Prof. A. Kemper, Ph. D
University of Passau

Proseminar:
Algorithms and Datastructures for
Database Systems
SS 2003

R-Tree

Sebastian Käbisch
18 June 2003
Moderation by Thomas Bernreiter

Contents

1 Introduction

One of the huge demands in geo-data applications is to response very quickly to spatial inquiry. Spatial data objects often cover areas in multi-dimensional spaces. The inquiry of the multi-dimension prevents from using classical indexing structures, for instance the B-Tree[Kemp01]. The reason is that database use one-dimensional indexing structures. However, in modern information processing like CAD (Computer Aided Design), cartography and multimedia applications use multi-dimensional data objects which means that the objects have more attributes. Thus, the database system needs an efficient multi-dimensional index structure.

A number of structures has been proposed for handling multi-dimensional point data. Antoine Guttman was one of the first persons to propose them. In 1984, Guttman published a book[Gutt84] in which he presented a data structure called R-Tree (Rectangle Tree) that represents data objects by intervals in several dimensions.

The following paper is concerned with the R-Tree. Firstly, it will outline the structure of a tree. In the following, the algorithms for searching, inserting and deleting will be introduced. Finally, some results of R-Tree index performance tests, some problems and developments of R-Trees will be presented.

On the basis of spatial data the examples in the paper are just two-dimensional. This is useful for clarification. However, it should not fall into oblivion that a R-Tree can represent spatial data in several dimensions.

2 R-Tree Index Structure

An R-Tree is a height-balanced tree similar to a B-Tree. Leaf nodes contain pointers to data objects. The index is completely dynamic. Structure is designed in such a way that a spatial search requires visiting only a small number of nodes.

The spatial data is comprised by a MBR (Minimal Bounding Rectangle) which become formated and comprised from a MBR again. This structure continues up to the root. Eventually the root comprise a MBR over all objects. The Figure 1 shows an example of an R-Tree.

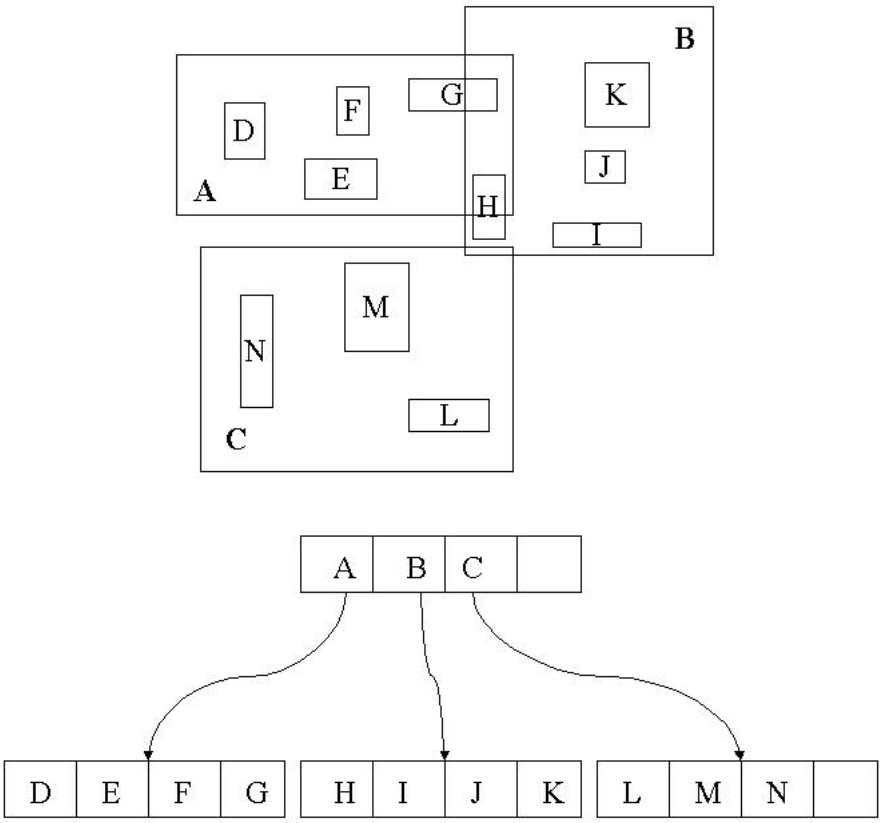

Figure 1: Structure of a simple R-Tree

A, B and C are the root nodes. A, for instance, covers child nodes D, E, F and G, and comprises them with a minimal bounding rectangle.

An R-Tree satisfies the following properties:

1. Every leaf node contains between m and M index records unless it is the root Thus, the root can have less entries than m
2. For each index record in a leaf node, I is the smallest rectangle that spatially contains the n-dimensional data object represented by the indicated tuple
3. Every non-leaf node has between m and M children unless it is the root
4. For each entry in a non-leaf node, i is the smallest rectangle that spatially contains the rectangles in the child node
5. The root node has at least two children unless it is a leaf
6. All leaves appear on the same level. That means the tree is balanced

2.1 Structure of a leaf-node

Leaf nodes in an R-Tree contain index record entries of the form

$$(I, tuple - identifier)$$

where *tuple-identifier* refers to a tuple in the database and I is an n-dimensional rectangle which is the bounding box of the spatial object indexed.

$$I = (I_0, I_1, ..., I_{n-1})$$

Here n is the number of dimensions and I_i is a closed bounded interval $[a, b]$ describing the extend of the object along dimension i. Following figure represent I as bounding box which contains two records (circle).

