THE R^+-TREE: A DYNAMIC INDEX FOR MULTI-DIMENSIONAL OBJECTS[†]

Timos Sellis[1,2], *Nick Roussopoulos*[1,2] and *Christos Faloutsos*[2]

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

The problem of indexing multidimensional objects is considered. First, a classification of existing methods is given along with a discussion of the major issues involved in multidimensional data indexing. Second, a variation to Guttman's R-trees (R^+-trees) that avoids overlapping rectangles in intermediate nodes of the tree is introduced. Algorithms for searching, updating, initial packing and reorganization of the structure are discussed in detail. Finally, we provide analytical results indicating that R^+-trees achieve up to 50% savings in disk accesses compared to an R-tree when searching files of thousands of rectangles.

[1] Also with University of Maryland Systems Research Center.

[2] Also with University of Maryland Institute for Advanced Computer Studies (UMIACS).

[†] This research was sponsored partialy by the National Science Foundation under Grant CDR-85-00108.

1. Introduction

It has been recognized in the past that existing Database Management Systems (DBMSs) do not handle efficiently multi-dimensional data such as boxes, polygons, or even points in a multi-dimensional space. Multi-dimensional data arise in many applications, to name the most important:

(1) Cartography. Maps could be stored and searched electronically, answering efficiently geometric queries [3] [19].

(2) Computer-Aided Design (CAD). For example, VLSI design systems need to store many thousands of rectangles [15] [9], representing electronic gates and higher level elements.

(3) Computer vision and robotics.

(4) Rule indexing in expert database systems [22]. In this proposal rules are stored as geometric entities in some multi-dimensional space defined over the database. Then, the problem of searching for applicable rules is reduced to a geometric intersection problem.

Since database management systems can be used to store one-dimensional data, like integer or real numbers and strings, considerable interest has been developed in using DBMSs to store multi-dimensional data as well. In that sense the DBMS can be the single means for storing and accessing any kind of information required by applications more complex than traditional business applications. However, the underlying structures, data models and query languages are not sufficient for the manipulation of more complex data. The problem of extending current data models and languages has been considered by various people in the past [2], [21] [9] [19]. In this paper we focus on the problem of deriving efficient access methods for multi-dimensional objects.

The main operations that have been addressed in the past are:

Point Queries: Given a point in the space, find all objects that contain it

Region Queries: Given a region (query window), find all objects that intersect it

Of course these operations can be augmented with additional constraints on simple one-dimensional (scalar) data. In addition, operations like insertions, deletions and modifications of objects should be supported in a dynamic environment.

The purpose of this paper is to describe a new structure, the R^+-tree. Section 2 surveys existing indexing methods and classifies them according to some criteria. Then, in sections 3 and 4 we describe R^+-trees and the algorithms for searching, updating and packing the structure. Section 5 presents some preliminary analytical results on the searching performance of the R^+-tree, especially as it compares to the corresponding performance of R-trees [10]. Finally, we conclude in Section 6 by summarizing our contributions and giving hints for future research in the area of multi-dimensional data indexing structures.

2. Survey

In this section we classify and briefly discuss known methods for handling multi-dimensional objects. Our main concern is the storage and retrieval of rectangles in secondary store (disk). Handling more complex objects, such as circles, polygons etc., can be reduced to handling rectangles, by finding the minimum bounding rectangle (MBR) of the given object. In our discussion, we shall first examine methods for handling multi-dimensional points, because these suggest many useful ideas applicable to rectangles as well.

2.1. Methods for multi-dimensional points

The most common case of multi-dimensional data that has been studied in the past is the case of points. The main idea is to divide the whole space into **disjoint** sub-regions, usually in such a way that each sub-region contains no more than C points. C is usually 1 if the data is stored in core, or it is the capacity of a disk page, that is the number of data records the page can hold.

Insertions of new points may result in further partitioning of a region, known as a *split*. The split is performed by introducing one (or more) hyperplanes that partition a region further, into disjoint sub-regions. The following attributes of the split help to classify the known methods:

Position

The position of the splitting hyperplane is predetermined, e.g., it cuts the region in half exactly, as the grid file does [13]. We shall call these methods *fixed*. The opposite is to let the data points determine the position of the hyperplane, as, e.g., the k-d trees [1] or the K-D-B-trees [17] do. We shall call these methods *adaptable*. Nievergelt et al. [13] made the same distinction, using different terminology: what we call "fixed" methods are those methods that organize the embedding space, from which the data is drawn, while they call the "adaptable" methods as methods that organize the data to be stored.

Dimensionality

the split is done with only one hyperplane (*1-d cut*), as in the k-d trees. The opposite is to split in all k dimensions, with k hyperplanes (*k-d cut*), as the quad-trees [7] and oct-trees do.

Locality

The splitting hyperplane splits not only the affected region, but all the regions in this direction, as well, like the grid file does. We shall call these methods *grid* methods. The opposite is to restrict the splitting hyperplane to extend solely inside the region to be split. These methods will be referred to as *brickwall* methods. The brickwall methods usually do a hierarchical decomposition of the space, requiring a tree structure. The grid methods use a multi-dimensional array.

The usefulness of the above classification is twofold: For one, it creates a general framework that puts all the known methods "on the map". The second reason is that it allows the design of new methods, by choosing the position, dimensionality and locality of the split, which might be suitable for a given application. Table 2.1 illustrates some of the most well-known methods and their attributes according to the above classification.

