEntry;
BEGIN
   IF Page IS DirectoryPage THEN
      MaxSimilarity := 0;
      FOR EACH E IN Page
         ActualSimilarity := P(R, E);
         IF ActualSimilarity > MaxSimilarity THEN
            BestEntry := E;
            MaxSimilarity := ActualSimilarity;
         END IF;
      END FOR;
      Insert(R, BestEntry);
   ELSE
      . . .
   END IF;
END;
```

Figure 6: Choosing entry for insert

Insert Position. In order to insert a rectangle R into a page we choose the most similar entry E : Select E so that $P(R, E)$ is maximal, i.e. take the subtree for which the probability of a query hitting R and E simultaneously becomes maximal (cf. Formula (08)). (In case this criterion is not unique, other criteria may be tested additionally). The algorithm is listed in Figure 6.

Due to the fact that all entries of a virtual page have to be tested, the selection of the "best" subtree results in a complexity linear to the number of entries of that virtual page. The cost of inserting a rectangle is therefore $O(L \cdot d \cdot logN)$, with L being the average number of entries per page, d the dimension, and N the total number of rectangles.

Split. As described above, our aim is to allow a variable fanout for each node to improve the structure of the tree in terms of clustering and similarity of the data. Therefore, we let the decision when a split should be performed also depend on probabilities. We split a node only if its entries are relatively dissimilar. However, in order to prevent having a high number of nodes that may be only sparsely filled, we consider a split only after the count of entries exceeds a specified common filling factor.

Let B be the minimal bounding rectangle of all entries $E_1, .. , E_L$ of a page. Assuming the page is already commonly filled, a split is performed iff the probability $P(E_1, .. , E_L | B)$ falls below a certain threshold (cf. Formula (09)). Figure 7 depicts the corresponding pseudocode.

```
PROCEDURE Split(Page)
   L          INTEGER;
   Threshold REAL;
BEGIN
   L := NumberOfEntries (Page);
   // Determine bounding boxes of page B and of all elements E_i
   Similarity := P(E_1,...,E_L | B);
   IF L <= CommonEntriesPerPage OR
      Similarity > Threshold THEN      // Don't split
      RETURN;
   ELSE
      // Split
      . . .
   END IF;
END;
```

Figure 7: Decision of point of time for a split

The calculation of $P(E_1, .. , E_L | B)$ results in a complexity of $O(d \cdot L)$ with L and d as defined above.

Partitioning. To explain the use of the heuristics on the partitioning of the entries when a node is split, we first describe the split algorithm: Assimilating the R*-tree split operation, we first reinsert the outermost rectangles. This improves the distribution of the rectangles and can in some cases prevent a split. Thus, the structure of the tree is widely independent from the insertion order. If a split cannot be avoided, the stochastic measure is used to assign the partitioning. Thereby, several possible partitions are examined and the best one is chosen.

The algorithm works as follows (cf. Figure 8):

- If *Reinsert* has not been started yet: Delete the *m* rectangles which are furthermost from the center of the page and insert them again (with *m* being the reinsertion rate).
- If *Reinsert* has already been carried out: Find a split axis (perpendicular to which the split is to be performed). For each dimension find the interval pair I_i, I_j with maximal normalized distance d_{max}. Choose the dimension i with maximal d_{max} as the split axis and I_i, I_j as the initial pair. This part of the splitting process is similar to the R*-tree (cf. Figure 8a).
