NEW EFFICIENT SPATIAL INDEX STRUCTURES, PML-TREE AND SMR-TREE, FOR SPATIAL DATABASES

By

KAP S. BANG

Bachelor of Science
Chung-Ang University
Seoul, Korea
1987

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1992

Submitted to the faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
December, 1995

NEW EFFICIENT SPATIAL INDEX STRUCTURES,

PML-TREE AND SMR-TREE, FOR

SPATIAL DATABASES

Thesis Approved:

Huizhu Lu

Thesis Adviser

J Chandler

J B Conrey

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my major advisor, Dr. Huizhu Lu for her intelligent supervision, constructive guidance, inspiration and friendship. My sincere appreciation extends to my committee members Dr. John P. Chandler, Dr. K. M. George and Dr. Brian J. Conrey, whose guidance, assistance, encouragement, and friendship are also invaluable.

I would also like to give my special appreciation to my wife, Hye-sun, for her strong encouragement at times of difficulty, love and understanding throughout this whole process. Thanks also go to my parents and parents-in-law for their support and encouragement.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

Background of the Spatial Index Structures 2
The Problems of the Existing Spatial Index Structures 3
Research Objectives 6

II. REVIEW OF THE LITERATURE 7

Point Index Structures 7
Spatial Index Structures 13
Parameter Space Indexing 13
Native Space Indexing 14
Non-disjoint decomposition 15
Disjoint decomposition 23
Projection Method 32
Applications 34

III. THE PML-TREE: A NEW PARALLEL SPATIAL INDEX STRUCTURE 35

Introduction 35
The Structure and Algorithms of the PML-tree 37
The PML-tree Structure 37
Algorithms of the PML-tree 43
Insertion of the PML-tree 46
Minimum Number of Entries 46
Absolute Crowd Index 48
Relative Crowd Index 53

Chapter Page

Node Split of the PML-tree 60
Search of the PML-tree 68
Deletion of the PML-tree 69

IV. PERFORMANCE ANALYSES
OF PARALLEL SPATIAL INDEX STRUCTURES 72

Implementations of the Spatial Index Structures 73
Test Data Sets and Types of Queries 75
Experiment Results 79

V. THE SMR-TREE: A NEW EFFICIENT
SERIAL SPATIAL INDEX STRUCTURE 92

Introduction 92
The Structure and Algorithms of the SMR-tree 93
The SMR-tree Structure 93
Algorithms of the SMR-tree 95
Insertion of the SMR-tree 97
Node Split of the SMR-tree 100
Search and Deletion of the SMR-tree 100
Performance Comparisons 102
Implementations of the Spatial Index Structures 103
Experiment Results 104
Summary 113

VI. CONCLUSIONS 115

REFERENCES 116

APPENDIX-- TIGER/Line™ files 124

LIST OF TABLES

Table | Page

1. Contents of Structure CI for Two Trees in Algorithm SelectTree_by_AC 53
2. Contents of Structure CI for Two Trees in Algorithm SelectTree_by_RC 58
3. Coordinates of the Ten Data Objects Used in Figure 25 65
4. List of Parameters Used for the PML-tree Split Algorithms for each of (a) Horizontal Axis and(b) Vertical Axis 67
5. Data Description Factors of the Four Data Sets 76
6. The Number of Re-inserted Entries for the Four Test Data Sets 90
7. Distributions of the Data Objects among the Trees in the SMR-tree 112

LIST OF FIGURES

Figure | Page

1. (a) Organization of Point Objects Using a k-d tree and (b) k-d tree Structure Corresponding to (a) 8

2. (a) Organization of Point Objects Using a BD-tree and (b) BD-tree Structure Corresponding to (a) 9

3. Space Mapping Using a Z-ordering: (a) 4 Sub-spaces and (b) 16 Sub-spaces 11

4. Space Mapping Using a Gray-code: (a) 4 Sub-spaces and (b) 16 Sub-spaces 11

5. Space Mapping Using a Hilbert-curve: (a) 4 Sub-spaces and (b) 16 Sub-spaces 11

6. (a) Organization of Point Objects Using G-tree and (b) G-tree Structure Corresponding to (a) 12

7. (a) Parameter Space and (b) Data Space 14

8. (a) Organizations of Bounding Rectangles (the Solid Lines Construct Object Rectangles; the Dash Lines Construct Intermediate Rectangles) and (b) the R-tree Structure 16

9. (a) Organizations of Bounding Rectangles (the Solid Lines Construct Object Rectangles; the Dash Lines Construct Intermediate Rectangles) and (b) the R*-tree Structure 18

Figure Page

10. (a) Space Decomposition Using the skd-tree with Bucket Capacity 3 and (b) the skd-tree Structure 19

11. The MXR-tree Structure with Three Disks 22

12. (a) Organizations of Bounding Rectangles (the Solid Lines Construct Object Rectangles; the Dash Lines Construct Intermediate Rectangles) and (b) the R^+-tree Structure 24

13. The R^+-tree (a) with MBR and (b) without MBR 26

14. The Grid file with CAP= 3 28

15. R-file with CAP= 3.(a) Data Space Cell, (b) Two Half Cells and (c) One Quarter Sub-cell 30

16. (a) Image, (b) Binary Image Array and (c) The Morton Code Addressing Scheme for Labeling Pixels 31

17. (a) Projections of Objects on Axes, (b) Array Structure for Horizontal Projections before Insertion of Object 4 and (c) Array Structure for Horizontal Projections after Insertion of Object 4 33

18. The PML-tree Layout with Three-layer Four-tree 40

19. Organizations of Bounding Rectangles (the Solid Lines Construct Object Rectangles; the Dash Lines Construct Intermediate Rectangles) on the (a) First Data Space, (b) Second Data Space and (c) Third Data Space 41

20. The PML-tree Structures for the (a) 1st Data Space in Figure 19 (a), (b) 2nd Data Space in Figure 36(b) and (c) 3rd Data Space in Figure 19(c) 42

Figure | Page

21. Parallel Disk Access Time of: (a) the MXR-tree in Figure 8(a); and (b) the PML-tree in Figure 19 43

22. An Example for Absolute Crowd Index and Relative Crowd Index Object Distribution Heuristics (a) the First Data Space and (b) Nodes on Insertion Path of the First Tree 52

23. An Example for Absolute Crowd Index and Relative Crowd Index Object Distribution Heuristics (a) the Second Data Space and (b) Nodes on Insertion Path of the Second Tree 52

24. Illustrate Intermediate Rectangles after Insertion of Data Object 7: (a) the First Data Space and (b) the Second Data Space 54

25. Split of an Overflowing Node 65

26. Organizations of Bounding Rectangles (the Solid Lines Construct Object Rectangles; the Dash Lines Construct Intermediate Rectangles) Using the PML-tree with Node Capacity 9 68

27. Average Numbers of Disk Accesses of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the Size of Query Using Data Set V 80

28. Average Numbers of Disk Accesses of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the Size of Query Using Data Set T 80

29. Average Numbers of Disk Accesses of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the Size of Query Using Data Set R 81

30. Average Numbers of Disk Accesses of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the Size of Query Using Data Set U 81

31. Average Response Time of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the Size of Query Using Data Set V 83

Figure Page

32. Average Response Time of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the Size of Query Using Data Set T 83

33. Average Response Time of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the Size of Query Using Data Set R 84

34. Average Response Time of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the Size of Query Using Data Set U 84

35. Performance Gain of the PME-tree, PAC-tree and PRC-tree over the MXR-tree vs. the Size of Query Using Data Set V 85

36. Performance Gain of the PME-tree, PAC-tree and PRC-tree over the MXR-tree vs. the Size of Query Using Data Set T 85

37. Performance Gain of the PME-tree, PAC-tree and PRC-tree over the MXR-tree vs. the Size of Query Using Data Set R 86

38. Performance Gain of the PME-tree, PAC-tree and PRC-tree over the MXR-tree vs. the Size of Query Using Data Set U 86

39. An Example of the MXR-tree Node Split for Uniformly Distributed Data: (a) before Split and (b) after Split 88

40. Actual Memory Sizes of the MXR-tree, PME-tree, PAC-tree and PRC-tree for Each of the four Data Sets V, T, R and U 89

41. Space Utilization of the MXR-tree, PME-tree, PAC-tree and PRC-tree for Each of the four Data Sets V, T, R and U 89

42. Construction Time of the MXR-tree, PME-tree, PAC-tree and PRC-tree for Each of the Four Data Sets V, T, R and U 90

Figure Page

43. (a),(b) Organizations of Bounding Rectangles (the Solid Lines Construct Object Rectangles; the Dash Lines Construct Intermediate Rectangles) and (c),(d)the SMR-tree structure 94

44. Average Number of Disk Accesses of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the Size of Query Using Data Set V 105

45. Average Number of Disk Accesses of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the Size of Query Using Data Set T 105

46. Average Number of Disk Accesses of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the Size of Query Using Data Set R 106

47. Average Number of Disk Accesses of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the Size of Query Using Data Set U 106

48. Average Response Time of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the Size of Query Using Data Set V 107

49. Average Response Time of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the Size of Query Using Data Set T 107

50. Average Response Time of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the Size of Query Using Data Set R 108

51. Average Response Time of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the Size of Query Using Data Set U 108

52. Performance Gain of the SMR-tree over the R-tree, R^+-tree and R^*-tree vs. the Size of Query Using Data Set V 110

53. Performance Gain of the SMR-tree over the R-tree, R^+-tree and R^*-tree vs. the Size of Query Using Data Set T 110

Figure Page

54. Performance Gain of the SMR-tree over the R-tree, R^+-tree and R^*-tree vs. the Size of Query Using Data Set R 111

55. Performance Gain of the SMR-tree over the R-tree, R^+-tree and R^*-tree vs. the Size of Query Using Data Set U 111

56. Actual Memory Sizes of the R-tree, R^+-tree, R^*-tree and SMR-tree for Each of the four Data Sets V, T, R and U 112

57. Space Utilization of the R-tree, R^+-tree, R^*-tree and SMR-tree for Each of the Four Data Sets V, T, R and U 113

58. Construction Times of the R-tree, R^+-tree, R^*-tree and SMR-tree for Each of the Four Data Sets V, T, R and U 113

CHAPTER I

INTRODUCTION

In modern database systems, geometric data (e.g., points, lines, polygons, polyhedra, or splines) play a very important role. Spatial data is frequently used in many non-standard applications, such as in geographic information systems (GIS), CAD, and VLSI layout; it is also used for computer vision, robotics and image databases. Additionally, spatial data can be used in a conventional database, where a record with k attributes can be represented as a point in a k-dimensional space. Efficient handling of spatial data is one of the most important requirements for future database management systems (DBMS). To use an existing DBMS for spatial retrieval, the DBMS has to be supplemented by new indexing structures and evaluation systems in order to process spatial queries [71].

During the last decade, several spatial index structures have been proposed, but none of them has been proven to outperform all others in every aspects of performance [100]. Most proposed spatial index structures perform query operations in a serial manner. Parallelization of the spatial index structures using multiple disks can increase the performance of the spatial index structures significantly, since the operations in most spatial index structure applications are highly I/O bound. Currently, parallelization of the spatial index structure is a comparatively unexplored topic. In this research, the author proposes and implements an efficient new parallel spatial index structure. In the following, the background of the spatial index structures, the problems of the existing spatial index structures and the research objectives of this study are discussed.

Background of the Spatial Index Structures

A spatial object may be a point or an object which has area or volume and can overlap with other spatial objects. In many cases, very large sets of spatial data are stored in secondary storage (e.g., disks). Therefore, the number of disk accesses, which is heavily determined by the spatial indexing technique, becomes a critical factor in fast retrieval of spatial data. Spatial index structures should be dynamic, i.e., insertion and deletion can be performed without completely re-organizing the index structure. Also, the index structures should be storage space efficient given any distribution of data. Currently, there are two major spatial indexing methods used for spatial data:

1. parameter space indexing that transforms spatial objects into points in higher dimensional space and
2. native space indexing that organizes data based on the locations of the data.

Parameter space indexing does not preserve spatial locality. That is, two closely located objects in the original data space may be mapped into parameter space arbitrarily far apart. Therefore, this indexing method is not proper for applications which require certain range queries (e.g., finding the nearest neighbors of a given point or object). The parameter space indexing method is fine for storage applications. Native space indexing method decomposes data space, in which the data objects are drawn, into sub-spaces. This method keeps spatial locality and enables spatial index structures to answer to various types of range queries.

Hashing [55, 59, 79, 80, 81, 84] has the best performance for exact match queries. However, hashing approaches are not proper to support range queries, since hashing structures cluster data by hashed key value, not by key value. Several hashing structures (e.g., order preserving hashing [44]) support range queries as well as exact match queries. These structures, however, cannot fully support range

queries, since hashing methods require approximately uniform key distributions. Tree structures, which locally grow as data is inserted and new index term(s) is posted for the growing space by insertion into parent node(s), are suitable for spatial data handling. The GP (grow and post) trees are always balanced; besides, entries inserted into the GP-trees that are close in data space are close in the tree location [62]. This property of the tree structure satisfies the spatial locality requirement of spatial index structures and provides a natural, high level, object oriented search [86].

In most spatial index structure applications (e.g., GIS, CAD and image processing), data objects do not conform to any fixed shape. Therefore, it is very expensive to perform spatial queries on exact locations and extents of the data objects. There are two methods, region decomposition and minimum bounding rectangle, for initial approximation (filtering) of irregularly shaped spatial objects. The region decomposition method decomposes a sub-space occupied by an object into disjoint raster squares of desired resolution. The minimum bounding rectangle method uses the smallest rectangle enclosing an irregularly shaped spatial object. The region decomposition method (e.g., quadtree) is suitable for image processing and the minimum bounding rectangle method is used in proximity query processing [72]. Native space indexing methods using the GP-tree structure, which are related to the design of new parallel spatial index structure in this research, are briefly discussed in the following; detailed classification of the spatial index structures are discussed in Chapter II.

The Problems of the Existing Spatial Index Structures

One of the most popular approaches to native space indexing methods is the R-tree and its variants. The R-tree is the extension of the B-tree in multi-dimensional space [40]. It is a height balanced tree. The R-tree and its variants use a minimum

bounding rectangle (MBR) to represent a spatial data object in the space. Although this approximation loses some information, this method requires small amounts of space to store an object and still reserves the most important information about the objects (e.g., location and extension of the object). The R-tree variants use two node types: intermediate nodes and leaf nodes. Leaf node entries consist of two parts: a tuple identifier used to refer to a tuple in a database; coordinates that define a n-dimensional rectangle for enclosing the spatial object. Each entry in an intermediate node consists of a child pointer to the node at a lower level, and coordinates representing a rectangle that completely encloses all rectangles at the lower levels. The R-tree allows overlapping intermediate rectangles and this increases the chance of redundant searching in query operations. If k overlaps exist in an area and a search range includes that area, it is necessary to search all k paths to find objects overlapping the search area. As a result, the number of disk accesses for the query are increased. Also, the split algorithms of the R-tree are not efficient with uniformly distributed data. In Chapter III, a drawback of the R-tree's split algorithms is discussed in detail with examples.

The R^+-tree is a variant of the R-tree and it uses a disjoint space decomposition method [95]. That is, the R^+-tree does not allow overlapping among the intermediate rectangles. Upon splitting, an intermediate rectangle is partitioned into two intermediate rectangles and all rectangles on the split line are divided. If a leaf node rectangle is on the split line, it is divided into two leaf nodes with the same object name in both intermediate rectangles. It causes redundancies at the leaf node level and as a result, the total number of nodes in the tree is increased. Redundancies in the R^+-tree provide faster access in point queries and in very small size range queries. However, note that redundancies at the leaf node level cause disadvantages in range queries and deletions as the search range grows. Also, the R^+-tree deletion algorithm provided in [95] cannot delete all objects overlapping a given deletion range.

It needs to search all objects overlapping a deletion range and then deletes one object at a time using obtained coordinates of objects from the range search.

The MXR-tree is the first parallel spatial index structure [51]. It is a variation of the R-tree. The MXR-tree distributes R-tree nodes over multiple disks. The MXR-tree suggests 4 different heuristics (round robin, minimum area, minimum intersection and proximity index) to distribute nodes over the disks. Among them, proximity index has the best performance. The purpose of the proximity index is to place a newly created node on a disk where the node has the least chance of retrieval together with the nodes already on the disk. To make an ideal node distribution, all sibling nodes on the same level have to be considered. However, this requires too many disk accesses. The proximity index heuristic of the MXR-tree only considers sibling nodes of the same parent node to calculate proximity. Therefore, the proximity heuristic can result in unbalanced node distributions in some cases. The MXR-tree has a redundant search path problem along with the R-tree, since the MXR-tree inherits structural properties (e.g., split algorithms) of the R-tree. In query operations on the parallel spatial index structures, D (the number of disk used) processes are created and each of processes searches the associated disk simultaneously. The MXR-tree has a single tree structure and all the nodes are distributed into D disks. Therefore, the MXR-tree needs inter-process communications during a query operation whenever each of the D processes finds an intermediate entry overlapping the given search or deletion range. Inter-process communication is another factor that slows down query performance of the MXR-tree. The time for inter-process communication is increased as the number of disks used for the MXR-tree increases. Detailed description of this method is discussed in Chapters III and IV with examples.

Research Objectives

The goal of this research is to design and implement a new dynamic parallel spatial index structure called a parallel multi-layer (PML) tree using native space indexing with a disjoint space decomposition method. The PML-tree avoids the redundant search path problem of the R-tree and the leaf node redundancy problem of the R^+-tree. Also, the PML-tree does not need inter-process communication in query operations. Three objects distribution heuristics, which distribute data objects over the multiple disks evenly, are proposed and implemented. Compared with the MXR-tree, the PML-tree increases space (node) utilization and improves query performances on a system with multiple disks. The author also proposes a serial spatial index structure called serial multi-R (SMR) tree using native space indexing with a disjoint space decomposition method. The SMR-tree improves the performances of currently existing serial spatial index structures (e.g., the R-tree, R^+-tree, R^*-tree, ..., etc.).

CHAPTER II

REVIEW OF THE LITERATURE

Since conventional database management systems (DBMS) were developed to handle one-dimensional data objects, such as integers, real numbers, or strings, they are not efficient for handling multi-dimensional data objects, e.g., boxes or polygons [18, 91, 92]. Large spatial data handling is required in databases for the geographic information systems (GIS), computer vision, CAD, VLSI and the image databases. Therefore, it is important to provide an efficient access method to improve the performance of spatial databases.

During the last decade, several spatial index structures have been proposed. However, there has been no single dominating spatial index structure for general purpose use. The performance of spatial index structures varies depending on data types and applications. In one-dimensional index structures, the B-tree and its variants are accepted as efficient general purpose structures. In a spatial (n-dimensional, $n \geq 2$) index structure, there are many parameters which affect performance of the spatial index structure and interact in a very complicated way with each other. A brief overview of the work done in this area is given based on the following classification: point index structures, spatial index structures and applications.

Point Index Structures

Multi-dimensional point index structures usually handle records with n attributes which can be represented as points in n-dimensional data space.

The k-d tree [10, 11, 12] is a *k*-dimensional binary tree. It partitions a n-dimensional data space into hyper-rectangles using a suitable discriminator, i.e., median value of one axis (attribute) selected by certain criteria. For example, Figure 1(a) shows space decomposition of the k-d tree with bucket capacity 3. Upon insertion of object 4, the data space is divided by vertical line *d1*. To resolve next level bucket overflowing, hyper planes *d2* and *d3* are used to sub-divide sub-spaces *A* and *B*, respectively. Figure 1(b) illustrates the k-d tree structure corresponding to Figure 1(a). It is a well-defined and efficient index structure for the storage and manipulation of multi-dimensional point objects. However, the k-d tree is a main memory resident structure, since this structure does not account for paging of secondary memory.

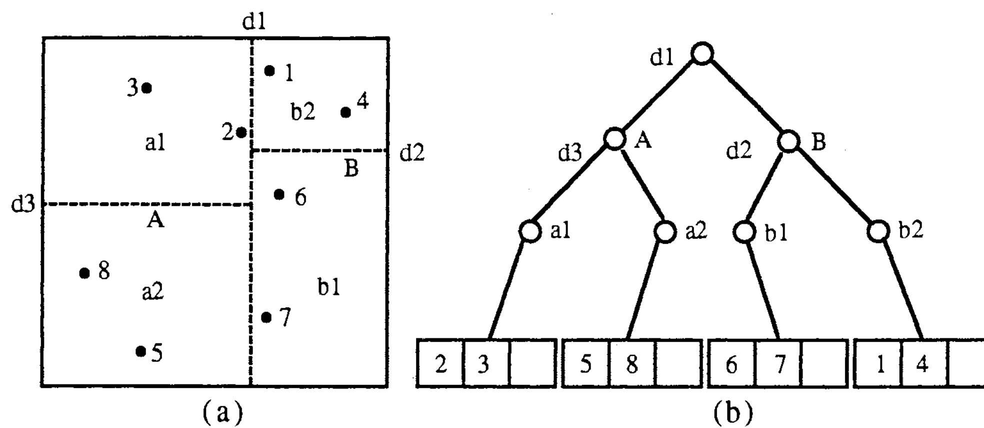


Figure 1 (a) organization of point objects using a k-d tree and (b) k-d tree structure corresponding to (a).

The Quintary-tree [56] consists of multi-level binary trees. For each level (attribute) the Quintary-tree has a binary tree as a skeleton and attached to each node of the skeleton tree are three additional sub-trees (two sub-trees for the two sub-files in the lower level separated by the hyper plane associated with the node and one sub-tree for the record on the hyper plane). This structure is also a main memory

resident structure and node redundancy is typical when all key values of the attributes are unique.

The BD-tree [69] uses a binary decomposition method to partition a data space into sub-spaces. The BD-tree is uses binary tree structure to organize sub-spaces. Intermediate nodes of the BD-tree consist of a partition number, a discriminator zone (DZ) expression , and two branches of the child nodes (in and out branches) depending upon whether a point belongs to the DZ expression or not. A DZ expression is a binary string representing the location of a sub-space. Figure 2(a) shows an example of the space decomposition method of the BD-tree. The capacity of a data bucket is 3. The DZ expression at each node represents one of two sub-spaces with greater cardinality. For example, the second level DZ expression in Figure 2(b) is selected as 110* which represents sub-space *a*, since sub-space *a* has more entries than sub-space *b2*. The BD-tree can typically be very deep, since this structure is not a balanced tree. Also, the BD-tree is a main memory resident structure.

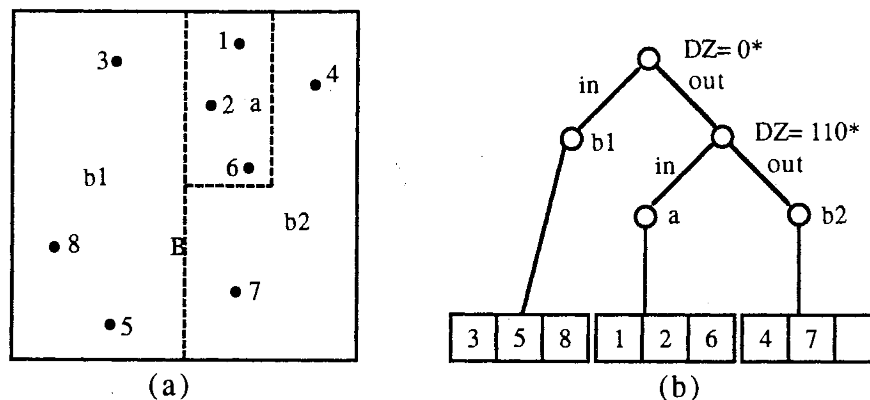


Figure 2 (a) organization of point objects using a BD-tree and (b) BD-tree structure corresponding to (a).

The k-d-b [85] tree is the first index structure that has been designed specially for paged secondary memory. The k-d-b tree is a generalization of the B-tree to higher dimensions for indexing points in arbitrary dimensions.

Another point indexing approach uses mapping functions e.g., Z-ordering [75, 76, 77, 78], Gray-code [23] and Hilbert-curve [48,49]. Mapping functions map multi-dimensional points into a single-dimensional space. For example, Z-ordering maps a multi-dimensional point object into a single-dimensional space with a single bit string. The single bit string can be obtained by interleaving the bits that represent the attribute values of the object on the space. Any existing one-dimensional index structures (e.g., B^+trees) can be used to organize these mapped bit strings. Z-ordering can be recursively defined. For example, in Figure 3(a) (page 11), a given region can be divided into quadrants and a Z curve can be drawn. Then each quadrant is sub-divided in turn into four as in Figure 3(b). Figures 4 and 5 (page 11) illustrate Gray-code and Hilbert-curve methods, respectively. However, these mapping functions cannot keep spatial locality, close spatially located objects should be closely located in the index structure, too. Two closely located objects in multi-dimensional space can be mapped into one-dimensional space arbitrarily far apart with these mapping functions, since there are only two adjacent neighbors of an object in one-dimensional space.

In geographic information systems, each spatial object can be assigned to a class (e.g., water, land use) such that multi-class range queries, which are range queries to an arbitrary sub-set of classes of objects, can be answered. The multi-class grid file [68] is proposed to handle multi-class range queries to multi-dimensional point objects. This structure is a variant of a multi-level grid file. Each class has its own data space partitioning and its own data blocks, but entries referring to directory blocks are stored in a common directory tree.

The generalized grid file (GGF) [13] is proposed to offer multi-attribute access. It behaves like B-tree if single attributes are supported and like a grid file for multiple attributes. The directory of the GGF is a tree of arbitrary depth with pages as nodes.

Each page consists of n scales and an n-dimensional pointer array to handle data with n-attributes.

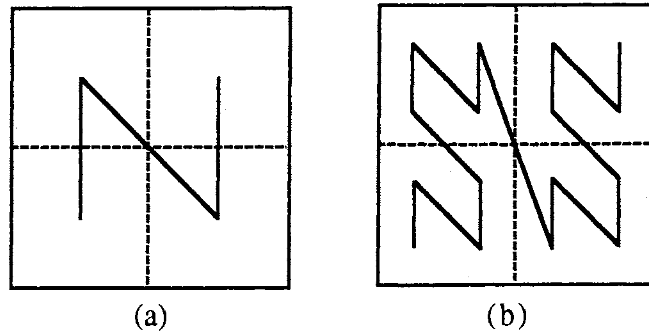


Figure 3 Space mapping using Z-ordering: (a) 4 sub-spaces and (b) 16 sub-spaces.

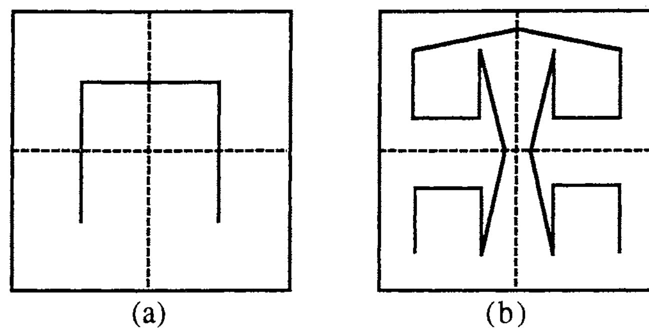


Figure 4 space mapping using Gray-code: (a) 4 sub-spaces and (b) 16 sub-spaces.

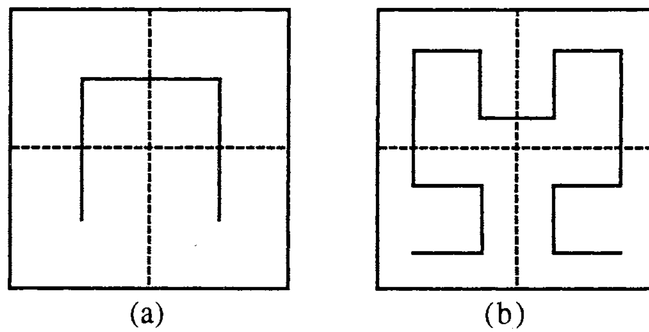


Figure 5 space mapping using a Hilbert-curve: (a) 4 sub-spaces and (b) 16 sub-spaces.

The G-tree [54] is a mixed structure of the grid file and the B-tree. This structure is very similar to the BD-tree [69]. Both structures use the same space decomposition method, binary decomposition. The G-tree directly maps a sub-space into a page (bucket) while in the BD-tree, buckets can exclude some sub-spaces from a given space. Figure 6(a) illustrates an example of the G-tree's space decomposition. Data space is decomposed by using a binary decomposition scheme and each sub-space can be represented using a DZ expression (e.g., sub-spaces *a*, *b*, *c* and *d* can be represented as 0*, 10*, 110* and 111*, respectively). Those sub-spaces are organized in the B-tree by associated DZ expressions as shown in Figure 6(b).

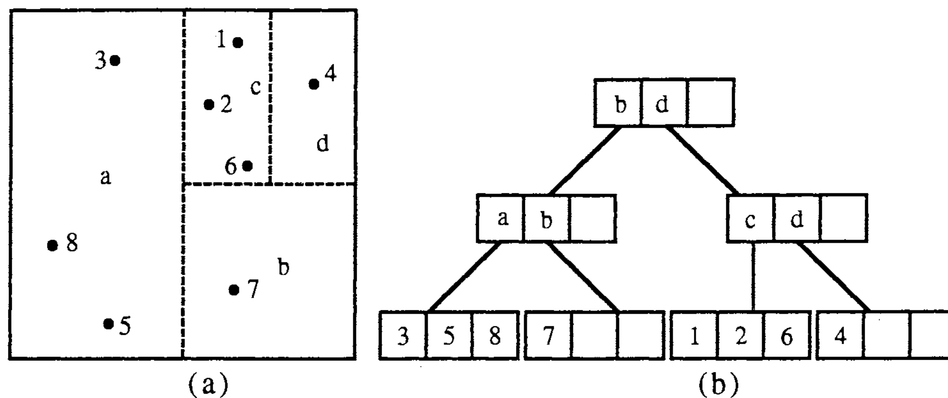


Figure 6 (a) organization of point objects using a G-tree and (b) G-tree structure corresponding to (a).

The parallel grid file [58] proposes an algorithm for initial loading of large existing data files into processing nodes. This method loads an initial file *F* into *N* processing nodes by using logical decomposition, then creates a local grid file in each node. This method assumes that the initial data file already exists in some format. Frequent update operations will degenerate the balance between processor nodes. Therefore, this bulk loading method is not for dynamic spatial indexing structures.

The BANG file [25] has a tree structured directory that has a self-balancing property. Its directory always expands at the same rate as the data regardless of the data distribution. The BANG file uses a set of hash functions which map the coordinates of a point in the data space to the number of the grid region in which it lies.

Spatial Index Structures

Spatial data is non-point data and it has extent on each dimension and can overlap with other spatial data. Many index structures have been proposed to handle point data; however, very little has been done with structures handling non-point data, i.e., lines, boxes and polygons [57]. For the multi-dimensional spatial objects, there are two different indexing approaches, parameter space indexing and native space indexing [43, 64].

Parameter Space Indexing

Parameter space indexing transforms a spatial object into a point in higher dimensional space, say parameter space. The number of dimensions of the parameter space is twice that of the native space. For example, a rectangle in 2-d can be transformed into a point in 4-d since a rectangle in 2-d can be represented by four coordinates. Once objects are transformed into points in parameter space then those points can be assigned into exactly one of the regions of the partitioned parameter space and any multi-attribute point indexing method discussed above can be used to organize them. One of the advantages of this indexing method is that non-spatial attributes can be indexed with the spatial ones at the same time within a single index structure [64]. Transforming objects to parameter space is not always appropriate for spatial objects since this methods does not preserve spatial locality. For example,

one-dimensional line segments *A1*, *A2* and *A3* in Figure 7(b) are transformed into points in two-dimensional parameter space in Figure 7(a) using the starting and ending points of the line segments. In Figure 7(a), the nearest neighbor of the point *A1* is the point *A3*. However, in Figure 7(b), the line segment *A2* is closer to the line segment *A1* in the data space than *A3* is. Besides the spatial locality problem, dimensionality of parameter space is too high, twice as high as that of original data space [43]. This indexing method is not proper for applications which require various range queries (e.g., finding the nearest neighbors of a given point or object). Parameter space indexing method is fine for storage applications.