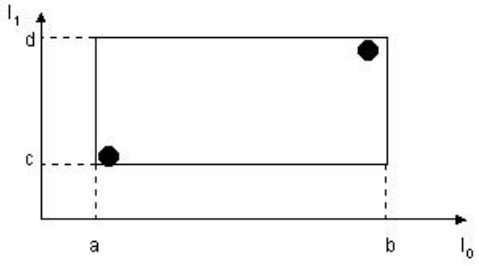

Figure 2: Representation of I with $n = 2$

2.2 Structure of a non-leaf node

The nodes which are no leafs contain entries of the form

$$(I, child - pointer)$$

where $child-pointer$ is the address of a lower node in the R-Tree and I covers all rectangles in the child node's entries.

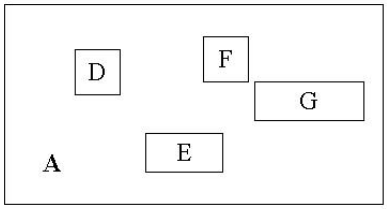

Figure 3: Root A covers child node's entries D, E, F and G (compare figure 1)

2.3 Variable m and M

M is the maximum of entries which is usally given and m is the minimum of entries in one node.

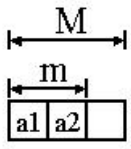

Figure 4: Representation of M and m

The minimum number of entries in a node is dependent on M with $\frac{M}{2} \geq m$. The maximum number of nodes is $\lceil \frac{N}{m} \rceil + \lceil \frac{N}{m^2} \rceil + 1$. Here N stands for the number of index records of the R-Tree. m is jointly responsible for the height of an R-Tree and the speed of the algorithm. The choice of M depends on the hardware, especially on hard disk properties such as capacity and sector size. If nodes have more than 3 or 4 entries, the tree is very wide, and almost all the space is used for leaf nodes containing index records.

2.4 Overflow and underflow

If m has a small value, it doesn't come so quickly to an overflow or an underflow so that the tree structure does not have to be reorganised. In the following example of deleting it comes to an underflow because of deleting $a3$. Here $M = 5$ and $m = 3$.

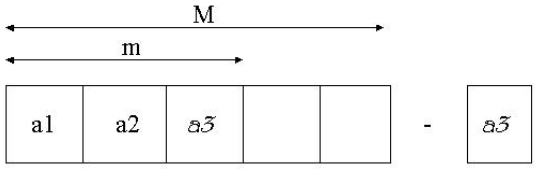

Figure 5: Underflow

There are now less then $m = 3$ entries in the node. Thus the tree has to be reorganised.

The other example explains the overflow through inserting.

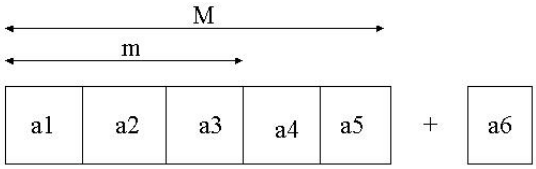

Figure 6: Overflow

Adding 1 to $M = 5$ entries thus result in reorganisation.

3 Algorithms

Here the algorithms are represented as pseudo-code. In the pseudo-code the rectangle parts of an index entry E are denoted by $E.I$ and the $tuple-identifier$ or $child-pointer$ part is denoted by $E.p$. The paper will show an example of every algorithm. It will always use the same tree with the values $M = 3$ and $m = 1$. In *insertion* and *deletion* is used the $SplitNode$ algorithm which is explained later in this section (3.4).

3.1 Searching

The search algorithm is similar to that of the B-Tree. It returns all qualifying records which the search rectangle overlaps. The algorithm descends the tree from the root. In the same time the algorithm checks the rectangle overlapping in the node with the searched rectangle. If the test is positive, the search just descends to the found overlapping nodes. This procedure is repeated until the leaf node. If the entries of the leaf node overlapps the searched rectangle then return this entries as a qualifying record.

Search pseudo-code:

T is root node of an R-Tree, find all index records whose rectangles overlap a search rectangle S.

S1 [Search Subtree] If T no leaf then check each entry E, weather $E.I$ overlaps S. For all overlapping entries, start **Search** on the subtree whose root node is pointed to by $E.p$.

S2 [Search leaf node] If T is a leaf, then check each entry E wheather $E.I$ overlaps S. If so, E is a suitable entry.

3.1.1 Example

In the Figure 7 there is a filled rectangle which is the search rectangle.