- In the next step, we group the hyper rectangles similar to the grouping performed by the R*-tree. Nevertheless, there is a decisive difference: We do not use a geometric distance measure but our stochastic heuristics for splitting the node into two partitionings *A* and *B*. In a first step, we determine for each rectangle of the node its projection I to the split axis. Then, we calculate $a = P(I|I_i)$ and $b = P(I|I_j)$, according to Formula (05). For a certain bandwidth $[-\varepsilon, \varepsilon]$ and in-

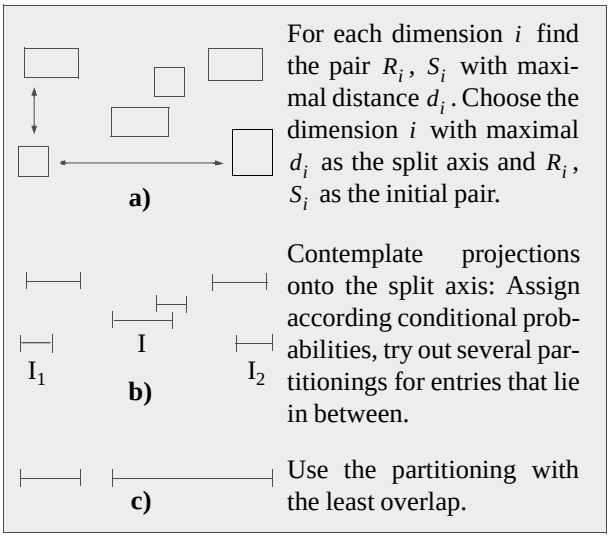

Figure 8: Illustration of the partitioning procedure

crement δ, we examine $k \in IN$ potential partitionings. For all $\tau := -\varepsilon + k \cdot \delta$ with $\tau \in [-\varepsilon, \varepsilon]$ we create two partitions A and B as follows: We insert a rectangle into partitioning A iff $a - b > \tau$, otherwise into B. Then we determine the overlap in the split dimension that is generated during this partitioning and, finally, use the partitioning that produces the least overlap out of the k computed ones.

This procedure does not guarantee minimal node filling factors. Nor is this necessary due to the fact that we are dealing with virtual pages. However, a future split on this page is only performed after reaching a common filling factor, as explained above.

We determine the initial pair and the split axis as supplied by the traditional R*-tree algorithm. We use the projections of the entries to the split axis to elect the partitioning. Using the complete rectangles, we may obtain high overlap. Figure 9 shows an example: The initial situation is depicted in Figure 9a. The page to be split holds six rectangles. R_1 and R_6 were selected as the initial pair. The calculation of the partitioning without projection is shown in Figure 9b. The probability of the complete rectangles assigns the split as follows: $\{R_1, R_2, R_4\}$ and $\{R_3, R_5, R_6\}$. A high overlap is obtained. In Figure 9c the partitioning is depicted which first projects each rectangle onto the split axis and then uses the discussed stochastic heuristics. This results in the non-overlapping partitioning $\{R_1, R_2, R_3\}$ and $\{R_4, R_5, R_6\}$.

Let N be the total number of rectangles in the tree, L the average number of entries per virtual page, and let d be the dimension. Furthermore, let k denote the number of computed partitionings per node. Then, our stochastic

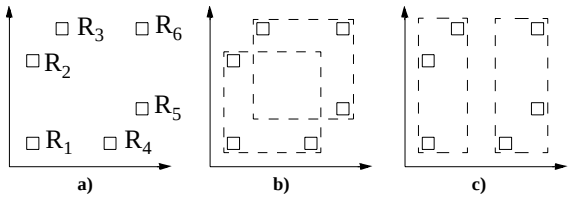

Figure 9: Use of projection for split
a) node to split **b)** stochastic partitioning without projection **c)** stochastic partitioning with projection

driven split algorithm has an overall complexity of $O((d \cdot L + k \cdot L) \cdot \log N)$.

5.2 Delete

The stochastic criteria does not interfere with the delete procedure. Let B be the minimal bounding rectangle of all entries $E_1, .., E_L$ of a page, $P(E_1, .., E_L | B)$ the probability of any query hitting all entries of the page when hitting its minimal bounding rectangle. A split is invoked iff the probability falls below a certain threshold. Without loss of generality, let us assume the entry to be deleted is E_L. Thus, the probability we have to consider is $P(E_1, .., E_{L-1} | B')$ where B' is the updated MBR. Of course, this probability is higher than $P(E_1, .., E_L | B)$. If an element is deleted within a specific node, the probability that all elements of that node are hit by a given query increases. Therefore, a delete operation can not instantiate a split.

However, deleting entries motivates the idea of merging spatially neighbored nodes to achieve an optimal structure of the tree. Within the scope of this work we did not deal with this possibility and leave it for future work.

5.3 Update

An update of an entry, is dealt with by deleting the entry and inserting it again as described in the foregoing sections.