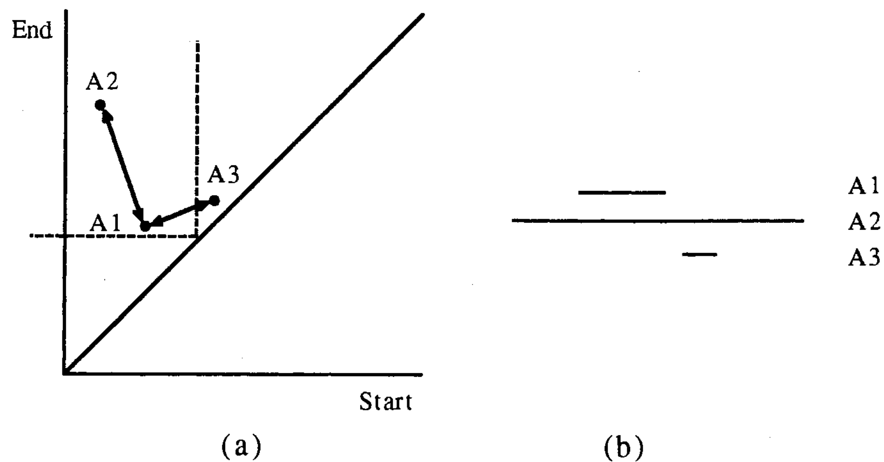


Figure 7 (a) parameter space and (b) data space.

Native Space Indexing

Native space indexing preserves spatial locality by decomposing the original space, in which the data objects are drawn. This method organizes data based on location of the data object. Spatial objects are grouped into sub-spaces and sub-spaces are grouped into upper level sub-spaces recursively. This method enables spatial index structures to answer to various types of range queries. Native space

indexing is much more complicated than parameter space indexing, since objects have their own extent in the space. There are two approaches to native space indexing: non- disjoint decomposition and disjoint decomposition .

Non-disjoint Decomposition

Structures with this method allow overlapping among intermediate sub-spaces. The R-tree developed by Güttman is an extension of the B-tree to n-dimensions ($n \geq 2$) [40]. 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: intermediate and leaf nodes. Leaf node entries consist of two parts: a tuple identifier used to refer to a tuple in a database; coordinates that define an n-dimensional rectangle for enclosing the spatial object. Each entry in an intermediate node consists of : a child pointer to the node at a lower level; coordinates representing a rectangle that completely encloses all rectangles at the lower levels [40]. The R-tree uses non-disjoint space decomposition; that is, the R-tree structure allows overlapping among the intermediate rectangles. In Figure 8(a), the pairs of the second level intermediate rectangles overlapping each other are (11, 12), (11, 15), (12, 13), (13, 14), and (14, 15). Figure 8(b) shows the R-tree structure corresponding to Figure 8(a). If k overlaps exist in an area and a search range includes that area, it is necessary to search all k paths to find objects overlapping the search area. To alleviate this problem, a packing technique was proposed [86]. The packing algorithm takes as input a set of data objects to be packed and produces as output a near-optimal packed R-tree. For each data object on the input list, the packing algorithm invokes the nearest neighbor search function, which returns the nearest neighbor on the list and deletes it from the list, as many times as the number of node branching factor. This algorithm assumes that the spatial database remains relatively static, and it takes a

lot of computation time. Therefore, this packing algorithm cannot be applied to every insertion [95]. The R-tree employs the quadratic and linear split algorithms. The quadratic split algorithm yields better split performance than the linear split algorithm.

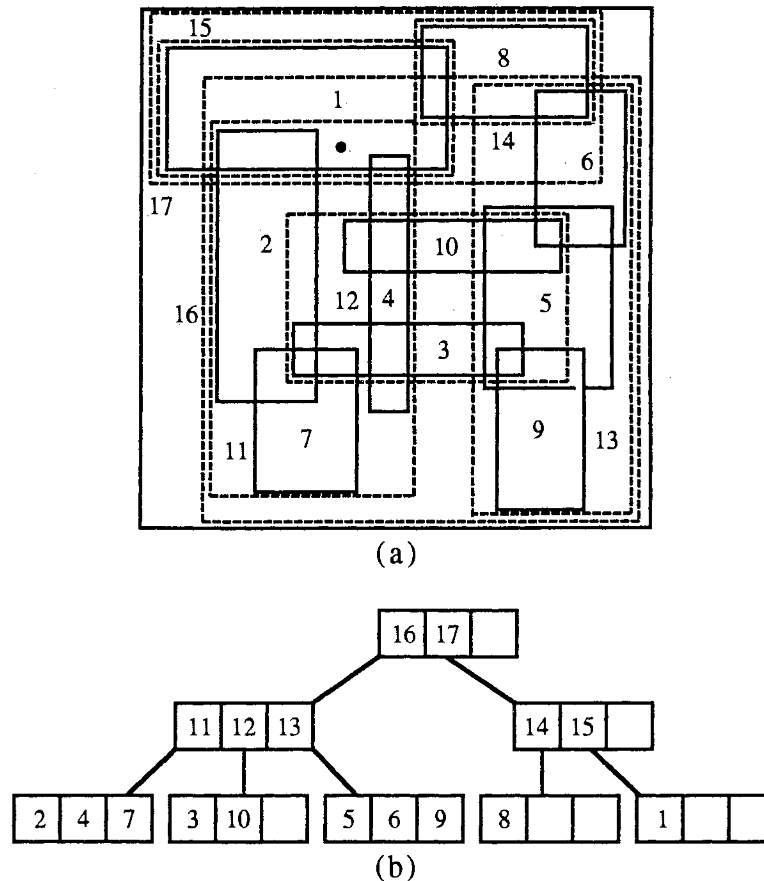
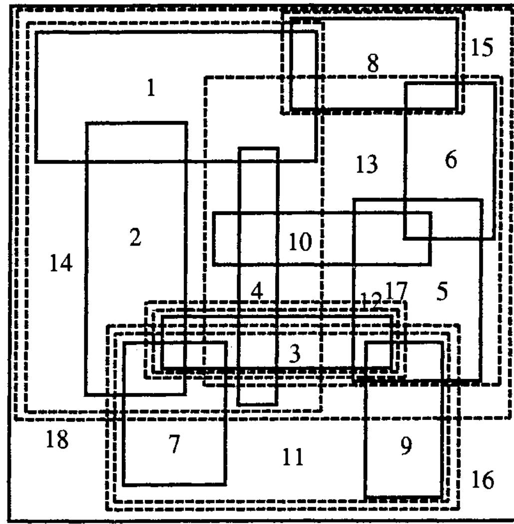


Figure 8 (a) organizations of bounding rectangles (the solid lines construct object rectangles; the dash lines construct intermediate rectangles) and (b) the R-tree structure.

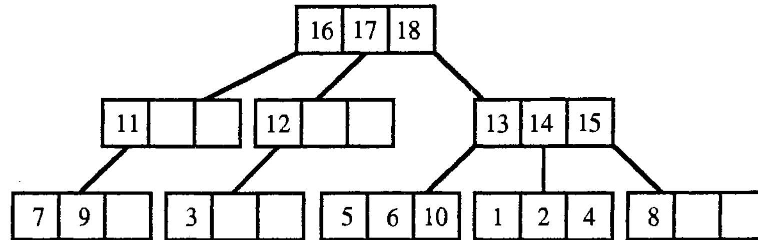
The R*-tree [7] is a more refined variant of the R-tree and also uses non-disjoint space decomposition. In insertion, the R*-tree algorithm selects a node that gives minimal overlap with its neighbors after it is enlarged to include the newly inserted object. It uses a modified split algorithm to reduce area, margin and overlap. In node splitting, for each axis, entries in the overflowing node are sorted using the upper and the lower coordinates of their rectangles. For each sort, the possibilities of dividing

CAP + 1 entries into two groups are CAP - 2m + 2 where CAP denotes node capacity and m denotes the minimum number of entries in a node [7]. Among all possible horizontal and vertical distributions, the axis that gives the minimal margin value is selected, and the distribution that gives the minimum overlap value is chosen. The minimum area value is used to break a tie. This method actually increases overlapping areas among the intermediate rectangles. The splitting algorithm of the R*-tree also uses forced re-insertion which removes a certain percentage, e.g., 30%, of the entries from an overflowing node and re-inserts them. Re-inserted entries tend to be inserted either in nodes near the original node from which they were removed or in the original node. This process makes adjacent intermediate rectangles grow and as a result overlapping areas between them are increased. This structure cannot avoid overlapping sub-regions. The splitting algorithm of the R*-tree requires much more construction time compared to that of the R^+-tree, almost 9 times longer [43]. The R*-tree takes less storage space than the R^+-tree but its performance is not as good as the R^+-tree because of the non-disjoint nature of space decomposition as in the R-tree [43]. Figure 9(a) (page 18) illustrates organization of bounding rectangles using the R*-tree and Figure 9(b) shows the corresponding R*-tree structure.

The spatial k-d (skd) tree [72] is a generalization of the k-d tree for multi-dimensional point data to handle non-zero sized data objects. This structure uses the same space decomposition method as the original k-d tree. At each node of the skd-tree, a discriminator value is chosen in one of the dimensions to partition a k-dimensional space into two sub-spaces. Since non-zero sized data objects may extend over to the other sub-space, the skd-tree introduces a virtual sub-space for each original sub-space such that all objects are completely enclosed in one of two virtual sub-spaces. In this method, the placement of an object in a sub-space is decided by the value of its centroid.



(a)

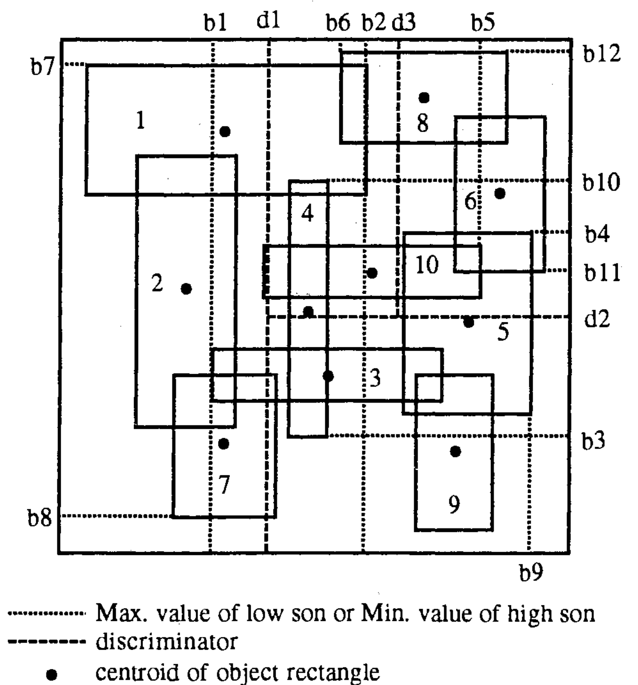


(b)

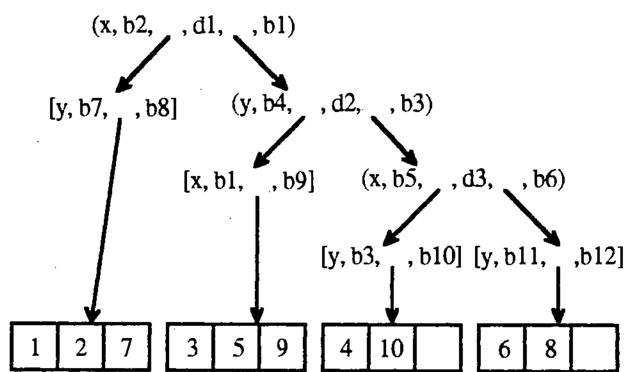
Figure 9 (a) organizations of bounding rectangles (the solid lines construct object rectangles; the dash lines construct intermediate rectangles) and (b) the R*-tree structure.

Figure 10(a) illustrates space decomposition using the skd-tree with bucket capacity 3. Upon insertion of object 4 into a node which already has three entries (i.e., 1, 2, and 3), the node overflows and needs to be split into two. Four objects are divided by a discriminator value *d1* on the horizontal axis . The maximum virtual boundary of entries in the left child node on the horizontal axis is *b2* and the minimum virtual boundary of the entries in the right child node on the horizontal axis is *b1*. An intermediate node consists of 6-tuples, two child pointers, a discriminator, a discriminator value, a maximum virtual boundary of objects in the left child node along the dimension specified by the discriminator and a minimum virtual boundary of objects

in the right child node along the dimension specified by the discriminator. A leaf node consists of 4-tuples, a discriminator, a pointer to a data page, minimum and maximum values of objects in the data page along the dimension specified by the discriminator. In Figure 10(b), node structures of the skd-tree are illustrated.



(a)



(b)

Figure 10 (a) space decomposition using the skd-tree with bucket capacity 3 and (b) the skd-tree structure.

The skd-tree uses virtual boundaries of the sub-spaces to process intersection searches (range queries) and uses tighter boundaries decided by a partition line to process containment searches. This structure can have an arbitrary size of data page. However, due to the nature of binary trees, small number of fanouts, intermediate and leaf nodes of the skd-tree do not account for paging of secondary memory. The skd-tree can be very deep, since this structure is not a balanced tree. Also, for intersection searches, this structure can suffer from overlapping sub-spaces as in the R-tree.

The minimum bounding sphere (MBS) method to approximate spatial data objects is used for the KD2B-tree and the Sphere-tree [74]. The split lines of the KD2B-tree have arbitrary angles and two split lines, Left-line and Right-line, are used to partition a sub-space. Left-line defines the smallest sub-space that completely enclose all the objects of the left sub-set. Right-line is defined analogously. The KD2B-tree has a binary tree structure, since it uses a binary space decomposition method. Therefore, to be adapted to the paging environment, an intermediate node of the KD2B-tree contains a tree structure consisting of a sub-set of the binary tree. This structure allows overlapping intermediate partitions and the modified binary tree structure, for paging systems, requires more complicated update processing. The Sphere-tree is very similar to the R-tree. It uses MBSs instead of minimum bounding rectangles (MBRs) to approximate arbitrarily shaped spatial data objects. One advantage of this structure is that representation of MBS requires less memory space than that of MBR. The MBS needs only one point and a radius, i.e., d+1 floating point numbers to represent an object on d-dimensional data space, and the MBR needs two points for each axis, i.e. 2d floating point numbers. The performances of the MBS and MBR approximation methods depend on the patterns of the spatial data objects. This structure still has all the disadvantages of the R-tree.

The GBD tree [70] is a mixed structure of the R-tree [40] and the grid file [67]. It uses a minimum bounding rectangle and a binary string representing binary space decomposition to manage space decomposition. The minimum bounding rectangle (MBR) is used for query process as in the R-tree and the binary string is used for insertion and deletion operations. Therefore, each entry in the GBD tree consists of three parts, a pointer to a child node, coordinates of the MBR and a binary string (DZ expression). To express the position and the size of the decomposed area, a DZ expression is used. The GBD tree also uses the center point of the object to decide the region in which the object is placed, as in the skd-tree. This structure inherits the problem of the R-tree, overlapping intermediate rectangles. The GBD tree requires more memory space than the R-tree and as a result the node capacity of the GBD tree is smaller than that of the R-tree when the same page size (e.g., 1 Kbyte) is used. If a node in the GBD tree is split, then both minimum bounding rectangles and DZ expressions associated with the entries on the insertion or deletion path have to be adjusted. The GBD tree splitting method only considers the number of overlapping sub-regions to make an even split of entries in an overflowing node. However, this method ignores one of the most important principles which the spatial index structures with a non-disjoint space decomposition method should satisfy to improve performance; that is, the minimization of overlapping areas among the intermediate rectangles to alleviate the number of redundant search paths. The split method of the GBD-tree can make two intermediate rectangles partially or completely overlap each other after a node split. In query operation, the GBD tree also uses minimum bounding rectangles to find overlapping objects. Heavy overlapping among the intermediate rectangles in the GBD tree will result in poor query performance. Assuming that this structure is implemented on secondary storage, insertion or deletion performances of the GBD-tree is not expected to outperform that of the R-tree.

The parallel R-tree (MXR-tree) [51] is an variation of the R-tree. The MXR-tree distributes R-tree nodes over several disks which can be accessed in parallel. The hardware architecture is a single processor with multiple disks. Total response time for a query is proportional to the sum of the maximum disk access time of each level. The MXR-tree suggests four different heuristics to distribute nodes among the disks. The proximity index has the best performance of the four heuristics. Proximity between two rectangles represents the spatial adjacency of two rectangles. All sibling nodes are grouped by the disk number and proximity is calculated between a new node and each of the nodes in each group. The proximity index (PI) selects maximum proximity for each group. The proximity index assigns a new node onto a disk with the lowest proximity index. The purpose of the proximity index is to reduce the chance of retrieval of the new node together with the nodes which are already on that disk. For ideal node distribution, all sibling nodes on the same level have to be considered. However, this requires many extra disk accesses. Therefore, the proximity index heuristic only considers sibling nodes under the same parent node to calculate proximity. In Figure 11, *D1*, *D2* , and *D3* represent disk numbers.

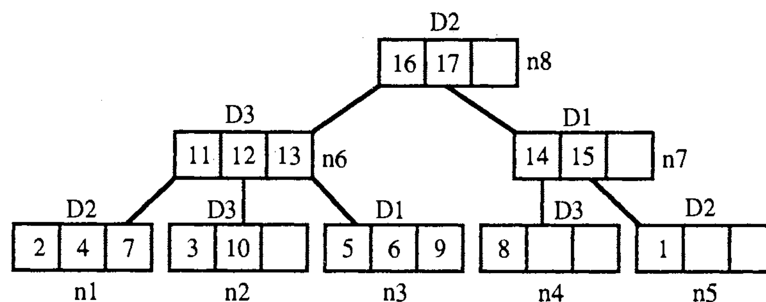


Figure 11 the MXR-tree structure with three disks.

As can be observed, node distribution among the siblings under the same parent node works well. However, the proximity index does not affect distribution among child nodes which have different parent nodes. For example, in Figure 11, nodes *n1* and *n5*

are in the same disk *D2* and corresponding intermediate rectangles for these two nodes are close in data space in Figure 8(a). To speed up parallel disk accessing, these two nodes should be placed on different disks. As mentioned by Kamel [51], 84% speed up of an MXR-tree with 5 disks over an R-tree with 1 disk is considered to be low.

Disjoint Decomposition

Structures with the disjoint decomposition method do not allow overlapping among intermediate sub-spaces at the same level. The hB-tree [60, 62] is derived from the k-d-b-tree [85]. It avoids downward split propagation and saves re-structuring cost and storage utilization. An hB-tree node may have multiple parents, so strictly speaking, this is not a tree structure. The k-d tree [10] is used as an internal structure to remember space decomposition that was performed to produce its child nodes. Node splitting requires that the k-d tree be split. This makes a region represented by the node like the holey brick. This is different from the k-d-b-tree in which the node is always represented as a rectangle.

The R^+-tree [95] is a variant of the R-tree. The major difference between the R-tree and the R^+-tree is that the R^+-tree does not allow overlapping among the intermediate rectangles. Upon node splitting, an intermediate rectangle is partitioned into two intermediate rectangles and all rectangles on the partition line are divided. If a leaf node rectangle is on the partition line, it is stored in both intermediate rectangles with exactly the same coordinates. Figure 12(a) shows a grouping of rectangles using the R^+-tree and Figure 12(b) illustrates the R^+-tree structure for Figure 12(a). In Figure 12(a), object rectangles 1, 2, 3, 4, 5, 6, 8, and 10 overlap with more than one intermediate rectangles. For example, rectangle 10 overlaps with intermediate rectangles 13, 14, 17, and 18. The object rectangle is not split but stored

in all intermediate rectangles that partially overlap the object rectangle [95]. In this case, coordinates for rectangle 10 in intermediate rectangles 13, 14, 17, and 18 remain the same. That is, intermediate rectangles 13, 14, 17, and 18 perform as if each of them enclosed rectangle 10 completely. This causes redundancies at the leaf node level and as a result the total number of nodes in the tree are increased as shown in Figure 12(b).

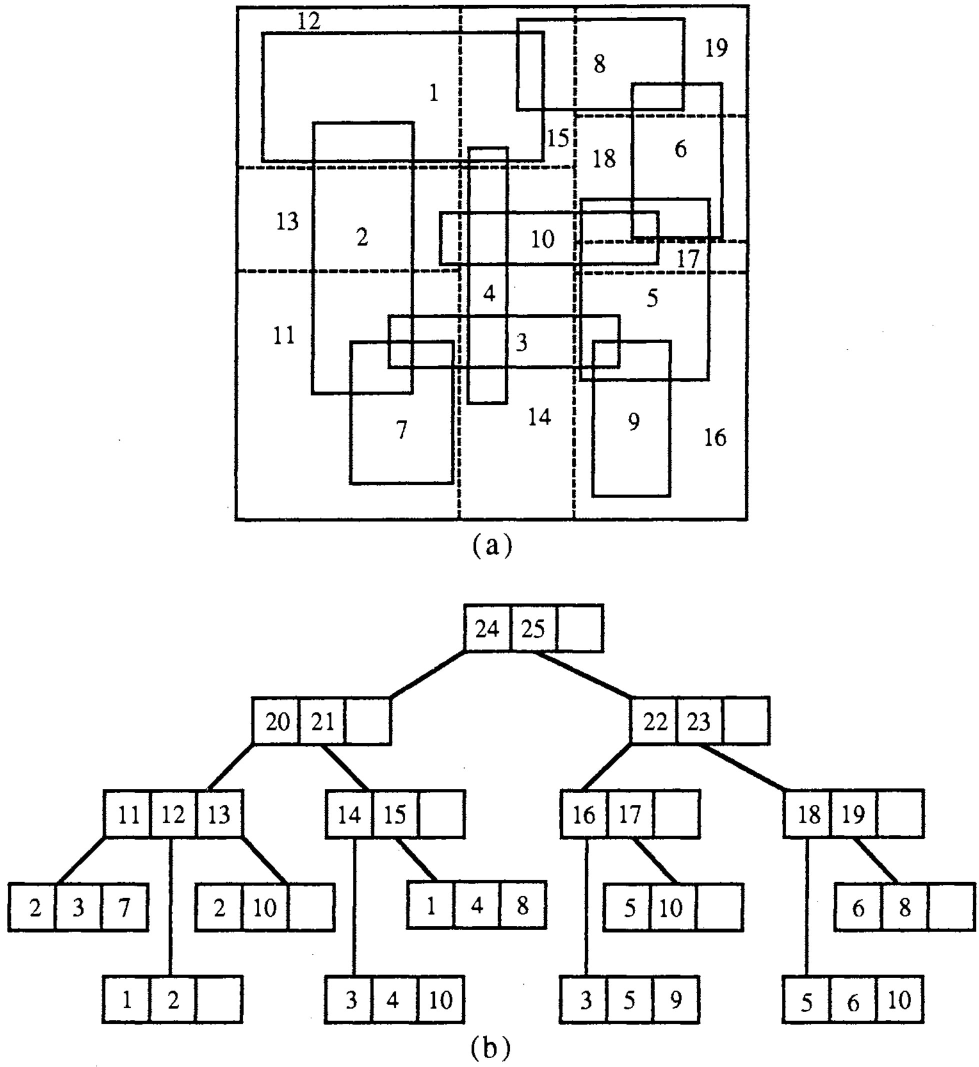


Figure 12 (a) organizations of bounding rectangles (the solid lines construct object rectangles; the dash lines construct intermediate rectangles) and (b) the R^{+}-tree structure.

In Figure 12(a), the top 2 level intermediate rectangle boundaries are omitted to avoid too many lines. Net space (node) utilization is decreased because of the redundancies. Redundancies in the R^+-tree provide faster access in point queries and in very small size range queries than the R-tree. However, it is also noted that redundancies at the leaf node level causes disadvantages in range queries and deletion operations, since these operations access all nodes containing redundant entries overlapping the search range. The R^+-tree has high leaf node redundancy for the data objects with a high degree of overlap.

The original literature for the R^+-tree [95] does not provide insertion and split algorithms in detail. The R^+-tree may become extremely complicated to implement if it uses minimum bounding rectangles (MBRs) to enclose lower level rectangles. For example, in Figure 13(a) (page 26), none of the intermediate rectangles, 1, 2, 3, and 4 can include a new data rectangle N without overlapping other intermediate rectangles. To resolve overlapping boundaries of intermediate rectangles after insertion of N, the R^+-tree insertion algorithm should search all the neighbors overlapping an intermediate rectangle in which data rectangle N is inserted and adjust boundaries between them. There is no guarantee that all the overlapping intermediate rectangles are siblings under the same immediate parent node. Therefore, extra node accesses may be required. Also, in some cases, adjusting boundaries of the intermediate rectangles can invoke a chain reaction; boundaries of some intermediate rectangles which do not overlap the intermediate rectangle, in which new data rectangle N is inserted, may need to be adjusted. If a minimum bounding rectangle for intermediate rectangles of the R^+-tree is not used, then there are not always gaps between intermediate rectangles as in the k-d-b tree. Figure 13(b) shows the R^+-tree space decomposition without using MBR.

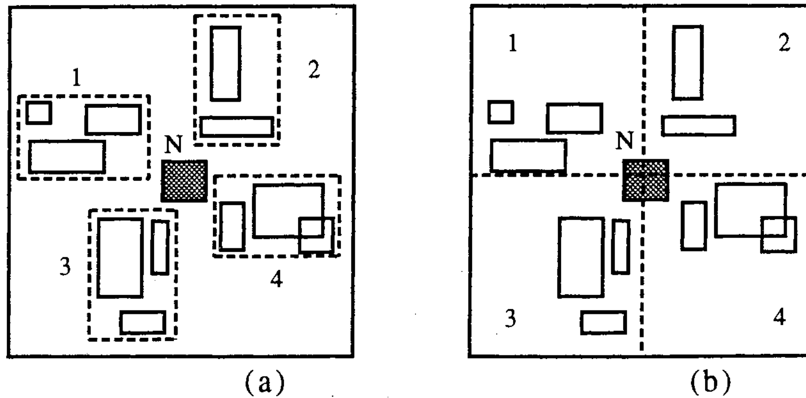


Figure 13 the R+-tree (a) with MBR and (b) without MBR.

The cell [35, 36, 37] method uses a disjoint decomposition approach. This structure can be viewed as a combination of a binary space partitioning (BSP) tree [28] and an R+-tree. The cell tree is a balanced tree and leaf nodes contain the cells and intermediate nodes correspond to a hierarchy of nested convex polyhedron. Each leaf node entry consists of 2-tuples (e.g., $E.Z$ represents the geometry of the cell and $E.D$ denotes the object ID). Intermediate nodes contain entries which consist of 3-tuples (e.g., Cp, P and C). Cp is a pointer to the child node. P is a convex k-dimensional polyhedron from the binary space partitioning. C is the container which is also a d-dimensional convex polyhedron. However, C is the sub-set of P and provides a more accurate localization of the cells in the sub-tree to speed up search processes. To insert an object, one must compute a convex cover for the object. If the object is not convex, then the object must be decomposed into a small set of convex polyhedra [34]. Each convex polyhedron is inserted into the cell-tree. Decomposition of an object into a set of small convex polyhedra may provide more accurate approximation of the object. However, this structure has a more serious redundancy problem than the R+-tree. While the leaf nodes of the R+-tree contain as many duplicated data objects as the number of intermediate rectangles intersecting a given data object, the leaf nodes of the cell-tree contain as many duplicated data objects as the number of

intermediate cells intersecting a given polyhedron which is a part of a data object. The cell-tree and the R^+-tree cannot properly perform deletion by range operations, i.e., delete all data object intersecting a given deletion range, since deletion algorithms provided by these structures cannot delete all duplicated data objects from the tree.

The grid file [67, 41] splits an overflowing cell into two sub-cells. The simplest splitting policy is to select the dimension cyclically with a fixed order. With other splitting policies, one dimension can be selected more frequently as the split dimension than the other. Since this splitting variation increases precision of the answer, it is good for the queries in which a favored attribute (dimension) is specified [67]. Figure 14 (page 28) shows the organization of ten rectangles, which are the same as in Figure 12(a), using the grid file. Capacity of the cell is three and cyclically a splitting dimension is selected. To organize ten rectangles, the grid file uses 23 cells. There are many rectangles which are stored in more than two cells; for example, rectangle 3 is stored in 7 cells. A storage space of 23 cells, with capacity 3, can hold 69 rectangles. However, only 10 rectangles are stored in 23 cells. Space utilization for this example is 14.49%. In geographic applications, overlapping and density of the data are higher than this example. Redundancy of the grid file degrades space utilization and query response time. The EXCELL method [101, 102] uses the same approach as the grid file. To index non-zero sized objects, this method duplicates objects in all cells that the objects intersect.

The multi-layer grid file [99] uses a multi-layer paradigm. This method limits the number of layers to three. In the first and second layers, there is no clipping, however clipping is allowed in the third layer . This method also has a redundancy problem as does the grid file. In this method, once a split position is fixed it cannot be adjusted unless a merge operation is done. Therefore, successive insertions and deletions can make this structure degenerated (e.g., low space utilization). Also, in the case of

bucket (node) under flow, merging a sub-space with one of its neighbors is not always possible.

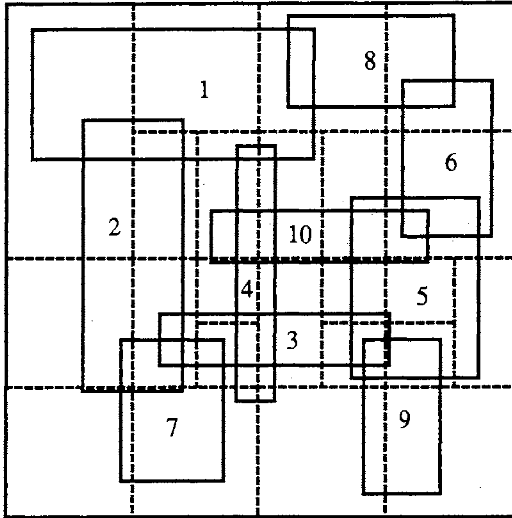


Figure 14 the grid file with CAP= 3.

The BANG file is extended [26, 27] to handle non-zero sized data objects. The extended BANG file uses the same partition scheme, binary division, as the original BANG file. The extended BANG file combines two distinct but related representations, one point and one spatial, within a single directory structure. Point representation is for processing containment queries and spatial representation is for processing intersection (partial containment) queries. For point representation, the center points of objects are used. However, processing of the containment query using point representation needs extra steps to remove objects whose center points lie in a specified region but are not completely enclosed by the specified region. An entry of the directory node consists of 4-tuples, a pointer to a child node, a cover region level number, a point region level number, and a binary partition string. Level numbers for point regions and cover regions indicate the number of bits in the binary partition string representing each region.

The splitting operation of the R-file [45] decomposes a splitting cell into three cells, two disjoint halves and an original cell. Then the rectangle in the splitting cell, is stored in the smallest cell which can enclose that rectangle completely. In Figure 15 (page 30), the R-file splitting method is illustrated using the same example as in Figure 12(a). One original data space cell is split into two as shown in Figure 15(b) and the right half cell is split into two sub-cells as shown in Figure 15(c). This method partitions the rectangles among the original cell and a sub-cell created from the original cell by repeated halving [45]. However, there are problems with this method. If more than CAP, capacity, rectangles are on the splitting line of one cell, then splitting this cell cannot resolve overflowing. For example, in Figure 15(a), insertion of rectangle 10 makes the original cell overflow. To solve this problem the R-file literature proposes to use one-dimensional the R-file to organize the rectangles according to their projection on the split line. In range queries, cells which do not contribute to answer the queries, have to be searched. Rectangles in the large cell tend to intersect the split line and this cell has empty space off the split line. To prevent excessive cell searching in range queries, the R-file associates each cell with one-dimensional interval bounding the coordinates of the rectangles in the split dimension. This information is stored in a directory.

The priority R (PR) file [6] is proposed for the maintenance of seamless, scaleless maps. In cartography, generalization is a technique which represents geographic objects depending on the scale. That is, a geographic object appears on a map only if its priority is high enough. For a priority query, data objects with the same associated priority can be stored in a separate spatial index structure. The PR-file integrates all single priority structures into one common access structure using the R-file.

The MR-tree [4, 5] is a spatial index structure using a multi-layer paradigm. It distributes spatial objects into several data spaces to avoid redundancy in the leaf

nodes. Space utilization of the MR-tree is considered to be low and the object distribution in the data spaces is not balanced.