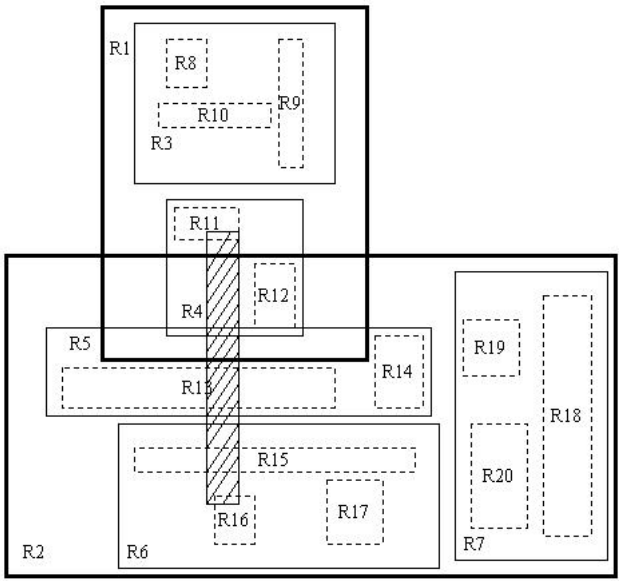

Figure 7: Searching

The algorithm is looking for qualifying records in the filled area. In Figure 8 one can see the way which was chosen by the algorithm:

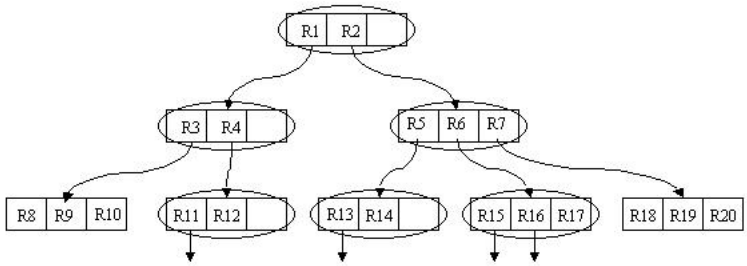

Figure 8: Searching

The filled rectangle overlaps the root entries $R1$ and $R2$, so the algorithm checks these entries. In $R1$ there is just $R4$ which overlaps the filled rectangle. Its entries are also checked. The algorithm arrives at the leaf node level. The entries of the leaf node are checked for qualifying records. $R11$ is the only one and so a first search result. In $R2$ there are two rectangle overlaps with the filled rectangle: $R5$ and $R6$. Both of them are checked and the algorithm recognises

that in the leaf node level the entries $R13$, $R15$ and $R16$ overlap with the search rectangle. Finally, the search result is $R11$, $R13$, $R15$ and $R16$.

3.2 Insertion

Inserting index records for new data is similar to insertion into a B-Tree. New data is added to the leaves, nodes that overflow are split, and splits are propagated up the tree. The *insertion* algorithm is more complex than the *searching* algorithm because inserting needs some help methods.

Insert pseudo-code:

New entry E will be inserted into a given R-Tree.

I1 [Find position for new record] Start **ChooseLeaf** to select a leaf node L in which to place E

I2 [Add record to leaf node] If node L has enough space for a new entry then add E. Else start **SplitNode** to obtain L and LL containing E and all the old entries of L.

I3 [Propagate changes upward] Start **AdjustTree** on node L and if a split was performed then also passing LL.

I4 [Grow tree taller] If node split propagation caused the root to split, create a new root whose children are the two resulting nodes.

ChooseLeaf pseudo-code:

ChooseLeaf selects a leaf node to place a new entry E.

CL1 [Initialize] Set N to be the root node.

CL2 [Leaf check] If N is a leaf then return N.

CL3 [Choose subtree] If N is not a leaf node then let F be the entry in N whose MBR $F.I$ needs least enlargement to include $E.I$. When there are more qualify entries in N, the entry with the rectangle of the smallest area is chosen.

CL4 [Descend until a leaf is reached] N is set to the child node F which is pointed to by $F.p$ and repeat from $CL2$

AdjustTree pseudo-code:

Leaf node L is upwarded to the root while adjusting covering rectangles. If necessary it comes to propagating node splits.

AT1 [Initalize] N is equal L.

AT2 [Check if done] stop if N is the root

AT3 [Adjust covering MBR in parent entry] P is the parent node of N and E_N the entry of N in P. $E_N.I$ is adjusted so that all rectangles in N are tightly enclosed.

AT4 [Propagate node split upward] If N has a partner NN which was split previously then create a new entry with $E_{NN}.p$ pointing to NN and $E_{NN}.I$ enclosing all rectangles in NN. If there is room in P then add E_{NN}. Otherwise start **SplitNode** to get P and PP which include E_{NN} and all old entries of P.

AT5 [Move up to next level] N is equal L and if a split occurred then NN is equal PP. Repeat from $AT2$

3.2.1 Example

A rectangle $R21$ is inserted (filled rectangle in Figure 9).

