

MULTI-R TREE: AN EFFICIENT INDEX STRUCTURE  
FOR MULTI-DIMENSIONAL OBJECTS

BY

KAP SAN BANG

Bachelor of Science

Chung-Ang University

Seoul, Korea

1987

Submitted to the Faculty of the  
Graduate College of the  
Oklahoma State University  
in partial fulfillment of  
the requirements for  
the degree of  
MASTER OF SCIENCE  
May, 1992

Thesis  
1992  
Bairm

MULTI-R TREE: AN EFFICIENT INDEX STRUCTURE  
FOR MULTI-DIMENSIONAL OBJECTS

Thesis Approved:

*Huizhu Lu*

Thesis Adviser

*D. E. Helms*

*D. W. Fisher*

*Thomas C. Collins*

Dean of the Graduate College

## ACKNOWLEDGMENT

I wish to express my sincere appreciation to Dr. Huizhu Lu for her kindness and advice throughout my graduate program. Also I would like to give Dr. Donald D. Fisher and Dr. George E. Hedrick many thanks for serving on my graduate committee. Their suggestions and corrections really helped me throughout the research.

I want to say thanks to Dr. Keith A. Teague for giving me an Hyper Cube account. Without Hyper Cube parallel implementation could not been made. I appreciate to Sei-Hun Chun who introduced me this field.

To my parents, Cheon-Keun Bang and Chun-Sook Kang, I really appreciate for their support and love. I would like to show my work to my father who passed away one year ago and to say thanks. My wife, Hey-Sun Kim, gave me a lot of support. To my family I extend my sincere thanks.

## TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION . . . . .	1
II. LITERATURE REVIEW . . . . .	3
R-tree . . . . .	4
Structure . . . . .	4
Operations . . . . .	7
Searching . . . . .	7
Insertion . . . . .	7
Deletion . . . . .	8
Splitting . . . . .	8
Exhaustive Algorithm . . . . .	9
Quadratic Cost Algorithm . . . . .	9
Linear Cost Algorithm . . . . .	9
Conclusion . . . . .	10
R <sup>+</sup> -tree . . . . .	13
Structure . . . . .	15
Operations . . . . .	16
Searching . . . . .	16
Insertion . . . . .	16
Deletion . . . . .	17
Node Splitting . . . . .	17
Conclusion . . . . .	18
III. MULTI-R TREE . . . . .	21
Introduction . . . . .	21
Structure . . . . .	23
Operations . . . . .	29
Searching . . . . .	29
Insertion . . . . .	31
Deletion . . . . .	36
Node Splitting . . . . .	38

Chapter	Page
IV. ANALYSIS PREVIEW . . . . .	44
Introduction. . . . .	44
Assumptions. . . . .	45
Analysis of R <sup>+</sup> -tree. . . . .	49
One-size case. . . . .	52
Two-size case . . . . .	53
Analysis of R-tree . . . . .	54
One-size case. . . . .	54
Two-size case . . . . .	56
Analytical result . . . . .	58
One-size case. . . . .	58
Two-size case . . . . .	60
V. ANALYSIS OF MULTI-R TREE. . . . .	63
Introduction. . . . .	63
Point Query . . . . .	64
One-size case. . . . .	64
Two-size case . . . . .	70
Range Query. . . . .	77
One-size case. . . . .	77
Two-size case . . . . .	82
Parallel processing. . . . .	86
VI. SUMMARY, CONCLUSION , AND SUGGESTED FUTURE WORK . . . . .	94
REFERENCES . . . . .	103

## LIST OF FIGURES

Figure	Page
1. Sample Rectangle Arrangement and the R-tree Structure . . . . .	6
2. Bad (a) and Good (b) Splits . . . . .	9
3. Linear-Cost Algorithm . . . . .	10
4. Search Performance vs. Amount of Data: Pages Touched . . . . .	11
5. Search Performance vs. Amount of Data: CPU Cost . . . . .	12
6. Space Required for the R-tree vs. Amount of Data. . . . .	12
7. Rectangle Arrangement. . . . .	14
8. The $R^+$ -tree for Figure 7 . . . . .	14
9. Rectangles on Partition Line . . . . .	18
10. Disk Accesses for the Segments of Two Different SIZES Using Point Query As a Function of $D_2$ , $N_2 = 10,000$ . . . . .	20
11. As a Function of $N_1$ , $D_1 = 5$ . . . . .	20
12. Disk Accesses for the Segments of Two Different SIZES Using Segment Query As a Function $D_2$ , $N_2 = 10,000$ . . . . .	20
13. Disk Accesses for the Segments of Two Different SIZES Using Segment Query As a Function $N_1$ , $D_1 = 5$ . . . . .	20
14. Rectangles in the 1st Subscreen of the MR-tree . . . . .	22

Figure	Page
15. Rectangle in the 2nd Subscreen of the MR-tree . . . . .	22
16. (a) The 1st Subtree for Figure 14 (b) The 2nd Subtree for Figure 15. . . . .	23
17. An Organization of Rectangles into the R <sup>+</sup> -tree . . . . .	24
18. R <sup>+</sup> -tree Structure for the Rectangles in Figure 17. . . . .	25
19. Organized Rectangles on the 1st Subscreen . . . . .	26
20. The 1st Subtree Structure of the MR-tree Corresponding to Figure 19 . . . . .	26
21. Organized Rectangles on the 2nd Subscreen . . . . .	27
22. The 2nd Subtree Structure of the MR-tree Corresponding to Figure 21 . . . . .	27
23. Rectangle Arrangement & Partition Sections . . . . .	42
24. (a) The 1st Subscreen for Figure 23 . . . . . (b) The 2nd Subscreen for Figure 23 . . . . .	42 43
25. Line Segments on the Screen . . . . .	45
26. The 2-d Representation of the Line Segments for Figure 25. . . . .	46
27. The Shaded Area Corresponds to Segments Containing the Point $x_0$ . . . . .	47
28. The Shaded Area Corresponds to Segments Intersecting with the Segment $(x_1, x_2)$ . . . . .	47
29. The Shaded Area corresponds to Segments Covered Completely by the Segment $(x_1, x_2)$ . . . . .	48
30. N= 5 Segments of Size $a = .25$ , Uniformly Distributed on the Screen . . . . .	48



Figure	Page
31. The 2-d Transformation of the Segments in Figure 30 . . . . .	49
32. N Segment of Size a (Represented as Points), Uniformly Distributed on the Screen . . . . .	51
33. Segments of Size .5 each: Overlap ( $O_v$ )= 2 . . . . .	52
34. Illustration of Father of Level 1 ( $C= 4$ ) . . . . .	54
35. Disk Access for One Size Segments, as a Function of $O_v$ ; $N= 100,000$ . . . . .	59
36. Disk Access for One Size Segments, as a Function of $N$ ; $O_v= 40$ . . . . .	59
37. Performance Gain for One Size Segments as a Function of $O_v$ ; $N= 100$ to $1,000,000$ . . . . .	60
38. Disk Accesses for Two Size Segments as a Function of $O_v$ ; $N_2= 10,000$ . . . . .	61
39. Disk Accesses for Two Size Segments as a Function of $N_2$ ; $O_{v1}= 5$ . . . . .	61
40. Performance Gain for Two Size Segments as a Function of $O_{v1}$ ; $N_2= 10,000$ to $100,000$ & $N= 100,000$ . . . . .	62
41. Uniformly Distributed Line Segments of Size a . . . . .	64
42. Data Pages When Using the $R^+$ -tree . . . . .	65
43. Overlap of Data Objects between Nodes of the $R^+$ -tree . . . . .	65
44. One Size Case Screen Division . . . . .	66
45. Magnified Picture of Figure 44 . . . . .	66
46. Illustrates the Splitting Algorithm of MR-tree . . . . .	67
47. The 1st Subscreen of the MR-tree for Figure 41 . . . . .	68

Figure	Page
48. Arrangement of Data Objects in Leaf Nodes for Figure 47 . . . . .	69
49. The 2nd Subscreen of the MR-tree for Figure 41 . . . . .	69
50. Arrangement of Data Objects in Leaf Nodes for Figure 49 . . . . .	69
51. Screen Division in Two Size Case . . . . .	71
52. Overlapping Data Objects in Two Size Case . . . . .	72
53. An Example of Screen Division in Two Size Case . . . . .	73
54. (a) Data Objects Belonging to the 2nd Subscreen . . . . .	74
(b) Length of a Stair in the 1st Subtree . . . . .	76
55. Data Objects Belonging to the 3rd Subscreen . . . . .	76
56. The Redundancy Between Leaf Level Nodes . . . . .	78
57. The Number of Redundant Areas . . . . .	78
58. The Redundancy of the R <sup>+</sup> -tree as a Function of Ov . . . . .	80
59. Total Number of Data Pages (leaf Level) vs. Ov . . . . .	80
60. Redundancy of the R <sup>+</sup> -tree at Leaf Node . . . . .	81
61. The Number of Disk Accesses vs. Search Range (N= 1,000,000 & Ov= 40) . . . . .	83
62. The Number of Disk Accesses vs. Search Range (N= 1,000,000 & Ov= 30) . . . . .	84
63. The Number of Disk Accesses vs. Search Range (N= 100,000 & Ov= 40) . . . . .	84
64. The Number of Disk Accesses vs. Search Range (N= 100,000 & Ov= 30) . . . . .	85

Figure	Page
65. Performance Gain of the MR-tree (In Serial Processing) over the R <sup>+</sup> -tree as a Function of Ov (N= 1000,000) . . . . .	85
66. Performance Gain of the MR-tree (In Serial Processing) over the R <sup>+</sup> -tree as a Function of Ov (N= 100,000) . . . . .	86
67. Time (Disk Access) Comparison Between the MR-tree (In Parallel Processing) and the R <sup>+</sup> -tree (Point Query) . . . . .	87
68. Time (Disk Access) Comparison Between the MR-tree (In Parallel Processing) and the R <sup>+</sup> -tree (Point Query) . . . . .	88
69. Time Comparison Between the MR-tree (in Parallel Processing) and the R <sup>+</sup> -tree (Range Query) . . . . .	89
70. Time Comparison Between the MR-tree (in Parallel Processing) and the R <sup>+</sup> -tree (Range Query) . . . . .	89
71. Time Comparison Between the MR-tree (in Parallel Processing) and the R <sup>+</sup> -tree (Range Query) . . . . .	90
72. Time Comparison Between the MR-tree (in Parallel Processing) and the R <sup>+</sup> -tree (Range Query) . . . . .	90
73. Performance Gain of the MR-tree (In Parallel Processing) over the R <sup>+</sup> -tree as a Function of Ov . . . . .	91
74. Performance Gain of the MR-tree (in Parallel Processing) over the R <sup>+</sup> -tree as a Function of SR . . . . .	92
75. Performance Gain of the MR-tree (In Parallel Processing) over the R <sup>+</sup> -tree as a Function of Ov . . . . .	92
76. Performance Gain of the MR-tree (in Parallel Processing) over the R <sup>+</sup> -tree as a Function of Search Range. . . . .	93
77. Graph Corresponding to Table 2 . . . . .	97
78. Graph Corresponding to Table 3 . . . . .	98

Figure	Page
79. Performance Gain of the MR-tree for Parallel Processing over Serial Processing vs. Cd (N= 10,000) . . . . .	99
80. Performance Gain of the MR-tree for Parallel Processing over Serial Processing vs. Search Range (N= 10,000) . . . . .	99
81. The Number of Disk Accesses for Parallel Processing vs. Cd (Cd= 5.52 and Cd= 6.47) . . . . .	100

## LIST OF TABLES

Table	Page
1. The Number of Disk Accesses for Serial and Parallel Processing (Cd= 4.61) . . . . .	95
2. The Number of Disk Accesses for Serial and Parallel Processing (Cd= 5.52) . . . . .	96
3. The Number of Disk Accesses for Serial and Parallel Processing (Cd= 6.47) . . . . .	96
4. The Number of Disk Accesses for Serial and Parallel Processing (Cd= 7.44) . . . . .	96
5. Time to Access Disk Pages in Search Area . . . . .	97
6. Distribution of Rectangles (Cd= 5.52) . . . . .	100
7. Distribution of Rectangles (Cd= 6.47) . . . . .	101
8. Performance Comparison Between Data Structures in Four Different Data Types (v: Better, vv: Much Better) . . . . .	101

## CHAPTER I

### INTRODUCTION

Since conventional data base management systems (DBMS) have developed for handling one-dimensional data objects, such as integers, real numbers, or strings, they are not proper for handling multi-dimensional data objects, e.g., boxes or polygons. For example, B-tree and ISAM indexes are the structures for one dimensional data objects. Multi-dimensional data have been increasing in many areas, for instance, map data processing in geographic information system (GIS) allows a user to collect, manage, and analyze large volumes of spatially referenced data [7,11,15] and computer aided design (CAD) needs to store and retrieve very large number of rectangles . To satisfy these new applications, several data structures have been proposed.

To be efficient, multi-dimensional data structures have to satisfy fast access to objects in the database as well as possess efficient space utilization. There are two types of search applications given below:

- (1)point query: find all objects containing a given point in the space,
- (2)range query: find all objects intersecting with a given range.

For example, finding all states that have land within 100 miles of a particular point is a kind of range search operation.

Multi-dimensional index structures proposed earlier, for example, the binary tree, the k-d tree [2] and the Quad tree [1] cause problems with secondary memory paging systems. The object of this thesis is to improve the performances of the R-tree (an extension of the B-tree in k-dimensions) [8] and the R<sup>+</sup>-tree (an updated version of the R-tree) [12]. We propose an improved data structure, the Multi-R tree, to remove the redundancy in leaf level nodes of the R<sup>+</sup>-tree and to provide better search performance. After we give a brief description of the R-tree and the R<sup>+</sup>-tree structures in Chapter II, the detailed structure and algorithms for searching, insertion, deletion and splitting of the MR-tree are introduced in Chapter III. In Chapter IV, the R-tree and the R<sup>+</sup>-tree analysis methods are briefly discussed as a preview of the MR-tree analysis in Chapter V. Chapter V provides performance analyses on the MR-tree for exact point query and range query and the results are compared with those of R<sup>+</sup>-tree. Chapter VI gives conclusion on the MR-tree.

## CHAPTER II

### LITERATURE REVIEW

In this Chapter, several data structures which have been proposed to handle multi-dimensional data objects are discussed briefly. Multi-dimensional data can not be represented by point very well. For example, a rectangle in 2-d space requires 4 coordinates (e.g., upper right and lower left corners) to represent its position in the space. To handle spatial objects efficiently, a data base system requires an index mechanism to retrieve data items according to their spatial locations. Since traditional one-dimensional data base indexing structures were not appropriate for multi-dimensional spatial searching, a new indexing structure was needed.

Some data structures have been considered to handle multi-dimensional point data [4]. Among those structures, cell methods are not general enough for a dynamic structure because cell boundaries must be decided in advance [3]. Quad tree [1] and K-d tree [2] do not allow paging of secondary memory. In a paging system, a disk page can hold about 50 nodes depending on its page size. Therefore, a tree with small fan-out (the number of pointers to subnodes) is not appropriate (expensive). The k-d-B tree [5] was developed to work in a paging environment but it can handle point data only. Grid file [6] and BANG file [14] can handle non-point data by mapping each object into a point in a higher dimensional space. The GBD tree [16] is a data structure with homogeneous nodes. It is a kind of mixed structure of the R-tree and grid file. In the GBD tree, an entry consists of



child pointer, DZ expression, and MBR. DZ expression is used to determine the position in the tree structure where an entity is located and MBR is used to eliminate irrelevant nodes during search operations. For DZ expression, binary division is used where dividing axis is selected alternatively between x and y axis for two dimensional cases. The MBR denotes the minimal bounding rectangle to enclose all data in the node pointed by the child pointer as in the R-tree. However, the GBD tree allows overlapping between intermediate rectangles (MBR).

The purpose of the MR-tree is to improve the R-tree and the R<sup>+</sup>-tree performances. So those data structures are discussed in detail in this Chapter.

### R-tree: The dynamic Index Structure for Spatial Searching

#### Structure

The R-tree is an extension of the B-tree to k-dimensions. It is a height-balanced tree with index records stored in leaf nodes containing pointers to data objects. The R-tree consists of two node types, an intermediate node and a leaf node. Leaf node entries consist of two parts, a tuple identifier being used to refer to a tuple in the data base and coordinates representing an n-dimensional rectangle which is a box enclosing the spatial object. An intermediate node can hold up to C (capacity of a node) entries and each of them consists of a child pointer pointing to the node at a lower level and coordinates representing a rectangle which completely encloses all rectangles in the target subtree. Let's define M and m as follows [8]:

M : maximum number of entries in one node,  
m : minimum number of entries in one node ( $m \approx M/2$ ).

The R-tree satisfies the following properties [8]:

- 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 leaf node, I is the smallest rectangle that spatially contains the n-dimensional 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 node has at least two children unless it is a leaf;
- 6) all leaves appear on the same level.

In Figure 1, rectangle 1 is a leaf level rectangle which is the minimum enclosing box of a data object. Rectangle 11, 12 and 13 are intermediate nodes, for example, rectangle 11 encloses rectangles 1, 2, and 3 completely.

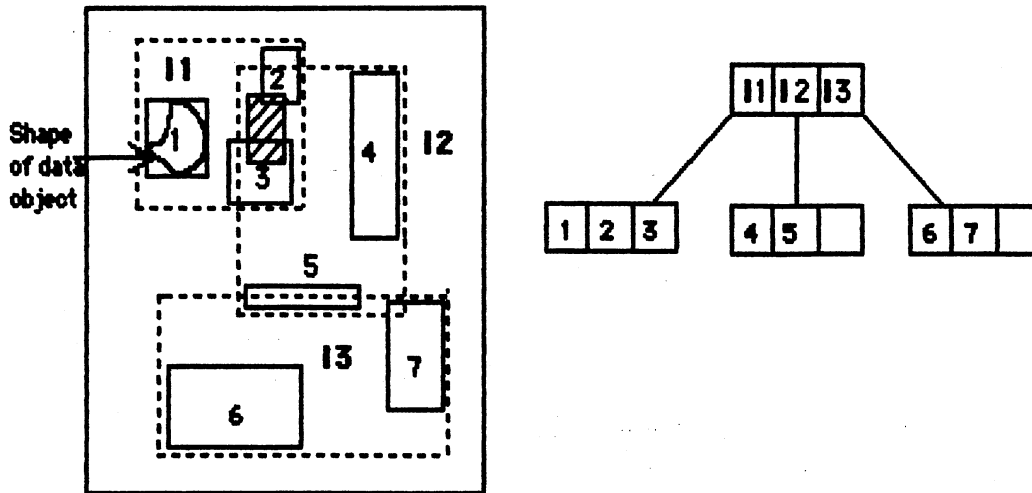


Figure 1. Sample Rectangle Arrangement and the R-tree Structure

Since the R-tree uses a dynamic insertion algorithm, it allows overlapping among the intermediate rectangles. In Figure 1, intermediate rectangles I1 and I2, and intermediate rectangles I2 and I3 overlap. If there exists  $k$ -overlapping of an area and a search range includes that area, it is necessary to take all  $k$  possible paths to find objects overlapping with search area. For examples, if shaded area represents the search range then intermediate rectangles I1 and I2 overlap with search range. However, only I1 contains data (leaf level) rectangles overlapping with search range. In this simple example, such a false search may look like a trivial problem. But if node capacity is 50 and tree height is more than 3 then the cost of false searching is increased. In order to solve this problem, a packing technique was proposed in [10]. But this packing algorithm can not be applied on every single insertion.

## Operations

### Searching

The search algorithm descends the tree from the root in a manner similar to a B-tree. But more than one subtree under a node visited may need to be searched. The search algorithm eliminates irrelevant regions of the indexed space and examines only data near the search area. Let  $E_i$  represent the rectangle part and  $EP$  denote a tuple-identifier or child-pointer [8].

Algorithm R-SEARCH: given an R-tree whose root node is  $T$ , find all index records whose rectangles overlap a search rectangle  $S$ .

S1.[search sub-tree] If  $T$  is not a leaf, check each entry  $E$  to determine whether  $E_i$  overlaps  $S$ . For all overlapping entries, invoke R-SEARCH on the tree whose root node is pointed to by  $EP$ .

S2.[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.

### Insertion

Inserting index records for new data tuples is similar to insertion in a B-tree in that new index records are added to the leaves, nodes that overflow are split, and splits propagate up the tree. The R-tree insertion algorithm has sub-algorithms ChooseLeaf, SplitNode, and AdjustTree. First, find a leaf node to insert new rectangle into. The ChooseLeaf algorithm

selects the proper leaf node which gives minimum enlargement to include the new rectangle. Second, if the leaf node is already full, invoke SplitNode to split a node into two nodes. Then the AdjustTree algorithm ascends from the leaf node to the root node to adjust covering rectangles and propagates node splits as necessary [8].

### Deletion

The deletion algorithm removes an index record from the R-tree. Sub-algorithms are FindLeaf and CondenseTree. FindLeaf finds the leaf node containing the index entry. CondenseTree eliminates the node if it has too few entries and relocates its entries. For this, the INSERTION algorithm is invoked. Reinsertion incrementally refines the spatial structure of the tree and prevents gradual deterioration [8].

### Splitting

In order to insert a new rectangle into a full leaf node containing  $M$  entries, it is necessary to divide the  $M+1$  entries between two nodes. The division should be done in a way that makes it as unlikely as possible that both new nodes will need to be examined on subsequent searches [8]. Since the decision whether to visit a node depends on whether its covering rectangle overlaps the search area, the total area of the two covering rectangles after a split should be minimized [10]. In Figure 2, (a) takes more space than (b). (a) is bad split and (b) is good split. There exist three different splitting mechanisms.

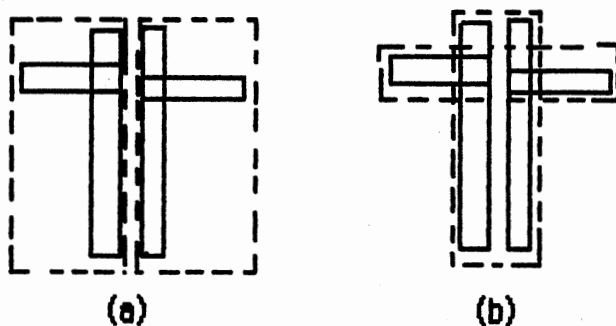


Figure 2. Bad (a) and Good (b) splits [8]

Exhaustive Algorithm. Straightforward but expensive method. Used as a standard for comparison with other algorithms [8].

Quadratic Cost Algorithm. This method attempts to find a small-area split but is not guaranteed to find the smallest area possible. It picks two of the  $M+1$  entries to be the first elements of the two new groups by choosing the pair that would waste the most area if both were put into the same group. For each pair of entries  $E1$  and  $E2$  compose a rectangle  $J$  including  $E1$  and  $E2$ . Calculate  $d = \text{area}(J) - \text{area}(E1) - \text{area}(E2)$ . Choose the pair with the largest  $d$ . Those two rectangles  $E1$  and  $E2$  become the 1st entry of two nodes. Then the remaining entries are assigned to groups one at the time [8].

Linear Cost Algorithm. Find extreme rectangles along all dimensions. After normalization, select the most extreme pair [8].

[find extreme rectangles along all dimensions] Along all dimensions find the entry whose rectangle has the highest low side and the one with lowest high side. Record the separation.

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

[Select the most extreme pair] Choose the pair with the greatest normalized separation.

In Figure 3, along each dimension (this case 2-d) find rectangles which have highest low side or lowest high side. Then normalize by using the width (e.g.,  $w_1$  and  $w_2$ ) of the intermediate rectangle for each dimension.

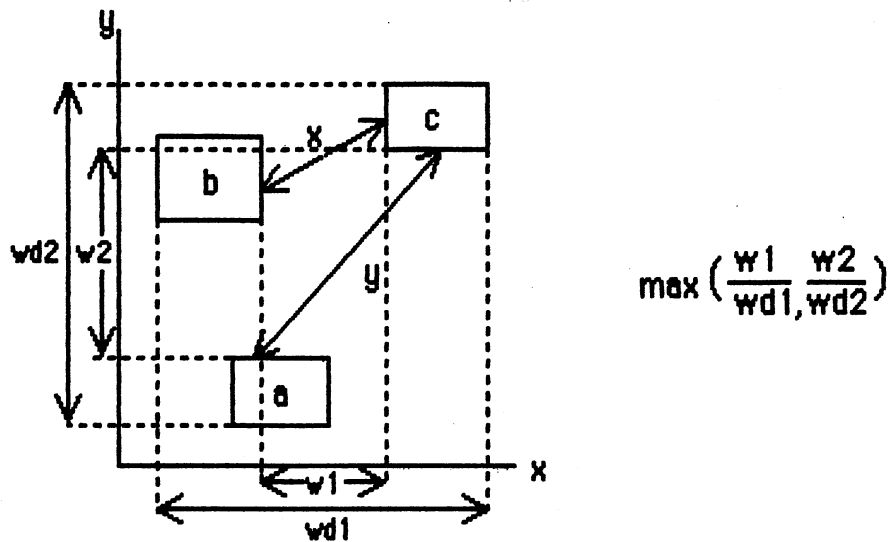


Figure 3. Linear-Cost Algorithm

### Conclusion

The R-tree structure has been shown to be useful for indexing spatial data objects that have non-zero sizes. Nodes corresponding to disk pages of reasonable size (e.g. 1024 bytes) have values of  $M$  that produce good performance.