6 Experimental Evaluation

We evaluated the stochastic driven relational R-tree by means of various experiments which we describe below. For comparison we used a relational implementation of the R*-tree [3]. Our trees were mainly filled with 2-dimensional data, the parameters were optimized for the 2-dimensional case, e.g. the probability threshold $t_{similarity}$ which is responsible for the decision whether or not a split is carried out.

6.1 Update Performance

In our first experiment, we measured the time required for building up a tree, i.e. inserting rectangles. The data con-

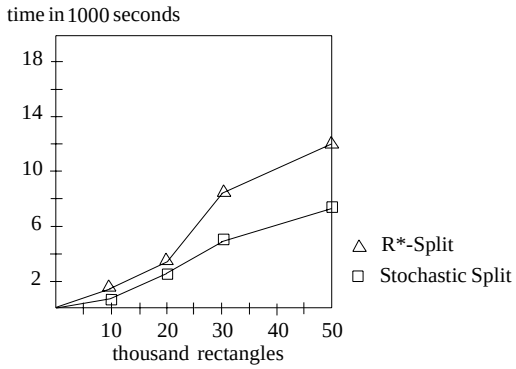

Figure 10: Insertion times

sisted of rectangles with a side length between 0 and 100 and center between 0 and 100,000. Side length as well as center are equally distributed.

In Figure 10, the x-axis shows the number of inserted rectangles and the y-axis corresponds to the time axis. Our stochastic insert and split routines were optimized to have the minimal possible number of SQL-statements to perform. The positive effect is noticeable especially with an increasing number of inserted rectangles.

The inserts are subject to big fluctuations. Tests have shown that the times depend on the fact if the database system has to expand during insertions or not. The presented numbers state the average values.

6.2 Query performance

High Selectivity. In the next experiment we measured the performance on highly selective window queries. As query objects we used rectangles out of the data set.

Figure 11 displays real-times for an amount of data of 10,000 up to 200,000 rectangles. It is clearly seen that the stochastic algorithms outperform the common R*-tree algorithms.

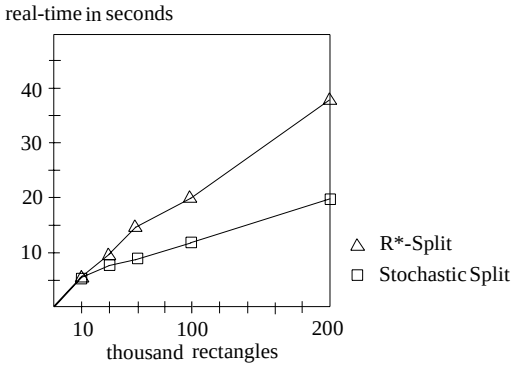

Figure 11: Window queries with very high selectivity

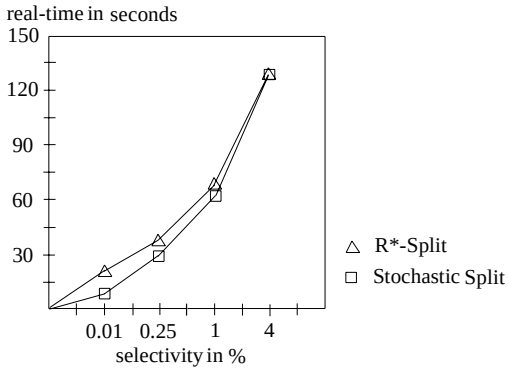

Figure 12: Window queries with decreasing selectivity

Varying Selectivity. In the following experiment we set the number of rectangles to a fix 50,000 and varied only the selectivity. In Figure 12, it can be detected that with a selectivity of up to 4% the stochastic algorithms outperform the R*-split. With decreasing selectivity, the differences vanish.

Varying Dimensions. Figure 13 shows real-times of window queries in R-trees of different dimensions. The values were normalized such that the values of the stochastic algorithms add up to 1 in each case. In this experiment, we performed queries with a selectivity of 0.0001% in R-trees of 50,000 objects.