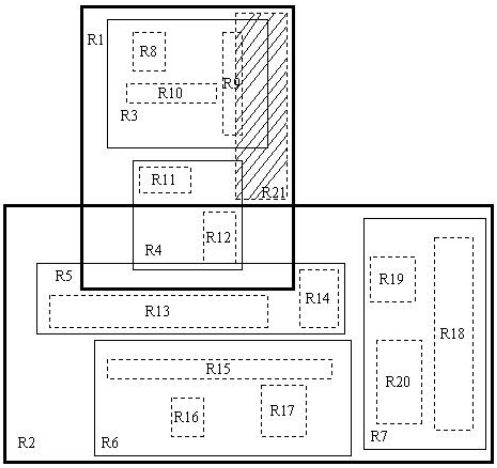

Figure 9: Insert a new rectangle

To find the best position for the new rectangle the algorithm starts with $ChooseLeaf$. The following figure shows the way of $ChooseLeaf$.

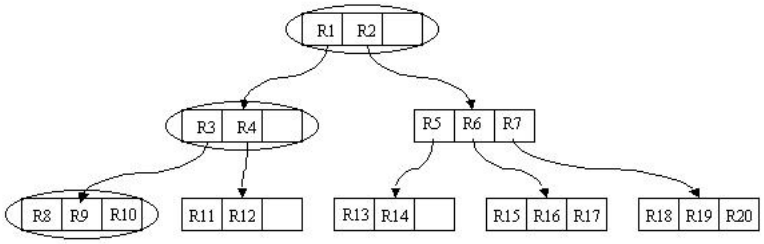

Figure 10: Way of *ChooseLeaf*

The first step is clear because $R21$ is in $R1$. Next *ChooseLeaf* chooses $R3$ because this rectangle needs less enlargement than $R4$. At the last step the algorithm finds the leaf node, however, all entries are full. Thus, it comes to a node split which the following figure shows.

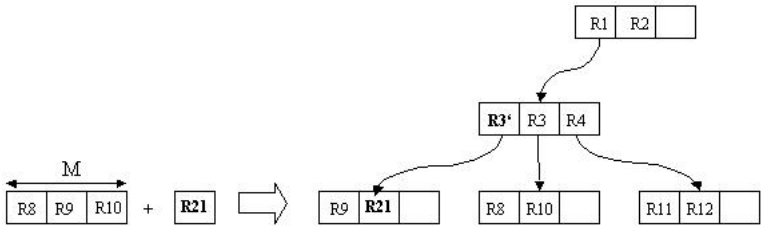

Figure 11: Splitting of a node

SplitNode tries to minimize rectangles as much as possible. That is the reason why the algorithm puts $R21$ and $R9$ in rectangle $R3$. $R8$ and $R10$ are put in the new parent rectangle $R3'$. $R3'$ is conveyed to *AdjustTree* where is it propagated upward. Since there is enough room to include $R3'$, it's not necessary to split this node again. $R3$ must be adjusted as well because it only points to $R9$ and to the new rectangle $R21$. At last, root node $R1$ is also adjusted because it includes a new entry $R3'$. So the structure of the tree is saved. The Insertion is now finished and the following figure shows the new included rectangle.

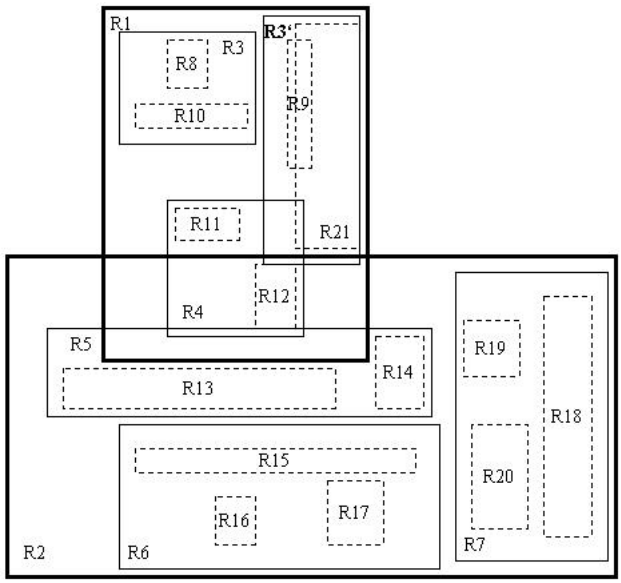

Figure 12: After insertion

3.3 Deletion

Deletion in R-Trees is different from deletion in B-Tree. The following pseudo-code will show it. Like insertion, the deletion needs help methods.

Delete pseudo-code:

Delete entry E from an R-Tree.

D1 [Find node containing record] Start **FindLeaf** to find the leaf node L containing E. If search unsuccessful then terminate.

D2 [Delete record] Remove E from L.

D3 [Propagate record] Start **CondenseTree** on L.

D4 [Shorten tree] If the root node has only one child after adjusting then make the child the new root.

FindLeaf pseudo-code:

Root node is T, the leaf node containing the index entry E is to find.

FL1 [Search subtree] If T is not a leaf, then check each entry F in T to determine when $F.I$ overlaps $E.I$. For all these entries **FindLeaf** starts on the subtree

whose root is pointed to by $F.p$ until E is found or each entry has been checked.