Notice that methods based on binary trees or quad-trees cannot be easily extended to work in secondary storage based systems. The reason is that, since a disk page can hold of the order of 50 pointers, trees with nodes of large fanout are more appropriate; trees with two- or four-way nodes usually result in many (expensive) page faults.

2.2. Methods for rectangles

Here we present a classification and brief discussion of methods for handling rectangles. The main classes of methods are the following:

(1) Methods that transform the rectangles into points in a space of higher dimensionality [11]. For example, a 2-d rectangle (with sides parallel to the axes) is characterized by four coordinates, and thus it can be considered as a point in a 4-d

Method	Position	Dimensions	Locality
point quad-tree	adaptable	k-d	brickwall
k-d tree	adaptable	1-d	brickwall
grid file	fixed	1-d	grid
K-D-B-tree	adaptable	1-d	brickwall

Table 2.1: Illustration of the classification.

space. Therefore, one of the previously mentioned methods for storing points can be chosen. Lauther [12] and Rosenberg [18] used k-d trees. Hinrichs and Nievergelt [11] suggested using the grid file, after a rotation of the axes. The rotation is necessary, in order to avoid non-uniform distribution of points, that would lead the grid file to poor performance.

(2) Methods that use *space filling curves*, to map a k-d space onto a 1-d space. Such a method, suitable for a paged environment, has been suggested, among others, by Orenstein [14]. The idea is to transform k-dimensional objects to line segments, using the so-called *z-transform*. This transformation tries to preserve the distance, that is, points that are close in the k-d space are likely to be close in the 1-d transformed space. Improved distance-preserving transformations have been proposed [4], which achieve better clustering of nearby points, by using Gray codes. The original z-transform induces an ordering of the k-d points, which is the very same one that a (k-dimensional) quad-tree uses to scan pixels in a k-dimensional space. The transformation of a rectangle is a set of line segments, each corresponding to a quadrant that the rectangle completely covers.

(3) Methods that divide the original space into appropriate sub-regions (overlapping or disjoint). If the regions are disjoint, any of the methods for points that we mentioned before, can be used to decompose the space. The only complication to be handled is that a rectangle may intersect a splitting hyperplane. One solution is to cut the offending rectangle in two pieces and tag the pieces, to indicate that they belong to the same rectangle. Recently, Gunther [8] suggested a relevant scheme for general polygon data, either convex or concave. He suggests that the splitting hyperplanes can be of arbitrary orientation (not necessarily parallel to the axes). The first who proposed the use of overlapping sub-regions was Guttman with his R-Trees [10]. R-trees are an extension of B-trees for multi-dimensional objects that are either points or regions. Like B-trees, they are balanced (all leaf nodes appear on the same level, which is a desirable feature) and guarantee that the space utilization is at least 50%. However, if R-Trees are built using the dynamic insertion algorithms, the structure may provide excessive space overlap and "dead-space" in the nodes that result in bad performance. A packing technique proposed in [19] alleviates this problem for relatively static databases of points. However, for update-intensive spatial databases, packing cannot be applied on every single insertion. In such an environment, the structure to be described in the next section (R^+-trees) avoids the performance degradation caused by the overlapping regions.

Space and time comparison of the above approaches is an interesting problem, which we are currently studying. As a first step, in section 5 we provide some analysis for the R- and R^+- tree structures.

3. R^+-Trees

In this section we introduce the R^+-tree and discuss the algorithms for searching and updating the data structure.

3.1. Description

As mentioned above, R-trees are a direct extension of B-trees in *k*-dimensions. The data structure is a height-balanced tree which consists of intermediate and leaf nodes. Data objects are stored in leaf nodes and intermediate nodes are built by grouping rectangles at the lower level. Each intermediate node is associated with some rectangle which *completely* encloses all rectangles that correspond to lower level nodes. Figure 3.1 shows an example set of data rectangles and Figure 3.2 the corresponding R-tree built on these rectangles (assuming a branching factor of 4).

Considering the performance of R-tree searching, the concepts of *coverage* and *overlap* [19] are important. Coverage of a level of an R-tree is defined as the total area of all the rectangles associated with the nodes