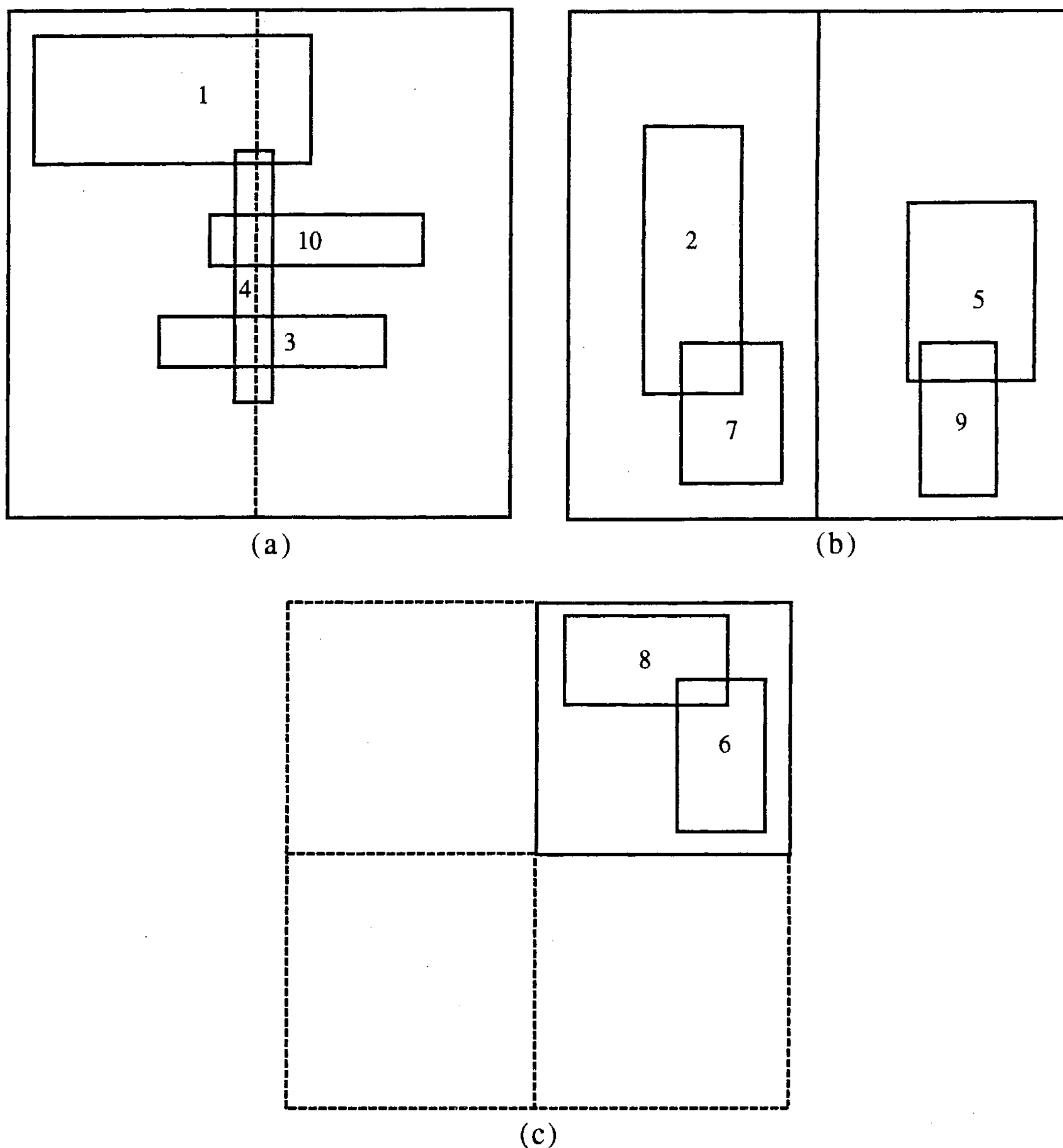


Figure 15 the R-file with CAP= 3.(a) data space cell, (b) two half cells and (c) one quarter sub-cell.

The quadtree [9, 21, 42, 46, 65, 66, 89, 90, 96, 97] is a hierarchical index structure which recursively decomposes space. The quadtree sub-divides the image array into

four equal-sized quadrants. If the array does not consist entirely of 1's or 0's, then it is sub-divided into sub-quadrants, until all blocks in the image array consist of entirely of 1's or 0's. The image array is represented by the tree of degree four. This type of quadtree is not efficient in dealing with paging and disk I/O buffering due to its small number of branching factors. The linear quadtree [29, 104] is introduced to partially alleviate this problem. It stores only black nodes, encodes each node using the Morton address scheme and encodes a path from the root to the node. Morton addresses are produced by interleaving the bits of binary representation of x and y coordinates. In Figure 16(c), block address 22 is created from bit string 1010 by using base-4 digits. The sequence, 12 21 30 31 32, represent the image in Figure 16(a). A node in the linear quadtree is split by replacing it with its four children.

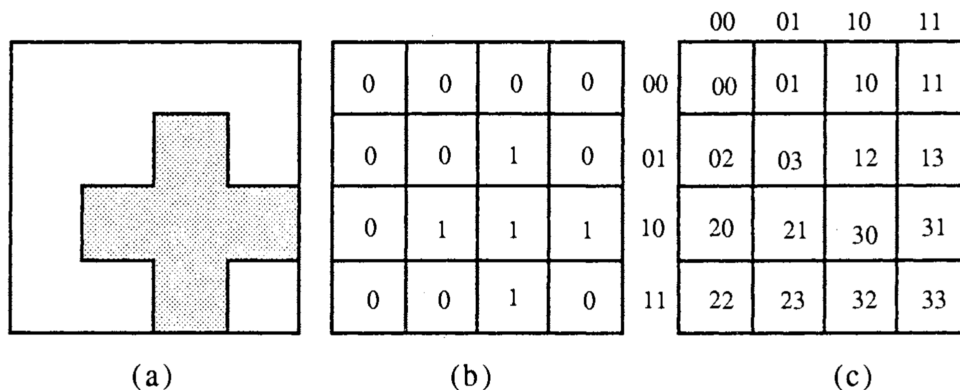


Figure 16 (a) image, (b) binary image array and (c) the Morton code addressing scheme for labeling pixels.

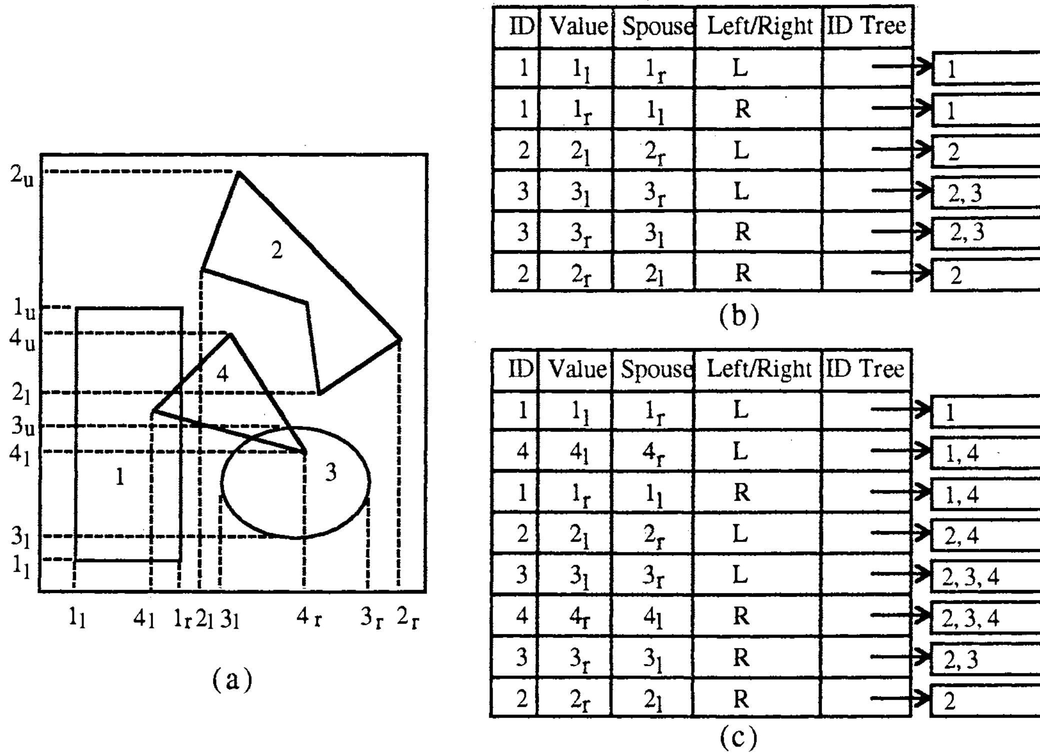
To organize a sorted list of quadtree blocks, some one-dimensional index structure (e.g., B-tree) should be used. This type of quadtree cannot handle overlapping data objects properly. The linear quadtree assumes that all the blocks are disjoint and cover the entire image. Spatial data usually does overlap (e.g., geographic data containing satellite photographs of regions of the earth [63]). The quadtree indiscriminately decomposes the objects into lower level pictorial primitives, such as

quadrants, line segments, or even pixels. In a search operation, the quadtree requires elaborate re-construction of the spatial object from the low level primitives of the leaves [86]. To re-construct the image, the quadtree requires a list of size, location and value of each of the blocks comprising the image. Usually, the quadtree methods are suitable for image processing

Projection Method

The spatial index structure proposed by Lee [57] uses projections of the minimum bounding rectangle (oriented parallel to the coordinate axis). Projections of the boundary points of the objects are shown in Figure 17(a). For example, boundary points of the object 2 on the horizontal axis are represented by 2_l and 2_r where l and r denote left and right, respectively. For the vertical axis, 2_l and 2_u represent lower and upper boundary points of the object 2, respectively. An array is associated with each axis and an entry in the array consists of 5-tuples (object ID, coordinate value of the projection boundary point, coordinate value of the matching projection boundary point, boundary indicator, and pointer to a 2-3-4 tree). Figure 17(b) shows an array structure organizing projected boundary points on the horizontal axis before the insertion of object 4. All object IDs which overlap a specified coordinate value are stored in the 2-3-4 tree associated with that coordinate. For example, objects 2 and 3 are stored in the 2-3-4 tree associated with the projected coordinate value 3_l, since these objects overlap value 3_l on the horizontal axis. To process a range query, for the horizontal axis, find a set of objects X from ID trees associated with the projected values from x_i to x_j ($i \leq j$ and $S_l \leq x_i \leq x_j \leq S_r$ where S denotes a search range). Similarly, for the vertical axis, a set of objects Y can be found. Objects overlapping a given search range S can be obtained from intersection of the two sets X and Y. Insertions and deletions of this method are very inefficient, since this method uses

arrays to organize projected boundary coordinates. To insert an object in the two-dimensional case, this structure needs to shift entries in the array down to make space for each of the four projected coordinate values of the object. Also, the object ID is inserted into all ID trees associated with the projected coordinate values which overlap the interval of the inserted object on the given axis.



ID	Value	Spouse	Left/Right	ID Tree
1	1_l	1_r	L	1
1	1_r	1_l	R	1
2	2_l	2_r	L	2
3	3_l	3_r	L	2, 3
3	3_r	3_l	R	2, 3
2	2_r	2_l	R	2

(b)

ID	Value	Spouse	Left/Right	ID Tree
1	1_l	1_r	L	1
4	4_l	4_r	L	1, 4
1	1_r	1_l	R	1, 4
2	2_l	2_r	L	2, 4
3	3_l	3_r	L	2, 3, 4
4	4_r	4_l	R	2, 3, 4
3	3_r	3_l	R	2, 3
2	2_r	2_l	R	2

(c)

Figure 17 (a) projections of objects on axes, (b) array structure for horizontal projections before insertion of object 4 and (c) array structure for horizontal projections after insertion of object 4.

For example, to insert object 4, in Figure 17(a), the entries of the array for the horizontal axis have to be shifted down to make space for values 4_l and 4_r. Then, the object ID for object 4 should be inserted into all ID trees in shaded area of Figure 17(c). Update of the array for the vertical axis is analogous to the case for the horizontal axis. Even though this method suggests a modified B-tree to organize the

projected coordinate values of the objects, the update processes in this method are too expensive.

Applications

Recently, applications which rely heavily on spatial data have increased in database area [33]. In spatial database systems, spatial information of the objects is stored in separate spatial index structures and non-spatial information can be stored in database relations while maintaining appropriate links between the spatial and non-spatial components of each object [1]. Spatial index structures have applications in the geographic information systems (GIS) [24, 30, 51, 53, 73, 88, 96, 105], computer vision [14, 47], CAD, VLSI [82] and the image databases [16, 17, 20, 32, 50, 76, 83, 87, 109]. It is also used in cartography [52 , 103] and remote sensing [38] Spatial index structures can be used to support 3-dimensional image reconstruction in structural biology [8]. For example, three-dimensional images of anatomical objects are reconstructed from sets of ordered two-dimensional cross-sectional slices.

The spatio-temporal (multi-version) database which is a spatial database in which data objects may change their shapes and/or spatial locations at different time intervals [19, 61, 93, 98, 106, 107] is another application of spatial index structures. This database can be viewed as a spatial database with an additional dimension, time.

CHAPTER III

THE PML-TREE: A NEW PARALLEL SPATIAL INDEX STRUCTURE

Introduction

In this chapter, design and implementation of the parallel spatial index structure called a parallel multi-layer (PML) tree using native space indexing with the disjoint space decomposition method are proposed.

In most applications, very large sets of spatial data are stored in secondary storage (e.g., disk) and the access time for one page of secondary storage is much longer than processing time for one page in main memory [51]. This means that the performances of the spatial index structures highly depends on the number of disk accesses to the secondary storage. Parallelization of I/O operations can reduce query response time of the spatial index structure significantly, since the operations in applications of most spatial index structures are highly I/O bound. The hardware architecture for parallel spatial index structures discussed in this chapter is multiple processors with multiple disks. Parallelization using multiple disk units is suitable to handle massive data. In parallel spatial index structures, the entries which are in different disks and overlap a given search range are simultaneously accessed during query processing. An efficient parallel spatial index structure should require a small number of total disk accesses and at the same time the disk accesses should be distributed evenly among the disks. Therefore, object distribution strategy is crucial to the performances of parallel spatial index structures.

Parallelization of the spatial index structure is a comparatively unexplored topic. The MXR-tree is the first parallel spatial index structure [51]. It is a variation of the R-tree. The MXR-tree distributes R-tree nodes over multiple disks using heuristics

(e.g., round robin, minimum area or proximity index) as discussed in Chapter II. The proximity index gives the MXR-tree the best performance. The purpose of the proximity index is to place a newly created node to the disk with the lowest proximity index (e.g., a disk with the least similar nodes, spatially, with respect to the newly created node). For ideal node distribution, all nodes on the same level of the MXR-tree have to be considered in the calculation of the proximity index. However, this method requires too many extra disk accesses. Therefore, the proximity index heuristic only considers sibling nodes under the same parent node to calculate the proximity index. The MXR-tree uses the same split algorithm as the R-tree. Thus, the MXR-tree has a redundant search path problem. Query operations in the parallel spatial index structures create the same number of processes as disks, with each process being associated with one disk. Each process performs a query operation simultaneously for a given search range on each disk. Since, the MXR-tree has a single tree structure on multiple disks, nodes in different disks are linked by the pointers. Therefore, during parallel processing of the query operation, inter-process communications are required and consequently all disks cannot be fully parallelized all the time. For example, assume that nodes N_1 and N_2 are in disks D_1 and D_2, respectively. If an entry E in node N_1 overlaps the given search range and a child node of entry E is node N_2, then node N_2 in disk D_2 cannot be accessed until the process associated with disk D_1 completes processing of the entry E and sends a signal to the process associated with disk D_2. Detailed inter-process communication of the MXR-tree is discussed in Chapter IV, Implementations of Spatial Index Structures.

The purpose the PML-tree is to increase query performances by:

1. reducing the total number of disk accesses,
2. distributing data objects evenly among the disks for parallel processing and
3. high space (node) utilization.

To reduce the total number of disk accesses in query processing, a spatial index structure should overcome disadvantages of the R-tree, R*-tree and R+-tree (see Chapter II). The PML-tree distributes data objects in multiple data spaces to remove overlaps between intermediate rectangles and to avoid leaf node redundancy. To speed up query processing in parallel disk I/O, the PML-tree distributes data objects evenly (in terms of the number of objects and spatial distribution) among the multiple disks using object distribution heuristics while maintaining the properties mentioned above. Object distribution heuristics of the PML-tree provide high space utilization. The author proposes three object distribution heuristics for the PML-tree in this chapter. Three different PML-trees with three object distribution heuristics and one MXR-tree with the proximity index heuristic are implemented in Chapter IV. Using four test data sets (e.g., two system generated data sets and two real application data sets), the performances of these four parallel spatial index structures are compared and analyzed. The structure and the algorithms of the PML-tree are discussed in the following.

The Structure and Algorithms of the PML-tree

The PML-tree Structure

An efficient spatial index structure should satisfy at least the following properties [60]:

1. good space (node) utilization with a large index fanout, resulting in a small number of nodes and a small number of disk accesses to process a query,
2. easy incremental re-organization as the file grows,
3. simple algorithms with an absence of special cases,
4. ability to handle range queries and exact match queries.

The PML-tree should satisfy the above properties and it must have proper structure for parallel processing. The PML-tree is a dynamic parallel spatial indexing structure using native space with the disjoint space decomposition scheme. In the following, three structural aspects of the PML-tree are discussed.

First, to decrease the number of disk accesses in a query, the PML-tree should overcome the redundant search path problem of the R-tree or R*-tree and the redundancy problem of the R^+-tree. The PML-tree uses native space indexing with a disjoint space decomposition method. The disjoint space decomposition method is used to remove redundant search paths. The PML-tree guarantees that all sub-rectangles are completely enclosed by an upper level intermediate rectangle to remove a redundancy problem. For this, multiple data spaces are used and these multiple spaces are suitable for parallelization using multiple disks.

Second, to increase query performance of the parallel spatial index structure on a system with multiple disks, some object distribution heuristics are required. In parallel spatial index structures, the query performance is proportional to the maximum number of disk accesses among the disks. An efficient object distribution heuristic reduces the query response time by distributing data objects spatially evenly among the disks. Some of the heuristics (e.g., round robin or minimum area [51]) do not consider spatial locations of the data objects to be inserted and these heuristics are structure independent. Some heuristics (e.g., proximity index of the MXR-tree) consider spatial locations of the data objects to be inserted with other data objects which already exist on the disks. The PML-tree uses three object distribution heuristics:

1. minimum number of entries - selecting a tree that has minimum number of entries among the trees in the same layer;
2. absolute crowd index - selecting a tree with the minimum probability of the node splitting along the insertion path among the trees in the same layer;

3. relative crowd index - selecting a tree with relatively less crowding along the insertion path among the trees in the same layer.

These three object distribution heuristics are discussed in detail in this chapter with algorithms and examples.

Third, space utilization is one of the factors that affects performance of the spatial index structures. A spatial index structure with higher space utilization is space efficient and has a smaller number of nodes than the spatial index structure with lower space utilization. In point queries and very small size range queries, high space utilization may not reduce the number of disk accesses. However, high space utilization becomes an important factor in reducing the number of disk accesses as the search range grows. For example, if the search range is an entire data space, then all the nodes in the structure have to be accessed. In this case, structures with higher space utilization take fewer disk accesses. The split algorithms and object distribution heuristics of the PML-tree provide high space utilization.

The PML-tree has a multiple-layer, multiple-tree structure. Each layer consists of multiple trees and the number of the trees is the same as the number of the disks used. Each tree is a height balanced tree and all trees in the same layer have the same heights. Each tree in each layer is associated with a data space. The PML-tree distributes spatial objects into multiple data spaces by using object distribution heuristics and splitting algorithms. Each data space is associated with a tree which is an independent structure and is placed on one of the disks. A disk can hold more than one tree. No inter-process communications during query operations is necessary, since in the PML-tree, each of the trees is an independent structure. Therefore, all disks can be fully parallelized. Figure 18 illustrates the PML-tree layout with three-layers and four-trees. Each tree can be identified by tree ID, T_{ij} where i denotes the layer number and j denotes the tree number; the tree number is also the disk number. Disk j contains trees with ID number T_{ij} (i= 1, 2, .. L, L is the number of layers). For

example, in Figure 18, disk 1 contains trees T_{11}, T_{21}, and T_{31}. Therefore, the query response time of the PML-tree in parallel accessing of the disks is proportional to the maximum number of the disk accesses. If P_{ij} is the processing time of the nodes in tree T_{ij} on a range query, then query response time (R) of the PML-tree can be obtained by:

$$R = \max_{j=1 \ldots D} \left(\sum_{i=1}^{L} P_{ij} \right)$$

where D is the number of disks used and L is the number of layers.

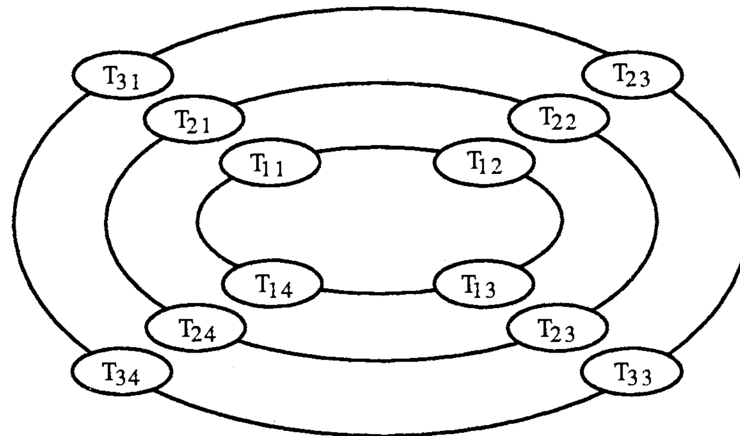


Figure 18 the PML-tree layout with three-layer four-tree.

Figure 19 illustrates an example of object distribution of the PML-tree with the same two-dimensional objects used in Chapter II. The PML-tree retains the same node structures as the R-tree, R*-tree and R^{+}-tree as described in Chapter II. All objects are stored in leaf nodes. Intermediate nodes consist of entries that represent intermediate rectangles. For example, entries in the intermediate node in Figure 20(a) are 11 and 12. Those entries represent the intermediate rectangles 11 and 12 in Figure 19(a), respectively. In this example, the PML-tree uses the simplest object distribution heuristic, minimum number of entries, to distribute the object rectangles 1

through 10 into the three data spaces. In Figure 19, all the data spaces contain disjoint intermediate rectangles and the rectangles are completely enclosed by an upper level intermediate rectangle. Figure 20(a), (b), and (c) show the corresponding trees of the 1st, 2nd and the 3rd data spaces. In Figure 20, *ni* denotes node ID, and all the nodes in tree T_{ij} are stored on disk j. At the leaf level, there are no duplicated object rectangles. In this case, the PML-tree has a one-layer, three-tree structure.

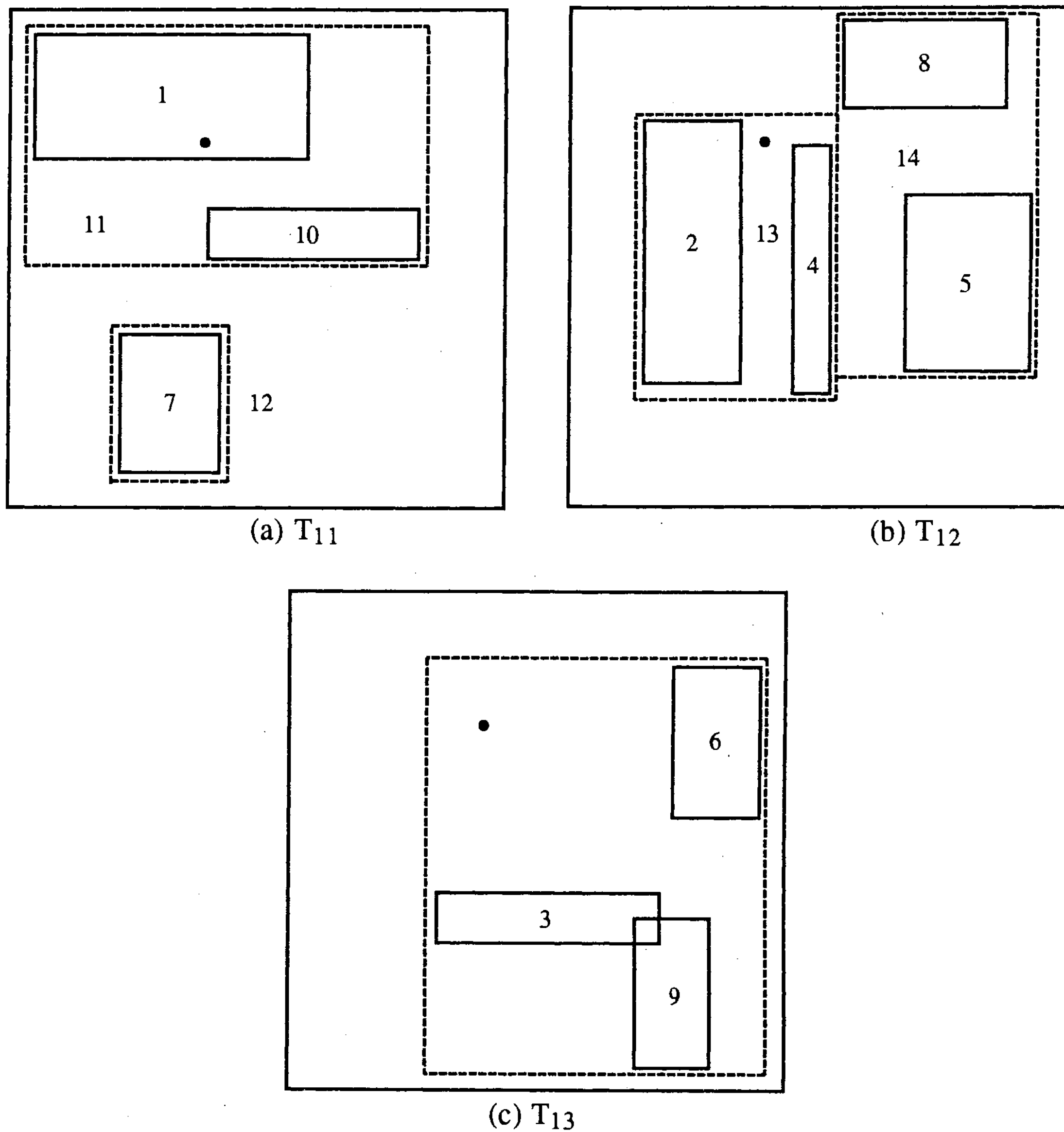


Figure 19 organizations of bounding rectangles (the solid lines construct object rectangles; the dash line construct intermediate rectangles) on the (a) first data space, (b) second data space and (c) third data space.

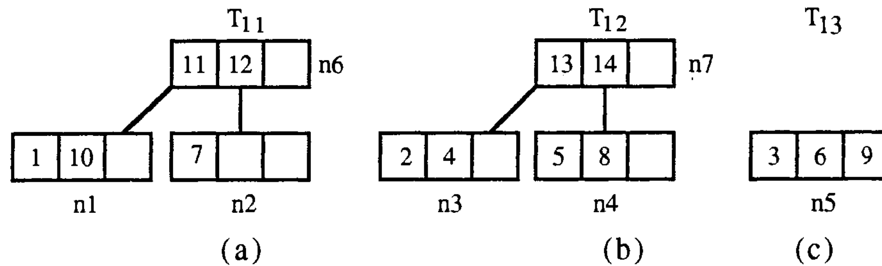


Figure 20 the PML-tree structures for the (a) 1st data space in Figure 19 (a), (b) 2nd data space in Figure 19(b) and (c) 3rd data space in Figure 19(c).

Figures 21(a) and 21(b) compare the parallel disk access methods of the MXR-tree in Figure 11 (page 22) and the PML-tree in Figure 20, respectively, to process a given point query. In Figure 11, *Di* denotes disk ID and *ni* denotes node ID. The search area is marked as a point in Figures 8(a) (page 16) and 19; those are the data spaces for the MXR-tree and the PML-tree, respectively. In Figure 21, a division on the horizontal axis represents unit time for a disk access. Let's assume that the time to access a disk is constant and the size of a node is one page. Only the data object 1 overlaps the given point. In Figure 21(a), the MXR-tree needs to access 5 nodes to process a given point query and the time to access these 5 nodes in parallel is 4. The total number of nodes accessed by the PML-tree for the same point query is 5 and time to access these 5 nodes in parallel is 2 as can be observed in Figure 21(b). To distribute nodes over the disks, the proximity heuristic of the MXR-tree only considers the sibling nodes under the same parent node; therefore, the proximity heuristic can produce unbalanced node distributions in some cases. For example, in Figure 11, nodes *n1* and *n5* are on the same disk *D2* and corresponding intermediate rectangles for these two nodes are close to each other in data space in Figure 8(a). To obtain good performance, these two nodes have been placed on different disks. As mentioned previously, inter-process communications are involved with every node accesses in the MXR-tree. For example, in Figure 21(a), a process associated with

disk *D1* has to send a signal after processing the entry 15 in node *n7* (see Figure 11) to the process associated with the disk *D2* to access node *n5*. In the PML-tree, each process associated with each disk only accesses nodes that have entries overlapping the given search range, and no inter-process communications are required. The algorithms of the PML-tree are discussed in the following section.

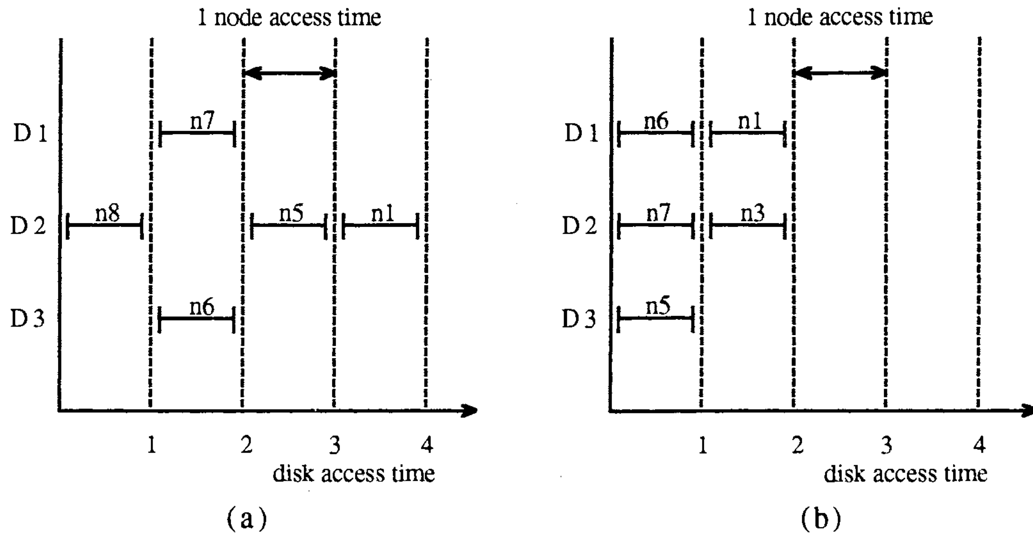


Figure 21 parallel disk access times of: (a) the MXR-tree in Figure 8(a); (b) the PML-tree in Figure 19.

Algorithms of the PML-tree

All the algorithms and examples proposed in this chapter are based on two-dimensional data objects. However, they can be easily extended to higher dimensional data objects. Using C-like structures, all symbols and data types used in algorithms are defined in the following:

```
#define   CAP                                  /* node capacity          */
#define   D                                    /* number of disks used   */
#define   W                                    /* weight                 */
#define   M        /* initial balancing factor; set to 30% of CAP */
```

```
struct RECT {                    /* the four coordinates of a 2-d rectangle      */
      int   lower_x, lower_y, upper_x, upper_y;
};

struct ENTRY {
      struct RECT   *R;                   /* the four coordinates of a 2-d rectangle */
      struct NODE   *P;  /*pointer to a child node or to a record in the database */
};

struct NODE {
      int   ent_num;                        /* number of entries in the node     */
      struct ENTRY   *ETY[CAP + 1];               /* array of entry pointer  */
};

struct TREE_INFO {
      int   total_ent;                      /* number of entries in the tree     */
      struct NODE   *P;                     /* root node pointer of the tree     */
};

struct LAYER {
      struct TREE_INFO   *TREES[D];
      struct LAYER   *next;
};

struct CROWD {
      int   order;                                          /* tree order     */
      int   value;                                   /* crowd index value     */
      char   status;                        /* tree status for NEW object     */
};

struct RECT          *W;                          /* search or delete range */
struct ENTRY         *E, *E', *NEW;                    /* entry pointers     */
struct NODE          *N, *SP, *P, *R, *REM;            /* node pointers      */
struct LAYER         *LAY, *LAY_LIST;                  /* layer pointers     */
struct CROWD         *CI, *CI_List[D];
int     i;                                             /* Tree or disk ID    */
```

Structure ENTRY consists of two parts, structure R and structure P. In the case of leaf level entry, R represents a minimum bounding rectangle of a spatial data object and P is a pointer to the record related to the spatial data object in the database. In the case of an intermediate entry, R represents a minimum bounding rectangle for all

the rectangles associated with the entries in the child node and P is a pointer to a child node; in the algorithms defined herein, P is defined only as a child node pointer. Structure NODE consists of two parts, ent_num which denotes number of entries in the node and structure ENTRY which holds entry pointers in the node. A node can hold up to CAP entries and one extra space is for an overflow entry in the case of a node split. The size of the node is set to a multiple of the actual page size. For example, if one page size is 1 Kbytes and about 30 entries can be store in 1 Kbytes, then node capacity should be set to multiple of 30 (30 * k, k= 1, 2, ..., n). Structure TREE_INFO is to keep information about each tree, the total number of entries in the tree and the root node pointer. The structure LAYER contains the structure TREE_INFOs for trees in the current layer. Structure CROWD is used for data insertion with the object distribution heuristics: absolute crowd index and relative crowd index. It consists of three parts: order, which denotes reverse height of the tree; value, which represents absolute crowd index or relative crowd index; status, which denotes status (e.g., *A*: accept, *S*: accept and split and *R*: reject) of the tree for entry NEW.

