A Revised R*-tree in Comparison with Related Index Structures

Norbert Beckmann
Department of Computer Science
University of Bremen
D-28359 Bremen
Germany
nb@informatik.uni-bremen.de

Bernhard Seeger
Department of Computer Science
University of Marburg
D-35032 Marburg
Germany
seeger@informatik.uni-marburg.de

ABSTRACT

In this paper we present an improved redesign of the R*-tree that is entirely suitable for running within a DBMS. Most importantly, an insertion is guaranteed to be restricted to a single path because re-insertion could be abandoned. We re-engineered both, subtree choice and split algorithm, to be more robust against specific data distributions and insertion orders, as well as peculiarities often found in real multidimensional data sets. This comes along with a substantial reduction in CPU-time.

Our experimental setup covers a wide range of different artificial and real data sets. The experimental comparison shows that the search performance of our revised R*-tree is superior to that of its three most important competitors. In comparison to its predecessor, the original R*-tree, the creation of a tree is substantially faster, while the I/O cost required for processing queries is improved by more than 30% on average for two- and three-dimensional data. For higher dimensional data, particularly for real data sets, much larger improvements are achieved.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS; H.2.2 [**Physical Design**]: Access methods; H.3.1 [**Content Analysis and Indexing**]: Indexing methods

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Index structures, Revised R*-tree, RR*-tree, R-tree, Hilbert-R-tree, Multi-dimensional data, Split, ChooseSubtree, Performance comparison

SIGMOD'09, June 29–July 2, 2009, Providence, RI, USA.

1. INTRODUCTION

R-trees [15] are among the most popular multidimensional access methods suitable for indexing two-dimensional spatial data and medium-dimensional point data. Due to their conceptual simplicity (and excellent performance), R-trees have been successfully implemented within commercial systems like IBM Informix [18], Oracle [22, 21], Sun MySQL [25], and open source database systems like PostgreSQL [27], and various research prototypes including Shore [9] and GIST [16]. Moreover, R-tree implementations are available in geographic information systems like MapInfo [32].

During the last two decades different proposals for R-trees have emerged, mainly differing in the split algorithm and the traversal algorithm for the insertion of new objects (commonly called *ChooseSubtree*). In different experimental studies, see for example [17], the R*-tree [6] has been shown to be the most efficient structure regarding query performance. Therefore, the R*-tree has served as a guideline for many R-tree implementations and has influenced the design of other access methods, especially those suitable for indexing high-dimensional point data [2].

However, applying the algorithms of the R*-tree completely was interfered by some deficiencies. First, re-insertion of entries, which is considered to be one of the essential contributions to improve query performance, is a cumbersome concept in DBMS. Its disadvantage is that it prevents the usage of efficient algorithms for concurrency control [19, 11]. Secondly, overlap optimization during ChooseSubtree, probably the most important optimization to enhance query performance, is only performed rudimentarily. It is restricted to the lowest directory level and limited by a constant parameter to keep CPU cost acceptable. This parameter however is difficult to set in view of the changing demands by different data sets and varying dimensionalities. Another deficiency of the R*-tree is that the scoring function of the split does not take into account the balance of the split pages. Though the imbalance of a split is limited by a parameter m, the degree of imbalance of insertions does not influence the split of a page. This deficiency is not unique to the R*-tree, but can be observed for all members of the R-tree family. Another problem refers to the fact that R-trees are not really well designed for high-dimensional data. As degenerated bounding boxes with zero volume are not uncommon for high dimensional data, the commonly used volume-based strategies for splitting are becoming ineffective.

In this paper, we present an update of the algorithms of the R*-tree. While basic design principles did not change, e.g. the nature of the split and important optimization criteria of ChooseSubtree, the idea of re-insertion was abandoned as it prevents the applicability in a DBMS. We redesigned the algorithm ChooseSubtree so that irrele-

vant entries, which are not able to host the new object, are pruned. This reduces CPU time considerably, so that an effective overlap optimization can be applied now on all directory levels. Our split algorithm is not limited to optimizing well-known criteria (overlap, volume, perimeter), but also takes into account the balance of the split. In general we prefer splits where half of the data is moved into a new page. However, balanced splits are not our first choice when the data is inserted in non-random order. By keeping track of the modification of a node's minimum bounding box (MBB), we are able to anticipate where data is likely to be inserted in the near future. This leads to a new highly adaptive split algorithm. Moreover, both, ChooseSubtree and split, identify when volume-based optimization is not effective anymore and switch in such situations to perimeter-based strategies.

In general, the Revised R*-tree (RR*-tree) should meet the following performance requirement. The query cost should be noticeable better than that of the original R*-tree for a large range of data files and arbitrary data dimensionality. Cost generally refers to the number of I/Os, but the RR*-tree should also provide acceptable CPU-performance when data is organized in main memory.

The performance of the RR*-tree is shown in different sets of experiments with data of different dimensionality. We considered artificial data sets as well as data sets obtained from real application scenarios. Our experimental comparison includes Guttman's quadratic R-tree [15], the original R*-tree [6], the Hilbert-R-tree [19], and the RR*-tree. To the best of the authors' knowledge, we are not aware of such a comprehensive performance comparison for R-trees. Therefore, a major contribution of the paper is the definition of our experimental setup that can serve as a test bed in future.

The paper is structured in the following way. Section 2 introduces our notation and gives a brief review of previous work. In Section 3 we present the details of our new algorithm for choosing a subtree. In Section 4 our new split algorithm is presented. Section 5 consists of a precise description of our experiments as well as a discussion of our results.

2. PRELIMINARIES

An R-tree [15] is a height-balanced tree, designed for organizing a set of *dim*-dimensional rectilinear rectangles in a dynamic setting. The primary goal of the R-tree is to assign rectangles to leaves in a way that spatial proximity is preserved within a leaf. Internal nodes contain entries which consist of a reference to a subtree *T* and the MBB of the rectangles in *T*. The entries of leaves consist of a reference to an object and the MBB of that object. A node of an R-tree corresponds to a page of fixed size. In the rest of the paper, *M* denotes the capacity of a node and *m* is the minimum occupation of a node, $m \leq \lceil M/2 \rceil$.

An insertion of a new rectangle into an R-tree starts at its root. A routine *ChooseSubtree* is called to determine a child node. ChooseSubtree is iteratively called until a leaf is found. The rectangle is inserted into the leaf and modifications of the leaf's MBB are propagated upwards along the insertion path. If the leaf already contains *M* objects, a routine termed *Split* is called. Split distributes the objects into two groups; the one remains in the original leaf, whereas the other is stored in a new leaf. Thereafter the original index entry is updated, and a new entry referring to the new leaf is inserted into the parent node, which may have to be split analogously.

2.1 Related Work

There are different kinds of R-trees which mainly differ according to their splitting strategy, see [24] for a comprehensive survey on R-trees. Originally Guttman proposed two greedy strategies [15], the one with linear run time and the other with quadratic. Both rely on first picking two seeds from the entries in the node. The remaining entries are then distributed among the seeds. Experiments have revealed that the algorithm with quadratic run time results in R-trees with better search performance. Many DBMSs like PostgreSQL make use of the quadratic greedy strategy. The R*-tree [6] is based on a quite different approach, where a split of a node is optimized with respect to different criteria of the MBBs of the resulting nodes. This approach has been justified later by a static cost model, introduced in [28] and [13], where the volume of the boxes and particularly their perimeter have been shown to be the dominant parameters for the I/O cost of queries. Another beneficial idea of the R*-tree is that splits are deferred by re-inserting the objects farthest away from the center of the MBB. This improves query performance and increases storage utilization.

The Hilbert-R-tree [19] is another kind of R-tree which actually behaves like a B+-tree when objects are inserted. The basic idea is to insert rectangles into the tree, using the one-dimensional Hilbert-value of the center of the rectangles. In addition to this routing information required for insertions, the MBB of a subtree is stored in the corresponding index entry. These rectangles are used for query processing only. This actually leads to a structure with excellent insertion performance, but our experimental results generally show, that the search quality of the tree suffers from the one-dimensional insertion strategy. Additionally, the structure needs a pre-defined data space. In fact, a main advantage of the original R-tree and most of its variants is that there is no need for a pre-defined data space. Unlike structures relying on transformation and one dimensional encoding, like the Hilbert-R-tree, R-trees do not require global reorganization when new data has to be inserted outside planned boundaries.

