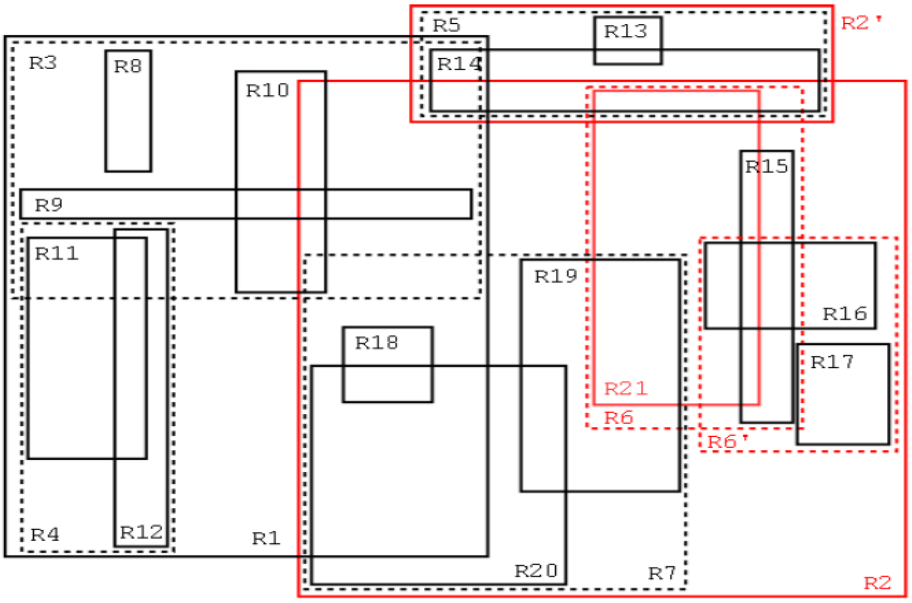
ALGORITHMS AND DATA STRUCTURES

FOR DATABASE SYSTEMS

R-TREES



JÜRGEN TREML

2005-06-08

Table of Contents

1 Introduction

With the speed of computers growing rapidly over the last 20 years, they have become indispensable in the field of geo-analysis, cartography, images processing and many more subjects that have one thing in common: the processing of spatial data. Finally with this being not only quite important but even the base of certain applications such as computer aided design or virtual reality, something has shown up clearly. Existing index structures for one or two dimensional data are not sufficient nor in any way efficient for saving and processing spatial, multi-dimensional data.

Among a huge amount of indexing structures there are well-developed, efficient forms for saving and accessing (i.e. searching, inserting, deleting) one dimensional data of which the binary tree is probably the most common and well-known one. But as soon as it comes to data consisting of more than one dimension these structures are no longer suitable. Although there are quite a few structures for storing spatial data, there is hardly that is to spatial data what the binary tree is to one-dimensional data. K-D-B trees for example, only work for point data [6] whereas k-d trees [1] as well as quad trees [2] do not take paging of memory into consideration and thus show up some disadvantages in the matter of efficiency. Cell methods [4] in contrast will reach there limits when it comes to extremely dynamic structures with changing boundaries. No need to emphasize that this makes up a certain problem when dealing with a vast amount of geographical data in cartography or highly dynamic structures in the field of virtual reality and computer aided design.

So this is why Antonin Guttman in 1984 finally came up with a completely new structure for indexing and processing spatial data, similar to and generally based on B-trees. This so called R-tree, standing for region tree, is what we will now examine in a little more detail.

2 R-tree Index Structure

2.1 General Structure of an R-tree

As already mentioned, R-trees are in one or another way quite similar to B-trees. Like those, R-trees contain all their data or reference to data in their leaf-nodes. Furthermore, they have features of a sort of binary tree known as AVL-tree, meaning that R-trees are height balanced. That is, the distance to the root node is the same for all leaf nodes. This is also the reason for the benefits R-trees have in the matter of dynamic indexing. In detail this means that all kind of actions like inserting, searching and deleting can be mixed without having to reorganize and restructure the spatial tree periodically in order to maintain performance.

The data itself that is to be indexed in the R-tree is represented by its minimum bounding rectangle, referred as MBR throughout the rest of the text. Leaf-nodes contain the MBR of the data objects they refer to, whereas non-leaf nodes are set up with the MBR of all there child's MBRs. Last but not least the root contains the MBR of all objects in the tree. The following figure is to illustrate this situation for a simple two-dimensional structure.

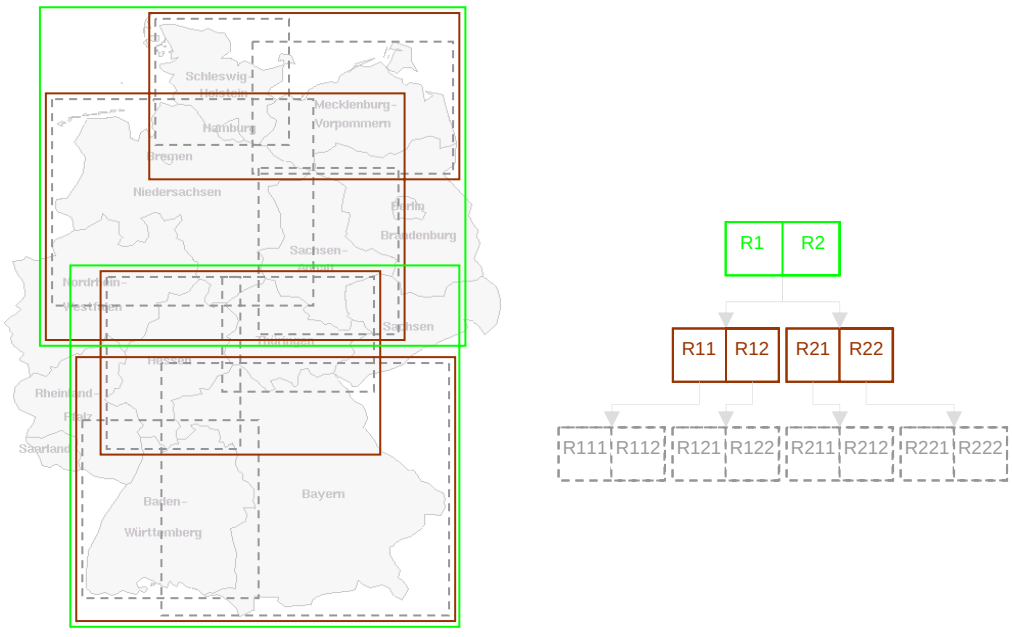


Figure 1: Example of a two-dimensional structure and the according R-tree

Following these principles, according to Guttman, an R-tree corresponds to the following six properties:

(1) Every leaf node contains between m and M index records unless it is the root.

(2) For each index record $(I, \text{tuple-identifier})$ 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 $(I, \text{child-pointer})$ 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.

2.2 Structure of R-tree Nodes

Very much like a B-tree, R-trees consist of two different kinds of nodes: Regular nodes pointing to child nodes and leaf nodes pointing to the indexed data itself.

Regular nodes consist of at least m but at most M references

$$(I_n, CP)$$

to child nodes, with CP standing for child pointer and thus being a reference to a child node on a lower level of the tree structure. I_n is the n-dimensional rectangle forming the MBR of all the child node's rectangles.

Leaf nodes in stead contain m to M tuples of the form

$$(I_n, TID)$$

where TID stands for tuple identifier and refers to a tuple of data objects in the database. I_n again is an n-dimensional rectangle, this time making up the MBR of the referenced data objects.

