A DYNAMIC INDEX STRUCTURE FOR SPATIAL DATABASE QUERYING BASED ON R-TREES

YANG Gui-jun[a], ZHANG Ji-xian[b]

[a] Institute Remote Sensing Applications, Chinese Academy of Sciences, No.9718 Mail Box, Road Datun, District Chaoyang, Beijing, P.R.China,100101,E-mail: guijun.yang@163.com
[b] Chinese Academy of Surveying and Mapping, No.16, Road Beitaiping, District Haidian, Beijing, P.R.China,100039
E-mail: stecsm@public.bat.net.cn

KEY WORDS: Spatial Database Querying, R-Tree, Dynamic Index Structure

ABSTRACT:

In recent years, spatial database has become an important area of people's interest and research. A fundamental issue in this area is how to store and operate spatial data efficiently. This paper describes a data structure, R-Tree, which is very easy to understand but prove to be very powerful, and the concerning operations on it.
In the traditional R-Tree, it's not specified where an R-Tree index can be stored. Either the main memory or the second memory can be used to store R-Tree. In this paper, I give an advanced R-Tree dynamic index structure which shows intuitively what an R-Tree is like. Then you can efficiently complete operations on R-Tree, such as search, insert, delete, node splitting, updates and other operations.

1. INTRODUCTION

R-Tree is a spatial access method (A data structure to search for lines, polygons, etc) which splits space with hierarchically nested and possibly overlapping boxes. The following is the description of R-Tree: Let M be the maximum number of entries that will fit in one node and let m<=M/2 be a parameter specifying the minimum number of entries in a node. An R-tree satisfies the follow properties.
1)Every leaf node contains between m and M index records unless it is the root.
2)For each index record (I, 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, child-pointer) in a non-leaf node, I is the smallest rectangle that spatially contains the rectangles in the child node.
5)The root nodes have at least two children unless it is a leaf.
6)All leaves appear on the same level.
In the above description, it's not specified where an R-Tree index can be stored. Either the main memory or the second memory can be used to store R-Tree.

2. SPATIAL OBJECTS ARRANGED IN R-TREE

In this paper, I give an advanced R-Tree dynamic index structure which shows intuitively what an R-Tree is like. Then you can efficiently complete operations on R-Tree, such as search, insert, delete, node splitting, updates and other operations. A more specified description is given which suppose the R-Tree should be stored in the second memory. And the analysis of space taken is also given. It's as follows:
Let N = # of rectangles, M = # of rectangles fitting in internal memory, B = # of rectangles per disk block, where N>>M and

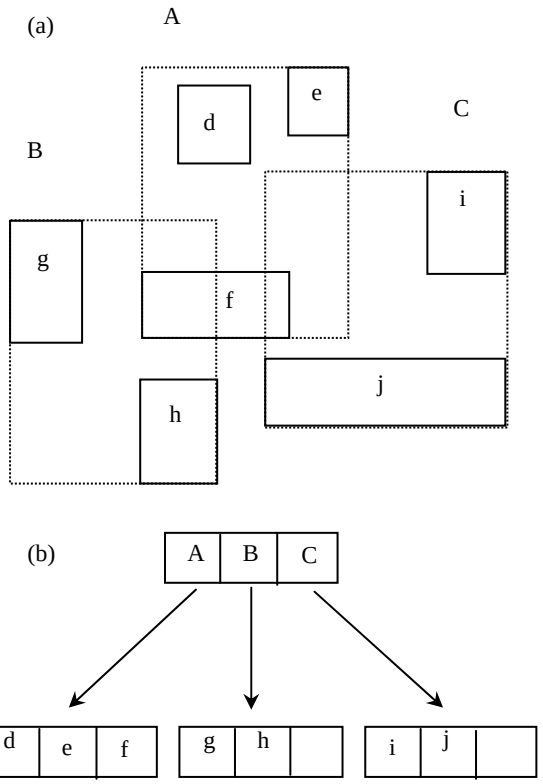

Figure 1: (a) Spatial objects arranged in R-Tree hierarchy. (b) R-Tree file structure on disk(Block size 3)

1<=B<=M/2. An R-Tree is a height-balanced multi-way tree similar to a B-Tree. The leaf nodes contain (B) data rectangles each, while internal nodes contain (B) entries of the form(Ptr, R), where Ptr is a pointer to a child node and R is the minimal bounding rectangle covering all rectangles in the sub-tree rooted in that child.

An R-tree occupies O(N/B) disk blocks and has height O(logBN);insertions can be performed in O(logBN)I/Os.