The problem of splitting a group of rectangles into two has been addressed in [4], where an algorithm with run time $O(M^{dim})$ is presented for computing an optimal split. Quite a similar approach has been presented in [14]. Experiments however showed, that an optimal splitting routine only marginally improves the overall performance of the R-tree in comparison to other strategies (since subsequent insertions will compensate the advantage). A substantial performance improvement can be achieved by global optimization techniques [29, 14]. Unfortunately, these techniques also require a substantial amount of reorganization that is only acceptable in off-line mode. Since efficient concurrency control is a major concern for index structures too, it is questionable whether these techniques will be integrated into the implementations of R-trees that are available in commercial DBMS.

Bulk-loading of R-trees has also received research attention. Different techniques [7, 1] have been proposed that substantially reduce the cost of the naive sequential insertion strategy where one tuple is inserted at a time. The bulk-loading method of [1] gener-

ates worst-case optimal R-trees, while their loading time is asymptotically equal to the one of external sorting. Bulk-loading is however beyond the scope of this paper where a dynamic setting is considered, requiring rectangles to be inserted one at a time.

While most of the different methods consider the problem of organizing two- or three-dimensional rectangles, R-trees also have been applied to the organization of high-dimensional point data [26]. Experiments [5] have however revealed that R-trees are not appropriate for this purpose, and as a consequence, a large number of different methods have been proposed to overcome their deficiencies, see [2] for an excellent survey. Among those methods is the X-tree [5], a generalization of the R*-tree, where bad splits are avoided by dynamically increasing page sizes. The management of variable page sizes is however not supported in database systems, as free-space management and buffering is becoming more complex. Other proposals like the SR-tree [23] additionally maintain minimum bounding spheres in an index entry. The A-tree [30] suggests a special compression technique, resulting in a higher capacity of the nodes, whereas the hybrid-tree [10] is a clever combination of kd-tree [3] and R-tree.

2.2 Notation

Let $R = [min_1, max_1] \times \ldots \times [min_{dim}, max_{dim}]$ be a *dim*-dimensional box. Let $\chi_d(R)$ be the center of $[min_d, max_d]$ and $\lambda_d(R)$ be its length, $1 \le d \le dim$. Then the perimeter and volume of R is given by

$$perim(R) = \sum_{d=1}^{dim} \lambda_d(R) \quad \text{and} \quad vol(R) = \prod_{d=1}^{dim} \lambda_d(R)$$, respectively. In the following, we use $argmin\{I_1, \ldots, I_k\}$ to denote the smallest index that achieves a global minimum in $\{I_1, \ldots, I_k\}$ according to an ordering relation defined on $I_1, \ldots, I_k$.

3. ChooseSubtree

Let Ω be an object, representing a point or rectangle that shall be inserted into the R*-tree. Given a node N, initially the root, the task of the algorithm ChooseSubtree is to determine a child node of N, to which Ω will be directed. The algorithm is important for the quality of the tree, particularly if one of the MBBs in N has to be enlarged to cover Ω.

Before presenting our new ChooseSubtree algorithm, we first introduce important definitions and review the ChooseSubtree algorithm of the original R*-tree.

Definition 1. Let $\{E_1, \ldots, E_k\}$ be the entries of a node N, and let R_i be the MBB of E_i, $1 \le i \le k$. Let $f \in \{vol, perim\}$, where *vol* and *perim* denotes the volume and the perimeter of a rectangle, respectively. The assignment of an object Ω to E_t increases

- $f(E_t)$ by $\Delta f_t = f(MBB(R_t \cup \Omega)) - f(R_t)$,
- the common overlap of E_t with E_j, $1 \le j \le k$ by
$$\Delta ovlp^f_{t,j} = f(MBB(R_t \cup \Omega) \cap R_j) - f(R_t \cap R_j),$$
- the common overlap of E_t with entries $\{E_p, \ldots, E_q\}$, $1 \le p \le q \le k$, by
$$\Delta ovlp^f_{t,[p,q]} = \sum_{j=p, j \ne t}^{q} \Delta ovlp^f_{t,j}.$$

Referring to the context of Definition 1, ChooseSubtree of the original R*-tree, denoted CSOrig, can be described as follows:

```
Index CSOrig(Node N, Object Ω)
// returns the index of the entry, to which Ω is assigned
// k = number of entries in N, r = cost limitation parameter
COV ← {i|MBB(R_i ∩ Ω) = Ω, 1 ≤ i ≤ k};
if ( COV ≠ ∅ )
    return argmin{vol(R_i)|i ∈ COV} ;

// assertion: COV = ∅, an MBB has to be enlarged
if (the entries of N do not refer to leaves)
    return argmin_{1≤i≤k}{Δvol_i} ;
else
    let E_1, …, E_k be the sequence of entries of N, sorted in
    ascending order of their Δvol;
    if (there is an index j with Δovlp^vol_{j,[1,k]} = 0 )
        return the smallest j where Δovlp^vol_{j,[1,k]} = 0 ;
    else // consider only the first r entries
        return argmin_{i=1,…,r}{Δovlp^vol_{i,[1,k]}} ;
end CSOrig;
```

Experimental results [17] have shown that CSOrig improves the search performance compared to other strategies, though overlap optimization, a very expensive strategy, is only performed in the lowest directory level. The parameter *r*, see third line from bottom, has originally been introduced to reduce the CPU cost to $O(M \log M + Mr), r \le M$. The setting of parameter *r* is quite important. If *r* is too small, the probability of missing the entry with the smallest overlap enlargement is high. For *r* being too large, however, the CPU cost becomes a crucial cost factor. Though the original setting of *r* to 32 was appropriate in the specific experimental setup, it is not useful as a general rule. The local density of the data and the dimensionality can have a major impact on a suitable setting of *r*.

Another serious problem of the algorithm is its pure volume-based optimization strategy that can become ineffective for high-dimensional data. In fact, bounding boxes with zero volume are not uncommon as data can be in lower-dimensional subspaces. Note that this deficiency is not limited to the R*-tree only, but can be observed for any member of the R-tree family.

The revised ChooseSubtree algorithm, termed CSRevised, addresses the above mentioned problems. First, a simple adaptive strategy is proposed for computing a candidate set of entries that are checked for overlap enlargement. Similar to the original algorithm, the worst-case performance is still $O(M^2)$, but the average-case performance is substantially better in practice. Due to these cost reductions, overlap optimization is performed now on all

directory levels. Secondly, the algorithm identifies situations when volume-based criteria become ineffective. It then switches from the normally preferred volume- to a perimeter-based strategy.

```
Index CSRevised(Node N, Object Ω)
// returns the index of the entry to which Ω is assigned
// k = number of entries in N
COV ← {i|MBB(R_i ∩ Ω) = Ω, 1 ≤ i ≤ k} ;
if ( COV ≠ ∅ )
    if (there is i ∈ COV with vol(R_i) = 0 )
        return argmin{perim(R_i)|i ∈ COV} ;
    else
        return argmin{vol(R_i)|i ∈ COV} ;
// assertion: COV = ∅, an MBB has to be enlarged
let E_1, …, E_k be the sequence of entries of N, sorted in ascending
order of their Δperim;
if (Δovlp^perim_1,[1,k] = 0 )
    return 1;
else
    p ← arg max_{i=2,…,k} { Δovlp^perim_1,i | Δovlp^perim_1,i ≠ 0 } ;
    // Consider only the first p entries in the remaining steps
    CAND ← ∅ ; success ← false ;
    if (there is an index i with vol(MBB(R_i ∪ Ω)) = 0 )
        CheckComp(1, perim);
    else
        CheckComp(1, vol);
        ---------------------------------------------------
        Procedure CheckComp(Index t, AggregateFunction f)
        CAND ← CAND ∪ {t} ;
        Δovlp[t] ← 0 ; // start computation of Δovlp^f_t,[1,p]
        for (j = 1, …, p, j ≠ t )
            Δovlp[t] ← Δovlp[t] + Δovlp^f_t,j ;
            if ( Δovlp^f_t,j ≠ 0 and j ∉ CAND )
                CheckComp(j,f);
                if (success) break;
            end
        end
        if ( Δovlp[t] = 0 ) // i.e. Δovlp^f_t,[1,p] = 0
            c ← t ; success ← true ;
        end CheckComp.
        ---------------------------------------------------
    if (success)
        return c;
    else
        // Δovlp[i] , i ∈ CAND already computed by CheckComp
        return argmin{Δovlp[i]|i ∈ CAND} ;
end CSRevised;
```

CSRevised starts with computing *COV*, the set of entries that entirely cover the new object Ω. If *COV* is not empty, the entry with minimum volume (perimeter) is selected from *COV*. Otherwise, the perimeter growth $\Delta perim$ is computed for each entry, pretending that it would receive the new object Ω. Then the entries are sorted according to their $\Delta perim$. Ω is assigned to E_1 if none of the other entries suffers from the assignment (by increased perimetric overlap with E_1). Otherwise the algorithm tries to achieve an overlap optimized choice, where the computation cost is considerably reduced by two successive filter steps. The first filter limits the focus to the first p entries of the sorted sequence, where entry E_p is the last entry whose perimetric overlap would increase from the assignment of Ω to E_1. Then CheckComp is called with the volume (perimeter) based strategy. In a depth-first manner, CheckComp tries to determine an index t, $1 \le t \le p$, for which the assignment of Ω to E_t would not increase the overlap with the other p-1 entries. During the depth-first traversal, all examined indices are registered in the set *CAND*, which serves as the second filter when CheckComp does not succeed in finding an assignment without additional overlap. In this case, the smallest index of *CAND* with minimum overlap increase is determined. Note that the size of *CAND*, denoted by q, is at most p, but generally smaller.