All the definitions of symbolic notation include only basic contents to simplify algorithms. In actual implementation, more information (e.g., node offset, number of nodes in a tree, file pointer, tree name, list of freed node offset, ..., etc.) should be added. Entries of the PML-tree do not need space for the disk ID of the child nodes, since all the nodes in a tree are on the same disk. However, in MXR-tree, each entry needs one extra space for the disk ID of the child node. These extra spaces reduce capacity of the MXR-tree.

In the following sections, algorithms of insertion, split, search and deletion for the PML-tree are proposed.

Insertion of the PML-tree

The PML-tree uses three object distribution heuristics. Therefore, an insertion method for each of three different object distribution heuristics is proposed. Algorithms Insert, SelectEntry and Adjust are shared by three different insertion methods.

```
Algorithm Insert ()
[Output]      a new PML-tree after insertion of NEW.
[Comment]     To insert a leaf entry NEW, invoke one of three insertion methods
              (Insert_by_ME (), Insert_by_AC () or Insert_by_RC ()). Removed
              entries are re-inserted.
1.  set LAY= LAY_LIST->next;  REM->ent_num= 0;
2.  invoke Insert_by_ME (LAY, NEW, REM), Insert_by_AC (LAY, NEW, REM)
    or Insert_by_RC (LAY, NEW, REM);
3.  if REM->ent_num > 0 then                /* if there is removed entry */
        for each entry E in REM,
        set NEW= E;  goto step 2 to re-insert removed leaf level entries;
```

Algorithm Insert takes one entry NEW to insert, sets LAY to the first layer in LAYER_LIST and invokes one of three distribution methods: Insert_by_ME, Insert_by_AC, or Insert_by_RC. If there exists data object(s) removed by node split during insertion of NEW, then each of them are re-inserted. First, the simplest object distribution heuristic, minimum number of entries, is considered.

Minimum Number of Entries Object distribution by minimum number of entries (abbreviated as ME) heuristic is the simplest distribution method. Insertion with ME distribution heuristic consists of six algorithms, Insert, Insert_by_ME, SelectTree_by_ME, InsertRect, SelectEntry, and Adjust.

```
Algorithm Insert_by_ME (LAY, NEW, REM)
[Input]       LAY- structure LAYER.     NEW- entry to be inserted.
              REM- store removed entries.
[Output]      a new PML-tree after insertion of NEW.
[Comment]     Select a tree using Select_by_ME(), then invoke InsertRect() to insert
              NEW.
1.  invoke i= SelectTree_by_ME (LAY, NEW);      /* i denotes selected tree ID */
    let N be the root node of the ith tree, N= LAY->TREES[i]->P;
2.  invoke SP= InsertRect(N, NEW, REM), where SP denotes a split node;
3.  if insertion of new object rectangle NEW is successful then
        if SP ≠ NULL then                                /* root node split */
            create entries E and E' for node N and SP;
            create a new root node R and insert E and E' into R as entries;
            reset root node in LAY, LAY->TREES[i]->P= R;
    else if there is no tree in which NEW can be inserted in the current layer then
        set LAY= LAY->next;  goto step 1;
    else goto step 1;
```

```
Algorithm SelectTree_by_ME (LAY, NEW)
[Input]       LAY- structure LAYER.     NEW- entry to be inserted.
[Output]      ID of the tree in which NEW can be inserted.
[Comment]     Select a tree in the current layer with a minimum number of entries.
1.  if LAY is NULL then
        create a new layer; for all i (i= 1, 2,...,D), initialize LAY->TREES[i]= NULL
        and LAY->TREES[i]->total_ent= 0;  and put LAY into LAY_LIST;
2.  select a tree with a minimum number of entries from trees in the current layer,
    i= {i | MIN(LAY->TREES[i]->total_ent, i= 1, 2 ... D)}, except the tree(s) which
    already rejects NEW;
3.  return i;
```

Algorithm Insert_by_ME calls SelectTree_by_ME to select a tree in the current layer. Algorithm SelectTree_by_ME selects a tree with a minimum number of data objects in it. The tree which already rejects an entry NEW is excluded from consideration. The ME distribution heuristic gives balance among the trees in terms of number of data objects. With a selected tree, algorithm Insert_by_ME invokes algorithm InsertRect. If entry NEW is inserted at the leaf node of the selected tree and root node is split, then a new root node is created. In the case that the selected

tree cannot accept entry NEW, goto step 1 and repeat the insertion process. If no trees in the current layer accept entry NEW, then move to the next lower layer and repeat the insertion process.

Absolute Crowd Index The ME distribution heuristic provides perfect balancing among the trees in terms of number of data objects. However, spatial data objects can have arbitrary shapes and sizes. Also, data object can overlap each other and are not necessarily uniformly distributed in the data space. That is, certain areas in the data space can be much denser than other areas. Average density of data objects can be obtained by dividing the sum of data objects' areas by area of data space. For example, average density 2 means that 2 data objects overlap any given point in data space. In query processing, D processes are created and each process searches trees in the associated disk for the same search range. Therefore, to obtain good performance, each of the trees in the same layer of the PML-tree should approximately the same number of nodes in the search path. Object distribution by the ME heuristic may not provide balance among the trees in terms of the number of nodes in the search range, since, in insertion, it does not consider the locations of the objects. A tree with higher probability of node splitting along the insertion path tends to create more nodes (intermediate nodes and leaf nodes). As a result more nodes have to be accessed upon query processing of the area corresponding to the insertion path, with a higher probability of node splitting. Object distribution by absolute crowd index (abbreviated as AC) finds a tree with the lowest probability of node splitting along the insertion path to insert entry NEW. The AC distribution heuristic measures crowdness of nodes along the insertion path to determine probability of node splitting. The AC index value for each of the trees can be calculated as follow:

$$\sum_{j=1}^{h} (W \times j \times N_j\text{->ent_num})$$

where h is the order of the tree, W is a weight constant and N_j is the node with order j in insertion path.

Weight constant W is used to give more weight to a node with a higher order. A higher ordered node is more likely split by the insertion of entry NEW, since NEW is inserted in a leaf node and the leaf node has the highest order. A tree with the lowest AC index value is selected for insertion of an entry NEW. Insertion by AC distribution heuristic provides high space (node) utilization. Space utilization of the parallel spatial index structures for various test data sets are given in Chapter IV.

Insertion by the AC distribution heuristic consists of seven algorithms, Insert, Insert_by_AC, SelectTree_by_AC, Calc_AC, InsertRect, SelectEntry and Adjust. Algorithm Insert_by_AC invokes algorithm SelectTree_by_AC to select a tree in which entry NEW can be inserted. Algorithm SelectTree_by_AC creates D processes and each process i (i= 1, 2, ..., D) invokes algorithm Calc_AC for each tree i in the current layer to obtain information (tree order, absolute crowd index value and tree status). Algorithm Calc_AC descends a tree from root node to leaf node by using algorithm SelectEntry. Algorithm Calc_AC is executed in parallel on all disks. Algorithm SelectEntry returns either an entry that can include entry NEW or NULL if no entry in node N can accept NEW. For each node in the insertion path, crowd index value is calculated and this value is cumulative. If a leaf node can accept NEW, then the tree status is set to either *A* or *S* depending on the number of entries in the node N. For the trees with status either *A* or *S*, algorithm SelectTree_by_AC selects a tree using four criteria (tree order, absolute crowd index value, tree status and number of objects in the tree). If no tree in the current layer accepts NEW, then the current layer is set to the next lower layer and the selection process is repeated.

```
Algorithm Insert_by_AC (LAY, NEW, REM)
[Input]      LAY- structure LAYER.      NEW- entry to be inserted.
             REM- store removed entries.
[Output]     a new PML-tree after insertion of NEW.
[Comment]    Select a tree using Select_by_AC (), then invoke InsertRect() to insert
             NEW.
1.  invoke i= SelectTree_by_AC (LAY, NEW);        /* i denotes selected tree ID */
    let N be the root node of the ith tree, N= LAY->TREES[i]->P;
2.  invoke SP= InsertRect(N, NEW, REM), where SP a denotes split node;
3.  if SP ≠ NULL then                                  /* root node split       */
        create entries E and E' for node N and SP;
        create a new root node R and insert E and E' into R as entries;
        reset root node in LAY, LAY->TREES[i]->P= R;
```

```
Algorithm SelectTree_by_AC (LAY, NEW)
[Input]      LAY- structure LAYER.      NEW- entry to be inserted.
[Output]     ID of the tree in which NEW can be inserted.
[Comment]    Select a tree using four criteria (height, crowd index value, tree status
             and number of entries).
1.  create D child processes;
2.  each child process i (i= 1, 2, ..., D) invokes Calc_AC (CI_list[i], NEW,
    LAY->TREES[i]->P) in parallel;  exit;
3.  if all CI_list[i]->status is R (Reject) then
        set LAY= LAY->next;
        if LAY is NULL then
            create a new layer; for all i (i= 1, 2, ..., D), initialize
            LAY->TREES[i]= NULL and LAY->TREES[i]->total_ent= 0;
            insert LAY into LAY_LIST;  select the 1st tree, i= 1;
        else goto step 1;
    else, for all trees with status A (Accept) or S (accept and Split),
        select a tree i with minimum height,
            i= {i | MIN (CI_list[i]->order, i= 1, 2, ..., D)};
        resolve ties by choosing a tree with minimum crowd index value,
            i= {i | MIN (CI_list[i]->value, i= 1, 2, ..., D)};
        resolve ties by selecting a tree with status A over status S;
        resolve ties by choosing a tree with a minimum number of entries,
            i= {i | MIN (LAY->TREES[i]->total_ent, i= 1, 2, ..., D)};
4.  return i;
```

```
Algorithm Calc_AC (CI, NEW, N)
[Input]      CI- structure CROWD.  NEW- entry to be inserted.   N- node.
[Output]     information of each tree (e.g., order, crowd index value and status).
[Comment]    Decide order of tree, crowd index value and status.  Status characters
             A, R and S denote Accept, Reject and Split, respectively.
1.   initialize CI->order= 1, CI->value= 0 and CI->status= R;
2.   if N is NULL then                                          /* empty tree   */
          set CI->status= A and exit;
3.   if N is not leaf node then
          E= SelectEntry (N, NEW);                       /* E is an entries in N    */
          if E ≠ NULL then                          /* calculate crowd index value */
               CI->value= CI->value + (CI->order * W * N->ent_num);
               set N= E->P;                               /* move to child node      */
               increase CI->order by one;  goto step 3;
          else exit;
4.   if N->ent_num < CAP then                                  /* N is a leaf node   */
          CI->status= A;                             /* can accept NEW in node N */
     else CI->status= S;           /* node N need to be split after insertion        */
     CI->value= CI->value + (CI->order * W * N->ent_num);
```

Figures 22 and 23 illustrate insertion by the AC distribution heuristic. In this example, node capacity is 3, the number of disks used is 2, weight constant W is set to 3 and data space is 40 x 40. Figures 22(a) and 22(b) represent the first data space and tree, respectively, before the insertion of data object 5. Figures 23(a) and 23(b) represent the second data space and tree, respectively, before the insertion of data object 5. In algorithm SelectTree_by_AC, the values of four criteria after invoking algorithm Calc_AC for two trees are illustrated in Table 1. The CI->values of the first and the second trees are calculated in the algorithm Calc_AC as follow:

$(3 \times 2) + (6 \times 2) + (9 \times 2) = 36$ and

$(3 \times 2) + (6 \times 3) + (9 \times 2) = 48$

Algorithm SelectTree_by_AC selects the first tree to insert data object 5. In this example, the first and the second trees have the same number of entries in leaf nodes.

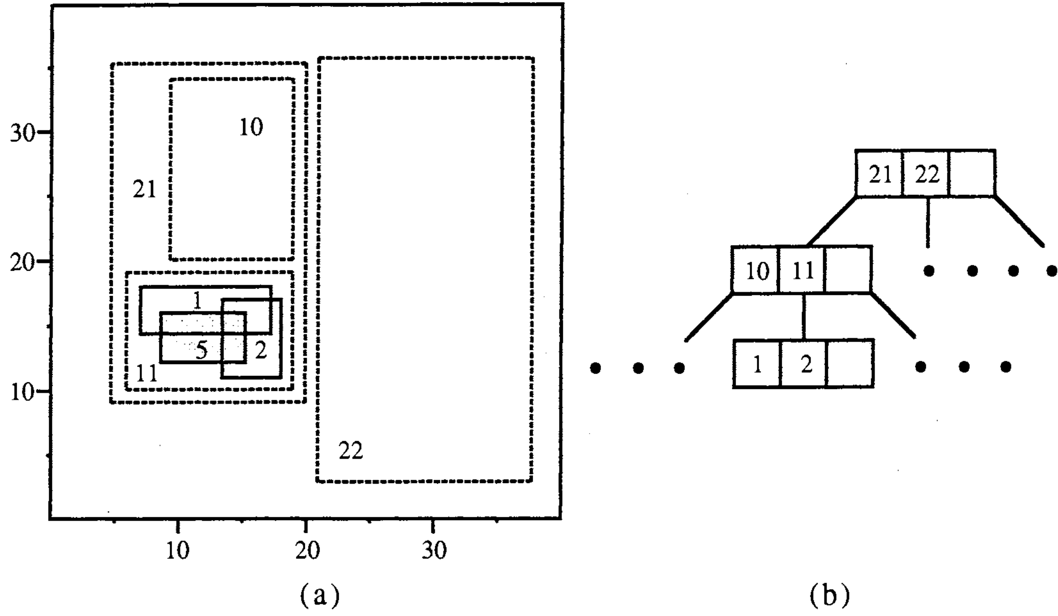


Figure 22 an example for absolute crowd index and relative crowd index object distribution heuristics (a) the first data space and (b) nodes on insertion path of the first tree.

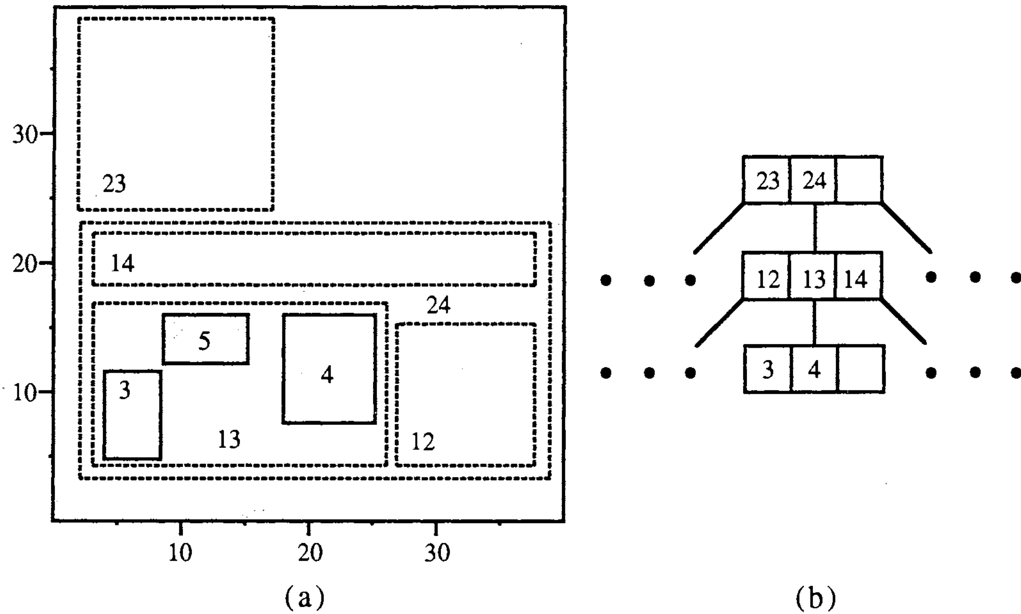


Figure 23 an example for absolute crowd index and relative crowd index object distribution heuristics (a) the second data space and (b) nodes on insertion path of the second tree.

Tree_id	CI->order	CI->value	CI->status	total_ent
1	3	36	A	-
2	3	42	A	-

Table 1 contents of structure CI for two trees in algorithm SelectTree_by_AC.

However, the node with order 2 in the second tree has one more entry than the node with order 2 in the first tree. Therefore, the probability of the node split for the second tree is higher than for the first tree.

Relative Crowd Index Relative crowd index (abbreviated as RC) distribution heuristic uses relative crowdness of the nodes in the insertion path to select a tree to insert an entry NEW. The RC distribution heuristic selects a tree with relatively less crowded insertion path considering the area sizes of the intermediate rectangles. To obtain good performance, each of the trees in the same layer of the PML-tree should have approximately the same number of nodes in the search path. That is, the number of intermediate rectangles overlapping given a search range in each of the data spaces should be nearly the same, since each of the nodes in the search path is associated with an intermediate rectangle in the data space and the PML-tree uses the disjoint space decomposition method. Therefore, each data space should have approximately the same size of intermediate rectangles at a specific area; to maintain this condition, the RC distribution heuristic calculates the average area size per entry at each visited node along the insertion path, except the root node, and accumulates that value. Obviously, the bigger the value, the less crowded the area. The RC distribution heuristic selects a tree with the least crowded nodes, relatively, along the insertion path. For example, Figure 24(a) and 24(b) show space decomposition of the two data space after insertion of data object 7. Insertions of object 7 causes the intermediate rectangles in each data space to split. However, the average intermediate rectangle size in the first data space is smaller than that in the second data space. If the shaded

rectangle represents a search range, then a search operation on the tree associated with the first data space has to access one more leaf node than a search operation on the tree associated with the second data space. Therefore, data object 7 should be inserted into the second data space to reduce the number of nodes in the search path. The RC index value for each of insertion paths can be calculated as follow:

$$\sum_{j=2}^{h} \left(\frac{M_j \times W^{(j-2)}}{N_j\text{->ent_num}}\right)$$

where h is the order of the tree, W is a weight constant, N_j is the node with order j in the insertion path and M_j denotes the area of the minimum bounding rectangle for N_j.

Weight constant W is used to normalize the sizes of intermediate rectangles associated with the nodes in the insertion path, since an intermediate rectangle associated with a higher ordered node is completely enclosed by an intermediate rectangle associated with a lower ordered node.

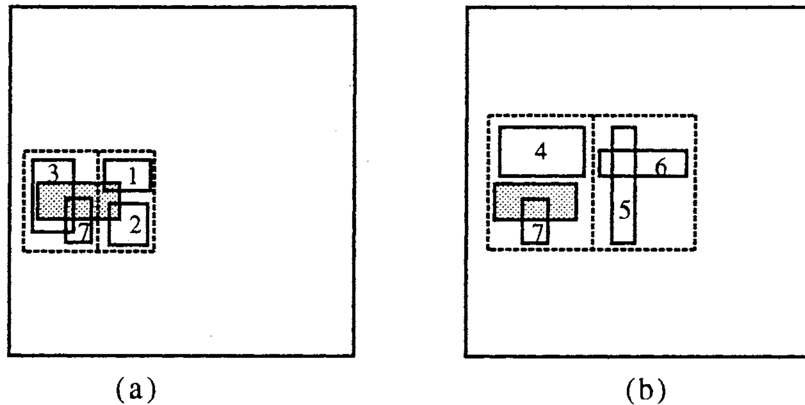


Figure 24 illustrate intermediate rectangles after insertion of data object 7: (a) the first data space and (b) the second data space.

```
Algorithm Insert_by_RC (LAY, NEW, REM)
[Input]      LAY- structure LAYER.      NEW- entry to be inserted.
             REM- store removed entries.
[Output]     a new PML-tree after insertion of NEW.
[Comment]    Select a tree using Select_by_RC (), then invoke InsertRect() to insert
             NEW.
1.  invoke i= SelectTree_by_RC (LAY, NEW);        /* i denotes selected tree ID */
    let N be the root node of the ith tree, N= LAY->TREES[i]->P;
2.  invoke SP= InsertRect(N, NEW, REM), where SP denotes split node;
3.  if SP ≠ NULL then                              /* root node split    */
        create entries E and E' for node N and SP;
        create a new root node R and insert E and E' into R as entries;
        reset root node in LAY, LAY->TREES[i]->P= R;
```

```
Algorithm SelectTree_by_RC (LAY, NEW)
[Input]      LAY- structure LAYER.      NEW- entry to be inserted.
[Output]     ID of the tree in which NEW can be inserted.
[Comment]    Select a tree using four criteria (height, tree status, index value and
             number of entries).
1.  create D child processes;
2.  each child process i (i= 1, 2 ... D) invokes Calc_RC (CI_list[i], NEW,
    LAY->TREES[i]->P) in parallel; exit;
3.  if all CI_list[i]->status is R (Reject) then
        set LAY= LAY->next;
        if LAY is NULL then
            create a new layer; for all i (i= 1, 2, ..., D), initialize
            LAY->TREES[i]= NULL and LAY->TREES[i]->total_ent= 0;
            insert LAY into LAY_LIST; select the 1st tree, i= 1;
        else goto step 1;
    else, for all trees with status A (Accept) or S (accept and Split),
        select a tree, i, with minimum height,
            i= {i | MIN (CI_list[i]->order, i= 1, 2, ..., D)};
        resolve ties by choosing a tree with status A over status S;
        resolve ties by selecting a tree with a maximum index value,
            i= {i | MAX (CI_list[i]->value, i= 1 ... D)};
        resolve ties by choosing a tree with a minimum number of entries,
            i= {i | MIN (LAY->TREES[i]->total_ent, i= 1, 2, ..., D)};
4.  return i;
```

```
Algorithm Calc_RC (CI, NEW, N)
[Input]      CI- structure CROWD.  NEW- entry to be inserted.   N- node.
[Output]     information for each tree (e.g., order, index value and status).
[Comment]    Decide order of tree, index value and status.  Status characters A, R
             and S denote Accept, Reject and Split, respectively.
int  Area;
1.   initialize CI->order= 1, CI->value= 0 and CI->status= R;
2.   if N is NULL then                                        /* empty tree  */
          set CI->status= A  and exit;
3.   if N is not a leaf node then
          E= SelectEntry (N, NEW);                   /* E is an entries in N     */
          if E ≠ NULL then                           /* calculate index value  */
               set N= E->P;                          /* move to child node       */
               Area= |E->upper_x - E->lower_x| * |E->upper_y - E->lower_y|;
               CI->value= CI->value + ((pow(W, CI->order - 1) * Area) /
                              N->ent_num);
               increase CI->order by one;  goto step 3;
          else exit;
4.   if N->ent_num < CAP then                              /* N is a leaf node  */
          CI->status= A;                       /* can accept NEW in node N        */
     else CI->status= S;               /* node N need to be split after insertion */
```

Insertion by the RC distribution heuristic consists of seven algorithms, Insert, Insert_by_RC, SelectTree_by_RC, Calc_RC, InsertRect, SelectEntry and Adjust. Algorithm Insert_by_RC invokes algorithm SelectTree_by_RC to select a tree in which the entry NEW can be inserted. Algorithm SelectTree_by_RC creates D processes and each process i (i= 1, 2, ..., D) invokes algorithm Calc_RC for each tree i in the current layer to obtain information (e.g., tree order, relative crowd index value and tree status). Algorithm Calc_RC descends a tree from root node to leaf node by using algorithm SelectEntry. Algorithm Calc_RC is executed in parallel on all disks. Algorithm SelectEntry returns either an entry if that can include entry NEW or NULL if no entry in node N can accept NEW. For each node in the insertion path, except the root node, the index value is calculated and this value is cumulative. If a leaf node can accept NEW, then the tree status is set to either *A* or *S* depending on the number of entries in the node N. For the trees with status either *A* or *S*, algorithm

SelectTree_by_RC selects a tree using four criteria (tree order, tree status, index value, and number of objects in the tree). In the RC distribution heuristic, the tree status has higher precedence order than the index value. When the index value has higher precedence order than the tree status, the PML-tree has better query performance, but space utilization is not high enough. Therefore, the RC distribution heuristic evaluates tree status before the index value to provide high space utilization, although the query performance decreased slightly. If no tree in the current layer accepts NEW, then the current layer is set to the next lower layer and the selection process is repeated.

Figures 22 and 23 (page 53) illustrate insertion by the RC distribution heuristic. In this example, node capacity is 3, the number of disks used is 2, weight constant W is set to 3 and data space is 40 x 40. Figures 22(a) and 22(b) represent the first data space and tree, respectively, before the insertion of data object 5. Figures 23(a) and 23(b) represent the second data space and tree, respectively, before the insertion of data object 5. The values of four criteria after invoking algorithm Calc_RC for two trees in algorithm SelectTree_by_RC are illustrated in Table 2. The CI->values of the first and the second trees are calculated in the algorithm Calc_RC as follow:

$((242 \times 1) / 2) + ((77 \times 3) / 2) = 236.5$

where 242 and 77 denotes sizes of the intermediate rectangles 21 and 11, respectively, in Figure 22(a) and

$((528 \times 1) / 3) + ((231 \times 3) / 2) = 522.5$

where 528 and 231 denotes sizes of the intermediate rectangles 24 and 13, respectively in Figure 23(a).

In this example, the first and the second trees have the same number of entries in leaf nodes. However, average area size per entry of the second tree is bigger than that of the first tree. Therefore, the nodes in insertion path of the second tree is relatively

less crowded than those of the first tree. Algorithm SelectTree_by_RC selects the second tree to insert data object 5.

Tree_id	CI->order	CI->value	CI->status	total_ent
1	3	236.5	A	-
2	3	522.5	A	-

Table 2 contents of structure CI for two trees in algorithm SelectTree_by_RC.

```
Algorithm InsertRect (N, NEW, REM)
[Input]       N- node.  NEW- entry to be inserted.  REM- stores removed entries.
[Output]      re-organized tree after insertion of NEW.
[Comment]     This algorithm is called recursively.  Split node pointer, SP, is returned.
1.   if N is not a leaf node then
         invoke E= SelectEntry (N, NEW);
         if E ≠ NULL then                   /* goto child node pointed to by E   */
             save node pointer N, P= N, and set N= E->P;
             invoke SP= InsertRect (N, NEW, REM) recursively;
             if NEW is inserted in a leaf node then
                 call Adjust (E);
                 if SP ≠ NULL then                      /* if node N is split  */
                     create an entry E' for split node SP at a lower level;
                     insert E' into node P, P->ETY[P->ent_num++]= E';
                     if P->ent_num ≤ CAP then
                         SP= NULL;
                     else invoke SP= Split (P, REM);   /* parent node split */
         else SP= NULL;
2.   insert NEW into N, N->ETY[N->ent_num++]= NEW;
     if N->ent_num ≤ CAP then
         SP= NULL;
     else invoke SP= Split (N, REM) to re-distribute entries in node N into
         two nodes after split;
3.   return SP;
```

Algorithm InsertRect descends a selected tree from root node to leaf node to insert an entry NEW. At each node visited, one entry is selected from the node by algorithm SelectEntry and the algorithm InsertRect invokes itself recursively until the leaf node is reached. If entry NEW is inserted in a leaf node, then algorithm Adjust calculates minimum bounding rectangles of the nodes in the insertion path. In the case

of node overflowing, algorithm Split is called to re-distribute CAP + 1 entries in node N into two nodes, original node N and split node SP.

```
Algorithm SelectEntry (N, NEW)
[Input]       N- node.  NEW- entry to be inserted.
[Output]      E- selected entry pointer.
[Comment]     Select an entry E in the node N.  A child node of entry E can accept
              NEW without violating PML-tree properties.
1.   for each entry E in node N, if more than one entry overlaps NEW then
          return NULL;
2.   if only one entry E overlaps with NEW then
          compute minimum enlargement of E->R to enclose NEW;
          if the enlarged rectangle overlaps with other rectangles in the node then
               return NULL;
          else return E;
3.   if there is no entry overlapping NEW then
          for each entry E, compute minimum enlargement of E->R and check
          whether each enlarged rectangle overlaps other rectangles;
     if there is no entry which can enclose NEW without overlapping other rectangles
     then
          return NULL;
     else select an entry E having minimum enlargement to enclose NEW;
          resolve ties by selecting an entry whose child node has a minimum number
          of entries;  return E;
```

```
Algorithm Adjust (E)
[Input]       E- entry.
[Output]      E- entry with new adjusted coordinates.
[Comment]     Calculate coordinates which completely enclose all rectangles
              associated with entries in the child node, E->P->ETY[j]->R.
1.   for all entries E' (E'= E->P->ETY[j], j= 1, 2, ..., E->P->ent_num) in the child
     node pointed to by E, find maximum and minimum coordinates along each axis;
2.   adjust the coordinates of E->R with those maximum and minimum coordinates;
```

Algorithm SelectEntry searches entries in the node to find if there is an entry which can enclose a new entry NEW and still satisfy PML-tree properties (e.g., disjoint intermediate rectangles and complete enclosure of the data rectangles). Minimum enlargement is the first choice parameter to select an entry. Reducing area

sizes of intermediate rectangles is crucial, since the PML-tree does not allow overlapping intermediate rectangles. Reducing area of intermediate rectangles allows more intermediate rectangles on a data space and this reduces the total number of data spaces of the PML-tree. The number of sub-rectangles in the intermediate rectangle is used as a tie breaker. Selecting an intermediate rectangle with a smaller number of sub-rectangles distributes rectangles evenly over the nodes and reduces the chance of node overflow. This parameter increases space utilization.

For each axis, the algorithm Adjust calculates minimum and maximum coordinates of the rectangles associated with the entries in the child node of E. Then, the coordinates for E->P is updated with these bounding coordinates.

Node Split of the PML-tree

In the case of node overflow during an insertion operation, a splitting process is needed. The PML-tree's split algorithm consists of three major sub-algorithms, Split, SelectSpline and SubSplit. The parameters for split algorithms and their precedence order are proposed as follows:

1. an even distribution of the entries among nodes to improve space utilization,
2. minimization of the number of rectangles intersecting the split line to minimize the number of re-insertions,
3. minimization of the sum of the two intermediate rectangle areas (sub-regions) on both sides of the split line.