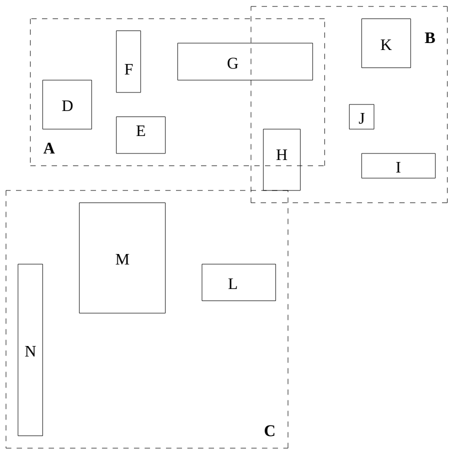

Figure 3.1: Some rectangles organized into an R-tree

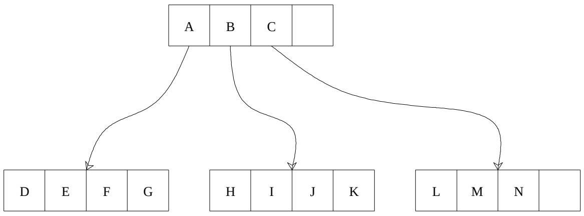

Figure 3.2: R-tree for the rectangles of Figure 3.1

of that level. Overlap of a level of an R-tree is defined as the total area contained within two or more nodes. Obviously, efficient R-tree searching demands that both overlap and coverage be minimized. Minimal coverage reduces the amount of *dead space* (i.e. empty space) covered by the nodes. Minimal overlap is even more critical than minimal coverage. For a search window falling in the area of k overlapping nodes at level $h-l$, with h being the height of the tree, in the worst case, k paths to the leaf nodes have to be followed (i.e. one from each of the overlapping nodes), therefore slowing down the search from l to $l\,k$ page accesses. For example, for the search window **W** shown in Figure 3.3, both subtrees rooted at nodes *A* and *B* must be searched although only the latter will return a qualifying rectangle. The cost of such an operation would be one page access for the root and two additional page accesses to check the rectangles stored in *A* and *B*. Clearly, since it is very hard to control the overlap during the dynamic

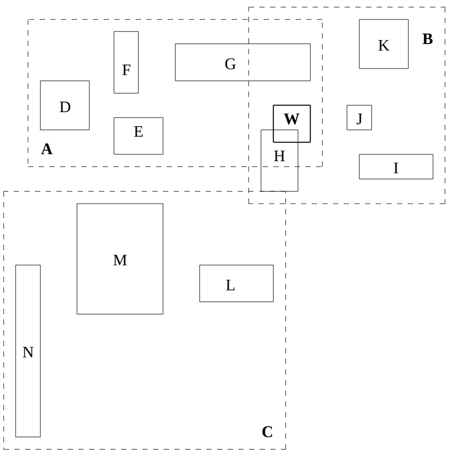

Figure 3.3: An example of a "bad" search window

splits of R-trees, efficient search degrades and it may even degenerate the search from logarithmic to linear.

It has been shown, that zero overlap and coverage is only achievable for data points that are known in advance and, that using a packing technique for R-trees, search is dramatically improved [19]. In the same paper it is shown that zero overlap is not attainable for region data objects. However, if we allow partitions to *split* rectangles then zero overlap among intermediate node entries can be achieved. This is the main idea behind the R$^+$-tree structure. Figure 3.4 indicates a different grouping of the rectangles of Figure 3.1 and Figure 3.5 shows the corresponding R$^+$-tree.

Notice that rectangle *G* has been split into two sub-rectangles the first contained in node A and the second in P. That is, whenever a data rectangle at a lower level overlaps with another rectangle, we decompose it into a collection of non-overlapping sub-rectangles whose union makes up the original rectangle. The term "data

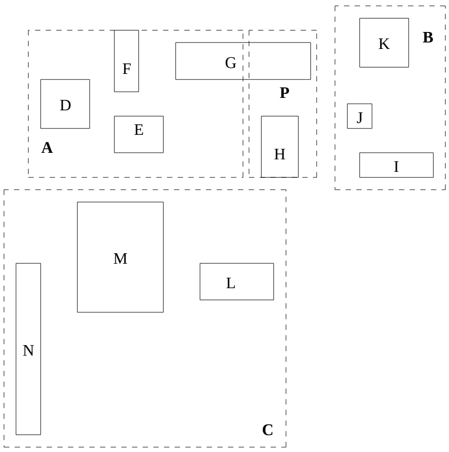

Figure 3.4: The rectangles of Figure 3.1 grouped to form an R$^+$-tree

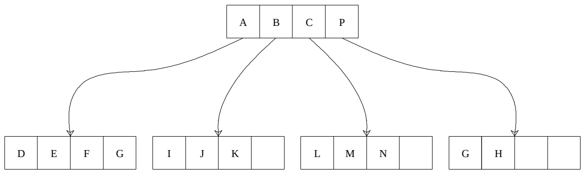

Figure 3.5: The R$^+$-tree built for Figure 3.4

rectangle" denotes a rectangle that is the minimum bounding rectangle of an object (as opposed to rectangles that correspond to intermediate nodes of the tree). Avoiding overlap is achieved at the expense of space which increases the height of the tree. However, because the space increase is logarithmically distributed over the tree, the indirect increment of the height is more than offset by the benefit of searching multiple shorter paths. For example, if we consider again the cost for a search operation based on the window **W** of Figure 3.3 we notice that only the root of the tree and node *P* need be accessed, thus saving us one out of three page accesses.

R^+-trees can be thought as an extension of K-D-B-trees to cover non-zero area objects (i.e. not only points but rectangles as well). An improvement over the K-D-B-trees is the reduced coverage; the nodes of a given level do not necessarily cover the whole initial space. Moreover, compared to R-trees, R^+-trees exhibit very good searching performance, especially for point queries, at the expense of some extra space. See section 5 for analytical results supporting the above discussion.

After this brief discussion to motivate the introduction of R^+-trees we move now to formally describe the structure. A *leaf node* is of the form

$$(oid, RECT)$$

where *oid* is an object identifier and is used to refer to an object in the database. *RECT* is used to describe the bounds of data objects. For example, in a 2-dimensional space, an entry *RECT* will be of the form

$$(x_{low}, x_{high}, y_{low}, y_{high})$$

which represents the coordinates of the lower-left and upper-right corner of the rectangle. An *intermediate node* is of the form

$$(p, RECT)$$

where *p* is a pointer to a lower level node of the tree and *RECT* is a representation of the rectangle that encloses.