In both cases

$$I_n = (I_0, I_1, ..., I_n)$$

describes an n-dimensional minimum bounding rectangle of one or more objects, where n is the number of dimensions and I_i is a closed bounded interval $[a,b]$ describing the extent of the object along dimension i.

2.3 Parameters m and M

For R-trees there are mainly two parameters that highly affect its memory consumption as well as the performance concerning insert and delete operations.

The first of these two parameters is M which is to be the maximum number of entries that fit in one node. The second one

$$m \leq \frac{M}{2}$$

specifies the minimum number of entries for a single node. Thus the height of an R-tree containing N index records calculates to at most

$$\lceil \log_m N \rceil - 1$$

with a maximum space utilization of

$$\frac{m}{M}$$

per node.

The latter especially shows that a bigger m will decrease space usage. This is quite clear as almost all the space will be consumed by leaf nodes containing MBRs of the indexed data objects and only a few ordinary nodes containing MBRs of their child nodes. Furthermore m decisively influences that number of occurring underflows when deleting indexed data.

M instead limits the maximum number of entries in a single node an thus influences the number of overflows that will happen when performing an insert. Furthermore when deciding on M, main and cache memory parameters should be taken into consideration as well as the number of dimensions which are to be stored.

Some of these effects will be discussed in chapter 4 when talking about the performance of R-trees under different conditions.

3 Algorithms on R-trees

3.1 Motivation

Just like it is with B-trees and all other kind of trees, the most important idea behind R-trees is not to store data but to structure it. In fact, this is why we call it a structure and not storage. If it was just to store data there would be by far more efficient ways to do so. But what we want when we use R-trees is a structured storage that allows us to efficiently search and modify data. And this is why we need a good set of algorithms to search an R-tree as well as insert, delete and update the indexed data.

Step by step we will now get to know basic implementations of a few quite useful algorithms for search, insert and delete operations on R-trees. Furthermore we will concentrate on node-splitting as an answer to over- and underflows that might occur when inserting or deleting data from the index. As there are quite a few more or less simple ways to deal with this and as they differ extremely in the matter of cost we will explain and compare three different implementations for this case. Last but not least a few words shall also be lost about update and other useful operations which are basically just modified version of the above mentioned algorithms. To round all this up a little, the formal description of each algorithm is followed by concrete application of the referenced algorithm to sample data at the end of each chapter.

3.2 Searching

3.2.1 Algorithm Description

The basic search algorithm on R-trees, similar to search operations on B-trees, traverses the tree from the root to its leaf nodes. However, as there is no rule that prohibits overlapping rectangles within the same node of an R-tree the search algorithm may need to search more than just one subtree of a node visited. This is the reason why it is not possible to give a guarantee on good worst-case performance, although this should seldom be the case due to update algorithms designed to keep the number of overlapping regions small. But now let's take a closer look at the algorithm itself.

Let T be the root of a given R-tree. Be S the search rectangle, the algorithm is intended to identify all index records whose rectangles overlap

S. Denoting an index entry E's rectangle by $E.I$ and its TID or CP by $E.P$, the algorithm looks like this:

Algorithm **SEARCH**

(1) [Search subtrees] If T is not a leaf, check each entry E to determine whether $E.I$ overlaps S. For all overlapping entries, invoke SEARCH on the tree whose root node is pointed to by $E.P$.

(2) [Search leaf node] If T is a leaf, check all entries E to determine whether $E.I$ overlaps S. If so, E is a qualifying record.

3.2.2 Example

In order to make things a little clearer we will now go through the algorithm step by step as we apply it to 2-dimensional sample data represented by the following spatial structure, where all the inner rectangles are MBRs of objects in the database and all other rectangles are MBRs of inner rectangles.

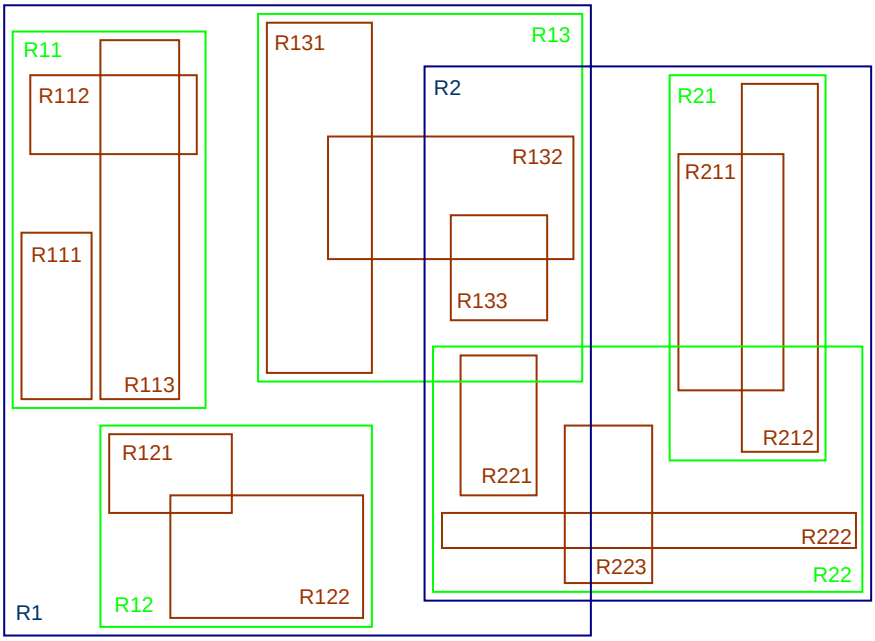


Figure 2: Geometric structure of 2-dimensional sample data

This structure is logically equivalent to an R-tree with the inner rectangles being leaf nodes and all other rectangles making up regular nodes on different levels of the tree according to their level in the above visualization.

Be $M = 3$ and $m = 1$, this data finally leads to the following R-tree structure with nodes named by the rectangles they point to.

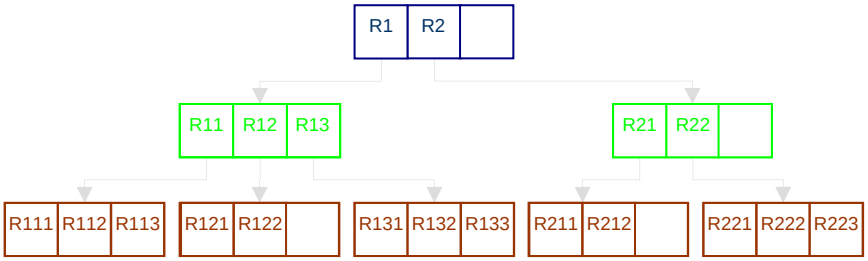


Figure 3: Exemplary R-tree for the given sample data

Right now we will use the search algorithm presented in the last chapter to find all data objects overlapping an object described by the hatched MBR in the following diagram.