The example depicted in Figure 1 illustrates the processing steps of CSRevised. The assignment of Ω to E_1 causes an increase of perimetric overlap with E_3 and E_4. It follows $p = 4$ because E_4 is the last index with overlap increase. The call of CheckComp with the volume-based strategy examines indices 1, 3 and 4, but does not succeed in finding an assignment without additional overlap. In the final step, index 3 is returned as it causes the minimum overlap increase among indices 1, 3 and 4. Note that index 2 is not considered anymore as it is not in *CAND*.

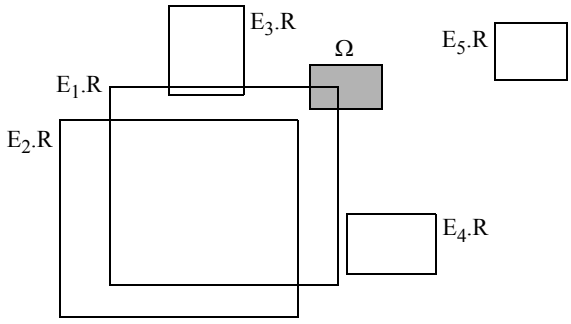

Figure 1. Assignment of Ω to one of five entries

We will conclude this section with a simple cost analysis. The CPU cost of our new algorithm is $O(M \log M + qp)$, where q denotes the size of *CAND*. The first term represents the cost for sorting, while the second represents the cost of CheckComp. Since $q \le p \le M$, this results in $O(M^2)$ worst-case performance. In our experimental study however, we observed that on average p and q are much smaller than M. The worst case CPU cost of CSOrig is $O(M \log M + Mr)$, $r \le M$ (proposal: $r = 32$), but overlap optimization is only performed in the lowest directory level, i.e. once for each insertion. Nevertheless, though optimizing more effectively, CSRevised yields a substantial reduction of the average CPU insertion cost.

4. Split

Our split algorithm follows the same guidelines as the one of the original R*-tree, where a split is performed by minimizing a goal function. The design of our goal function is different however, as it does not solely depend on the criteria overlap, volume and perime-

ter, but also takes into account the degree of the balance of a split. When data is randomly inserted, we generally prefer splits where the entries are evenly distributed among the two target nodes. For non-random insertion orders however, our split algorithm is sufficiently adaptive to prefer an unbalanced split by keeping track of the unbalanced growth of the MBB of the split node.

Our new adaptive strategy is actually a substitute of re-insertion, a technique that has been considered to be among the most beneficial contributions of the R*-tree for improving its search performance. In particular, re-insertion provides an effective solution to prevent search performance degradation for non-random insertion orders.

In the following we describe the details of our new algorithm. We first begin with a few important definitions. Then we give a review of the split algorithm of the original R*-tree. Finally, we provide a detailed description of the new goal function and its application to splitting a node.

4.1 Review of the Split of the R*-tree

The strategy of the split of a node into two is based on sorting the rectangles with respect to their projection to each dimension, and to consider for each sorted sequence the split at the i-th element ($m \le i \le M+1-m$), where the first i entries are assigned to the first node, and the remaining $M+1-i$ entries are assigned to the second node. Two different kinds of orders are considered for each dimension. The one refers to the minimum boundary of the rectangles and the other to their maximum boundary. Overall, there are $2dim$ different orders. For the sake of clarity, we restrict the following description of the algorithm to one order for each dimension.

Definition 2: Let $SC_{i,d} = (F_{i,d}, S_{i,d})$ be the i-th split candidate, $i = m, \ldots, M+1-m$, with respect to the ordering of the d-th dimension. $F_{i,d}$ represents the first i entries of the node, whereas the remaining $M+1-i$ entries constitute $S_{i,d}$.

- The perimeter and volume of a candidate $SC_{i,d}$ is given by $f(SC_{i,d}) = f(MBB(F_{i,d})) + f(MBB(S_{i,d}))$ where $f = perim$ and $f = vol$, respectively.
- The overlap of a candidate w.r.t. $f \in \{perim, vol\}$ is given by $ovlp^f(SC_{i,d}) = f(MBB(F_{i,d}) \cap MBB(S_{i,d}))$
- If $MBB(F_{i,d}) \cap MBB(S_{i,d}) = \emptyset$, $SC_{i,d}$ is termed an overlap-free (o.f.) split candidate of the d-th dimension.

The split of a node corresponds to computing the optimum split candidate $SC_{u,a}$ where u is the *split index* on the *split axis* a. Except for the difference mentioned in the footnote, the original R*-tree algorithm proceeds the following two steps:

1. // computation of the split axis a

$$a \leftarrow argmin_{1 \le d \le dim}\left\{\sum_{i=m}^{M+1-m} perim(SC_{i,d})\right\};$$

2. if (there are overlap-free split candidates on split axis a)
   // thereof, choose the candidate with minimum perimeter
   $u \leftarrow argmin_{m \le i \le M+1-m}\{perim(SC_{i,a}) \mid SC_{i,a}$ o.f.$\}$ [1];
   else
   //choose the candidate with minimum overlap

$$u \leftarrow argmin_{m \le i \le M+1-m}\left\{ovlp^{vol}(SC_{i,a})\right\};$$

The revised split algorithm mainly differs from this one according to the second step. Its basic enhancement is addressed in the following sections. Moreover, the revised algorithm distinguishes between splitting internal nodes and leaves, concerning the choice of the split axis. For splitting a leaf, the split axis is still determined as described in step 1. In case of internal nodes however, the first step is left out and all dimensions are considered in the computation of the second step. Hence, we choose among the $dim*(M+2-2m)$ split candidates the one that minimizes the corresponding goal function of the second step. This split candidate implicitly defines the split axis. Finally, the revised algorithm will switch from a volume-based optimization strategy (last line of step 2) to a perimeter-based one, if

$$vol(MBB(F_{m,a})) = 0 \quad \text{or} \quad vol(MBB(S_{M+1-m,a})) = 0$$

is satisfied. From the negation of this condition, it follows that the volume of every split candidate is greater 0.

4.2 Design of the Goal Function for the Split

In this section, we introduce a new goal function $w: [m, \ldots, M+1-m] \rightarrow \Re$ that returns a real value for each split candidate $SC_{i,a}$. The minimum of the goal function determines the split. The basic idea of the design of the function w is to minimize a composition of the original goal function wg of the R*-tree (criteria volume, perimeter and overlap) and a weighting function wf that is introduced in the next subsection. The weighting function wf serves for adapting the split to the local imbalance of insertions, a criterion that has not been considered in previous splitting strategies. In order to achieve a composed function, we propose to design w as a product of wg and wf. The details are presented in subsection 4.2.4.

4.2.1 Design of the Weighting Function wf

The weighting function wf should be designed to reflect the change of the MBB of the overfilled node N, from the time when N was created to the time when the split occurs. For sake of simplicity, we first assume the domain of wf being the continuous interval [-1,1]. In order to be aware of a change, the center χ of the original MBB $OBox$ of N is stored in N at the time when N is created. When N is split later, χ is used to compute the parameter $asym$ for the split axis a: $asym = \frac{2(\chi_a(MBB(N)) - \chi_a(OBox))}{\lambda_a(MBB(N))}$. It follows that $asym$ is in [-1,1], when we assume insertions only[2].

If $asym = 0$, we should split N in the middle with highest probability, whereas an unbalanced split should occur with lower probability. In fact we consider $\mu = (1 - 2m/(M+1)) \cdot asym$ as the point in the domain [-1,1] that should receive the highest probabil-

[1] Note that the original R*-tree used the criterion vol instead.