In algorithm Split, for each axis, the coordinates of starting and ending points of rectangles associated with entries in an overflowing node are stored in Split_pos[] and sorted. The total number of starting and ending points along each axis in the overflowing node on an axis is 2(CAP + 1), since split algorithms are invoked only when a node has CAP + 1 entries. The balancing factor, Xmin or Ymin, is defined as a

variable and set to M (the initial balancing factor value). Experiments have shown that M set to 30% of the CAP yields the best performance for the PML-tree. The algorithm Split invokes the algorithm SelectSpline.

```
Algorithm Split (N, REM)
[Input]        N- overflowing node with CAP + 1 entries.
               REM- store removed leaf entries.
[Output]       SP- split node pointer.
[Comment]      For each axis, sort lower and upper coordinates of the entries and call
               SelectSpline () to obtain a split position. Then select the split axis and
               split position and invoke SubSplit () to re-distribute CAP + 1 entries.
int  Xmin, Ymin;            /* balancing factors; initially set to M (30% of CAP) */
int  X_ov, Y_ov;            /* number of entries overlapping selected split line */
int  Split_pos[2(CAP + 1)]; /* keeps lower and upper coordinates of the entries */
int  j;                                           /* j= 1, 2, ..., 2(CAP + 1) */
1.   for the horizontal axis,
          for each entry E in N,
               Split_pos[j++]= E->R->lower_x;  Split_pos[j++]= E->R->upper_x;
          sort Split_pos;  set Xmin= M;
          invoke Xpos= SelectSpline (N, Xmin, Split_pos, X_ov);
     for the vertical axis,
          for each entry E in N,
               Split_pos[j++]= E->R->lower_y;  Split_pos[j++]= E->R->upper_y;
          sort Split_pos;  set Ymin= M;
          invoke Ypos= SelectSpline (N, Ymin, Split_pos, Y_ov);
2.   select an axis giving maximum entry distribution, MAX(Xmin, Ymin);
     resolve ties by choosing axis with minimum overlap, MIN(X_ov, Y_ov);
     resolve ties by selecting axis having the smaller area size for two groups after
     split;
3.   if the horizontal axis is selected as the split axis then
          invoke SP= SubSplit(N, Xpos, REM);
     else invoke SP= SubSplit(N, Ypos, REM);
4.   return SP;
```

For each coordinate in Split_pos[j], j= 1, 2, ..., 2(CAP + 1), the algorithm SelectSpline computes the number of entries in overflowing node N whose upper and lower coordinates on given axis are all less than or equal to Split_pos[j] and that number is stored in LO_count[j]. Also, the number of the entries in N whose upper and lower coordinates on an axis are all greater than or equal to Split_pos[j] is

computed and that number is stored in UP_count[j]. The number of entries in N overlapping Split_pos[j] is computed and stored in OV_count[j]. The summation of three values of LO_count[j], UP_count[j] and OV_count[j] is CAP + 1. Therefore, the value of OV_count[j] can be obtained by (CAP + 1) - (LO_count[j] + UP_count[j]). For example, for Split_pos[9] in Table 4(a), OV_count[9] is calculated as follows: 10 - (3 + 4)= 3, where 10= (CAP + 1); 3 and 4 are values of LO_count[9] and UP_count[9], respectively. For each LO_count[j] and UP_count[j] (j= 1, 2, ..., 2(CAP + 1)), the smaller number is stored in Temp[j].

```
Algorithm SelectSpline (N, m, Split_pos, ov)
[Input]        N- overflowing node.
               m- balancing factor; initially set to M (initial balancing factor).
               Split_pos[]- holds sorted coordinates of the entries in N.
               ov- to return number of entries overlapping the selected split position.
[Output]       a split position on an axis.
[Comment]      Select a split position on given axis using three parameters for the
               PML-tree split algorithms.
int  UP_count[2(CAP + 1)], LO_count[2(CAP + 1)], OV_count[2(CAP + 1)],
     Temp[2(CAP + 1)], j;
1.   for each coordinate in Split_pos[j],
          compute the number of entries in N whose coordinates on the given axis are
          less than or equal to Split_pos[j] and store that number in LO_count[j];
          compute the number of entries in N whose coordinates on the given axis are
          greater than or equal to Split_pos[j] and store that number in UP_count[j];
          compute the number of entries in N whose lower coordinates are less than
          or equal to Split_pos[j] and upper coordinates are greater than or equal to
          Split_pos[j] and store that number in OV_count[j];
2.   Temp[j] = MIN(LO_count[j], UP_count[j]), j= 1, 2, ..., 2(CAP + 1);
3.   if there is no Temp[j] ≥ m then          /* set m with maximum distribution */
          m= MAX(Temp[j], j= 1, 2, ..., 2(CAP + 1));
     if m > 0 then
          for all j for which Temp[j] ≥ m, select j with minimum overlap entry count,
          j= {j | MIN(OV_count[j], j= 1, 2, ..., 2(CAP + 1))};  ov= OV_count[j];
          resolve ties by choosing Split_pos[j] with the minimum area for the two
          groups after split;
          return Split_pos[j];
     else return NULL;
```

Balancing factor m is used is to provide balanced entry distribution between nodes N and SP. In the case that there is no Temp[j] whose value is greater than or equal to m, m is set to MAX(Temp[j], j= 1, 2, ..., 2(CAP + 1)). For all j, for which Temp[j] values are greater than or equal to m, the algorithm SelectSpline selects one j for which OV_count[j] has a minimum value. If there is more than one minimum OV_count[j], then resolve ties by choosing j with the minimum summation of areas of two intermediate rectangles associated with nodes N and SP after split by split position, Split_pos[j].

After invoking algorithm SelectSpline for each axis, algorithm Split obtains one split position and the number of entries overlapping selected split position for each axis unless m is 0. If Xmin and Ymin values become 0, then the node capacity should be increased. The node capacity for spatial index structures with the disjoint space decomposition method should be greater than the maximum entry overlap; otherwise, it is impossible to split an overflowing node.

```
Algorithm SubSplit (N, Spos, REM)
[Input]       N- overflowing node with CAP + 1 entries.
              Spos- split position on selected axis.
              REM- stores removed leaf entries.
[Output]      SP- split node pointer.
[Comment]     Create split node SP and re-distribute entries into two nodes. If an
              entry is on the split line and it is an intermediate entry then its child
              node is recursively split by calling SubSplit (). A leaf entry on the split
              line is removed and stored in REM.
1.  create split node SP;  save node pointer, P= N;
2.  for each entry E in P,
    if entry E overlaps with Spos and is a leaf level entry then
         put E into REM->ETY[REM->ent_num++]= E;
    else if E is on the split line and is an intermediate level entry then
         set N= E->P and invoke E'= SubSplit(N, Spos, REM).
         P->ETY[P->ent_num++]= E;  SP->ETY[SP->ent_num++]= E';
    else if coordinates of E->R on the split axis are less than or equal to Spos then
         P->ETY[P->ent_num++]= E;
    else SP->ETY[SP->ent_num++]= E;
```

For example, if the given node capacity is 3 and there are four rectangles (1, 2, 3, 4) with each rectangle completely enclosed by another (1 is enclosed by 2, 2 is enclosed by 3, 3 is enclosed by 4), then no split line can be found under the given node capacity. Algorithm Split selects a split axis with a bigger balancing factor value to provide even entry distribution. If Xmin and Ymin are the same, then an axis with minimum overlap value, MIN (X_ov, Y_ov), is selected. Minimum area for two groups after the split is used to resolve ties. After selecting the axis and split position, algorithm Split invokes algorithm SubSplit.

Algorithm SubSplit actually distributes CAP + 1 entries into two nodes (original overflowing node N and newly created split node SP) using the selected split position, Spos, on the split axis. A leaf level entry which has coordinates overlapping split position Spos is removed and stored in REM for re-insertion. In Chapter IV, the number of re-inserted entries for the four test data sets are illustrated. An intermediate level entry which has coordinates overlapping split position Spos is sub-divided by calling SubSplit recursively with the selected split axis and split position. An entry whose coordinates are less than or equal to the split position Spos is stored in the original node N. An entry whose coordinates are greater than or equal to the split position Spos is stored in the split node SP. A split node pointer SP is returned.

An example of a node split is shown in Figure 25 (page 65). The node capacity in this example is 9 and M is set to 3. The starting and ending coordinates of the rectangles for each axis are listed with their object rectangle IDs in Table 3. In the algorithm Split, starting and ending coordinates of the rectangles are sorted into Split_pos[] for each axis as shown in the second columns of Tables 4(a) and 4(b), respectively. For each Split_pos[j], the algorithm SelectSpline computes number of entries for LO_count[j] and UP_count[j]. Then, the minimum value of LO_count[j] and UP_count[j] is stored in Temp[j]. The algorithm SelectSpline selects a split position using three criteria: balancing factor m, OV_count and area size. First, the

algorithm SelectSpline selects all possible split positions which have a corresponding Temp[j] greater than or equal to m.

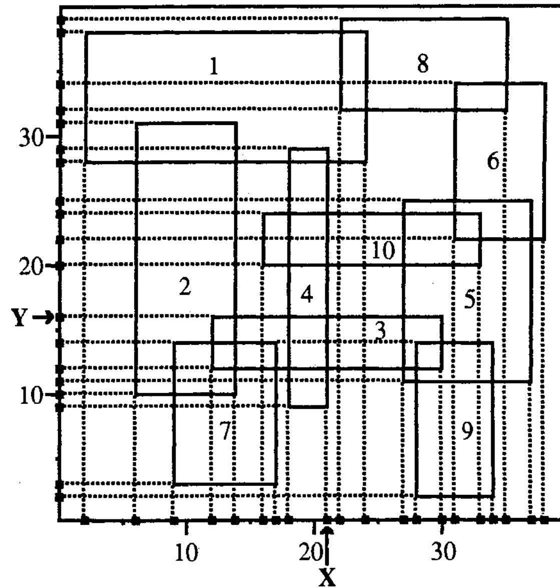


Figure 25 split of an overflowing node.

object_id	lower_x	lower_y	upper_x	upper_y
1	2	28	24	38
2	6	10	14	31
3	12	12	30	16
4	18	9	21	29
5	27	11	37	25
6	31	22	38	34
7	9	3	17	14
8	22	32	35	39
9	28	2	34	14
10	16	20	33	24

Table 3 coordinates of the ten data objects used in Figure 25.

Therefore, four possible split positions for the horizontal axis and three possible split positions for the vertical axis are selected. Those selections are marked with shaded area as shown in the 5th columns of Tables 4(a) and 4(b). In Table 4(a), the values of OV_count[] for all the selected possible split positions are the same. Therefore, the area for two groups after the split for each Split_pos[] selected is considered. The

values in the 7th columns of Table 4(a) and 4(b) are the sum of the areas of the two minimum bounding intermediate rectangles on both sides of the split position. For example, area_size value corresponding to Split_pos[9] in Table 4(a) is the sum of the minimized areas of the two intermediate rectangles that enclose the group of object rectangles 2, 4, 7 and another group of object rectangles 5, 6, 8, 9, respectively. The sum of the minimized areas equals 1012. The algorithm SelectSpline chooses Split_pos[9] as the final split position for the horizontal axis.

A similar process is carried out for the vertical axis. The final split position Split_pos[9] is selected for the vertical axis by the algorithm SelectSpline. Then, the algorithm Split chooses one split position from Split_pos[9]= 21 (horizontal axis) and Split_pos[9]= 16 (vertical axis) as the final split position for the split process by using the three parameters specified. In this example, the split position Split_pos[9]= 16 (vertical axis) is selected as the final split position, since the summation of minimum areas of the two bounding intermediate rectangles (sub-regions) on two sides of the split position on the vertical axis is smaller than that on the horizontal axis. Consequently, the ten entries are distributed into two nodes, N= {3, 7, 9} and SP= {1, 6, 8, 10}. If rectangles 2, 4 and 5, intersect the split line, represent leaf level entries, then they are removed from this space and re-inserted into one of the available data spaces or a new data space in the PML-tree. If rectangles 2, 4 and 5 represent intermediate entries, then each of them are sub-divided by calling algorithm SubSplit recursively. Figure 26 shows the resulting original data space after the split.

j	Split_pos[j]	LO_count[j]	UP_count[j]	Temp[j]	OV_count[j]	area_size
1	2	0	10	0	0	
2	6	0	9	0	1	
3	9	0	8	0	2	
4	12	0	7	0	3	
5	14	1	6	1	3	
6	16	1	6	1	3	
7	17	2	5	2	3	
8	18	2	5	2	3	
9	21	3	4	3	3	1012
10	22	3	4	3	3	1012
11	24	4	3	3	3	1122
12	27	4	3	3	3	1122
13	28	4	2	2	4	
14	30	5	1	1	4	
15	31	5	1	1	4	
16	33	6	0	0	4	
17	34	7	0	0	3	
18	35	8	0	0	2	
19	37	9	0	0	1	
20	38	10	0	0	0	

(a)

j	Split_pos[j]	LO_count[j]	UP_count[j]	Temp[j]	OV_count[j]	area_size
1	2	0	10	0	0	
2	3	0	9	0	1	
3	9	0	8	0	2	
4	10	0	7	0	3	
5	11	0	6	0	4	
6	12	0	5	0	5	
7	14	2	4	2	4	
8	14	2	4	2	4	
9	16	3	4	3	3	1009
10	20	3	4	3	3	1009
11	22	3	3	3	4	
12	24	4	2	2	4	
13	25	5	2	2	3	
14	28	5	2	2	3	
15	29	6	1	1	3	
16	31	7	1	1	2	
17	32	7	1	1	2	
18	34	8	0	0	2	
19	38	9	0	0	1	
20	39	10	0	0	0	

(b)

Table 4 list of parameters used for the PML-tree split algorithms for each of (a) horizontal axis and(b) vertical axis.

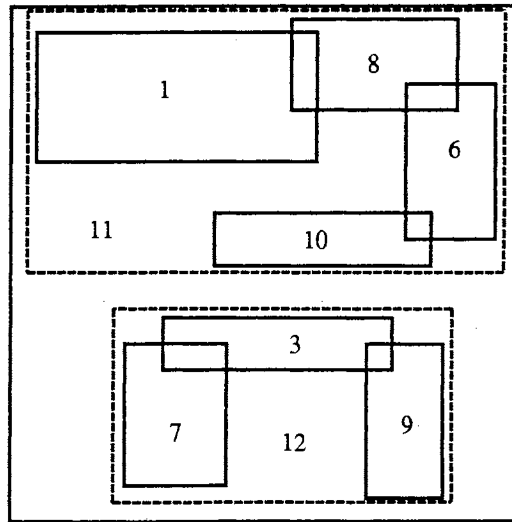


Figure 26 organizations of bounding rectangles (the solid lines construct object rectangles; the dash lines construct intermediate rectangles) using the PML-tree with node capacity 9.

<u>Search of the PML-tree</u>

Search operations consist of two algorithms, Search and SearchTree. The algorithm Search creates D processes, where D is the number of trees in a layer or number of disks used in the PML-tree. Each process i invokes algorithm SearchTree using the root node of the ith tree in the current layer, in parallel. The algorithm SearchTree traverses the tree recursively and finds all data objects overlapping the search range. Since all the nodes in a tree of the PML-tree are in the same disk, there is no need of inter-process communications. On the other hand, the MXR-tree search algorithm needs inter-process communications among the D processes whenever each process finds an intermediate entry overlapping the search range. Implementational details of search algorithms for the PML-tree and MXR-tree are discussed in Chapter IV.

```
Algorithm Search (W)
[Input]      W- structure RECT; search range.
[Output]     all data objects overlapping W.
[Comment]    For each layer, create D processes and invoke SearchTree ().
1.   set LAY= LAY_LIST->next;
2.   create D child processes;
3.   if LAY ≠ NULL then
          each child process i (i= 1, 2, ..., D) invokes
          SearchTree (LAY->TREES[i]->P, W) in parallel;
          LAY= LAY->next;  goto step 3;
     exit;
```

```
Algorithm SearchTree (N, W)
[Input]      N- node.  W- structure RECT; search range.
[Output]     data objects overlapping W.
[Comment]    Traverse tree recursively and save leaf entries overlapping W.
1.   save node pointer, P= N;
2.   if P is not a leaf node then, for each entry E in P,
          if E->R overlaps W then
               set N= E->P and invoke SearchTree (N, W) recursively;
3.   if P is a leaf node then for each entry E in P,
          if E->R overlaps W then                     /* qualifying entries */
               save entry pointer;
```

Deletion of the PML-tree

Deletion operations consist of three algorithms, Delete, DeleteTree and Adjust. The algorithm Delete creates D processes, where D is the number of trees in a layer or number of disks used in the PML-tree. Each process i, where i= 1, 2, ..., D, invokes algorithm DeleteTree using the root node of the ith tree in the current layer, in parallel. The algorithm DeleteTree traverses the tree recursively and removes all the data objects overlapping the search range. If there is a removed leaf level entry, then all the entries in the deletion search path need to adjust bounding rectangles associated with them. If any node becomes empty, then the parent entry for that node is

removed. An empty root node results in removal of the tree from the current layer of the PML-tree. As in search operations, the PML-tree does not need inter-process communications.

```
Algorithm Delete (W)
[Input]       W- structure RECT; delete range.
[Output]      re-structured PML-tree.
[Comment]     For each layer, create D processes and invoke DeleteTree ().
int  Flag;           /* set to TRUE if there is at least one leaf entry removed */
1.   set LAY= LAY_LIST->next;  set Flag= False;
2.   create D child processes;
3.   if LAY ≠ NULL then
          in each child process i (i= 1, 2, ..., D),
          if LAY->TREES[i] ≠ NULL then                     /* if not empty tree */
               invoke DeleteTree (LAY->TREES[i]->P, W, Flag, LAY->TREES[i]);
               if LAY->TREES[i]->P->ent_num is 0 then        /*root node empty */
                    set LAY->TREES[i]->P= NULL;     /* remove tree from LAY */
          set LAY= LAY->next;  goto step 3;
     exit;
```

```
Algorithm DeleteTree (N, W, Flag, Tree)
[Input]       N- node. W- structure RECT; delete range. Flag- integer.
              Tree- structure TREE_INFO.
[Output]      re-structured PML-tree.
[Comment]     Traverse tree recursively. Remove leaf entries overlapping W.
1.   save node pointer, P= N;
2.   if P is not a leaf node then for each entry E in P,
          if E->R overlaps W then
               set N= E->P and invoke DeleteTree (N, W, Flag, Tree) recursively;
               if Flag then         /* there exists at least one removed leaf entry */
                    if N->ent_num > 0 then                  /* not empty node */
                         invoke Adjust (E) to re-adjust boundary of E;
                    else remove E from P;  P->ent_num--;
3.   if P is a leaf node then for each entry E in P,
          if E->R overlaps W then
               remove entry E from P;  P->ent_num--;  Tree->total_ent--;
               and set Flag= True;
```

In the PML-tree, all object rectangles associated with the entries in a node are completely enclosed by an upper level intermediate rectangle associated with an entry pointing to that node. Therefore, deletion algorithm of the PML-tree does not need to take multiple deletion paths to delete an overlapping object as in the R^{+}-tree.

CHAPTER IV

PERFORMANCE ANALYSES OF PARALLEL SPATIAL INDEX STRUCTURES

Parallel spatial index structures can improve query performance significantly by introducing parallelism in I/O operations with multiple disks. In serial spatial index structures (e.g., R-tree, R^+-tree, R^*-tree, ..., etc.), the performance of query operations highly depends on the total number of disk accesses. However, in the parallel spatial index structures, the performance of query operations depends on the maximum number of disk accesses among the disks. Therefore, the distributions of data objects among the disks is important in parallel spatial index structures. An efficient distribution heuristic for the parallel spatial index structure should distribute data objects over the disks evenly in terms of the number of objects and spatial distribution to obtain maximum performance. The MXR-tree uses the proximity index (PI) distribution heuristic and the PML-tree proposes three distribution heuristics: minimum number of entries (ME); absolute crowd index (AC); relative crowd index (RC).

In this chapter, the performances of parallel spatial index structures are compared using various test data sets: two system generated data sets and two real application data sets. The structures compared are the MXR-tree with the PI distribution heuristic and the PML-tree with three distribution heuristics. The MXR-tree was chosen, since this is the only parallel spatial index structure known to the author. The MXR-tree uses a native space approach and a non-disjoint space decomposition method. The MXR-tree has good space utilization, as in original R-tree, and this factor makes range queries more efficient as the query size increases. The performance of serial R-tree is illustrated in Chapter V. Numbers of disk accesses

and the actual response times to process a range query are given for the four parallel spatial index structures: one MXR-tree and three PML-trees with three different object distribution heuristics. The relative performance gains of the PML-trees over the MXR-tree are also given. Construction speed, space utilization and actual memory size of the four index structures are compared.

Implementations of the Spatial Index Structures

In Chapter III, three different object distribution heuristics were proposed for the PML-tree. For each object distribution heuristic, the PML-tree is implemented based on the algorithms proposed in Chapter III. Let PME-tree, PAC-tree and PRC-tree represent the PML-tree with the three object distribution heuristics, ME, AC and RC, respectively. In insertion operation, the PME-tree does not need to create D processes to calculate the index values, since the ME distribution heuristic simply considers the number of data objects in each tree to select a tree. However, the PAC-tree and PRC-tree create D (number of disks used) processes, respectively, to calculate heuristic index values, and each of the D processes descends the associated tree from root node to leaf node simultaneously to calculate heuristic index values. In this experiment, algorithm Calc_AC for PAC-tree and algorithm Calc_RC for PRC-tree are implemented in parallel. In search or deletion operations of the PME-tree, PAC-tree and PRC-tree, D processes are created. Each process searches or deletes data objects overlapping the given search or deletion range from the tree(s) on an associated disk simultaneously. No communication is required among the processes, since all nodes in a tree are stored on the same disk. Therefore, implementation of search and deletion algorithms for the PML-trees are simple.

The MXR-tree with the proximity index heuristic is implemented based on the functions proposed in [51]. The reason that the proximity index was selected as a

heuristic for the MXR-tree is that it has the best performance among the four proposed heuristics in [51]; in Chapter III, the proximity index heuristic is briefly described. The MXR-tree has the same basic structure as that of the R-tree presented by Güttman [40]. The quadratic split algorithm is used for the MXR-tree implementation, since the quadratic splitting algorithm has better performance than the linear splitting algorithm does. The minimum node fill factor of the MXR-tree is set to 40% of the node capacity to result in the best splitting performance.

Insertion operations of the MXR-tree are performed in serial manner. In search or deletion operation, the MXR-tree also creates D processes to access D disks in parallel. However, as mentioned in Chapter III, the MXR-tree needs inter-process communications whenever any of the D processes finds an intermediate entry overlapping the search or deletion range. For example, assume that process i is associated with disk i, where i= 1, 2, ..., D. If process i finds that one intermediate entry in currently uploaded node overlaps the search or deletion range and the child node of this entry is on disk j (j= 1, 2, ..., D), then process i has to put a child node pointer in the queue associated with process j for later disk access. The search and deletion implementations of the MXR-tree use semaphore operations to provide mutual exclusion among the D processes, since at any moment there should be only one process inserting a node pointer in a queue. The time for inter-process communications increases as the number of disks used for the MXR-tree increases. Also, all the queues have to be shared memory. Of course, the MXR-tree can use breadth first search (BFS) method to avoid inter-process communications. In BFS of the MXR-tree, at each level of the tree, all nodes whose parent entries overlap search or deletion range are uploaded by the D processes in parallel, and all uploaded nodes are processed in serial. However, the BFS method takes even more time than the inter-process communication method. All entries and nodes in the MXR-tree need extra memory space for disk IDs and this space reduces the capacity of the node.

In this experiment, buffering was not used. That is, nodes pointed to in insertion, search and deletion operations have to be uploaded from disks to main memory using disk ID, tree ID and node offset. The offsets for empty nodes are stored in a list and assigned to the newly created node(s). After completion of operations, all the nodes updated or newly created are downloaded from main memory to disks.

All programs are written in C language and tested with two-dimensional test data sets. Characteristics of the test data sets used in the experiments are discussed in the following.

Test Data Sets and Types of Queries

Two types of data, system generated data (uniformly distributed data and randomly distributed data) and real application data (Tiger/Line™ files of the TIGER (geographic information system) database and VLSI layout data generated by the Magic system), are used with the proposed parallel spatial index structures and the MXR-tree. Let V, T, R and U represent the VLSI data set, the Tiger/Line™ data set, the randomly distributed data set and the uniformly distributed data set, respectively. The test data sets can be characterized by four data description factors, the number of objects, object density, range of object sizes and data space. The object density or the average data density is defined as:

$$\text{data density (d)} = \frac{\text{summation of space area occupied by data objects}}{\text{total data space area}}$$

If an area of the data space is occupied by two overlapping data objects, then that area is counted twice in the summation of space area occupied by data objects. The range of object sizes is the range of the ratio of object sizes to the data space. Table 5 illustrates data description factors for the four data sets.

data type	object number	density	object size range	data space
V	4085	0.34	0.00017 - 0.45249%	1320 x 1768
T	41058	0.25	0.00000 - 0.34798%	533552 x 349200
R	30000	4.35	0.00250 - 0.14000%	3000 x 3000
U	30000	5.38	0.00111 - 0.04000%	3000 x 3000

Table 5 data description factors of the four data sets.

System generated data sets are easily obtained and data description factors can be easily changed. These data sets are useful to test performances of the spatial index structures under the various conditions. The data set R randomly distributes two-dimensional data objects in virtual data space by using the coordinates generated with a random number generator. The data set U contains six square sets of different sizes as spatial objects. For uniform distribution, center points of squares are predetermined according to the number of squares and the coordinates of the squares are calculated.

Experiments with real application data sets have a credibility that simulation with system generated data frequently lacks [64]. The data set V is generated by VLSI layout systems (Magic system). VLSI design layouts consist of a large number of geometries that describe the layout of transistors and signal wires that inter-connect them on a number of mask layers [2]. Since most VLSI design layouts use manhattan or rectilinear geometries, all design objects (e.g., transistors and wires) are represented as rectangles parallel to the horizontal and vertical axes. The primitive geometry operations in VLSI design layouts can be defined as follows [2].

1. Exact match: Given the coordinates (lower_x, lower_y, upper_x, upper_y), find rectangles that have exactly the same coordinates.
2. Point search: Given the coordinates of a point, (x, y), find all rectangles that contain a that point.
3. Overlap detection: Given a rectangular search area, find all rectangles that overlap that area.

4. Abutting search: Given a rectangular search area, find all rectangles that abut the boundary of the area.
5. Containment search: Given a rectangular search area, find all rectangles that are completely contained within that area.
6. Upper/lower rectangle search: Given a vertical axis value, find all rectangles that are completely contained in the upper/lower half of that line.
7. Left/right rectangle search: Given a horizontal axis value, find all rectangles that are completely contained in the left/right half of that line.
8. Nearest neighbor search: Given a rectangular search area, find the nearest rectangle(s) to that area.

All the operations described above are basically the range searches. For example, if the data space is 100 x 100 and a horizontal axis value for operation 7 is set to 50, then the operation 7 is exactly the same as the range search with coordinates (0, 0 , 50, 100) or (50, 0, 100, 100). Operation 8 is a sequence of one or more range searches. First, a range search is performed for the area bigger than given search area (e.g., average object size can be used as an initial area size for range search). If at least one rectangle is found, then determine the nearest neighbor from the rectangle(s) found. If no rectangle is found, then double the search area and repeat the range search until at least one rectangle is found.

The data set T is from the Tiger/Line™ files of the TIGER database system. The TIGER database system provides information that describes the points, lines and areas on Census Bureau maps. Information related to Tiger/Line™ files are given in Appendix. Data set T is prepared from the file (file ID: TGR40109.F51) for Oklahoma City County in Oklahoma. Even though average density of this data set is very low (about 0.25), some areas have very high densities (e.g., over 60). Typical query operations used in this application are described in [43] as follows.

1. Given an end point of a line segment, find all the line segments that are incident to it (all line segments having the same end points).
2. Given an end point of a line segment, find all the line segments that are incident at the other end point of the line segment.
3. Given a point in the two-dimensional data space, find all the nearest line segments.
4. Given a point in the two-dimensional data space, find the minimal enclosing polygon by outputting its constituent line segments.
5. Given a two-dimensional rectangular search area, find all line segments overlapping that area.

As in VLSI application, all the query operations are simple variations of a range query. Query 1 and 2 are simple point queries. A point query is a special case of a range query. Queries 3, 4 and 5 are variations of range queries. For example, query 4 can be executed by performing query 3 and query 2 recursively. First, query 3 finds the nearest line segment, say segment A, to the given point. Second, starting from the line segment A, query 2 finds the a line segment, say segment B, that is incident at the other end of the line segment A and has exactly the same polygon identification number (see Appendix) as the line segment A. The second step is repeated until the line segment found by query 2 is the first line segment found by query 3; then, all the line segments found construct the minimal enclosing polygon.

All the query operations in VLSI and Tiger/Line™ applications are simple variations of a range query. Therefore, range queries can be used to measure performances of spatial index structures.

Experiment Results

All the programs are run on a *Sequent Symmetry S81* with twenty-four 80386 processors running at 20 Mhz each. In this experiment, page size is 1 Kbytes and the number of disks used for parallel I/O is four.

In the PME-tree, PAC-tree and PRC-tree, an entry size is 35 bytes (e.g., 7 bytes for each of four coordinates and one child node pointer). Node capacity is 58 (2 pages) for test data sets T, R and U and 29 (1 page) for the test data set V, since the number of data objects in data set V is 7 to 10 times smaller than data sets R, U and T. In the MXR-tree, an entry size is 37 bytes, since each entry needs 2 bytes for the disk ID. Node capacity of the MXR-tree for the data sets T, R and U is 54 (2 pages). For data set V, node capacity of the MXR-tree is 27 (1 page).

A query rectangle represents the area in a data space that has to be searched for processing a range query (a point query is included in the category of range query, with zero area). Range of the query rectangle's area is 0% to 12% of the data space. For each search range, e.g., 4% of a data space, 500 search regions are randomly selected and searched. The corresponding average number of disk accesses and the response time for each search range are presented.

Each of four Figures 27, 28, 29 and 30 illustrates the average number of total disk accesses for the four index structures vs. the size of the query for the four data sets V, T, R and U. In Figures 27 through 38, n and d denote the number of objects and the average density of the data set, respectively. The PAC-tree and PRC-tree require fewer disk accesses than the MXR-tree for most of search ranges (e.g., a query range > 2%). Even for the small search ranges (a query range ≤ 2%), the PAC-tree and PRC-tree have fewer overlapping leaf nodes than the MXR-tree. However, intermediate nodes along the search paths of the multiple trees increase the total number of disk accesses of the PML-trees. As the size of search range increases, the

PML-trees reduce the ratio of intermediate nodes to the total number of nodes overlapping the search range. The PME-tree requires more disk accesses than the MXR-tree for data sets V, T and R. The PAC-tree requires the smallest number of disk accesses among the three PML-trees, except for data set V. The structural drawback of the MXR-tree is redundant search path problem, as mentioned in Chapter II, and it increases the number of nodes involved in redundant search paths as the search range increases.

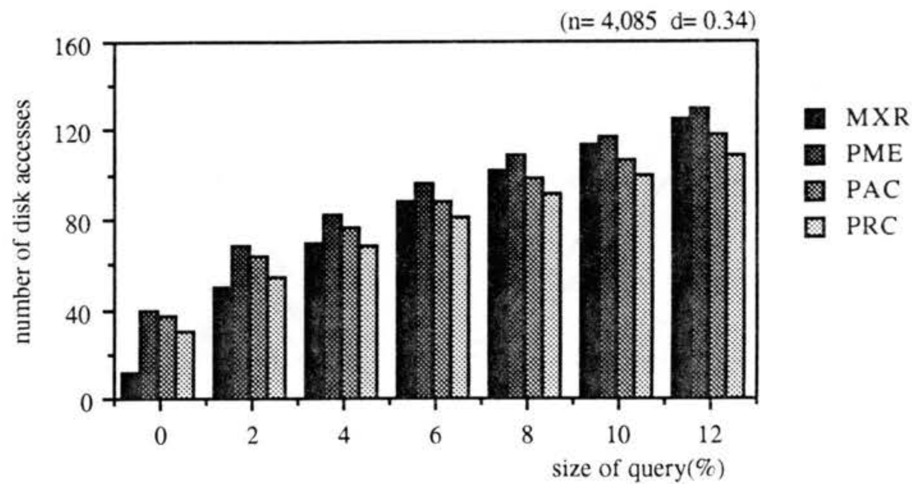


Figure 27 average numbers of disk accesses of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the size of query using data set V.

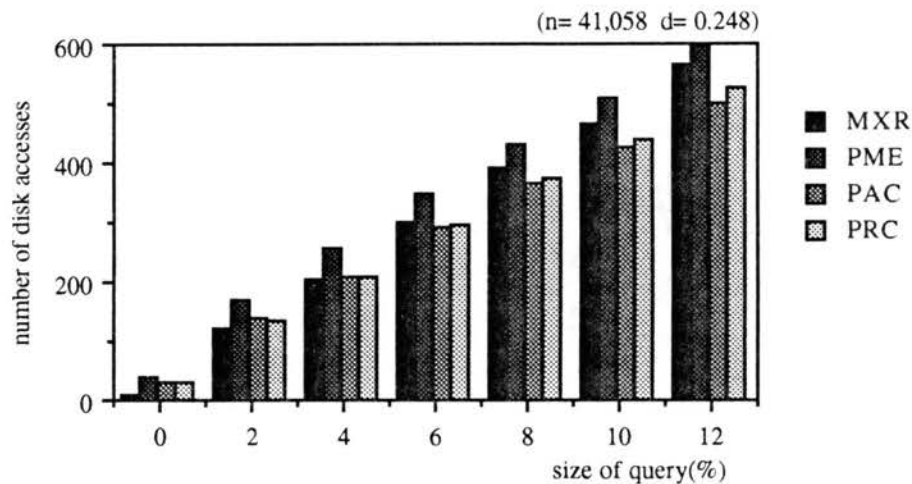


Figure 28 average numbers of disk accesses of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the size of query using data set T.

