The R*-tree: An Efficient and Robust Access Method for Points and Rectangles+

Norbert Beckmann, Hans-Peter Kriegel
Ralf Schneider, Bernhard Seeger
Praktische Informatik, Universitaet Bremen, D-2800 Bremen 33, West Germany

Abstract

The R-tree, one of the most popular access methods for rectangles, is based on the heuristic optimization of the area of the enclosing rectangle in each inner node By running numerous experiments in a standardized testbed under highly varying data, queries and operations, we were able to design the R*-tree which incorporates a combined optimization of area, margin and overlap of each enclosing rectangle in the directory Using our standardized testbed in an exhaustive performance comparison, it turned out that the R*-tree clearly outperforms the existing R-tree variants Guttman's linear and quadratic R-tree and Greene's variant of the R-tree This superiority of the R*-tree holds for different types of queries and operations, such as map overlay, for both rectangles and multidimensional points in all experiments From a practical point of view the R*-tree is very attractive because of the following two reasons 1 it efficiently supports point and spatial data at the same time and 2 its implementation cost is only slightly higher than that of other R-trees

1.Introduction

In this paper we will consider spatial access methods (SAMs) which are based on the approximation of a complex spatial object by the minimum bounding rectangle with the sides of the rectangle parallel to the axes of the data space The most important property of this simple approximation is that a complex object is represented by a limited number of bytes Although a lot of information is lost, minimum bounding rectangles of spatial objects preserve the most essential geometric properties of the object, i e the location of the object and the extension of the object in each axis

In [SK 88] we showed that known SAMs organizing (minimum bounding) rectangles are based on an underlying point access method (PAM) using one of the following three techniques clipping, transformation and overlapping regions

The most popular SAM for storing rectangles is the R-tree [Gut 84] Following our classification, the R-tree is based on the PAM B+-tree [Knu 73] using the technique over-lapping regions Thus the R-tree can be easily implemented which considerably contributes to its popularity

The R-tree is based on a heuristic optimization The optimization criterion which it persues, is to minimize the area of each enclosing rectangle in the inner nodes This criterion is taken for granted and not shown to be the best possible Questions arise such as Why not minimize the margin or the overlap of such minimum bounding rectangles Why not optimize storage utilization? Why not optimize all of these criteria at the same time? Could these criteria interact in a negative way? Only an engineering approach will help to find the best possible combination of optimization criteria

Necessary condition for such an engineering approach is the availability of a standardized testbed which allows us to run large volumes of experiments with highly varying data, queries and operations We have implemented such a standardized testbed and used it for performance comparisons particularly of point access methods [KSSS 89]

As the result of our research we designed a new R-tree variant, the R*-tree, which outperforms the known R-tree variants under all experiments For many realistic profiles of data and operations the gain in performance is quite considerable Additionally to the usual point query,

+ This work was supported by grant no Kr 670/4-3 from the Deutsche Forschungsgemeinschaft (German Research Society) and by the Ministry of Environmental and Urban Planning of Bremen

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery To copy otherwise, or to republish requires a fee and/or specific permission
© 1990 ACM 089791 365 5/90/0005/0322. $1 50

rectangle intersection and rectangle enclosure query, we have analyzed our new R*-tree for the map overlay operation, also called spatial join, which is one of the most important operations in geographic and environmental database systems

This paper is organized as follows In section 2, we introduce the principles of R-trees including their optimization criteria In section 3 we present the existing R-tree variants of Guttman and Greene Section 4 describes in detail the design our new R*-tree The results of the comparisons of the R*-tree with the other R-tree variants are reported in section 5 Section 6 concludes the paper

2. Principles of R-trees and possible optimization criteria

An R-tree is a B^+-tree like structure which stores multidimensional rectangles as complete objects without clipping them or transforming them to higher dimensional points before

A non-leaf node contains entries of the form *(cp, Rectangle)* where *cp* is the address of a child node in the R-tree and *Rectangle* is the minimum bounding rectangle of all rectangles which are entries in that child node A leaf node contains entries of the form *(Oid, Rectangle)* where *Oid* refers to a record in the database, describing a spatial object and *Rectangle* is the enclosing rectangle of that spatial object Leaf nodes containing entries of the form *(dataobject, Rectangle)* are also possible This will not affect the basic structure of the R-tree In the following we will not consider such leaf nodes

Let M be the maximum number of entries that will fit in one node and let m be a parameter specifying the minimum number of entries in a node ($2 \leq m \leq M/2$) An R-tree satisfies the following properties

- The root has at least two children unless it is a leaf
- Every non-leaf node has between m and M children unless it is the root
- Every leaf node contains between m and M entries unless it is the root
- All leaves appear on the same level

An R-tree (R*-tree) is completely dynamic, insertions and deletions can be intermixed with queries and no periodic global reorganization is required Obviously, the structure must allow overlapping directory rectangles Thus it cannot guarantee that only one search path is required for an exact match query For further information we refer to [Gut84]
We will show in this paper that the overlapping-regions-technique does not imply bad average retrieval performance Here and in the following, we use the term directory rectangle, which is geometrically the minimum bounding rectangle of the underlying rectangles