[2] After deletions, χ is set to the center of the re-computed MBB of N.

ity. Our weighting function $wf: [-1, 1] \rightarrow [0, 1]$ should therefore be designed according to the following criteria:

1. $wf(\mu) = 1$
2. *wf* is monotonically increasing and decreasing in the range $[-1,\mu]$ and $[\mu,1]$, respectively
3. wf(-1) = 0 if $asym \geq 0$ and wf(1) = 0 if $asym \leq 0$

We obtained the best results with a Gaussian function $\exp\left(-\left(\frac{x-\mu}{\sigma}\right)^2\right)$, where $\sigma = s(1+|\mu|)$ and s is a constant parameter controlling the shape of the weighting curve. For s close to 0 *wf* produces a Gaussian curve with a narrow peak. For s greater than about 3 the curve quickly adapts to that of the parabola $1 - x^2$.

So far *wf* meets criterion 2. Moreover $wf(\mu) = max(wf)$, and most important, $wf(-1) = c$ if $asym \geq 0$ whereas $wf(1) = c$ if $asym \leq 0$ (c depending on s only). The latter is due to the given computation of σ. In order to satisfy criteria 1 and 3, the Gaussian function requires a shift by the term $y_1 = \exp(-1/s^2)$ and a scaling by the factor $y_s = 1/(1-y_1)$. Finally, the transformation $x_i = 2i/(M+1) - 1$ is used to map each index i, $0 \leq i \leq M+1$, to a value in [-1,1], which then serves as input to *wf*. Consequently, we can also view *wf* as a function with the discrete domain $\{0, \ldots, M+1\}$. Altogether, our weighting function, is a modified Gaussian function given by

$$wf(i) = wf(i, asym, s, M, m) = y_s \cdot \left(\exp\left(-\left(\frac{x_i - \mu}{\sigma}\right)^2\right) - y_1\right).$$

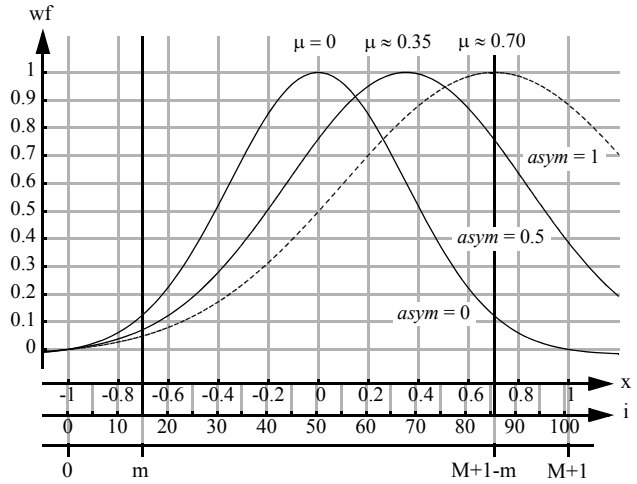

Figure 2. *wf* as a function of *x* and *i*, respectively for *asym* = 0, 0.5 and 1 (*s* = 0.5, *M* = 100, *m* = 15)

4.2.2 Motivation of the Weighting Function

After having precisely defined our weighting function, we will explain the motivation behind it in a rather graphical manner, referring to Figure 2.

The weighting function *wf* is constructed in such a way that if without loss of generality μ moves right, the left root of *wf* is fixed at 0, while the right root of *wf* moves to the right. The maximum value of the right root of *wf* is 2(*M*+1-*m*) which is reached for *asym* = 1. The interval between the roots of *wf* corresponds to the split axis of a *virtual* node. If μ > 0, this virtual node not only contains the *M*+1 rectangles of *N*, but additionally rectangles that are positioned right of N and expected to be inserted into *N* in the near future. While from view of this virtual node μ is always centered, μ may be positioned asymmetrically regarding the actual node *N*.

4.2.3 Cost of the Weighting

The CPU cost of the weighting is low. Most of the components of *wf* have to be calculated only once during a split, or are even constant over the life cycle of a tree. Only x_i and *wf* itself have to be computed for each split candidate, i.e. 2(*M*+2-2*m*) times (for two sorts) for splitting a leaf, and 2*dim*(*M*+2-2*m*) times for internal nodes. Recall that for internal nodes there is no pre-computation of a split axis. Note also that for the computation of *wf* only a single evaluation of the exponential function is necessary.

The additional storage overhead consists of a *dim*-dimensional point that has to be kept in each of the nodes. This lowers the capacity of a node by at most one entry.

4.2.4 Application of the Weighting Function

In this section we present our approach to combining the weighting function *wf* and the goal function *wg* of the R*-tree. Let us recall, that we distinguish between two different situations. If there is at least one overlap-free split candidate on the given split axis, we have to restrict our consideration to the sorted subset of split candidates with this property. Otherwise, the complete sorted sequence of candidates has to be examined. In both cases, the goal function *wg* should return a (raw) goodness value for every split candidate.

In order to build a unique goal function *w*, we ran tests with different combinations of *wg* and *wf*, which finally lead to the simple multiplicative combination, provided at the end of this section.

A meaningful interpretation of the value of *w* requires that each weighting function delivers a value in a bounded interval. Note that *wf* ranges in [0,1] and therefore satisfies this criterion. The following Lemma states that this also holds for the goal function of the R*-tree that will be explicitly defined accordingly.

Lemma 1. The goal function of the R*-tree computes a value within a bounded range.

Proof: The goal function of the R*-tree minimizes overlap. If overlap is 0, it uses the perimeter as the second criterion. Obviously the minimum for both is 0. The upper bound of the criterion overlap can be computed in the following way:

Let N be an overfilled node, and let R_N = MBB(*N*). The maximum possible overlap of a split candidate $SC_i = (F_i, S_i)$ is

$$ovlp^f_{max} = max(f(MBB(F_i) \cap MBB(S_i))), f \in \{perim, vol\}$$
$$= f(R_N \cap R_N) = f(R_N)$$

Now assume that overlap-free split candidates exist. Let $\lambda_{min}(R_N) = \min\{\lambda_1(R_N), \ldots, \lambda_{dim}(R_N)\}$ be the minimum length of an interval, resulting from the projection of R_N to its dimensions. The maximum possible perimeter of a split candidate SC_i is then

$perim_{max} = 2\left(\sum_{d=1}^{dim} \lambda_d(R_N)\right) - \lambda_{min}(R_N)$, which is the sum of the perimeters of the two rectangles, obtained by a cut perpendicular to a shortest edge of R_N. Altogether this proofs the statement of the Lemma. □

Due to Lemma 1, the goal function of the R*-tree can also be transformed into the following explicit form:

$$wg(i) = \begin{cases} perim(SC_i) - perim_{max} & if ovlp^f(SC_i) = 0 \\ ovlp^f(SC_i) & otherwise \end{cases}.$$

Note that for the overlap-free case, we do not search for the candidate with the minimum perimeter, but for the one with the maximum "free perimeter" (like free space). The function *wg* maps each index *i* to a value in the interval $[-perim_{max}, ovlp^f{}_{max}]$, where the first part returns negative values and the second part positive values. The function mirrors the second step of the split of the original R*-tree, whose strategy to choose the appropriate candidate can be re-written as $u_a \leftarrow argmin_{m \le i \le M+1-m}\{wg(i)\}$.

Now we are able to present our composed weighting function, which is given by

$$w(i) = \begin{cases} wg(i) \cdot wf(i) & if ovlp^f(SC_i) = 0 \\ \frac{wg(i)}{wf(i)} & otherwise \end{cases}.$$

Note that $wf(i) > 0$ since $0 < m \le i \le M+1-m < M+1$. The function *w* maps each index *i* to a value in the interval $[-perim_{max}, \infty)$. As *wg*, *w* yields 0 for the case, that SC_i is overlap-free, and $MBB(F_i) \cup MBB(S_i) = MBB(N)$. The second step of our new split algorithm is then: $u_a \leftarrow argmin_{m \le i \le M+1-m}\{w(i)\}$.

4.3 Parameter Settings

For the split of the original R*-tree, a single constant has to be specified: the minimum node occupation *m*, $m \le \lceil M/2 \rceil$. In [6] it was suggested to set *m* to 40% of *M*. Due to our new split algorithm, the minimum node occupation is reduced to 20% of *M*, which yields the best search performance in our experiments. This increases the probability to find an overlap-free split, but also increases the cost for splitting a node, since more split candidates are examined. An additional constant of the split algorithm is *s* that has to be set appropriately. Our experiments turned out that for *s* in the range [0.47 … 0.54] the best average search performance is achieved with a very small variance. Our final choice was $s = 0.5$.

5. EXPERIMENTS

In this section we give a detailed description of our experiments. We first provide a sketch on the generation of our data distributions as well as query distributions. The generated sets are publicly available [12]. Then, we introduce our experimental setup and discuss the most important results of our experiments.