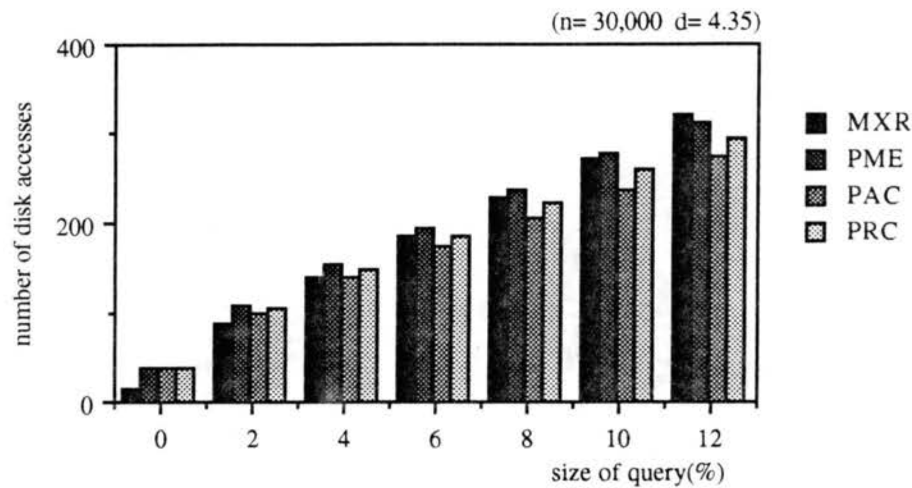


Figure 29 average numbers of disk accesses of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the size of query using data set R.

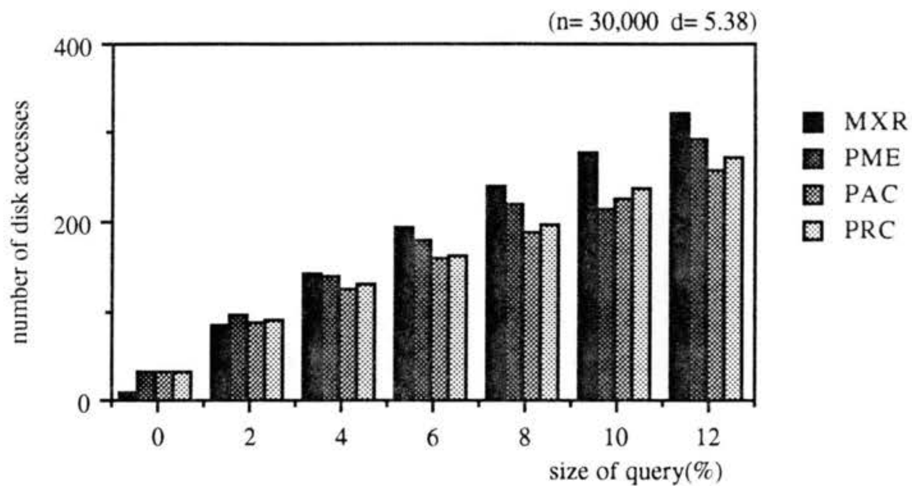


Figure 30 average numbers of disk accesses of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the size of query using data set U.

Each of four Figures 31, 32, 33 and 34 illustrates the average query response time for the four index structures vs. the size of the query for the four data sets V, T, R and U. In serial spatial index structures (e.g., R-tree, R^+-tree and R^*-tree), the query performance of the structures is approximately proportional to the total number of disk accesses to process a given query. Time to process nodes uploaded into main memory does not make a significant differences in the query performance of the spatial

index structures, since one node access time from secondary storage takes much longer than one node processing time in main memory. In the parallel spatial index structures, query performance of the structures is not exactly proportional to the total number of disk accesses for the given search range. For example, let's assume that four disks are used for each of the two parallel spatial index structures A and B In query processing for a certain search range, the number of disk accesses of structure A is 10, 10, 10 and 30 for each disk, respectively, and the number of disk accesses of structure B is 20, 20, 20 and 20 for each disk, respectively. The maximum number of disk accesses of the structures A and B are 30 and 20, respectively. Therefore, structure B is faster than structure A in response time although the structure B has more disk accesses than the structure A. Query processing time in parallel spatial index structures is proportional to the maximum number of disk accesses among the disks used. The PRC-tree takes more disk accesses than the PAC-tree (see Figure 28) for search ranges greater than 2%. However, the response times of the PRC-tree are faster than the PAC-tree, in Figure 32, throughout the search range. In range queries with very small search sizes (e.g., point query), the effect of parallel disk accessing for the MXR-tree is comparatively small. Since the MXR-tree has a single tree structure, the nodes in the search path most likely have to be accessed sequentially. For example, in Figure 30, the MXR-tree requires fewer disk accesses than the PML-trees for point query, but in Figure 34, the MXR-tree has a slower response time than the PML-trees. Another factor that affects the query performance of the parallel spatial index structures is the search strategy. As mentioned in Implementations of the Spatial Index Structures, inter-process communications of the MXR-tree further slow down query processing. On the other hand, the PML-trees do not need inter-process communications in search or deletion operations.

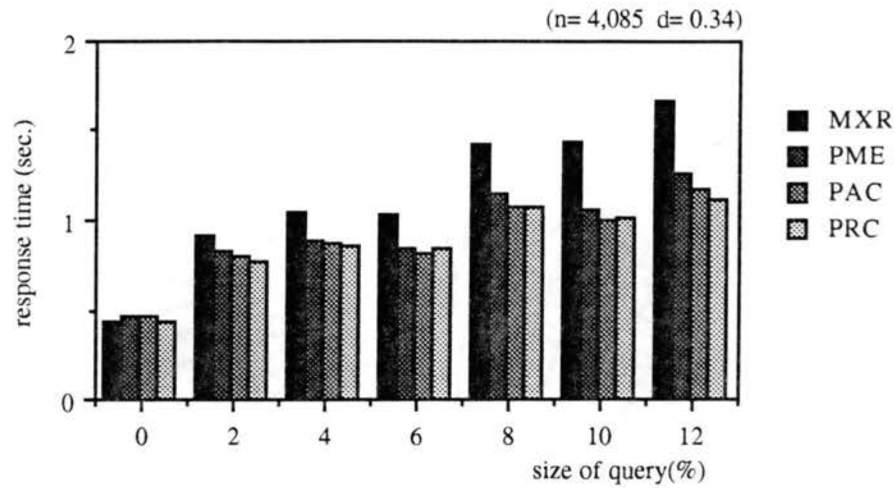


Figure 31 average response time of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the size of query using data set V.

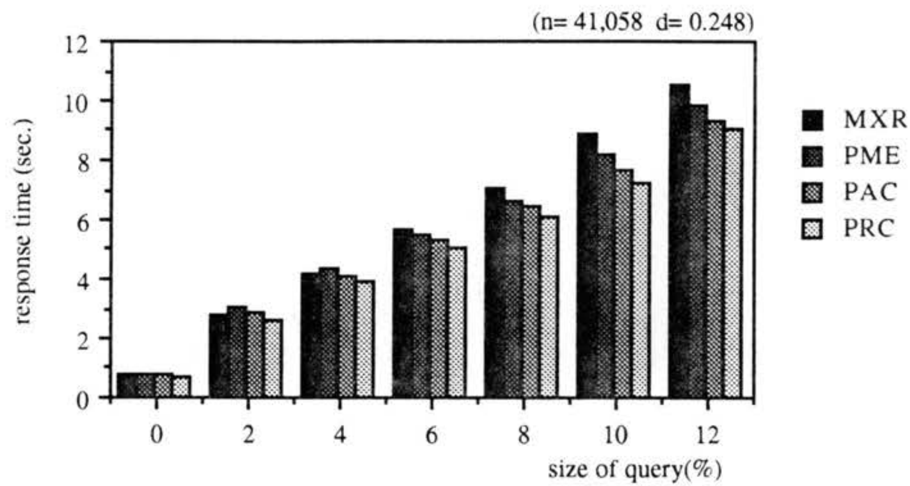


Figure 32 average response time of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the size of query using data set T.

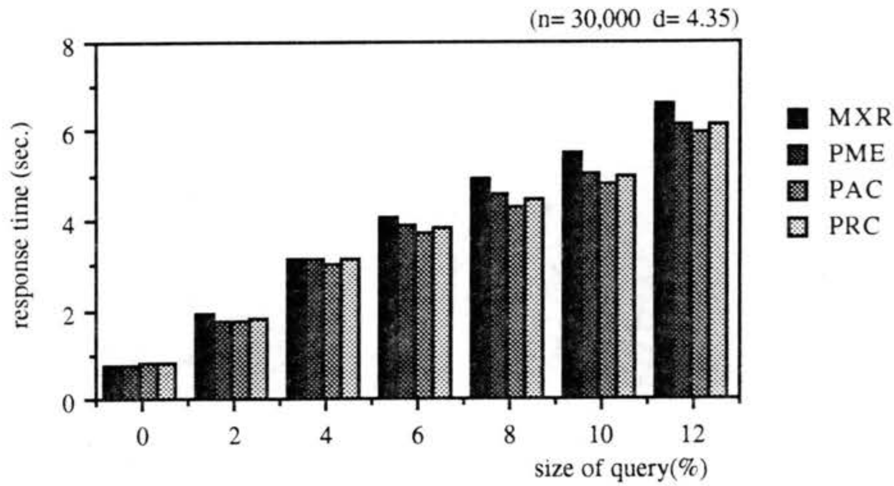


Figure 33 average response time of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the size of query using data set R.

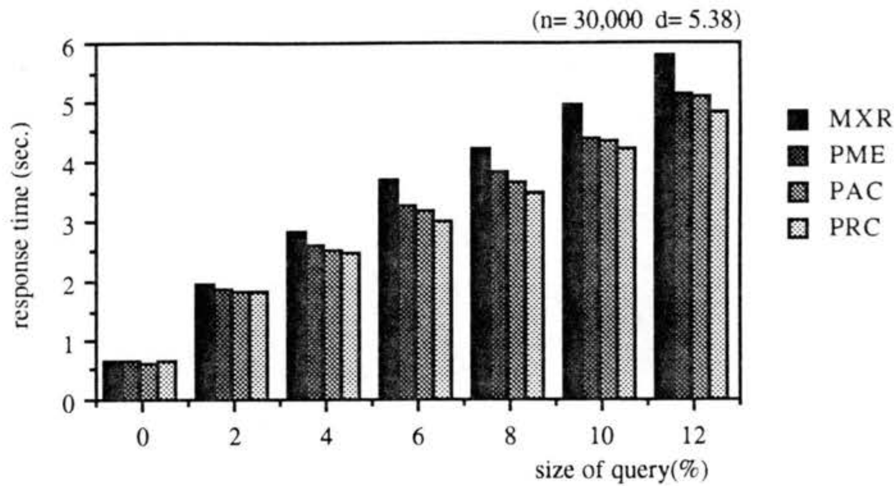


Figure 34 average response time of the MXR-tree, PME-tree, PAC-tree and PRC-tree vs. the size of query using data set U.

Let RT_MXR, RT_PME, RT_PAC and RT_PRC denote response times of the MXR-tree, PME-tree, PAC-tree and PRC-tree, respectively. Also, let PME/MXR, PAC/MXR and PRC/MXR denote the performance gains of the PME-tree, PAC-tree and PRC-tree over the MXR-tree, respectively. Figures 35, 36, 37 and 38 illustrate performance gains of the three PML-trees over the MXR-tree with the four data sets. The performance gain PME/MXR can be defined as:

$$\text{performance gain PME/MXR (\%)} = \frac{RT_MXR - RT_PME}{RT_PME} \times 100$$

The PAC/MXR and PRC/MXR are defined correspondingly.

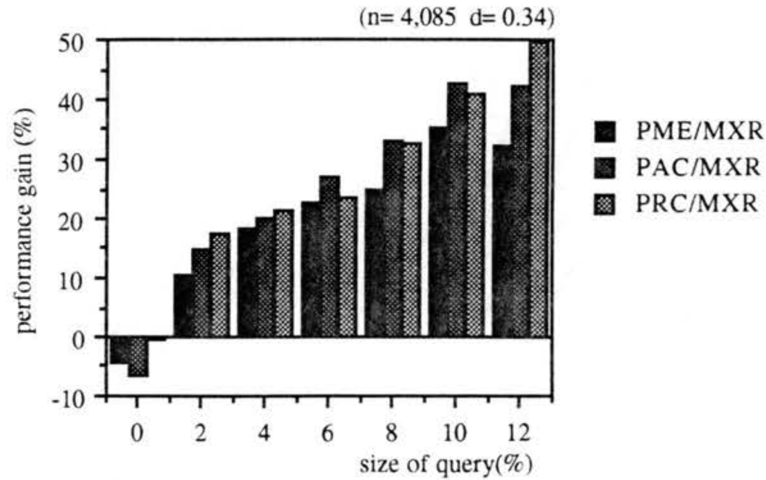


Figure 35 performance gains of the PME-tree, PAC-tree and PRC-tree over the MXR-tree vs. the size of query using data set V.

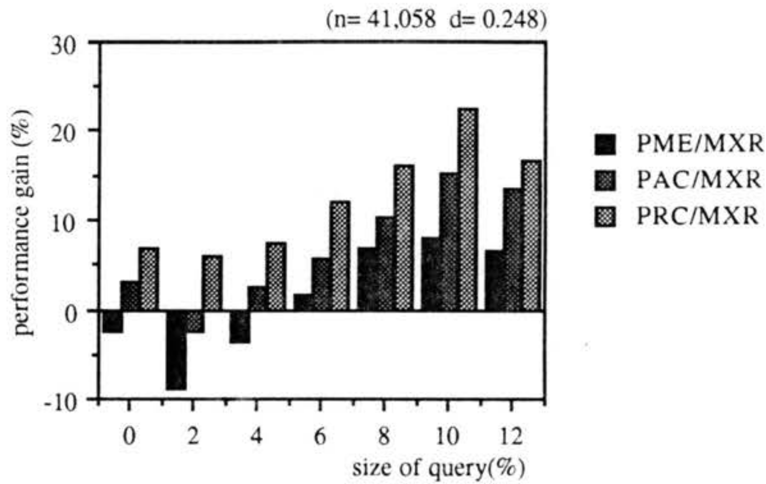


Figure 36 performance gains of the PME-tree, PAC-tree and PRC-tree over the MXR-tree vs. the size of query using data set T.

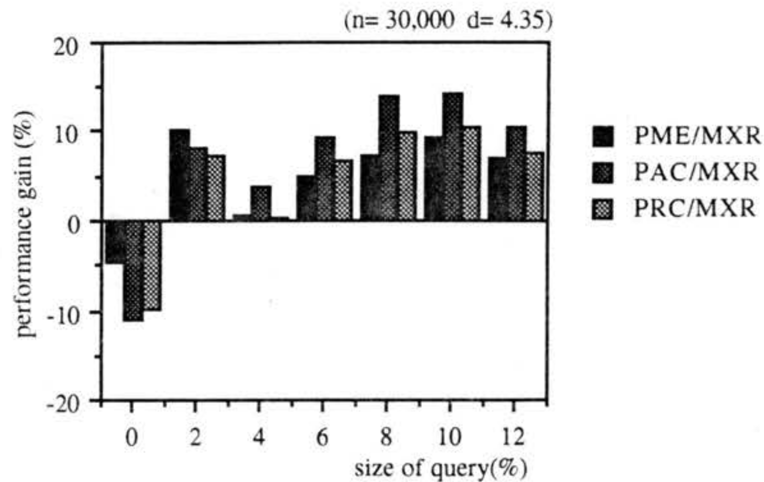


Figure 37 performance gains of the PME-tree, PAC-tree and PRC-tree over the MXR-tree vs. the size of query using data set R.

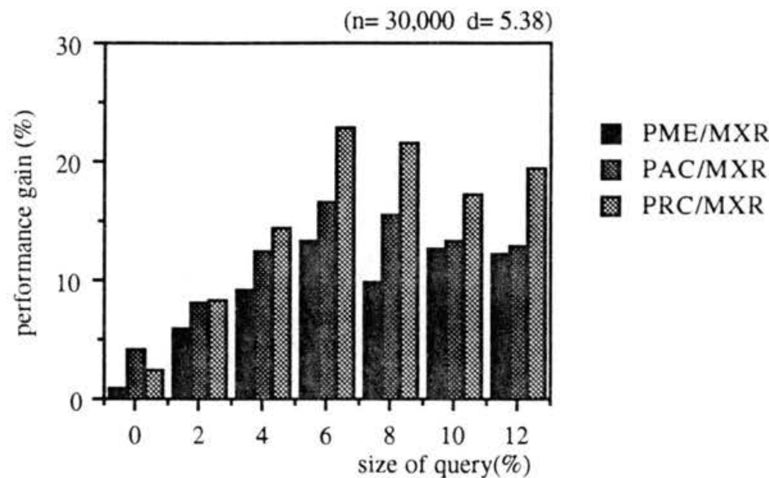


Figure 38 performance gains of the PME-tree, PAC-tree and PRC-tree over the MXR-tree vs. the size of query using data set U.

All three PML-trees outperform the MXR-tree throughout the search ranges, except point query with data sets V and R (see Figures 35 and 37). With data set U, all three PML-trees have better query performance than MXR-tree throughout all search ranges (see Figure 38). The MXR-tree has the lowest query performance for data set U, since the split algorithms of the MXR-tree are not efficient when uniformly distributed data is used. The MXR-tree uses the same split algorithms as the R-tree.

In a node split, the split algorithm of the MXR-tree selects a pair of entries that would waste the most area if both were put in the same node. Those two entries are the seed entries for two nodes, the original node and the split node. Then, the split algorithm selects one entry at a time giving the greatest preference for one of the nodes from the remaining entries and inserts the selected entry into one of the nodes [40]. With this split algorithm, the two nodes can severely or completely overlap after the node split for uniformly distributed data objects. For example, in Figure 39, assume that node capacity is 15. Insertion of entry 16 makes the node overflow. Let *A* and *B* in Figures 39(a) and 39(b) denote intermediate rectangles associated with the nodes *X* and *Y* in Figures 39(c) and 39(d), respectively. To split the overflowing node, the split algorithm of the MXR-tree selects entries 1 and 16 as the seed entries for nodes *X* and *Y*, respectively, as shown in Figures 39(a) and 39(c). In this example, entries 4, 7, 10 and 13 give the same preferences to nodes *X* and *Y* and each of these entries can be put in either *X* or *Y*. Figure 39(a) shows one of the possible distributions. Intermediate rectangles *A* and *B* overlap heavily after the node split. In some case, one of the intermediate rectangles can be completely enclosed by the another intermediate rectangle after the node split. Increased overlapping areas among the intermediate rectangles in the MXR-tree increase the number of disk accesses and as a result query performance is reduced. On the other hand, the PML-trees have disjoint intermediate rectangles after the node split as shown in Figures 39(b) and 39(d), since split algorithms of the PML-tree split an overflowing intermediate rectangle using a selected split line on one of the axes. With data set T, the PAC-tree and PRC-tree have better query performances than the MXR-tree throughout all search ranges, and the PME-tree has better query performance than the MXR-tree for the range queries with search sizes greater than 4% (see Figure 36).

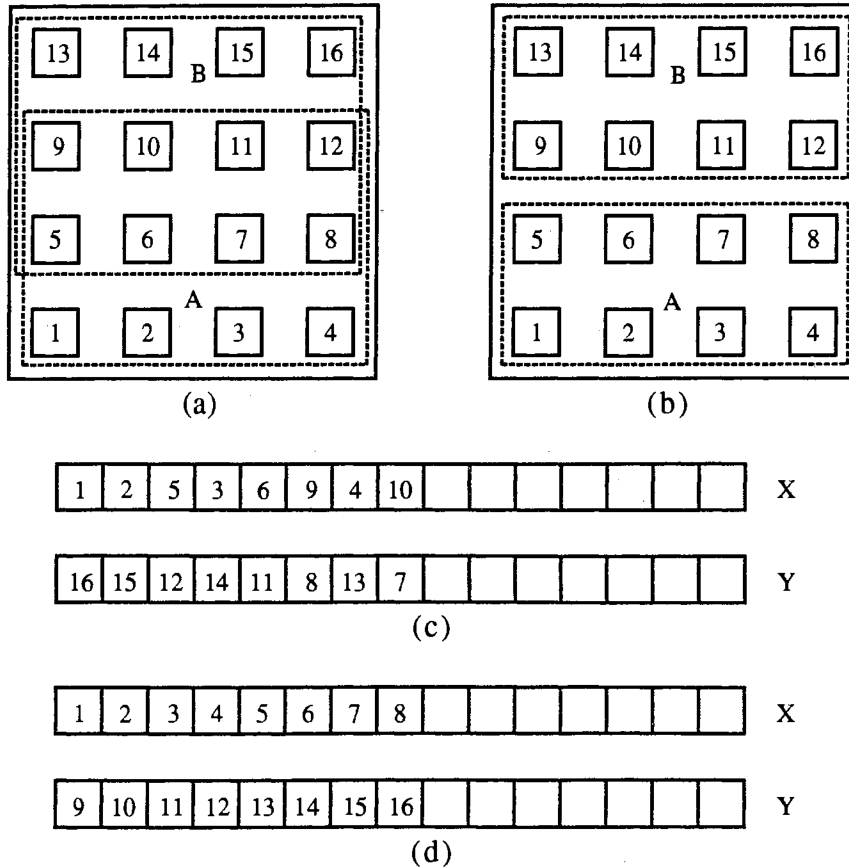


Figure 39 an example of the MXR-tree node split for uniformly distributed data: (a) before split and (b) after split.

Figure 40 illustrates sizes of the memory space required by the four parallel spatial index structures. The PAC-tree requires the smallest memory spaces and the MXR-tree uses the biggest memory space. The following formula is used to determine space (node) utilization of the four implemented parallel spatial index structures for Figure 41:

$$\text{space utilization } (\%) = \frac{\text{total number of distinct entries}}{\text{total number of nodes} \times \text{CAP}} \times 100$$

where total number of distinct entries is the sum of the number of intermediate node entries and leaf node entries.

The PAC-tree has the highest space utilization, except for data set V. The MXR-tree has the lowest space utilization.

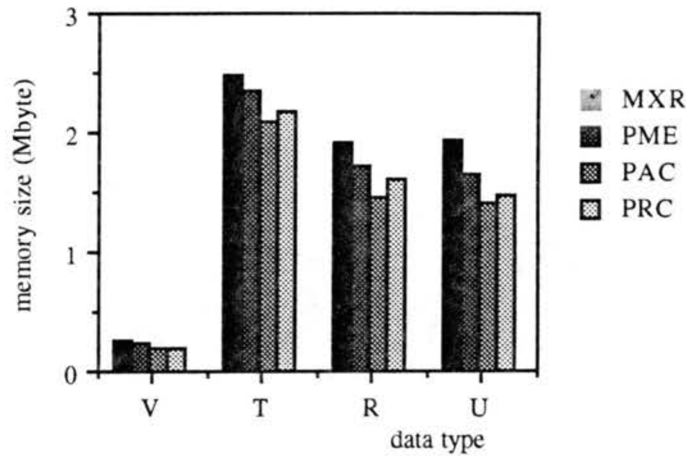


Figure 40 actual memory sizes of the MXR-tree, PME-tree, PAC-tree and PRC-tree for each of the four data sets V, T, R and U.

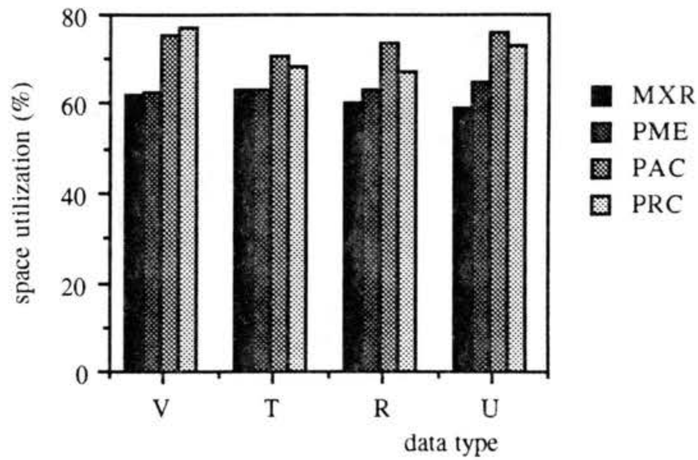


Figure 41 space utilization of the MXR-tree, PME-tree, PAC-tree and PRC-tree for each of the four data sets V, T, R and U.

Figure 42 illustrates construction time for each of the four index structures. The PME-tree has the fastest construction time for all data sets due to its simple object distribution heuristic. The PAC-tree and PRC-tree have longer construction times than the MXR-tree and PME-tree, since their object distribution heuristics take extra time. However, construction times for these two structures can be improved further by fully parallelizing the insertion processes. This is one of the future researches; in

this research, the author is focusing on improvement of query performance. Table 6 illustrates the number of re-inserted entries during the construction of the three PML-trees for the four test data sets.

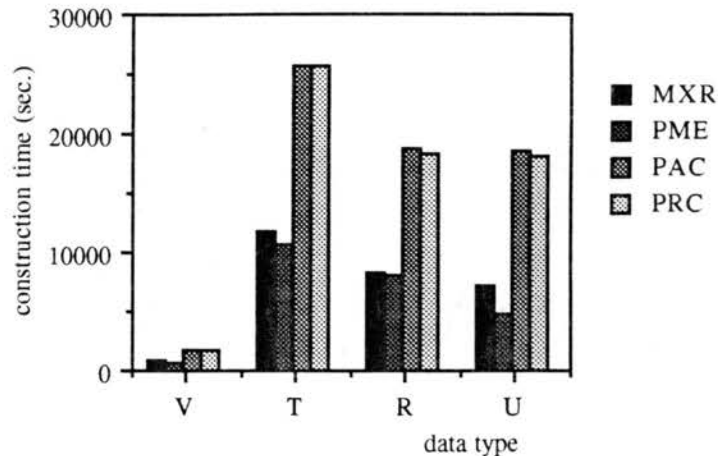


Figure 42 construction times of the MXR-tree, PME-tree, PAC-tree and PRC-tree for each of the four data sets V, T, R and U.

data type	PME	PAC	PRC
V	43(1.05%)	40(0.98%)	29(0.71%)
T	127(0.31%)	236(0.57%)	277(0.67%)
R	1813(6.04%)	1639(5.46%)	1782(5.94%)
U	1608(5.36%)	1435(4.78%)	1260(4.20%)

Table 6 the number of re-inserted entries for the four test data sets.

As mentioned in Chapter II, the MXR-tree has the same structure as the R-tree. Therefore, the same complexity analyses for the R-tree given in [91] can be used for the MXR-tree, too. The performance analysis for the R-tree given in [22] uses one-dimensional intervals of equal length and transforms them to points in two-dimensional space. This method provides limited performance analysis [91], since using uniformly distributed one-dimensional objects to analyze the performance of the spatial (multi-dimensional) index structures is inadequate. In the complexity analyses given in [91], spatial index structures are assumed to be constructed in batch manner. That is, all data objects are known prior to the construction.

The construction time and space requirements of the MXR-tree for n data objects are both $O(n)$, as mentioned in [91]. There is no leaf node redundancy in the MXR-tree; that is, a data object appears only in one of the leaf nodes; also the PML-tree does not have leaf node redundancy. For n data objects, the construction time and space requirements of the PML-tree are both $O(n)$.

CHAPTER V

THE SMR-TREE: A NEW EFFICIENT SERIAL SPATIAL INDEX STRUCTURE

Introduction

In this chapter, an efficient spatial index structure called the Serial Multi-R tree (SMR-tree) for spatial databases is proposed. The SMR-tree is designed to improve performance of existing serial spatial index structures (e.g., the R-tree, R^+-tree, R*-tree, ..., etc.) [5]. The SMR-tree is a variation of the PML-tree for serial disk I/O. The split algorithms and part of insertion algorithms of the SMR-tree are the same as those proposed for the PML-tree in Chapter III. The SMR-tree improves query performance by distributing spatial objects into several data spaces instead of one data space in the R-tree, the R^+-tree or the R*-tree. Each data space is associated with a tree in the SMR-tree. The structure of the SMR-tree avoids the node redundancy which appears in the R^+-tree at the leaf level and uses disjoint intermediate rectangles.

Three popular spatial index structures, the R-tree, R^+-tree and R*-tree, are implemented, based on the algorithms given in the original literature, and are compared with the SMR-tree. An experimental performance analysis for the four implemented structures is given with various types of testing data sets: random data, uniformly distributed data, VLSI layout data and TIGER/Line™ file data. The number of disk accesses and actual response time for each of those four index structures to process a query are compared. Construction time, space utilization and the actual memory size of the four index structures are also given.

The Structure and Algorithms of the SMR-tree

The SMR-tree Structure

The purpose of proposing the SMR-tree is to improve shortcomings of the R-tree, R^+-tree and R^*-tree, to provide better search and deletion performances. The SMR-tree is a dynamic serial spatial index structure using native space indexing with a disjoint space decomposition method. The SMR-tree is a variant of the PML-tree for serial disk I/O. In the PML-tree proposed in Chapter III, data objects are distributed into multiple trees stored on different disks using object distribution heuristics. In each tree of the PML-tree, all intermediate rectangles associated with intermediate nodes are disjoint and there is no redundancy in the leaf nodes. This way, the PML-tree speeds up query processing in parallel disk accessing with multiple disks. However, in a serial disk accessing environment, balanced multiple trees of the PML-tree increase the total number of disk accesses for query operations, since all intermediate nodes in the multiple trees have to be accessed in serial. The SMR-tree inserts all data objects into the first tree and removed data objects are inserted into other trees. Therefore, the first tree contains most of data objects (e.g., for TIGER/Line™ data set, about 90% of the data objects are stored in the first tree). The SMR-tree remains the same node structures as the PML-tree (see Chapter III). All data objects are stored at leaf nodes. Intermediate nodes consist of entries that represent intermediate rectangles. For example, entries in the intermediate node in Figure 43(c) are 11, 12, 13, 14, 17 and 18. Those entries represent the intermediate rectangles 11, 12, 13, 14, 17 and 18 in Figure 43(a), respectively. Each intermediate rectangle completely encloses all the rectangles in the child node at the lower level. This is one of the properties that is different from the R^+-tree; since an intermediate rectangle in the R^+-tree may not completely enclose all the rectangles in the child node

at the lower level. It can be clearly seen that this property makes the leaf nodes without redundancy possible. Furthermore, the intermediate rectangles are disjoint in the SMR-tree; thus, it makes fast search and deletion operations possible. The SMR-tree keeps the advantage of the R-tree, no redundancy at leaf level, and retains the good property of the R$^+$-tree, disjoint intermediate rectangles. In addition, the SMR-tree distributes data objects into several data spaces to eliminate duplication of data objects in leaf nodes.

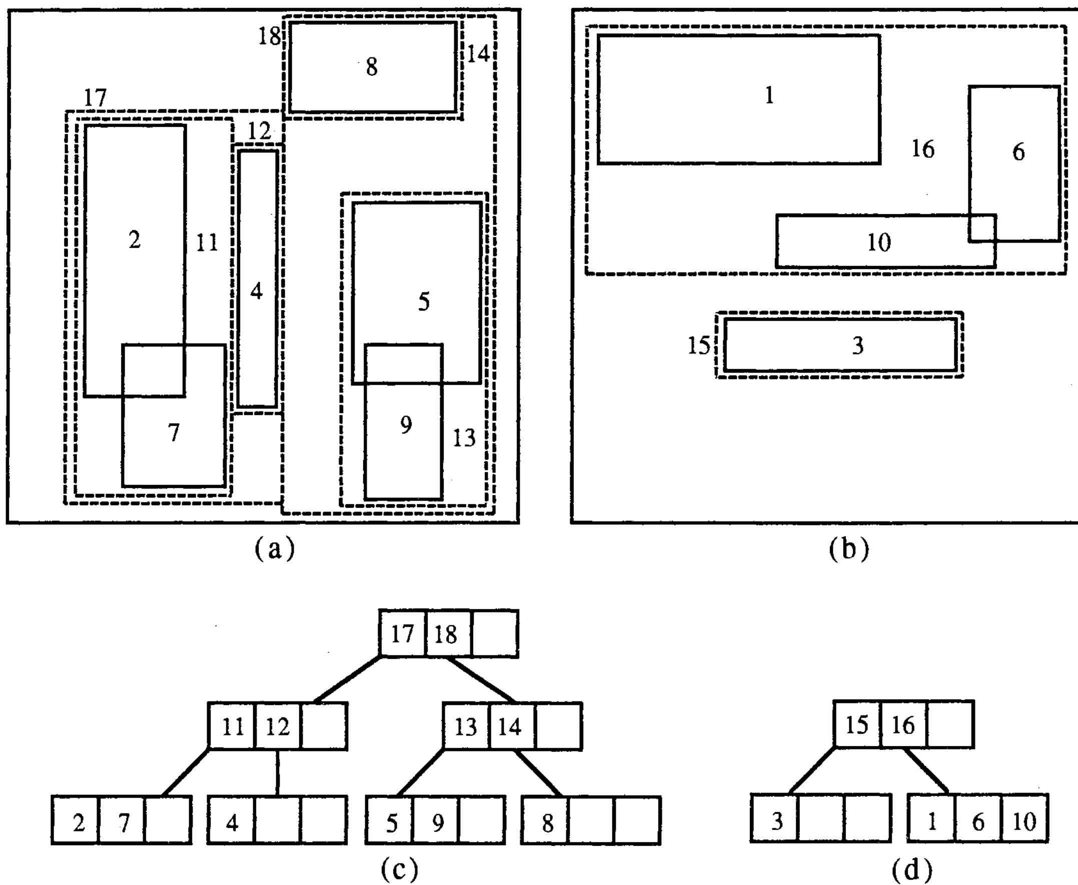


Figure 43 (a),(b) organizations of bounding rectangles (the solid lines construct object rectangles; the dash lines construct intermediate rectangles) and (c), (d) the SMR-tree structure.