The main problem in R-trees is the following For an arbitrary set of rectangles, dynamically build up bounding boxes from subsets of between m and M rectangles, in a way that arbitrary retrieval operations with query rectangles of arbitrary size are supported efficiently The known parameters of good retrieval performance affect each other in a very complex way, such that it is impossible to optimize one of them without influencing other parameters which may cause a deterioration of the overall performance Moreover, since the data rectangles may have very different size and shape and the directory rectangles grow and shrink dynamically, the success of methods which will optimize one parameter is very uncertain Thus a heuristic approach is applied, which is based on many different experiments carried out in a systematic framework

In this section some of the parameters which are essential for the retrieval performance are considered Furthermore, interdependencies between different parameters and optimization criteria are analyzed

(O1) *The area covered by a directory rectangle should be minimized*, i e the area covered by the bounding rectangle but not covered by the enclosed rectangles, the dead space, should be minimized This will improve performance since decisions which paths have to be traversed, can be taken on higher levels

(O2) *The overlap between directory rectangles should be minimized* This also decreases the number of paths to be traversed

(O3) *The margin of a directory rectangle should be minimized* Here the margin is the sum of the lengths of the edges of a rectangle Assuming fixed area, the object with the smallest margin is the square Thus minimizing the margin instead of the area, the directory rectangles will be shaped more quadratic Essentially queries with large quadratic query rectangles will profit from this optimization More important, minimization of the margin will basically improve the structure Since quadratic objects can be packed easier, the bounding boxes of a level will build smaller directory rectangles in the level above Thus clustering rectangles into bounding boxes with only little variance of the lengths of the edges will reduce the area of directory rectangles

(O4) *Storage utilization should be optimized* Higher storage utilization will generally reduce the query cost as the height of the tree will be kept low Evidently, query types with large query rectangles are influenced more since the concentration of rectangles in several nodes will have a stronger effect if the number of found keys is high

Keeping the area and overlap of a directory rectangle small, requires more freedom in the number of rectangles stored in one node Thus minimizing these parameters will be paid with lower storage utilization. Moreover, when applying (O1) or (O2) more freedom in choosing the shape is necessary Thus rectangles will be less quadratic With (O1) the overlap between directory rectangles may be affected in a positive way since the covering of the data space is reduced As for every geometric optimization, minimizing the margins will also lead to reduced storage utilization However, since more quadratic directory rectangles support

packing better, it will be easier to maintain high storage utilization Obviously, the performance for queries with sufficiently large query rectangles will be affected more by the storage utilization than by the parameters of (O1)-(O3)

3. R-tree Variants

The R-tree is a dynamic structure Thus all approaches of optimizing the retrieval performance have to be applied during the insertion of a new data rectangle The insertion algorithm calls two more algorithms in which the crucial decisions for good retrieval performance are made The first is the algorithm ChooseSubtree Beginning in the root, descending to a leaf, it finds on every level the most suitable subtree to accomodate the new entry The second is the algorithm Split It is called, if ChooseSubtree ends in a node filled with the maximum number of entries M Split should distribute M+1 rectangles into two nodes in the most appropriate manner

In the following, the ChooseSubtree- and Split-algorithms, suggested in available R-tree variants are analyzed and discussed We will first consider the original R-tree as proposed by Guttman in [Gut 84]

Algorithm ChooseSubtree

CS1 Set N to be the root
CS2 If N is a leaf,
  return N
  else
  Choose the entry in N whose rectangle needs least area enlargement to include the new data Resolve ties by choosing the entry with the rectangle of smallest area
  end
CS3 Set N to be the childnode pointed to by the childpointer of the chosen entry an repeat from CS2

Obviously, the method of optimization is to minimize the area covered by a directory rectangle, i e (O1) This may also reduce the overlap and the cpu cost will be relatively low

Guttman discusses split-algorithms with exponential, quadratic and linear cost with respect to the number of entries of a node All of them are designed to minimize the area, covered by the two rectangles resulting from the split The exponential split finds the area with the global minimum, but the cpu cost is too high The others try to find approximations In his experiments, Guttman obtains nearly the same retrieval performance for the linear as for the quadratic version We implemented the R-tree in both variants However in our tests with different distributions, different overlap, variable numbers of data-entries and different combinations of M and m, the quadratic R-tree yielded much better performance than the linear version (see also section 5) Thus we will only discuss the quadratic algorithm in detail

Algorithm QuadraticSplit

[Divide a set of M+1 entries into two groups]
QS1 Invoke PickSeeds to choose two entries to be the first entries of the groups
QS2 Repeat
  DistributeEntry
  until
  all entries are distributed or
  one of the two groups has M-m+1 entries
QS3 If entries remain, assign them to the other group such that it has the minimum number m

Algorithm PickSeeds

PS1 For each pair of entries E1 and E2, compose a rectangle R including E1 rectangle and E2 rectangle
Calculate d = area(R) - area(E1 rectangle) - area(E2 rectangle)
PS2 Choose the pair with the largest d

Algorithm DistributeEntry

DE1 Invoke PickNext to choose the next entry to be assigned
DE2 Add it to the group whose covering rectangle will have to be enlarged least to accommodate it Resolve ties by adding the entry to the group with the smallest area, then to the one with the fewer entries, then to either

Algorithm PickNext

PN1 For each entry E not yet in a group, calculate d_1 = the area increase required in the covering rectangle of Group 1 to include E Rectangle
Calculate d_2 analogously for Group 2
PN2 Choose the entry with the maximum difference between d_1 and d_2