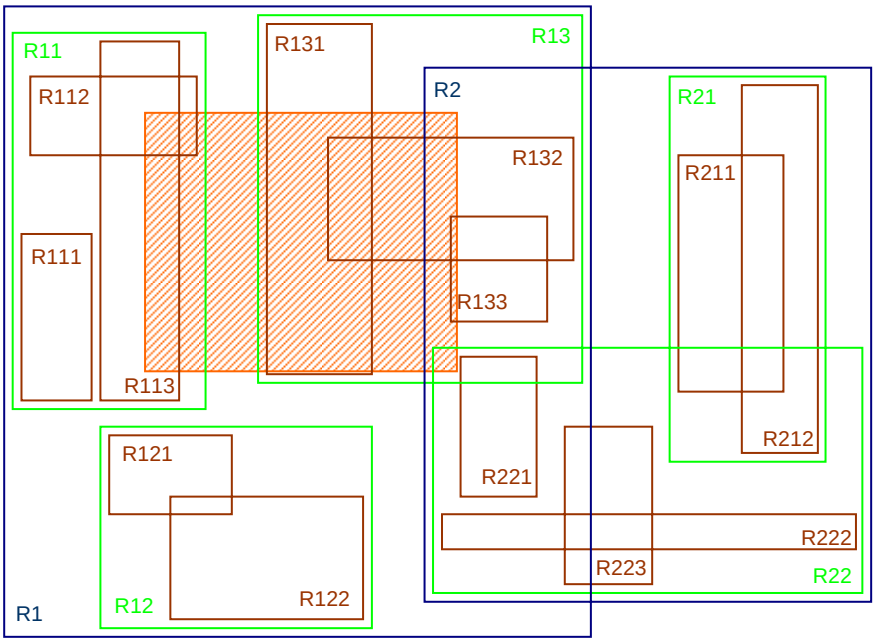


Figure 4: Sample search object within the given data

The algorithm of course starts its work at the root node. As the object's MBR overlaps R1 as well as R2, search will continue in the subtrees of both of the two. Now let's first take a look at the subtree of R2. The algorithm checks R21 and R22 for overlapping with the search rectangle. It will find only R22 overlapping and therefore continue to search its subtree. Being on the leaf level now, it will find none of the referenced rectangles overlapping

the search object, which means that search in the subtree of R2 has come to an end without bringing up any results. Now the algorithm will continue with R1's child node. Rectangles R11 and R13 are found to overlap the search object and their subtrees therefore are going to be searched. Finally search will return nodes R112, R113, R131, R132 and R133 overlapping the search rectangle and therefore the objects referenced by those rectangles respectively by the tuples containing those rectangles make up the result of our search. At last, the following graph is meant to visualize the path of this search through the tree structure.

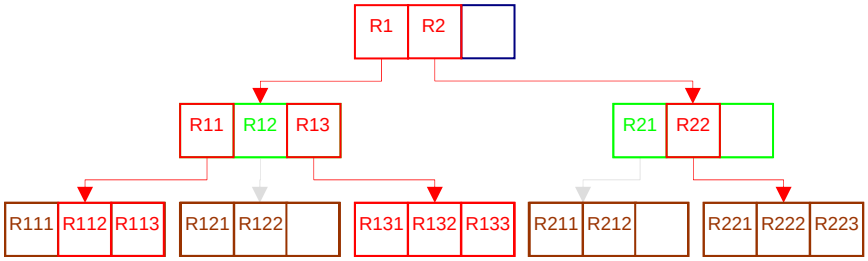


Figure 5: Search path for the given search object

3.3 Insertion

3.3.1 Algorithm Description

Being able to search an R-tree, now let's take a closer look on ways to alter the indexed data. A good start to do so might be to discuss an algorithm that can be used for inserting new data into the tree.

Same as the search algorithm, insert operations on R-trees are quite similar to their equivalent algorithms on B-trees. Basically the insert algorithm consists of three parts. First of all a modified search algorithm will choose an adequate node to insert the new data into. Having located that node, the main part is to insert the new object into this node, followed by a check if any restrictions for the maximum number of entries in a node are violated and if necessary splitting the node according to these restrictions calling the **SPLITNODE** algorithm (see 3.5). Finally the last part is to adjust all preceding nodes according to the new extent of the MBR of the changed node(s).

In detail the algorithm for inserting a new entry E into a given R-tree looks like this:

Algorithm **INSERT**

(1) [Find position for a new record] Invoke **CHOOSELEAF** to select a leaf node L in which to place E.

(2) [Add record to leaf node] If L has room for another entry, install E. Otherwise invoke **SPLITNODE** to obtain L and LL containing E and all the old entries of L.

(3) [Propagate changes upward] Invoke **ADJUSTTREE** on L, also passing LL if a split was performed.

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

Algorithm **CHOOSELEAF** is to search a leaf node L in which to place the new index entry E.

Algorithm **CHOOSELEAF**

(1) [Initialize] Set N to be the root node.

(2) [Leaf check] If N is a leaf, return N.

(3) [Choose subtree] If N is not a leaf, let F be the entry in N whose rectangle $F.I$ needs least enlargement to include $E.I$. Resolve ties by choosing the entry with the rectangle of smallest area.

(4) [Descend until a leaf is reached] Set N to be the child node pointed to by $F.P$ and repeat from (2).

Finally algorithm **ADJUSTTREE** is to ascend from the changed leaf node L to the root, adjusting all MBRs according to the altered MBR of the L as well as to propagate the changes of a node split if one occurred.

Algorithm **ADJUSTTREE**

(1) [Initialize] Set $N = L$. If L was split previously, set NN to be the resulting second node.

(2) [Check if done] If N is the root, stop.

(3) [Adjust covering rectangle in parent entry] Let P be the parent node of N, and let E_N be N's entry in P. Adjust E_N so that I tightly encloses all entry rectangles in N.

(4) [Propagate node split upward] If N has a partner NN resulting from an earlier split, create a new entry E_{NN} with $E_{NN}.P$ pointing to NN and $E_{NN}.I$ enclosing all rectangles in NN. Add E_{NN} to P if there is room. Otherwise, invoke SPLITNODE to produce P and PP containing E_{NN} and all P's old entries.

(5) [Move up to next level] Set $N = P$ and set $NN = PP$ if a split occurred. Repeat from (2).

3.3.2 Example

As we already did when introducing a search algorithm on R-trees, let's again walk through a concrete example of an insert operation step by step and see what happens to our sample R-tree structure already used in chapter 3.2.2.

The following figure shows our sample structure as well as a hatched rectangle being the MBR of an object in the database that is to be inserted into the given index. All other parameters of the given sample structure remain the same as in chapter 3.2.2.

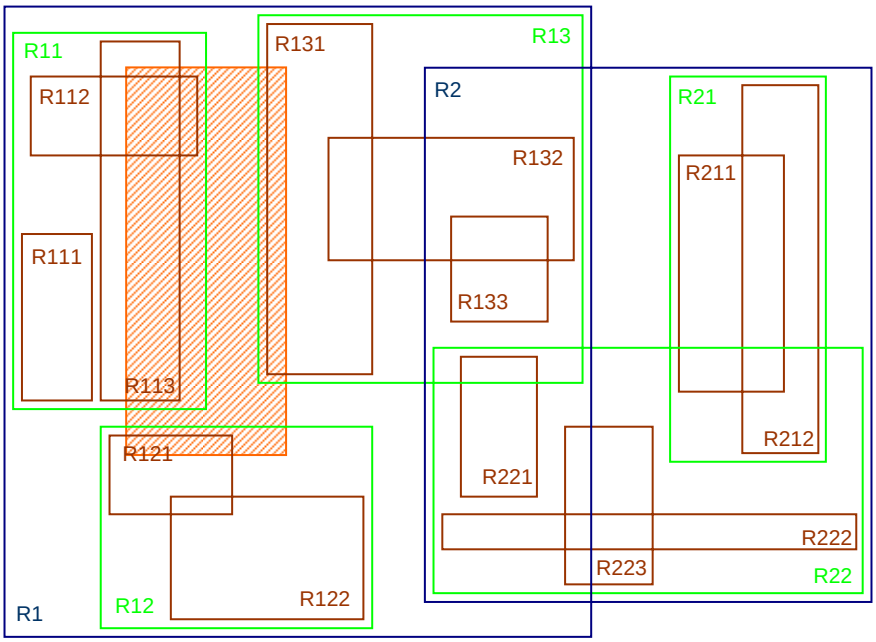


Figure 6: Sample object to insert into a given R-tree

Calling the **INSERT** algorithm now will first of all lead to a call to **CHOOSELEAF**. This algorithm traverses the tree from the root to its leafs and finally identifies R11 as the rectangle which would need the least extension to contain the object we'd like to insert. The algorithms path through the tree is shown by the following figure.