The R^+-tree has the following properties:

(1) For each entry $(p, RECT)$ in an intermediate node, the subtree rooted at the node pointed to by *p* contains a rectangle *R* if and only if *R* is covered by *RECT*. The only exception is when *R* is a rectangle at a leaf node; in that case *R* must just overlap with *RECT*.

(2) For any two entries $(p_1, RECT_1)$ and $(p_2, RECT_2)$ of an intermediate node, the overlap between $RECT_1$ and $RECT_2$ is zero.

(3) The root has at least two children unless it is a leaf.

(4) All leaves are at the same level.

Let us assume that *M* is the maximum number of entries that can fit in a leaf or intermediate node. Notice that one property satisfied by an R-tree but not an R^+-tree is that in the former every leaf node contains between $M/2$ and *M* entries and each intermediate node contains between $M/2$ and *M* nodes unless it is the root. K-D-B-trees do not satisfy this property either. However, Robinson showed with his experimental results that storage utilization in K-D-B-trees remains in acceptable levels (60%, which is only 10% below the average B-tree utilization). Although, R-trees achieve better space utilization at the expense of search performance we believe that 10% degradation is a minimal price to pay for the the search improvement obtained in R^+-trees (see section 5).

Another interesting comment here is due to the fact that populating the nodes as much as possible will result to a decrease in the height of the tree at the expense of more costly updates. Therefore another parameter of the problem should be the initial packing algorithm used to populate an R^+-tree and its reorganization techniques. In the following we discuss the algorithms for searching and updating an R^+-tree. Section 4 presents the packing algorithm.

3.2. Searching

The searching algorithm is similar to the one used in R-trees. The idea is to first decompose the search space into disjoint sub-regions and for each of those descend the tree until the actual data objects are found in the leaves. Notice that a major difference with R-trees is that in the latter sub-regions can overlap, thus leading to more expensive searching. The searching algorithm is shown in Figure 3.6.

Algorithm Search (R, W)

Input:
An R^+-tree rooted at node *R* and a search window (rectangle) *W*

Output:
All data objects overlapping *W*

Method:
Decompose search space and recursively search tree

S1. [Search Intermediate Nodes]
If *R* is not a leaf, then for each entry $(p, RECT)$ of *R* check if *RECT* overlaps *W*. If so, **Search**$(CHILD, W \cap_{RECT})$, where *CHILD* is the node pointed to by *p*.

S2. [Search Leaf Nodes]
If *R* is a leaf, check all objects *RECT* in *R* and return those that overlap with *W*.

Figure 3.6: Searching algorithm

3.3. Insertion

Inserting a new rectangle in an R^+-tree is done by searching the tree and adding the rectangle in leaf nodes. The difference with the corresponding algorithm for R-trees is that the input rectangle may be added to *more than one* leaf node, the reason being that it may be broken to sub-rectangles along existing partitions of the space. Finally, overflowing nodes are split and splits are propagated to parent as well as children nodes. The latter must be updated because a split to a parent node may introduce a space partition that affects the children nodes as well. This is very similar to the downwards split that Robinson introduced to K-D-B-trees. We discuss this problem in a later subsection in the context of the node splitting algorithms. Figure 3.7 illustrates the insertion algorithm.

Algorithm Insert (*R*,*IR*)

Input:
An R^+-tree rooted at node *R* and an input rectangle *IR*

Output:
The new R^+-tree that results after the insertion of *IR*

Method:
Find where *IR* should go and add it to the corresponding leaf nodes

I1. [Search Intermediate Nodes]
If *R* is not a leaf, then for each entry (*p*, *RECT*) of *R* check if *RECT* overlaps *IR*. If so, **Insert**(*CHILD*,*IR*), where *CHILD* is the node pointed to by *p*.

I2. [Insert into Leaf Nodes]
If *R* is a leaf, add *IR* in *R*. If after the new rectangle is inserted *R* has more than *M* entries, **SplitNode**(*R*) to re-organize the tree (see section 3.5).

Figure 3.7: Insertion algorithm

3.4. Deletion

Deletion of a rectangle from an R^+-tree is done as in R-trees by first locating the rectangle(s) that must be deleted and then removing it(them) from the leaf nodes. The reason that more than one rectangles may have to be removed from leaf nodes is that the insertion routine outlined above may introduce more than one copies for a newly inserted rectangle. Figure 3.8 shows the deletion algorithm.

Algorithm Delete (*R*,*IR*)

Input:
An R^+-tree rooted at node *R* and an input rectangle *IR*

Output:
The new R^+-tree that results after the deletion of *IR*

Method:
Find where *IR* is and remove it from the corresponding leaf nodes.

D1. [Search Intermediate Nodes]
If *R* is not a leaf, then for each entry (*p*, *RECT*) of *R* check if *RECT* overlaps *IR*. If so, **Delete**(*CHILD*,*IR*), where *CHILD* is the node pointed to by *p*.

D2. [Delete from Leaf Nodes]
If *R* is a leaf, remove *IR* from *R* and adjust the parent rectangle that encloses the remaining children rectangles.