3. OPERATIONS ON R-TREE

When dynamic structure R-Tree be designed, you can you can efficiently complete operations on spatial database, such as search, insert, delete, node splitting, updates and other operations. The algorithms given in this paper look more like description rather than procedure. I revised them according to the criteria of the pseudo code.

3.1 Search

Search algorithm accomplishes the following task, given an R-Tree whose root node is T, find all index records whose rectangles overlap a search rectangle S. We denote an entry in a node as E(EI, EP), where EI represents the smallest rectangle bounding the sub-tree or the spatial object, EP is the pointer to the sub-tree or the spatial object.

```
SearchSubTree(t, s)
1. If t is not a leaf
2.then for each entry E in t do
3. if EI overlaps S
4. then SearchSubTree(EP, s)
5. else SearchLeaf(T, s)

SearchLeaf(t, s)
1. for each entry E in t
2. do if EI overlaps s
3. then output E
```

Searching an R-Tree is unlike searching an B-Tree, All internal nodes whose minimal bounding rectangles intersect with the search rectangle may need to be visited during a search. So a worst case performance is O(N) instead of O(lgN). Intuitively, we want the minimal bounding rectangles stored in a node to overlap as little as possible so that we need to search as little nodes as possible.

We can apply the searching of an R-tree to find objects that overlap a search object, say o, by the following steps.

```
SearchObj(t, o)
1. s←bounding box of the search object o
2. SearchSubTree(t,s)
```

and revise the above SearchLeaf(t,s) as follows:

```
SearchLeaf(t,s)
1. for each entry E in t
2. do if EI = s
3. then if EP = o
4.then output E
```

3.2 Insertion

Like insertion in B-Tree, inserting new data tuple into R-Tree may cause splitting nodes and the splits propagate up the tree. Furthermore, an insertion of a new rectangle can increase the overlap of the nodes. Choosing which leaf to insert a new rectangle and how to split nodes during re-balancing are very critical to the performance of R-Tree. The heuristics to do node-splitting are discussed later in this paper.

Algorithm Insert: Insert a new index entry E into an R-Tree T.

```
Insert(E, t)
1. L←ChooseLeaf(E, t) > select a leaf node L where to place E
2. If L need not split
3. then install E
4. else SplitNode(L)
5. AdjustTree(L)

ChooseLeaf(E,t)
1. N←t
2. while N is not a leaf
3.do choose the entry F in N whose rectangle FI needs least enlargement to include EI
4.N←FP
5. return N

AdjustTree(L)
1. N←L
2. if L was split previously
3. NN←resulting second node
4. While N is not the root
5. do P←parent(N)
6. En←entry in P which points to N
7. Adjust EnI so that it tightly encloses all entry rectangles in N
8. if N has a partner NN which results from an earlier split.
9. then create Enn so that EnnP point s to NN and EnnI enclose all rectangles in NN.
10. if there is room in P
11. then add Enn to P
12. else splitNode(P) to produce P and PP
13. N←P
14. if there exists PP
15. then  NN←PP
16. return
```

SplitNode() is shown later.

One insertion takes O(lgN) I/O. To construct an R-Tree Index Structure, we have to repeatedly insert an objection. It will take O(NlgN) I/Os and is very slow. Moreover, it has other disadvantages: sub-optimal space utilization, and, most important, poor R-tree structure which requires the retrieval of an unduly large number of nodes in order to satisfy a query. To improve the operations, people do a lot of work later.

One of the improvement is called R-Tree Packing algorithm. The general algorithm is as follows.

1. Pre-process the data file so that the r rectangles are ordered in r/n consecutive groups of n rectangles, where each group of n is intended to be placed in the same leaf level node.
2. Load the r/n groups of rectangles into pages and output the (MBR, page-number) for each leaf level page onto a temporary file. The page-numbers are used as the child pointers in the nodes of the next higher level.
3. Recursively pack these MBRs into nodes at the next level, proceeding upwards, until the root node is created.

3.3 Deletion

Algorithm Delete: Remove an index record E from an R-Tree

```
Delete(E, t)
1. L← FindLeaf(E, t)
2. If  L is null
3. Then return
4. Remove E from L
5. CondenseTree(L)
6. If the root node has only one child.
7. then make the child the new root.

FindLeaf(E, t)
1. if t is not a leaf
2. then for each entry F in t
3. do if FI overlaps EI
4. then FindLeaf(E, FP)
```

5. else for each entry F in T
6. do if FI = EI & FP=EP
7. then return T

CondenseTree(L)
1. N← L
2. Q← empty
3. while N is not the root
4. do P←parent(N)
5. En← the entry of N is P
6. if N had fewer than m entries
7. then delete En from P
8. add N to set Q
9. else adjust EnI to tightly contain all entries in N
10. N←P
11. reinsert all entries of nodes in set Q according to their level.