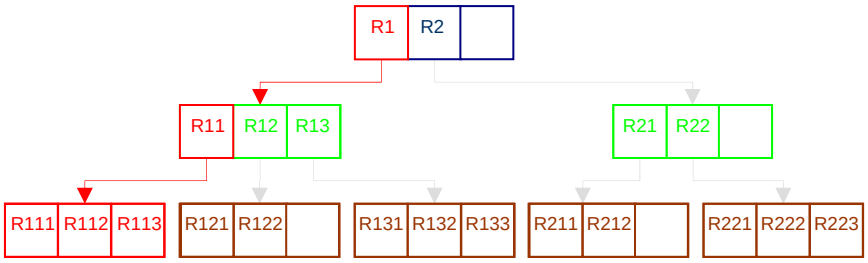


Figure 7: Path of CHOOSELEAF algorithm for inserting the sample object

Having now identified the place where to insert the new object into the tree, the main part of the insert algorithm will find that there is not enough space left to insert the new object as a leaf of R11. Thus **SPLITNODE** is called on R11 to split it into two nodes R11 and R11' with the resulting node's MBRs being minimal. This process of splitting the node is illustrated in the figure below.

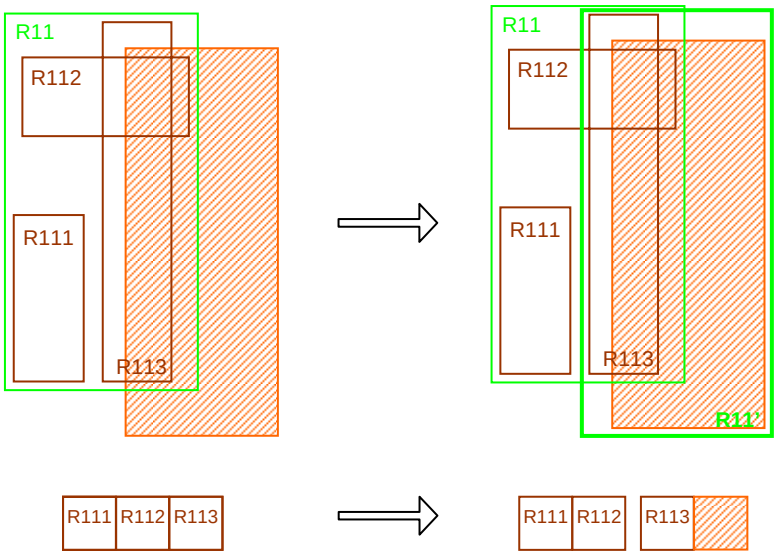


Figure 8: Splitting of the chosen leaf

Now, as nodes have been split and the new object has been inserted into the tree, a call to **ADJUSTTREE** is made to ensure that all nodes preceding the node that's been changed are accordingly changed themselves. In this case, it means that the algorithm tries to insert the new node R11' into R1, which, as you can see, is already at its limit. So again **SPLITNODE** is called and R1 split up into R1 and R1' with R1 containing R11' and R1' containing R11. As there is still space left in the root the resulting call to **ADJUSTTREE** this time just inserts R1' into the root. No further split is required and therefore no new root will be created. Finally we will find the resulting tree containing the new object looking as shown below.

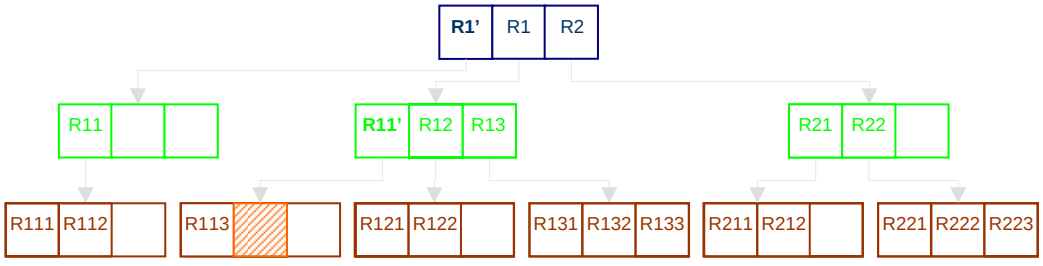


Figure 9: R-tree after insert operation

Last but not least the following figure shows the geometric structure represented by the altered tree.

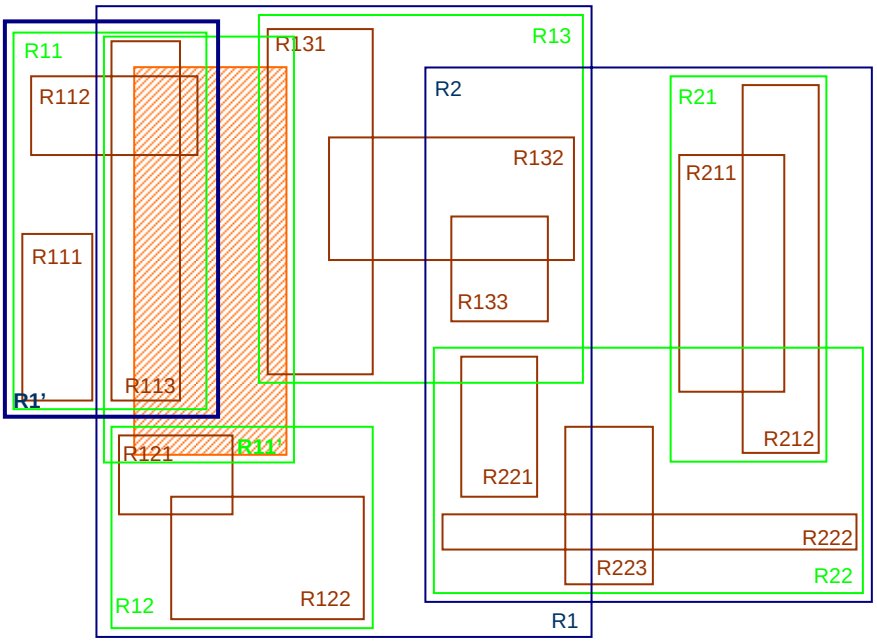


Figure 10: Geometric structure represented by the altered R-tree

3.4 Deletion

3.4.1 Algorithm Description

Having discussed search an insert operations on R-tress we will now focus on the third and last basic function a simple R-tree implementation should provide. That is the deletion of indexed objects.

Although the last chapters all started pointing out that R-tree operations are quite similar to their corresponding operations on B-trees, this time we will at least to some extent miss those similarities. The differences concern the treatment of under-full nodes which may occur when deleting objects from the index. In B-trees, this matter is being dealt with by just merging two or more adjacent nodes. Now on a first quick look, technically, nothing would keep you from doing the same when deleting entries in an R-tree. Nevertheless Guttmann gives two good reasons why this might not be the best approach to deal with underflows and why deleting under-full nodes and reinserting their entries may be considered a better solution. The first one is that deciding on the delete-reinsert version allows us to re-use the insert routine introduced in the chapter before while accomplishing the same thing as when simply merging under-full nodes with sibling ones. This argument of course can only matter if performance of both implementations is about the same. But as Guttmann points out that this will be the case as pages visited

during re-insertion are basically the same visited during the preceding search and therefore should be in memory already. As a second reason for this alternative implementation Guttmann accounts that re-insertions incrementally refine the spatial structure of the tree.