The algorithm PickSeeds finds the two rectangles which would waste the largest area put in one group In this sense the two rectangles are the most distant ones It is important to mention that the seeds will tend to be small too, if the rectangles to be distributed are of very different size (and) or the overlap between them is high The algorithm DistributeEntry assigns the remaining entries by the criterion of minimum area PickNext chooses the entry with the best area-goodness-value in every situation

If this algorithm starts with small seeds, problems may occur If in d-1 of the d axes a far away rectangle has nearly the same coordinates as one of the seeds, it will be distributed first Indeed, the area and the area enlargement of the created needle-like bounding rectangle will be very small, but the distance is very large This may initiate a very bad split Moreover, the algorithm tends to prefer the bounding rectangle, created from the first assignment of a rectangle to one seed Since it was enlarged, it will be larger than others Thus it needs less area enlargement to include the next entry, it will be enlarged again, and so on Another problem is, that if one group has reached the maximum number of entries M-m+1, all remaining entries are assigned to the other group without considering geometric properties Figure 1 (see section 4 3) gives an example showing all

these problems The result is either a split with much overlap (fig 1c) or a split with uneven distribution of the entries reducing the storage utilization (fig 1b)

We tested the quadratic split of our R-tree implementation varying the minimum number of entries m = 20%, 30%, 35% ,40% and 45% relatively to M and obtained the best retrieval performance with m set to 40%

On the occasion of comparing the R-tree with other structures storing rectangles, Greene proposed the following alternative split-algorithm [Gre 89] To determine the appropriate path to insert a new entry she uses Guttman's original ChooseSubtree-algorithm

Algorithm Greene's-Split
[Divide a set of M+1 entries into two groups]
GS1 Invoke ChooseAxis to determine the axis perpendicular to which the split is to be performed
GS2 Invoke Distribute

Algorithm ChooseAxis
CA1 Invoke PickSeeds (see p 5) to find the two most distant rectangles of the current node
CA2 For each axis record the separation of the two seeds
CA3 Normalize the separations by dividing them by the length of the nodes enclosing rectangle along the appropriate axis
CA4 Return the axis with the greatest normalized separation

Algorithm Distribute
D1 Sort the entries by the low value of their rectangles along the chosen axis
D2 Assign the first (M+1) div 2 entries to one group, the last (M+1) div 2 entries to the other
D3 If M+1 is odd, then assign the remaining entry to the group whose enclosing rectangle will be increased least by its addition

Almost the only geometric criterion used in Greene's split algorithm is the choice of the split axis Although choosing a suitable split axis is important, our investigations show that more geometric optimization criteria have to be applied to considerably improve the retrieval performance of the R-tree In spite of a well clustering, in some situations Greene's split method cannot find the "right" axis and thus a very bad split may result Figure 2b (see p 12) depicts such a situation

4. The R*-tree

4.1 Algorithm ChooseSubtree
To solve the problem of choosing an appropriate insertion path, previous R-tree versions take only the area parameter into consideration In our investigations, we tested the parameters area, margin and overlap in different combinations, where the overlap of an entry is defined as follows

Let $E_1, \ldots, E_p$ be the entries in the current node Then

$$\text{overlap}(E_k) = \sum_{i=1, i \neq k}^{p} \text{area}(E_k.\text{Rectangle} \cap E_i.\text{Rectangle}) \quad , 1 \leq k \leq p$$

The version with the best retrieval performance is described in the following algorithm

Algorithm ChooseSubtree
CS1 Set N to be the root
CS2 If N is a leaf,
  return N
 else
  if the childpointers in N point to leaves [determine the minimum overlap cost],
   choose the entry in N whose rectangle needs least overlap enlargement to include the new data rectangle Resolve ties by choosing the entry whose rectangle needs least area enlargement,
  then
   the entry with the rectangle of smallest area
  if the childpointers in N do not point to leaves [determine the minimum area cost],
   choose the entry in N whose rectangle needs least area enlargement to include the new data rectangle Resolve ties by choosing the entry with the rectangle of smallest area
 end
CS3 Set N to be the childnode pointed to by the childpointer of the chosen entry and repeat from CS2

For choosing the best non-leaf node, alternative methods did not outperform Guttman's original algorithm For the leaf nodes, minimizing the overlap performed slightly better

In this version, the cpu cost of determining the overlap is quadratic in the number of entries, because for each entry the overlap with all other entries of the node has to be calculated However, for large node sizes we can reduce the number of entries for which the calculation has to be done, since for very distant rectangles the probability to yield the minimum overlap is very small Thus, in order to reduce the cpu cost, this part of the algorithm might be modified as follows

> [determine the *nearly* minimum overlap cost]
> Sort the rectangles in N in increasing order of their area enlargement needed to include the new data rectangle
> Let A be the group of the first p entries
> From the entries in A, considering all entries in N, choose the entry whose rectangle needs least overlap enlargement Resolve ties as described above

For two dimensions we found that with p set to 32 there is nearly no reduction of retrieval performance to state For more than two dimensions further tests have to be done Nevertheless the cpu cost remains higher than the original version of ChooseSubtree, but the number of disc accesses

is reduced for the exact match query preceding each insertion and is reduced for the ChooseSubtree algorithm itself

The tests showed that the ChooseSubtree optimization improves the retrieval performance particulary in the following situation *Queries with small query rectangles on datafiles with non-uniformly distributed small rectangles or points*

In the other cases the performance of Guttman's algorithm was similar to this one Thus principally an improvement of robustness can be stated