5.1 Data Distributions

The experiments consist of 28 data sets, 21 of them are artificially generated, whereas the others are from real application scenarios. The artificial data sets belong to seven groups. Each of the groups contains a 2D, 3D and 9D data set that follows the same data distribution. The real data sets correspond to 2D, 3D, 5D, 9D, 16D, 22D and 26D distributions.

5.1.1 Artificial distributions

The most important criterion for the generation of the artificial data sets was that the distributions should be difficult to handle for index structures. This implies that some of them should be inserted in a non-random order. The seven distributions, chosen from a larger set, are characterized by their capability to reveal certain weaknesses in the R-trees. Data set *Uniform* (uniformly distributed data) is not an exception in this sense.

Before going into the details of the distributions, we need to define the terms *dithering vector*, *dithering a vector*, and the *density* of a set of objects. A *dithering vector* is a vector $D = (d_1, \ldots, d_{dim})$, where d_i randomly ranges in $[-c_i, +c_i]$, and $|c_i|$ is the dithering intensity, generally identical for all *i*. A vector *X* is *dithered* by simply adding a dithering vector *D*. The *density* is the sum of the volumes (*vol*) of a set of objects, divided by the volume of the corresponding MBB.

The artificial data sets were generated by algorithms which produce a certain distribution in an arbitrary dimensionality. Thereby the fundamental characteristic of the distribution does not change. All generated data sets consist of at least 1 million objects from $[0,1]^{dim}$, the exact number depending on the applied algorithm. Due to dithering, objects can be out of the given space. In the sequel we will provide a brief description of the artificial distributions. A detailed specification of all distributions, including a plot can be found in [12].

- **Absolute** is an equidistant distribution of equal sized squares (density 0.7), slightly dithered concerning positions and extensions. Input order: Row order, left to right.
- **Bit** is a power law distribution of points. Input order: Random.
- **Diagonal** is derived from a set of squares, equidistantly arranged along the main diagonal of the data space. Positions and extensions are dithered. The input order is along the main diagonal.
- **Parcel** is derived from a rectangle distribution, obtained by recursively splitting the data space into two partitions of random size, thereby alternating the split axis after each split. Caused by position dithering, huge rectangles often cover tiny ones. The density is 0.5, and the input order follows a z-order on the centers of the (undithered) rectangles.
- **P-edges** is a distribution of thin stripe-shaped clusters of points. Its production is derived from that of *Parcel*. Input order: Random.
- **P-haze** is a distribution of ellipse-shaped clusters of points and also related to *Parcel*. Input order: The distribution grows together from multiple points.
- **Uniform** is a uniformly distributed set of points. Input order: Random.

Table 1: Important distribution attributes / *QR0* applied

distribution	abs	abs	abs	bit	bit	bit	dia	dia	dia	par	par	par	ped	ped	ped	pha	pha	pha	uni	uni	uni	rea	rea	rea	rea	rea	rea	rea
dimensionality	02	03	09	02	03	09	02	03	09	02	03	09	02	03	09	02	03	09	02	03	09	02	03	05	09	16	22	26
data kind	rct	rct	rct	pt	pt	pt	rct	rct	rct	rct	rct	rct	pt	pt	pt	pt	pt	pt	pt	pt	pt	pt/rct	pt	pt	pt	pt	pt	pt
input order	oth	oth	oth	rand	rand	rand	oth	oth	oth	oth	oth	oth	rand	rand	rand	oth	oth	oth	rand	rand	rand	oth	rand	oth	rand	oth	oth	oth
QR0 characteristics																												
qu. index (QR0)	1	4	7	10	13	16	19	22	25	28	31	34	37	40	43	46	49	52	55	58	61	64	67	70	73	76	79	82
% empty qu.	0	0	0	0	0	0	0	0	0	0	0	0	94.3	97.8	99.9	0	0	0	0	0	0	0	0	0	0	0	0	0
min. #answers	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
max. #answers	1	1	1	1	1	1	4	3	2	10	10	5	390	520	1926	1	1	1	1	1	1	9	9	2	1	27	1	4196
avg. #answers	1	1	1	1	1	1	1.26	1.06	1.00	2.11	2.12	1.08	1.02	0.95	0.16	1	1	1	1	1	1	1.20	1.01	1.00	1	1.03	1	87.0
% std. deviat.	0	0	0	0	0	0	36.9	23.1	0.32	52.4	52.7	27.1	681	1030	4670	0	0	0	0	0	0	40.5	30.3	0.41	0	36.6	0	536

5.1.2 Real distributions

In this section we provide a very brief description of the real data sets we used in our experiments.

- **CaliforniaStreets** is a 2D distribution of 1,888,012 rectangles and points, derived from streets in California [7]. Input order: Sub-regions of roughly 20,000 objects are inserted similar to row order.
- **BioData** is a 3D distribution of 11,958,999 points extracted from a biological data set. Input order: Random.
- **CoverType** is a 5D distribution of 581,012 points from a file of the US Forest Service [31]. Input order: Non-random; directional streams are recognizable.
- **ColorMoments** is a 9D distribution of 68,040 points from image features of a Corel image collection [31]. Input order: Random.
- **Fourier** is a 16D distribution of 1,312,173 points from Fourier coefficients of CAD data. This data set was used in [5]. Input order: Some 2D views show an insertion order following a curve.
- **ActivityReport** is a 22D distribution of 500,000 points representing snapshots of the system activity report, generated by a Unix kernel. Input order: Chronological.
- **OccurrenceCount** is a 26D distribution of 427,060 points, each representing the occurrence of the letters 'a' … 'z' in a text file. The number of duplicates in this distribution is very high. Input order: Non-random.

5.2 Query Distributions

For each of the data files, we distinguish three query files, *QR0*, *QR2* and *QR3*, with a nominal number of 1, 100 and 1000 answers-per query, respectively. Altogether, there are 84 query files. Apart from a single exception, we discuss below, all queries are generated in the same manner.

File *QR0* simply consists of the center-points of the objects of the original data file. The number of answers per query is at least 1. Dithered points of *QR0* constitute the source for generating the query files *QR2* and *QR3*. For each of these points a *k*-nearest-neighbors query was performed, with an average number of neighbors $k = 100$ and $k = 1000$, the actual number of neighbors for a single query randomly set to $k_i \in [0.5k, 1.5k]$. Thereby we used the L_∞ metrics. This produces rectangular search regions containing at least k_i objects. In order to limit the query cost, we decided to produce *QR0*, *QR2* and *QR3* only for each 10-th, 100-th and 316-th, respectively of the original objects.

The advantage of creating rectangle query files in the described way is, that this allows to determine the performance of an index structure for a fixed (average) number of answers per query, but we barely measure the behavior in the empty parts of the data space. In order to include a test for this intension too, we decided to generate the query files for *P-edges* according to a volume based approach (*P-edges* contains big empty regions).The query distributions for *P-edges* are obtained as follows: First a uniformly distributed set of squares is computed. Each of them covers k / *total* of the entire data space, where *total* is the number of points in *P-edges*. In a second step the squares are dithered concerning their extensions. We consider three files *QR0*, *QR2* and *QR3* that refer to $k = 1$, $k = 100$ and $k = 1000$, respectively. To ensure validity in spite of a very few number of answers, the number of queries for *P-edges* is of the magnitude of the number of objects in the original distribution.

In Table 1 some important properties of the data distributions are illustrated by considering query *QR0*. The columns refer to the data files in the sequence introduced in Section 5.1. We will also use this order in the following figures and tables. The data files are identified by a label followed by the dimensionality. The query index (1…84) identifies the derived query files (*QR0*, *QR2*, *QR3*) and is given for *QR0* in the table. The rows of Table 1, top down, specify the following attributes: a label of the data file and its dimensionality; the data kind, i.e. whether the file consists of rectangles (rct) or points (pt) or both; the input order, i.e. random (rand) or other than random (oth); the query index with values for *QR0*; the fraction of queries with an empty result set (in percent); the minimum, maximum and average number of answers, respectively per query; finally the standard deviation of the number of answers in percent of the average.

5.3 Experimental Setup

In our experimental comparison we consider the quadratic R-tree [15], the original R*-tree [6] with ChooseSubtree parameter r set to 32, the RR*-tree and the Hilbert R-tree [19] with 8 bytes Hilbert encoding. All these structures are implemented in C within the same framework. They only differ in the layout of the nodes, the subtree choice and the split algorithm.