FL2 [Search leaf node for record] If T is a leaf, then check each entry to see when it matches E. If E is found, then return T.

CondenseTree pseudo-code:

Given is a leaf node L from which an entry has been deleted. If L has too few entries then eliminate it from the tree. After that, the remaining entries in L are reinserted in the tree. This procedure is repeated until the root. Also adjust all covering rectangles on the path to the root, making them smaller, if possible.

CT1 [Initalize] N is equal L. Initialize a list Q which consists of eliminate nodes as empty.

CT2 [Find parent entry] If N is the root, then go to $CT6$. Else P is the parent node of N, and E_N the entry of N in P.

CT3 [Eliminate underflow node] If N has fewer than m entries, then eliminate E_N from P and add N to list Q.

CT4 [Adjust covering rectangle] If N has not been deleted, then adjust $E_N.I$ to tightly contain all entries in N.

CT5 [Move up one level in tree] N is equal P and repeat from $CT2$.

CT6 [Re-insert orphaned entries] Every entry in Q is inserted. Leaf nodes are inserted like in $Insertion$. However, entries from higher-level nodes must be placed higher in the tree, so that leaves of their dependent subtrees will be on the same level as leaves of the main tree.

In the $CondenseTree$ lies the difference to the B-Tree. Firstly, if a node has an underflow, it is eliminated and inserted again. In a B-Tree, however, the node is fused with an other node. Secondly, the R-Tree is more efficient: Implementation of *deletion* is easier because $Insertion$ routine can be used. Through the deletion and reinsertion the spatial structure of the tree is incrementally refined.

3.3.1 Example

In the following example nothing is changed except that $m = 2$ and $M = 4$. $R11$ and $R12$ are visible records for clarification (Figure 13). Record c is deleted. At first, the *delete* algorithm starts $FindLeaf$ to get the position of c.

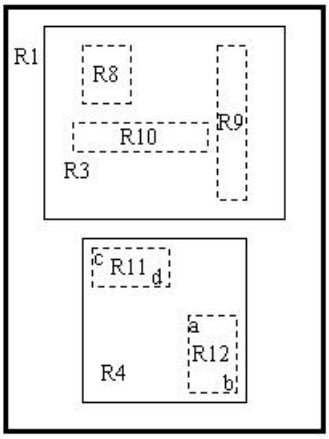

Figure 13: Visible records a, b, c and d

$R11$ is returned as result because it includes c. After that c is removed from $R11$. Now $CondenseTree$ is started. Figure 14 shows the first procedures of this algorithm.

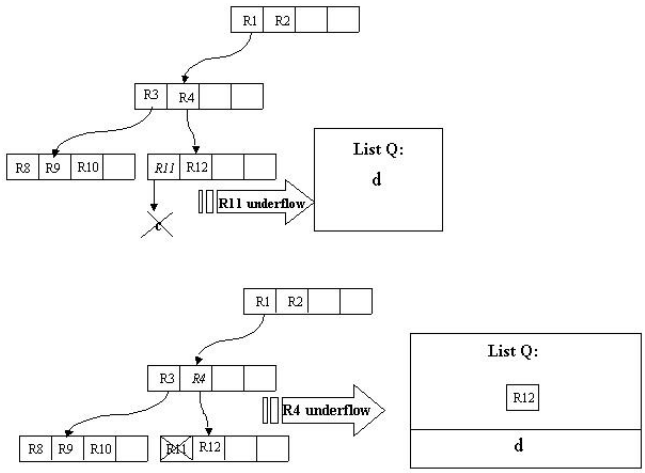

Figure 14: Set Q

With the new value $m = 2$, $R11$ has an underflow. It is eliminated from the tree but the last entry d of $R11$ is saved in list Q. Now $R4$ has an underflow. The entry $R11$ of $R4$ are also set in the list Q, and $R4$ is eliminated.

Through the deletion of $R4$, $R1$ has an underflow as well and thus $R1$ is eliminated (not represented in the Figure 14). Its entry $R3$ is saved in the list. Next, all entries of the list Q are reinserted in the tree (Figure 15).

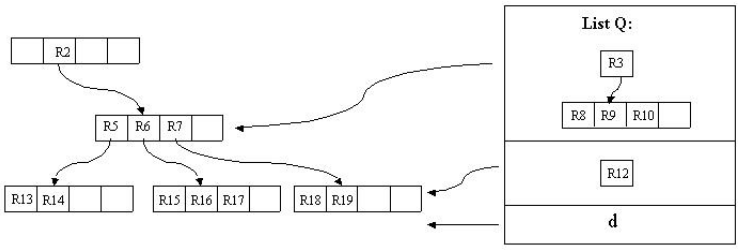

Figure 15: Reinsert

Firstly, the node $R3$ has to be placed in the same level again where it was before having been set in Q. After that leaf node $R12$ is reinserted in $R5$ because $R5$ nearest rectangle which has to enlarge least. Finally, record b is inserted in $R12$ because it is the nearest rectangle which hast to enlarge least. $CondenseTree$ is finished. The root node has only one child and thus the child is the new root. The following Figure shows the new structure of the tree after deletion.