Considering these arguments we will now take a detailed look at the delete-reinsert version and leave it to the reader to give it a try and implement an algorithm corresponding to the B-tree handling of under-full nodes. The algorithm below will remove index record E from a given R-tree.

Algorithm **DELETE**

(1) [Find node containing record] Invoke **FINDLEAF** to locate the leaf node L containing E. Stop if the record was not found.

(2) [Delete record] Remove E from L.

(3) [Propagate changes] Invoke **CONDENSETREE**, passing L.

(4) [Shorten tree] If the root node has only one child after the tree has been adjusted, make the child the new root.

Herein the following algorithm is used to identify the leaf node containing index entry E in an R-tree with root T.

Algorithm **FINDLEAF**

(1) [Search subtrees] If T is not a leaf, check each entry F in T to determine if $F.I$ overlaps $E.I$. For each such entry invoke **FINDLEAF** on the tree whose root is pointed to by $F.P$ until E is found or all entries have been checked.

(2) [Search leaf node for record] If T is a leaf, check each entry to see if it matches E. If E is found return T.

As underflows may occur when deleting index records, the following algorithm will finally eliminate node L from which an entry has been removed if it has too few records and relocate its entries. Furthermore node elimination will be propagated upward adjusting all covering rectangles on the way to the root if they can be tightened as a result of the split.

Algorithm **CONDENSETREE**

(1) [Initialize] Set $N = L$. Set Q, the set of eliminated nodes, to be empty.

(2) [Find parent entry] If N is the root, go to (6). Otherwise let P be the parent of N, and let E_N be N's entry in P.

(3) [Eliminate under-full node] If N has fewer than m entries, delete E_N from P and add N to set Q.

(4) [Adjust covering rectangle] If N has not been eliminated, adjust $E_N.I$ to tightly contain all entries in N.

(5) [Move up on level in tree] Set $N = P$ and repeat from (2).

(6) [Re-insert orphaned entries] Re-insert all entries of nodes in set Q. Entries from eliminated leaf nodes are re-inserted in tree leafs as described in algorithm **INSERT**, but entries from higher-level nodes must be placed higher in the tree, so that leafs of their dependent subtrees will be on the same level as leafs of the main tree.

3.4.2 Example

As with the algorithms before we will now again take a look at an exemplary delete operation on our sample tree. Nevertheless, to make this example a non-trivial one, we will modify our sample structure's parameters and set $M = 4$ and $m = 2$. The resulting tree will look as shown below with the hatched entry R212 being the one to be deleted.

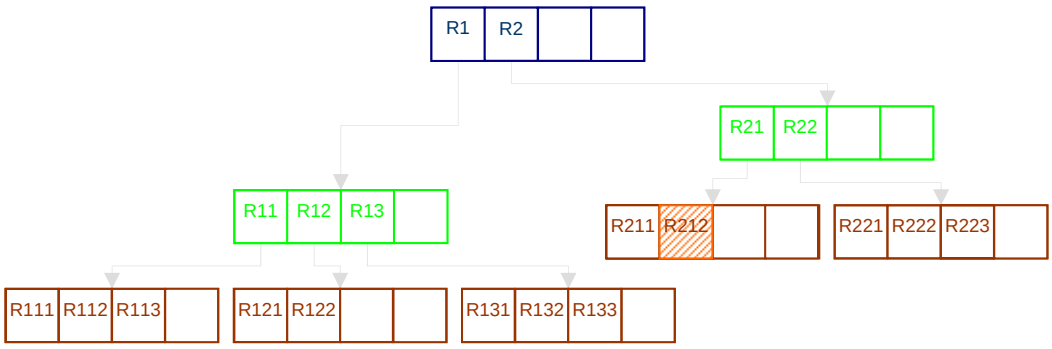


Figure 11: Modified sample tree

Our initial call to algorithm **DELETE** on entry R212 will first of all lead to a call to **FINDLEAF** in order to locate the node containing the entry pointing to

R212. Returning R21 as a result, the main algorithm will delete R212's entry in R21 and call **CONDENSETREE** on this node. Considering the new $m = 2$, R21 is now facing an underflow. Therefore it will be added to the empty set Q with its entry R12 in R2 being deleted. Calling **CONDENSETREE** again on R2 and given the fact that this node is facing an underflow itself now, R2 will be added to Q as well and its entry R2 in the parental node deleted. The last call to **CONDENSETREE** will now find itself at the root level of the tree and according to the algorithm's definition not apply any changes. Therefore the root node remains unchanged.

Right now all entries in Q will be re-inserted into the tree, starting with those entries that have been records in non-leaf nodes before their removal from the tree. These entries will all be inserted on a level such that there leafs will be on the general leaf-node level of the tree. In our case this applies to node R22 only. As all former records of non-leaf nodes haven been inserted, the algorithm now continues with former records of leaf nodes inserting them at the leaf-level. Therefore R211 will be inserted into node R22 and **CONDENSETREE** will finish. The following figures are to illustrate these operations.

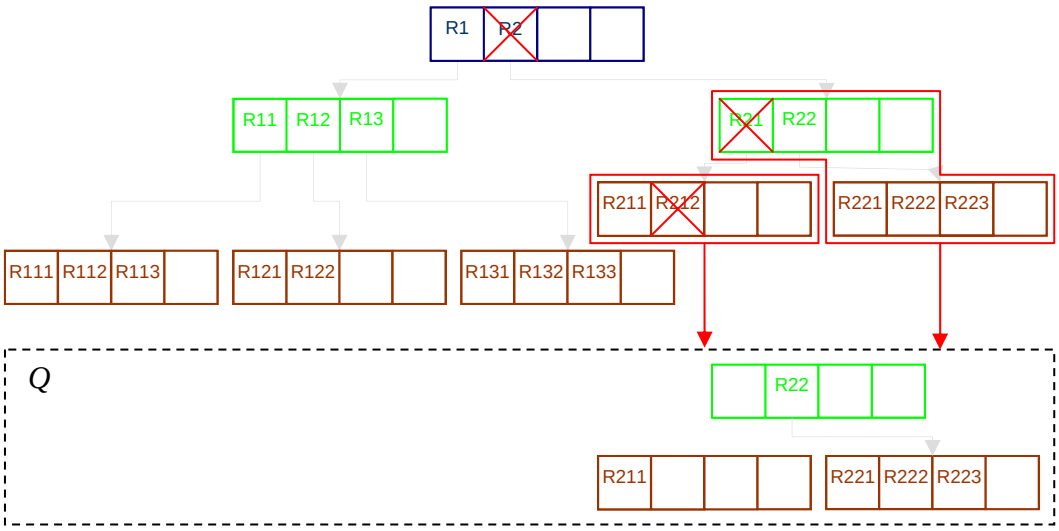


Figure 12: Saving nodes whose references have been removed from tree

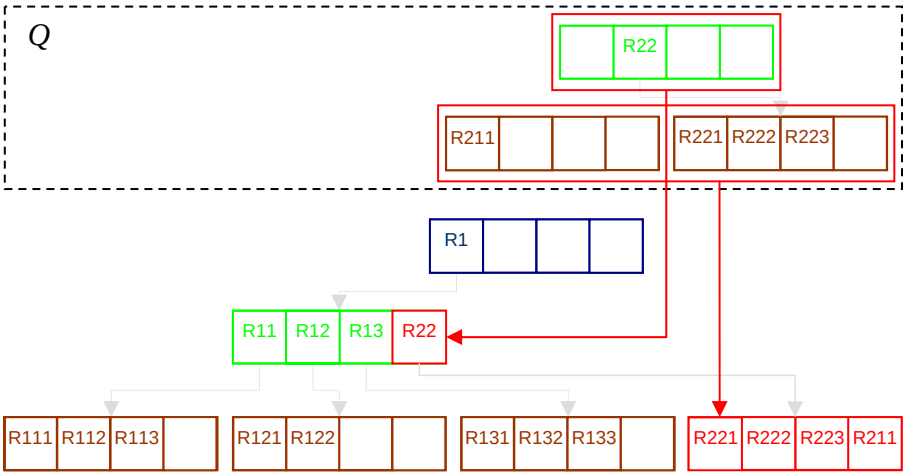


Figure 13: Re-insertion of saved entries

Eventually a last check turns out the root node to contain only a single entry which will now become the new root of the tree and we'll see the new R-tree structure without the deleted entry R212 as shown below.