Like B-tree, the deletion of an index record may result in a leaf containing less than m entries(called "node underflow") and requires the need for restructuring of the index. And the restructuring may propagate up the tree.

The restructuring can be done two ways. One is just like B-Tree. The underflow node is merged with whichever sibling will have its area increased least. Or the orphaned entries can be distributed among sibling nodes. Another way is reinserting the orphaned entries just like the algorithm shown above. The main reason of choosing reinsertion is to try to obtain a better index by forcing a global reorganization of the structure instead of the local reorganization of a node merge constitutes.

3.4 Updates and Other Operations

When a data tuple indexed by an R-Tree is updated, its bounding rectangle my need to be changed. In that case the update of R-Tree is needed. There is no straight way to do the update. The only way is for the index entry to do deletion, update and then re-insertion.

Lots of other kinds of searching or deletion which not mentioned above can be done by alternating the above algorithms a little bit.

3.5 Node Splitting

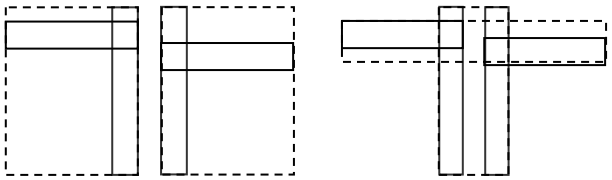

Figure 2 a good split and a bad split based on the same groups of rectangles

Now we go to the most interesting part of this paper – how to split a node. To add a new entry to a full node, it's necessary to divide the collection of M+1 entries into two nodes. And the important thing is that the total area of the two covering rectangles after a split should be minimized.

In figure 2, a good split and a bad split are shown based on the same groups of rectangles.

3.5.1 An Exhaustive Algorithm: This is the most straightforward but slowest way. It is to generate all possible groupings and choose the best such that the area of the bounding rectangle is the smallest. Exponential number of choices $O(2^{M-1})$ must be examined. In practice, M is the number of rectangles stored in one memory block. It's usually very large. The algorithm is obviously too slow. This algorithm is only a theoretic one not a practical one.

3.5.2 A Quadratic-Cost Algorithm: The idea of the algorithm is very simple. It first takes each pair of entries, calculates the waste area of them and chooses the pair which waste area is the most. Here waste area is defined as follows: the area of the bounding box of two elements together minus the areas of the bounding box of them separately. The elements in the chosen pair are put into two groups. These two groups are the result groups. Then it continuously calculate the other elements to see whether there will be large different if they are put in one group rather than other group. If there be, put the element into the concerning group. If there isn't, leave the element until one group is almost underflow, and put the all the remaining elements into this group. This method takes O(M2) time. It does not guarantee the smallest area but it's practical and easy to implement.

QuadraticSplit(T)
1. PickSeeds(T, G1, G2)
2. assign G1 to a group, G2 to another group.
3. while not all entries have been assigned
4. do if m = number of entries in one group + numbers entries not assigned
5. then assign all the entries to this group.
6. else F = PickNext(T)
7. add F to the group whose covering rectangle will have to be enlarged least

Pickseeds(T, G1, G2)
1. for each pair of entry E1 and E2 in node T
2. do compose a rectangle J including E1 and E2.
3. d←area(J)-area(E1I)-area(E2I)
4. Choose the pair with the largest d and G1←E1, G2←E2.

PickNext(T,E,D)
1. For each entry E not yet in a group,
2. do d1←the area increase required for the covering rectangle of Group 1 to include EI
3. d2←the area increase required for the covering rectangle of group 2 to include EI
4. Chose any entry with the maximum difference between d1 and d2.
5. Return the entry E and the group D

3.5.3 A Linear-Cost Algorithm: The steps are the same with Quadratic cost algorithm. Only the methods to pick seed and pick next are different. It's linear in M and in the number of dimensions.

LinearPickSeeds()
1.Along each dimension, find the entry whose rectangle has the highest low side and the one with the lowest high side, record the separation
2.Normalize the separations by dividing it by the width of the entire set along the corresponding dimension
3.Choose the pair with the greatest normalized separation along any dimension.

PickNext() simply chooses any of the remaining entries.

To make LinearPickSeeds Algorithm clear, let's look at figure 3. Which is a 2-dimensional example. We will show x-dimension in detail.