4 2 Split of the R*-tree

The R*-tree uses the following method to find good splits Along each axis, the entries are first sorted by the lower value, then sorted by the upper value of their rectangles For each sort M-2m+2 distributions of the M+1 entries into two groups are determined, where the k-th distribution (k = 1, ,(M-2m+2)) is described as follows The first group contains the first (m-1)+k entries, the second group contains the remaining entries

For each distribution goodness values are determined Depending on these goodness values the final distribution of the entries is determined Three different goodness values and different approaches of using them in different combinations are tested experimentally

(i)	area-value	area[bb(first group)] + area[bb(second group)]
(ii)	margin-value	margin[bb(first group)] + margin[bb(second group)]
(iii)	overlap-value	area[bb(first group) ∩ bb(second group)]

Here bb denotes the bounding box of a set of rectangles

Possible methods of processing are to determine

- the minimum over one axis or one sort
- the minimum of the sum of the goodness values over one axis or one sort
- the overall minimum

The obtained values may be applied to determine a split axis or the final distribution (on a chosen split axis) The best overall performance resulted from the following algorithm

Algorithm Split

S1 Invoke ChooseSplitAxis to determine the axis, perpendicular to which the split is performed

S2 Invoke ChooseSplitIndex to determine the best distribution into two groups along that axis

S3 Distribute the entries into two groups

Algorithm ChooseSplitAxis

CSA1 For each axis
Sort the entries by the lower then by the upper value of their rectangles and determine all distributions as described above Compute S, the sum of all margin-values of the different distributions
end

CSA2 Choose the axis with the minimum S as split axis

Algorithm ChooseSplitIndex

CSI1 Along the chosen split axis, choose the distribution with the minimum overlap-value Resolve ties by choosing the distribution with minimum area-value

The split algorithm is tested with m = 20%, 30%, 40% and 45% of the maximum number of entries M As experiments with several values of M have shown, m = 40% yields the best performance Additionally, we varied m over the life cycle of one and the same R*-tree in order to correlate the storage utilization with geometric paremeters However, even the following method did result in worse retrieval performance Compute a split using m_1 = 30% of M, then compute a split using m_2 =40% If split(m_2)yields overlap and split(m_1) does not, take split(m_1), otherwise take split(m_2)

Concerning the cost of the split algorithm of the R*-tree we will mention the following facts For each axis (dimension) the entries have to be sorted two times which requires O(M log(M)) time As an experimental cost analysis has shown, this needs about half of the cost of the split The remaining split cost is spent as follows For each axis the margin of 2*(2*(M-2m+2)) rectangles and the overlap of 2*(M-2m+2) distributions have to be calculated

4 3 Forced Reinsert

Both, R-tree and R*-tree are nondeterministic in allocating the entries onto the nodes i e different sequences of insertions will build up different trees For this reason the R-tree suffers from its old entries Data rectangles inserted during the early growth of the structure may have introduced directory rectangles, which are not suitable to guarantee a good retrieval performance in the current situation A very local reorganization of the directory rectangles is performend during a split But this is rather poor and therefore it is desirable to have a more powerful and less local instrument to reorganize the structure

The discussed problem would be maintained or even worsened, if underfilled nodes, resulting from deletion of records would be merged under the old parent Thus the known approach of treating underfilled nodes in an R-tree is to delete the node and to reinsert the orphaned entries in the corresponding level [Gut 84] This way the ChooseSubtree algorithm has a new chance of distributing entries into different nodes

Since it was to be expected, that the deletion and reinsertion of old data rectangles would improve the retrieval performance, we made the following simple experiment with the linear R-tree Insert 20000 uniformly distributed rectangles Delete the first 10000 rectangles and insert them again The result was a performance improvement of 20% up to 50%(!) depending on the types of the queries Therefore to delete randomly half of the data and then to insert it again seems to be a very simple way of tuning existing R-tree datafiles But this is a static situation, and for nearly static datafiles the pack algorithm [RL 85] is a more sophisticated approach

To achieve dynamic reorganizations, the R*-tree forces entries to be reinserted during the insertion routine The

following algorithm is based on the ability of the insert routine to insert entries on every level of the tree as already required by the deletion algorithm [Gut 84] Except for the overflow treatment, it is the same as described originally by Guttman and therefore it is only sketched here

Algorithm InsertData

ID1 Invoke Insert starting with the leaf level as a parameter, to insert a new data rectangle

Algorithm Insert

I1 Invoke ChooseSubtree, with the level as a parameter, to find an appropriate node N, in which to place the new entry E

I2 If N has less than M entries, accommodate E in N
If N has M entries, invoke OverflowTreatment with the level of N as a parameter [for reinsertion or split]

I3 If OverflowTreatment was called and a split was performed, propagate OverflowTreatment upwards if necessary
If OverflowTreatment caused a split of the root, create a new root

I4 Adjust all covering rectangles in the insertion path such that they are minimum bounding boxes enclosing their children rectangles

Algorithm OverflowTreatment

OT1 If the level is not the root level and this is the first call of OverflowTreatment in the given level during the insertion of one data rectangle, then
    invoke ReInsert
else
    invoke Split
end

Algorithm ReInsert

RI1 For all M+1 entries of a node N, compute the distance between the centers of their rectangles and the center of the bounding rectangle of N