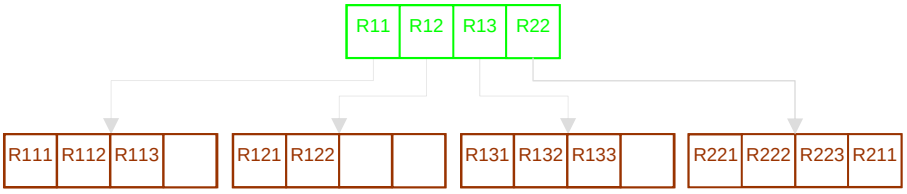


Figure 14: Sample R-tree structure after deleting R212

3.5 Node Splitting

3.5.1 Exhaustive Approach

Right now we have introduced two ways of manipulating an R-tree structure by inserting or deleting data. As we have seen, in either algorithm we have to deal with occurrences of underflow and overflow, i.e. that we may want or have to insert or delete data into or from a leaf that already contains the maximum or in the other case the minimum number of entries. The way how we deal with that matter is splitting up nodes into two and adjusting all preceding nodes in the tree that may also have to be split up as a result of this. As in the chapters before we just mentioned calls to **SPLITNODE** we are now going to have a closer look at the algorithms that perform the split.

As Guttmann presents us with three different algorithms to implement the split node routine, we are first of all going to take a look at a quite exhaustive

and naïve yet simple approach to this problem. The basic and quite comprehensible idea behind this implementation is to simply try all possible split-ups and then simply select the best one. Now this brings up two questions, the first of which refers to what we mean by “the best one”.

According to Guttmann a good split minimizes the total area of the MBRs generated by the split. This issue is to be illustrated by the follow figure.

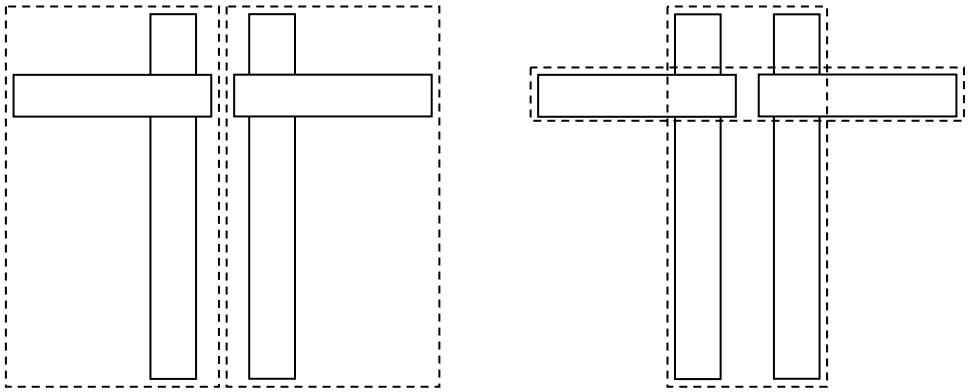


Figure 15: Different quality of node splitting

Now the second question is about the efficiency of the above mentioned algorithm. It should be easy to understand that there are 2^{M-1} different possibilities to split up a node containing M entries. Common sense tells us that this is not an option for an efficient implementation of an R-tree structure, which is why we are going to get to know two more advanced algorithms to deal with the splitting no R-tree nodes.

3.5.2 Quadratic-Cost Algorithm

This approach to the matter of node splitting will try to find a good split although there is no guarantee it will find the best split possible. Nevertheless this leads to a cost quadratic in M as well as linear in the number of dimensions.

The idea behind this implementation is to pick the two entries out of the M entries in the original node together with the new entry that would consume the most space if put together in a node (subtracting their own area), that is that would waste the most space it they were part of the same node. One of these two entries is then put into the first node, whereas the

other one is put into the second one. For all the remaining entries, the algorithm will step by step pick the one which creates the biggest difference in area when added to one of the new nodes and finally assigns it to the node which gains less in area by adding this entry. This is repeated till all of the remaining entries are assigned two one of the nodes and we finally get two new nodes containing all the entries of the original node plus the new entry that was to be inserted (assuming we were performing an insert operation). In case we were performing a delete operation things are but slightly different.

Given this short and intuitive explanation we can now put it into a more formal and exact description of a quadratic-cost algorithm dividing $M+1$ index records into two groups, which would look like the following.

Algorithm **QUADRATICSPLIT**

(1) [Pick first entry for each group] Apply algorithm **PICKSEEDS** to choose two entries to be the first elements of the groups. Assign each to a group.

(2) [Check if done] If all entries have been assigned, stop. If one group has so few entries that all the rest must be assigned to it in order for it to have the minimum number m, assign them and stop.

(3) [Select entry to assign] Invoke algorithm **PICKNEXT** to choose the next entry to assign. 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 smaller area, then to the one with fewer entries, then to either. Repeat from (2).

As you can see, algorithm **QUADRATICSPLIT** makes use of two more algorithms, the first one of which is to select two entries to be the first ones of the new groups and looks as follows.

Algorithm **PICKSEEDS**

(1) [Calculate inefficiency of grouping entries together] For each pair of entries E_1 and E_2, compose a rectangle J including $E_1.I$ and $E_2.I$. Calculate $d = area(J) - [area(E_1.I) + area(E_2.I)]$.

(2) [Chose the most wasteful pair] Choose the pair with the largest d.

Last but not least the second algorithm is used to select one of the remaining entries for classification in a group.

Algorithm **PICKNEXT**

(1) [Determine cost of putting each entry in each group] Be G_1 and G_2 the two new groups to which the entries are to be assigned. For each entry E not yet in a group, compose a rectangle J including $G_1.I$ and $E.I$ and calculate $d_1 = area(J) - area(E.I)$. Calculate d_2 similarly for the second group.

(2) [Find entry with greatest preference for one group] Choose any entry with the maximum difference between d_1 and d_2.

3.5.3 Linear-Cost Version

To come to an end with this chapter we'll at last have a short look on a linear-cost algorithm which is basically the same as the above quadratic-cost version. The only differences can be found in modifications to **PICKSEEDS** and **PICKNEXT**. In fact PICKNEXT is changed to simply select any of the remaining entries.

The modifications to PICKSEEDS which make the algorithm's cost linear in M as well as linear in the number of dimensions of the stored data are as shown below.

Algorithm **PICKSEEDS**

(1) [Find extreme rectangles along all dimensions] Along each dimension, find the entry whose rectangle has the highest low side, and the one with lowest high side. Record the separation.

(2) [Adjust for shape of the rectangle cluster] Normalize the separations by dividing by the width of the entire set along the corresponding dimension.

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

3.6 Updates and Further Operations

Besides the operations presented above, there are quite a few more that might be useful and are also well supported by R-tree structures. A very obvious one is updating the index when parts of the underlying data have changed. Basically this will result in deleting the index record of the affected data and re-inserting it after updating its MBR according to the changed data record. Thus the record will finally be in the right place in the index again.