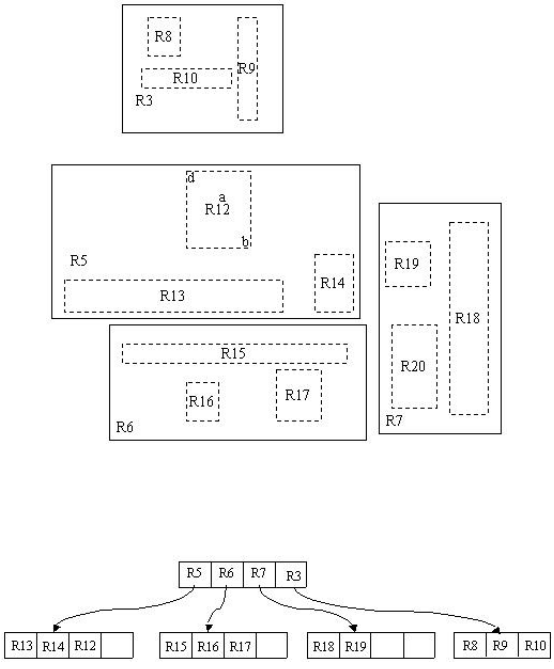

Figure 16: After deleting

3.4 SplitNode

In the case of adding a new entry to a full node containing M entries, it is necessary to divide the collection of $M + 1$ entries between two nodes. *Insertion* and *Deletion* have to use this method to save the tree structure. The division should be done in a way that makes it as unlikely as possible that both new nodes will need to be checked on subsequent searches. The total area of the two covering rectangles after a split should be minimized. Following figure shows a 'good' split and a 'bad' split.

For *SplitNode* there are three versions of algorithms which each have differences in quality and complexity. The paper will just introduce two of them because the *ExhaustiveAlgorithm* is normally not used. It is the best split algorithm in quality because it finds the best way to minimize the area of all rectangles of the R-Tree. The cost, however, would be 2^{M-1} and so the algorithm would be too slow with a large node size.

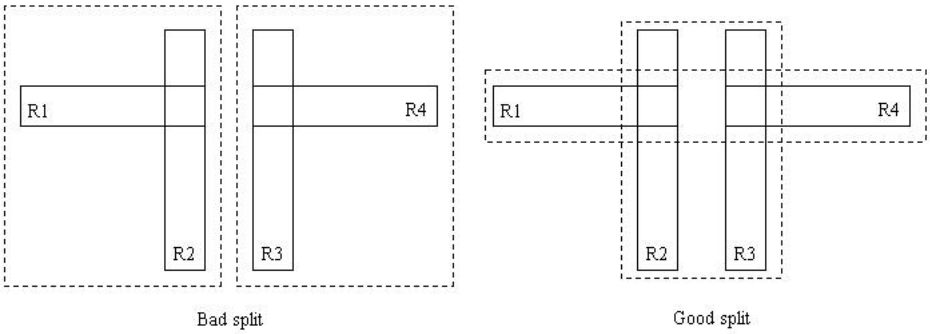

Figure 17: Kind of splits

3.4.1 Quadratic-Cost

This algorithm tries to find a small-area split, however, it is not guaranteed that it finds one with the smallest area possible. $Quadratic-Cost$ chooses two of the $M+1$ entries which use most of the area and puts them in new nodes. Concerning the remaining entries that entry is selected which needs the largest area if it is inserted in one of the two nodes. The algorithm then puts the selected entry in that node where less enlargement is needed. The procedure is repeated until all nodes are divided or one node has less then m entries. The cost of *Quadratic-Cost* is M^2.

Quadratic Split pseudo-code:

Divide a set of $M + 1$ index entries into two groups.

QS1 [Pick first entry for each group] start **PickSeeds** to find two entries to be the first elements of the groups. Assign each to a group.

QS2 [Check if done] If all entries have been assigned, then break up. If a group has too few entries that all the rest must be assigned to it in order for it to have the minimum number m, then assign them and break up.

QS3 [Select entry to assign] start **PickNext** to choose the next entry to assign. This is put in the group whose area has to be least enlarged. If the algorithm enlarges both groups with the same size, then add to the group with smaller area, then to the one with fewer entries, then to either. Repeat from $QS2$.

PickSeeds pseudo-code:

Choose two entries to be the first elements of the group.

PS1 [Calculate inefficiency of grouping entries together] For all pairs of entries E_1 and E_2 a rectangle J is created which includes $E_1.I$ and $E_2.I$. Calculate

$$d = area(J) - area(E_1.I) - area(E_2.I)$$

PS2 [Choose the most wasterful pair] Choose the pair with the largest d.

PickNext pseudo-code:

Choose one remaining entry for classification in group.

PN1 [Determine cost of putting each entry in each group] For each entry E which is not in a group yet, d_1 is calculated. d_1 is the area-increase required in the covering rectangle of group 1 to include $E.I$ and also d_2 for group 2.

PN2 [Find entry with greatest preference for one group] Select any entry with the maximum difference between d_1 and d_2.