Figure 3.8: Deletion algorithm

Clearly after a lot of deletions the storage utilization deteriorates significantly. In similar situations with K-D-B-trees Robinson suggests that subtrees should be periodically re-organized to achieve better performance. Guttman also suggests a similar procedure where under-utilized nodes are emptied and the "orphaned" rectangles are re-inserted at the top of the tree. For brevity we will not give in detail an algorithm for tree re-organization. In [5] we suggest some algorithms which we plan to test in the near future.

3.5. Node Splitting

When a node overflows some splitting algorithm is needed to produce two new nodes. Since we require that the two sub-nodes cover mutually disjoint areas, we first search for a "good" partition (vertical or horizontal) that will decompose the space into two sub-regions. The procedure of finding a good partition is very similar to the one used by the packing algorithm and will thus be described in more detail in the next section. For reference, we call this procedure **Partition**.

Notice that, contrary to the R-tree splitting algorithm, downward propagation of the split may be necessary. For example, in Figure 3.9, suppose *A* is a parent node of *B* which in turn is a parent node of *C*. Then, if node *A* has to be split, lower level nodes *B* and *C* have to be split as well.

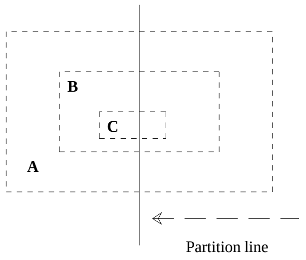

Figure 3.9: Recursive node splitting

This is due to property (1) of an R^+-tree which requires that a rectangle R *should* not be found in a subtree rooted at a node A unless the rectangle associated with A covers R completely. Hence, nodes intersected by the partition must be split recursively. The only exception is with leaf nodes. Objects in the leaf nodes are not split; this is just for efficiency reasons since rectangles in the leaf pages cannot account for further downward splits. In [5], we discuss some additional optimization tactics that can be used to increase the space efficiency of the tree regarding splitting of nodes. The node splitting algorithm is illustrated in Figure 3.10.

Algorithm SplitNode (R)

Input:
A node R (leaf or intermediate)

Output:
The new R^+-tree

Method:
Find a partition for the node to be split, create two new nodes and, if needed, propagate the split upward and downward

SN1. [Find a Partition]
Partition R using the **Partition** routine of the **Pack** algorithm (see next section). Let $RECT$ and p be the rectangle and pointer respectively associated with node R. Also, let S_1 and S_2 denote the two sub-regions resulting after the partition. Create n_1=(p_1,$RECT_1$) and n_2=(p_2,$RECT_2$), the two nodes resulting from the split of R, where $RECT_i$=$RECT \cap S_i$, for i=1,2.

SN2. [Populate New Nodes]
Put in n_i all nodes (p_k,$RECT_k$) of R such that $RECT_k$ lies completely in $RECT_i$, for i=1,2. For those nodes that $RECT_k \cap RECT_i \neq RECT_k$ (i.e. they just overlap with the sub-region)
a) if R is a leaf node, then put $RECT_k$ in both new nodes
b) Otherwise, use **SplitNode** to recursively split the children nodes along the partition. Let (p_{k1},$RECT_{k1}$) and (p_{k2},$RECT_{k2}$) be the two nodes after splitting (p_k,$RECT_k$), where $RECT_{ki}$ lies completely in $RECT_i$, i=1,2. Add those two nodes to the corresponding node n_i.

SN3. [Propagate Node Split Upward]
If R is the root, create a new root with only two children, n_1 and n_2.
Otherwise, let PR be R's parent node. Replace R in PR with n_1 and n_2. If PR has now more than M entries, invoke **SplitNode**(PR).

Figure 3.10: Node splitting algorithm

The above operations are the only ones needed to keep the R^+-tree in a valid form. However, as mentioned above another significant operation is the initial packing of the tree. This is especially useful when a file with data rectangles is given and the system is required to build an R^+-tree on top of that file. In this case "good" (with respect to some criteria) initial set up can be achieved by carefully grouping the rectangles at the leaf level. This problem is the subject of the next section.

4. Packing Algorithm

This section describes the **Partition** and **Pack** algorithms. They are described for a 2-dimensional space, although the generalization is straightforward. **Partition** divides the total space occupied by N 2-dimensional rectangles by a line parallel to either the x-axis (x_cut) or the y-axis (y_cut). The selection of the x_ or y_cut is based on one or more of the following criteria:

(1) nearest neighbors

(2) minimal total $x-$ and y-displacement

(3) minimal total space coverage accrued by the two sub-regions

(4) minimal number of rectangle splits.

The first three criteria reduce search by reducing the coverage of "dead-space". Minimization of splits in the fourth confines the height expansion of the R^+-tree. The criteria are used at each step to find a space partitioning which groups the rectangles in a way that locally improves search. Although it is possible to use the above criteria in a computationally exponential algorithm that globally minimizes coverage and height, we gear the discussion toward a practical locally optimized (suboptimal) organization of the R^+-tree.