Furthermore modified search algorithms might be very handy. One may want to look for all data object completely containing the search object or vice versa all objects that are completely within the search area. Both of these operations can be easily implemented with slight modifications to the search algorithm presented above. Besides that one might also have use for a search of a specific know entry which we have already implemented as a part of the DELETE algorithm when calling FINDLEAF.

As a last aspect to be mentioned one might think about several variants of range deletion which also shouldn’t be too hard to implement modifying the given DELETE algorithm and combining it with parts of the SEARCH algorithm.

In the end one could say that on the matter of supported algorithms R-trees are quite as well extensible and efficient for spatial data as B-trees in the subject of one-dimensional.

4 Performance Tests and Benchmarking

As we have discussed a good set of algorithms useful when working on R-trees we will now take a short look on some performance and memory benchmarks that were made by Guttmann to proof the practicality of his R-tree structure and the algorithms on it.

All of the performance tests have been done using a C-implementation of R-trees on a Vax 11/780 computer running Unix. Test data has been gained from the layout data of a RISC-II computer chip consisting of 1057 rectangles. Trying not only to proof the practicality of the structure but also to choose suitable values for m and M as well as to evaluate different node-splitting algorithms, Guttmann performed several tests with varying page sizes and accordingly changed M. The table below shows the page sizes used for benchmarking.

Bytes per Page	Max. Entries per Page (M)
128	6
256	12
512	25
1024	50
2048	102

Figure 16: Different page size and according parameter M

As discussing Guttmann's testing methods in detail would lead way too far at this point we will have a look at a few diagrams resulting from these tests and point out only the most important aspects they show up. We'll start by taking a look at insert and delete operations. The two figures below show the time needed to perform a certain insert and delete operation depending on the page size, using three different versions of the SPLITNODE algorithm. As expected linear implementation is the fastest. Nevertheless you will also notice that CPU cost for the linear algorithm is to a very small extent only depending on the page size and parameter m, strongly in contrast to the quadratic and exhaustive implementation when performing an insert operation. Concerning deletions you can see that they too a greatly influenced by the chosen m.

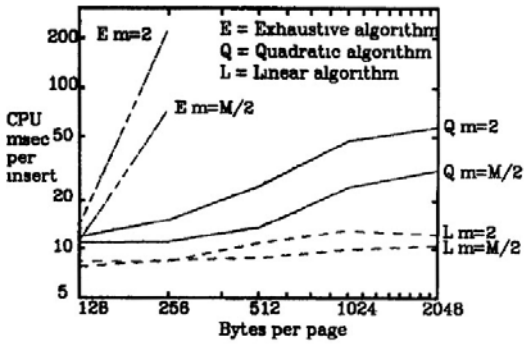


Figure 17: CPU cost of inserting records

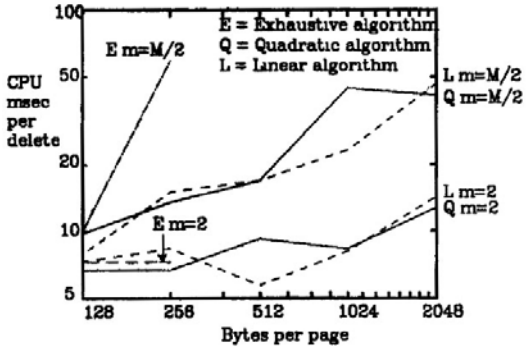


Figure 18: CPU cost of deleting records

Let us now take a closer look at the effects of different node-splitting algorithms on search performance on an R-tree. As figures below point out, search performance remains almost constant no matter which node-splitting implementation has been used to fill the tree structure although exhaustive implementation must be granted a slight advantage. Further on you can see that search is quite insensitive to parameter m.

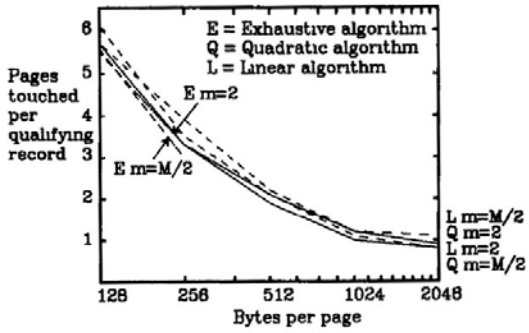


Figure 19: Search performance – Pages touched

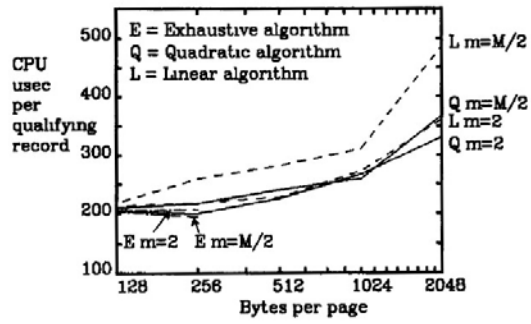


Figure 20: Search performance – CPU cost

As a next step, we are going to analyze space efficiency of an R-tree again compared by the split-node algorithm used and the page size but especially depending on the chosen node restrictions, which is parameter m. Given bellows diagrams one will find that exhaustive as well as quadratic-cost implementations are especially sensitive to m whereas the linear version is the only one that shows at least to some extent a certain constance. As one expects space usage is heavily improved by applying stricter node fill criteria. According to Guttmann this may reduce space usage by up to 50 per cent.

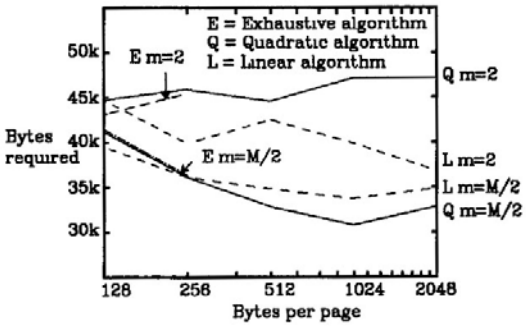


Figure 21: Space efficiency

A second series of tests is now to give us an impression of how the performance of R-tree operations depends on the amount of data stored in the index. All benchmarks have been performed using a linear split-node implementation with $m = 2$ and quadratic-cost version with $m = \frac{M}{3}$. Page size has been set to 1024 Byte for either implementation. The amount of data indexed has been varied starting at 1057 rectangles, increasing it to 2238 and 3295 objects and finally performing on 4559 rectangles.

Again we'll at first take examine the performance of insert delete operations this time depending on the amount of data indexed in the tree. The diagram below states nearly constant costs for a quadratic algorithm except where the tree increases in height, as this leads to more levels on which a node split may occur. As the linear algorithm's curve doesn't show any jump at all we can assume that linear node splitting accounts for only a small part of the cost of insert operations.

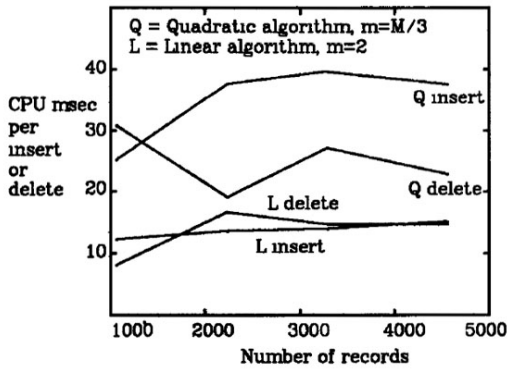


Figure 22: CPU cost of inserts and deletes vs. amount of data

Regarding delete operations we shall mention that no node splits occurred during the benchmark which is according to Guttmann due to relaxed node fill requirements and the relatively small number of data items. This is why jumps only appear where the height of the tree increases. In contrast the quadratic-cost version produced a varying number of splits and thus seems to be very rough. Basically you can interpret these results as that the cost of insert and delete operations is independent of the tree width whereas it is influenced by its height which grows slowly with the number of data items.