The linear node-split algorithm proves to be as good as more expensive techniques. It is fast and the slightly worse quality of the split does not affect search performance noticeably. Figure 4, Figure 5, and Figure 6 show the performance comparisons between quadratic and linear split methods [8].

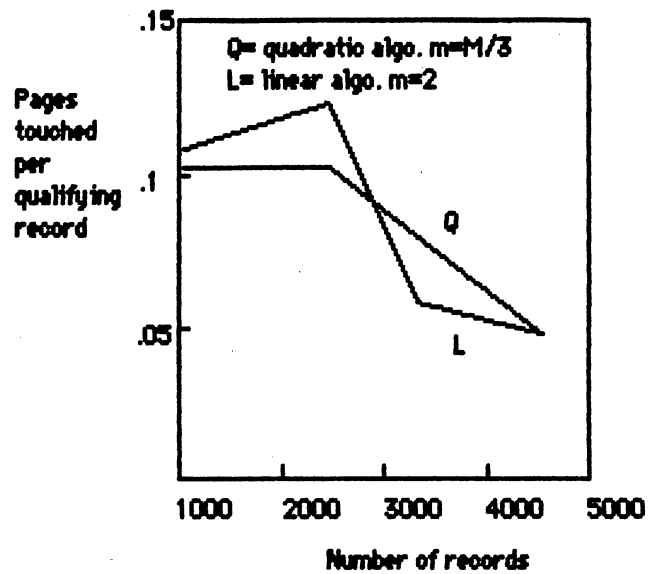


Figure 4. Search Performance vs. Amount of Data: Pages Touched [8]



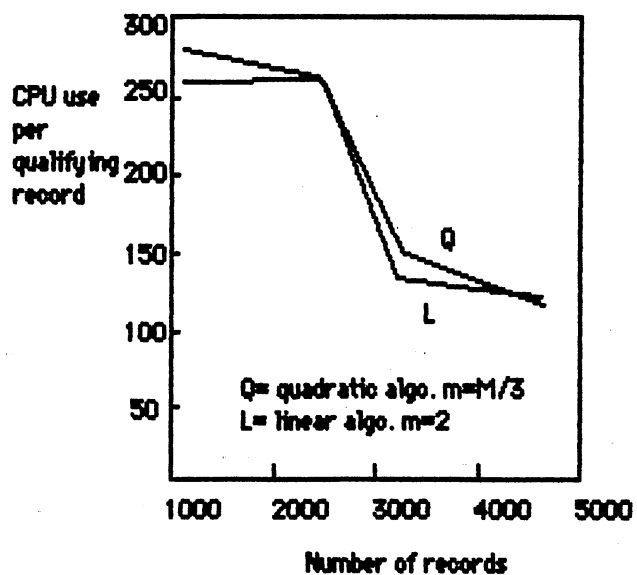


Figure 5. Search Performance vs. Amount of Data: CPU Cost [8]

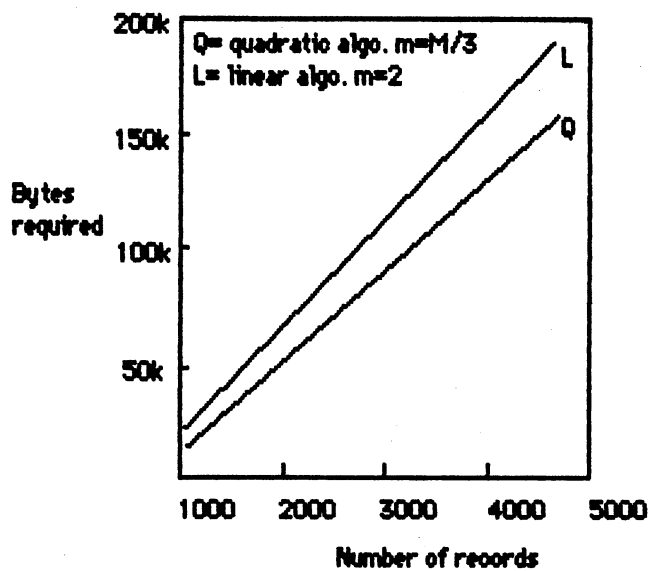


Figure 6. Space Required for the R-tree vs. Amount of Data [8]

## R<sup>+</sup>-Tree: A Dynamic Index Structure for Multi-dimensional Objects

The R-tree is a direct extension of the B-tree for multi-dimensional objects. The R-tree is a balanced tree and has at least 50% space utilization. Since R-trees are built using dynamic insertion algorithms, the structure may provide excessive space overlap and dead space in the nodes. That gives bad performance. When we consider the performance of the R-tree, the concept of coverage and overlap are important. An efficient R-tree demands minimal coverage and overlap which are defined as follows [12]:

coverage: total area of all the rectangles associated with the node of that level;

overlap: total area contained within two or more nodes.

→ The main difference between the R-tree and the R<sup>+</sup>-tree is that the the R<sup>+</sup>-tree does not allow overlapping among the intermediate rectangles. The R<sup>+</sup>-tree allows the partitioning of rectangles to provide zero overlap between intermediate rectangles. Upon splitting, an intermediate rectangle is partitioned into two different intermediate rectangles and all sub-rectangles on the partition line are also divided. If a leaf node rectangle is on the partition line, it is divided into two with the same object name in both intermediate rectangles.

Figure 7 shows grouping of rectangles using R<sup>+</sup>-tree and Figure 8 is the structure of the R<sup>+</sup>-tree. In Figure 7, rectangle 3 is on the split line. If a data rectangle (leaf level rectangle) is on a partition line, it is not split and just stored in both intermediate rectangles [12]. Rectangle 3 is stored in intermediate rectangles I1 and I2. In this case its coordinates in both

intermediate rectangles are not changed. If an intermediate rectangle is on the partition line, it splits into 2 sub-regions.

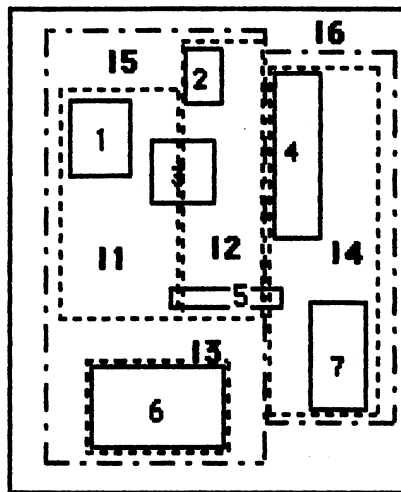


Figure 7. Rectangle Arrangement

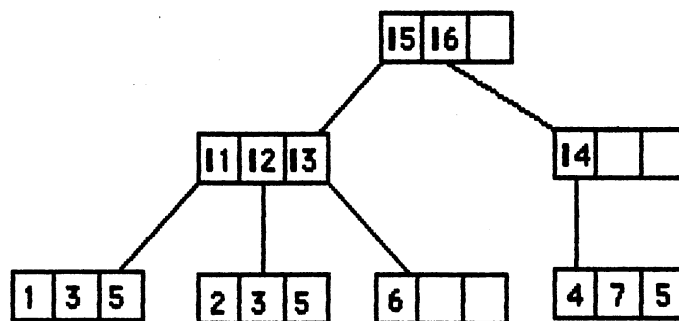


Figure 8. The R<sup>+</sup>-tree for Figure 7

The R-tree structure may have excessive space overlap and dead space in the nodes that results in bad performance. The R<sup>+</sup>-tree attempts to improve R-tree performance.

## Structure

The R<sup>+</sup>-tree has the same structure as the R-tree. It has intermediate nodes and leaf nodes defined as follows [12]:

leaf node: (oid , RECT)

where oid is an object identifier used to refer to an object in the data base. RECT is used to describe the bounds of data objects. For example, in a 2-dimensional space, an entry RECT is of the form (x<sub>low</sub>,x<sub>high</sub>,y<sub>low</sub>,y<sub>high</sub>).

intermediate node: (p,RECT)

where p is a pointer to a lower level node of the tree.

The R<sup>+</sup>-tree allows partitions to split rectangles so zero overlap among intermediate node entries can be achieved. This is one difference of the R<sup>+</sup>-tree from the R-tree. The R<sup>+</sup>-tree has the following properties [12]:

- 1) For each entry (p,RECT) in an intermediate node, the sub-tree rooted at the node pointed to by p contains a rectangle R if and if only 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<sub>1</sub>,RECT<sub>1</sub>) and (p<sub>2</sub>,RECT<sub>2</sub>) of an intermediate node, the overlap between RECT<sub>1</sub> and RECT<sub>2</sub> is zero.
- 3) The root has at least two children unless it is a leaf.
- 4) All leaves are at the same level.

## Operations

### Searching

The R<sup>+</sup>-tree search algorithm is similar to the one used in the R-tree. The idea is to first decompose the search space into disjoint sub-regions and for each of these descend the tree until the actual data objects are found in the leaf nodes. Notice that a major difference between the R<sup>+</sup>-tree and the R-tree is that in the latter, sub-regions can overlap, thus leading to more expensive searching [12].

RP\_SEARCH (N,W): N is a node pointer and W is a search area. Let NR be a node pointed by node pointer N.

S1. If the NR is not a leaf node, then for each entry (p,RECT) of NR check if RECT overlaps W. If so, RP\_SEARCH (p, W).

S2. If NR is a leaf node, check all objects RECT in NR and return those that overlap with W.

### Insertion

In the R<sup>+</sup>-tree, insertion is done by searching the tree and inserting the rectangle into a leaf node. An input rectangle may be inserted into more than one leaf node, the reason being that it may be broken to sub-rectangles along existing partitions of the space. Overflowing nodes are split and splits are propagated to parent and child nodes. A split of a parent node may introduce a space partition that affects the child nodes [12].

RP\_INSERT (N,IR): N is root node pointer and IR is input rectangle. Let NR be a node pointed by node pointer N. Find where IR should go and add it to the corresponding leaf node.

I1. If NR is not a leaf node, then for each entry (p,RECT) of NR check if RECT overlaps IR. If so, RP\_INSERT (p, IR).

I2. If NR is a leaf node, insert IR into NR. After the insertion of new rectangle if NR has more than M entries, call SPLITNODE (N) to re-organize the tree.

### Deletion

Deletion of a rectangle from the  $R^+$ -tree is done as in the R-tree. In the  $R^+$ -tree, it may be necessary to remove a data object from more than one leaf node because the  $R^+$ -tree allows a rectangle to be partitioned [12].

RP\_DELETE (R,IR): R is node pointer and IR is the input rectangle.

D1. If R is not a leaf, then for each entry (p,RECT) of R check if RECT overlaps IR. If so, RP\_DELETE (CHILD,IR), where CHILD is the node pointed to by p.

D2. If R is a leaf, remove IR from R and adjust the parent rectangle that encloses the remaining rectangles.

### Node Splitting

When a node overflows a splitting algorithm is needed to produce two new nodes. Two sub-nodes cover mutually disjoint areas so it is necessary to search for a good partition that decomposes the space into two sub-

regions. In the  $R^+$ -tree splitting algorithm, contrary to the R-tree splitting algorithm, downward propagation of the split may be necessary. For example, in Figure 9, suppose A is a parent node of B which in turn is a parent node of C. If node A has to be split, then lower level nodes B and C have to be split too by the property (1) of the  $R^+$ -tree. But objects in the leaf node are not split; this is just for efficiency reasons since rectangles in the leaf pages cannot account for further downward splits [12].

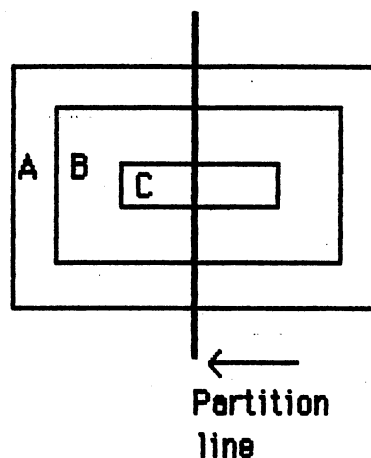


Figure 9. Rectangles on Partition Line [12]

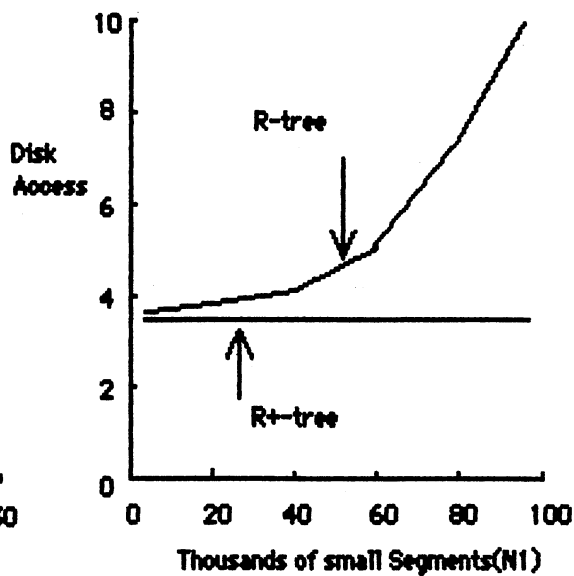
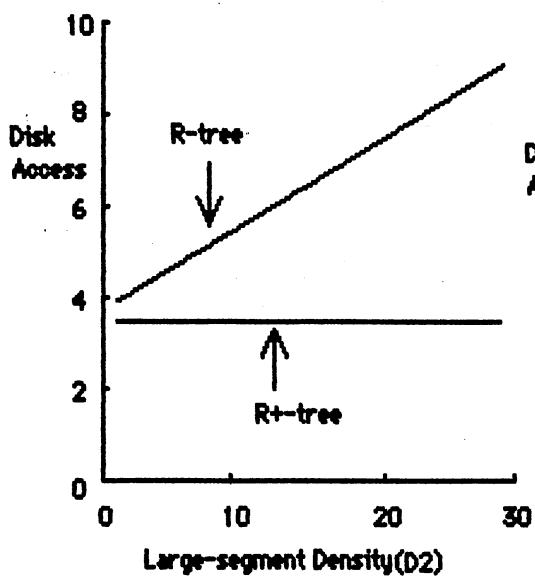
The  $R^+$ -tree node splitting algorithm consists of 4 sub-routines, SplitNode (R), Partition (S,ff), Sweep (axis,Oxy,ff), and Pack (S,ff).

### Conclusion

The main advantage of the  $R^+$ -tree as a variation of the R-tree is improved search performance, especially in point queries, where there can be even more than 50% saving in disk accesses over an R-tree [12]. Also this structure behaves exactly as a k-d-B tree in the case where the data are points instead of non zero area objects (rectangles). This is significant

in the sense that k-d-B trees have been shown (by empirical means) to be very efficient for indexing point data. Figure 10 through Figure 13 show the performance comparison between the R-tree and the R<sup>+</sup>-tree. Let's define density as the number of segments which contain a given point. Figure 10 and Figure 11 show disk accesses required for search used to index 100,000 segments with total density of 40. Figure 10 shows the number of disk accesses as a function of large segment density (D<sub>2</sub>) when the large segments account for 10% of the total number of segments (N<sub>1</sub>= 90,000 and N<sub>2</sub>= 10,000). In Figure 10, the large segment density is increased as the number of long segments increase. In such a case, an R-tree may require more than twice the page accesses as the number required by an R<sup>+</sup>-tree. In Figure 11, D<sub>1</sub> represents small segments density. In Figure 12 and Figure 13 the number of disk accesses is illustrated for segment queries on an R-tree and an R<sup>+</sup>-tree. In Chapter IV, R-tree and R<sup>+</sup>-tree performances are analyzed.

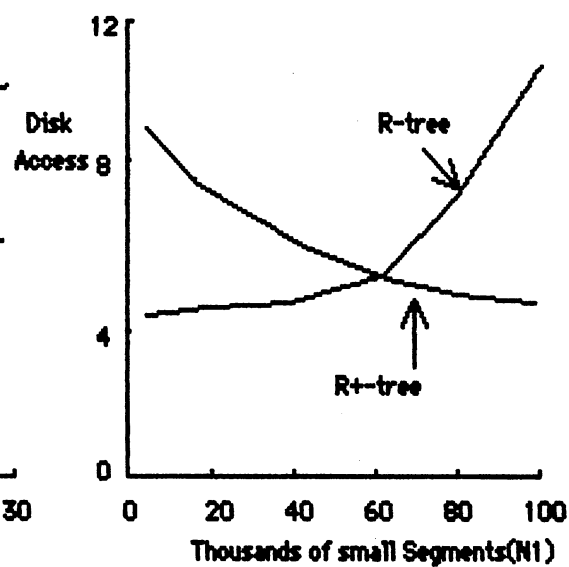
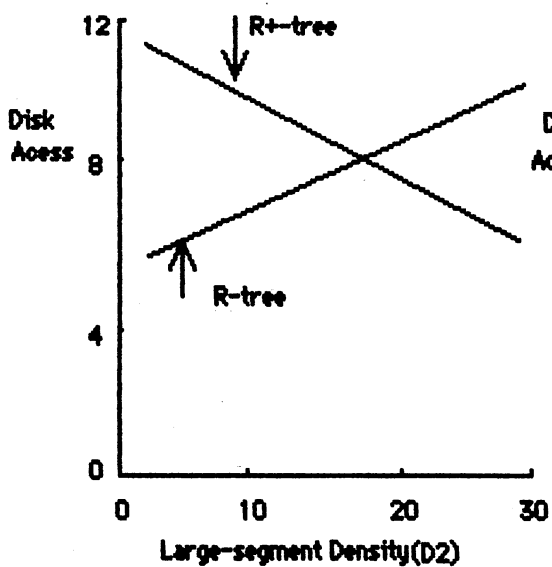




Disk Accesses for the Segments of Two Different Sizes Using Point Query

Figure 10. As a Function of D2, N2=10,000 [12]

Figure 11. As a Function of N1, D1=5 [12]



Disk Accesses for the Segments of Two Different Sizes Using Segment Query

Figure 12. As a Function of D2, N2=10,000 [12]

Figure 13. As a Function of N1, D1=5 [12]

## CHAPTER III

### MULTI-R (MR) TREE

#### Introduction

The MR-tree is a data structure to handle multi-dimensional data. The MR-tree has the same node structure as the R-tree and the R<sup>+</sup>-tree. It consists of leaf nodes and intermediate nodes. All objects are stored in leaf level nodes and intermediate nodes consist of intermediate rectangles. Each intermediate rectangle completely encloses all of the rectangles at lower levels.

The main idea of the MR-tree is to distribute data objects into several subscreens (data spaces) to provide zero duplication of data objects among leaf level nodes. The MR-tree provides zero overlap among intermediate nodes and further removes redundancy in leaf nodes. By removing the rectangle(s) which are selected by the MR-tree splitting algorithm from a screen, the MR-tree eliminates the redundancy at leaf level nodes. Each subscreen is associated with a subtree. For example, Figure 14 and Figure 15 show subscreens of the MR-tree for Figure 7 in Chapter II. By excluding rectangle 5 from Figure 7, we obtain the 1st subscreen with a neat arrangement of intermediate rectangles as shown in Figure 14. The rectangle 5 now is inserted into the 2nd subscreen as shown in Figure 15. The corresponding subtrees of the 1st and the 2nd subscreens are shown in Figure 16. Obviously, at the leaf level, there is no duplicated data

rectangle. In R<sup>+</sup>-tree (as shown in Figure 8), each of data rectangles 3 and 5 was located at more than one leaf node. The MR-tree is a forest of subtrees.

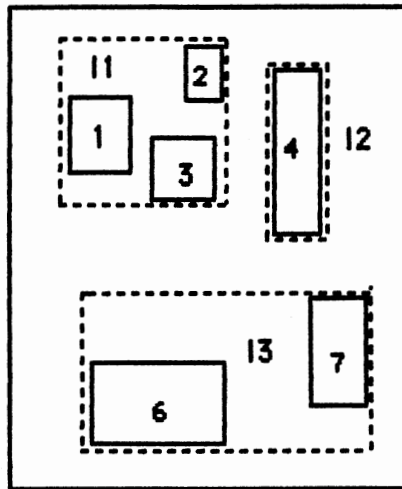


Figure 14. Rectangles in the 1st Sub-screen of the MR-tree

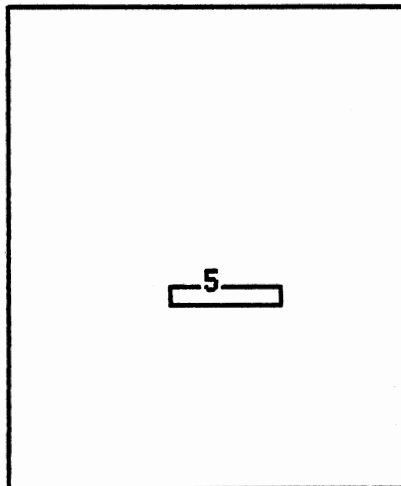


Figure 15. Rectangle in the 2nd Sub-tree of the MR-tree

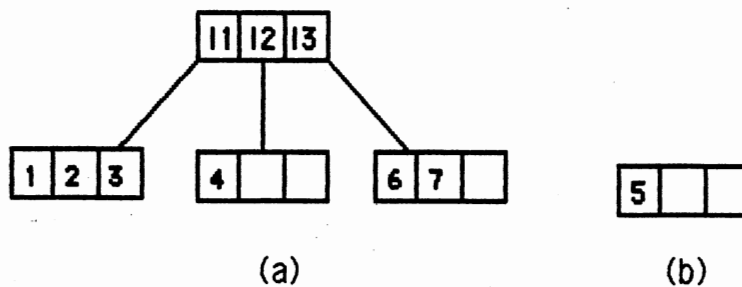


Figure 16. (a)the 1st Subtree for  
Figure 14

(b)the 2nd Subtree for  
Figure 15

In Chapter V, we further provide the method to obtain the number of sub-trees for the MR-tree with uniformly distributed segment objects. If parallel processors are used, search procedures for all sub-trees can be executed simultaneously. Thus the MR-tree provides better performance for both range queries and point queries than those of the  $R^+$ -tree and R-tree. Query performances are discussed in Chapter V in detail.

### Structure

A leaf node entry of the MR-tree is of the form

(t-id, COOR)

where t-id is a tuple identifier used to refer to a data object in the data base and COOR are the coordinates which define the minimum enclosing box of a data object. In the 2-dimensional case, COOR consists of 4 coordinates representing lower left and upper right corners. An intermediate node entry is of the form

(Pt, COOR)

where Pt is the pointer to its child node at the lower level and COOR are the rectangle coordinates enclosing all sub-rectangles in the child node. The MR-tree has the properties described as follows.

- 1) For each entry (Pt , COOR) in an intermediate node , the subtree pointed to by Pt contains rectangle R if and only if R is covered by COOR. This property is similar to the property of the  $R^+$ -tree. However, it is noted that the  $R^+$ -tree allows an exception of this property at leaf level nodes but the MR-tree does not. This is one of the major differences between the  $R^+$ -tree and the MR-tree.
- 2) No duplicated data objects appear at the leaf level.
- 3) Zero overlap occurs between entries of intermediate nodes.
- 4) The root node has at least two children unless it is the leaf.
- 5) All leaves are at the same level.

Let's consider the following example.