RI2 Sort the entries in decreasing order of their distances computed in RI1

RI3 Remove the first p entries from N and adjust the bounding rectangle of N

RI4 In the sort, defined in RI2, starting with the maximum distance (= far reinsert) or minimum distance (= close reinsert), invoke Insert to reinsert the entries

If a new data rectangle is inserted, each first overflow treatment on each level will be a reinsertion of p entries This may cause a split in the node which caused the overflow if all entries are reinserted in the same location Otherwise splits may occur in one or more other nodes, but in many situations splits are completely prevented The parameter p can be varied independently for leaf nodes and non-leaf nodes as part of performance tuning, and different values were tested experimentally The experiments have shown that p = 30% of M for leaf nodes as well as for non-leaf nodes yields the best performance Furthermore, for all data files and query files close reinsert outperforms far reinsert Close reinsert prefers the node which included the entries before, and this is intended, because its enclosing rectangle was reduced in size Thus this node has lower probability to be selected by ChooseSubtree again

Summarizing, we can say

- Forced reinsert changes entries between neighboring nodes and thus decreases the overlap
- As a side effect, storage utilization is improved
- Due to more restructuring, less splits occur
- Since the outer rectangles of a node are reinserted, the shape of the directory rectangles will be more quadratic As discussed before, this is a desirable property

Obviously, the cpu cost will be higher now since the insertion routine is called more often This is alleviated, because less splits have to be performed The experiments show that the average number of disc accesses for insertions increases only about 4% (and remains the lowest of all R-tree variants), if Forced Reinsert is applied to the R*-tree This is particularly due to the structure improving properties of the insertion algorithm

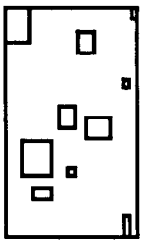

Figure 1a Overfilled node

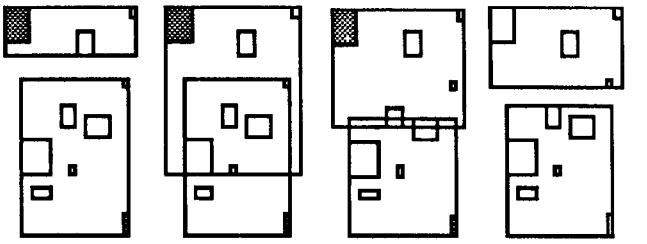

Figure 1b: Split of the quadratic R-tree, m = 30%

Figure 1c: Split of the quadratic R-tree, m = 40%

Figure 1d Greene's split

Figure 1e Split of the R*-tree, m = 40%

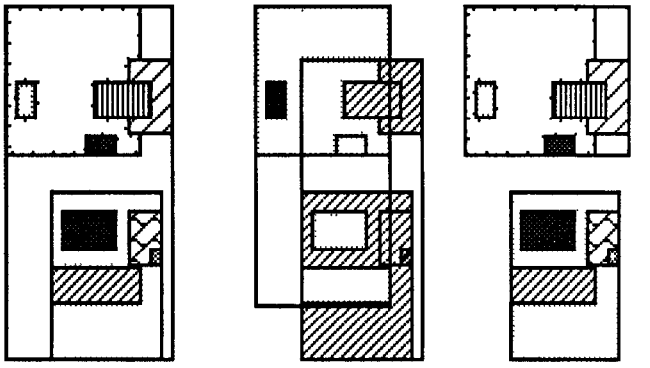

Figure 2a Overfilled node

Figure 2b: Greene's split where the splitaxis is horizontal

Figure 2c: Split of the R* tree where the splitaxis is vertical

5. Performance Comparison

5 1 Experimental Setup and Results of the Experiments

We ran the performance comparison on SUN workstations under UNIX using Modula-2 implementaions of the different R-tree variants and our R*-tree Analogously to our performance comparison of PAM's and SAM's in [KSSS 89] we keep the last accessed path of the trees in main memory If orphaned entries occur from insertions or deletions, they are stored in main memory additionally to the path

In order to keep the performance comparison manageable, we have chosen the page size for data and directory pages to be 1024 bytes which is at the lower end of realistic page sizes Using smaller page sizes, we obtain similar performance results as for much larger file sizes From the chosen page size the maximum number of entries in directory pages is 56 According to our standardized testbed we have restricted the maximum number of entries in a data page to 50

As candidates of our performance comparison we selected the R-tree with quadratic split algorithm (abbre- viation qua Gut), Greene's variant of the R-tree (Greene) and our R*-tree where the parameters of the different structures are set to the best values as described in the previous sections Additionally, we tested the most popular R-tree implementation, the variant with the linear split algorithm (lin Gut) The popularity of the linear R-tree is due to the statement in the original paper [Gut84] that no essential performance gain resulted from the quadratic version vs the linear version For the linear R-tree we found m=20% (of M) to be the variant with the best performance

To compare the performance of the four structures we selected six data files containing about 100,000 2-dimensional rectangle Each rectangle is assumed to be in the unit cube $[0,1)^2$ In the following each data file is described by the distribution of the centers of the rectangles and by the tripel (n, μ_{area} , nv_{area}) Here n denotes the number of rectangles, μ_{area} is the mean value of the area of a rectangle and $nv_{area} = \sigma_{area} / \mu_{area}$ is the normalized variance where σ_{area} denotes the variance of the areas of the rectangles Obviously, the parameter nv_{area} increases independently of the distribution the more the areas of the rectangles differ from the mean value and the average overlap is simply obtained by $n^* \mu_{area}$