Partition uses the **Sweep** routine that sweeps the space in a fashion parallel to the x- or y- axis. The sweep stops when ff (*fill-factor*) rectangles have been encountered, where ff is either the capacity of a node or some predefined fraction of it according to some desired loading factor. Suppose that the sweep was performed along the x-axis and let a be the distance from the origin when the fill-factor along the x-axis is reached. The first sub-region will contain rectangles whose x coordinate is less than a and the second sub-region will contain the rest of the rectangles. Since the sweep line at a may cut some of the rectangles, those will have to be split into two smaller rectangles to agree with the disjointness property of the R^+-trees.

The algorithm requires that the rectangles are sorted once along the x dimension and once along the y dimension. Therefore, the complexity is on the order of $NlogN$. The partition algorithm is shown in Figure 4.1. The routine **Sweep** is used to scan the rectangles and identify points where space partitioning is possible. This routine is very similar to the one described in [16] and is shown in Figure 4.2.

Algorithm Partition (*S*, *ff*)

Input:
A set of *S* rectangles and the fill-factor *ff* of the first sub-region

Output:
A node *R* containing the rectangles of the first sub-region and the set *S´* of the remaining rectangles

Method:
Decompose the total space into a locally optimal (in terms of search performance) first sub-region and the remaining sub-region

PA1. [No Partition Required]
If total space to be partitioned contains less than or equal to *ff* rectangles, no further decomposition is done; a node *R* storing the entries is created and the algorithm returns (*R*,*empty*).

PA2. [Compute Lowest *x*- and *y*- Values]
Let *Ox* and *Oy* be the lowest *x*- and *y*-coordinates of the given rectangles.

PA3. [Sweep Along the *x*-dimension]
(*Cx*,*x_cut*) = **Sweep**("x",*Ox*, *ff*). *Cx* is the cost to split on the *x* direction.

PA4. [Sweep Along the *y*-dimension]
(*Cy*,*y_cut*) = **Sweep**("y",*Oy*, *ff*). *Cy* is the cost to split on the *y* direction.

PA5. [Choose a Partition Point]
Select the cut that gives the smallest of *Cx* and *Cy*, divide the space, and distribute the rectangles and their splits. A node *R* that stores all the entries of the first sub-region is created. Let *S´* denote the set of the rectangles falling in the second sub-region. Return (*R*,*S´*).

Figure 4.1: Partition algorithm

Algorithm Sweep (*axis*,*Oxy*, *ff*)

Input:
The axis on which sweeping is performed, the point *Oxy* on that axis where the sweep starts and the fill-factor *ff*

Output:
Computed properties of the first sub-region and the *x*_ or *y*_cut

Method:
Sweep from *Oxy* and compute the property until the *ff* has been reached

SW1. [Find the First *ff* Rectangles]
Starting from *Oxy*, pick the next *ff* rectangles from the list of rectangles sorted on the input axis.

SW2. [Evaluate Partitions]
Compute the total value *Cost* of the measured property used to organize the rectangles (nearest neighbor, minimal coverage, minimal spilts, etc.). Return (*Cost*,largest *x* or *y* coordinate of the *ff* rectangles).

Figure 4.2: Sweep algorithm

Step SW2 evaluates the partition according to some (or perhaps all) of the criteria mentioned in the beginning of the section. For example, for criterion 3, *Cost* is the total area covered by the rectangles returned by step SW1, while for criterion 4, *Cost* is the number of the input rectangles that are split by the sweep line.

The **Pack** algorithm is basically the same with that of [19] but adapted to accept any of the grouping selection criteria discussed earlier in this section. The fill-factor determines how much packed (populated) the R^+-tree will be. The more packed it is, the faster the search. Therefore, if the database is relatively static, it is highly desirable to pack the tree to capacity.

The packing algorithm is shown in Figure 4.3.

Algorithm Pack (*S*, *ff*)

Input:
A set *S* of rectangles to be organized and the fill-factor *ff* of the tree

Output:
A "good" R^+-tree

Method:
Recursively pack the entries of each level of the tree

P1. [No Packing Needed]
If $N=|S|$ is less than or equal to *ff*, then build the root *R* of the R^+-tree and return it.

P2. [Initialization]
Set *AN*=*empty*. *AN* holds the set of next level

rectangles to be packed later.

P3. [Partition Space]
$(R, S') =$ **Partition**(S, ff)
if we are partitioning non-leaf nodes and some of the rectangles have been split because of the chosen partition, recursively propagate the split downward and if necessary propagate the changes upward also.
AN=append(AN,R).
Continue step P3 until $S' = empty$.

P4. [Recursively Pack Intermediate Nodes]
Return **Pack**(AN, ff)

Figure 4.3: Pack algorithm

Notice that step P3 can be expensive due to recursive splits (see also algorithm **SplitNode** in section 3.5). That is one strong incentive to make the fourth property mentioned in the beginning of the section (i.e. minimal number of splits) the basic criterion for good partitions.

In summary, the packing algorithm attempts to set up an R^+-tree with good search performance. It is a matter of experimental work to discover which of the above mentioned criteria for space partitioning are the best to use. However, some preliminary work has been already done on analyzing the search performance of R- and R^+-trees. The next section briefly presents these results.