Except for the Hilbert R-tree, which always splits in the middle, the minimum node occupation m is a crucial parameter for the considered R-tree candidates. To establish comparability, we tested the candidates with m being set to 5%, 10%, …, 45% of M. Our final choices for m provide a compromise of best average query perfor-

mance and robustness. The determined values are partly considerably different from suggestions in earlier publications. The right column of Table 2 shows our settings for *m*.

The experimental results were computed on a Sun Fire 4200 with 2.4 GHz (64 bit) and 8 GB of memory under Solaris 10. The performance of the different structures was determined according to CPU time and I/O cost for the insertions and according to I/O cost for the queries. The CPU time is pure process (no system) time, measured with the Unix function getitimer, whereas the I/O cost was measured by simply counting the number of node accesses. The results are based on the assumption, that a single path is buffered in main memory, as a path buffer is a necessary part of any R-tree implementation. During an insertion sequence this path buffer is never cleared. As a consequence, the insertion order affects the number of node accesses. During a query sequence, the path buffer is cleared before each query, except for the root. The complete test was performed on a RAM disk. We observed similar CPU-times when a magnetic disk was used.

In this paper we report the results, obtained from experiments, where *M* is set to about 100 entries, independently of the dimensionality of the underlying distribution. Our goal was a fair comparison in a scenario adapted to the requirements of a real DBMS. Thus, for a certain dimensionality, all the index structures work on pages of the same size, which we set to a multiple of 4KB. The structures' different space demands per entry lead to the individual values of the parameter *M*, outlined in Table 2. For other choices of *M* comparable results were obtained.

Table 2: Page capacities M, setting of m

dimensionality	2D	3D	5D	9D	16D	22D	26D	m in %
page size in KB	4	4	8	16	24	36	40	
Quadratic R-tree	102	73	93	107	93	102	96	15
R*-tree								30
RR*-tree	101	72	92	107	92	101	96	20
Hilbert R-tree	85	63	85	102	90	100	94	(50)

5.4 Results

In this section we report the experimental results. The first subsection addresses the insertion performance, whereas the second is dedicated to the cost of the queries.

5.4.1 Insertion Performance

The differences in the insertion algorithms of the various R-trees influence the CPU-cost of an insertion. At the time when the original R*-tree was developed, CPU-time was generally an important part of the total insertion cost. But since then the CPU-speed has been accelerated dramatically, whereas the time for a random disk access could only be reduced moderately. Moreover the availability of main memory has largely increased over the time. Altogether this is a motivation to analyze the situation on a today's machine. In order to express the total cost of an insertion in time units, we make the following assumptions.

- The complete directory is kept in main memory, thus disk accesses only occur for the leaf level
- The CPU-speed is 2.4 GHz (64 bit)
- The time for a disk access is 5 ms

Following these assumptions, we simulated insertion of uniformly distributed data with different buffer sizes. The results are depicted in Figure 3. The horizontal axis runs over the relative buffer size, starting at 0% (pure disk) and ending at 100% (pure main memory). Note that the figure is cut into two, where the right-hand side is a magnification of the upper two percent of the left-hand side. The vertical axis shows the total runtime relative to the one of the RR*-tree (given in percent). Hence, the performance of the RR*-tree is 100% independently of the buffer size. The analysis of Figure 3 leads to the following statements:

- For insertion in main memory (100% buffer), the Hilbert R-tree is more than twice as fast as the RR*-tree and thus the clear winner.
- The original R*-tree is the most time consuming choice. The performance loss of about 15% on the left-hand side of the figure is caused by access time and due to re-insertion. The CPU-expensive ChooseSubtree algorithm of this structure leads to the upswing of the curve on the right-hand side.
- The most important result is that for all tested structures the insertion of entries is clearly I/O-bound on a modern machine. For the RR*-tree the ratio (total time) / (CPU-time) is 10.1 ms / 7.56 µs ≈ 1336. In fact, this ratio is of the same magnitude for all candidates, and their curves run nearly parallel below 98% buffer size. The ratio would be even greater if directory accesses had been taken into account too. Considering these circumstances, we decided to restrict the report of the insertion performance to the number of leaf accesses only.

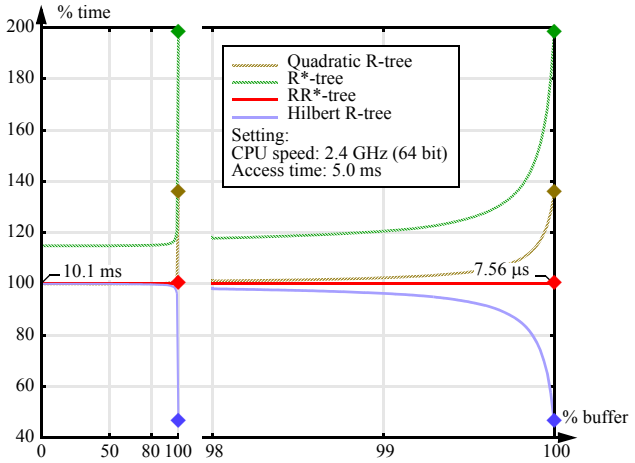

Figure 3. Insertion of *Uniform* with varying buffer sizes

All tested structures except the original R*-tree limit the number of nodes, visited during the insertion of an entry, to one single path. If *Uniform* is inserted, this leads to little more than two leaf accesses per insertion (see Table 3), which is about the value for bufferless insertion. This is due to the fact that the pages in the path buffer are hardly ever reused if the entries are inserted randomly. For the other data files the behavior is quite different. The third row of Table 3 shows the average number of leaf accesses of the RR*-tree for each data file. The last 4 rows of this table depict the performance of the four structures in relation to that of the RR*-tree (in percent). The performance of the RR*-tree is therefore always

Table 3: Absolute Insertion performance of the RR*-tree in Leaf Accesses, Comparison with its competitors in percent

distribution / dimensionality	abs 02	abs 03	abs 09	bit 02	bit 03	bit 09	dia 02	dia 03	dia 09	par 02	par 03	par 09	ped 02	ped 03	ped 09	pha 02	pha 03	pha 09	uni 02	uni 03	uni 09	rea 02	rea 03	rea 05	rea 09	rea 16	rea 22	rea 26	
avg. #leafAcc	0.37	0.70	1.10	2.01	2.02	2.01	0.04	0.06	0.04	0.47	1.10	1.88	2.01	2.02	2.01	2.00	2.02	2.01	2.01	2.02	2.01	0.52	2.02	0.60	1.92	0.67	1.95	1.82	avg
Quadratic R-tree (%)	16.0	12.3	35.9	100	100	100	141	142	144	101	96.4	98.4	100	100	100	100	100	100	100	100	100	105	100	143	101	56.3	64.1	87.9	94.5
R*-tree (%)	99.7	104	161	111	119	119	437	1570	4150	172	154	158	111	115	129	110	120	125	119	118	108	164	128	192	127	192	104	136	337
RR*-tree (%)	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
Hilbert R-tree (%)	65.1	85.7	159	100	100	100	170	101	153	127	108	101	100	100	100	100	100	100	100	100	100	103	100	129	101	230	35.3	3.51	106

100%. The right most column depicts the averages of the averages over all data files.

We observe that the RR*-tree utilizes the locality of insertions quite good. The original R*-tree needs much more leaf accesses due to re-insertion. Its extremely bad relative behavior for the *dia*-files is caused by alternating between re-inserting the entries into the current page and the least recently created neighbor page. An additional buffer path would have solved this problem. The Quadratic R-tree yields excellent insertion performance for *abs* which is inserted linewise, where the structure tendsto create very longish MBBs along the input order. Therefore buffered pages are very likely to be reused. The Hilbert R-tree yields an extremely positive outlier for *rea26* which is difficult to interpret in view of the large number of dimensions.

5.4.2 Query performance

The detailed results of the query performance of the 4 tested R-trees are presented in Figure 4. The chart shows the average number of leaf accesses in percent for the queries (*QR0*, *QR2*, *QR3*) for each of the 28 data files. The horizontal axis of Figure 4 represents the query index (1…84). Tic marks are set at every query *QR0* (compare Table 4, 7-th row). Vertically the average number of leaf accesses in percent is depicted in relation to the result of the RR*-tree which is set to 100% for each query file..