(F1) **"Uniform"**
The centers of the rectangles follow a 2-dimensional independent uniform distribution
(n = 100,000, μ_{area} = 0001, nv_{area} = 9505)

(F2) **"Cluster"**
The centers follow a distribution with 640 clusters, each cluster contains about 1600 objects
(n = 99,968, μ_{area} = 00002, nv_{area} = 1 538)

(F3) **"Parcel"**
First we decompose the unit square into 100,000 disjoint rectangles Then we expand the area of each rectangle by the factor 2 5
(n = 100,000, μ_{area} = 00002504, nv_{area} = 30 3458)

(F4) **"Real-data"**
These rectangles are the minimum bounding rectangles of elevation lines from real cartography data
(n = 120,576, μ_{area} = 0000926, nv_{area} = 1 504)

(F5) **"Gaussian"**
The centers follow a 2-dimensional independent Gaussian distribution
(n = 100,000, μ_{area} = 00008, nv_{area} = 89875)

(F6) **"Mixed-Uniform"**
The centers of the rectangles follow a 2-dimensional independent uniform distribution
First we take 99,000 small rectangles with μ_{area} = 0000101 Then we add 1,000 large rectangles with μ_{area} = 001 Finally these two data files are merged to one
(n = 100,000, μ_{area} = 00002, nv_{area} = 6 778)

For each of the files (F1) - (F6) we generated queries of the following three types

- *rectangle intersection query* Given a rectangle S, find all rectangles R in the file with $R \cap S \neq \emptyset$
- *point query* Given a point P, find all rectangles R in the file with $P \in R$
- *rectangle enclosure query* Given a rectangle S, find all rectangles R in the file with $R \supseteq S$

For each of these files (F1) - (F6) we performed 400 rectangle intersection queries where the ratio of the x-extension to the y-extension uniformly varies from 0 25 to 2 25 and the centers of the query rectangles themselves are uniformly distributed in the unit cube In the following, we consider four query files (Q1) - (Q4) of 100 rectangle intersection queries each The area of the query rectangles of each query file (Q1) - (Q4) varies from 1%, 0 1%, 0 01% to 0 001% relatively to the area of the data space For the rectangle enclosure query we consider two query files (Q5) and (Q6) where the corresponding rectangles are the same as in the query files (Q3) and (Q4), respectively Additionally, we analyzed a query file (Q7) of 1,000 point queries where the query points are uniformly distributed

For each query file (Q1) - (Q7) we measured the average number of disc accesses per query In the performance comparison we use the R*-tree as a measuring stick for the other access methods, i e we standardize the number of page accesses for the queries of the R*-tree to 100% Thus we can observe the performance of the R-tree variants relative to the 100% performance of the R*-tree

To analyze the performance for building up the different R-tree variants we measured the parameters insert and stor Here insert denotes the average number of disc accesses per

insertion and stor denotes the storage utilization after completely building up the files In the following table we present the results of our experiments depending on the different distributions (data files) For the R*-tree we also depict "# accesses", the average number of disk accesses per query

Uniform

	point	intersection 0.001	0.01	0.1	1.0	enclosure 0.001	0.01	stor	insert
lin Gut	225.8	212 6	207 7	183 0	144.5	224.7	248 1	64.1	7 43
qua. Gut	124.8	121.9	124.4	124.1	114.2	116 7	121.9	69 5	4.27
Greene	140 0	136 1	135 4	130 1	115 1	132.8	153.8	70.3	4.67
R*-tree	100 0	100 0	100 0	100 0	100 0	100 0	100 0	75.8	4.42
# accesses	5.26	6 04	7 63	13.29	53 42	4.85	3 66		

Cluster

	point	intersection 0.001	0.01	0.1	1.0	enclosure 0.001	0.01	stor	insert
lin Gut	250.9	231 0	219 7	176 6	136.9	247.8	249 4	61 7	6 13
qua Gut	166 1	152.7	160 7	139 1	120.4	155.4	182.9	66.9	4.97
Greene	159 9	151.8	152.2	144.3	116.9	151 6	153.2	69.2	4.32
R*-tree	100 0	100 0	100 0	100 0	100 0	100 0	100 0	72.2	3 77
# accesses	2 00	2.26	2 95	7 13	36 0	1.86	1 58		

Parcel

	point	intersection 0 001	0 01	0 1	1 0	enclosure 0 001	0 01	stor	insert
lin Gut	264 1	265 0	258 6	214.3	177.9	269 4	281.0	60.2	23 07
qua Gut	129.5	132.3	129.9	126 1	122 1	131 0	125 6	67 0	13.30
Greene	199.8	196.2	206 9	184 1	156.5	195.8	207.5	68.9	16.02
R*-tree	100 0	100.0	100 0	100 0	100 0	100 0	100 0	72.5	10 73
# accesses	5 67	6.26	7.36	13.29	36 76	5 42	4.96		

Real Data

	point	intersection 0.001	0.01	0.1	1.0	enclosure 0.001	0.01	stor	insert
lin Gut	245 6	246 7	220.8	181 6	143.8	268 1	284.1	62.9	7.30
qua Gut	147.3	153 1	143.3	132.5	116 4	158.8	160 1	68 1	5 08
Greene	147.8	144 0	146.5	130.2	115.9	155 1	169.8	69 6	5 05
R*-tree	100 0	100 0	100.0	100 0	100.0	100 0	100 0	70.5	4.22
# accesses	4 78	5.29	7.35	14 65	60.84	4.08	3 08		