Figure 13 shows that at dimension smaller or equal to 3, the stochastic R-tree is faster in comparison with the R*-tree. It has to be mentioned, however, that we used parameters optimized for the two-dimensional case. Within the scope of this work, we confined ourselves to two and three dimensional data as they form the foundation of modern GIS applications. If efficient query processing for

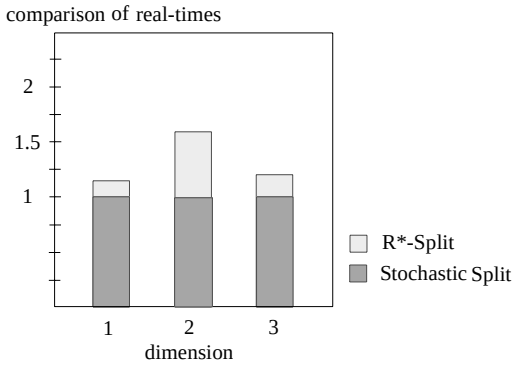

Figure 13: Selective window queries with different dimensions (values normalized to 1)

high dimensional data is required, we suggest to adopt the presented stochastic heuristics to the X-tree [1, 2].

7 Conclusion and Future Work

In this paper, we introduced stochastic heuristics and adopted them to the Relational R-tree. We thereby took advantage of the relational mapping that frees us from physical factors which prevail using the non-relational version of the R-tree and thus offer a wide range of potential improvements. Here, we elaborated a concept to improve the index-structure especially in terms of highly clustered data. We described algorithms that consider the quality of the entries of a node rather than just the quantity. Using stochastic heuristics for evaluation, we allow an individual fanout for each node. Thus, we are able to postpone a split while it is not beneficial for the performance of the tree. For the split we also optimized the partitioning algorithm. Our experiments clearly demonstrate the advantages of the technique.

Aside from the introduced concept, we developed and evaluated further possible extensions to the Relational R-tree:

Page Clustering. In order to achieve a good clustering among the entries of each tree node, a built-in primary index can be defined on the *page_id* column. For bulkloads of Relational R-trees, the clustering can be further improved by carefully choosing the page identifiers: by assigning linearly ordered *page_id*s corresponding to a breadth-first traversal of the tree, a sibling clustering of nodes [8] can be very easily achieved.

Positive Pruning. By ordering the *page_ids* according to a depth-first tree traversal, a hierarchical clustering of the R-tree nodes is materialized in the primary index. In consequence, the page identifiers of any subtree form a consecutive range. Similarly, if the leaf pages are hierarchically clustered in a separate B^+-tree, a single range query on the *page_id* column yields a blocked output of all data objects stored in any arbitrary subtree of the R-directory. Thus, the recursive tree traversal below a node completely covered by the query region can be replaced by an efficient range scan on the leaf table. Consequently, the tree traversal is not only pruned for all-negative nodes (if no intersection of the node region with the query region is detected), but also for all-positives (the node region is completely covered by the query region). Moreover, statistic based heuristics to prune already largely covered nodes can also be very beneficial.

References

[1] S. Berchtold, C. Böhm, H.-P Kriegel., U. Michel: *Implementation of Multidimensional Index Structures for Knowledge Discovery in Relational Databases*, Proc. Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK'99), Florence, Italy 1999, in: Lecture Notes in Computer Science, Vol. 1676, Springer, 1999, pp. 261-270.

[2] S. Berchtold, D. A. Keim, H.-P. Kriegel, *The X-tree: An Index Structure for High-Dimensional Data*, Proc. 22nd Int. Conf. on Very Large Databases (VLDB): 28-39, 1996.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, *The R*-tree: an efficient and robust access method for points and rectangles*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1990.

[4] Council of the European Communities. Council Directive (90/313/EEC) of 7 June 1990 on the freedom of access to information on the environment. Official Journal of the European Communities, L158:56-58, 1990.

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

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

[7] R. L. Graham, D. E. Knuth, O. Patashnik, *Concrete Mathematics*, second edition, Addison-Wesley, 1994.

[8] K. Kim, S. K. Cha, *Sibling Clustering of Tree-based Spatial Indexes for Efficient Spatial Query Processing*, Proc. ACM CIKM Int. Conf. on Information and Knowledge Management: 398-405, 1998.

[9] I. Kamel, C. Faloutsos, *Parallel R-trees*, Proc. ACM SIGMOD Int. Conf. on Management of Data: 195-204, 1992.

[10] H.-P. Kriegel, M. Pfeifle, M. Pötke, T. Seidl: *The Paradigm of Relational Indexing*: *a Survey.* BTW 2003: 285-304.

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

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

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