Mathematical Problems of Computer Science 24, 2005, 133-143.

An Efficient and Robust Access Method for Indexing of Spatial Objects

Mihran S. Grigoryan

Institute for Informatics and Automation Problems of NAS of RA
e-mail mihran.grigoryan@buy.am

Abstract

There are considered the problems connected with the organization of probably faster and effective access to geometrical objects (points, lines, and polygons), coded in standard way in object-relational databases.

For maintenance of effective access the well-known method of so-called R-trees is often used; improved versions of this method which allow to raise the speed of access to objects of the mentioned specified type in comparison with usually applied variants of a method of R-trees are considered.

For achievement of the mentioned purpose there are entered into considerations R'- Trees, which represent certain updating of R-Trees, and R'- Trees allow to model methods of linear and quadratic R-Trees of Gutman, and the variant of R-Trees offered by Green.

So a number of advantages of R'-Trees in comparison with R-Trees are established.

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 [7] it is shown 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 [2]. The R-tree is based on the PAM B+-tree [9] using the over-lapping regions technique. 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 do not minimize the margin or the overlap of such minimum bounding rectangles? Why do not optimize storage utilization? Why do 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 [4].

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, rectangle intersection and rectangle enclosure query, the new R'-tree was analyzed 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 of 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 m that child node. A leaf node contains entries of the form (Old, Rectangle) where *Old* 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 (data object, 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 ($2 <= m <= M/2$). 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 [2].

We will show in this paper that the overlapping-regions-technique does not imply a 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.

1) 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.

2) The overlap between directory rectangles should be minimized. This also decreases the number of paths to be traversed.

3) 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, optimization 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 m 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.

4) 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 1 or 2 more freedom in choosing the shape is necessary. Thus rectangles will be less quadratic. With 1 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 1-3.

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 sub tree to accommodate 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 rectangle 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 [2].

```
Algorithm ChooseSubtree

Step 1) Set N to be the root
Step 2) If N is a leaf,
            return N
        else
            Choose the entry m 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
Step 3) Set N to be the childNode pointed to by the
        childPointer of the chosen entry an repeat from Step 2
```

Obviously, the method of optimization is to minimize the area covered by a directory rectangle. 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 obtained nearly the same retrieval performance for the linear as for the quadratic version We implemented the R-tree m both variants However m 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]

Step 1) Invoke PickSeeds to choose two entries to be the first entries of the groups
Step 2) Repeat
DistributeEntry();
until
all entries are distributed or one of the two groups has M-m+1 entries
Step 3) If entries remain, assign them to the other group
such that it has the minimum number m

Algorithm PickSeeds:
Step 1) For each pair of entries E_1 and E_2, compose a
rectangle R including E_1 rectangle and E_2 rectangle
Calculate d = area(R) - area(E_1 rectangle) - area(E_2 rectangle).
Step 2) Choose the pair with the largest d.

Algorithm DistributeEntry:
Step 1) Invoke PickNext() to choose the next entry to be assigned
Step 2) 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:
Step 1) For each entry E not yet m a group, calculate d_1 = the area increase
required in the covering rectangle of Group 1 to include E Rectangle
Step 2) Calculate d_2 analogously for Group 2
Step 3) 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 and 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 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 [1], 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 entry into two groups]

Step 1) Invoke ChooseAxis to determine the axis
perpendicular to which the split is to be performed
Step 2) Invoke Distribute
Algorithm ChooseAxis:
Step 1) Invoke PickSeeds (see p. 5) to find the two most
distant rectangles of the current node
Step 2) For each axis record the separation of the two seeds
Step 3) Normalize the separations by dividing them by the length of
the nodes enclosing rectangle along the appropriate axis
Step 4) Return the axis with the greatest normalized separation
Algorithm Distribute:
Step 1) Sort the entries by the low value of then rectangles
along the chosen axis.
Step 2) Assign the first (M+1) div 2 entries to one group, the
last (M+1) div 2 entries to the other
Step 3) 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 m 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, m some situations Greene's split method cannot find the "right" axis and thus a very bad split may result, Figure 2b depicts such a situation.

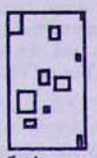

Figure 1a
Overfilled node

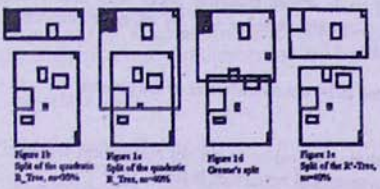

Figure 1b
Split of the quadratic R_Tree, m=50%

Figure 1c
Split of the quadratic R_Tree, m=40%

Figure 1d
Greene's split

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

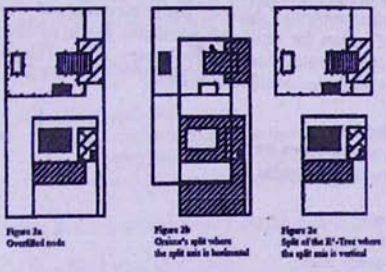

Figure 2a
Overfilled node

Figure 2b
Greene's split where the split axis is horizontal

Figure 2c
Split of the R*-Tree where the split axis is vertical

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 was defined as follows:

Let $E_1, E_2, \ldots, E_p$ be the entries m the current node.