Gaussian

	point	intersection 0.001	0.01	0.1	1.0	enclosure 0.001	0.01	stor	insert
lin Gut	171 1	165 6	168 1	150 1	143.8	171 1	180.2	63.8	19 12
qua Gut	116.2	108 0	116 0	117 6	119.2	106 4	106.8	68.8	14.0
Greene	123.2	118 7	131.2	122.9	114.2	120 7	130 6	69.9	11 41
R*-tree	100.0	100.0	100 0	100.0	100 0	100.0	100 0	73.8	9 15
# accesses	4.83	5.87	7 69	10.88	46 19	4.39	3.24		

Mixed Uniform

	point	intersection 0.001	0.01	0.1	1.0	enclosure 0.001	0.01	stor	insert
lin Gut	354.1	332.5	311 7	233 1	165.9	358 1	401 6	63 4	12 70
qua Gut	127 6	126.3	122.7	119 0	113 0	119 6	124.7	68 2	4 94
Greene	121 4	116 7	116 0	114.5	109.3	114 0	116.3	70 1	4.58
R*-tree	100.0	100 0	100 0	100 0	100 0	100 0	100 0	73 1	4 46
# accesses	4.87	5.51	7.27	13 76	52 06	4.44	3 69		

Additionally to the conventional queries like point query, intersection query and enclosure query we have considered the operation spatial join usually used in applications like map overlay We have defined the spatial join over two rectangle files as the set of all pairs of rectangles where the one rectangle from $file_1$ intersects the other rectangle from $file_2$

For the spatial join operation we performed the following experiments

(SJ1) $file_1$ "Parcel"-distribution with 1000 rectangles randomly selected from file (F3)
$file_2$ data file (F4)

(SJ2) $file_1$ "Parcel"-distribution with 7500 rectangles randomly selected from data file (F3)
$file_2$ 7,536 rectangles generated from elevation lines
(n = 7,536, μ_{area} = 00148, nv_{area} = 1 5)

(SJ3) $file_1$ "Parcel"-distribution with 20,000 rectangles randomly selected from data file (F3)
$file_2$ $file_1$

For these experiments we measured the number of disc accesses per operation The normalized results are presented in the following table

Spatial Join

	(SJ 1)	(SJ 2)	(SJ 3)
lin.Gut	296 6	229.2	257 8
qua.Gut	142 4	154 7	144 8
Greene	187.1	166.3	160 4
R*-tree	100.0	100 0	100.0

5.2 Interpretation of the Results

In table 1 for the parameters *stor* and *insert* we computed the unweighted average over all six distributions (data files) The parameter *spatial join* denotes the average over the three spatial join operations (SJ1) - (SJ3) For the average query performance we present the parameter *query average* which is averaged over all seven query files for each distribution and then averaged over all six distributions

	query average	spatial join	stor	insert
lin Gut	227 5	261 2	62 7	12 63
qua.Gut	130 0	147 3	68 1	7 76
Greene	142 3	171 3	69 7	7 67
R*-tree	100 0	100 0	73 0	6 13

Table 1 unweighted average over all distributions

The loss of information in the parameter *query average* is even less in table 2 where the parameter is displayed separately for each data file (F1) - (F6) as an average over all seven query files and in table 3 where the parameter query average is depicted separately for each query (Q1) - (Q7) as an average over all six data files

	gaussian	cluster	mix uni	parcel	real data	uniform
lin Gut	164 3	216 0	308 1	247.2	227.2	206 6
qua Gut	112 9	153 9	121 8	128 1	144.5	121 1
Greene	123 1	147 1	115 5	192 4	144.2	134 8
R*-tree	100 0	100 0	100 0	100 0	100 0	100 0

Table 2 unweighted average over all seven types of queries depending on the distribution

	point	intersection				enclosure		stor	insert
		0.001	0.01	0.1	1.0	0.001	0.01		
lin. Gut	251.9	242.2	231 1	189.8	152 1	256.5	274 1	62.7	12.63
qua. Gut	135.3	132.4	132.8	126.4	117 6	131.3	137.0	68.1	7.76
Greene	148 7	143.9	148 0	137.7	121.3	145.0	155.2	69 7	7 67
R*-tree	100 0	100 0	100.0	100 0	100 0	100 0	100.0	73.0	6.13

Table 3 unweighted average over all six distributions depending on the query type

First of all, the R*-tree clearly outperforms the R-tree variants in all experiments Moreover the most popular variant, the linear R-tree, performs essentially worse than all other R-trees The following remarks emphasize the superiority of the R*-tree in comparison to the R-trees

- The R*-tree is the most robust method which is underligned by the fact that for every query file and every data file less disk acesses are required than by any other variants To say it in other words, there is no experiment where the R*-tree is not the winner

- The gain in efficiency of the R*-tree for smaller query rectangles is higher than for larger query rectangles, because storage utilization gets more important for larger query rectangles This emphasizes the goodness of the order preservation of the R*-tree (i e rectangles close to each other are more likely stored together in one page)

- The maximum performance gain of the R*-tree taken over all query and data files is in comparison to the linear R-tree about 400% (i e it takes four times as long as the R*-tree !!), to Greene's R-tree about 200% and to the quadratic R-tree 180%