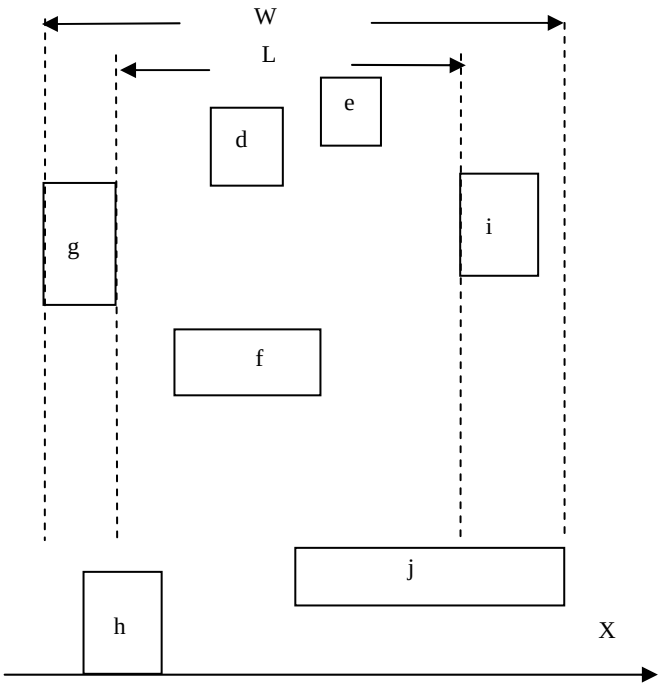

Figure 3 The workflow of Linear-Cost Algorithm

4. CONCLUSIONS

The analysis is based on some experimental result. The linear algorithm proved to be as good as more expensive techniques. It's fast and the slightly worse quality in the splits did not affect the search performance noticeably.

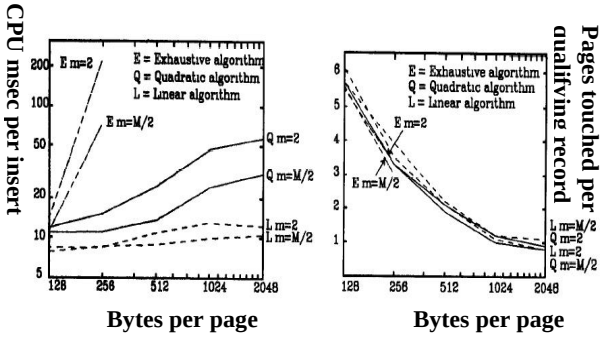

Fig.4 CPU cost of insert records and search performance pages touched

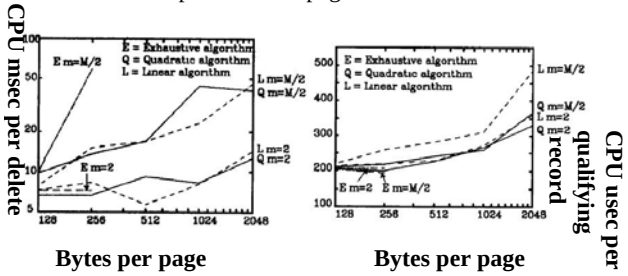

Fig.5 CPU cost of delete records and search performance CPU cost

The algorithm introduced in this paper is simple. A lot of further researches are done on R-Tree to improve its performance. Bulk operations, which mean a large number of operations are performed on the index at the same time, are one of them. And some special techniques, such as lazy buffering technique, are also considered. R-Tree proves to be very useful data structure in controlling spatial data especially in second memory. It's now being used by IBM Informix system as a rapid and efficient indexing method to process data.

Besides operating spatial database, R-Tree can be used to control other types of data as follows. Combinations of numerical values treated as multidimensional values. Range values, as opposed to single point values, such as the time of a course.

More work still needs to be done to improve the performance of R-Tree operations. And much more applications of R-Tree need us to explore.

REFERENCES:

M Astrakhan, et al, System R Relational Approach to Database Management, *ACM Transactions on Database Systems 1*,2(June 1996),p97-137

D Comer, The Ubiquitous B-tree, Computing Surveys 11,2(1989),p121-138

G Yuval, Finding Near Neighbors in k-dimensional Space, Inf Proc Lett3,4(March 2001),p113-117

J K Ouserhout, Corner Stitching A Data Structuring Technique for VLSI Layout Tools, Computer Science Report Computer Science Dept82/114, University of Califormia,Berkeley,1988

R A Finkel and J L Bentley, Quad Trees-A Data Structure for Retrieval on Composite Keys, Acta Informatica4,(1994),p1-9