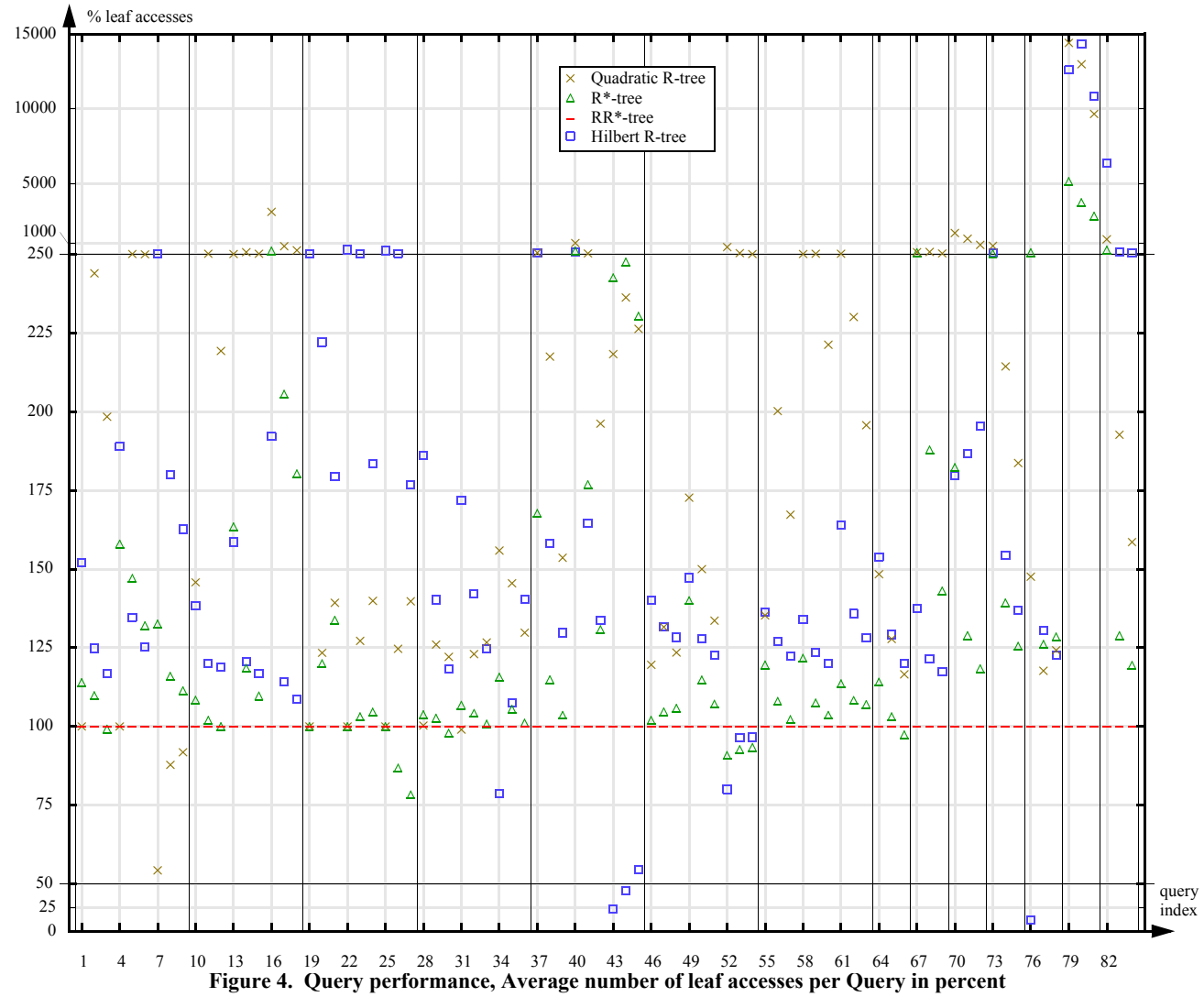

Figure 4. Query performance, Average number of leaf accesses per Query in percent

Table 4: Absolute Query performance of the RR*-tree in Leaf Accesses

distribution	abs	abs	abs	bit	bit	bit	dia	dia	dia	par	par	par	ped	ped	ped	pha	pha	pha	uni	uni	uni	rea	rea	rea	rea	rea	rea	rea
dimensionality	02	03	09	02	03	09	02	03	09	02	03	09	02	03	09	02	03	09	02	03	09	02	03	05	09	16	22	26
#leafPages in thsd	16.5	21.7	27.3	14.4	20.1	13.5	13.7	19.0	12.9	17.8	23.8	14.7	14.9	20.7	14.3	15.0	21.2	13.9	14.3	20.1	13.5	29.3	241	9.70	0.94	20.9	7.23	6.45
data kind	rct			pt			rct			rct			pt			pt			pt			pt/rct	pt	pt	pt	pt	pt	pt
peculiarities	ovlp						ovlp			ovlp +			vol. based qu.									ovlp/dup	dup	dup		dup		dup +
% perim splits	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.00	0.38	0	31.2	100	40.4
qu. index (QR0)	1	4	7	10	13	16	19	22	25	28	31	34	37	40	43	46	49	52	55	58	61	64	67	70	73	76	79	82
avg #answers	1	1	1	1	1	1	1.26	1.06	1.00	2.11	2.12	1.08	1.02	0.95	0.16	1	1	1	1	1	1	1.20	1.01	1.00	1	1.03	1	87.0
avg #leafAcc	1.00	1.00	3.13	1.07	1.04	1.01	1.00	1.00	1.00	3.11	10.3	229	0.20	0.10	0.35	1.02	1.10	3.71	1.02	1.06	1.03	1.28	1.28	1.19	1.04	45.4	3.62	82.6
qu. index (QR2)	2	5	8	11	14	17	20	23	26	29	32	35	38	41	44	47	50	53	56	59	62	65	68	71	74	77	80	83
avg #answers	99.8	99.8	99.8	99.8	99.8	99.8	99.8	99.8	99.8	99.9	99.9	99.9	102	93.6	13.6	99.8	99.8	99.9	99.8	99.8	99.8	101	100	102	99.4	105	230	13700
avg #leafAcc	4.92	9.65	156	4.51	9.45	126	2.35	2.87	2.27	7.34	23.5	859	2.28	2.69	2.75	4.72	10.8	259	4.64	10.6	156	4.89	11.6	18.0	55.5	582	27.3	763
qu. index (QR3)	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	57	60	63	66	69	72	75	78	81	84
avg #answers	992	992	992	992	992	992	992	992	992	994	994	994	1000	882	111	992	993	992	992	992	992	999	999	1000	1000	1010	1150	23400
avg #leafAcc	24.9	47.0	538	21.8	43.5	371	14.5	19.8	13.8	28.5	73.7	1780	16.8	20.8	9.77	22.6	47.9	751	22.3	47.9	459	23.5	51.1	60.9	141	1860	44.1	996

Note that on the vertical axis the interval [50% … 250%] is spread. The actual number of leaf accesses at 100% (RR*-tree) can be extracted from Table 4, which consists of a header part (6 rows) followed by three body parts (3 rows each). The header part contains the label of the data file (2 rows) followed by the total number of leaf pages (in thousand). The 4-th row contains the data kind (see also Table 1). The 5-th row specifies whether the data file contains overlapping rectangles (ovlp) or duplicates (dup) and labels the queries generated by the volume bound approach; a "+" is used as a marker for high degree. The 6-th row contains the fraction of splits (in percent), where the base of optimization was switched from volume to perimeter. Each of the three body parts is dedicated to one of the three query types and contains the following information: The query index for the considered query; the average number of answers for the specified query; finally the average number of leaf accesses for the specified query.

Let us first take a closer look at Figure 4. An overview of the graph shows that none of the tested index structures yields superior query performance in every situation. The quadratic R-tree beats all its competitors for *abs09*, the original R*-tree for *dia09*, the Hilbert R-tree for *ped09*. The latter additionally yields a single extremely positive outlier of 12.1% for *QR0* on *rea16*, which is difficult to interpret in view of the large number of dimensions. At *pha09* finally the RR*-tree is beaten by the original R*-tree and the Hilbert R-tree by about 10% on average. For the remaining 23 of 28 distributions the RR*-tree clearly outperforms its competitors concerning the average cost of the three query types.

Before we will consider some absolute results of the RR*-tree, we want to mention that the space utilization in the leaf level is 68% on average with a standard deviation of only 3.4% over all 28 distributions.

Table 4 provides the average number of leaf accesses as a measure for the absolute performance of the RR*-tree. For *QR0* (except for *ped*, whose queries may yield no answers) the theoretically achievable optimum for this parameter is 1 if the average number of answers is 1. If the number of answers is greater due to overlap or duplicates, we have to accept a greater number of accesses. The table shows that the performance of the RR*-tree for *QR0* is almost always very good, sometimes excellent, at least for dimensions 2D … 5D. Even the values for *par02* and *par03* are acceptable in view of the kind of these distributions. The performance for the 9D … 26D data files is not that good anymore, apart from some pleasing exceptions. For *par09*, *rea16* and *rea26* the performance of the RR*-tree is quite disappointing. But only for *rea16* a competitor (the Hilbert R-tree) yields an acceptable performance value of 5.49.