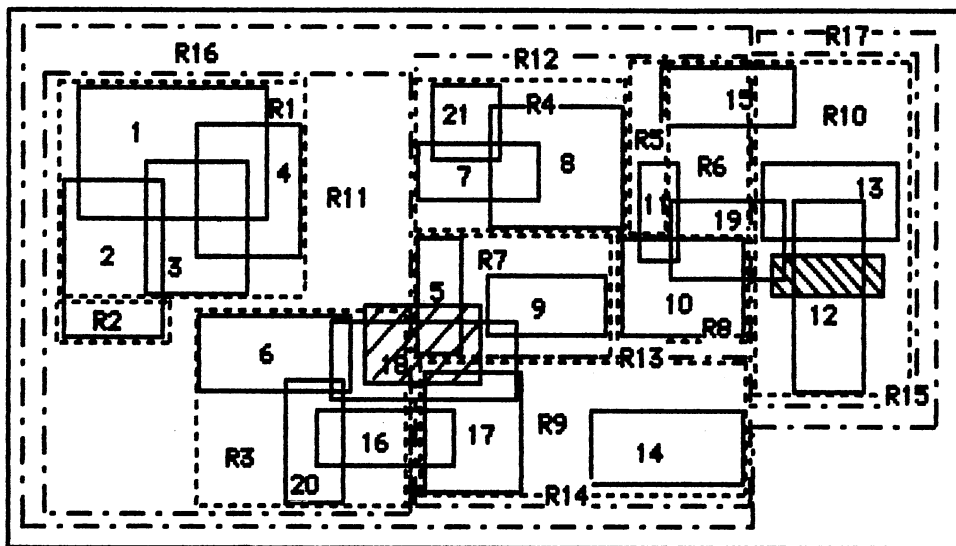


Figure 17. An Organization of Rectangles into an  $R^+$ -tree

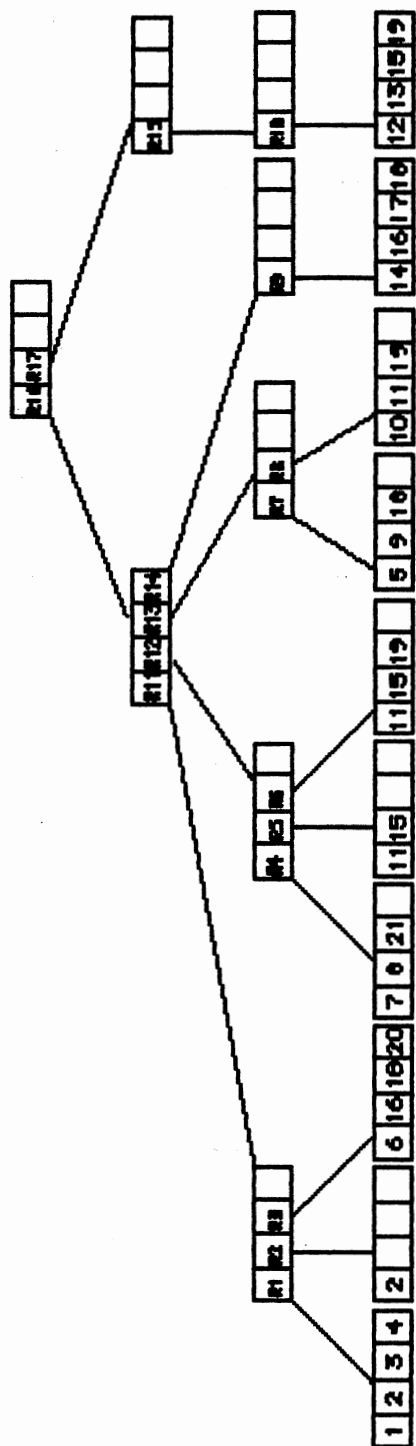


Figure 18. R\*-tree Structure for the Rectangles in Figure 17

Figure 18 is the R+-tree structure for the rectangles in Figure 17. Obviously, the redundancy at leaf level of the R+-tree increases as the overlapping of data objects increases.

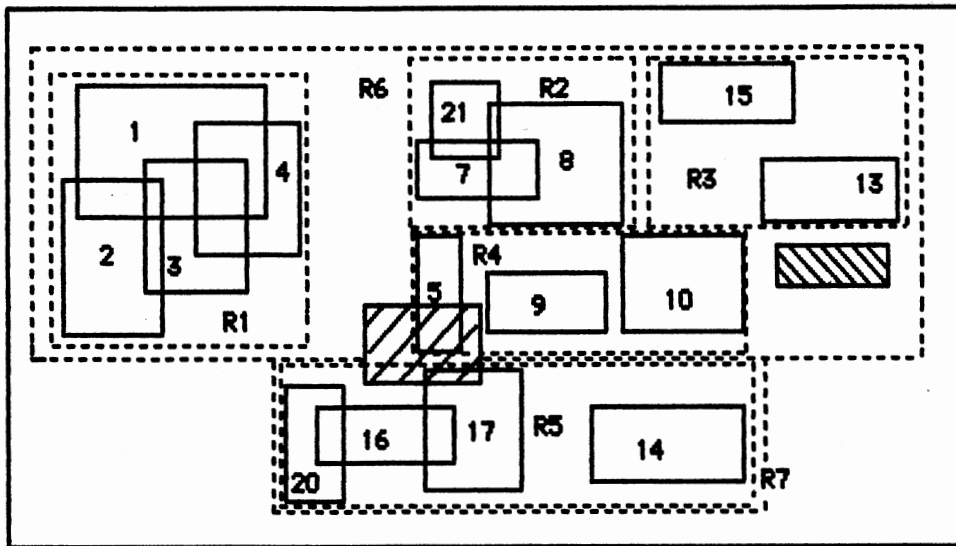


Figure 19. Organized Rectangles on the 1st Subscreen

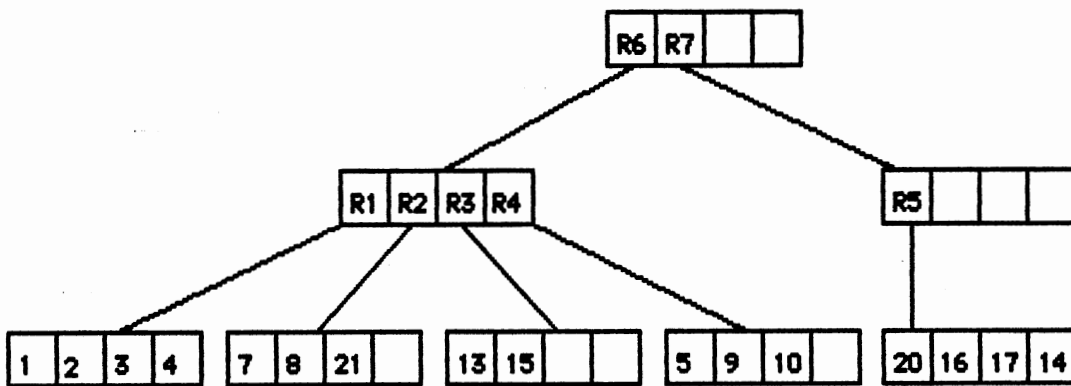


Figure 20. The 1st Subtree Structure of the MR-tree Corresponding to Figure 19

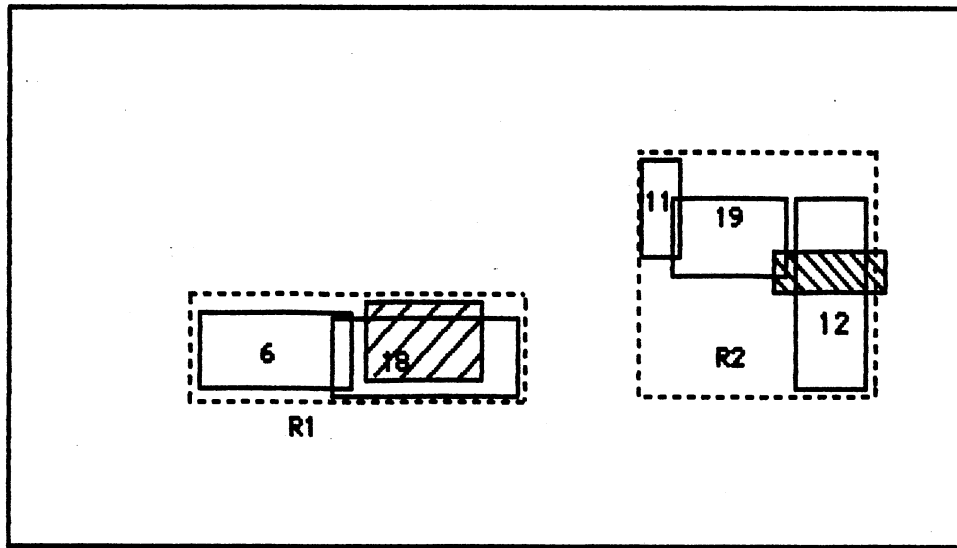


Figure 21. Organized Rectangles on the 2nd Subscreen

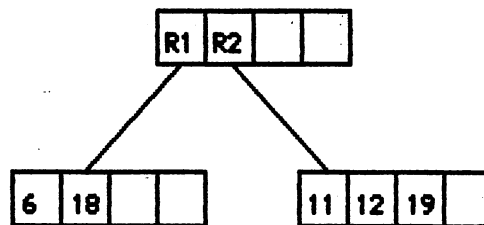




Figure 22. The 2nd Subtree Structure of the MR-tree Corresponding to Figure 21

Figure 19 and Figure 21 show sub-screens of the MR-tree for rectangles as shown in Figure 17. Figure 20 and Figure 22 are the subtrees for rectangles in Figure 19 and Figure 21 respectively. The height of the 1st subtree is 2 and that of the 2nd subtree is 1. The number of disk accesses in point query is the height plus one. In this example, the number of disk accesses of the  $R^+$ -tree is 4 while the MR-tree needs 5 disk accesses, (3 for the 1st subtree and 2 for 2nd subtree). However, parallel processing of the MR-tree can reduce the time for point query by parallel



searching on all subtrees. If shaded rectangle (  ) represents search area, the R<sup>+</sup>-tree has to access 8 pages, pointed by R16, R17, R11, R13, R14, R3, R7, and R9. The MR-tree only needs 7 page accesses, 5 for the 1st sub-tree and 2 for the 2nd sub-tree for the same search range. Again, parallel processors can execute search processes for sub-trees simultaneously and the time to access data pages which contain objects overlapping with the search range in the MR-tree can be reduced. In this case, the MR-tree for parallel processing will require the time needed to access 5 disk pages instead of 7.

The R<sup>+</sup>-tree can not handle a range deletion operation efficiently. In the R<sup>+</sup>-tree data rectangle (leaf level rectangle) may not be completely enclosed by its upper level intermediate rectangle(s) by the property of the R<sup>+</sup>-tree which allows partition of intermediate rectangles. For example, in Figure 17, data rectangle 19 is not completely enclosed by the intermediate rectangle R10. If a data rectangle is on a partition line, it possibly resides across more than one intermediate rectangle. And in each intermediate rectangle, it has complete coordinates no matter what the boundary coordinates of intermediate rectangle(s). If deletion is done by range (delete all data objects overlapping with given range), not by exact coordinates as usual, duplicated data objects may not be completely deleted from the R<sup>+</sup>-tree. This could happen if the deletion range does not overlap all intermediate rectangles containing such a data object inside of them. In Figure 17, for example, if the shaded rectangle (  ) denotes the deletion range, rectangles 19 and 12 are involved in the deletion. Data rectangle 19 exists in intermediate rectangles R6, R8, and R10. However, deletion range only overlaps R10. This means that only rectangle 19 in R10 can be deleted and the other intermediate rectangles, R6 and R8, are

transparent to the  $R^+$ -tree deletion algorithm. To solve this problem, the  $R^+$ -tree may have to use a two step process: search all data rectangles overlapping with the deletion range by using search algorithm and then delete data objects one by one by using deletion algorithm with exact coordinates of each object. In the MR-tree, there is no such a problem since data rectangles are completely enclosed by upper level intermediate rectangles. If the  $R^+$ -tree uses a two step process as in the above example, the total number of page accesses is 17 (4 for search and 13 for deletion). On the other hand, the MR-tree just needs 4 page accesses. Also deletion can be done by parallel processing. If so, only need time to access 2 disk pages.

More quantitative analyses of performance are given in Chapter IV and Chapter V.

## Operations

### Searching

The searching algorithm consists of two parts, SEARCH1 and SEARCH2. The purpose of SEARCH1 is used to call SEARCH2 as many times as the number of sub-trees in the MR-tree. SEARCH2 algorithm is almost the same as those of the R-tree and the  $R^+$ -tree. SEARCH2 recursively searches the tree to get all overlapping data objects. If parallel processors are used then SEARCH1 can be eliminated, since searching procedures on the subtrees can be initiated at the same time.

---

### SEARCH1 (W)

Input :

W: coordinates of the search area

Output:

Data objects which overlap with W

P: Array containing root pointers of each sub-tree

n: Number of sub-trees

i: subscript

N: node pointer

i = 0

S1.1[ Repeat the search for all sub-trees]

  If  $i < n$

    N = P[i]

    call Search2 (N , W) and  $i = i + 1$

  else

    return all data objects overlapping W

---

---

### SEARCH2 (N, W)

Input:

N: node pointer

W: coordinates of the search area

Output:

Data objects which overlap with search area W

NR: node pointed by node pointer N

S2.1[ Search intermediate rectangles ]

If NR is not a leaf node then

check each entry ( consists of Pointer to the child node and COORDinates which define the minimum rectangle enclosing the rectangles in child node, (Pt,COOR)) in the node NR

If there exists an entry having overlapping COOR in node NR then

call SEARCH2 (Pt,W)

else

exit

else

Goto S2.1

S2.1[ Search leaf node rectangles ]

If NR is a leaf node then

check all entries in NR

If there exist COOR overlapping with the search area W then

return all of them

### Insertion

The insertion algorithm consists of four parts: INSERT1, INSERT2, CHECKENTRY and ADJUST. The purpose of INSERT1 is to find a proper sub-tree to locate rectangle NEW. If there exists a proper sub-tree then insert NEW into that sub-tree. If not, then create a new root

and insert NEW into the root. CHECKENTRY checks whether NEW can be safely inserted into a node or not. ADJUST adjusts entry coordinates of an intermediate rectangle after insertion of NEW into the entry.

---

INSERT1 (P, NEW)

Input:

P: Array containing root pointers of each sub-tree

NEW: coordinates of newly inserted rectangle

output:

A new MR-tree after insertion of NEW

n: number of sub-trees

i: subscript

N: node pointer

SN: splitnode pointer

i = 0

11.1[Find a proper subtree to insert NEW]

N = P[i]

SN = INSERT2 (N , NEW)

If insertion of NEW is successful

    If SN is not NULL

        create a new root and locate N and SN as its entries

    Go to 11.2

else

    i = i + 1

    if (i < n)

Go to 11.1

else

create a new subtree root and insert NEW into that root

exit

### 11.2[ Reinsert rectangles]

After insertion of NEW into a sub-tree, if there is rectangle(s) removed during SPLIT procedure then reinsert those rectangle(s) into subsequent subtrees.

---



---

INSERT2 (N , NEW)

Input:

N: node pointer

NEW: coordinates of newly inserted rectangle

output:

Insertion of rectangle NEW into a sub-tree

Pt: node pointer in an entry pointed by E

E: entry pointer

SN: splitnode pointer

NR: node pointed by node pointer N

### 12.1[Search intermediate node]

If NR is not intermediate node

E= CHECKENTRY (N, NEW)

If E is not NULL

SN= INSERT2 (Pt , NEW)

If insertion is successful

ADJUST (E)

else

Return NULL to INSERT1

12.2[Insert NEW into leaf node]

If NR is a leaf node, insert NEW into NR. If the rectangle NEW makes node have more than M entries (node capacity), then call SPLIT (N) to re-organize the sub-tree.

---



---

CHECKENTRY (N , NEW)

input:

N: node pointer

NEW: coordinates of newly inserted rectangle

output:

If NEW can be included into an entry (upper level intermediate rectangle) of node NR safely, then return that entry pointer to INSERT2

If not, then return NULL.

NR: node pointed by node pointer N

C1[If more than 1 entry overlaps]

If more than one entry (Pt, COOR) overlaps with NEW, then return NULL

C2[One entry overlaps with NEW]

If one entry overlaps with NEW, then get the minimum enlargement of that entry to enclose rectangle NEW

If enlarged rectangle overlaps with other entries, then

return NULL

else

return that entry pointer

C3[No entry overlap with NEW]

If there is no entry overlapping rectangle NEW, then for each entry, get minimum enlargement and check whether each enlarged rectangle overlaps with other entries

If there is no entry which can enclose NEW without overlapping with other entries

return NULL

else

select an entry which has minimum enlargement to enclose NEW

return that entry pointer

ADJUST (E)

Input:

E: entry pointer (an entry (Pt, COOR))

output:

entry pointer with new adjusted coordinates

A1[get MAX and MIN coordinates along all dimensions]



Along all dimensions get MAX and MIN coordinates for the entries in Pt

(Pt is the child node pointer in an entry pointed by E)

A2[Adjust coordinates in entry]

Adjust coordinates of entry with those extreme coordinate values

---

### Deletion

The deletion operation consists of two parts: DELETE1 and DELETE2. DELETE1 calls DELETE2 as many times as the number of sub-trees in the MR-tree. DELETE2 finds rectangle(s) first and then deletes the rectangle(s) from a sub-tree as in R<sup>+</sup>-tree. Again parallel processing can remove DELETE1 by executing DELETE2 processes simultaneously. Also the MR-tree can handle range deletion operation efficiently as mentioned before.

---

DELETE1 (P,DEL)

Input:

P: pointer array containing the roots of sub-trees

DEL: coordinates of rectangle to be deleted

Output:

new tree after deletion

k: number of subtrees

i: subscript

N: node pointer

NR: node pointed by node pointer N

$i = 0$

D1.1[ Repeat searching for all sub-trees]

If  $i < k$

$N = P[i]$

DELETE2 (N , DEL) and  $i = i + 1$

If the number of entries of NR is zero, then remove N from P

else

exit

DELETE2 (N,DEL)

Input:

N: node pointer

DEL: coordinates of rectangle to be deleted

Output:

new structure after deletion of rectangle DEL

NR: node pointed by node pointer N

D2.1[Search intermediate nodes]

If NR is not a leaf node, then for each entry (Pt , COOR) of node NR, check whether COOR overlaps DEL. If so, call DELETE2 (Pt , DEL)

If deletion is successful then adjust coordinate of the entry.

D2.2[ Delete DEL from leaf node]

If NR is leaf, then

remove DEL from leaf node.

In deletion, the  $R^+$ -tree may have to traverse an excessive number of paths

to delete a data object, since it allows duplicated data objects among the leaf nodes. For example, if node capacity is 50 and a data object overlaps with 40 different intermediate nodes, then the R<sup>+</sup>-tree has to take 40 different paths to delete that object. As in R<sup>+</sup>-tree, many deletions degrade the space utilization of the MR-tree. To have better performance, subtrees need to be reorganized periodically.

### Node Splitting

The difference of the MR-tree from the R<sup>+</sup>-tree is the splitting algorithm. The node splitting algorithm consists of three parts: SPLIT, PARTRECT and DECOMPOSE. SPLIT routine calls PARTRECT to decide a partition line. PARTRECT decides a partition line and returns S1 and S2, sets of rectangles in partitioned sub-regions. Data (leaf level) rectangles on the partition line are removed and re-inserted into subsequent subtree.

---

#### SPLIT (N)

Input:

N: node pointer

Output:

re-structured tree

SN: splitnode pointer

S: set of rectangles in the node pointed by N

S1, S2: sets of rectangles in each sub-region

L: set of remaining rectangles

SP1[Find a proper partition line]

Call PARTRECT (N). Let G1 and G2 represent two sub-regions decided by PARTRECT. Create a new node pointed by SN for S2.

SP2[Store rectangles into sub-nodes]

Put the rectangles in sets S1 and S2 into the nodes pointed by N and SN respectively. And return SN and the set of remaining rectangle(s)  $L = S - S1 - S2$ .

PARTRECT (N)

Input:

N: node pointer

Output:

Two sets of rectangles S1 and S2

NR: node pointed by node pointer N

SECT: array containing the number of rectangles in each section (the number of sections is  $2(M+1) - 1$ ). Starting and ending coordinates of entries along all dimensions decide sections (Figure 23).

P1[No split]

If the number of entries in NR is less than or equal to M (capacity of node), then do not need to split the node.

P2[Set lowest coordinates]

Decide the lowest coordinates along all dimensions for the rectangles in NR and let those coordinates be the starting position.

P3[Check partition section]

For all dimensions, call DECOMPOSE (N, SECT). Then check if there exists a SECT[i] that gives 0. If SECT[i]= 0 , then a clear cut (no rectangle is on partition line) exists. Select that section. If there exists more than one clear cut, then select i for which SECT[i] provides minimal coverage.

P4[Calculate costs]

For all dimensions, calculate the cost between the factors which decide the efficiency of the MR-tree (minimal coverage and minimal number of splitting rectangles). Then select the partition line which gives the smallest cost.

P5[Store rectangles in each sub-region into sets S1 and S2]

Store rectangles in the 1st sub-region and the 2nd sub-region into S1 and S2, respectively. Then return S1 and S2.

DECOMPOSE routine decomposes intermediate nodes into sections by projecting starting and ending coordinates of each entry and decides the number of entries overlapping with each section. At any position in a section, the number of overlapping rectangles is the same.

DECOMPOSE (N,SECT)

Input:

N: node pointer

SECT: array containing the number of rectangles in each section

Output:

SECT (SECT[i] contains the number of intersecting rectangle(s)  
in a section)

DC1[get the number of rectangle(s)]

i= 0

while ( i < 2 (M+1))

for all entries in NR and NEW, check overlap. If an entry overlaps  
with SECT[i] then increase SECT[i] by 1.

i= i + 1

DC2[return]

return SECT

---

In Figure 23, assuming  $C = 4$ , insertion of rectangle 5 makes the node overflow. PARTRECT algorithm removes rectangle 4 and puts it into the 2nd sub-screen (Figure 24 (b)) to provide minimal coverage. Along all dimensions calculate the number of rectangles in a section. The number of rectangle(s) is the same at any position in a section. Then select a section having minimum number of rectangles. If there exists more than one section, then select a section which gives minimum coverage. The rectangle(s) in a selected section are eliminated from the sum of coverage.

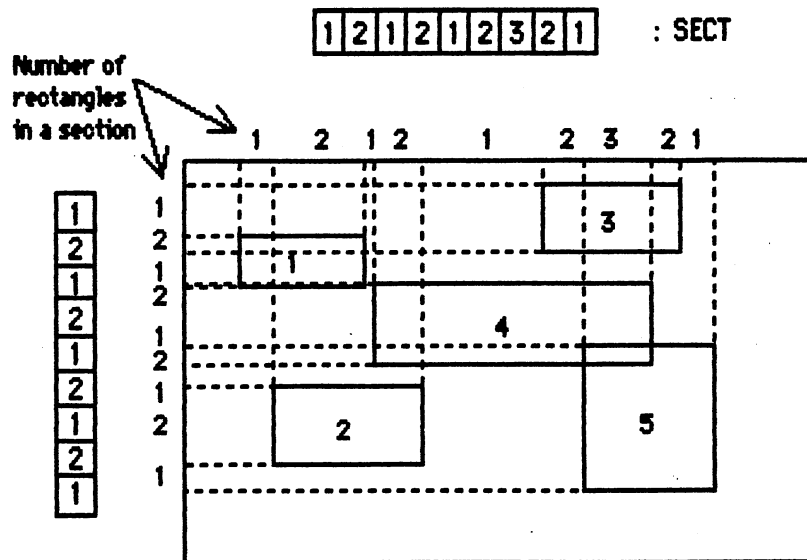


Figure 23. Rectangles Arrangement & Partition Sections

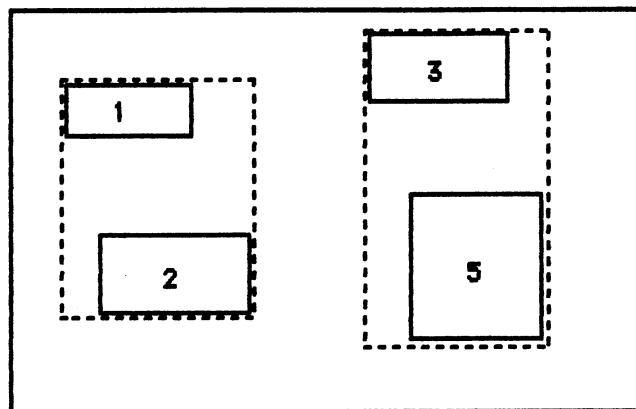


Figure 24. (a) The 1st Subscreen for Figure 23

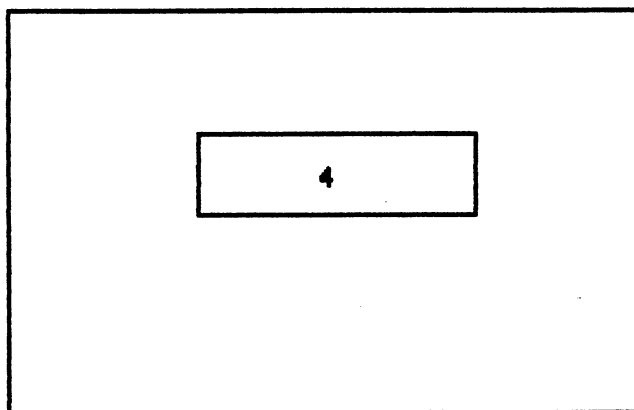


Figure 24. (b) The 2nd Subscreen for Figure 23



## CHAPTER IV

### ANALYSIS PREVIEW

#### Introduction

The purpose of this chapter is to provide the preview for analysis on the MR-tree by introducing analyses of the R-tree and the R<sup>+</sup>-tree. The contents of this chapter are based on [13]. Both data structures have been proposed to handle multi-dimensional data objects. The R<sup>+</sup>-tree is a variation of the R-tree [8]. The R<sup>+</sup>-tree differs from the R-tree in that it avoids overlapping between intermediate data rectangles by allowing partition of rectangles. The analyses are made for uniformly distributed line segments but can be generalized for objects in higher dimensions. The transformation of a data object into a point in higher dimension is introduced and the formulas to get the number of disk accesses in point query for the R-trees and the R<sup>+</sup>-trees are introduced. Also performance comparisons between those two data structures are given. As we expected, the R<sup>+</sup>-tree outperforms the R-tree in searching performance, especially in point query. The R<sup>+</sup>-trees has good performance in the cases where there are few long segments and many small ones [13].