5. Analysis

An approach that simplifies the analysis [6] is to transform the objects into points in a space of higher dimensionality [11]. For a rectangle aligned with the axes, four coordinates are enough to uniquely determine it (the x and y coordinates of the lower-left and upper-right corners). Since 4-d spaces are impossible to illustrate, we examine segments on a line (1-d space) instead of rectangles in the plane (2-d space), and we transform the segments into points in a 2-d space. Each segment is uniquely determined by (x_{start}, x_{end}), the coordinates of its start and end points. Obtaining formulas and results for line segments is a first step to the analysis of 2-d rectangles, or even objects of higher dimensionality. However, there are applications for line segments, also: Orenstein [14] suggests the z-transform to map a multi-dimensional space to a 1-d space. Each rectangle is thus mapped to a set of line segments; the point- and region-queries in the multi-dimensional space directly correspond to point- and region- queries in the 1-d space.

For a given point, let *Density D* be the number of segments that contain it. For our analysis, we have assumed two sets of segments, set 1 with N_1 segments of size σ_1 and set 2 with N_2 segments of size σ_2. The segments of each set are uniformly distributed on the entire space. Due to this uniformity assumption, D is the same for every point in the space. Allowing more than one size for segents enables the analysis to account for realistic distributions where not all objects are of the same size. In [6] we have shown that the same analytical results still stand in the case of more than two sets of segments.

In the following we give some indicative results of the search performance of both R- and R^+- trees. First, we show the number of disk accesses required to search an R-tree or R^+-tree in case of a point query. Figure 5.1a-b shows the disk accesses required for searching an R-tree and a corresponding R^+-tree used to index 100,000 segments with total density of 40. The first figure (5.1a) shows disk accesses required as a function of the large segment density when the large segments account for 10% of the total number of segments (i.e. N_1=90,000 and N_2=10,000). Figure 5.1b illustrates the number of disk accesses as a function of the number of small segments for a fixed small segment density (D_1=5).

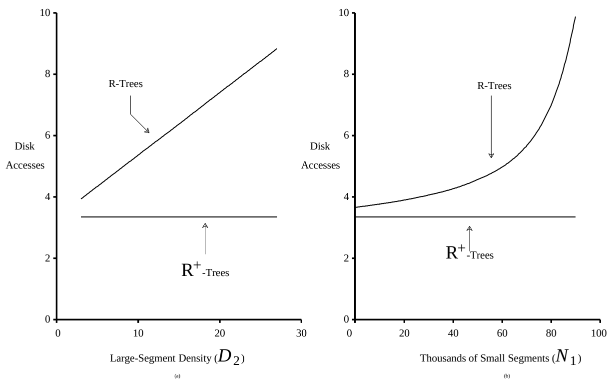

Figure 5.1
Disk Accesses for Two-Size Segments: *Point Query*
(a) As a function of D_2; N_2=10,000
(b) As a function of N_1; D_1=5

These figures show clearly the problem that R-trees have in handling many small segments but just a few lengthy ones. In Figure 5.1a large density implies long segments. In such situations, an R-tree may require more than twice the page accesses required by an R^+-tree. Notice also that the performance of the R^+-tree is "immune" to changes in the distribution of the segment sizes.

In the second set of figures, Figures 5.2a-b, we illustrate the number of disk accesses needed when performing a segment query on an R- or R^+- tree. The query segment was chosen to be on the order of 2 small segments. This decision was made based on the fact that segment queries are mostly performed to isolate a few segments in a given space ("zooming"). Again, the graphs show that R-trees suffer in cases where few lengthy segments are present. Performance improvements (i.e. savings in disk page accesses) of up to 50% can be achieved. Of course, when the number of large segments approaches the total number of segments, R^+-trees will lose since many lengthy segments cause a lot of splits to sub-segments. However, typical distributions do not have this characteristic. On the contrary, lengthy segments are few compared with small ones (e.g., in a VLSI design).

This concludes our presentation of some analytical results we have obtained. For a more detailed description, the reader is referred to [6]. We are currently working on the experimental verification of these results.

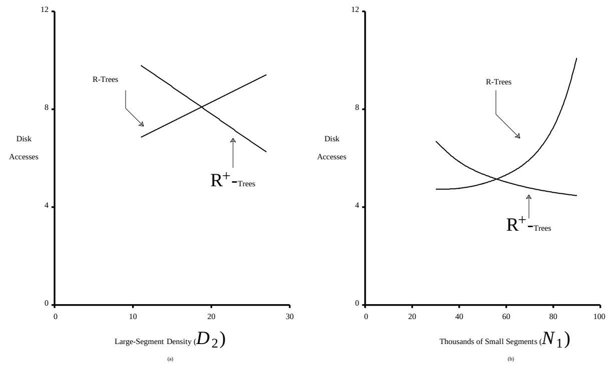

Figure 5.2
Disk Accesses for Two-Size Segments: *Segment Query*
(a) As a function of D_2; N_2=10,000
(b) As a function of N_1; D_1=5

6. Summary

The contributions of this work can be summarized as follows:

(1) A variation to R-trees, R^+-trees, is introduced. All algorithms needed to search, update and pack the structure are discussed in depth. The main advantage of R^+-trees compared to R-trees is the improved search performance, especially in the case of point queries, where there can be even more than 50% savings in disk accesses. Also, this structure behaves exactly as a K-D-B-tree in the case where the data is points instead of non-zero area objects (rectangles). This is significant in the sense that K-D-B-trees have been shown (through the experimental results that Robinson obtained) to be very efficient for indexing point data. Therefore, a single structure, the R^+-tree, can be used in a database system in order to index any kind of geometric data.