For *QR2* and *QR3* a theoretical optimum for the number of disk accesses cannot easily be determined. But it is often stated as a rule of thumb, that the usage of an index structure will be profitable only if the number of disk accesses would be less than 10% of the total number of pages of the data file, because a simple sequential scan would be faster otherwise. Obviously the ratio #accesses/ #pages strongly depends on the size of a data file as well as on the number of actually retrieved objects. Nevertheless, we observed that for *QR2* the limit of 10% is only exceeded at *rea26* (where 13700 answers are returned). For *par09*, *rea09* and *rea16* a critical magnitude is reached. Similar weaknesses occur for query *QR3* on the same set of data files.

Regarding *QR2* and *QR3* it is remarkable that the increase of accesses with a growing number of dimensions, the "curse of dimensionality", is large for the artificially generated distributions (except *dia*), while for the real distributions the effect is much smaller on average. This is probably due to the following reasons. The artificial distributions are unitary constructed for different dimensionalities, and there is nearly no correlation among the different dimensions. An exception is *dia*, where the correlation is extremely high. Thus the so-called intrinsic dimensionality of *dia* is almost only 1 independently of the native dimensionality. In real distributions however, the data dimensions tend to be highly correlated. Moreover, groups of points are often located in lower dimensional subspaces. Hence their intrinsic dimensionality is frequently much smaller than their native dimensionality. As an indicator for this phenomenon the 6-th row of Table 4 shows the fraction of splits, where the RR*-tree switched to perimeter based optimization (see Section 4.1). For the most distinctive example, *rea22*, the following effects are notable: First, our query generation for *QR2* and *QR3* inevitably lead to more answers than planned (see Table 4). Secondly, R-trees limited to volume-based optimization can barely cope with such kind of data files (see Figure 4).

5.4.3 Query performance on average

Table 5 shows the relative query performance of the 4 tested structures in percent. The columns, from left to right, depict the average

over a growing range of dimensionalities in relation to the result of the RR*-tree, which is set to 100% for each range. The last column presents the overall performance, i.e. the average of the averages of all tests and is thus a summarization of Figure 4. We can state that the RR*-tree is superior to the other R-trees by at least 30%. Regarding its closest competitor, the original R*-tree, the performance gain increases with a growing number of dimensions. One reason for the poor performance of the Hilbert R-tree for dimensionalities greater 16 might be that the size of the Hilbert-value is limited to 64 bits, i.e. to 2 bits only per dimension then. However, a larger number of bits, say 128, would also lead to a smaller page capacity of the Hilbert R-tree. Regarding the quadratic R-tree, which is the current implementation of R-trees in mySQL and PostgeSQL, the RR*-tree performs already better by a factor of 2 for low-dimensional data, while larger improvements can be observed for high-dimensional data.

Table 5: Average Query performance, Leaf accesses in percent

-	2D … 3D	2D … 9D	2D … 16D	2D … 26D
Quadratic R-tree	209	310	304	742
R*-tree	131	139	142	280
RR*-tree	100	100	100	100
Hilbert R-tree	165	164	161	683

6. Conclusions

In this paper, we revealed deficiencies in the design of previously proposed R-trees, in particular of the R*-tree. None of the members of the R-tree family has taken into account neither the imbalance of a split as an optimization criterion nor the fact that volume-based optimization becomes ineffective when bounding boxes are in lower-dimensional subspaces. Moreover, a complex optimization technique of the R*-tree, the re-insertion, can only be used off-line in a DBMS.

In order to overcome these deficiencies, we proposed a revision of the R*-tree termed Revised R*-tree (RR*-tree). Since an insertion is guaranteed to be restricted to a single path, standard concurrency protocols are also eventually applicable to the RR*-tree. The search performance could be further improved due to new advanced optimization strategies. The treatment of multidimensional data has been improved, as adaptive strategies were proposed that switch to a perimeter-based optimization when volume-based optimization fails. The experimental setup presented in the paper is unique, as a large variety of synthetic and real data files were examined. Furthermore, queries were considered with a limited number of answers rather than with a limited volume. Our experimental performance comparison showed that the RR*-tree was the most robust R-tree and superior for most of the data files and queries.

We believe that the RR*-tree is ideally suited for being implemented within a DBMS as it provides excellent performance and does not collide with the requirements of common concurrency protocols.

7. References

[1] L. Arge, M. Berg, H. Haverkort, K. Yi: The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree. SIGMOD Conference 2004: 347-358

[2] C. Böhm, S. Berchtold, D. Keim: Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Computing Surveys 33(3): 322-373 (2001).

[3] J. L. Bentley: Multidimensional Binary Search Trees Used for Associative Searching. Commun. ACM 18(9): 509-517 (1975).

[4] B. Becker, P. Franciosa, S. Gschwind, T. Ohler, G. Thiemt, P. Widmayer: Enclosing Many Boxes by an Optimal Pair of Boxes. STACS 1992: 475-486.

[5] S. Berchtold, D. Keim, H.-P. Kriegel: The X-tree : An Index Structure for High-Dimensional Data. VLDB Conference 1996: 28-39.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger: The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD Conference 1990: 322-331.

[7] J. van den Bercken, B. Seeger, P. Widmayer: A Generic Approach to Bulk Loading Multidimensional Index Structures. VLDB 1997: 406-415.

[8] Bureau of the Census: Tiger/Line Precensus Files: 1995 technical documentation, Bureau of the Census, Washington DC 1996.

[9] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, M. Zwilling: Shoring Up Persistent Applications. SIGMOD Conference 1994: 383-394.

[10] K. Chakrabarti, S. Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. ICDE 1999: 440-447.

[11] K. Chakrabarti, S. Mehrotra: Efficient Concurrency Control in Multidimensional Access Methods. SIGMOD Conference 1999: 25-36.

[12] http://www.mathematik.uni-marburg.de/~seeger/rrstar/, 2009.

[13] C. Faloutsos, I. Kamel: Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension. PODS 1994: 4-13.

[14] Y. García, M. Lopez, S. Leutenegger: On Optimal Node Splitting for R-trees. VLDB Conference 1998: 334-344.

[15] A. Guttman. R-trees: A dynamic index structure for spatial searching. SIGMOD Conference 1984: 47-57, 1984.

[16] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems (Extended Abstract). VLDB Conference 1995: 562-573.

[17] E. Hoel, H. Samet: Benchmarking Spatial Join Operations with Spatial Output. VLDB Conference 1995: 606-618.

[18] IBM Informix R-tree Index, User's Guide, 2005. IBM Informix Dynamic Server v10 Information Center, Feb. 2009.

[19] I. Kamel, C. Faloutsos, "Hilbert R-tree: An Improved R-tree Using Fractals", VLDB Conference 1994: 500-509.

[20] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and Recovery in Generalized Search Trees. SIGMOD Conference 1997: 62-72.

[21] K. Kanth, S. Ravada, D. Abugov: Quadtree and R-tree Indexes in Oracle Spatial: A Comparison using GIS Data. SIGMOD Conference 2002.

[22] K. Kanth, S. Ravada, J. Sharma, J. Banerjee: Indexing Medium-dimensionality Data in Oracle. SIGMOD Conference 1999: 521-522.

[23] N. Katayama and S. Satoh, The sr-tree: An index structure for high-dimensional nearest neighbor queries, SIGMOD Conference 1997: 369-380.

[24] Y. Manolopoulos, A. Nanopoulos, A.N. Papadopoulos, Y. Theodoridis: R-trees: Theory and Applications, Springer, 2005.

[25] MySQL 5.0 Reference Manual, http://downloads.mysql.com/docs/refman-5.0-en.a4.pdf, 2008.

[26] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin. The QPIC project: Quering images by content using color, texture, and shape. Poc. of the SPIE Conference on Storage and Retrieval for Image and Video Databases, 2-3 February '93, San Jose, CA, pages 173-187, 1993.

[27] The PostgreSQL Comprehensive Manual, http://www.postgresql.org/docs/manuals/).

[28] B.-U. Pagel, H.-W. Six, H. Toben, P. Widmayer: Towards an Analysis of Range Query Performance in Spatial Data Structures. PODS 1993: 214-221.

[29] B.-U. Pagel, H.-W. Six, M. Winter: Window Query- Optimal Clustering of Spatial Objects. PODS 1995: 86-94.

[30] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima: "The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation", VLDB Conference 2000: 516--526.

[31] UC Irvine KDD Archive, http://kdd.ics.uci.edu/, 2002.

[32] J. Weinberger: SpatialWare Extends Microsoft SQL Server Capabilities. MapInfo Magazine 6(2), 2001