The main operations that have been considered by other researchers in the past can be summarized as point query and range query. Of course, insertion, deletion, and modification have to be supported in a dynamic

environment. In this chapter, performance analyses are made on point query only.

### Assumptions

If we consider boxes as points in 4-dimensional space, the proofs of forthcoming analyses can be easily derived [13]. To represent a box aligned with the axes, 4-coordinates are needed (the x and y coordinates of the lower-left and upper-right corners). Since 4-d spaces are difficult to illustrate, examine line segments (1-d objects) instead of boxes (2-d objects) and 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. In the case of line segments, the screen collapses to a line segment, which, by convention, starts at 0 and ends at 1. Figure 25 shows some line segments and Figure 26 shows their 2-d representations.

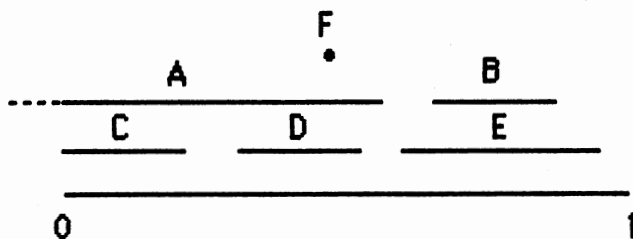


Figure 25. Line Segments on the Screen [13]

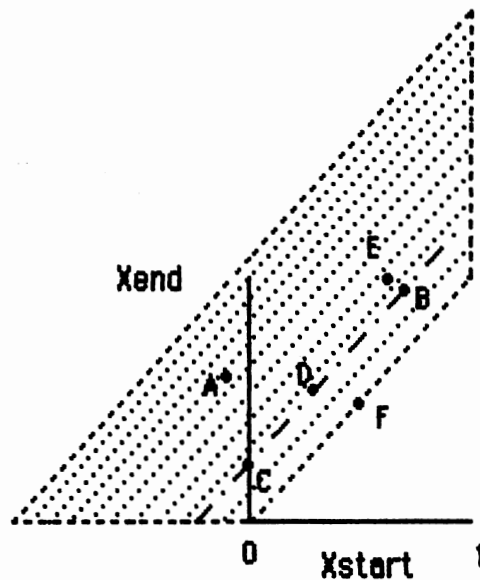


Figure 26. The 2-d Representation of the Line Segments of Figure 25 [13]

There exist several properties related to this transformation as follow [13]:

- 1) There are no points below the diagonal, since  $x_{start} \leq x_{end}$ .
- 2) Line segments of equal size like B and C in Figure 25, are represented by points that lie on a line parallel to the diagonal.
- 3) Line segments not entirely within the screen, such as A are allowed. In analysis, retain only screen clipped portion.
- 4) Points outside the shaded area in Figure 26 are of no interest, because the corresponding segments do not intersect with the screen.
- 5) The segments covering a given point  $X_0$  of the screen are transformed to points in the shaded area as shown in Figure 27.
- 6) The shaded area in Figure 28 correspond to all the segments intersecting with the segment  $(x_1, x_2)$ .
- 7) The shaded area in Figure 29 corresponds to all the segments covered completely by the segment  $S(x_1, x_2)$ .

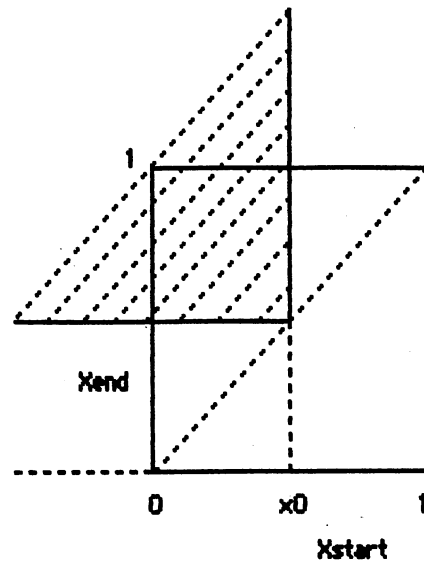


Figure 27. The Shaded Area Corresponds to Segments Containing the Point  $x_0$  [13]

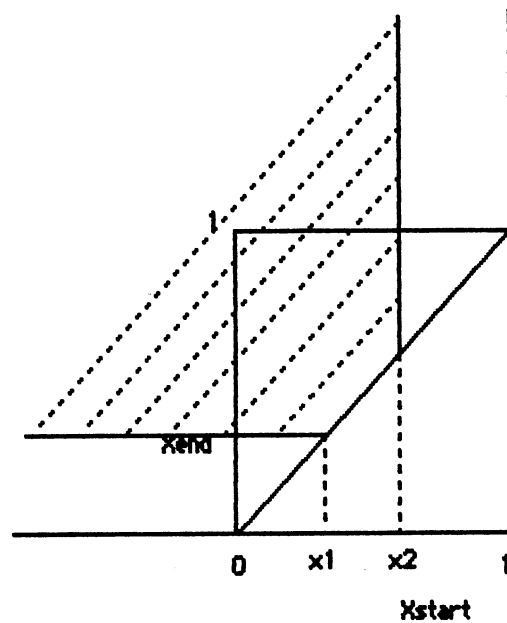


Figure 28. The Shaded Area Corresponds to Segments Intersecting with the Segment  $(x_1, x_2)$  [13]

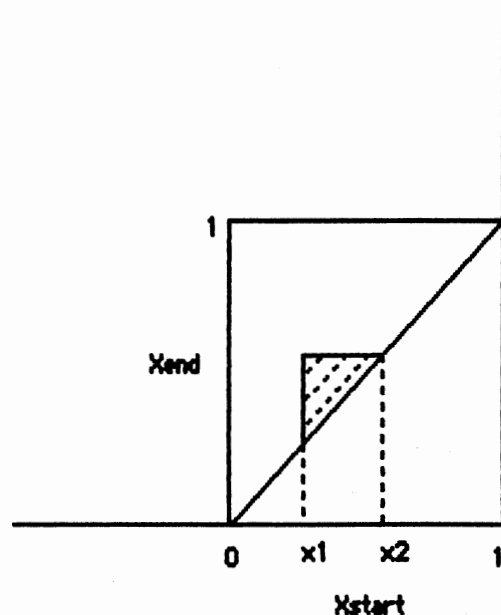


Figure 29. The shaded Area Corresponds to Segments Covered Completely by the Segment  $(x_1, x_2)$  [13]

We assume that [13]

- 1) the line segments of a given size are uniformly distributed,
- 2) they need not be totally within the screen.

The starting points of these segments divide the interval  $(-a, 1)$  into  $N+1$  equal subintervals, where  $N$  is the number of segments and  $a$  is the size. For example, in Figure 31 the starting points of each segment on the  $x_{start}$  divide  $(-a, 1)$  into 6 equal size subintervals.

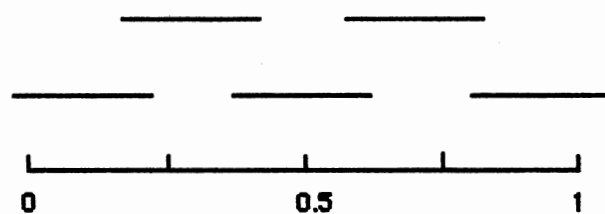


Figure 30.  $N=5$  Segments of Size  $a=.25$ , Uniformly Distributed on the Screen [13]

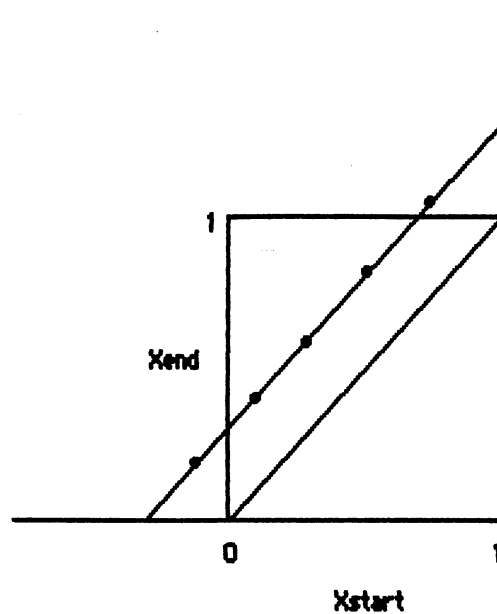


Figure 31. The 2-d Transformation of the Segments of Figure 30 [13]

The reason for considering the clipped-off portion of the segments is to maintain a constant overlap for the points on the screen. Let  $O_v$  (overlap) be the number of segments that contain a given point ( $O_v < C$ ).

#### Analysis of $R^+$ -tree

We examine two cases, one size case and two size case. In the one size case,  $N$  segments of size  $a$  uniformly distributed on the screen are used. In the two size case,  $N_1$  segments of size  $a_1$  and  $N_2$  segments of size  $a_2$  are considered. The segments of each set are uniformly distributed on the screen.

Parameter	Description [13]
C	The capacity of the data pages (= data records per page)
f	the fanout of the internal node of the tree: f sons per node
h	height of an R-tree (the root considered at level 1)
h+	height of an R+tree

Lemma: Given  $N$  segments of size  $a$ , uniformly distributed on the screen, a line segment (query segment) of size  $q$  intersects with  $\text{intersect}(N, a, q)$  segments [13], where

$$\text{intersect}(N, a, q) = \frac{(a + q)}{(1 + a)} (N + 1) \quad (1)$$

Proof: Consider Figure 32. We have line  $AB'$ , of size  $1+a$ , on horizontal axis by projecting the line  $AB$ . The query region intersects the line  $AB$  on a segment  $CD$  by property 6, whose projection  $C'D'$  is of size  $a+q$ . The fraction of the intersected segments (= points on line  $CD$ , or points on line  $C'D'$ ) is  $\text{length}(CD) / \text{length}(AB) = (a+q) / (1+a)$ . Since the line  $AB$  is divided into  $N+1$  equal intervals by the points, the line  $CD$  will contain on the average

$$\frac{a + q}{1 + a} (N + 1)$$

points [13]. This is exactly the number of line segments that intersect with given query segment  $q$ . This formula does not depend on the position of the

query segment. If  $q=0$ , that is query segment is a point, then we have the formula for the overlap:

Corollary: Given  $N$  segments with size  $a$ , uniformly distributed on the screen, the overlap is constant, and given by the formula [13]:

$$Ov(N,a) = \frac{a}{1+a} (N+1) \quad (2)$$

From (1) and (2) we have

$$\text{intsect}(N,a,q) = Ov(N,a) + q(N+1 - Ov(N,a)) \quad (1')$$

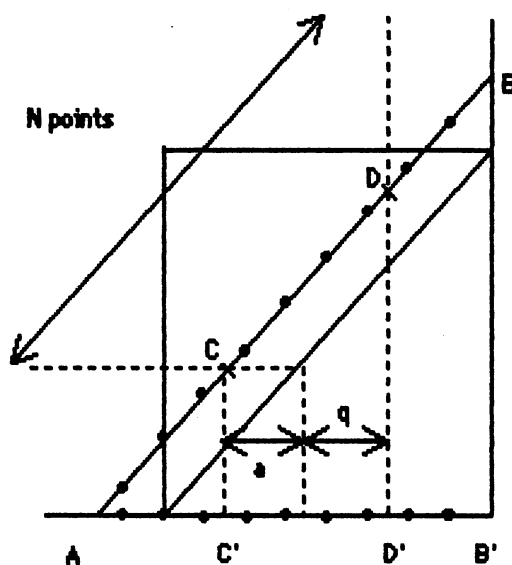


Figure 32.  $N$  Segments of Size  $a$  (represented as points), Uniformly Distributed on the Screen [13]

For example, in Figure 33, we can get the number of overlapping segments by equation:

$$Ov(N, a) = Ov(5, 0.5) = [0.5 / (1+0.5)] * (5+1) = 2.$$



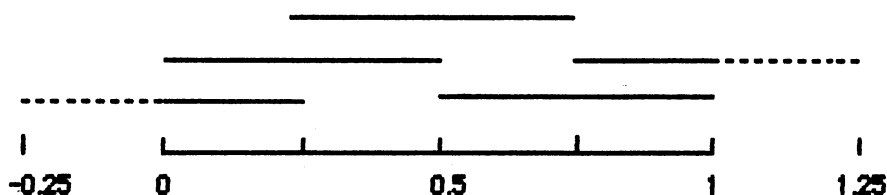


Figure 33. Segments of Size .5 each : overlap (Ov)= 2 [13]

### R<sup>+</sup>-trees, One-size Case

Assume that we have  $N$  segments of size  $a$ , uniformly distributed on the screen. Let  $h^+$  be the height of the R<sup>+</sup>-tree, which is assumed to be full, that is every data page contains  $C$  (capacity) entries, each internal node has  $f$  sons. The total number of data pages is  $f^{h^+}$  dividing the screen into  $f^{h^+}$  intervals. Each interval size is  $1 / f^{h^+}$  and one page corresponds to one interval. Since each page is full, we should have [13]

$$C = \text{intsect}(N, a, \frac{1}{f^{h^+}})$$

or

$$C = 0 + \frac{1}{f^{h^+}} (N + 1 - 0)$$

or

$$h^+ = \log_f \frac{N + 1 - 0}{C - 0}$$

Where  $O$  is the overlap  $O = Ov(N, a)$  of given set of segments [13]. The total number of disk accesses  $r^+da$  for point (exact) query is the height increased by one, to account for the retrieval of the data page [13].

$$r^+da = 1 + \log_f \frac{N + 1 - O}{C - O} \quad (3a)$$

or, since  $N \gg 1$  and  $N \gg 0$ ,

$$r^+ d_0 \doteq 1 + \log_f \frac{N}{C - 0} \quad (3b)$$

In a point query, the number of disk accesses to find all objects overlapping with a given point is just its height plus one. This means that all objects overlapping with a point can be found in one leaf node.

### R<sup>+</sup>-tree, Two-Size Case

Assume there are two sets of segments,  $N_1$  segments of size  $a_1$  and  $N_2$  segments of size  $a_2$ . Each set of segments is uniformly distributed on the screen respectively. Again a tree is assumed to be full. So we can use the same arguments as in one size case [13].

$$C = \text{intsect}(N, a_1, \frac{1}{r^{h^+}}) + \text{intsect}(N, a_2, \frac{1}{r^{h^+}})$$

or

$$C - O_{v1} - O_{v2} = \frac{1}{r^{h^+}} (N_1 + 1 - O_{v1} + N_2 + 1 - O_{v2})$$

or

$$h^+ = \log_f \frac{N_1 + N_2 + 2 - O_{v1} - O_{v2}}{C - O_{v1} - O_{v2}}$$

or

$$h^+ \doteq \log_f \frac{N}{C - O_v}$$

where  $N = N_1 + N_2$

$O_{v1} = O_v(N_1, a_1)$  the overlap due to set 1 segments

$O_{v2} = O_v(N_2, a_2)$  the overlap due to set 2 segments

$O_v = O_{v1} + O_{v2}$  the total overlap

Therefore, we have

$$r^+ d_0 \doteq 1 + \log_f \frac{N}{C - O_v} \quad (4)$$

Eq. (4) holds for any number of sets of segments (not just two) [13]. The result depends on the total number of segments and the total overlap.

### Analysis of R-trees

#### R-trees, One-Size Case

Again we assume that  $N$  uniformly distributed segments of size  $a$  are on the screen and that the tree is full. Therefore, the segments are grouped into  $N/C$  pages, in groups of size  $C$ . Each page is characterized by the "minimum enclosing segment", that covers all the segments in the page [13]. In the R-tree, the upper level rectangles just group lower level rectangles without concern about overlapping between rectangles. For example, in Figure 34, A is the minimum enclosing segment for the segments 1,2,3, and 4.

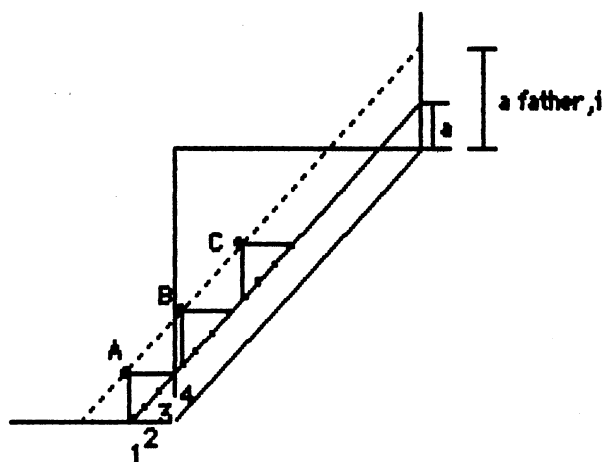


Figure 34. Illustration of Father of Level 1 ( $C=4$ ) [13]

The segments A,B,and C are referred to by "father of level 1" [13].

$$a_{\text{father},i} = \frac{(1+a)}{(N+1)}(C-1) + a \quad (5)$$

and they are also uniformly distributed on the screen [13].

Data pages are grouped into  $N / (Cf)$  groups, each group containing  $f$  data pages (therefore  $Cf$  segments), to form the lowest level of internal nodes of the R-tree ("father of level 2"). So the general equation is

$$a_{\text{father},i} = \frac{1+a}{N+1}(Cf^{i-1} - 1) + a \quad (6)$$

The fathers of level  $i$  are uniformly distributed on the scree [13]. The root of the R-tree corresponds to the father of level  $h+1$ . The  $Ov_{\text{father},i}$  for the father of level  $i$  is the number of fathers of level  $i$  containing a given point on the screen. The number of fathers in level  $i$  is [13]

$$N_{\text{father},i} = \frac{N}{Cf^{i-1}} \quad (7)$$

So  $Ov_{\text{father},i}$  is ,from the corollary (Eq. 2) [13]

$$Ov_{\text{father},i} = \frac{a_{\text{father},i}}{1+a_{\text{father},i}}(N_{\text{father},i} + 1) \quad (8)$$

which becomes

$$Ov_{\text{father},i} = 1 + \frac{Ov-1}{Cf^{i-1}} \quad (9)$$

To get the number of disk accesses in the R-tree for a point query, one needs to find the number of fathers which contain the given point. That is

$$rda = \sum_{i=1}^{h+1} O_{v, \text{father}, i} \quad (10)$$

where  $h$  is the height of the R-tree [13].

$$h = \log_r \frac{N}{C}$$

Finally, we have the following equation from (11) and (9) [13].

$$rda = h + 1 + \frac{O_v - 1}{C} \frac{f}{f-1} \left( 1 - \frac{1}{f^{h+1}} \right) \quad (12)$$

In equation (12), we assume that  $h$  is integer. If not, we can either use linear interpolation, or replace the sum of Eq. (10) with an integral from  $i=0.5$  to  $i=h+1.5$ . The difference in the two approximations is small (less than 2%) [13].

### R-tree, Two-Size Case

We consider two sets of segments,  $N_1$  segments of size  $a_1$  and  $N_2$  segments of size  $a_2$ . The segments in each set are uniformly distributed on the screen respectively. A tight packing of segments in data pages is obtained by storing  $C_1 = C * (N_1/N)$  and  $C_2 = C * (N_2/N)$  consecutive segments of set1 and set2 respectively in each data page, starting from left to right ( $N = N_1 + N_2$ ).  $C_1$  and  $C_2$  are referred to by effective capacity for the set1 and set2. Each set has  $N/C$  fathers of level 1, with sizes

$$a_{1,father,1} = \frac{1+a_1}{N_1+1}(C_1-1)+a_1$$

$$a_{2,father,1} = \frac{1+a_2}{N_2+1}(C_2-a)+a_2$$

by (5) [13]. This means that every father of large size covers completely the corresponding father of smaller size. If the subscript  $dom$  represents the dominating set, the R-tree behaves exactly the same as if [13]

- we had  $N_{dom}$  segments of size  $a_{dom}$
- the fanout was  $f$  as before
- the data page capacity was  $C_{dom}$

The only effect of the dominated set is that it occupies a fraction  $1-N_{dom}/N$  of the data pages reducing their effective capacity to  $C_{dom}$  for elements of the dominating set. So we obtain the following formula for the two size case [13].

$$rda = h + 1 + \frac{D_{vdom} - 1}{C_{dom}} \frac{f}{f-1} \left(1 - \frac{1}{f^{h+1}}\right) \quad (13)$$

where

$$h = \log_f \frac{N}{C} = \log_f \frac{N_1}{C_1} = \log_f \frac{N_2}{C_2}$$

and

$$dom = \begin{cases} 1 & \text{if } a_{1,father,1} > a_{2,father,1} \\ 2 & \text{otherwise.} \end{cases}$$

The Equation (13) holds for any number of sets of segments (not just two).

## Analytical Results

The number of disk accesses between the R-tree and the R<sup>+</sup>-tree are compared in the case of point query. The performance gain can be defined as following [13]:

$$\text{Perf.Gain} = ((rda - r+da) / (rda)) * 100$$

where rda is the number of disk accesses in the R-tree and r+da is the number of disk accesses in the R<sup>+</sup>-tree.

### One-Size Case

Figure 35 and Figure 36 show the number of disk accesses required to handle point query in an R-tree and R<sup>+</sup>-tree for a file of N segments with  $O_v$  overlap. The R-tree has worse performance than the R<sup>+</sup>-tree because more searching required.

In Figure 37, the gain in performance decreases as the total number of segments increases since the R<sup>+</sup>-tree creates many sub-segments trying to keep all intermediate node segments non-overlapping. Up to 30% improvement can be achieved [13].

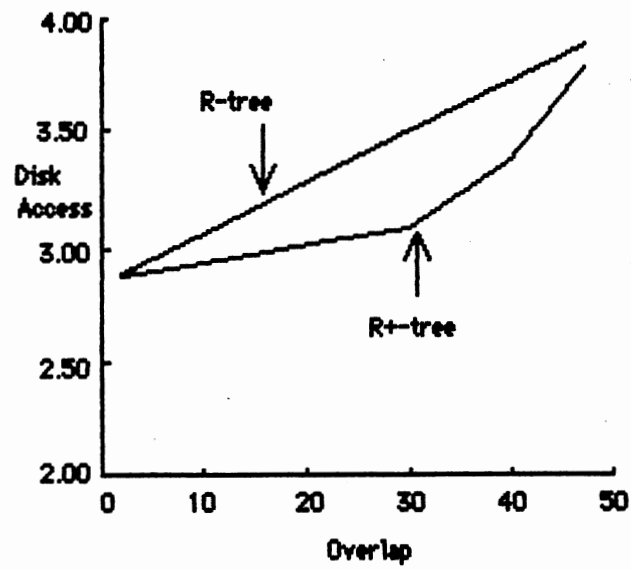


Figure 35. Disk Access for One-size Segments, as a Function of  $Ov$ ;  $N=100,000$  [13]

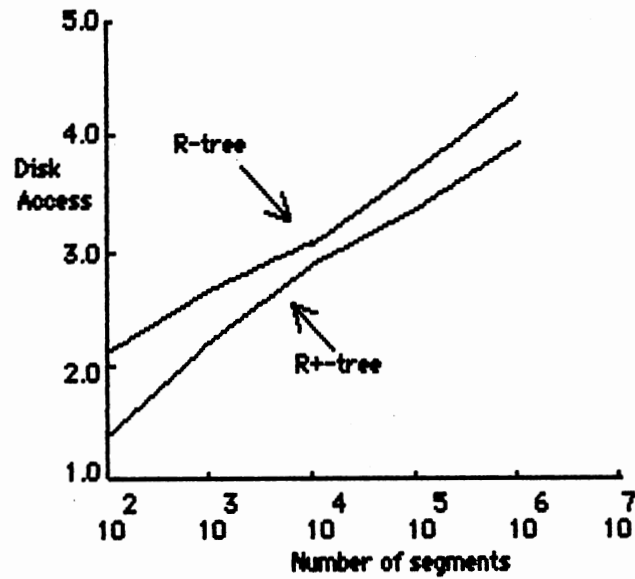


Figure 36. Disk Accesses for One-Size Segments, as a Function of  $N$ ;  $Ov= 40$  [13]



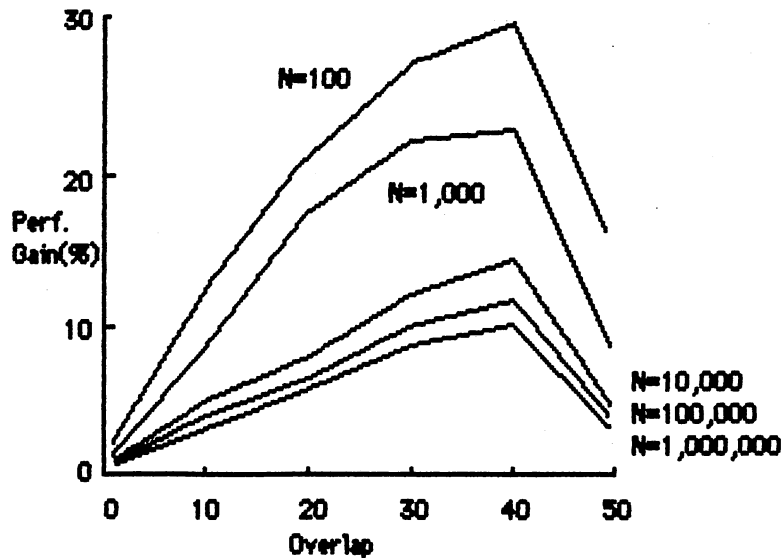


Figure 37. Performance Gain for One-Size Segments, as a Function of  $Ov$ ;  $N = 100$  to  $1,000,000$  [13]

### Two-size Case

Figure 38 and Figure 39 show the disk accesses between the R-tree and the R<sup>+</sup>-tree in point query used to index  $N_1 + N_2 = 100,000$  segments with total overlap  $Ov = Ov_1 + Ov_2 = 40$ . Figure 39 shows the number of disk accesses as a function of the number of large segments  $N_2$ , when the small segment overlap  $Ov_1 = 5$ . These figures show the problem that the R-tree has in handling few, lengthy segments. Improvements up to 70% can be achieved [13]. As the number of lengthy segments increase, the R<sup>+</sup>-tree loses its performance because of many splits necessary to prevent intermediate node overlapping. But lengthy segments are fewer than small segments in actual situation (e.g., in a VLSI design) [13].