Next two diagrams now show search performance on R-trees as a function of the amount of data being indexed. Basically those two figures show no more than the fact that search efficiency is nearly the same for either version of the SPLTNODE algorithm and that the index is quite effective in directing the search to small subtrees. Jumps in those figures can be explained by a decreasing significance of higher level nodes with increasing tree height.

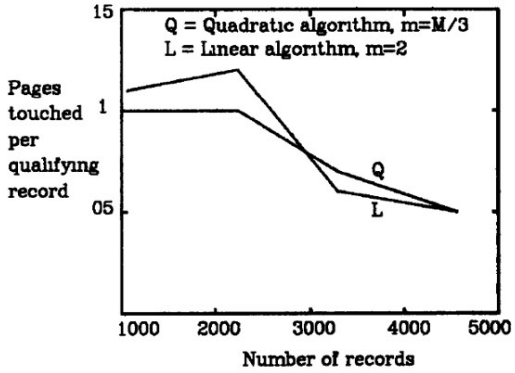


Figure 23: Search performance vs. amount of data – Pages touched

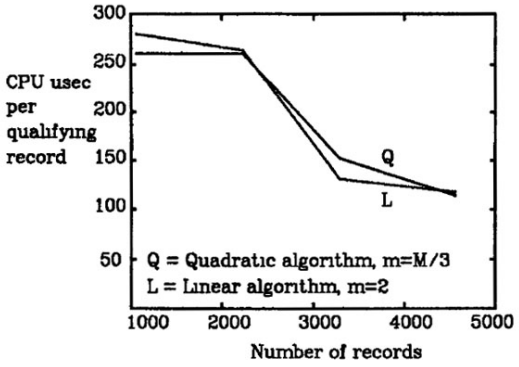


Figure 24: Search performance vs. amount of data – CPU cost

Last but not least we will again take a quick look at space efficiency of an R-tree. Therefore we'll draw the space required fort he tree structure as a function of the amount of data indexed. This is exactly what is shown by the diagram below. As you can see the results are quite straight lines without jumps. The reason for this is the fact that most of the space in an R-tree is consumed by its leafs. For the linear benchmark the structure used up 40 bytes per data item compared to 33 bytes with the quadratic-cost implementation. 20 bytes thereof were consumed by the index entry itself.

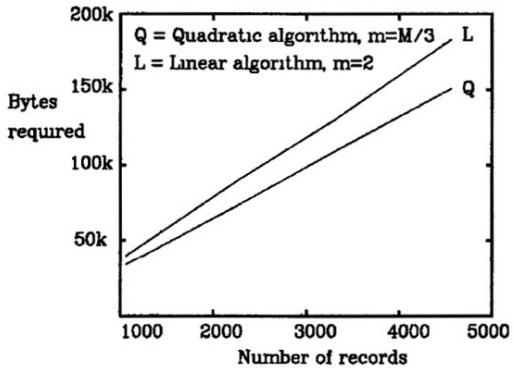


Figure 25: Space required for R-tree vs. amount of data

As a conclusion one can say that R-trees show up really great performance as long as node parameters are in reasonable relation to the memory page size. In addition to that we have learned the linear **SPLITNODE** algorithm to be as effective as its quadratic-cost and exhaustive correspondents.

5 R-tree Modifications

Although we have learned that R-trees are a very efficient and comfortable way to structure and work on spatial data there is no hint that they are the ultimate way to do so. Since its occurrence, Guttmann's original concept has been reviewed and reconsidered by quite a few computer scientists over the years. Some of them have made only small modifications to the original R-tree, whereas others have even tried to combine them with other data structures to gain a performance advantage. Before presenting you with a final conclusion on the topic of R-trees, at least a few of these modified R-tree structures shall be mentioned.

The first structure to be mentioned here is the so called packed R-tree. The basic idea behind this structure is to remove unused space from structure in order to reduce memory usage.

R^+-trees in stead are designed to reduce overlapping rectangles by dividing them into disjunctive MBR. As you might expect, this results in a more complex structure that becomes harder to maintain. Nevertheless this drastically increases the search performance on the tree.

Structurally identical to the R-tree, R^*-trees take their advantage out of modified insert and delete routines which take overlapping and circumference of MBRs into consideration.

Another quite interesting concept can be found when looking at TV-trees. In contrast to R-trees, TV-tree structures abandon the concept of representing spatial data by its MBR and thus allow the use of more complex approximations for the objects to be indexed.

X-trees again try to gain an advantage by avoiding overlapping of nodes that especially increases the greater the number of dimensions is. The basic concept how X-trees deal with this aspect is to dynamically increase the R-tree node's capacities.

As a last modified R-tree structure QR-trees should be mentioned. Pursuing a hybrid concept, QR-trees are a combined structure trying to unite the advantages of quad trees with those of regular R-trees. As the discussion

of those would lead way too far, interested readers may like to refer to H. Samet's comments on the design and analysis of spatial data structures [7].

6 Conclusion

As already mentioned in the introduction the importance of processing and storing spatial data has increased significantly over the last years. No matter if we talk about image processing, computer aided design or cartography. What they all have in common is the fact that they are usually dealing with vast amounts of data which are simply hard and inefficient to be handled by common index structures. Therefore most current databases that are to store multi-dimensional data rely on spatial structures such as R-trees and there modifications. These structures are an advanced, very well-developed and efficient way to deal with spatial data and in contrast to most other multi-dimensional indexes allow the processing not only of point data but also areal objects.

Therefore one could say that when it comes to the processing and indexing of spatial objects the R-tree is what B-trees are in the field of one-dimensional data indexes.

7 Glossary

- **CP,** child-pointer
  Reference stored in a node, pointing to another node on a lower level of the index structure.
- **MBR,** minimum bounding rectangle
  The smallest n-dimensional rectangle that can be laid around an n-dimensional object so that it completely covers the object along each dimension.
- **Overflow**
  The insertion of data into a node exceeds the maximum number of entries for nodes of the given R-tree.
- **TID,** tuple-identifier
  Reference stored in leaf nodes pointing to tuples of objects in the database.
- **Underflow**
  The deletion of data from a node under-runs the minimum number of entries for nodes of the given R-tree.

8 Figures

9 References

1 J. L. Bentley: Multidimensional Binary Search Trees Used For Associative Searching, *Communications of the ACM 18*, 1975.

2 R. A. Finkel, J. L. Bentley: Quad Trees – A Data Structure For Retrieval And Composite Keys, *Acta Informatica 4*, 1974.

3 A. Guttman: R-trees: A Dynamic Index Structure for Spatial Searching, *Proceedings of the 1984 ACM-SIGMOD Conference*, S. 46 -57, Boston, MA, 1984.

4 A. Guttmann, M. Stonebraker: Using A Relational Database Management System For Computer Aided Design Data, *IEEE Database Engineering 5*, 1982.

5 T. Rabl: R-Bäume, Proseminar – Algorithmen und Datenstrukturen für Datenbanken, Universität Passau, 2001

6 J. T. Robinson: The K-D-B Tree: A Search Structure For Large Multidimensional Dynamic Indexes, *ACM-SIGMOD Conference Proc*, 1981.

7 H. Samet: The Design and Analysis of Spatial Data Structures, *Addison-Wesley*, Reading, MA, 1990

10 Index