In the SMR-tree, each data space is associated with a tree. Figure 18 uses the same set of two-dimensional data objects that were used in the examples for the R-tree, the R^+-tree, and the R^*-tree in Chapter II. The split algorithms for the SMR-tree extract the object rectangles 1, 3, 6, 10 and place them into another data space, say the second data space, to avoid overlapping as shown in Figure 43(b). Figure 43(a) shows the resulting object distribution in the original data space, i.e., the first data space. Figures 43(c) and 43(d) show the corresponding trees of the first and the second data spaces. At the leaf level of the SMR-tree, there are no duplicated object rectangles while each of the object rectangles 1, 2, 3, 4, 5, 6, 7, 8, and 10 was located at more than one leaf node in the R^+-tree as shown in Figure 12(b) (page 24).

Algorithms of the SMR-tree

There are some parameters that affect performance of the spatial index structure and decide insertion and split policies. Parameters affect each other independently; a change in one parameter may cause changes in other parameters. Also, this interrelation among the parameters acts differently on different data sets and different index structures. Therefore, it is difficult to optimize all parameters. The parameters in the following list are often considered in the design of spatial index structures:

1. area of intermediate rectangles,
2. overlapping areas between intermediate rectangles,
3. number of rectangles on the split lines,
4. perimeter of intermediate rectangles,
5. space utilization of nodes.

Parameters 1 through 4 should be minimized for good performance, and parameter 5 should be maximized. Not all combinations of the parameters can be applied to every spatial index structure. For example, parameter 3 cannot be used for index structures

with a non-disjoint space decomposition approach, e.g., R-tree or R*-tree, since these methods allow overlapping intermediate rectangles. In the split algorithm of some index structures with a disjoint decomposition method, parameter 1 is always the same no matter which split axis and split line are selected if the index structure does not use a minimum bounding rectangle. Parameter 2 is inapplicable to index structures using a disjoint space decomposition approach, e.g., the R^+-tree or the SMR-tree, since intermediate rectangles of these methods are disjoint. Different combinations and precedence orders of parameters also give different performance. Therefore, in order to optimize the performance of a spatial index structure, one must carefully select parameters and properly set the precedence order for those parameters during the design of the index structure with operational algorithms.

In the design of the SMR-tree and its associated algorithms, parameters 1, 3, and 5 are taken into account; parameter 5 is given the highest priority for split algorithms and is maximized. For a very small search range, an index structure with higher space utilization may not reduce the number of disk accesses. However, as the search range grows, maximizing parameter 5 has more effect on reducing the total number of disk accesses in a query than other parameters. Higher space utilization reduces the total number of nodes in the index structure, and as a result the height of the tree is most likely decreased. Even for the same height, a higher space utilization structure requires fewer disk accesses than a structure with lower space utilization. The SMR-tree's split algorithms evenly distributes entries into the nodes to increase space utilization. The insertion algorithm of the SMR-tree applies parameter 1 first to select an entry in every node along the insertion path and parameter 5 is used to resolve tie. Using C-like structures, all symbols and data types used in algorithms are defined in the following:

```
#define    CAP                                        /* node capacity      */
#define    M                  /* initial balancing factor; set to 30% of CAP */

struct RECT {                 /* the four coordinates of a 2-d rectangle      */
       int  lower_x, lower_y, upper_x, upper_y;
};

struct ENTRY {
       struct RECT   *R;      /* the four coordinates of a 2-d rectangle      */
       struct NODE   *P;  /* pointer to a child node or to a record in database */
};

struct NODE {
       int  ent_num;                     /* number of entries in the node     */
       struct ENTRY   *ETY[CAP + 1];  /* array of entry pointer               */
};

struct TREE_INFO {
       struct NODE   *P;                 /* root node pointer of the tree     */
       struct TREE_INFO   *next;
};

struct RECT          *W;                        /* search or delete range */
struct ENTRY         *E, *E', *NEW;                  /* entry pointers    */
struct NODE          *N, *SP, *P, *R, *REM;          /* node pointers     */
struct TREE_INFO     *TREE_LIST, *TREE;
```

In the following sections, of insertion, split, search and deletion algorithms for the SMR-tree are proposed.

Insertion of the SMR-tree

The insertion operation of the SMR-tree consists of five algorithms: Insert, SelectTree, InsertRect, SelectEntry and Adjust. Algorithms InsertRect, SelectEntry and Adjust are the same as those used for the PML-tree in Chapter III; these three algorithms are not given in this chapter (see Chapter III, page 58).

Algorithm Insert takes one entry NEW to insert, sets TREE to the first tree in the SMR-tree and invokes algorithm SelectTree. Algorithm SelectTree invokes the algorithm InsertRect until either the entry NEW is inserted into one of the trees or none of trees in the SMR-tree accepts the entry NEW without violating the SMR-tree properties (e.g., the properties of disjoint intermediate rectangles and complete enclosure of the object rectangles in an intermediate rectangle). If no tree accepts entry NEW, then a new tree is created and the entry NEW is inserted into it. Algorithm InsertRect invokes itself recursively to select the insertion path using the algorithm SelectEntry. Algorithm SelectEntry examines entries in the node to select an entry that satisfies the SMR-tree properties and the entry's associated intermediate rectangle can encloses the object rectangle associated with entry NEW. The SMR-tree uses minimum area as a criterion to select an entry in a node on the insertion path. The minimum area parameter is superior to the minimum enlargement parameter for insertion. In the minimum enlargement approach, entries associated with larger intermediate rectangles have more chance to be selected than entries with smaller intermediate rectangles from the probability point of view; this effect leads to uneven growth, in size, of the intermediate rectangles. The minimum area parameter provides balanced growth among intermediate rectangles in the SMR-tree and as a result, space utilization is improved. Reducing the area of intermediate rectangles is critical since the SMR-tree does not allow overlapping intermediate rectangles. If intermediate rectangles have smaller area, more intermediate rectangles can be placed in a data space. This may reduce the total number of data spaces for the SMR-tree. If there is a tie, an intermediate rectangle with fewer sub-rectangles in it is chosen to increase space utilization and reduce the chance of node overflow. After the entry NEW is inserted into one of the trees, minimum bounding rectangles of the all entries along the insertion path are adjusted by using the algorithm Adjust. If a node split during the insertion process, then a new entry for the split node is created and

inserted into the parent node. In the case of root node splitting, a new root node is created and the root pointer in TREE is updated. If entries are removed during the node splitting, these entries are re-inserted in one of the subsequent trees. For example, if an entry is removed from the ith tree, then this entry is re-inserted into the $(i + 1)^{th}$ tree.

```
Algorithm Insert()
[Output]      a new SMR-tree after insertion of NEW.
[Comment]     Invoke algorithm SelectTree() to insert an entry NEW. If an entry is
              removed during the insertion process, that entry is re-inserted into one
              of the subsequent trees.
1.  set TREE= TREE_LIST->next;              /* TREE is set to the first tree */
2.  invoke SelectTree(TREE, NEW, REM);
3.  if REM->ent_num > 0 then                /* if there is removed entry     */
        for each entry E in REM,
        set NEW= E;
        if E is removed from the ith tree (TREE) then
            set TREE= TREE->next;
        goto step 2 to re-insert the removed leaf level entry;
```

```
Algorithm SelectTree(TREE, NEW, REM)
[Input]       TREE- structure TREE_INFO.    NEW- entry to be inserted.
              REM- store removed entries.
[Output]      a new SMR-tree after insertion of NEW.
[Comment]     Invoke InsertRect() to insert NEW.
1.  let N be the root node of the tree, N= TREE->P;
2.  invoke SP= InsertRect(N, NEW, REM), where SP denotes a split node;
3.  if insertion of the new object rectangle NEW is successful then
        if SP ≠ NULL then                              /* root node split */
            create entries E and E' for node N and SP;
            create a new root node R and insert E and E' into R as entries;
            reset root node in TREE, TREE->P= R;
    else if there is no tree in which NEW can be inserted then
        create a root node R and insert NEW into R;
        create a TREE and set TREE->P= R and put TREE in TREE_LIST;
    else set TREE= TREE->next;  goto step 1;
```

Node Split of the SMR-tree

In the case of a node overflow during the insertion operation, a splitting process is needed. As mentioned, the SMR-tree uses exactly the same split algorithms as the PML-tree. The split algorithms consists of three sub-algorithms, Split, SelectSpline and SubSplit (see Chapter III, page 61).

Search and Deletion of the SMR-tree

The search and deletion algorithms of the SMR-tree consist of two sub-algorithms, respectively. The algorithm Search invokes SearchTree and algorithm Delete calls DeleteTree for all trees in the SMR-tree. The algorithms SearchTree and DeleteTree are almost the same as those used for most members of the R-tree family.

```
Algorithm Search (W)
[Input]       W- structure RECT; search range.
[Output]      all data objects overlapping W.
[Comment]     For each tree in TREE_LIST, invoke SearchTree ().
1.   set TREE= TREE_LIST->next;
2.   if TREE ≠ NULL then                              /* if the tree is not empty */
          invoke SearchTree (TREE->P, W);
          TREE= TREE->next;  goto step 2;
```

```
Algorithm SearchTree (N, W)
[Input]       N- node. W- structure RECT; search range.
[Output]      data objects overlapping W.
[Comment]     Traverse tree recursively and save leaf entries overlapping W.
1.   save node pointer, P= N;
2.   if P is not a leaf node then, for each entry E in P,
          if E->R overlaps W then
               set N= E->P and invoke SearchTree (N, W) recursively;
3.   if P is a leaf node then for each entry E in P,
          if E->R overlaps W then
               save entry pointer;
```

```
Algorithm Delete (W)
[Input]       W- structure RECT; delete range.
[Output]      re-structured PML-tree.
[Comment]     For each tree in TREE_LIST, invoke DeleteTree ().
int  Flag;                /* set to TRUE if at least one leaf entry is removed */
1.   set TREE= TREE_LIST->next;  set Flag= False;
2.   if TREE ≠ NULL then                            /* if the tree is not empty */
         invoke DeleteTree (TREE->P, W, Flag);
             if TREE->P->ent_num is 0 then              /*root node empty */
                 remove TREE from TREE_LIST;
         set TREE= TREE->next;  goto step 2;
```

```
Algorithm DeleteTree (N, W, Flag)
[Input]       N- node.  W- structure RECT; delete range.  Flag.
[Output]      re-structured SMR-tree.
[Comment]     Traverse tree recursively. Remove leaf entries overlapping W.
1.   save node pointer, P= N;
2.   if P is not a leaf node then for each entry E in P,
         if E->R overlaps W then
             set N= E->P and invoke DeleteTree (N, W, Flag) recursively;
             if Flag then          /* there exists at least one removed leaf entry */
                 if N->ent_num > 0 then                   /* not empty node */
                     invoke Adjust (E) to re-adjust boundary of E;
                 else remove E from P;  P->ent_num--;
3.   if P is a leaf node then for each entry E in P,
         if E->R overlaps W then
             remove entry E from P;  P->ent_num--;  set Flag= True;
```

An object rectangle, namely a rectangle at leaf level, in the R^+-tree may not be completely enclosed by its upper level intermediate rectangle(s). Therefore, an object rectangle may appear in more than one leaf node in the R^+-tree and results in redundancy. If one wants to delete all object rectangles overlapping a given range (called deletion by range), duplicated entries at the leaf level may not be completely deleted from the R^+-tree with the deletion algorithm provided [95]. In the SMR-tree,

there is no such problem, since object rectangles are completely enclosed by upper level intermediate rectangles.

Performance Comparisons

The SMR-tree's performance is compared with the performance of the R-tree, the R^+-tree and the R*-tree. All four index structures use native space indexing. The R-tree and the R*-tree employ a non-disjoint space decomposition method. The R^+-tree and the SMR-tree use a disjoint space decomposition method. The R-tree, the R^+-tree and the R*-tree were chosen for comparison to the SMR-tree because of the following reasons:

1. the original R-tree provides comparatively good space utilization and this factor makes range queries more efficient as the query size increases,
2. the R*-tree, one of the variants of the R-tree, has good space utilization because of forced re-insertion,
3. the R^+-tree has good query performance in comparatively small size search ranges.

It is well known that the performance of serial spatial index structures in query operations depends on the total number of disk accesses to the secondary storage (e.g., disk). An efficient spatial index structure requires fewer disk accesses to process a query. In this chapter, an experimental performance evaluation method is used to measure the performances of the above four spatial index structures. Two types of data, system generated data (e.g., uniformly distributed data and randomly generated data) and real application data (e.g., Tiger/Line™ files for TIGER, geographic information system, database, VLSI layout data generated by Magic system), are used with the proposed spatial index structure, the SMR-tree and its competitors, i.e., the R-tree, the R^+-tree and the R*-tree. Let V, T, R and U represent

the VLSI data set, the Tiger/Line™ data set, the randomly distributed data sets and the uniformly distributed data set respectively. Descriptions of the four data sets and query types for real application data sets are described in Chapter V (page 76). Various search areas (e.g., a point to 12% of the data space) were randomly generated and searched. The average number of disk accesses and actual response time, for each of the four implemented spatial index structures to process a query, are recorded. In addition, memory sizes, space (node) utilization and construction time of the four structures are compared.

Implementations of Spatial Index structures

The SMR-tree is implemented based on the algorithms proposed in this chapter. The R-tree is implemented with algorithms presented by Güttman [40]. The quadratic split algorithm is used for the implementation of the R-tree, since it is well known that the quadratic splitting algorithm has better performance than the linear splitting algorithm. The minimum node fill factor, m, of the R-tree is set to 40% of the node capacity to give the best splitting performance. The R^+-tree is implemented based on the algorithms provided in [95]. As mentioned in Chapter II, the R^+-tree's implementation does not use MBRs (minimum bounding rectangles) for intermediate rectangles. At each level, say j, of the R^+-tree, the data space should be completely filled with disjoint, level j, intermediate rectangles. The node split process of the R^+-tree may create empty node(s). However, this empty node(s) should be kept in the R^+-tree for later insertions. The R*-tree is implemented based on the algorithms given in [7]. To give the best performance for this structure, the minimum node fill factor is set to 40% of the node capacity and the number of entries removed in forced re-insertion is set to 30% of the node capacity. All programs are written in computer C language and simulated on two-dimensional data space.

Experiment Results

All the programs were run on the *Sequent Symmetry S81* with twenty four 80386 processors running at 20Mhz each. In this experiment, page size is 1 Kbytes.

Size of an entry is 35 bytes (e.g., 7 bytes for each of four coordinates and one child node pointer). The size of the node is set to multiple of actual page size for reasons of efficiency. Node capacity is 87 (3 pages) for test data sets T, R and U and 29 (1 page) for the test data set V.

A query rectangle represents the area in a data space that has to be searched for processing a range query (a point query is included in the category of range queries with zero area). Range of the query rectangles area sizes is 0% - 12% of the data space. For each search range (e.g., 4% of a data space) 500 search regions are randomly selected and searched. The corresponding average number of disk accesses and the response time for each search range are calculated.

Figures 44, 45, 46 and 47 illustrate the average number of total disk accesses for the four index structures vs. the size of the query for data sets V, T, R and U. The SMR-tree requires fewer disk accesses than the R-tree, R^+-tree and R^*-tree for most of search ranges (greater than 2%). For very small search ranges (less than 1%), the R^+-tree has the smallest number of disk accesses. For example, in a point query, the R^+-tree only needs its height plus one disk accesses, since all objects overlapping a given point can be found in one leaf node in the R^+-tree. In the structures with a disjoint space decomposition method, all data objects overlapping a given point in the data space can be found in a leaf node. As the size of the search range increases, redundancy in the R^+-tree increases the total number of disk accesses. The R-tree has a redundant search path problem because of non-disjoint space decomposition method, as mentioned in Chapter II and it increases the number of nodes involved in redundant search paths as the search range increases. Forced re-insertion of the R^*-

tree increases the area of overlapping intermediate rectangles, especially in test data set R, since removed object rectangles are re-inserted into intermediate rectangles around the intermediate rectangle from which rectangles are removed.

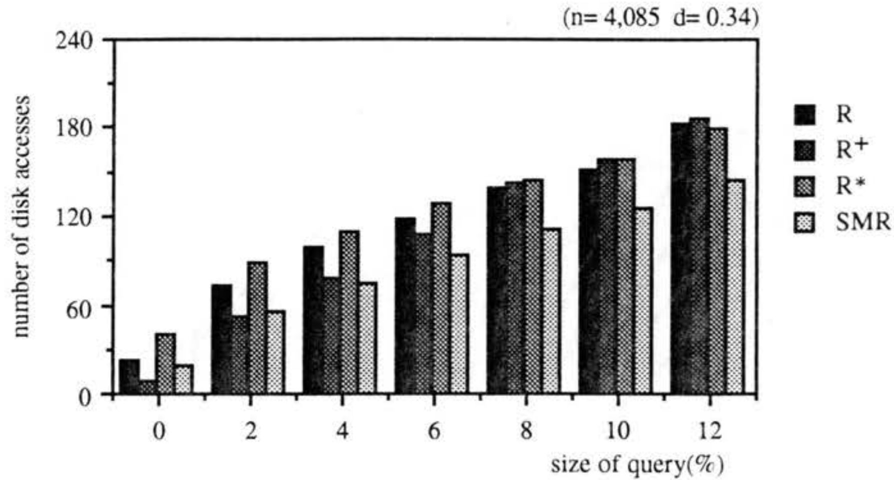


Figure 44 average numbers of disk accesses of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the size of query using data set V.

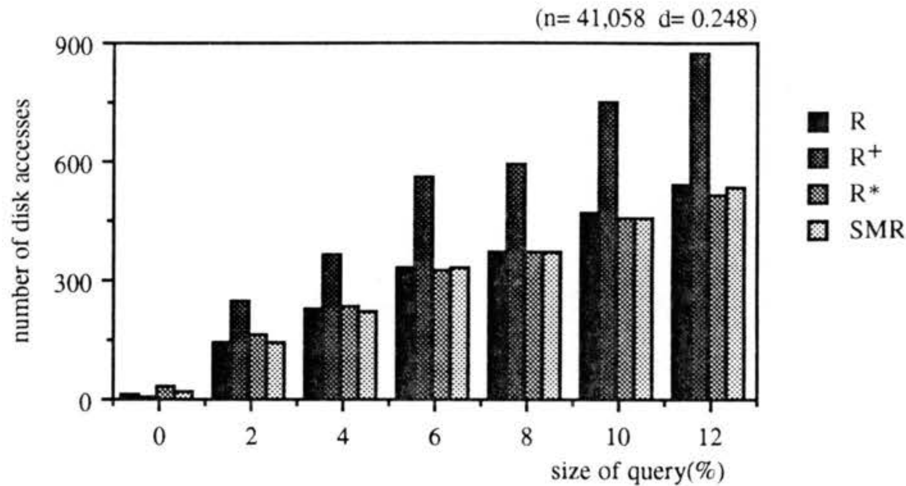


Figure 45 average numbers of disk accesses of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the size of query using data set T.

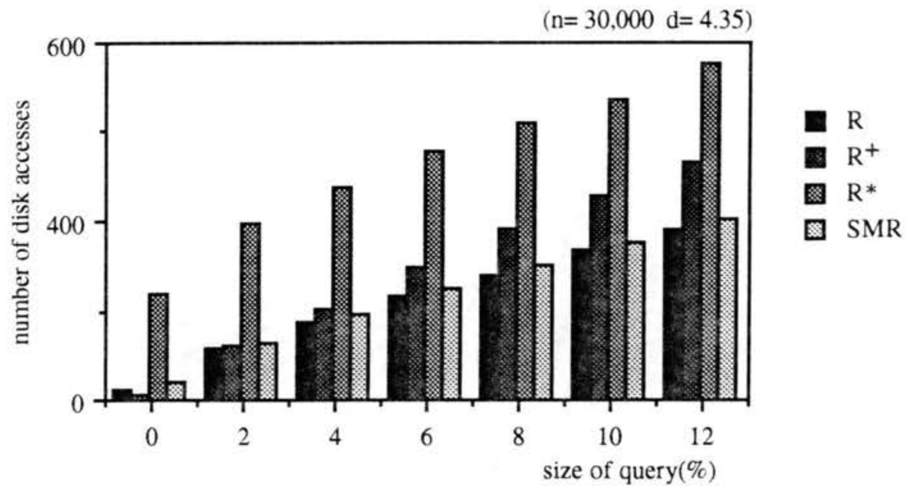


Figure 46 average numbers of disk accesses of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the size of query using data set R.

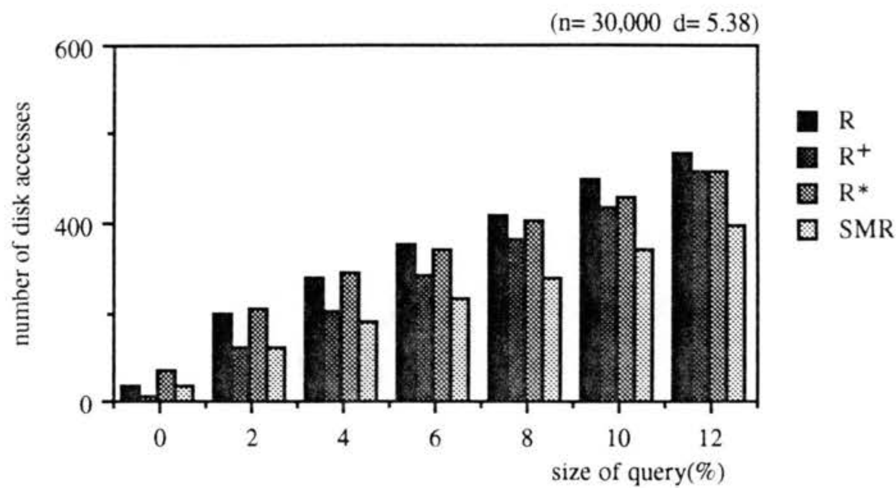


Figure 47 average numbers of disk accesses of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the size of query using data set U.

Figures 48, 49, 50 and 51 illustrate the average response time of the four index structures vs. the size of the query for the four data sets (V, T, R and U). In serial spatial index structures, the query performance of the structures is almost proportional to the total numbers of disk accesses to process a given query. Time to process nodes uploaded into main memory does not make a significant difference in the query

performance of the spatial index structures, since one node access time from secondary storage takes much longer than one node processing time in main memory.

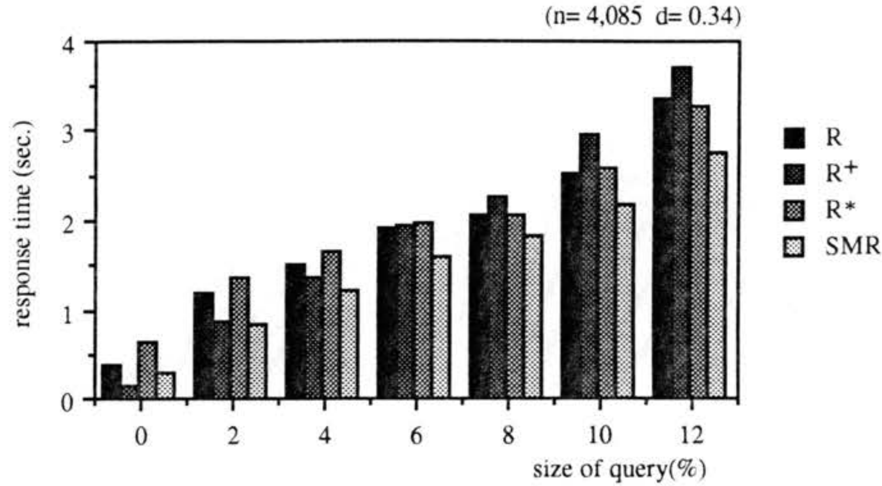


Figure 48 average response time of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the size of query using data set V.

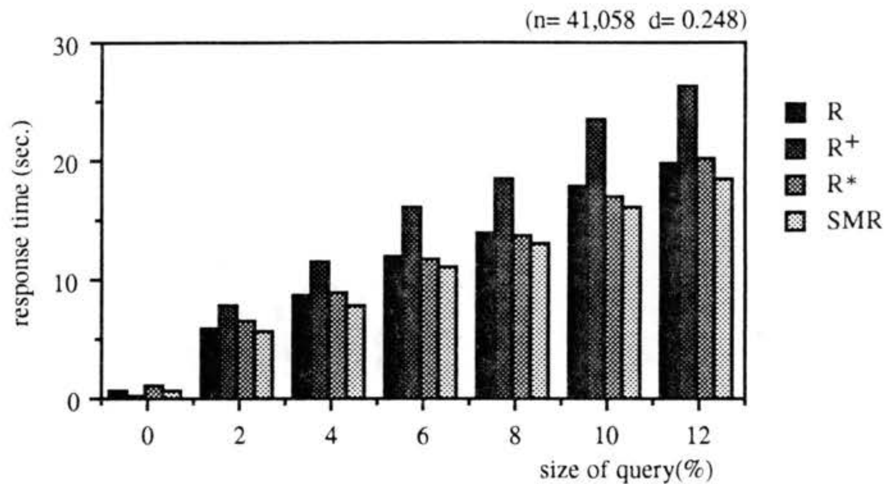


Figure 49 average response time of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the size of query using data set T.

The response time of the four spatial index structures in Figures 48, 49, 50 and 51 is almost proportional to the total numbers of disk accesses in Figures 44, 45, 46 and 47, respectively. If two structures have almost the same number of disk accesses, then

the structure that has fewer entries in the nodes accessed, is faster. For example, in Figure 44, the R^+-tree has almost the same number of disk accesses as the R^*-tree for the 10% search range, but in Figure 48, the R^+-tree takes more processing time than the R^*-tree for the same search range. Redundant entries in the nodes in the R^+-tree increases query processing time.

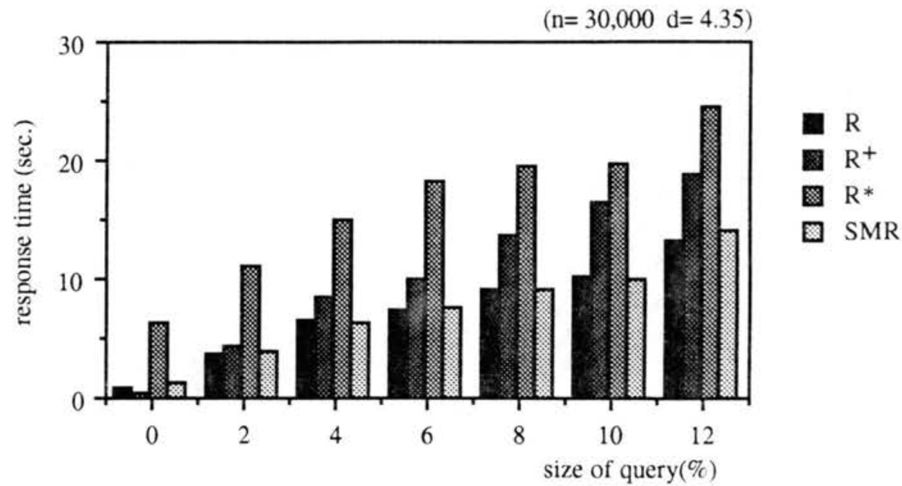


Figure 50 average response time of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the size of query using data set R.

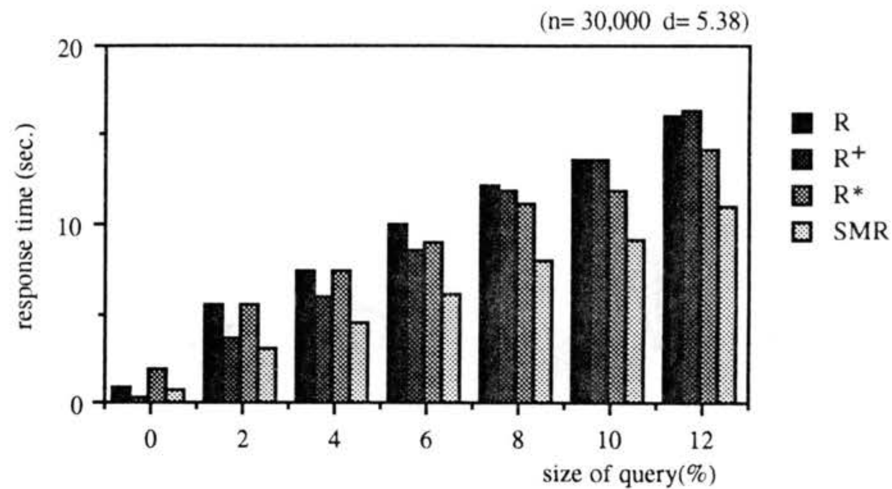


Figure 51 average response time of the R-tree, R^+-tree, R^*-tree and SMR-tree vs. the size of query using data set U.

Let RT_R, RT_R^+, RT_R^* and RT_SMR denote response time of the R-tree, R^+-tree, R^*-tree and SMR-tree, respectively. Let SMR/R, SMR/R^+ and SMR/R^* denote the performance gains of the SMR-tree over the R-tree, R^+-tree and R^*-tree, respectively. Figures 52, 53, 54 and 55 illustrate the performance gains of the SMR-tree over the R-tree, R^+-tree and R^*-tree with the four different data sets. The performance gain SMR/R is defined as:

$$\text{performance gain SMR/R (\%)} = \frac{RT_R - RT_SMR}{RT_SMR} \times 100$$

The SMR/R^+ and SMR/R^* are defined correspondingly. The SMR-tree outperforms the R-tree, R^+-tree and R^*-tree throughout the search ranges, except very small search ranges.

The performance of the SMR-tree is very stable among the four test data sets. The R-tree has comparatively low query performance for the data set U compared to its performance for other data sets. The split algorithm of the R-tree cannot split a node efficiently when data objects are uniformly distributed. The drawback of the R-tree's node split for uniformly distributed data objects is discussed in Chapter IV using Figure 39 (page 88). The split algorithm of the R-tree examines all entries in an overflowing node, then inserts a selected entry into one of two nodes, the original node and the split node. With this algorithm, the two nodes can severely or completely overlap after the node split for uniformly distributed data objects. On the other hand, for the object distribution in Figure 39(a), the R^+-tree, R^*-tree and SMR-tree can have disjoint intermediate rectangles after a node split, since each of their split algorithms uses a selected split line on one of the axes. Query performance of the R^*-tree is comparatively low for the data set R compared to its performances for other data sets due to its forced re-insertion, as mentioned above. The R^+-tree has low query performance in comparatively large search ranges for all test data sets. Redundancy of the R^+-tree increases when data density is high. For example, even

though data set T has very low overall density (D= 0.248), some areas in the data space have very high densities (e.g., D= 60). Therefore, the R+-tree has low query performance, especially for data set T.

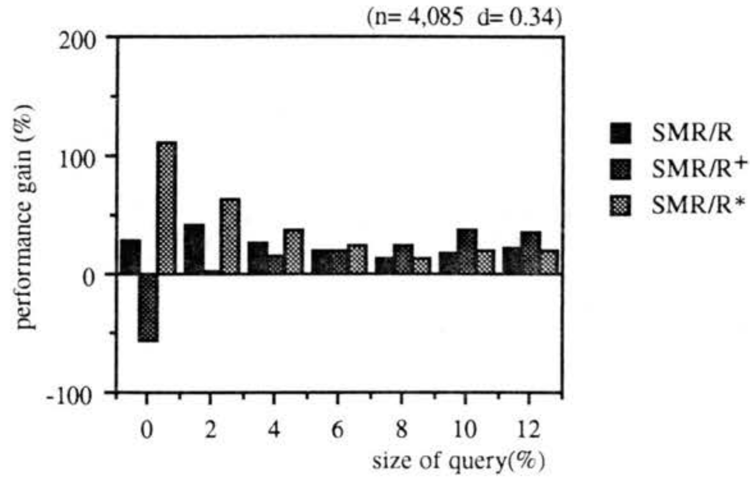


Figure 52 performance gains of the SMR-tree over the R-tree, R+-tree and R*-tree vs. the size of query using data set V.