3.4.2 A Linear-Cost Algorithm

This algorithm is quite similar to $Quadratic - Cost$. The difference is just in $PickSeeds$, because of this changes the cost is linear in M and in the number of dimension.

LinearPickSeeds pseudo-code:

Choose two entries to be the first elements of the groups

LPS1 [Find extreme rectangles along all dimensions] In each dimension the entry is found whose rectangle has the highest low side and the one with lowest high side. Save the separation.

LPS2 [Adjust for shape of the rectangle cluster] Normalize the seperations by dividing by the width of the entire set along the corresponding dimension.

LPS3 [Select the most extreme pair] Choose the pair with the greatest normalized separation along any dimension.

4 Performance Tests

Antoine Guttman also presents in his book different tests with the algorithms. This paper will present some results in order to get an idea of costs.

Guttman tried to find out the best performance and the best value for M and m. For this test 5 pages sizes were used (Figure 18). The minimum number of

Bytes per Page	Max Entries per Page (M)
126	6
256	12
512	25
1024	50
2048	102

Figure 18: Pages sizes

entries in a node were $\frac{M}{2}$, $\frac{M}{3}$ and 2. All tests used two-dimensional data. The implemention was in C under $Unix$ on a Vax 11/780 computer.

Each test began to run the program by reading in geometry data (layout data from the $RISC - II$ computer chip) with 1057 rectangles from files and inserting in an empty tree. After that, the program called the function $Search$ and searched rectangles made up by using random numbers. Finally, the test read the input file again and called the function $Delete$ to remove the index record. Following short cuts are used in the diagramms: E (Exhaustic algorithm), Q (Quadratic algorithm) and L (Linear algorithm).

4.1 Results of inserting records

The first diagramm shows the cost in CPU time for inserting the last 10% of the records as a function of page size.

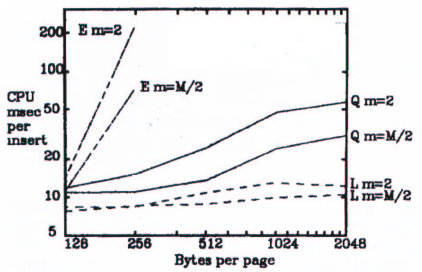

Figure 19: CPU cost of inserting records

The *Exhaustive* algorithm needs a lot of time with already less pages. The linear algorithm is fastest, as expected. With more bytes per pages the CPU cost doesn't increase so much.

4.2 Results of searching

The program called the function 100 times. Guttman examined two kinds of searching. One time the touched pages per qualifying record and the other time the CPU cost.

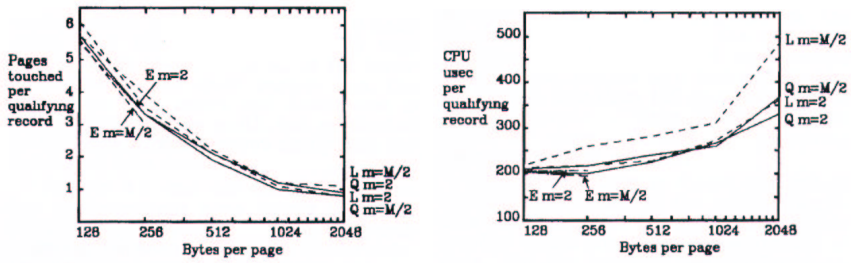

Figure 20: Search performance

The diagrams show almost the same result with the different algorithm. The reason is that *Searching* doesn't use *SplidNote*. However, the results show that the *Exhaustive* algorithm which produces a better index structure has the best values.

4.3 Results of deleting records

At last, the programm removed the index record for every tenth data item (Figure 21).

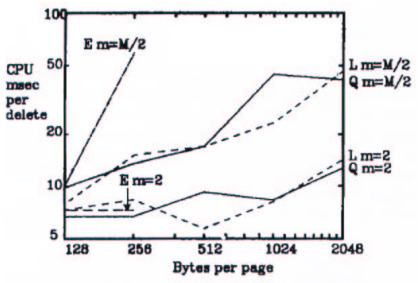

Figure 21: CPU cost of deleting records

The result was strongly affected by the minimum node fill requirement. If the value of m is small, the nodes often become an $underflow$. Thus, their entries must be reinserted and reinsertion sometimes causes nodes to split because of an $overflow$.

4.4 More performance tests

Antione Guttman presents a second series of tests measuring of R-Tree performance as a function of the amount of data in the index. The same sequence of test operations as before was run on samples containing 1057, 2238, 3295 and 4559 rectangles. Two combinations of split algorithm and node fill requirement were chosen. The *linear* with $m = 2$ and the *quadratic* with $m = \frac{M}{3}$. The page size is 1024 bytes, thus $M = 50$.
The following figure shows the CPU cost of inserts and deletes.