- As expected, the R*-tree has the best storage utilization

- Surprisingly in spite of using the concept of Forced Reinsert, the average insertion cost is not increased, but essentially decreased regarding the R-tree variants

- The average performance gain for the spatial join operation is higher than for the other queries The quadratic R-tree, Greene's R-tree and the linear R-tree require 147%, 171% and 261% of the disc accesses of the R*-tree, respectively, averaged over all spatial join operations

5.3 The R*-tree: an efficient point access method

An important requirement for a spatial access method is to handle both spatial objects and point objects efficiently Points can be considered as degenerated rectangles and in most applications rectangles are very small relatively to the data space If a SAM is also an efficient PAM, this would underlign the robustness of the SAM Moreover, in many applications it is desirable to support additionally to the bounding rectangle of an object at least an atomar key with one access method

Therefore we ran the different R-tree variants and our R*-tree against a benchmark proposed and used for point access methods The reader interested in the details of this benchmark is referred to [KSSS 89] In this paper, let us mention that the benchmark incorporates seven data files of highly correlated 2-dimensional points Each data file contains about 100,000 records For each data file we considered five query files each of them containing 20 queries The first query files contain range queries specified by square shaped rectangles of size 0 1%, 1% and 10% relatively to the data space The other two query files contain partial match queries where in the one only the x-value and in the other only the y-value is specified, respectively

Similar to the previous section, we measured the storage utilization (stor), the average insertion cost (insert) and the average query cost averaged over all query and data files The results are presented in table 4 where we included the 2-level grid file ([NHS84], [Him85]), a very popular point access method

	query average	stor	Insert
lin.Gut	233.1	64.1	7 34
qua.Gut	175.9	67.8	4.51
Greene	237.8	69.0	5.20
GRID	127.6	58.3	2.56
R*-tree	100 0	70.9	3.36

Table 4: unweighted average over all seven distributions

We were positively surprised by our results The performance gain of the R*-tree over the R-tree variants is considerably higher for points than for rectangles In particular Greene's R-tree is very inefficient for point data It requires even more accesses than the linear R-tree and 138% more than the R*-tree, whereas the quadratic R-tree requires 75% more disc accesses than the R*-tree Nevertheless, we had expected that PAMs like the 2-level grid file would perform better than the R*-tree However in the over all average the 2-level grid file performs essentially worse than the R*-tree for point data An advantage of the grid file is the low average insertion cost In that sense it might be more suitable in an insertion-intensive application Let us mention that the complexity of the algorithms of the R*-trees is rather low in comparison to highly tuned PAMs

6 Conclusions

The experimental comparison pointed out that the R*-tree proposed in this paper can efficiently be used as an access method in database systems organizing both, multidimensional points and spatial data As demonstrated in an extensive performance comparison with rectangle data, the R*-tree clearly outperforms Greene's R-tree, the quadratic R-tree and the popular linear R-tree in all experiments Moreover, for point data the gain in performance of the R*-tree over the other variants is increased Additionally, the R*-tree performs essentially better than the 2-level grid file for point data

The new concepts incorporated in the R*-tree are based on the reduction of the area, margin and overlap of the directory rectangles Since all three values are reduced, the R*-tree is very robust against ugly data distributions Furthermore, due to the fact of the concept of Forced Reinsert, splits can be prevented, the structure is reorganized dynamically and storage utilization is higher than for other R-tree variants The average insertion cost of the R*-tree is lower than for the well known R-trees Although the R*-tree outperforms its competitors, the cost for the implementation of the R*-tree is only slightly higher than for the other R-trees

In our future work, the we will investigate whether the fan out can be increased by prefixes or by using the grid approximation as proposed in [SK 90] Moreover, we are generalizing the R*-tree to handle polygons efficiently

References:

[Gre 89] D Greene 'An Implementation and Performance Analysis of Spatial Data Access Methods', Proc 5th Int Conf on Data Engineering, 606-615, 1989

[Gut 84] A Guttman 'R-trees a dynamic index structure for spatial searching', Proc ACM SIGMOD Int Conf on Management of Data, 47-57, 1984

[Hin 85] K Hinrichs 'The grid file system implementation and case studies for applications', Dissertation No 7734, Eidgenössische Technische Hochschule (ETH), Zuerich, 1985

[Knu 73] D Knuth 'The art of computer programming', Vol 3 sorting and searching, Addison-Wesley Publ Co , Reading, Mass , 1973

[KSSS 89] H P Kriegel, M Schiwietz, R Schneider, B Seeger 'Performance comparison of point and spatial access methods', Proc Symp on the Design and Implementation of Large Spatial Databases', Santa Barbara, 1989, Lecture Notes in Computer Science

[NHS 84] J Nievergelt, H Hinterberger, K C Sevcik 'The grid file an adaptable, symmetric multikey file structure', ACM Trans on Database Systems, Vol 9, 1, 38-71, 1984

[RL 85] N Roussopoulos, D Leifker 'Direct spatial search on pictorial databases using packed R-trees', Proc ACM SIGMOD Int Conf on Managment of Data, 17-31, 1985

[SK 88] B Seeger, H P Kriegel 'Design and implementation of spatial access methods', Proc 14th Int Conf on Very Large Databases, 360-371, 1988

[SK 90] B Seeger, H P Kriegel 'The design and implementation of the buddy tree', Computer Science Technical Report 3/90, University of Bremen, submitted for publication, 1990