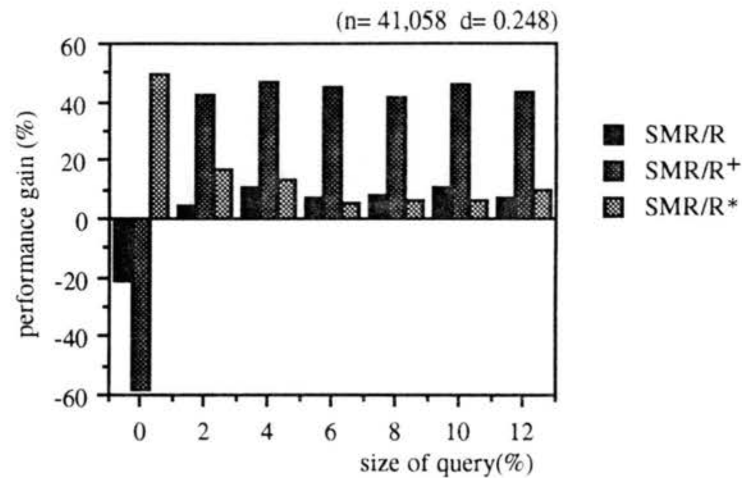


Figure 53 performance gains of the SMR-tree over the R-tree, R+-tree and R*-tree vs. the size of query using data set T.

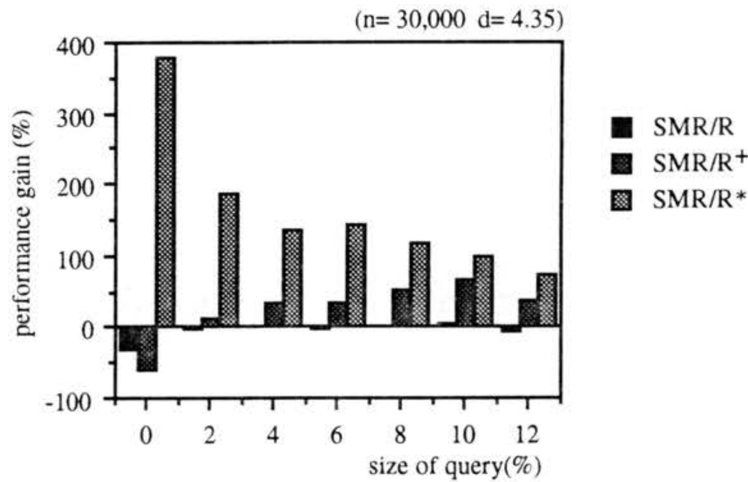


Figure 54 performance gains of the SMR-tree over the R-tree, R^+-tree and R^*-tree vs. the size of query using data set R.

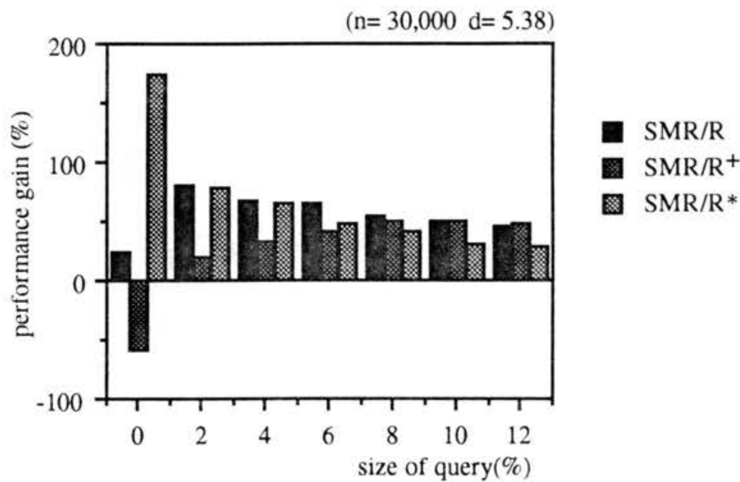


Figure 55 performance gains of the SMR-tree over the R-tree, R^+-tree and R^*-tree vs. the size of query using data set U.

Table 7 illustrates the distribution of the data objects among the trees in the SMR-tree for the four test data sets. Figure 56 illustrates the size of the memory spaces required by the four spatial index structures. The R^*-tree requires the smallest memory space and the R^+-tree uses the largest memory space. Figure 57 shows space utilization of the four implemented spatial index structures using the equation given in Chapter IV (page 88). The R^*-tree has the highest space utilization

and the R^+-tree has the lowest space utilization for all data sets. Nodes in the R^+-tree are filled to over 60%, but only the number of distinct entries is considered in the calculation of space utilization. Figure 58 illustrates construction time for each of the four spatial index structures. The R-tree has the fastest construction time for all data sets due to its simple split algorithm. The R*-tree takes the longest construction times among the four implemented spatial index structures due to entry removals and re-insertions to resolve node overflowing. The R-tree, R^+-tree and SMR-tree require similar construction times.

	V	V	T	T	R	R	U	U
	obj_num	obj_perc	obj_num	obj_perc	obj_num	obj_perc	obj_num	obj_perc
T_1	3418	83.627%	36609	89.164%	20615	68.717%	22068	73.560%
T_2	348	8.519%	4155	10.120%	6442	24.473%	6657	22.190%
T_3	212	7.809%	290	0.706%	2334	7.780%	1266	4.220%
T_4	0	0%	4	0.009%	569	1.897%	9	0.030%
T_5	0	0%	0	0%	40	0.1333%	0	0%

Table 7 distribution of the data objects among the trees in the SMR-tree (obj_num and obj_perc denote the number of objects and the percentage of the objects in each tree, respectively).

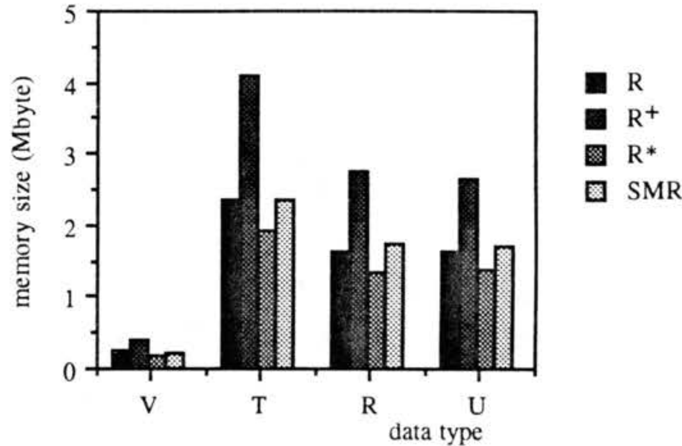


Figure 56 actual memory sizes of the R-tree, R^+-tree, R*-tree and SMR-tree for each of the four data sets V, T, R and U.

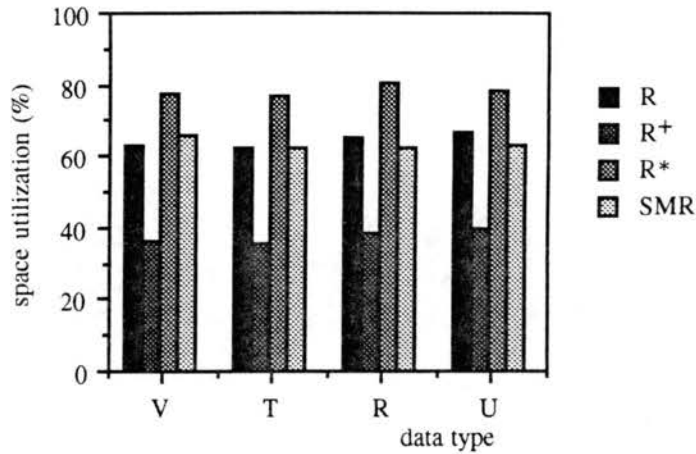


Figure 57 space utilization of the R-tree, R^+-tree, R^*-tree and SMR-tree for each of the four data sets V, T, R and U.

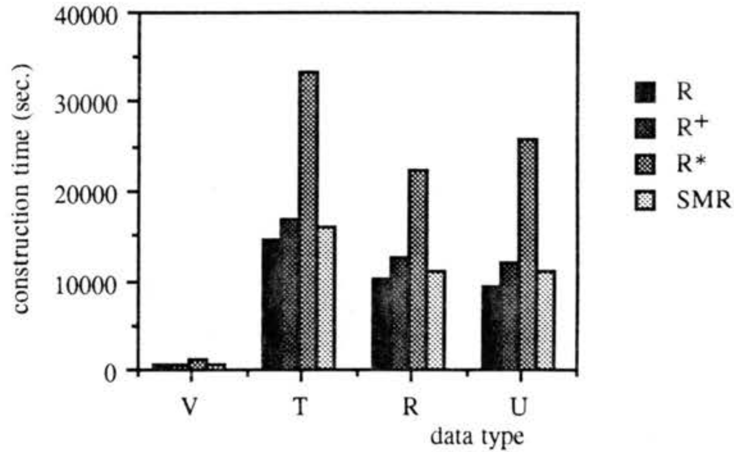


Figure 58 construction times of the R-tree, R^+-tree, R^*-tree and SMR-tree for each of the four data sets V, T, R and U.

Summary

In this chapter, the SMR-tree has been designed and implemented. A new split algorithm is proposed with a set of other operational algorithms. The performance of the SMR-tree has been compared with the R-tree, the R^+-tree and the R^*-tree using four test data sets. The comparison results show that the SMR-tree has improved performance over the R-tree, the R^+-tree and the R^*-tree in most cases. The R-tree

may require redundant paths in search and deletion processes due to its non-disjoint space decomposition method and its split algorithms are not efficient for uniformly distributed data set. The R^+-tree has redundancy in leaf nodes. This redundancy provides the fastest processing of a point query and range query with very small search range (e.g., less than 1% of the data space), but degrades performance in range query processing and deletion operations. The performance of the R^+-tree is very low for data set T, since data set T has very high local densities in some areas of the data space. The R*-tree takes long construction time and has low query performance for data set R due to forced re-insertions. By distributing data objects in several data spaces and using a disjoint space decomposition approach with each intermediate rectangle completely enclosing its children at lower levels, the SMR-tree structure improves speed for searching and deletion and avoids leaf node redundancy. As a result, the SMR-tree provides better query performance than the R-tree and the R*-tree for most search ranges, except for very small size range searches (e.g., < 1%) as shown in Figures 52, 53, 54 and 55. The time complexity of deletion operations with the SMR-tree is lower than those of the R-tree, the R^+-tree and the R*-tree. The SMR-tree is even more efficient with high density data. The SMR-tree has very stable query performance for all test data sets, thus, the SMR-tree can be used as an efficient general purpose index structure for spatial databases.

CHAPTER VI

CONCLUSIONS

In this research, a new parallel spatial index structure called the PML-tree, has been designed and implemented. The PML-tree improves query performance significantly by introducing parallelism in I/O operations with multiple disks. To distribute data objects evenly among the disks, three object distribution heuristics (ME, AC and RC) are proposed with a set of other operational algorithms. By distributing data objects in multiple data spaces and using a disjoint space decomposition approach with each intermediate rectangle completely enclosing its children at lower levels, the PML-tree improves query speed and avoids leaf node redundancy. Three PML-trees (PME-tree, PAC-tree and PRC-tree) for the three object distribution heuristics are implemented and the performance of these structures is compared with that of the MXR-tree using various test data sets. All three PML-trees outperform the MXR-tree for most of search ranges, except for very small size range queries (e.g., point queries with data sets V and R) as can be observed in Figures 27 through 38 (pages 91 - 97). The PAC-tree and PRC-tree have very good space utilization. The PRC-tree has the best query performance among the three PML-trees, except with data set R. Currently, the PAC-tree and PRC-tree have longer construction times than the PME-tree and MXR-tree. By fully parallelizing the insertion processes for the PAC-tree and PRC-tree, the construction times for these structures are expected improve. The PML-trees have very stable query performance with all test data sets. On the other hand, the MXR-tree has the lowest query performance for data set U compared to its performance for other data sets due to its split algorithms. All experiment results indicate that the PML-trees are efficient general purpose spatial index structures for spatial data bases.

REFERENCES

1. Aref, W. G. and Samet, H. Optimization strategies for spatial query processing. In Proceedings of the 17th International Conference on Very Large Data Bases, pp. 81-90, 1991.

2. Banerjee, J. and Kim. W. Supporting VLSI geometry operations in a database system. In Proceedings of the IEEE 2nd International Conference on Data Engineering, pp. 409-415, 1986.

3. Bang, K. S. and Lu, Huizhu. A simulation on an index structure for the spatial object. In Proceedings of the International Simulation Technology Conf. (SIMTEC'92), November, pp. 178-183, 1992.

4. Bang, K. S. and Lu, Huizhu. An application of the multiple-R tree to the VLSI circuit layout design. In Proceeding 9th International Conf. on System Engineering, University of Nevada Las Vegas, pp. 295-299, 1993.

5. Bang, K. S. and Lu, Huizhu. SMR-tree: an efficient index structure for spatial databases. In Proceeding of the 1995 ACM Symposium on Applied Computing, Nashville, February, pp. 46-50, 1995.

6. Becker, B., Six, H. W. and Widmayer, P. Spatial priority search: An access technique for scaleless maps. ACM SIGMOD, pp. 128-137, 1991.

7. Beckmann, N. and Kriegel, H. P. The R*-tree: An efficient and robust access method for points and rectangles. ACM SIGMOD, pp. 322-331, 1990.

8. Benson, D. and Zick, G. Symbolic and spatial database for structural biology. In Proceedings of the ACM Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) Conference, pp. 329-339, 1991.

9. Bentley, J. L. and Finkel, R. K. Quad-trees: A data structure for retrieval on composite keys. ACTA Informatica 4(1): 1-9, 1974.

10. Bentley, J. L. Multiple-dimensional binary search trees used for associative searching. Communications of the ACM 18(9): 509-517, 1975.

11. Bentley, J. L. Multiple-dimensional binary search trees in database applications. IEEE Transactions on Software Engineering, SE-5(4): 333-340, 1979.

12. Bentley, J. L. and Friedman, J. H. Data structures for range searching. ACM Computing Surveys, 11(4): 397-409, 1979.

13. Blanken, H., IJbema, A.,Meek, P. and Akker, B. The generalized Grid file: Description and performance aspects. In Proceedings of the IEEE 6th International Conference on Data Engineering, pp. 380-388, 1990.

14. Brolio, J., Draper, B. A., Beveridge, J. R. and Hanson, A. R. ISR: A database for symbolic processing in computer vision. IEEE Computer, pp. 22-30, 1989.

15. Bureau of the Census. TIGER/Line Files. 1992 Technical Documentation, Bureau of the Census, Washington, DC, 1993.

16. Chang, S. K. Pictorial data-base systems. IEEE Computer, November, pp. 13-21, 1981.

17. Chang, S. K., Yan, C. W., Dimitroff, D. C. and Arndt, T. An intelligent image database system. IEEE Transactions on Software Engineering, 14(5): 681-688, 1988.

18. Chang, N. S. and Fu, K. S. Picture query lagguages for pictorial data-base systems. IEEE Computer, November, pp. 23-33, 1981.

19. Charlton, M. E., Openshaw, S. and Wymer, C. Some experiments with an adaptive data structure in the analysis of space-time data. In Proceeding of the 3rd Spatial Data Handling Conference, pp. 1030-1039, 1990.

20. Chock, M. Cardenas, A. F. and Klinger, A. Manipulating data structures in pictorial information systems. IEEE Computer, November, pp 43-50, 1981.

21. Csillag, F. and Kummert, A. Spatial complexity and storage requirements of maps represented by region Quad-trees. In Proceedings of the 3rd Spatial Data Handling Conference, pp. 928-937, 1990.

22. Faloutsos, C. Sellis, T. and Roussopoulos, N. Analysis of object oriented spatial access methods. In Proceedings of the ACM SIGMOD Conference,16(3), pp. 426-439, 1987.

23. Faloutsos, C. Gray codes for partial match and range queries. IEEE Transactions on Software Engineering,14(5): 1381-1393, 1988.

24. Frank, A. U. Properties of geographic data: Requirements for spatial access methods. Lecture Notes in Computer Science 525, Advances in Spatial Databases (SSD'91), pp. 225-234, 1991.

25. Freestone, M. The BANG file: A new kind of Grid file. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 260-269, 1987.

26. Freestone, M. A well-behaved file structure for the storage of spatial objects. Symposium on the Implementation of Large Spatial Databases, pp. 287-300, 1989.

27. Freestone, M. The comparative performance of BANG indexing for spatial objects. In Proceedings of the 5th Spatial Data Handling Conference, 1, pp. 190-199, 1992.

28. Fuchs, H., Abram, G. D. and Grant,E. D. Near real-time shaded display of rigid objects. Computer Graphics, 17(3): 65-72, 1983.

29. Gargantini, I. An efficient way to represent Quad-trees. Communications of the ACM, 25(12): 905-910, 1982.

30. Gold, C. M. An object-based dynamic spatial model, and its application in the development of a user-friendly digitizing system. In Proceedings of the 5th Spatial Data Handling Conference, 2, pp. 495-504, 1992.

31. Goodchild, M. F. and Shiren, Y. A hierarchical spatial data structure for global geographic information systems. Graphical Models and Image Processing, 54(1): 31-44, 1992.

32. Grosky, W. I. Image database management. IEEE Computer, December, pp. 7-8, 1989.

33. Guenther, and Buchmann, A. Research issues in spatial databases. ACM SIGMOD RECORD, 19(4): 61-68, 1990.

34. Günther, O. and Wong, E. A dual space representation for geometric data. In Proceeding of the 13th Very Large Data Base Conference, pp. 501-506, 1987.

35. Günther, O. and Bilmes, J. The implementation of the Cell tree: Design alternatives and performance evaluation. Technical Report of Coputer Science Dept. of University of California Santabarbara, TRCS88-23, 1988.

36. Günther, O. The Cell tree: An index for geometric databases. Technical Report of Coputer Science Dept. of University of California Santabarbara, TR-88-002, 1988.

37. Günther, O. The design of the Cell tree: An object-oriented index structure for geometric databases. In Proceedings of the IEEE 5th International Conference on Data Engineering, pp. 598-605, 1989.

38. Günther, O. and Riekert, W. F. Spatial database techniques for remote sensing. In Proceeding of the 3rd Spatial Data Handling Conference, pp. 961-970, 1990.

39. Günther, O. and Lamberts, J. Object-oriented techniques for the management of geographic and environmental data. The Computer Journal 37(1): 16-25, 1994.

40. Güttman, A. R-trees: A dynamic index structure for spatial searching. In Proceedings of the ACM SIGMOD, pp. 47-57, 1984.

41. Hinrichs, K. Implementation of the Grid file: Design concepts and experience. BIT 25, pp. 569-592, 1985.

42. Hoel, E., G. and Samet, H. Efficient processing of spatial queries in line segment databases. Lecture Notes in Computer science 525, Advances in Spatial Databases (SSD'91), pp. 237-256, 1991.

43. Hoel, E. G. and Samet, H. A qualitative comparison study of data structures for large segment databases. ACM SIGMOD, pp. 205-214, 1992.

44. Hutflesz, A., Six, H. W. and Widmayer, P. Globally order preserving multidimensional linear hashing. In Proceedings of the IEEE 4th International Conference on Data Engineering, pp. 572-579, 1988.

45. Hutflesz, A., Six, H. W. and Widmayer, P. The R-file: An efficient access structure for proximity queries. In Proceedings of the IEEE 6th International Conference on Data Engineering, pp. 372-379, 1990.

46. Ibbs, T. J. and Stevens, A. Quadtree storage of vector data. International Journal of Geographical Information Systems, 2(1): 43-56, 1988.

47. Jagadish, H. V. and O'Gorman, L. An object model for image recognition. IEEE Computer, December, pp. 33-41, 1989.

48. Jagadish, H. V. Linear clustering of objects with multiple attributes. In Proceedings of the ACM SIGMOD, pp. 332-342, 1990.

49. Jagadish, H. V. Spatial searching with polyhedra. In Proceeding of the 6th Data Engineering Conference, pp. 311-319, 1990.

50. Joseph, T. and Cardenas, A. PICQUERY: A high level query language for pictorial database management. IEEE Transaction on Software Engineering, 14(5): 630-638, 1988.

51. Kamel, I. and Faloutsos, C. Parallel R-trees. ACM SIGMOD, pp. 195-204, 1992.

52. Kasturi, R. and Alemany, J. Information extraction from image of paper-based maps. IEEE Transactions on Software Engineering 14(5): 671-675, 1988.

53. Kasturi, R. Fernandez, R. Amlani, M. L. and Feng, W. C. Map data processing in geographic information systems. IEEE Computer, December, pp. 10-21, 1989.

54. Kumar, A. G-tree: A new data structure for organizing multidimensional data. IEEE Transactions on Knowledge and Data Engineering, 6(2): 341-347, 1994.

55. Larson, P. Dynamic hashing. BIT 18: 184-201, 1978.

56. Lee, D. T. and Wong, C. K. Quintary trees: A file structure for multidimensional database systems. ACM Transactions on Database Systems, 5(3): 339-353, 1980.

57. Lee, J. T. and Belford, G. An efficient object-based algorithm for spatial searching, insertion and deletion. In Proceeding of the IEEE 8th Data Engineering Conference, pp. 40-47, 1992.

58. Li, J. Rotem, D. and Srivastava, J. Algorithms for loading parallel grid files. ACM SIGMOD pp. 347-356, 1993.

59. Litwin, W. Linear hashing: A new tool for file and table addressing. In Proceeding of the 6th IEEE International Conference on Very Large Data Base, pp. 212-223, 1980.

60. Lomet D. and Salzberg B. A robust multiple-attribute search structures. IEEE 5th International Conference on Data Engineering, pp. 296-304, 1989.

61. Lomet D. and Salzberg B. Access method for multiversion data. ACM SIGMOD, pp. 315-324, 1989.

62. Lomet, D. and Salzberg, B. The hB-tree: A multiple attribute indexing method with good guaranteed performance. ACM Transactions on Database Systems 15(4): 625-658, 1990.

63. Lomet, D. Grow and post index trees: Role, techniques and future potential. Lecture Notes in Computer science 525, Advances in Spatial Databases (SSD'91), pp. 183-206, 1991.

64. Lomet, D. A review of recent work on multiple-attribute access methods. ACM SIGMOD Record 21(3): 56-63, 1992.

65. Mark, D. M. Lauzon, J. P. and Cebrian, J., A. A review of Quad-tree-based strategies for interfacing coverage data with digital elevation models in grid form. International Journal of Geographical Information Systems 3(1): 3-14, 1989.

66. Nelson, R. C. and Samet H. A population analysis for hierarchical data structures. In Proceeding of the ACM SIGMOD Conference, pp. 270-277, 1987.

67. Nievergelt, J. and Hinterberger, H. The Grid file: An adaptable, symmetric multiple key file structure. ACM Transaction on Database Systems 9(1): 38 - 71, 1984.

68. Ohler, T. The multiclass Grid file: An access structure for multi class range queries. In Proceedings of the 5th Spatial Data Handling Conference, pp. 260-271, 1992.

69. Ohsawa, Y. and Sakauchi, M. The BD-tree- A new n-dimensional data structure with highly efficient dynamic characteristics. Information Processing, pp. 539-544, 1983.

70. Ohsawa, Y. and Sakauchi, M. A new tree type data structure with homogeneous nodes suitable for a very large spatial database. In Proceedings of the IEEE 6th international Conference on Data Engineering, pp. 296-303, 1990.

71. Ooi, B. C., Davis, R. S. and McDonell, K. J. Extending a DBMS for geographic applications. IEEE 5th International Conference on Data Engineering, pp. 590-597, 1989.

72. Ooi, B. C., Davis, R. S. and McDonell, K. J. Spatial indexing in binary decomposition and spatial bounding. Information Systems, 16(2): 211-237, 1991.

73. Oosterom, P. V. and Den Bos, J. V. An object-oriented approach to the design of geographic information systems. Computers & Graphics, 13(4): 409-418, 1989.

74. Oosterom, P. V. and Claassen, E. Orientation insensitive indexing methods for geometric objects. In proceeding of the 3rd Spatial Data Handling Conference, pp. 1016-1029, 1990.

75. Orenstein, J. A.. Spatial query processing in an object-oriented database system. ACM SIGMOD, pp. 326-336, 1986.

76. Orenstein, J. A. and Manola, F. A. PROBE spatial data modeling and query processing in an image database application. IEEE Transactions on Software Engineering, 14(5): 611-629, 1988.

77. Orenstein, J. A. Redundancy in spatial databases, ACM SIGMOD, pp. 294-305, 1989.

78. Orenstein, J. A. A comparison of spatial query processing techniques for native and parameter spaces. ACM SIGMOD, pp. 343-352, 1990.

79. Otoo, E. J. A multidimensional digital hashing scheme for files with composite keys. ACM SIGMOD, pp. 214-229, 1985.

80. Otoo, E. J. Linearizing the directory growth in order preserving extendible hashing. In Proceedings of the IEEE 4th international Conference on Data Engineering, pp. 580-588, 1988.

81. Otoo, E. J. An adaptive symmetric multidimensional data structure for spatial searching. In Proceeding of the 3rd Spatial Data Handling Conference, pp. 1003-1015, 1990.

82. Ousterhout, J. K. Hamachi, G. T.; Mayo, R. N.; Scott, W. S.; and Taylor, G. S. Magic: a VLSI layout system. IEEE 21st Design Automation Conference, pp. 152-159, 1984.

83. Pizano, A., Klinger, A. and Cardenas, A. Specification of spatial integrity constraints in pictorial databases. pp. 59-71, 1989.

84. Ramamohanrao, K. and Sacks-Davis, R. Recursive linear hashing. ACM Transactions on Database Systems 9(3): 369-391, 1984.

85. Robinson, J. T. The K-D-B tree: A search structure for large multiple-dimensional dynamic indexes. In Proceedings of the ACM SIGMOD Conference, pp. 10-18, 1981.

86. Roussopoulos, N. and Leifker, D. Direct spatial search on pictorial database using Packed R-tree. In Proceedings of the ACM SIGMOD Conference, pp. 17-31, 1985.

87. Roussopoulos, N. Faloutsos, C. and Sellis, T. An efficient pictorial database system for PSQL. IEEE Transactions on Software Engineering, 14(5): 639-649, 1988.

88. Sakauchi, M. and Ohsawa, Y. A new interactive geographical information system based on effective image-type map representation. Information Processing, pp. 95-100, 1983.

89. Samet, H. The Quadtree and related hierarchical data structures. Computing Surveys, 16(2): 187-260, 1984.

90. Samet, H. Storing a collection of polygons using quadtrees. ACM Transactions on Graphics, 4(3): 182-222, 1985.

91. Samet, H. The design and analysis of spatial data structures. Addison-wesley, Reading, MA, 1990.

92. Samet, H. Application of spatial data structures: Computer graphics, image processing, and GIS. Addison-wesley, Reading, MA, 1990.

93. Schneider, R. and Kriegel, H. Indexing the spatiotemporal monitoring of a polygonal object. In Proceeding of the 5th Spatial Data Handling Conference, pp. 200-209, 1992.

94. Sedgewick, R. Algorithms. Addison Wesley, Reading Massachusetts, 1983.

95. Sellis, T. Roussopoulos, T. and Faloutsos, C. R^+-tree: A dynamic index for multiple-dimensional objects. In Proceedings of the 13th VLDB Conference, pp. 507-518, 1987.

96. Shaffer, C. A. Samet, H. and Nelson, R. C. QUILT: A geographic information system based on quad-trees. International Journal of Geographical Information Systems 4(2): 103-131, 1990.

97. Shaffer, C. A. Large scale editing and vector to raster conversion via Quad-tree spatial indexing. In Proceedings of the 5th Spatial Data Handling Conference, 2, pp. 505-513, 1992.

98. Shen, H. Ooi, B. C. and Lu, H. The TP-Index: A dynamic and efficient indexing mechanism for temporal databases. IEEE 10th International Conference on Data Engineering, pp. 274-281, 1994.

99. Six, H. W. and Widmayer, P. Spatial searching in geometric databases. IEEE 4th International Conference on Data Engineering, pp. 496-503, 1988.

100. Smith, T. R. Experimental performance evaluations on spatial access method. In Proceedings of the 3rd Spatial Data Handling Conference, pp. 991-1002, 1990.

101. Tamminen, M. The extendible cell method for closest point problems. BIT, 22: 27-41, 1981.

102. Tamminen, M. THE EXCELL method for efficient geometric access to data. ACTA POLYTECHNICA SCANDINAVICA Mathmatics and Computer Science Series No. 34, 1981.

103. Tanaka, M. and Ichikawa, T. A visual user interface for map information retrieval based on semantic significance. IEEE Transactions on Software Engineering, 14(5): 666-670, 1988.

104. Unnikrishnan, A. Shankar, P. and Venkatesh, Y. V. Threaded linear hierarchical Quadtrees for computation of geometric properties of binary images. IEEE Transactions on Software Engineering, 14(5): 660-665, 1988.

105. Vijlbrief, T. and Oosterom, P. The GEO++ System: An extensible GIS. In Proceedings of the 5th Spatial Data Handling Conference, pp. 40-50, 1992.

106. Worboys M. F. A unified model for spatial and temporal information. The Computer Journal, 37(1): 26-34, 1994.

107. Xu, X., Han, J. and Lu, W. RT-tree: An improved R-tree index structure for spatiotemporal databases. In Proceeding of the 3rd Spatial Data Handling Conference, pp. 1040-1049, 1990.

108. Zhou, Y., Shekhar, S., and Coyle, M. Disk allocation methods for parallelizing Grid files. IEEE 10th International Conference on Data Engineering, pp. 243-252, 1994.

109. Zobrist, A. and Nagy, G. Pictorial information processing of landsat data for geographic analysis. IEEE Computer, November, pp. 34-41, 1981.

APPENDIX

TIGER/Line^{TM} files

The TIGER/Line^{TM} files are extracts of selected geographic and cartographic information from the Census Bureau's TIGER (Topologically Integrated Geographic Encoding and Referencing) database system [15]. The goal of the TIGER database system is to provide automated access to and retrieval of relevant geographic information about the United State and its territories. The TIGER/Line^{TM} files are organized on a county basis. Fourteen files typically make up the file set for a county. Each of the fourteen files has a unique code denoting the record type of the file. For example, in file name TGR40109.F51, 40 denotes the state code (Oklahoma state), 109 denotes the county code (Oklahoma City county) and F51 represents the record type. The size of the file sets for counties varies from less than 1 Mbytes to over 100 Mbytes. Various geographic features (e.g., roads, railroads, power lines, pipe lines, transportation features, hydrography, ... etc.) are represented by three types of spatial objects (e.g., points, lines and polygons) and these spatial objects are inter-related. The TIGER database uses these spatial objects to provide a disciplined, mathematical description of the earth's surface features. The spatial objects in the TIGER/Line^{TM} files incorporate both geometry (coordinate location and shape) and topology (the relationship between points, lines and polygons) [15]. In the Spatial Data Transfer Standard, nodes represent point objects that identify the start and end positions of lines and chains representing one-dimensional lines. In the TIGER/Line^{TM} files, chains are complete chains, since they construct the polygon boundaries, they also identify the polygon identification numbers and geographic entity codes for these polygons. The coordinates (longitude and latitude) of all complete chains for each county are stored in record type 1.

2

VITA

Kap S. Bang

Candidate for the Degree of

Doctor of Philosophy

Thesis: NEW EFFICIENT SPATIAL INDEX STRUCTURES, PML-TREE and SMR-TREE, FOR SPATIAL DATABASES

Major Field: Computer Science

Biographical:

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

Education: Graduated form Sungnam High School, Seoul, Korea in January 1979; received Bachelor of Science degree in Chemistry from Chung-Ang University in Seoul, Korea in February 1987; received Master of Science degree in Computer Science from Oklahoma State University, Stillwater, Oklahoma in May 1992. Completed the requirements for the Doctor of Philosophy with a major in Computer Science at Oklahoma State University in December 1995.

Professional Experience: Graduate Assistant, Department of Computer Science, Oklahoma State University, from August 1989 to December 1995.