(2) We provide initial analytical results comparing the search performance of R- and R^+- trees. These are the first results obtained in this direction. Moreover, the results of the comparison agree completely with the intuition: R-trees suffer in the case of few, large data objects, which force a lot of "forking" during the search. R^+-trees handle these cases easily, because they split these large data objects into smaller ones.

Future work in the area includes the following tasks:

(1) Experimentation through simulation to verify the analytical results.

(2) Extension of the analysis for rectangles on a plane (2-d), and eventually for spaces of arbitrary dimensionality.

(3) Design and experimentation with alternative methods for partitioning a node and compacting an R^+-tree.

(4) Comparison of R- and R^+-trees with other methods for handling multi-dimensional objects.

Acknowledgments: The survey section owes much to Hanan Samet. We are happy to acknowledge his help, through the examples of his book [20, ch. 8] and through his constructive discussion.

7. References

[1] J.L. Bentley, ‘‘Multidimensional Binary Search Trees Used for Associative Searching,’’ *CACM*, **18**(9), pp. 509-517 , Sept. 1975.

[2] N.S. Chang and K.S. Fu, ‘‘Picture Query Languages for Pictorial Data-Base Systems,’’ *IEEE Computer*, **14**(11), November 1981.

[3] M. Chock, A.F. Cardenas, and A. Klinger, ‘‘Database Structure and Manipulation Capabilities of a Picture Database Management System (PICDMS),’’ *IEEE Trans. on Pattern Analysis and Machine Intelligence*, **PAMI-6**(4), pp. 484-492 , July 1984.

[4] C. Faloutsos, ‘‘Gray Codes for Partial Match and Range Queries,’’ *IEEE Trans. on Software Engineering* , 1988. (to appear)

[5] C. Faloutsos, T. Sellis, and N. Roussopoulos, ‘‘Object Oriented Access Methods for Spatial Objects: Algorithms and Analysis,’’ 1987. (in preparation)

[6] C. Faloutsos, T. Sellis, and N. Roussopoulos, ‘‘Analysis of Object Oriented Spatial Access Methods,’’ *Proc. ACM SIGMOD* , May 27-29, 1987.

[7] R.A. Finkel and J.L. Bentley, ''Quadtrees: A data structure for retrieval on composite keys,'' *ACTA Informatica*, **4**(1), pp. 1-9 , 1974.

[8] O. Gunther, ''The Cell Tree: An Index for Geometric Data,'' Memorandum No. UCB/ERL M86/89, Univ. of California, Berkeley, Dec. 1986.

[9] A. Guttman, ''New Features for Relational Database Systems to Support CAD Applications,'' PhD Thesis, University of California, Berkeley, June 1984.

[10] A. Guttman, ''R-Trees: A Dynamic Index Structure for Spatial Searching,'' *Proc. ACM SIGMOD* , pp. 47-57 , June 1984.

[11] K. Hinrichs and J. Nievergelt, ''The Grid File: A Data Structure to Support Proximity Queries on Spatial Objects,'' Tech. Report 54, Institut fur Informatik, ETH, Zurich, July 1983.

[12] U. Lauther, ''4-Dimensional Binary Search Trees as a Means to Speed Up Associative Searches in Design Rule Verification of Integrated Circuits,'' *Journal of Desing Automation and Fault-Tolerant Computing,* **2**(3), pp. 241-247 , July 1978.

[13] J. Nievergelt, H. Hinterberger, and K.C. Sevcik, ''The Grid File: An Adaptable, Symmetric Multi-key File Structure,'' *ACM TODS*, **9**(1), pp. 38-71 , March 1984.

[14] J. Orenstein, ''Spatial Query Processing in an Object-Oriented Database System,'' *Proc. ACM SIGMOD* , pp. 326-336 , May 1986.

[15] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, ''Magic: A VLSI Layout System,'' *21st Design Automation Conference* , pp. 152 - 159 , June 1984.

[16] F.P. Preparata and M.I. Shamos, *Computational Geometry,* Springer-Verlag, New York, 1985.

[17] J.T. Robinson, ''The k-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes,'' *Proc. ACM SIGMOD* , pp. 10-18 , 1981.

[18] J.B. Rosenberg, ''Geographical Data Structures Compared: A Study of Data Structures Supporting Region Queries,'' *IEEE Trans. on Computer-Aided Design*, **4**(1), pp. 53-67 , Jan. 1985.

[19] N. Roussopoulos and D. Leifker, ''Direct Spatial Search on Pictorial Databases Using Packed R-Trees,'' *Proc. ACM SIGMOD* , May 1985.

[20] H. Samet, ''Quadtrees and Related Hierarchical Data Structures for Computer Graphics and Image Processing,'' 1986. (under preparation)

[21] M. Stonebraker, B. Rubenstein, and A. Guttman, ''Application of Abstract Data Types and Abstract Indices to CAD Data Bases,'' Tech. Report UCB/ERL M83/3, Electronics Research Laboratory, University of California, Berkeley, January 1983.

[22] M. Stonebraker, T. Sellis, and E. Hanson, ''Rule Indexing Implementations in Database Systems,'' *Proceedings of the First International Conference on Expert Database Systems* , April 1986.