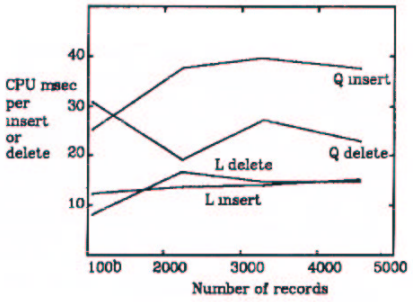

Figure 22: CPU cost of inserts and deletes

The *quadratic* algorithm is nearly constant from 2500 records through insertion except where the tree increases in height. The *linear* algorithm doesn't use so much cost through insertion and so there is no jump in the curve. Deletion with the *quadratic* configuration produced only 1 to 6 node splits. That's the reason why the curve is very rough. In the *linear* configuration there are no node splits and so the curve shows only a small jump. The test shows that the cost is independent of tree width but is affected by tree height, which grows slowly with the number of data items.
The last diagramms show the fact that almost all space in an R-Tree index is used for leaf nodes.

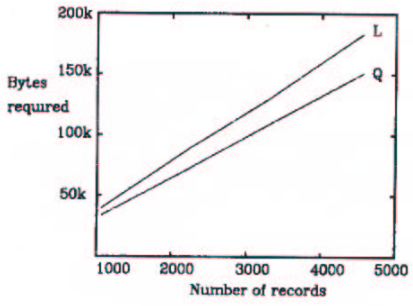

Figure 23: Space required for R-Tree

For the *linear* configuration the total space occupied by the R-Tree about 40 bytes per data item, compared to 20 bytes per item for the index records alone. The *quadratic* configuration was 33 bytes per item.

5 Problems

Some problems arise from the realisation of an R-Tree [Saak01]. Specially *searching* and *inserting* show some negative effects:

1. [Searching] An unfavourable index record can be overlapped by other regions, however, it's saved in just one region. For instance, in the search for a point data entry. The *search* algorithm would check some subtrees in vain. The more dimension there are, the more overlaps exists and the more checks are done in vain. Thus the R-Tree has an efficiency problem if there is an exact search. However, an exact search is specially necessary for *deletion* and *insertion*.

2. [Inserting] Sometimes a region has to enlarge in order to include a new rectangle. This can be propagated upward. Because of that, the risk of overlaps arise again.

When handling with small and favourable records, these problems are not very significant. However, if there a lot of unfavourable and multi-dimension records the efficiency is quite impaired.

6 Developments

In the course of time the R-Tree was improved, specially structure. Additionally, there was specialization for particular application. Here are some variants of the R-Tree.

R$^+$-Tree [Sell87]

This tree tries to minimized the overlapping of regions. Here, objects are saved disjunctive. The *search* algorithm is faster but the tree structure is more complicated.

R*-Tree [Beck90]

The $SplitNode$ algorithm takes volume and the extend of overlapping into consideration. All other properties are similar to the R-Tree.

X-Tree [Berc96]

It includes a $Split - History$ and a function for node enlargement. This prevents overlapping of regions.

7 Summary

In our modern application world the using of spatial index structure is unimaginable. To handle spatial data efficiently a database system needs an special index mechanism. This paper introduced the first kind of such an mechanism, Guttman's R-Tree. This tree is a dynamic index structure.

The structure is quite simple. An R-Tree is a height-balanced tree. It include leaf nodes which contain index record entries and include non-leaf nodes which contain the address of the children nodes. The variable M decides the maximum number of entries and the variable m the minimum number of entries in a node.

The algorithms are quite similar to ones of the B-Tree except for the *delete* algorithm. *Deletion* eliminates a node which has a underflow and insert it into the R-Tree again. The *SplitNode* algorithm is an important part of $Insertion$ and *Deletion*. Three different implementation decide which quality the tree structure has and how the cost is.

Even though the R-Tree has some efficiency problems if there are a lot of unfavourable and multi-dimension records, it's still was a great achievement and opened the door to handling spatial data indexes. In the course of time many new variants of R-Tree were developped to improve the efficiency and thus to improve complex applications.

8 Bibliographie

[Kemp01] A. Kemper/A. Eickler: Datenbanksysteme, Eine Einfuehrung, 4. Auflage, S. 207-211, 2001

[Gutt84] A. Guttman: R-Trees: A dynamic index structure for spatial searching, Proceedings of the 1984 ACM SIGMOUND Conference, S. 47.-57, Boston, MA, 1984

[Saak01] Gunter Saake, Andreas Heuer: Datenbanken: Implementierungstechniken, 1. Auflage, S. 257 - 259, 2001.

[Sell87] T. Sellis, N. Roussopoulos and C. Faloutsos: The R^+-Tree: A dynamic index for multidimensional objects, Proceedings of the 13th VLDB Conference, p.507-518, UK, 1987

[Beck90] N.Beckmann, H. P. Kriegel and B. Seeger: The R*-Tree: An efficient and robust method for points and rectangles, Proceedins of the 1990 ACM SIGMOND Conference, p.322-331, Atlantic City, NJ, 1990

[Berc96] S. Berchtold, D.A. Keim, H.-P. Kriegel: The X-Tree: An index structure for high-dimension data, Proceedings of the 22nd International Conferene on Very Large Databases, Bombay, India, 1996