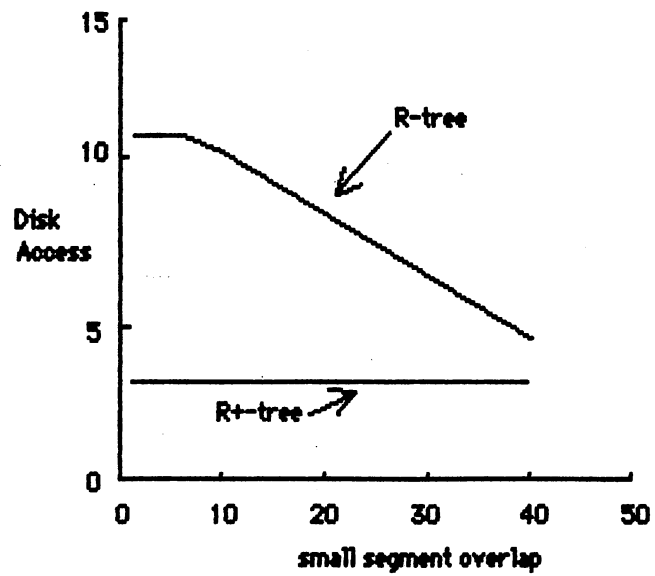


Figure 38. Disk Accesses for Two-Size Segments, as a Function of  $Ov1$ ;  $N2= 10,000$  [13]

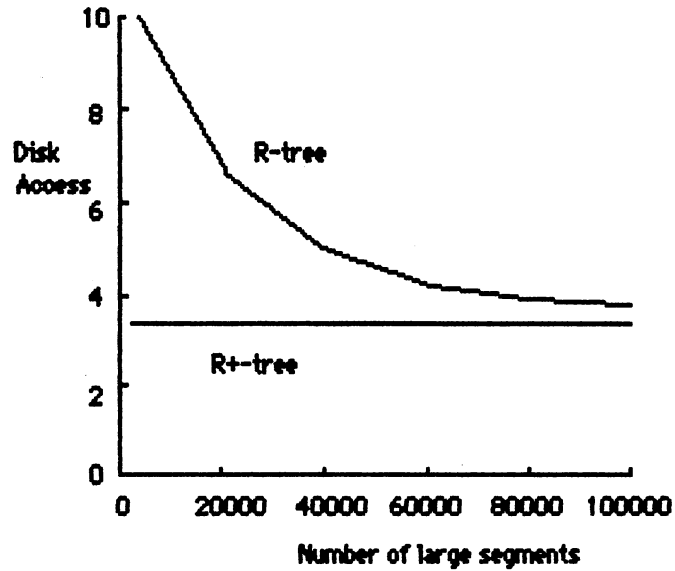


Figure 39. Disk Accesses for Two-Size Segments, as a Function of  $N2$ ;  $Ov1=5$  [13]

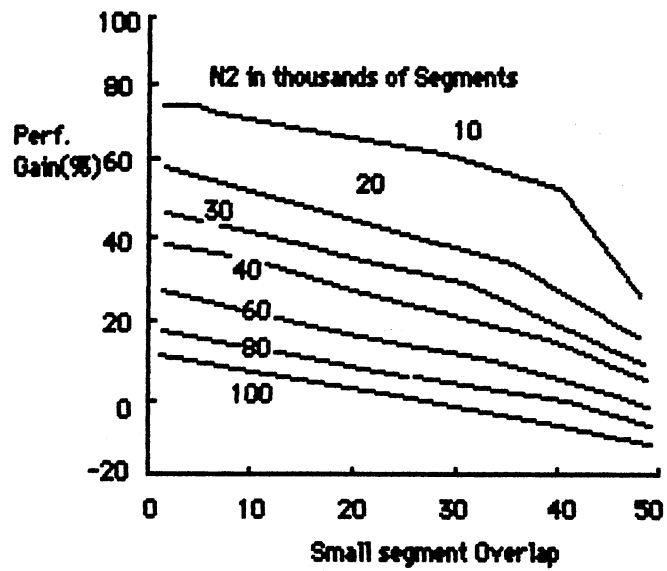


Figure 40. Performance Gain for Two-size Segments, as a Function of  $Ov_1$ ;  $N_2 = 10,000$  to  $100,000$ ,  $N = 100,000$  [13]

## CHAPTER V

### ANALYSIS OF THE MR-TREE

#### Introduction

Each subtree in the MR-tree has the same structure as the  $R^+$ -tree except that there are no duplicated data objects at leaf nodes. We consider point query and range query to analyze its performance. Line segments are used again instead of boxes and they are transformed into points in 2-d space by their starting and ending coordinates. Detailed methodology for this transformation was given in Chapter IV.

In the  $R^+$ -tree, the number of disk accesses is the height of the tree increased by one in point query. Since the  $R^+$ -tree does not allow overlapping between intermediate rectangles, there exists only one entry in the node which overlaps with the given point in each level if there exist overlapping objects at leaf level node. Eq. 3 (b) in Chapter IV denotes the number of disk accesses of the  $R^+$ -tree in point query.

We examine two cases as in Chapter IV, one size case and two size case. In the one size case, we consider  $N$  segments of size  $a$ , uniformly distributed on the screen. In the two size case, we consider  $N_1$  segments of size  $a_1$  and  $N_2$  segments of size  $a_2$ , the segments of each set are distributed uniformly on the screen respectively.

## Point Query

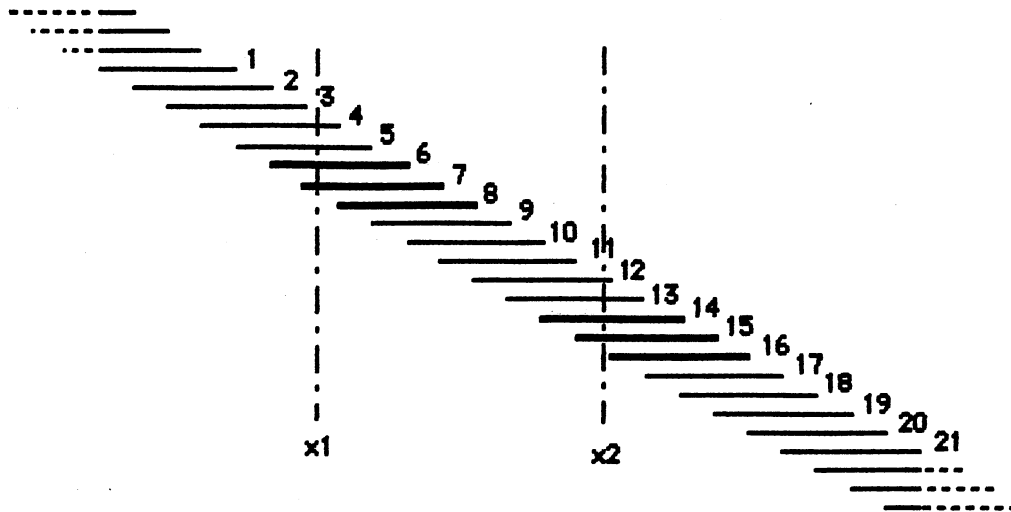
One Size Case

Figure 41. Uniformly Distributed Line Segments of Size a

Figure 41 shows uniformly distributed line segments. Let's assume that capacity ( $C$ ) of a node is 5 and overlap ( $O_v$ ) is 4 as above. If we use the  $R^+$ -tree to organize these line segments, we need 21 leaf nodes to hold all line segments. We can get this value from the Eq. (4) which estimates the number of disk accesses in a point query.

$$r^+da \approx 1 + \log_r \frac{N}{C - O_v} \quad (4)$$

where

$$\frac{N}{C - O_v} \text{ (Number of leaf level nodes)}$$

Such a property of the  $R^+$ -tree makes good performance in point query. The number of disk accesses of the  $R^+$ -tree in point query is just its height plus

one. For example, all line segments overlapping with a given point  $x_1$  in Figure 41 can be found in one leaf node ( $n_3$ ) in Figure 43.

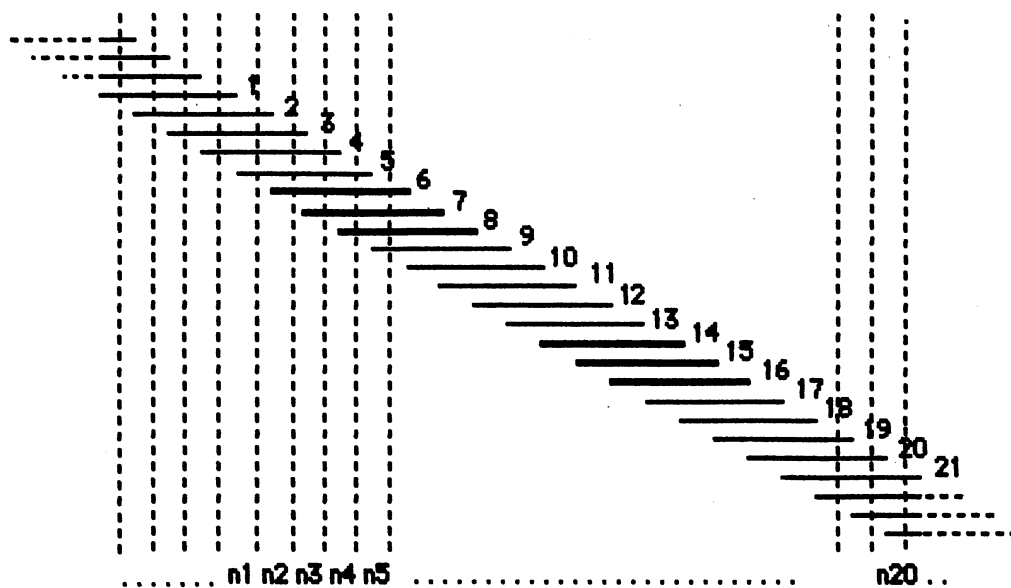


Figure 42. Data Pages when Using the R<sup>+</sup>-tree

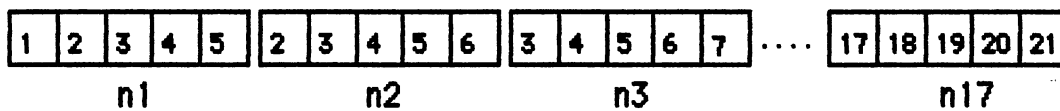


Figure 43. Overlap of Data Objects Between Nodes of the R<sup>+</sup>-tree

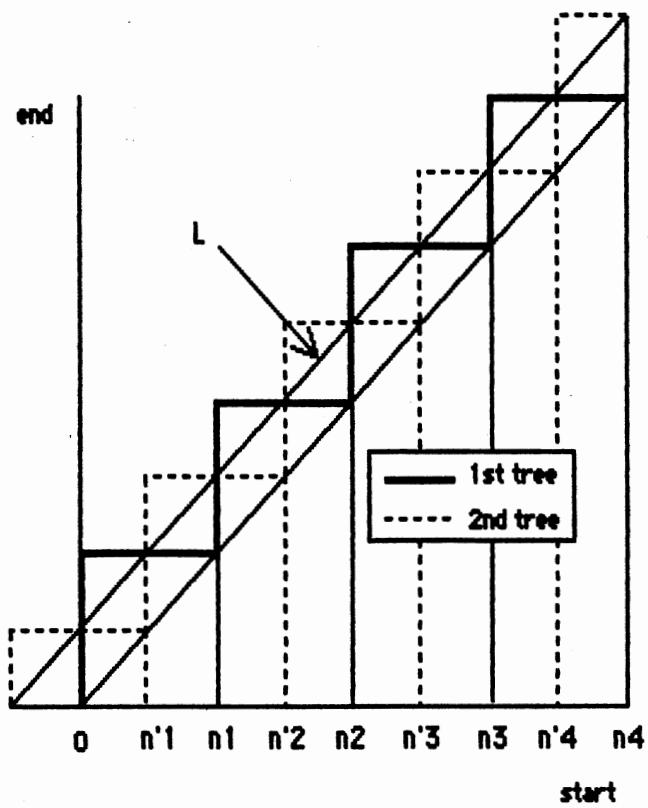


Figure 44. One Size Case Screen Dividing

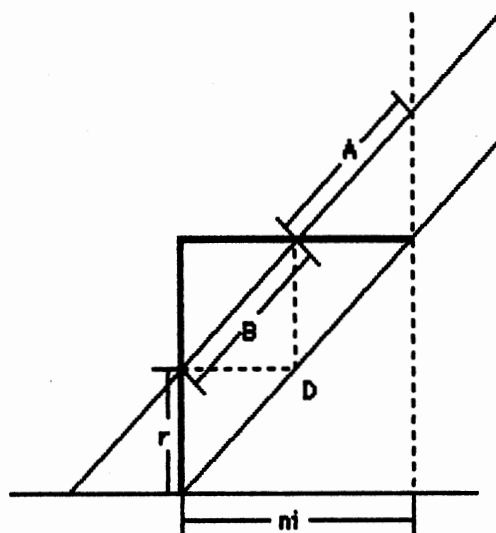


Figure 45. Magnified Picture of Figure 44

In Figure 44, all uniformly distributed line segments are on the line L. Since we consider only one size case, there exists only one line parallel with the diagonal. The capacity of a node is greater than or equal to the overlap,  $C \geq O_v$ . This means that in Figure 45, an enlargement of a segment of Figure 44, the length of the line segment A is smaller than or equal to that of line segment B. Therefore, as in Figure 44, two subtrees are enough to enclose all line segments in the one size case.

Now, the number of nodes in the leaf level can be obtained. The number of leaf nodes in the 1st subtree is the same as the number of stairs in Figure 44. One section of a "stair step" consists of two parts, capacity and overlap. So can get the equation

$$S_1 = \frac{N}{C + O_v} \quad (14)$$

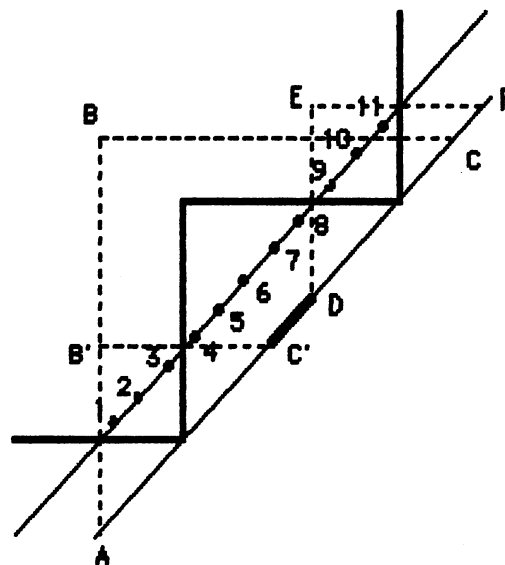


Figure 46. Illustrates the Splitting Algorithm of the MR-tree

For example, in Figure 46, if we assume that  $O_v$  is 3 and  $C$  is 5 then a node of the 1st subtree can contain as many line segments as its capacity. A



stair (ABC) holds 5 line segments, 1,2,3,9 and 10. However, insertion of line segment 11 makes the node overflow and the node needs to be split by splitting algorithm. The MR-tree splitting algorithm finds out that a disjoint zone exists between line 3 and 9, represented by thick line C'D. This property of splitting algorithm is to reduce the number of subtrees. Therefore, line segments in ABC splits into two nodes, AB'C' and DEF, and those two nodes can stay in the same subtree S2 without overlapping. The number of leaf nodes in the 2nd subtree can be obtained by

$$S_2 = \frac{S_1}{\left\lfloor \frac{C}{Oy} \right\rfloor} \quad (15)$$

where  $\left\lfloor \frac{C}{Oy} \right\rfloor$  is used to satisfy the property of the MR-tree split algorithm as mentioned above.

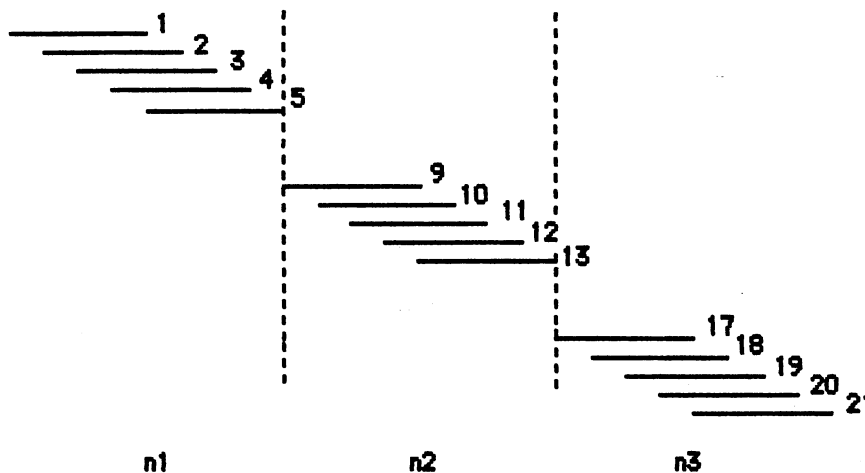


Figure 47. The 1st Subscreen of the MR-tree for Figure 41

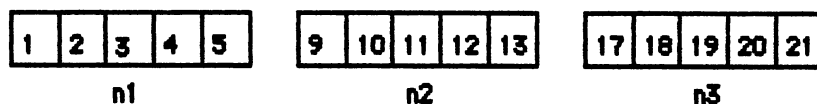


Figure 48. Arrangement of Data Objects in Leaf Nodes for Figure 47

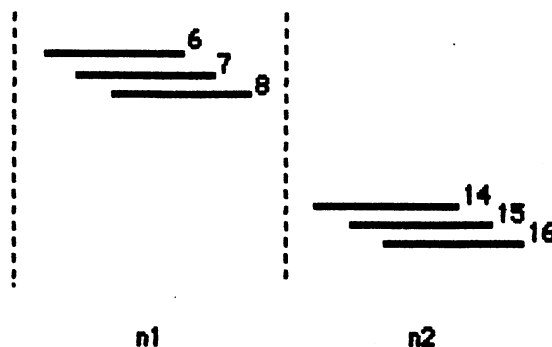


Figure 49. The 2nd Subscreen of the MR-tree for Figure 41

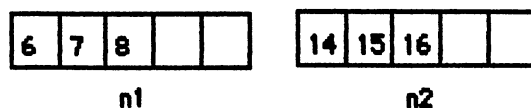


Figure 50. Arrangement of Data Objects in Leaf node for Figure 49

Figures 47 through Figure 50 show the organization of line segments using the MR-tree. The number of disk accesses is the height of a subtree plus one. The height of a subtree is

$$h_i = \log_p S_i$$

where  $h_i$  is the height of  $i$ th subtree. So the number of disk accesses in point query (Mda) is

$$Mda = \sum_{i=0}^n (1 + h_i) \quad (16)$$

where  $n$  is the number of subtrees (1 size case,  $n=2$ ).

If parallel processors are used, searching for each subtree can be executed simultaneously and can reduce the time to access overlapping data objects in a point query. For example, the MR-tree for Figure 41 has  $S_1$  (the number of data pages for subtree 1) =  $N / (C + O_v) = 21 / 9 = 2.3$  and  $h_1$  (the height of subtree 1) =  $\log_f S_1 = \log_f 2.3 = 0.5$

As we can see in Figure 47, the actual number of disk accesses of the 1st subtree is one. On the other hand, the  $R^+$ -tree has 1.9 as its height which can be obtained by the same way. This means that the number of disk accesses in the  $R^+$ -tree for point query is 2.9. Parallel processing of the MR-tree can result in reducing disk access time by almost 200% in this simple example. Even though parallel processing requires time for communication, it is nominal.

### Two Size Case

We consider two sets of segments,  $N_1$  segments of size  $a_1$  and  $N_2$  segments of size  $a_2$ . The segments of each set are uniformly distributed on the screen. Figure 51 shows an arrangement of two different sized line segments. For each set of segments in Figure 52, we can obtain  $O_v$  (the number of segments which contain given point  $x_0$ ). For example, if ( $N_1 = 6$  and  $a_1 = 0.5$ ) and ( $N_2 = 4$  and  $a_2 = 0.4$ ),  
 then  $O_{v1} = 0.5 (6+1) / (1+0.5) = 2.3$   
 $O_{v2} = 0.4 (4+1) / (1+0.4) = 1.4$

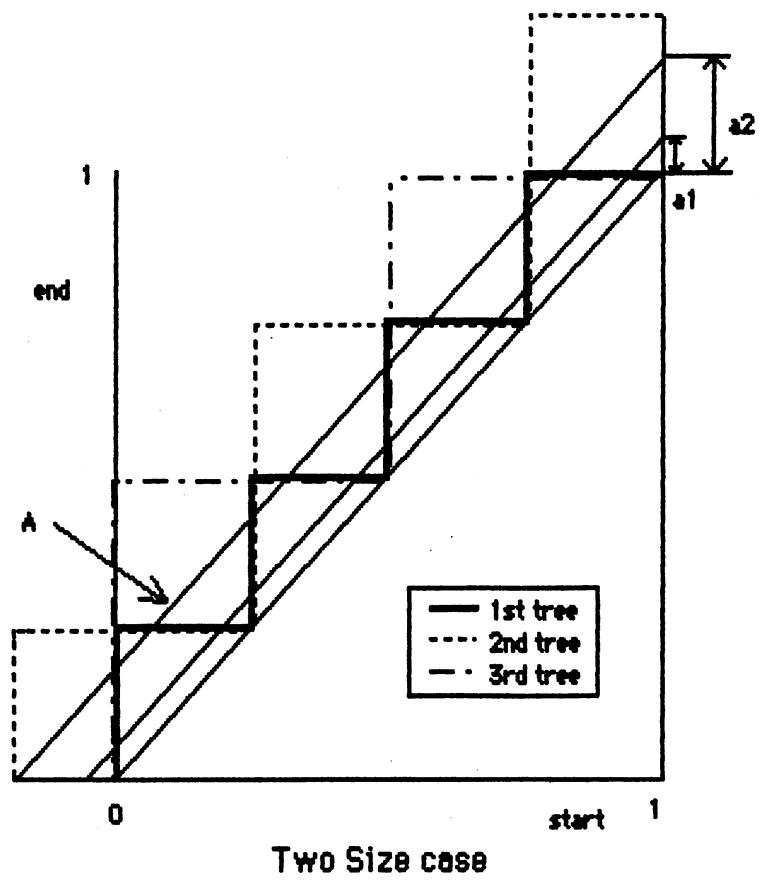


Figure 51. Screen Division in Two Size Case

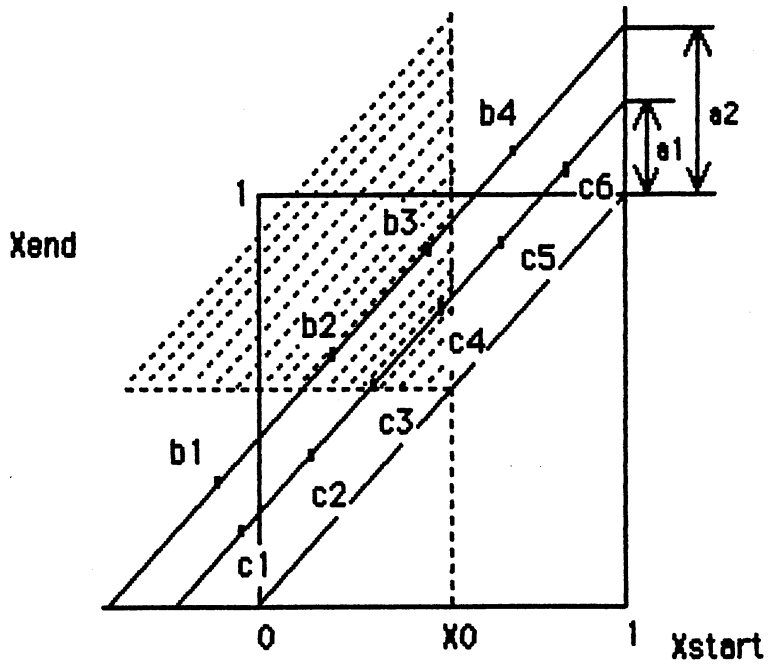


Figure 52. Overlapping Data Objects  
in Two Size Case

From this ,we can get the average overlap  $O_v$  for two size segment sets.

$$O_v = O_{v1} + O_{v2}$$

Figure 54, Figure 55, and Figure 56 show how data objects can be distributed into subtrees.

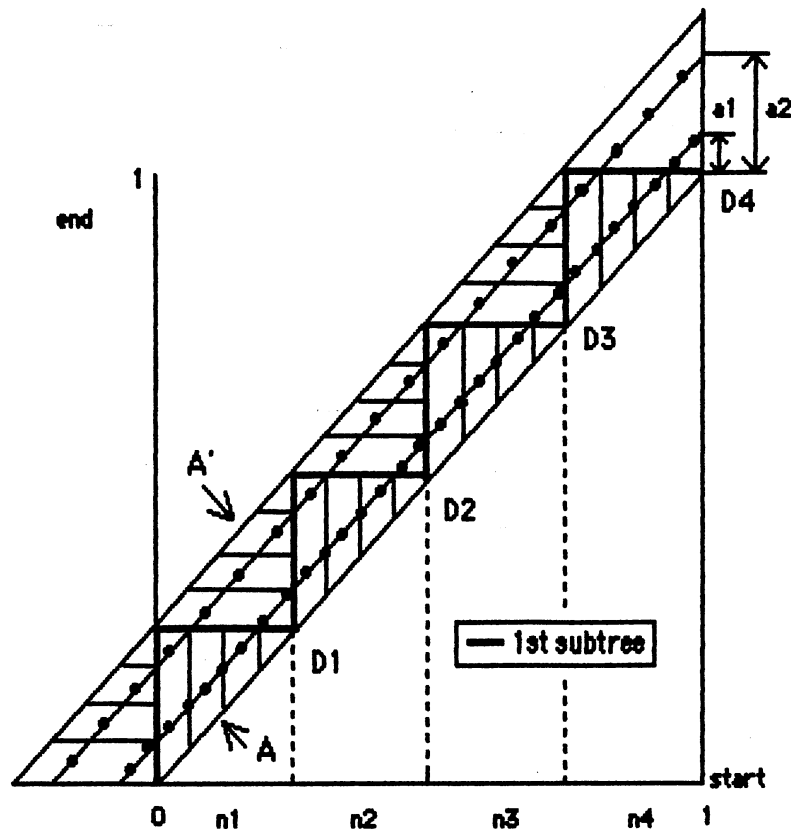


Figure 53. An Example of Screen Division in Two Size Case

In Chapter IV, we indicated that all line segments completely enclosed by a given line segment are under one section of the stair. Leaf nodes,  $n_1, n_2, n_3,$  and  $n_4,$  are disjoint with each other and the entries of each node are completely enclosed by their parent level entry.  $D_i$  represents disjoint point between leaf nodes,  $n_i$  and  $n_{i+1},$  of the 1st subtree. As in one size case, we can obtain the number of nodes at leaf level for each subtree. For the 1st sub-tree, the same equation as in one size case can be used

$$S_1 = \frac{N}{C + Dv}$$

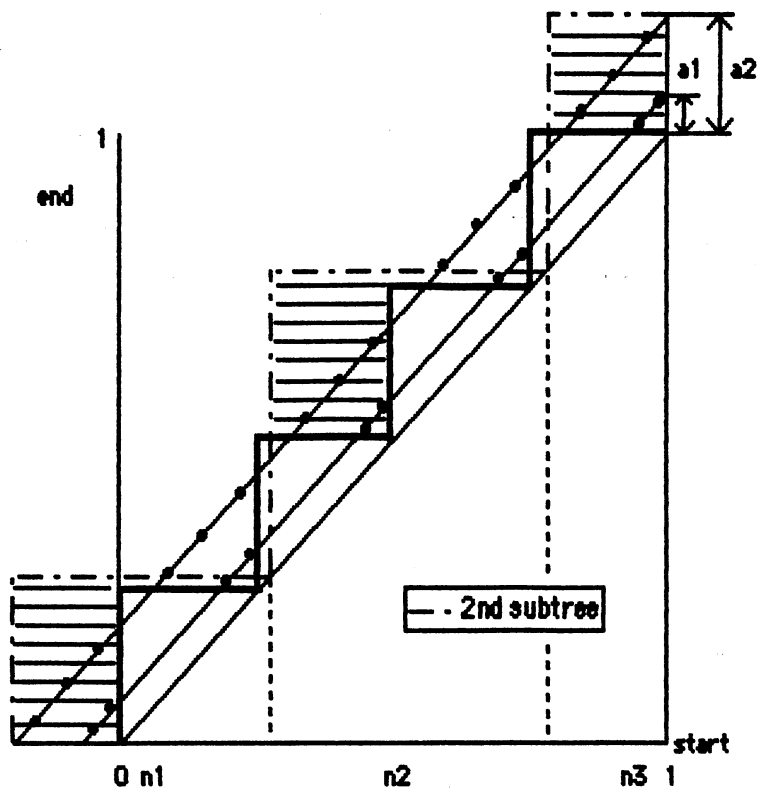


Figure 54 (a). Data Objects belonging to the 1st Subscreen

The shaded area in Figure 54 (a) includes line segments in the 2nd subtree. The 2nd subtree of the two size case is different from the one size case. Since disjoint point can not be guaranteed between sections (above the stair), the splitting algorithm sends selected line segment(s) from the 2nd subtree to next subtree to avoid overflow. The number of leaf nodes in the 2nd sub-tree is

$$S_2 = \frac{S_1}{r + \frac{C}{Dv}}$$

where  $r$  is the range of the segment sizes and the one stair length of the 1st subtree is as in Figure 54 (b). So

$$S_2 = \frac{S_1}{\frac{C}{O_v} + 1} \quad (17)$$

where  $C/O_v \geq 1$ , since  $C \geq O_v$ .

The 3rd subtree acts exactly the same way with the 2nd subtree in the one size case, since there exist disjoint points between sections. The number of leaf nodes in the 3rd subtree is

$$S_3 = \frac{S_2}{\lfloor \frac{C}{O_v} \rfloor} \quad (18)$$

where  $\lfloor \frac{C}{O_v} \rfloor$  is used to group the remaining line segments by using the splitting algorithm. So the heights of subtrees can be obtained as follow,

$$h_i = \log_2 S_i$$

where  $1 \leq i \leq n$  ( $n$  is the number of subtrees). Therefore, the number of disk accesses of the MR-tree in point query is

$$Mda = \sum_{i=0}^n (1 + h_i)$$

where  $n$  is 3.

If parallel processors are used, then searching for each tree can be executed simultaneously. Therefore, the time for disk accesses depends on maximum height of subtrees

$$T_{mda} = U * (\text{MAX}(h_i) + 1)$$

where  $T_{mda}$  is the time to access data objects in point query and  $U$  is the unit time required to access a disk page. Usually the 1st subtree has maximum height. For example, the 1st subtree for Figure 19 in Chapter III was taller than the 2nd subtree.



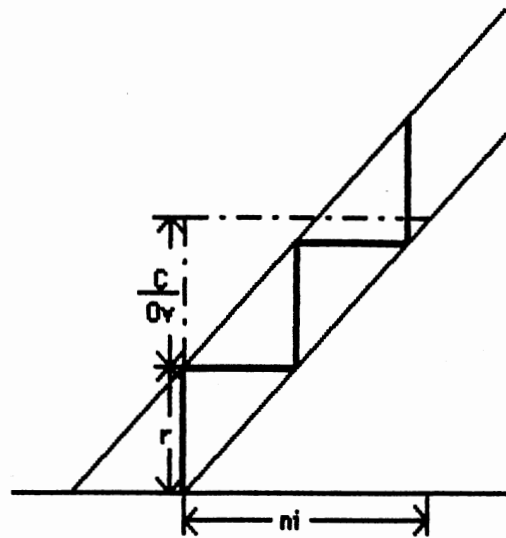


Figure 54 (b). Length of a Stair in the 1st Subtree.

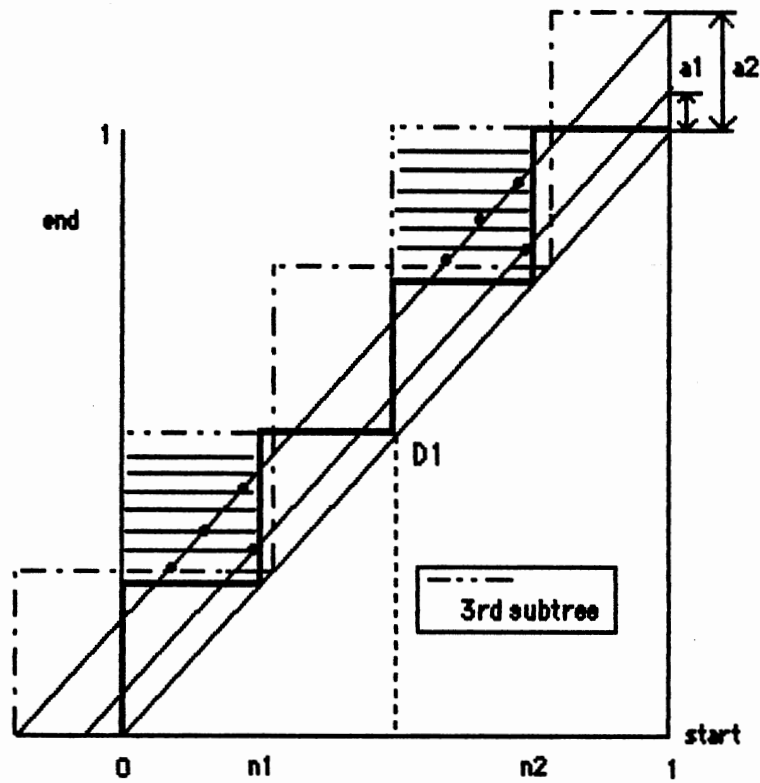


Figure 55. Data Objects belonging to the 3rd Subscreen

## Range Query

### One Size Case

Again we assume  $N$  segments of size 'a' uniformly distributed on the screen. In point query, the  $R^+$ -tree has a good performance since the duplicated data objects in leaf level nodes avoid overlapping between intermediate rectangles. When we consider range query instead of point query, the  $R^+$ -tree loses its performance by excessive data page accesses caused by redundant data objects in leaf level nodes. In Figure 56, the blackened area shows overlapping segments between page  $p_1$  and  $p_2$ . This means that all segments in blackened area exist in data pages  $p_1$  and  $p_2$ . In point query, such redundancy does not matter because we just need to find data objects overlapping with a given point. Such objects can be found in one leaf level node since  $0 \leq C$  (capacity). Leaf node redundancy of the  $R^+$ -tree increases the total number of data pages and results in more data pages being involved in search area. The number of redundant areas (blackened areas in Figure 56) is

$$f^{h^+} - 1$$

where  $f^{h^+}$  is the number of data pages in the  $R^+$ -tree. We can obtain this formula from the  $R^+$ -tree height equation in Chapter IV. In Figure 57, the number of data pages is 6. So the number of redundant areas is 5. Since  $1 \ll$  the number of data pages, the number of redundant areas can be considered as the same with the number of data pages.

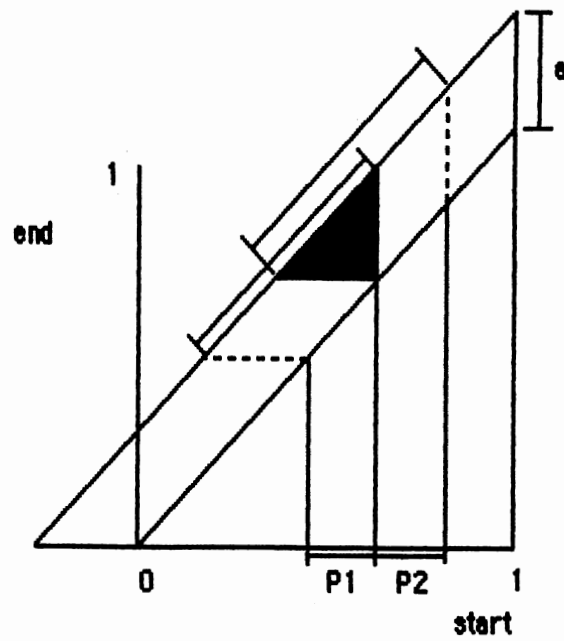


Figure 56. The Redundancy Between Leaf Level Nodes

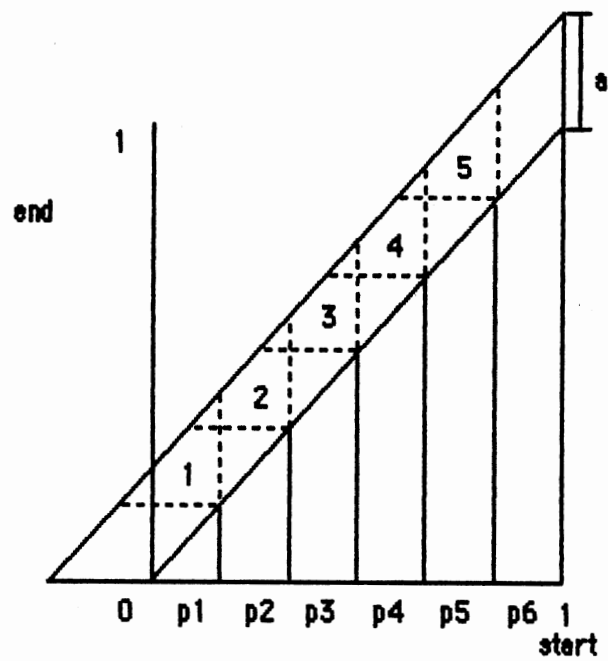


Figure 57. The Number of Redundant Area

The number of segments in 1 redundant area is the same as the number of segments overlapping with a point in point query. The number of segments in a redundant area does not depend on the size of data page. It depends on the size of segment. The total number of segments in a redundant area is

$$(f^{h^+} - 1) * O_v$$

$f^{h^+} * C$  : total number of segments in  $R^+$ -tree

$(f^{h^+} - 1) * O_v$  : total number of redundant segments in  $R^+$ -tree

The redundancy of the  $R^+$ -tree can be obtained from

$$\text{Redundancy in } R^+\text{-tree} = \frac{(f^{h^+} - 1) * O_v}{N} * 100 \quad (19a)$$

Another formula to get the redundancy of the  $R^+$ -tree is

$$\frac{\frac{N}{C - O_v} - \frac{N}{C}}{\frac{N}{C}} * 100 = \frac{O_v}{C - O_v} * 100 \quad (19b)$$

where  $N / (C - O_v)$  is the number of leaf nodes when  $O_v$  is not 0 and  $N / C$  is the number of leaf nodes when  $O_v$  is 0. Both formulas produce the same results. Figure 58 shows the redundancy of the  $R^+$ -tree as a function of  $O_v$  and Figure 59 shows the comparison of total number of nodes between the  $R^+$ -tree and the MR-tree. The  $R^+$ -tree's redundancy is increased when  $O_v$  is increased. If  $O_v$  is greater than 40, the redundancy of the  $R^+$ -tree is increased drastically.

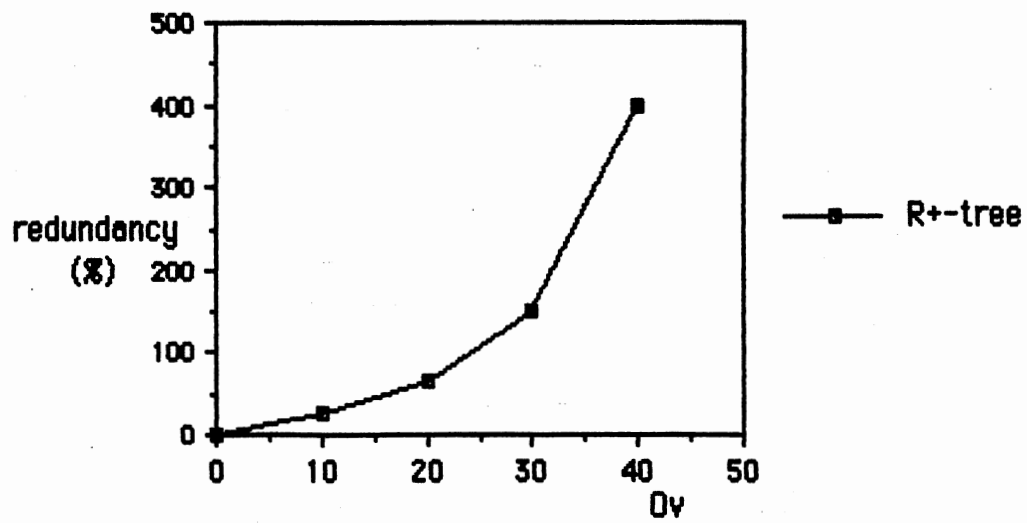


Figure 58. The Redundancy of the R<sup>+</sup>-tree as a Function of Ov

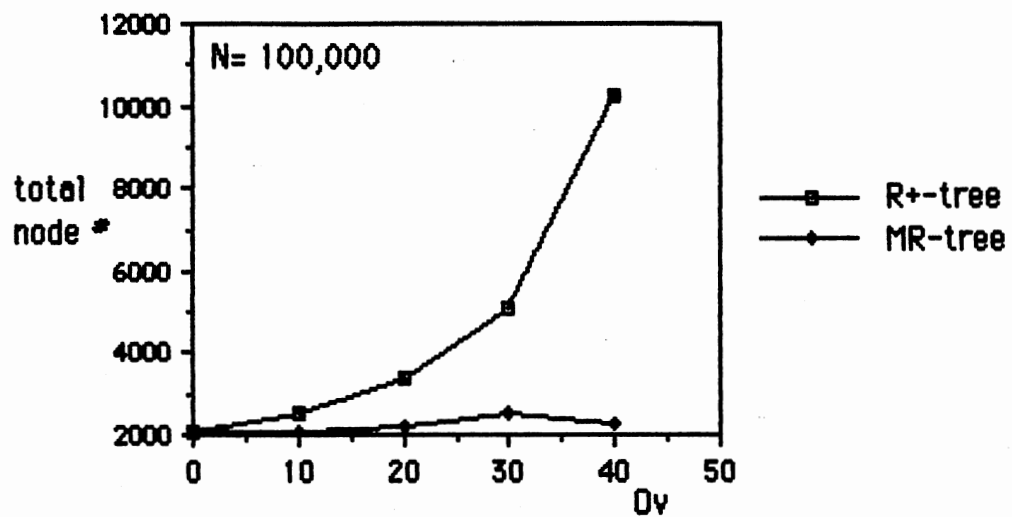


Figure 59. Total Number of Data Pages (leaf level) vs. Ov

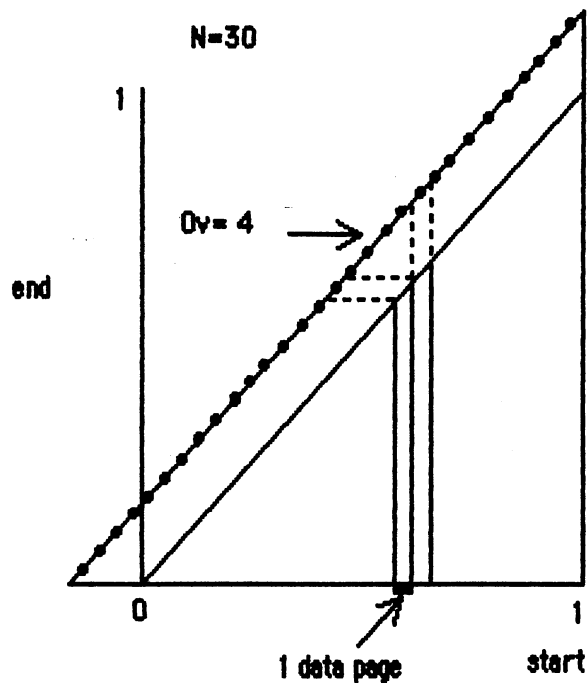


Figure 60. Redundancy in the R<sup>+</sup>-tree  
in Leaf Level Nodes

In Figure 60, for instance, if we assume 30 uniformly distributed segments of size 'a',  $Ov = 4$ ,  $C = 5$  and  $f = 5$  then the number of data pages required to hold 30 segments is 26 by Eq. (20). This means 20 data pages out of 26 are wasted. As redundancy is increased, the insertion and the deletion costs of the R<sup>+</sup>-tree are increased by splitting and also search cost, especially range search, is increased by excessive page accesses. The MR-tree has the advantage in range query as the size of search range and  $Ov$  are increased even for serial processing. In the R<sup>+</sup>-tree, the number of disk accesses can be obtained from height formula [13] discussed in Chapter IV.

$$Dp = \frac{N}{C - Ov} \quad (20)$$

Let SR represent search range. Then  $R^+_{dp}$  denotes the number of data pages involved in the search range of the  $R^+$ -tree.

$$R^+_{dp} = Dp * SR$$

$$R^+_{da} = \sum_{i=0}^{\lceil h \rceil} \left\lceil \frac{R^+_{dp}}{r^i} \right\rceil \quad (21)$$

where  $0 \leq SR \leq 1$ .  $R^+_{da}$  represents total number of disk accesses to retrieve all data pages overlapping with search range SR.

For the MR-tree, the number of data pages in leaf level overlapping with search area can be obtained as follow

$$M_{idp} = S_i * SR$$

where  $S_i$  denotes the number of data pages in  $i$ th subtree and  $M_{idp}$  represents the number of data pages overlapping with SR. Therefore, the number of disk accesses of the MR-tree for a range query is

$$M_{da} = \sum_{i=0}^n \sum_{j=0}^{\lceil h \rceil} \left\lceil \frac{M_{idp}}{r^j} \right\rceil \quad (22)$$

where  $n$  is the number of subtrees (in the one size case  $n=2$ ).

### Two Size Case

We assume two sets of segments,  $N_1$  segments of size  $a_1$  and  $N_2$  segments of size  $a_2$ . The segments of each set are uniformly distributed on the screen. In the two size case, only the number of segments  $N$  and  $O_v$  are different factors. As in the two size case of the point query,  $N = N_1 + N_2$  and  $O_v = O_{v1} + O_{v2}$ . So the same equations as in the one size case can be used for the two size case. In the two size case, 3 subtrees were enough to hold all

line segments. In two size case point query, we verified that 3 subtrees are enough to hold  $n$  size line segments. Therefore, Eq. (22) can be used for  $n$  size case.

Figure 61, Figure 62, Figure 63, and Figure 64 show point and range query performances between the  $R^+$ -tree and the MR-tree. In this analysis, we use search range (SR) from 0% (point query) to 1% of data space. As we can observe, the  $R^+$ -tree has the advantage over the MR-tree in point query. However, the increment of the search range gives the MR-tree advantage. A search range of 1% of the data space is considered as comparatively small size. Therefore, the MR-tree further increases its performance when search range is increased above 1% of the data space.

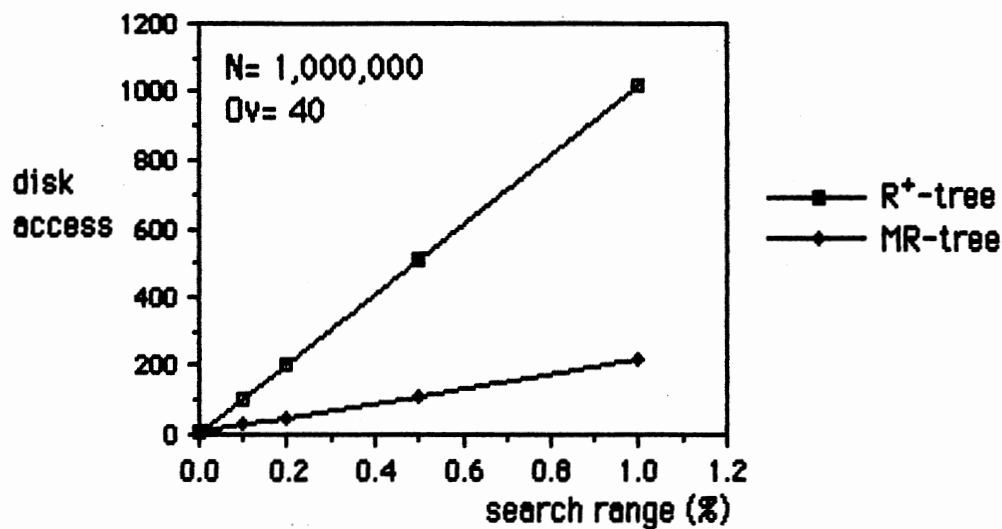


Figure 61. The Number of Disk Accesses vs. Search Range



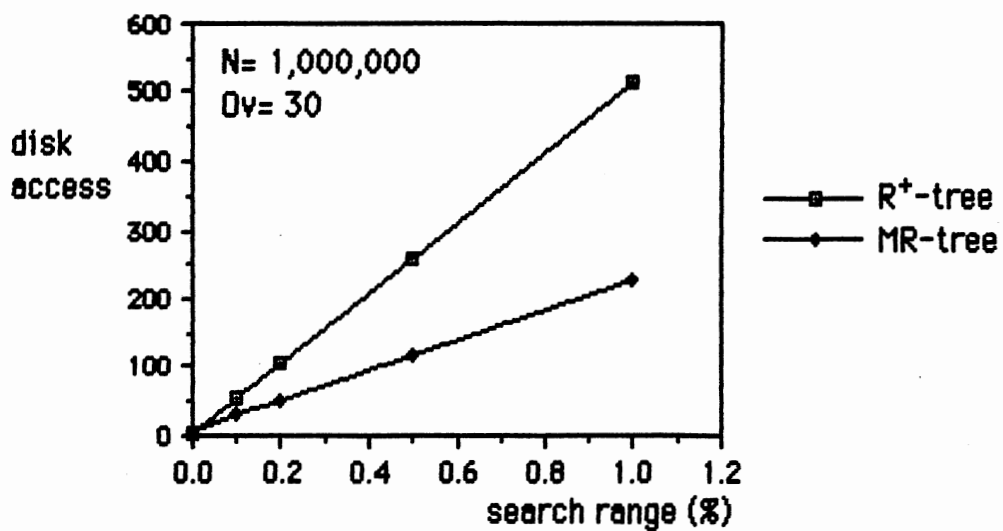


Figure 62. The Number of Disk Accesses vs. Search range

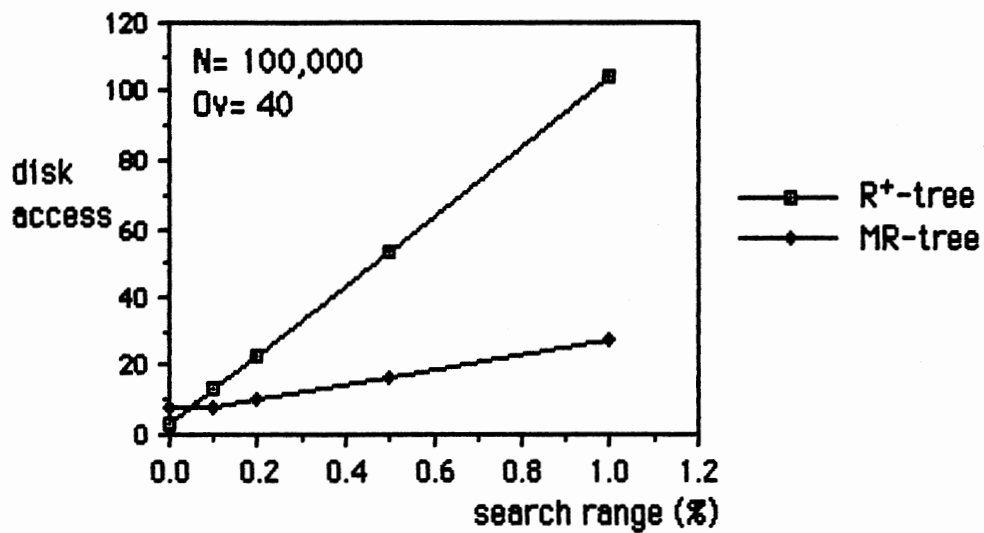


Figure 63. The Number of Disk Accesses vs. Search range

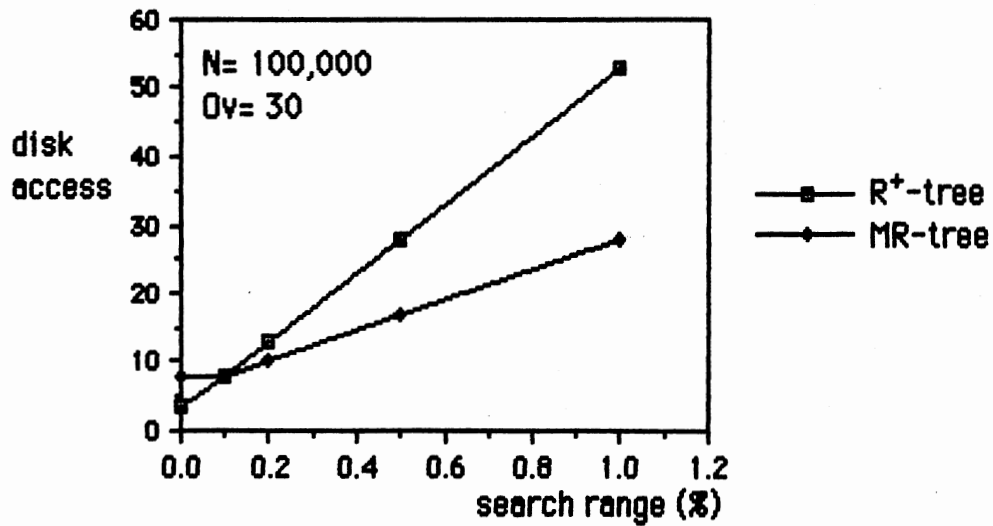


Figure 64. The Number of Disk Accesses vs. Search Range

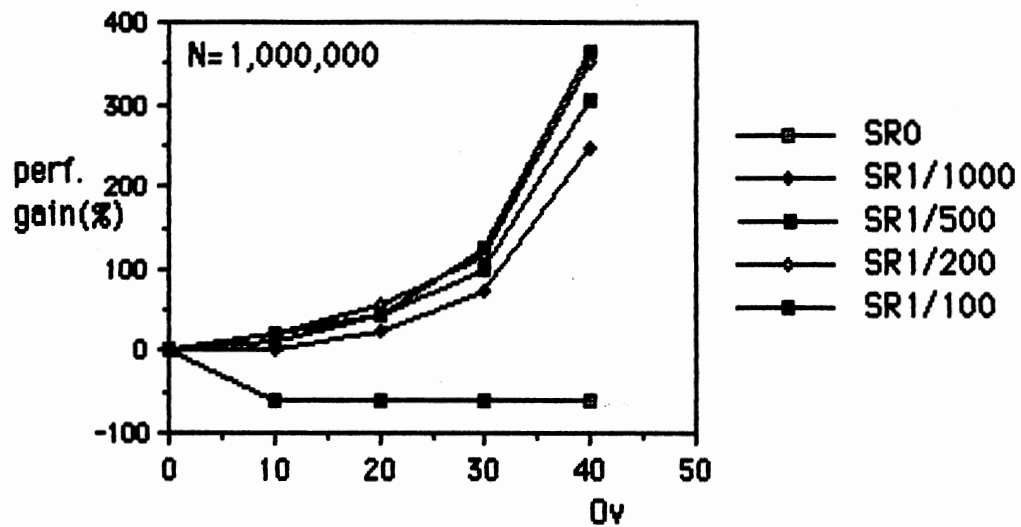


Figure 65. Performance Gain of the MR-tree (in serial processing) over the R<sup>+</sup>-tree as a Function of Ov

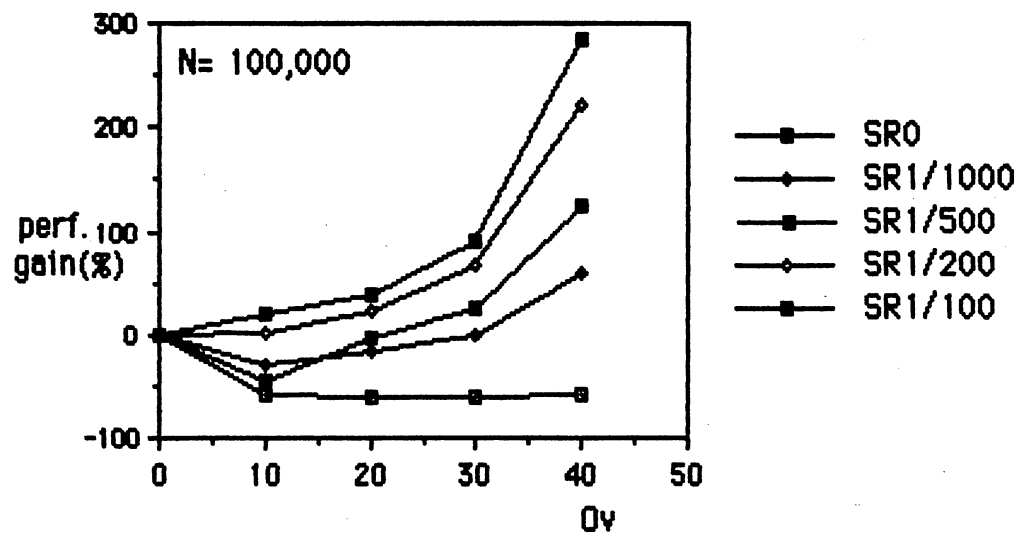


Figure 66. Performance Gain of the MR-tree (in serial processing) over the R<sup>+</sup>-tree as a Function of Ov

Figure 65 and Figure 66 show the performance gain of the MR-tree over the R<sup>+</sup>-tree as a function of Ov (cases of N= 1,000,000 and N= 100,000). We define performance gain as

$$\text{Perf. gain(\%)} = \frac{R^+da - MRda}{MRda} \times 100$$

where R<sup>+</sup>da is the number of disk accesses for the R<sup>+</sup>-tree and MRda is the number of disk accesses for the MR-tree. Performance gain of the MR-tree is proportional to Ov and search range. The MR-tree is an efficient data structure when the density of data is high on data space.

### Parallel Processing

As mentioned, parallel processing gives the MR-tree advantages for both range queries and point queries. In parallel processing of the MR-tree, each processor executes query operations on a subtree simultaneously.

There is no communication overhead or bottleneck time between processors since the host processor just needs to send the search or deletion range to all node processors and to receive the results from node processors. Figure 67 and Figure 68 show point query performances of the MR-tree for parallel processing and the R<sup>+</sup>-tree for serial processing. The MR-tree takes less access time to retrieve overlapping data objects than the R<sup>+</sup>-tree does. The MR-tree and the R<sup>+</sup>-tree has the same access time when  $O_v$  is 0 because the MR-tree has only 1 subtree. In point query, the disk access time of the MR-tree for parallel processing depends on the height of dominant subtree (usually the 1st subtree holds more data objects than the other subtrees do). For instance, if the MR-tree has 3 subtrees and the heights of the 3 subtrees are 5, 4, and 3 respectively then total disk access time of the MR-tree depends on the 1st subtree (height 5).

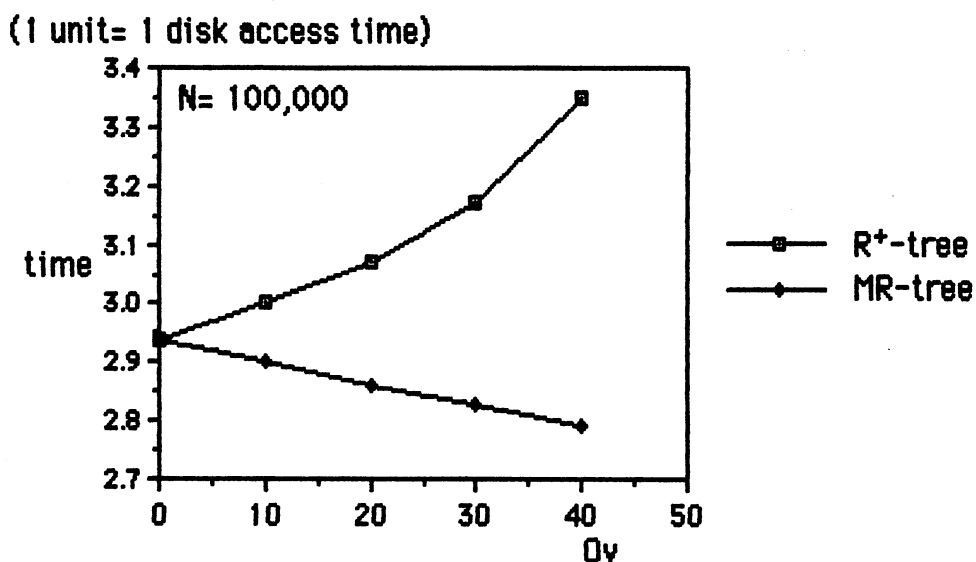


Figure 67. Time Comparison Between the MR-tree (in Parallel Processing) and the R<sup>+</sup>-tree (Point Query)

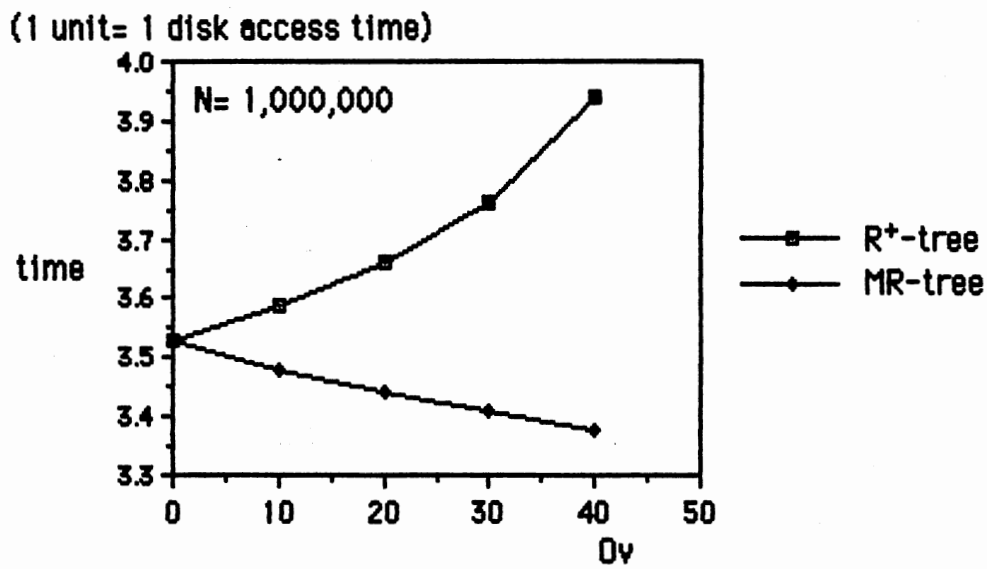


Figure 68. Time Comparison Between the MR-tree (in Parallel Processing) and the R<sup>+</sup>-tree (Point Query)

Figure 69 through Figure 72 compare range query performances of the MR-tree for parallel processing and the R<sup>+</sup>-tree. As Ov is increased, the MR-tree reduces query time. Because the MR-tree tends to reduce the number of rectangles in the dominant subtree(s) (usually the 1st subtree) when Ov is increased. However, the R<sup>+</sup>-tree increases the query time when Ov is increased. The number of nodes in the search range is increased since an increase in Ov increases the number of duplicated rectangles in the R<sup>+</sup>-tree. As mentioned in Chapter III, the MR-tree is efficient in deletion operation done by exact coordinates or deletion range. Also parallel processing can save many disk accesses by executing the deletion processes simultaneously.

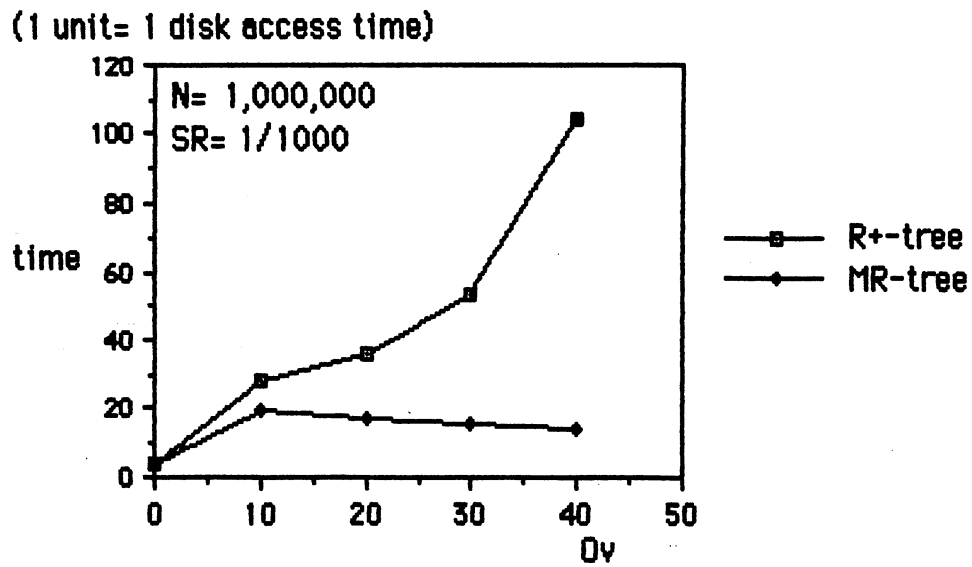


Figure 69. Time Comparison Between the MR-tree (in Parallel Processing) and the R<sup>+</sup>-tree (Range Query)

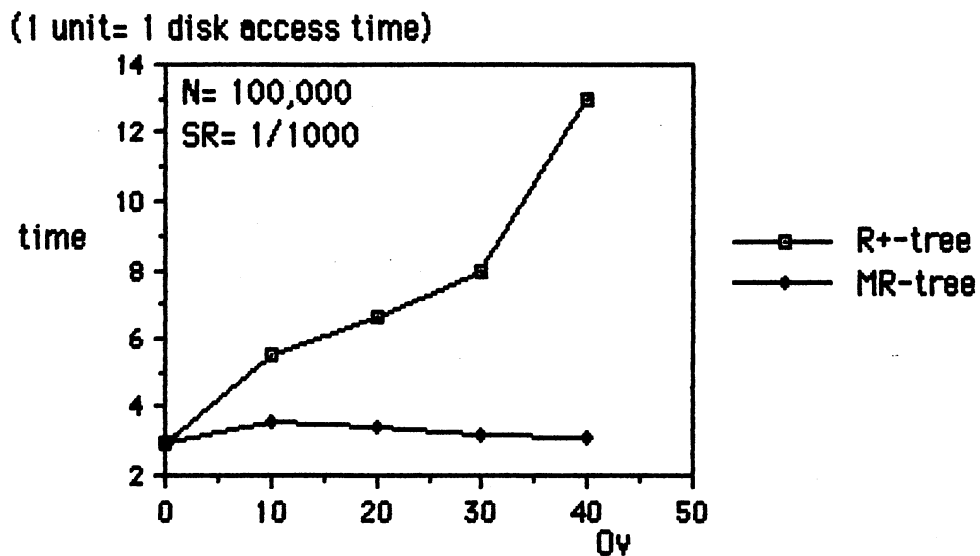


Figure 70. Time Comparison Between the MR-tree (in Parallel Processing) and the R<sup>+</sup>-tree (Range Query)

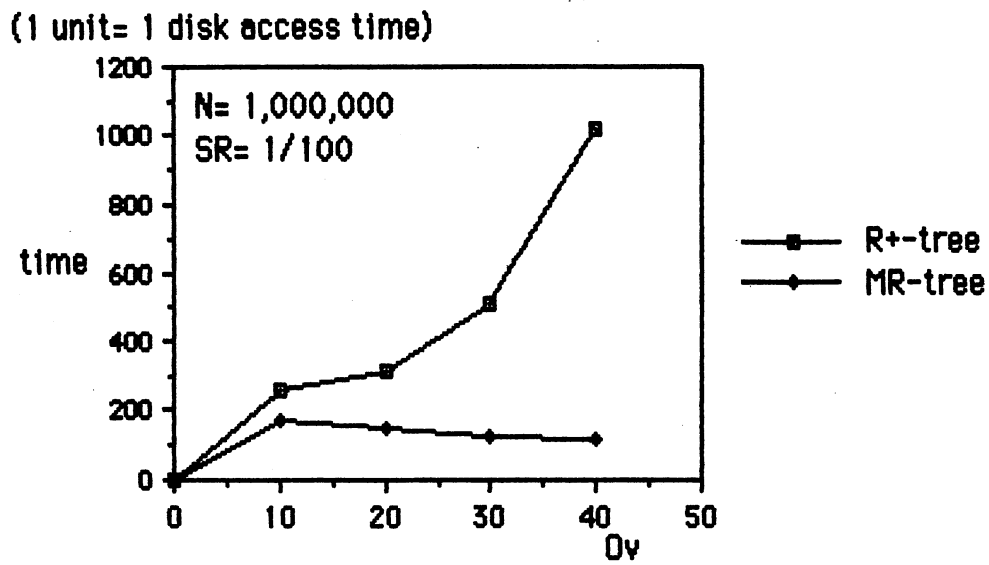


Figure 71. Time Comparison Between the MR-tree (in Parallel Processing) and the R<sup>+</sup>-tree (Range Query)

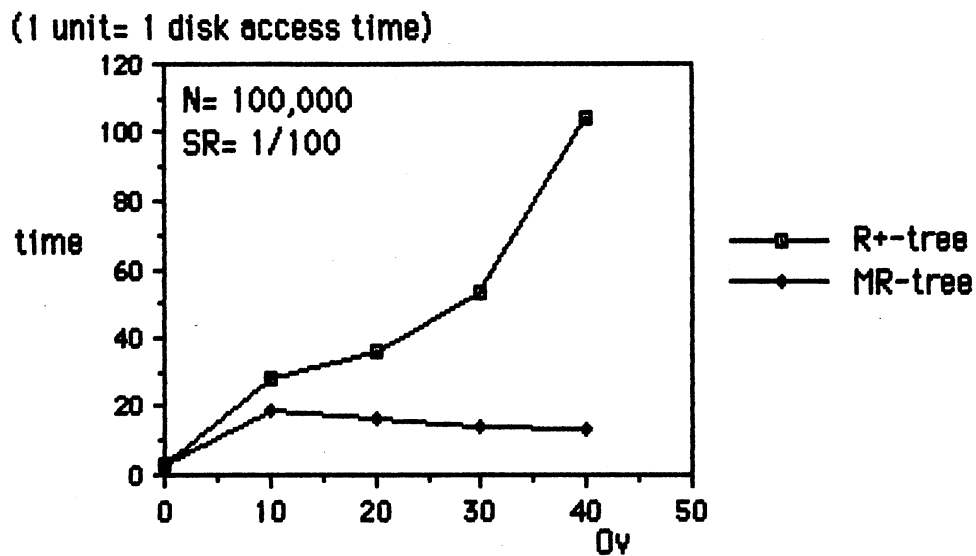


Figure 72. Time Comparison Between the MR-tree (in Parallel Processing) and the R<sup>+</sup>-tree (Range Query)

Figure 73 through Figure 76 show performance gains of the MR-tree for parallel processing over the R<sup>+</sup>-tree. In this case, we define perf. gain as

$$\text{Perf. gain(\%)} = \frac{R^+da - PMRda}{PMRda} \times 100$$

where PMRda is the maximum number of disk access among the subtrees. As we can observe, the MR-tree has better performances than the R<sup>+</sup>-tree does in point query and range query. Figures 73 and 75 show the MR-tree performance gain as a function of Ov. Figures 74 and 76 show performance gain as a function of search range. In deletion operation by exact coordinates of data object, the MR-tree has the same performance gain as in search operation. However, the MR-tree further increases its performance in deletion operation by deletion range.

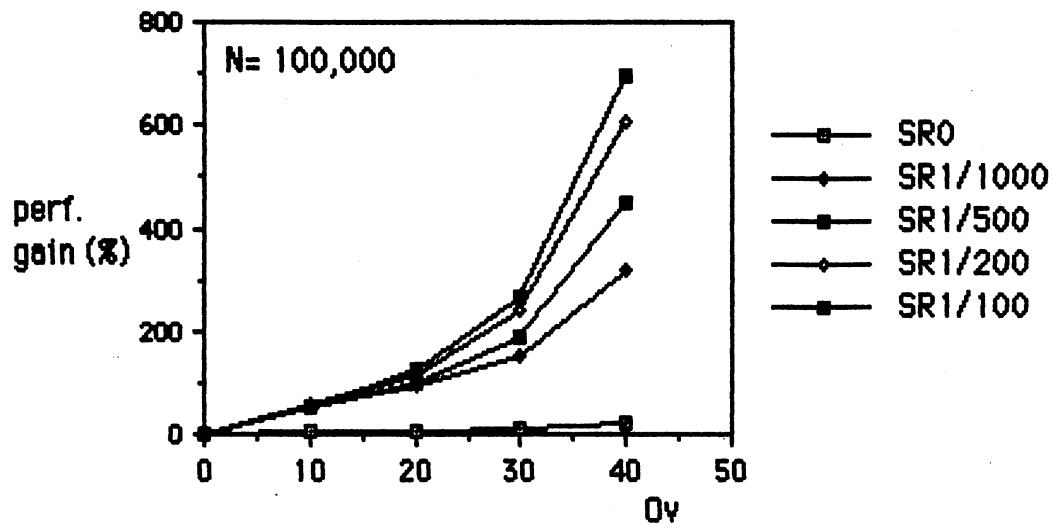


Figure 73. Performance Gain of the MR-tree (In Parallel Processing) over R<sup>+</sup>-tree as a Function of Ov



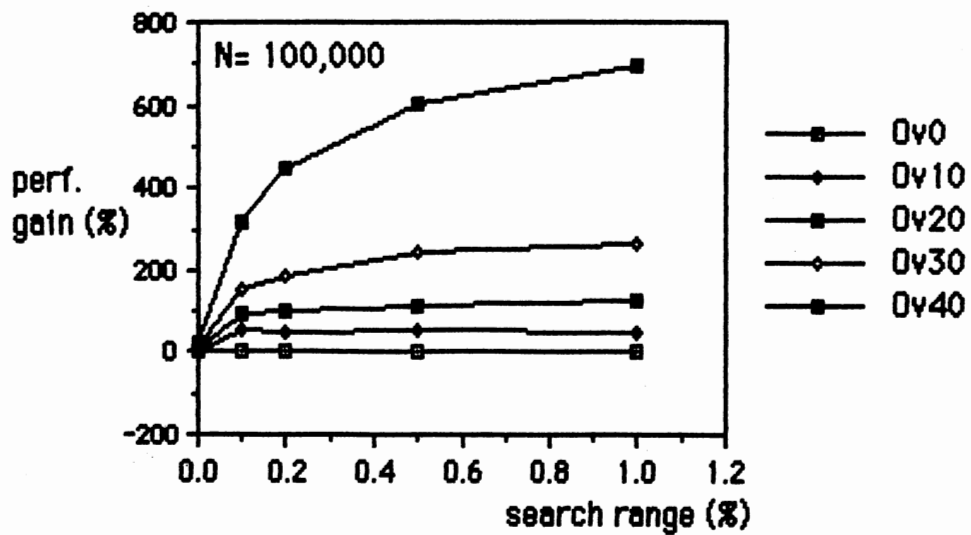


Figure 74. Performance Gain of the MR-tree (in Parallel Processing) over the R<sup>+</sup>-tree as a Function of SR

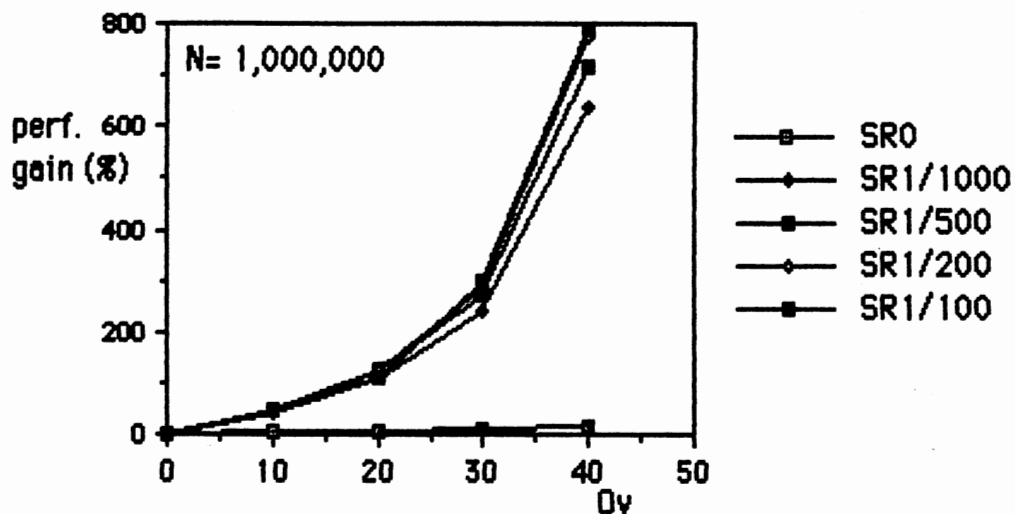


Figure 75. Performance Gain of the MR-tree (in Parallel Processing) over the R<sup>+</sup>-tree as a Function of Ov

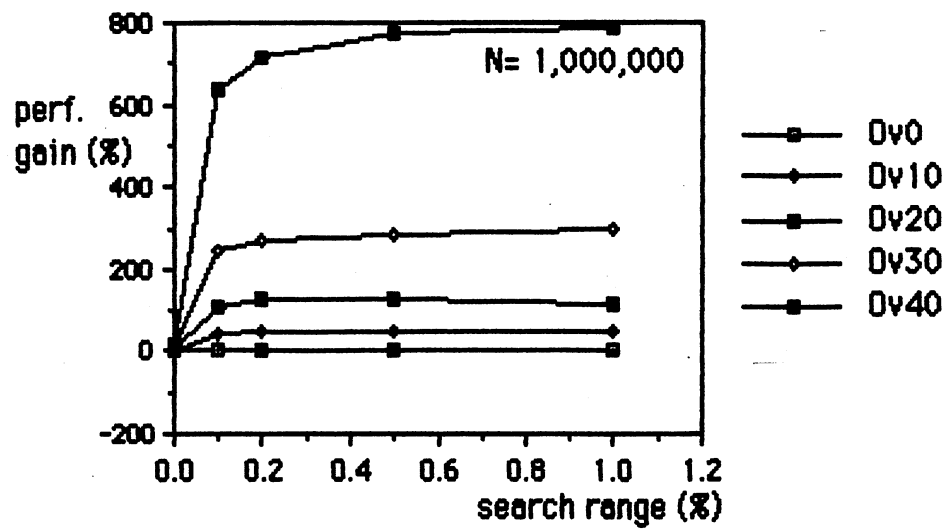


Figure 76. Performance Gain of the MR-tree (In Parallel Processing) over the R<sup>+</sup>-tree as a Function of Search Range

## CHAPTER VI

### SUMMARY, CONCLUSION, AND

### FUTURE WORK

The purpose of the MR-tree is to remove the redundancy in leaf level nodes of the  $R^+$ -tree and to provide better search and deletion performances. The MR-tree still keeps the advantage of the  $R^+$ -tree for disjoint intermediate rectangles. As in performance analysis of the  $R^+$ -tree and the R-tree, uniformly distributed line segments of the one size case and the two size case are used for the MR-tree performance analysis. Since we need 4 coordinates to represent a rectangle, a 4-d space is required to transform a rectangle into a point. However it is difficult to illustrate a 4-d space. The MR-tree method increases the number of subtrees. However, the number of subtrees is very small. When uniformly distributed line segments are used, the number of subtrees was two in the one size line segments case and three in the two size line segments case respectively. Three subtrees were enough to hold  $n$  size line segments (in Chapter V). In typical distribution situation also, small number of subtrees are expected as in performance analysis. We have some results from simulation studies verifying the MR-tree performance analyses.

We implemented the MR-tree in serial and parallel modes by using C language. Our test used two dimensional rectangles, generated by a random number generator. In one test 10,000 rectangles were used and the capacity

of node was 30. To find the density of rectangles, called covering density (Cd), the sum of rectangle areas was divided by the area of data space. The covering density was proportional to the Ov of rectangles. In this test, we considered four cases of Cd, Cd= 4.61, Cd= 5.52, Cd= 6.47, and Cd= 7.44. Data rectangles consisted of many very small sized rectangles and a few large ones. For parallel processing, iPSC/2 concurrent super computer (hypercube) was used. An iPSC/2 system consists of computer nodes, I/O nodes, and a front end processor. A node is a processor/memory pair. In parallel processing, a node processor is assigned to a subtree of the MR-tree. Tables 1 through 4 show the number of disk accesses of the MR-tree for serial and parallel processing (where SR represents search range). In both cases, serial and parallel, the actual numbers of disk accesses are the same but parallel processing has smaller values. Since execution time in parallel processing depends on the maximum number of disk accesses among subtrees, we used maximum number of disk accesses. For each search range, 20 searches were done and the average number of disk accesses were obtained from those. SR 0 means point query.

SR TY	0	1/1000	1/500	1/200	1/100
SER	14.2	17.0	19.5	24.5	30.9
PAR	3.0	4.6	6.0	9.1	12.3

(unit: # of da)

Table 1. The Number of Disk Accesses for Serial and Parallel Processing (Cd= 4.61)

SR TY	0	1/1000	1/500	1/200	1/100
SER	15.1	18.8	21.0	25.0	32.7
PAR	3.8	5.5	6.7	8.3	12.9

(unit: # of da)

Table 2. The Number of Disk Accesses for Serial and Parallel Processing (Cd= 5.52)

SR TY	0	1/1000	1/500	1/200	1/100
SER	15.7	19.5	20.3	24.8	32.0
PAR	3.0	4.8	5.2	7.1	11.1

(unit: # of da)

Table 3. The Number of Disk Accesses for Serial and Parallel Processing (Cd= 6.47)

SR TY	0	1/1000	1/500	1/200	1/100
SER	17.1	21.1	23.3	29.2	36.0
PAR	3.0	4.8	5.7	8.4	11.1

(unit: # of da)

Table 4. The Number of Disk Accesses for Serial and Parallel Processing (Cd= 7.44)

SR TY	0	1/1000	1/500	1/200	1/100
SER	2.501	3.940	3.229	3.957	5.133
PAR	0.469	0.792	0.856	1.174	1.853

(unit: second)

Table 5. Time to Access Disk Pages in Search Area (Cd= 6.47)

Table 5 shows the times required to handle search operations (point and range queries). As we can observe, parallel processing reduces query time drastically. Figures 77 and 78 are graphic representations for Tables 2 and 3.

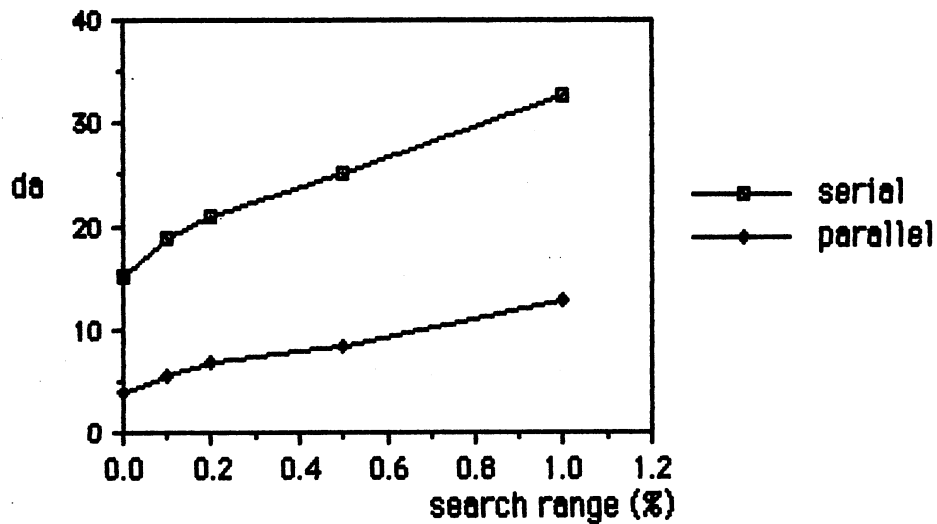


Figure 77. Graph Corresponding to Table 2

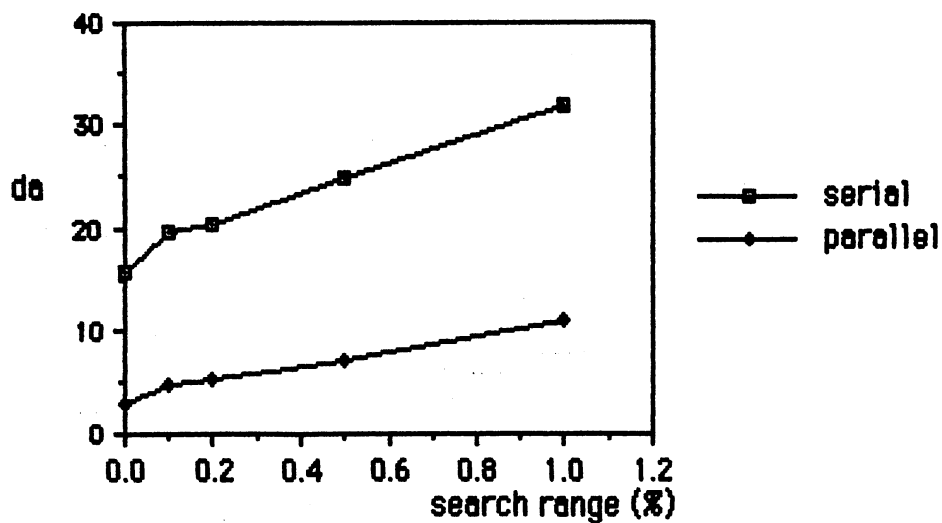


Figure 78. Graph Corresponding to Table 3

Perf. gain of the MR-tree for parallel processing over the MR-tree for serial processing can be obtained by

$$\text{Perf. gain(\%)} = \frac{\text{SMRda} - \text{PMRda}}{\text{PMRda}} \times 100$$

where SMRda represents the number of disk accesses for serial processing and PMRda denotes the number of disk accesses for parallel processing. Figures 79 and 80 show perf. gain of the MR-tree as a function of Cd and search range respectively. In Figure 81, the number of disk accesses of the MR-tree with higher Cd is smaller than the that of lower Cd. This result verifies the performance analysis of the MR-tree for parallel processing. In Figures 67 through 72, the number of disk accesses of the MR-tree in query operations is reduced when Ov is increased. The increment of Cd increases Ov of rectangles in data space.

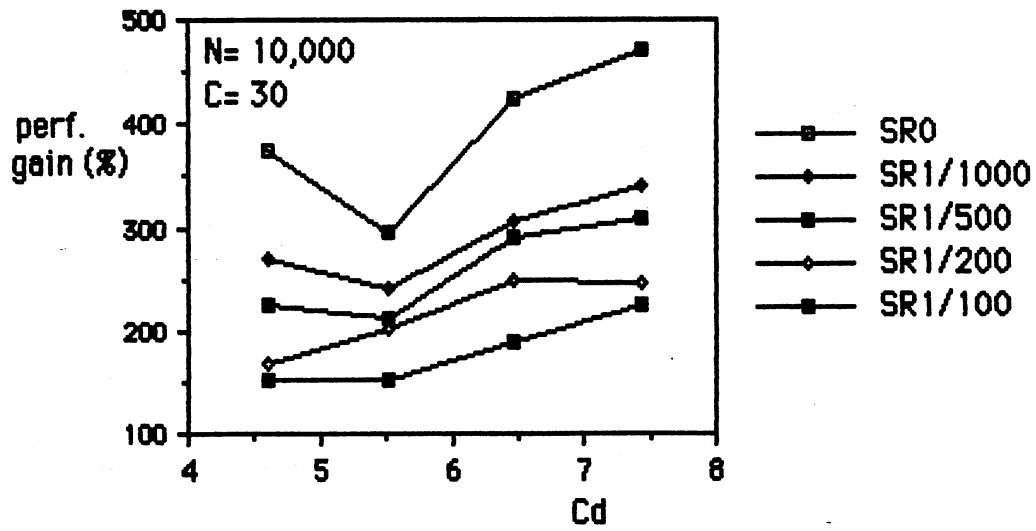


Figure 79. Performance Gain of the MR-tree for Parallel Processing over Serial Processing vs. Cd (Covering Density)

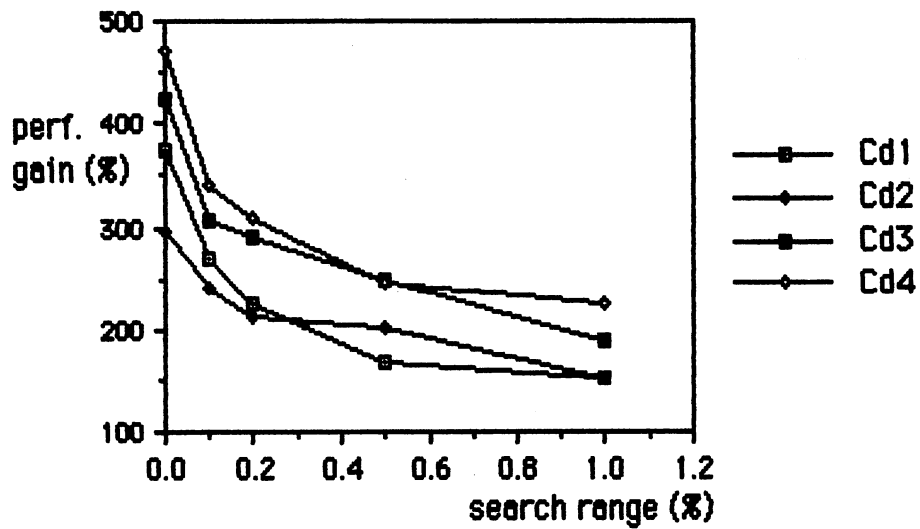


Figure 80. Performance Gain of the MR-tree for Parallel Processing over Serial Processing vs. Search Range



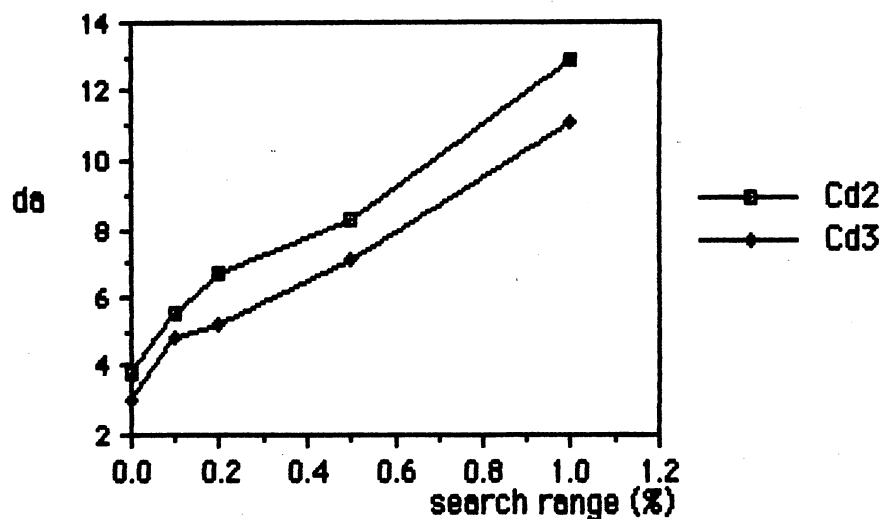


Figure 81. The Number of Disk Accesses for Parallel Processing vs. Cd (Cd2= 5.52 and Cd3= 6.47)

Tables 6 and 7 show the distribution of data rectangles between subtrees. In the MR-tree, the 1st subtree takes more rectangles than the other subtree do. Higher Cd tends to reduce the number of rectangles and the height of S1 to avoid overlapping of intermediate rectangles. As a result, query time in parallel processing is reduced.

subtree	# of rectangles	height
S1	5656	3
S2	2698	2
S3	1142	2
S4	352	1
S5	83	1
S6	40	1
S7	29	0

Table 6. Distribution of Rectangles (Cd= 5.52)

subtree	# of rectangles	height
S1	4944	2
S2	2880	2
S3	1471	2
S4	483	2
S5	123	1
S6	77	1
S7	22	0

Table 7. Distribution of Rectangles (Cd= 6.47)

Table 8 shows performance comparison between data structures (R-tree, R<sup>+</sup>-tree, and MR-tree) in different data types. The MR-tree obtains good performance as the large size data increase (increment of Cd).

data type \ structures	R-tree	R <sup>+</sup> -tree	MR-tree
few large, few small	v		
few large, many small		v	vv
many large, few small	v		vv
many large, many small		v	vv

Table 8. Performance Comparison Between Data Structures in Four Different Data Types (v: Better, vv: Much Better)

The MR-tree has advantages in deletion (by exact coordinates and range) and range query. Using of parallel processors gives the MR-tree

better performance over the  $R^+$ -tree not only in range query but also point query.

Additional future research includes the following tasks:

- 1) Comparison of the MR-tree with the other methods (e.g., R-tree and  $R^+$ -tree) for handling multi-dimensional objects.
- 2) Maximization of parallel processing of the MR-tree.
- 3) Design more efficient partition method for the MR-tree.

## REFERENCES

1. Bentley, J. L. & Finkel, R. K. (1974). Quad-trees: A Data Structure for Retrieval on Composite Keys. ACTA Informatica, 4(1), 1-9.
2. Bentley, J. L. (1975). Multi-dimensional Binary Search Trees Used for Associative Searching. Communications of the ACM, 18(9), 509-517.
3. Bentley, J. L., Stanat, D. F., & Williams, Jr., E. H. (1977). The Complexity of Fixed Radius Near Neighbor Searching. Inf. Proc. Lett. Conference, 6(6), 209-212.
4. Bentley, J. L. & Friedman, J. H. (1979). Data Structures for Range Searching. ACM Computing Surveys, 11(4), 397-409.
5. Robison, J. T. (1981). The K-D-B Tree: A Search Structure for Large Multi-dimensional Dynamic Indexes. Proc. ACM SIGMOD Conference, 10-18.
6. Nievergelt, J. & Hinterberger, H. (1984). The Grid File: An Adaptable, Symmetric Multikey File Structure. ACM Transaction on Database Systems, 9(1), 38 - 71.
7. Chock, M., Cardenas, A. F., & Klinger, A. (1984). Database Structure and Manipulation Capabilities of a Picture Database Management System (PICDMS). IEEE Trans. on Pattern Analysis and Machine Intelligence, PAMI-6(4), 484-492.
8. Guttman, A. (1984). R-trees: A Dynamic Index Structure for Spatial Searching. Proc. of the ACM SIGMOD Conference, 47-57.
9. Ousterhout, J. K., Hamachi, G. T., Mayo, R. N., & Scott, W. S., & Taylor, G. S. (1984). Magic: A VLSI Layout System. Proc. of the 21st Design Automation Conference, 152-159.

10. Roussopoulos, N. & Leifker, D. (1985). Direct Spatial Search on Pictorial Database Using Packed R-tree. Proc. of the ACM SIGMOD Conference, 17-31.
11. Rosenberg J. B. (1985). Geographical Data Structures Compared: A Study of Data Structures Supporting Region Queries. IEEE Trans. on Computer- Aided Design, 4(1), 53-67.
12. T. Sellis, T., Roussopoulos, T., & Faloutsos, C. (1987). R<sup>+</sup>-tree: A Dynamic Index for Multi-dimensional Objects. Proc. of the 13th VLDB Conference, Brighton, 507 - 518. ←
13. T. Sellis, T., Roussopoulos, N., & Faloutsos, C. (1987). Analysis of Object Oriented Spatial Access Methods. Proc. of the ACM SIGMOD Conference, 16(3), 426-439.
14. Freestone, M. (1987). The BANG File: a New Kind of Grid File. Proc. of the ACM SIGMOD International Conference on Management of Data, 260-269. ←  
621.3819505  
S1045, v. 15, no. 2
15. Kasturi, R. , Fernandez, R. , Amlani, M. L. , & Feng, W. C. (1989) Map Data Processing in Geographic Information Systems. IEEE Computer, 22, 10-21.
16. Ohsawa, Y. & Sakauchi, M. (1990). A New Tree Type Data Structure with Homogeneous Node Suitable for a Very Large Spatial Database. Proc. of the IEEE 6th International Conference on Data Engineering, 296-303. ←  
005.74 I645 1990a

VITA 2

Kap San Bang

Candidate for the Degree of

Master of Science

Thesis: MULTI-R TREE: AN EFFICIENT INDEX STRUCTURE FOR MULTI-DIMENSIONAL OBJECTS

Major Field: Computer Science

Biographical:

Personal Data: Born in Seoul, Korea, December 6, 1960, the son of Cheun K. Bang and Chun S. Kang.

Education: Graduated from Sunnam High School, Seoul, Korea, in January 1979; received Bachelor of Science Degree in Chemistry from Chung-Ang University in Seoul, Korea in February, 1987; completed requirements for the Master of Science degree at Oklahoma State University in February, 1992.

Professional Experience: Teaching Assistant, Department of Computer Science, Oklahoma State University, August, 1989, to May, 1992.