Then overlap $(E_k) = \sum_{i=1, i \neq k}^{p}$ area(E_k Rectangle □ E_i Rectangle) , 1 □ k □ p

The versjon with the best retrieval performance is described in the following algorithm:

Algorithm ChooseSubtree:

Step 1) Set N to be the root
Step 2) 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
Step 3) Set N to be the childNode pointed to by the childPointer of the chosen entry and repeat from Step 2.

For choosing the best non-leaf node, alternative methods did not outperform Guttman's original algorithm. For the leaf nodes, minimizing the overlap was performed 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 then area
enlargement needed to include the new data rectangle

Let A be the group of the first p entries
From the entries m 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 Choose Sub-tree. 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 Choose Subtree optimization improves the retrieval performance particularly in the following situation: *Queries with small query rectangles on data files 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 then 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:

Step 1) Invoke Choose Split Axis to determine the axis,
perpendicular to which the split is performed
Step 2) Invoke Choose Split Index to determine the best
distribution into two groups along that axis
Step 3) Distribute the entries into two groups

Algorithm ChooseSplitAxis:

Step 1) 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
Step 2) Choose the axis with the minimum S as split axis

Algorithm ChooseSplitIndex:

Step 1) 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 parameters. However, even the following method did result m 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 1-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 performed 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 [2]. 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 data files. But this is a static situation and for nearly static data files the pack algorithm [6] 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 [2]. Except for the overflow treatment, it is the same as described originally by Guttman and therefore it is only sketched here:

Algorithm InsertData:

Step 1) Invoke Insert starting with the leaf level as a
parameter, to Insert a new data rectangle

Algorithm Insert:

Step 1) Invoke ChooseSubtree. With the level as a parameter, to find an
appropriate node N, in which to place the new entry E
Step 2) 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]

Step 3) 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
Step 4) Adjust all covering rectangles in the insertion path such that they
are minimum bounding boxes enclosing then children rectangles

Algorithm OverflowTreatment:

Step 1) If the level is not the root level and this is the first
call of OverflowTreatment m the given level
during the Insertion of one data rectangle, then
invoke Reinsert
else
invoke Split
end
Algorithm Reinsert:
Step 1) 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
Step 2) Sort the entries m decreasing order of their distances computed in Step 1
Step 3) Remove the first p entries from N and adjust the bounding rectangle of N
Step 4) In the sort, defined 111R 12, 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 experiments have shown that p = 30% of M for leaf nodes as well as for nonleaf 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 unproved
- 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.

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.

5. Performance Comparison

We ran the performance comparison on Dell servers under Windows Serever 2003 using C# implementations of the different R-tree variants and our R'-tree. Analogously to performance comparison of PAM's and SAM's in [4] keept 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.

As candidates of our performance comparison we selected the R-tree with quadratic split algorithm, Greene's variant of the R-tree 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.

To compare the performance of the four structures we selected six data files containing about 100,000 2-dimensional rectangles.

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 m comparison to the R-trees.

- The R'-tree is the most robust method which is underlined by the fact that for every query file and every data file less disk accesses 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 (1e 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% (1e It takes four tunes as long as the R'-tree If), 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 Jam 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 Jam Operations.

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.

References

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

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

[3] K Htnrlchs 'The grid file system implementation and case studies for applIcations', DIssertation No 7734, Eldgen6ssIsche TechnIsche Hochschule (ETH), Zuerich. 1985

[4] H P Krregel, M Schlwletz, 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.

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

[6] N Roussopoulos, D Leifker 'Direct spatial search on pictorial databases using packed R-trees', Proc ACM SIGMOD Int. Conf on Management of Data, 17-31, 1985
[7] B Seeger. H P Kriegel 'Design and implementation of spatial access methods', Proc 14th Int Conf on Very Large Databases, 360-371, 1988
[8] 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
[9] D Knuth 'The art of computer programming', Vol. 3 sorting and searching, Addison-Wesley Publ. Co., Reading, Mass, 1973

Տարածական օբյեկտների ինդեքսավորման մի արդյունավետ և ռոբաստ եղանակ

Մ. Գ-ի գ ո ր յ ա ն

Ամփոփում

Առարկայապես-կողմնորոշված տվյալների բազաներում նկատվում են խնդիրներ՝ կապված երկրաչափական օբյեկտներին (կետերին, գծերին, բազմանկյուններին), կողավորված ստանդարտ մեթոդով, նմարավորին չափ ավելի արագ եվ էֆեկտիվ դիմման կազմակերպման հարցում:

Էֆեկտիվ դիմումը ապահովելու համար հաճախ օգտագործվում է այսպես կոչված R-ծառների մեթոդը. քննարկվում են այդ մեթոդի կատարելագործված տարբերակները, որոնք, ի տարբերություն R-ծառերի մեթոդի սովորաբար գործածվող տարբերակին, թույլ են տալիս բարձրացնել նշված տիպի երկրաչափական առարկաների դիմելու արագությունը:

Նշված նպատակին հասնելու համար ներմուծվում է R'-ծառային տարբերակը, որն իրենից ներկայացնում է R-ծառների որոշակի մոդիֆիկացիա, ընդ որում R'-ծառները թույլ են տալիս ձեվափոխել Գուտմանի գծային եվ սխեմատիկ (տարածական) R-ծառների մեթոդները, ինչպես նաեվ Գրինի առաջարկած R-ծառների տարբերակը:

Այսպիսով հաստատվում են R'-ծառների մի շարք առավելություններ R-ծառների նկատմամբ: