

UNIVERSITY of SCIENCE & TECHNOLOGY

Kumasi Ghana

College of Science

School of Graduate Studies

**Optimizing the Dynamic Index Structure for Spatial Searching
Using R-Tree**

A Dissertation submitted in partial fulfillment of the
Requirements for the degree
Master of Philosophy
in
Computer Science
by


Aboagye Saim Philip

August 2012

Declaration

I, Aboagye Saim Philip, hereby declare that this dissertation, “Optimizing the Dynamic Index Structure for Spatial Searching Using R-Tree”, consists entirely of my own work produced from research undertaken under supervision and that no part of it has been published or presented for another degree elsewhere, except for the permissible excepts/references from other sources, which have been duly acknowledged.

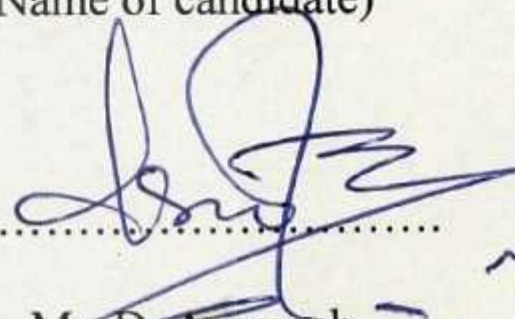
Date 15-03-2013

Sgd. 

Aboagye Saim Philip

(Name of candidate)

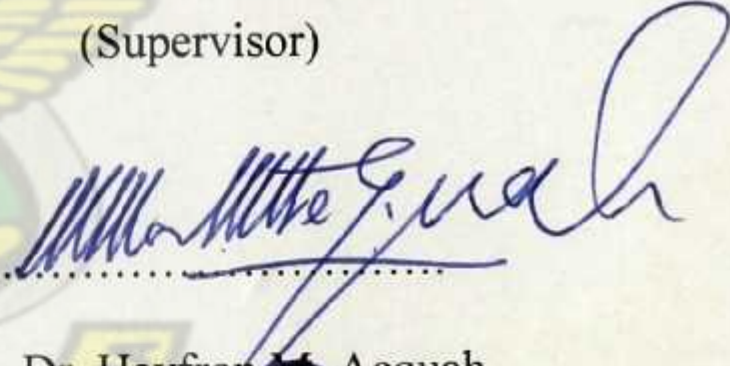
Date 15-03-13.

Sgd. 

Mr. D. Asamoah

(Supervisor)

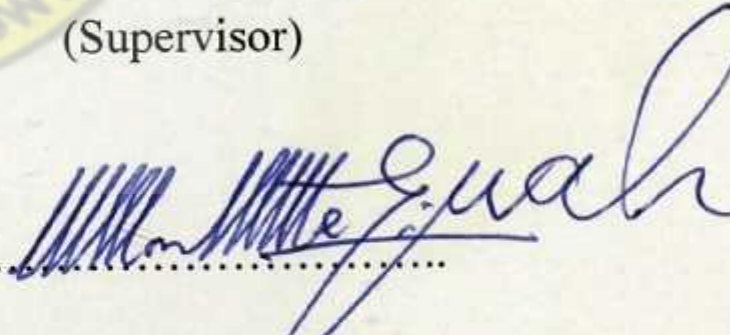
Date 2/4/13

Sgd. 

Dr. Hayfron Acquah

(Supervisor)

Date 2/4/13

Sgd. 

Dr. Hayfron Acquah

“Head of Department”

August 2012

Optimizing the Dynamic Index Structure for Spatial Searching Using R-Tree

Copyright © 2012

by

Aboagye Saim Philip

UNIVERSITY of SCIENCE & TECHNOLOGY

Kumasi Ghana



Dedication

This research is dedicated to my sweet mother Harriet Brew, my children: Reinhard Nana Kwame Asare Saim, Harriet Esi Moasiwah Saim, Rhena Efua Smith Saim and Brew Kwesi Andoh Saim, my dearest wife Dina Dugan a.k.a Yaaba who for their prayers, love and support had brought me to this far.

Whatever you might have lost because of my achievement will never be in vain.

May God richly bless you and may you live long to enjoy the fruits of your labour.



Acknowledgement

First, I would like to thank the board that admitted me into this university to undertake this research. I would also like to thank my brother Joseph Ntiamoah Saim and my sister Constance Obresi Saim for all the financial support needed to complete this research. May God richly bless them.

My greatest thanks go to my supervisor Mr. Dominic Asamoah for his invaluable role in the creation of this work. I was largely inspired by his deep insight, extensive knowledge, and talent of seeing the big picture. I am very grateful for his time and pain for constructive criticism, correction and offered many suggestions and confirming the ideas incorporated in this research.

I am also indebted to Dr. Hayfron M. Acquah my second supervisor who was always there and offered many help when the need arose.

I wish to acknowledge all the references/sources cited for I have learnt a lot from them and gave me insight of how to plan my research.

My special thanks go to my colleagues William Boateng (Willie Boat), without him I wouldn't had been to this institution to pursue this degree and Ernest Duker Jnr. who taught me all the intricacies of java programming language to finish this work. Another colleague I would like to thank is Chief Superintendent Samuel Alfred Winful who introduce me to the world of database and data structures.

Finally, I would like to thank my dear wife D. Dugan and my loving family for their constant support and understanding throughout my study. To God be all the glory for the success of this research.

Abstract

In recent years, there has been an upsurge of interest in spatial databases. A major issue is how to efficiently manipulate massive amounts of spatial data stored on disk in multidimensional spatial indexes (data structures). Construction of spatial indexes has been studied intensively in the database community. The continuous arrival of massive amounts of new data makes it important to efficiently update existing indexes. This dissertation is concerned with optimizing the dynamic index structure for spatial search, a class of data structure that organizes multidimensional data in database systems. The R-tree, one of the most popular access methods for spatial data, is based on the heuristic optimization of the area of the enclosing rectangles in each inner node. The method proposed in this thesis is based on the technique of geometry, area and overlap optimization of each enclosing rectangle in the directory, and by running several tests and comparing with other R-tree variants, the new optimized R-tree is efficient in terms of disk storage, internal computation time and number of pages stored in memory. It also produces a better quality index in terms of query performance. One important novel feature of this technique is that in most cases it allows updates and queries to be performed simultaneously in environments where queries have to be answered even while the index is being updated and reorganized. Although the optimized R-tree outperforms its competitors, the cost for the implementation of the structure is only slightly higher than the other R-trees. Notwithstanding the benefit of this research, there are still more to be done in topology because in real world spatial objects are more static. It must therefore follow an engineering approach to improve complex applications.

Table of Contents

List of Tables xi

List of Figures xii

List of Acronyms xiii

1. Chapter One: Introduction 1

1.1 Introduction 1

1.2 Statement of Problems 3

1.3 Objectives of the Proposed Research 5

1.4 Hypothesis 6

1.5 Methodology of Study 6

1.6 Justification of the Study 8

1.7 Scope of the Study 8

2. Chapter Two: Literature Review 10

2.1 Introduction 10

2.1.0 Review of Relevant Literature 10

2.1.1 The Review of B-Trees in file System 10

2.1.2 The Structure of B-Tree 12

2.1.3 Searching, Inserting and Deleting in B-Tree 16

2.1.4 B-Trees in File and Database System 19

2.1.5 Some Variations in B-Trees 22

2.2. R-Tree 24

2.2.1 Operation on R-Trees 27

2.2.2	Visualizing R-Tree in 3-Dimension (3-D)	32
2.2.3	Variations in R-Tree	33
2.2.3.1	The R+ Tree	33
2.2.3.2	The R* Tree	35
2.2.3.3	The Hilbert R-Tree	35
2.2.3.4	Linear Node Splitting	36
2.2.3.5	Optimal Node Splitting	36
2.2.3.6	Branch Grafting	38
2.2.3.7	Compact R-Tree	42
2.2.3.8	Priority R-Tree	43
2.2.3.9	Deviating Variations	44
2.3.	Static Versions of R-Tree	46
2.3.1	The Packed R-Trees	46
2.3.2	The Hilbert Packed R-Trees	46
2.3.3	Small-Tree-Large-Tree	47
2.3.4	Buffer R-Trees	48
2.2.1	Trajectory Buddle-Tree (TB-Tree)	48
2.3.	Summary and Conclusion	48

3. Chapter Three: Methodology 50

3.0	Introduction	50
3.1	Study Area	50
3.2	Design of Study	51

3.3	The Principles of R-Tree	53
3.4	The Optimized R-Tree	57
3.4.1	Algorithms for Optimized R-tree	58
3.4.2	Node Splitting	58
3.5	Constraints/Problems	62
4.	Chapter Four: Analysis of Findings	63
4.0	Introduction	63
4.1	Variable m and M	64
4.2	Empirical Result	66
4.2.1	R-Tree Variants	66
4.2.2	Analyzing the Quadratic Split Algorithm.	69
4.2.3	Analyzing Greene's Split Algorithm	71
4.2.4	Analyzing the Optimized R-Tree Algorithm	71
4.3	Performance Tests	72
4.3.1	Results of Inserting Data.	73
4.3.2	Results of Searching.	75
4.3.3	Results of Deleting.	76
4.4	Performance Comparisons	77
5.	Chapter Five: Conclusion And Recommendation	82
5.0	Summary	82
5.1	Conclusion	83

5.2 Problems	84
5.3 Recommendations.....	84
Bibliography	85

KNUST



List of Figures

1	Sample of B-Tree Data Structure	13
2	Data Storage of B-Tree Data Structure	13
3	Internal Node of B-Tree	14
4	Inserting into B-Tree of order two.	18
5	Comparing Binary Search Tree with B-Tree.	20
6	Spatial Object of R-Tree	25
7	A representation of R-Tree	30
8	R+-Tree Data Structure	34
9	Branch Grafting Techniques	39
10	Overfilled Node with Quadratic Splits	56
11	Variable M and m	64
12	Underflow of R-Tree	65
13	Overflow of R-Tree	65
14	CPU Cost of Inserting Data into R-Tree	74
15	CPU Cost of Search Performance	75
16	CPU Cost of Deleting Record	76

List of Tables

1 Performance Test Using page size	74
2 Building up R-Tree using Uniform	80
3 Building up R-Tree using Cluster	80
4 Building up R-Tree using Mixed Uniform	81

KNUST



List of Acronyms

1 B-Tree:	Balanced Tree.....	4
2 R-Tree:	Rectangular Tree.....	4
3 MBR:	Minimum Bounding Rectangle.....	5
4 Lin Gut:	Gutmann Linear Split.....	80
5 Qua Gut:	Gutmann Quadratic Split.....	80
6 Optz:	Optimized R-Tree.....	80



CHAPTER ONE

1.1 Introduction

Tree structure is one of the best recipe in data structures for database and it's applicable to organization charts and files systems. Since its inception, it has contributed to how data are structured and indexed in databases. Notwithstanding the fact that B-Trees (Balanced Tree) which came to strengthen Binary Tree in data structures still had limitations with respect to spatial location of data which had more attributes.

In database application where data had more attributes, enquiry into such multi-dimensional data objects is difficult when classical indexing structure is used and a typical example is that of a Spatial and Multimedia Databases.

Spatial databases provide concepts for databases that keep track of object in a multidimensional space. For example, cartographic databases that store maps includes two dimensional space description of their objects-from cities and towns to rivers, seas, roads and so on. These applications are known as Geographical Information Systems (GIS), and are used in areas such as environmental, emergency and battle management. Others such as meteorological information for weather forecasting is based on spatial point

Multimedia databases provide features that allow users to store and query different types of multimedia information, which includes images (such as photos or drawings), video clips (such as movies, newsreels or home videos) and audio clips. The main types of database queries that are needed involve locating multimedia sources that contain certain objects of interest. A typical image database query would be to find images in the database that are similar to a given image.

The given image could be an isolated segment that contain, say a pattern of interest and the query is to locate other images that contain the same pattern.

In general, a spatial database store objects that have spatial characteristics that describes them. The spatial relationship among the objects are important, and they are often needed when querying the database.

The main extensions that are needed for spatial databases are models that can interpret spatial characteristics. Additionally, special indexing and storage structures are often needed to improve performance.. The basic extension needed include two-dimensional geometric concepts, such as points, lines segments, circles, polygons and arcs, in order to specify the spatial characteristics of the objects. Also, spatial operation are needed to operate on the objects, for example, to compute the distance between two objects and as well check whether these two objects spatially overlapped. Some of these objects generally have static spatial characteristics, such as streets and highways, police stations, fire stations, hospitals, water pumps (for fire control) etc. Other objects have dynamic spatial characteristics that change over time and it includes police vehicles, ambulances or fire trucks.

There are a number of ways by which the above scenarios could be achieved and these include:

Range query-where we find the objects of a particular type that are within a given spatial area or within a particular distance from a given location. For example, find all hospital within a given city area or find all ambulances within five miles from accident locations.

Nearest Neighbour Query-here, an object of a particular type that is closest to a given location is found. A typical example could be finding all police car that is closest to a particular location.

Spatial Join or Overlay-Typically joins the objects of two types based on their spatial condition, such as the objects intersecting or overlapping spatially or being within a certain location of one another. An example is to find all cities located on a major highway or find all homes that are within three miles from a lake.

However, for these and other types of spatial queries to be achieved efficiently, special techniques for spatial indexing are needed. One of the best known techniques is the use of R-Trees and their variations, which was initially proposed by **Antoine Guttman**.

R-Trees group objects that are in close spatial physical proximity on the same leaf nodes of a tree structured index. Since a leaf node can point to a certain number of objects only, algorithms for dividing the space into rectangular subspaces that include the objects are needed. Typical criteria for dividing the space include minimizing the rectangle areas, since this would lead to a quicker narrowing the search space. Problems such as having objects with overlapping spatial areas would be handled differently by the many different variations of R-Trees. Internal nodes of R-Trees are associated with rectangles whose areas covers all the rectangles in its **SubTree**. Hence, R-Trees can easily answer queries, such as find all objects in a given area by limiting the tree search to those subtrees whose rectangle intersect with the area given in the query.

Other spatial storage would include quad trees and their variations. Quadtrees generally divide each space or subspace into equally sized areas, and proceed with the subdivisions of each subspace to identify the positions of various objects. Hence, there is the need to go further into this spatial access structures to improve on the existing R-Trees.

1.2 Statement of problems

In our modern application world the using of spatial index structure for indexing multi-dimensional information according to their special location is enormous. To handle spatial data efficiently a database system needs a special index mechanism. Investigation shows that Contemporary Multi-dimensional database technology is severely limited at managing indexed data types of keys for many advanced application. One of the huge demands in geo-data i.e. geographical data applications is to response very quickly to spatial inquiry, for example, "finding all supermarkets within 2 km of my current location". Spatial data objects often cover areas in multi-dimensional spaces. Further Investigation shows that multi-dimension data prevents using classical indexing structures, for instance the **B-Tree** (Kemper, 2001). The reason is that database use one-dimensional indexing structures. However, in modern information processing like CAD (Computer Aided Design), cartography and multimedia applications use a multi-dimensional data object, which means that the objects have more attributes. Thus, the database system needs an efficient multi-dimensional index structure.

A number of structures had been proposed for handling multi-dimensional point data. **Antoine Guttman** was one of the first persons to propose them. In 1984, Guttman published a book (Guttman, 1984) in which he presented a data structure called R-Tree (Rectangle Tree) that represents data objects by intervals in several dimensions.

Others included the following:

- **Priority R-Tree, or PR-Tree**, (Herman J. et al., 2001) which is the first R-Tree variant that was practically efficient and worst-case optimal R-Tree, was also based on area optimization.

- **R+ Tree:** (T. Sellis et al., 1987). This tree tries to minimize the overlapping of regions. Here, objects are saved disjunctive. The search algorithm is faster but the tree structure is more complicated.
- **R* Tree:** (M. Astrahan et al., 1976). The Split-Node algorithm that takes volume and the extend of overlapping into consideration, but faster for spatial and point access queries. All other properties are similar to the R-Tree.

An optimized elaborative retrieval method to extend the traditional R-Tree, can serve as an indexing structure in some multi-dimensional database systems. This method will try to addresses the problem of the traditional R-Tree where the only criterion is to minimize the area of index coordinate data.

1.3 Objectives of the proposed research

The main or specific objectives of this research are to address the following:

1. To eliminate or improve large distances of data types which will initiate a bad split. According to the spatial location of data, the structure is designed in such a way that a spatial search requires visiting only a small number of nodes.

The spatial data is comprised by a MBR (Minimal Bounding Rectangle) which become formatted and comprised from a MBR again. Therefore, there is the need to eliminate large distances of data types to have a better split.

2. To develop a method that will take into consideration the rest of assigned nodes by taking their geometry after a group data type has reached a maximum number

of entries. This is very important so that the tree structure is maintained and balanced.

1.4 Hypothesis

In an effort to trying to address the above problems I put across the following Hypothesis.

H1: The decision whether to visit a node depends on whether its covering rectangle overlaps the search area, the total area of the two covering rectangles after a split should be minimized.

H2: In order to add a new entry to a full node containing M entries, it is necessary to divide the collection of M+1 entries between two nodes so that I will have the best split for the search space.

These Hypotheses would be discussed and fully analyzed in the methodology.

1.5 Methodology of the Study

A lot of research had already been conducted on R-Tree. Antoine Guttman was one of the first persons to propose them. However, there are still more to be done on the algorithms used.

The method to use will examine in detail the algorithms listed below, which would be written in pseudo-code, and the program implemented in Java.

In the pseudo-code the rectangle parts of an index entry E would be denoted by E:I and the tuple-identifier or child - pointer part would be denoted by E:p.

The algorithms to be considered includes the following:

- Searching
- Insertion
- Deletion
- SplitNode

these will be compared with B-Tree and other forms of tree data structures.

For example: In **searching**, the search algorithm is similar to that of the B-Tree. It returns all qualifying records, which the search rectangle overlaps. The algorithm transverse the tree from the root. In the same time, the algorithm checks the rectangle overlapping in the node with the searched rectangle. If the test is positive, the search just descends to the found overlapping nodes. This procedure is repeated until the leaf node overlaps. If the entries of the leaf node overlap the searched rectangle then return these entries as a qualifying record.

For **inserting** index records for new data is similar to inserting into a B-Tree. New data is added to the leaves, nodes that overflow are split, and splits propagate up the tree.

Deletion in R-Trees is different from deletion in B-Tree. This is so because after deletion, the tree had to be condensed.

Split Note: In the case of adding a new entry to a full node containing n number of entries, it is necessary to divide the collection of $n+1$ entry between two nodes and insertion and deletion method can be adopted to save the tree structure from coalescing. The division should be done in such a way that both new nodes will need to be checked on subsequent searches.

The problems associated with searching and inserting algorithms would also be discussed.

1.6 Justification of the Study

Though, a lot of modifications on R-Tree had been made in recent years specifically structure, there was specialization for particular application. So, in this research I will try to develop an algorithm to optimize the dynamic index structure of R Tree to improve or eliminate large distances of data types which will initiate a bad split of nodes.

This work when completed will result in the development of a new algorithm that will bring to knowledge the best splitting algorithm that will maintain the tree after a group data type has reached a maximum number of entries. Also, it will highlight the minimization of the overlaps of the MBRs to determine Node Splitting, unlike the original R-tree which considers only the minimization of the enclosing rectangle.

1.7 Scope of the Study

An R-Tree is a height-balanced tree similar to a B-Tree. Leaf nodes contain pointers to data objects. The index is completely dynamic. Structure is designed in such a way that a spatial search requires visiting only a small number of nodes. The spatial data is comprised by a MBR (Minimal Bounding Rectangle) which become formatted and comprised from a MBR again. This structure continues up to the root. Eventually the root comprises a MBR over all objects.

The maximum coordinates to use is 300, maximum width: 60, maximum number of Rectangle to use is 100 with three dimension.

The research seeks to do in depth analysis of the following:

- The review of B Trees in file systems.
- R Tree and B Trees in spatial data
- Visualization of R Tree for 3D Cubes
- Difference between R*-Tree, R+-Tree, M-Tree, Priority-Tree, etc and R-Tree
- Structure of R Tree
- R Tree and the usage of memory page disk.



CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

Spatial databases provide concepts for databases that keep track of object in a multidimensional space. And for spatial it means the data have more attributes that spread across wide geographical area and spatial database store objects that have spatial characteristics that describes them. The spatial relationships among the objects are important, and they are often needed when querying the database. The purpose of this review is the inception of various researches that tries to solve spatial enquiries using various methods for which R-Tree is no exception. On this, B-Tree will be reviewed with reference to Binary Tree, searching, insertion and deleting methods, file systems and database systems and its associated problems. The development of R-Tree as a prerequisite to solving spatial data in multidimensional space, other types of tree structure that support R-Tree and the usage of disk space provided by R-Tree.

2.1.0 Review of Relevant Literature

2.1.1 The Review of B-Trees in File Systems

In computer science, a **B-Tree** is a tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic amortized time. The B-Tree is a generalization of a binary search tree in that a node can have more than two children. Comer et al., (1979) unlike self-balancing binary search trees, the B-Tree is optimized for systems that read and write large blocks of data. It is commonly used in databases and filesystems

In B-Trees, internal (non-leaf) nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-Trees do not need re-balancing as frequently as other self-balancing search trees (Binary Search Tree), but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3 B-Tree (often simply referred to as a **2-3 tree**), each internal node may have only 2 or 3 child nodes.

Each internal node of a B-Tree will contain a number of keys. Usually, the number of keys is chosen to vary between d and $2d$. In practice, the keys take up the most space in a node. The factor of 2 will guarantee that nodes can be split or combined. If an internal node has $2d$ keys, then adding a key to that node can be accomplished by splitting the $2d$ key node into two d key nodes and adding the key to the parent node. Each split node has the required minimum number of keys. Similarly, if an internal node and its neighbour each have d keys, then a key may be deleted from the internal node by combining with its neighbour. Deleting the key would make the internal node have $d - 1$ keys; joining the neighbour would add d keys plus one more key brought down from the neighbour's parent. The result is an entirely full node of $2d$ keys.

The number of branches (or child nodes) from a node will be one more than the number of keys stored in the node. In a 2-3 B-Tree, the internal nodes will store either one key (with two child nodes) or two keys (with three child nodes). A B-Tree is sometimes described with the parameters $(d + 1) - (2d + 1)$ or simply with the highest branching order, $(2d + 1)$.

A B-Tree is kept balanced by requiring that all leaf nodes are at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more node further away from the root.

B-Trees have substantial advantages over alternative implementations when node access times far exceed access times within nodes (Folk et al., 1992). This usually occurs when the nodes are in secondary storage such as Hard Drives. By maximizing the number of child nodes within each internal node, the height of the tree decreases and the number of expensive node accesses is reduced. In addition, rebalancing the tree occurs less often. The maximum number of child nodes depends on the information that must be stored for each child node and the size of a full disk block or an analogous size in secondary storage. While 2-3 B-Trees are easier to explain, practical B-Trees using secondary storage want a large number of child nodes to improve performance.

2.1.2 The Structure of B-Tree

Unlike a binary-tree, each node of a B-Tree may have a variable number of keys and children. The keys are stored in non-decreasing order as shown in the figure below. Each key has an associated child that is the root of a subtrees containing all nodes with keys less than or equal to the key but greater than the preceding key. A node also has an additional rightmost child that is the root for a subtrees containing all keys greater than any keys in the node.

Sample B-Tree

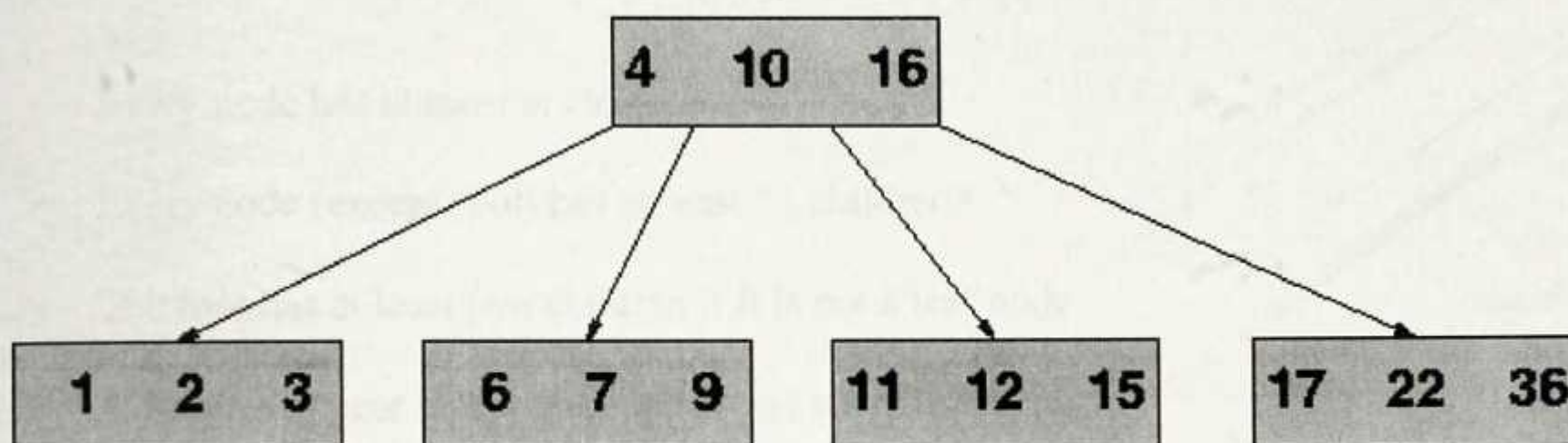


Figure 1

Data storage in B-Tree structure as shown below: d_1 , d_2 , d_3 , d_4 , d_5 , d_6 and d_7

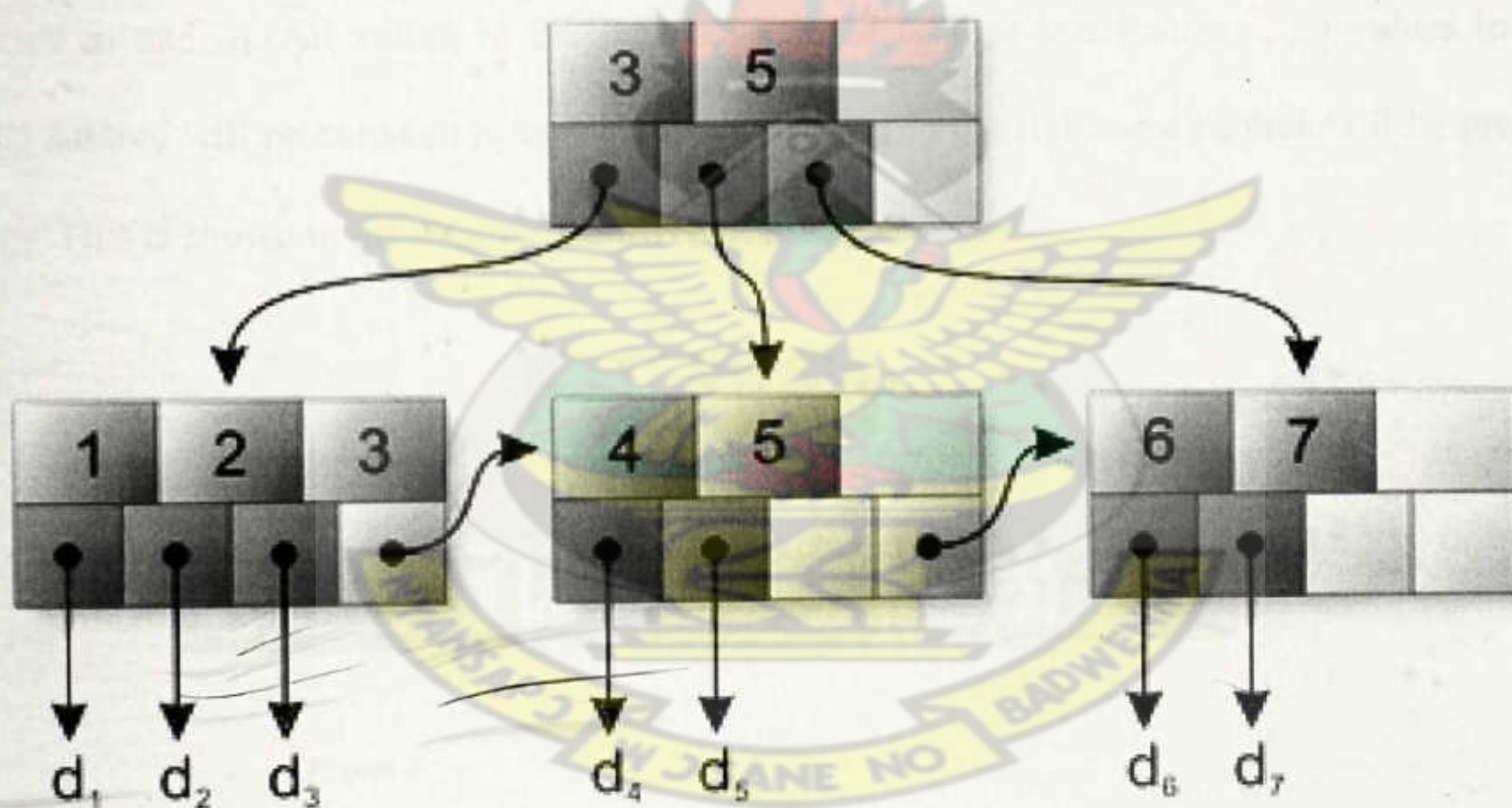


Figure 2

According to Knuth's definition (Knuth Donald 1997), a B-Tree of order m (the maximum number of children for each node) is a tree which satisfies the following properties:

- Every node has at most m children.
- Every node (except root) has at least $\lceil m/2 \rceil$ children.
- The root has at least two children if it is not a leaf node.
- All leaves appear in the same level, and carry information.
- A non-leaf node with k children contains $k-1$ keys.

Each internal node's elements act as separation values which divide its subtrees. For example, if an internal node has three child nodes (or subtrees) then it must have two separation values or elements a_1 and a_2 . All values in the leftmost subtree will be less than a_1 , all values in the middle subtree will be between a_1 and a_2 , and all values in the rightmost subtree will be greater than a_2 . This is shown in the diagram below:

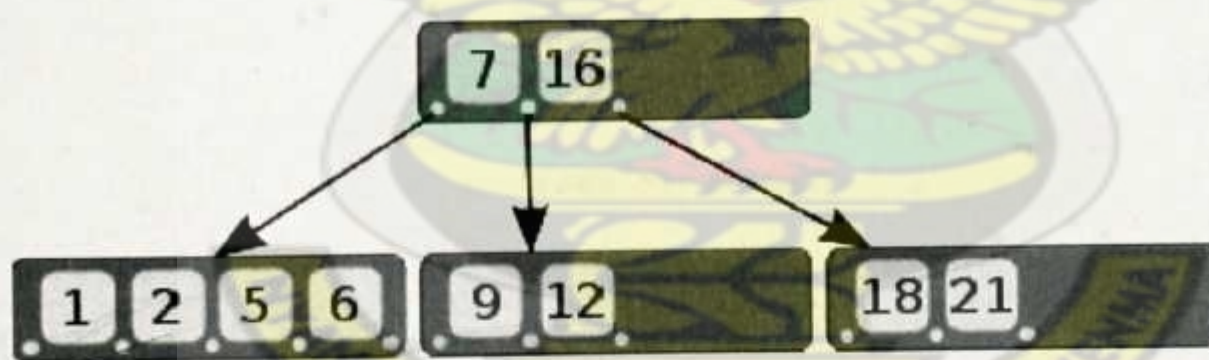


Figure 3

A B-Tree has a minimum number of allowable children for each node known as the *minimization factor*. If t is this *minimization factor*, every node must have at least $t - 1$ keys. Under certain circumstances, the root node is allowed to violate this property by having fewer than $t - 1$ keys. Every node may have at most $2t - 1$ keys or, equivalently, $2t$ children.

Since each node tends to have a large branching factor (a large number of children), as seen in the figure below, it is typically necessary to traverse relatively few nodes before locating the desired key. If access to each node requires a disk access, then a B-Tree will minimize the number of disk accesses required. The minimization factor is usually chosen so that the total size of each node corresponds to a multiple of the block size of the underlying storage device. This choice simplifies and optimizes disk access. Consequently, a B-Tree is an ideal data structure for situations where not all data can reside in primary storage and accesses to secondary storage are comparatively expensive (or time consuming).



Tree Structure

According to (Rivest et al., 1991), for n greater than or equal to one, the height of an n -key B-Tree T of height h with a minimum degree t greater than or equal to 2, is given by

$$h \leq \log_t \frac{n+1}{2}$$

Where h is the height of the tree,

and n is the number of nodes.

The best case height of a B-Tree is:

$$\log_m n.$$

The worst-case height of a B-Tree is:

$$\log_{m/2} n$$

Where m is the maximum number of children a node can have.

Since the "branches" of a B-Tree can be large compared to many other balanced tree structures, the base of the logarithm tends to be large; therefore, the number of nodes visited during a search tends to be smaller than required by other tree structures. Although this does not affect the asymptotic worst case height, B-Trees tend to have smaller heights than other trees with the same asymptotic height.

2.1.3 Searching, Inserting and Deleting in B-Tree

Searching is similar to searching a binary search tree. Starting at the root, the tree is recursively traversed from top to bottom. At each level, the search chooses the child pointer (subtree) whose separation values are on either side of the search value. Binary search is typically (but not necessarily) used within nodes to find the separation values and child tree of interest. Instead of choosing between a left and a right child as in a binary tree, a T-Tree search must make an n-way choice. The correct child is chosen by performing a linear search of the values in the node. After finding the value greater than or equal to the desired value, the child pointer to the immediate left of that value is followed. If all values are less than the desired value, the rightmost child pointer is followed. Of course, the search can be terminated as soon as the desired node is found.

Since the running time of the search operation depends upon the height of the tree, *B-Tree-Search* is $O(\log_t n)$.

Inserting in B-Tree is not the same as binary tree. According to Kruse in (Kruse et al., 1991), to perform an insertion on a B-Tree, the appropriate node for the key must be located using an algorithm similar to *B-Tree-Search*. Next, the key must be inserted into the node. If the node is not full prior to the insertion, no special action is required; however, if the node is full, the node must be split to make room for the new key. Since splitting the node results in moving one key to the parent node, the parent node must not be full or another split operation is required. This process may repeat all the way up to the root and may require splitting the root node. This approach requires two passes. The first pass locates the node where the key should be inserted; the second pass performs any required splits on the ancestor nodes.

Since each access to a node may correspond to a costly disk access, it is desirable to avoid the second pass by ensuring that the parent node is never full. To accomplish this, the presented algorithm splits any full nodes encountered while descending the tree. Although this approach may result in unnecessary split operations, it guarantees that the parent never needs to be split and eliminates the need for a second pass up the tree. Since a split runs in linear time, it has little effect on the big O -Notation, i.e. $O(t \log_t n)$ running time of *B-Tree-Insert*.

Splitting the root node is handled as a special case since a new root must be created to contain the median key of the old root. Observe that a B-Tree will grow from the top.

Below is an example of inserting in a B-Tree of order two.

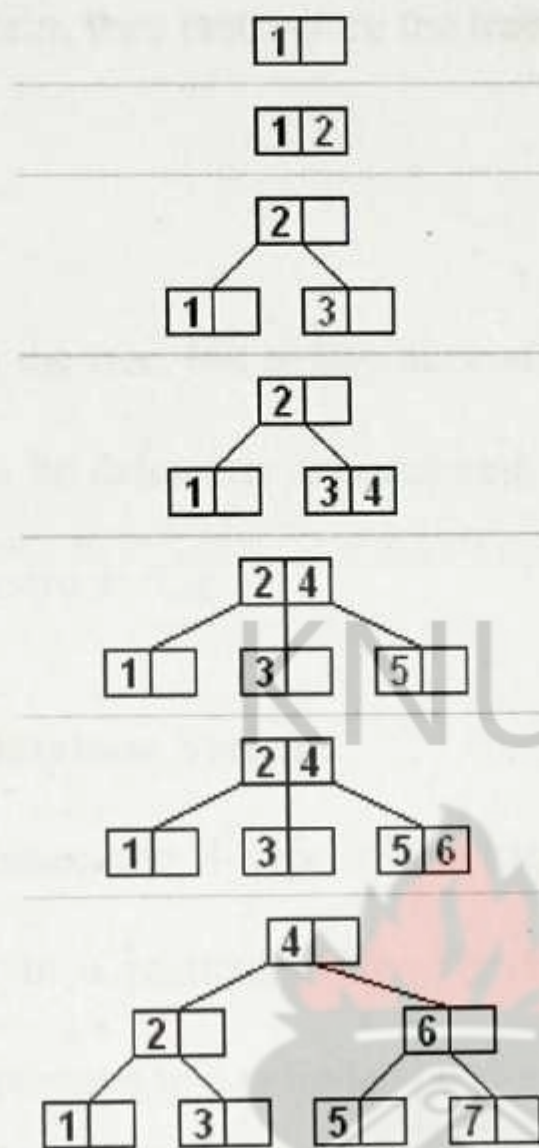


Fig: 4: A B-Tree insertion example with each iteration.

Note that all insertions start at a leaf node.

Deletion of a key from a B-Tree is possible; however, special care must be taken to ensure that the properties of a B-Tree are maintained. Several cases must be considered. If the deletion reduces the number of keys in a node below the minimum degree of the tree, this violation must be corrected by combining several nodes and possibly reducing the height of the tree. If the key has children, the children must be re-arranged (Rivest, *ibid*).

There are two popular strategies for deletion from a B-Tree.

- locate and delete the item, then restructure the tree to regain its invariants

Or

- do a single pass down the tree, but before accessing (visiting) a node, restructure the tree so that once the key to be deleted is encountered, it can be deleted without triggering the need for any further restructuring

2.1.4 B-Trees in File and Database Systems

In addition to its use in databases, the B-Tree is also used in filesystems to allow quick random access to an arbitrary block in a particular file. The basic problem is turning the file block address into a disk block (or perhaps to a cylinder-head-sector) address.

Some operating systems require the user to allocate the maximum size of the file when the file is created. The file can then be allocated as contiguous disk blocks. Converting to a disk block: the operating system just adds the file block address to the starting disk block of the file. The scheme is simple, but the file cannot exceed its created size. Other operating systems allow a file to grow. The resulting disk blocks may not be contiguous, so mapping logical blocks to physical blocks is more involved.

B-trees are preferred when decision points, called nodes, are on **hard disk** rather than in random-access memory (**RAM**). It takes thousands of times longer to access a data element from hard disk as compared with accessing it from RAM, because a disk drive has mechanical parts, which read and write data far more slowly than purely electronic media. B-trees save time by using nodes with many branches (called children), compared with binary trees, in which each node has

only two children. When there are many children per node, a record can be found by passing through fewer nodes than if there are two children per node. A simplified example of this principle is shown below.

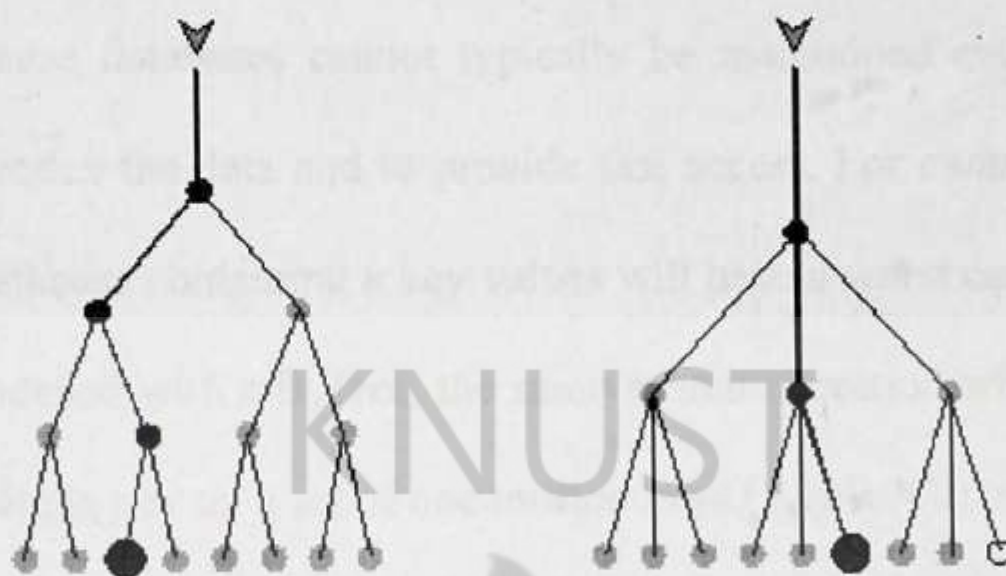


Figure 5

In a tree, records are stored in locations called leaves. This name derives from the fact that records always exist at end points (as shown in fig 1 above); there is nothing beyond them. The maximum number of children per node is the order of the tree. The number of required disk accesses is the depth. The image at left shows a binary tree for locating a particular record in a set of eight leaves. The image at right shows a B-tree of order three for locating a particular record in a set of eight leaves (the ninth leaf is unoccupied, and is called a null). The binary tree at left has a depth of four; the B-tree at right has a depth of three. Clearly, the B-tree allows a desired record to be located faster, assuming all other system parameters are identical. The tradeoff is that the decision process at each node is more complicated in a B-tree as compared with a binary tree. A sophisticated program is required to execute the operations in a B-tree. But this program is stored in RAM, so it runs fast.

It is not uncommon for a database to contain millions of records requiring many gigabytes of storage. For examples, TELSTRA, an Australian telecommunications company, maintains a customer billing database with 51 billion rows (yes, billion) and 4.2 terabytes of data. In order for a database to be useful and usable, it must support the desired operations, such as retrieval and storage, quickly. Because databases cannot typically be maintained entirely in memory, B-Trees are often used to index the data and to provide fast access. For example, searching an un-indexed and unsorted database containing n key values will have a worst case running time of $O(n)$; if the same data is indexed with a B-Tree, the same search operation will run in $O(\log n)$. To perform a search for a single key on a set of one million keys (1,000,000), a linear search will require at most 1,000,000 comparisons. If the same data is indexed with a B-Tree of minimum degree 10, 114 comparisons will be required in the worst case. Clearly, indexing large amounts of data can significantly improve search performance. Although other balanced tree structures can be used, a B-Tree also optimizes costly disk accesses that are of concern when dealing with large data sets.

According to (Bayer R. M. et al., 1994), databases typically run in multiuser environments where many users can concurrently perform operations on the database, a B-Tree suffers from similar problems in a multiuser environment. If two or more processes are manipulating the same tree, it is possible for the tree to become corrupt and result in data loss or errors.

In an attempt to address the above problems (Gray 1994) introduced serialize access to the data structure. In other words, if another process is using the tree, all other processes must wait. Although this is feasible in many cases, it can place an unnecessary and costly limit on performance because many operations actually can be performed concurrently without risk.

Locking, introduced by Gray and refined by many others, provides a mechanism for controlling concurrent operations on data structures in order to prevent undesirable side effects and to ensure consistency.

2.1.5 Some Variations in B-Trees

According to Shaffer in (Clifford A. Shaffer 1997), the B-tree and variations on it are commonly used in large commercial databases to provide quick access to the data. In fact, he says that they are "the standard file organization for applications requiring insertion, deletion, and key range searches". The variant called the B+ tree is the usual one. Another variant is the B* tree, which is very similar to the B+ tree, but tries to keep the nodes about two-thirds full at a minimum.

In a B+ tree, data records are only stored in the leaves. Internal nodes store just keys. These keys are used for directing a search to the proper leaf. If a target key is less than a key in an internal node, then the pointer just to its left is followed. If a target key is greater or equal to the key in the internal node, then the pointer just to its right is followed. The leaves are also linked together so that all of the keys in the B+ tree can be traversed in ascending order, just by going through all of the nodes in this linked list along the bottom level of the tree.

When a B+ tree is implemented on disk, it is likely that the leaves contain key, pointer pairs where the pointer field points to the record of data associated with the key. This allows the data file to exist separately from the B+ tree, which functions as an "index" giving an ordering to the data in the data file. This is how B+ trees are used in a database. Of course, the pointers are record numbers, our typical fake pointers used when creating dynamic data structures on disk.

Notice that this B+ tree indexing scheme allows one data file to have several such indices, each giving an ordering by a different key field, something highly desirable to have.

As an example, consider a B+ tree of order 200, whose leaves can each contain up to 199 keys (approximately 200). Let's assume that the root node has at least 100 children (though we know it is allowed to have as few as two). A 2 level B+ tree that meets these assumptions can store up to about 10,000 records, since there are at least 100 leaves, each containing at least 99 keys (approximately 100). A 3 level B+ tree of this type can store up to about 1 million keys. A 4 level B+ tree can store up to about 100 million keys. To get faster access to the data, the root node is commonly kept in main memory. Maybe even the child nodes of the root can fit in main memory. Thus one can find one of 100 million keys with only 2 or 3 disk reads. If, as is common, the associated data records are stored in a separate file, there is one additional read to get the data associated with a key. Also, note that if the root node has fewer than our assumed 100 children, this slows down the lookup further.

In a practical B-Tree, there can be thousands, millions, or billions of records. Not all leaves necessarily contain a record, but at least half of them do. The difference in depth between binary-tree and B-Tree schemes is greater in a practical database than in the example illustrated here, because real-world B-Trees are of higher order (32, 64, 128, or more). Depending on the number of records in the database, the depth of a B-Tree can and often does change. Adding a large enough number of records will increase the depth; deleting a large enough number of records will decrease the depth. This ensures that the B-Tree functions optimally for the number of records it contains.

Though B-Tree is a data structure for storing sorted data with amortized run times for insertion and deletion often used for data stored on long latency I/O (filesystems and DBs) because child nodes can be accessed together (since they are in order) and provided a foundation for R-Trees, it cannot store new types of data specifically geometrical data and multi-dimensional data where data have more attributes that spread across wide geographical area and spatial database store objects that have spatial characteristics that describes them.

2.2 R-Tree

R-Tree is a tree data structure used for spatial access methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons. R-Tree is a spatial indexing technique such that given a query rectangle, we can quickly locate the spatial object results. For location-based search, it is very common to search for objects based on their spatial location. A query can be represented as another rectangle. The query is about locating the spatial objects whose Minimum Bounding Rectangle (MBR) overlaps with the query rectangle as shown in the figure 5 below:

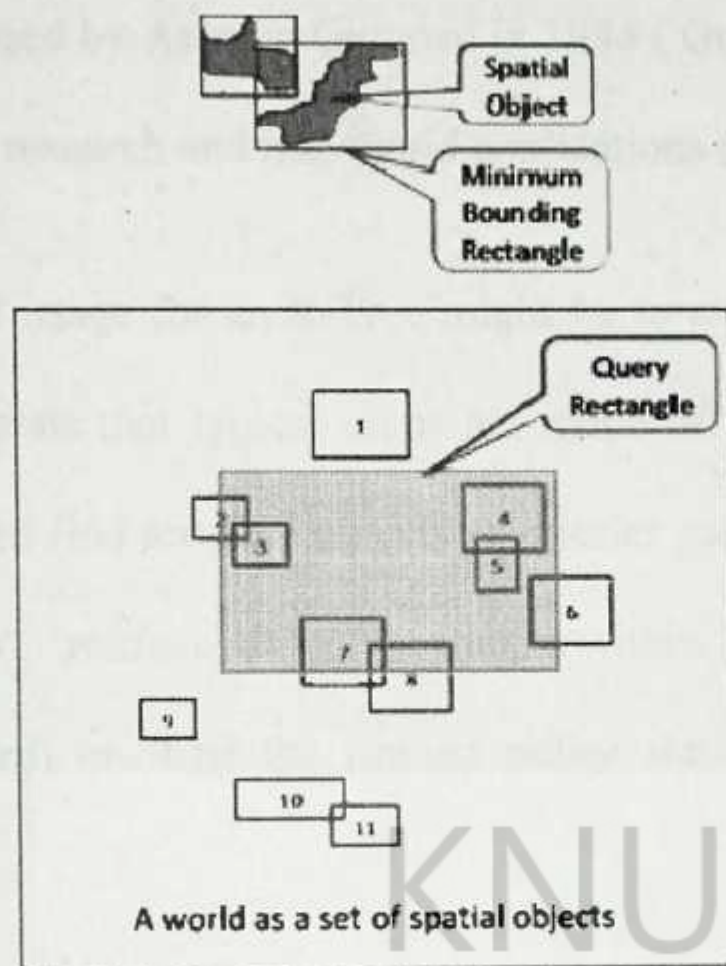


Figure 6: Source: <http://horicky.blogspot.com/2010/02/spatial-index-rtree.html#links>

The concept is similar to B-Tree, because spatial objects that are close by are grouped and each of them form a tree whose intermediate nodes contains "close-by" objects. Since the MBR of the parent node contains all MBR of its children, the Objects are close by if their parent's MBR is minimized.

It's a balanced search tree like B-Tree and so all leaf nodes are at the same height, organizes the data in pages, and is designed for storage on disk as used in databases. Each page can contain a maximum number of entries, often denoted as M . Researches show that it also guarantees a minimum fill (except for the root node), however best performance has been experienced with a minimum fill of 30% – 40% of the maximum number of entries, B-Trees guarantee 50% page fill, and B*-Trees even 66%). The reason for this is the more complex balancing required for spatial data as opposed to linear data stored in B-trees.

The R-Tree was proposed by Antoine Guttman in 1984 (Gutt 84, op. cit. pp. 47) and has found significant use in both research and real-world applications (Y. Manolopoulos et.al., 2009).

A common real-world usage for an R-Tree might be to store spatial objects such as restaurant locations or the polygons that typical maps are made of: streets, buildings, outlines of lakes, coastlines, etc. and then find answers quickly to queries such as "Find all museums within 2 km of my current location", "retrieve all road segments within 2 km of my location" (to display them in a navigation system) or "find the nearest police station" (although not taking roads into account).

The key idea of R-Tree data structure is to group nearby objects and represent them with their minimum bounding rectangle (MBR) in the next higher level of the tree. Since all objects lie within this bounding rectangle, a query that does not intersect the bounding rectangle can also not intersect any of the contained objects. At the leaf level, each rectangle describes a single object; at higher levels the aggregation of an increasing number of objects. This can also be seen as an increasingly coarse approximation of the data set.

Each node of an R-Tree has a variable number of entries up to some pre-defined maximum. Each entry within a non-leaf node stores two pieces of data: a way of identifying a child node and the bounding box of all entries within this child node. So, according to Antoine Guttman an R-Tree satisfies the following properties:

- Every leaf node contains between m and M index records unless it is the root. Thus, the root can have less entries than m .

- For each index record in a leaf node, I is the smallest rectangle that spatially contains the n -dimensional data object represented by the indicated tuple
- Every non-leaf node has between m and M children unless it is the root
- For each entry in a non-leaf node, i is the smallest rectangle that spatially contains the rectangles in the child node
- The root node has at least two children unless it is a leaf
- All leaves appear on the same level. That means the tree is balanced

2.2.1. Operation on R-Trees

As with most trees, **the searching** algorithms are rather simple. The key idea is to use the bounding boxes to decide whether or not to search inside a subtree. In this way, most of the nodes in the tree are never read during a search. Searching start from the root, (Scott T. et al., 1996) and each children's MRB is examine to see if it overlaps with the query MBR. The whole subtree is skipped if there is no overlapping, otherwise, an alternative search is adopted by drilling into each child.

Notice that unlike other tree algorithm where only traverse down one path. The search needs to traverse down multiple path if the overlaps happen. Therefore, one need to structure the tree to minimize the overlapping as high in the tree as possible. This means that the sum of MBR areas along each path must be minimized (from the root to the leaf) as much as possible.

The insertion in R-Trees are the same with slight difference in B-Trees. To insert a new spatial object, starting from the root node, pick the children node whose MBR will be extended least if the new spatial object is added, walk down this path until reaching the leaf node.

If the leaf node has space left, insert the object to the leaf node and update the MBR of the leaf node as well as all its parents (Tan, T. C. et al., 1997). Otherwise, split the leaf node into two (by creating a new leaf node and copy some of the content of the original leaf node to this new one). And then add the newly created leaf node to the parent of the original leaf node. If the parent has no space left, the parent will be split as well.

If the split goes all the way to the root, the original root will then be split and a new root is created.

Deleting a spatial node will first search for the containing leaf node. Remove the spatial node from the leaf node's content and update its MBR and its parent's MBR all the way to the root. If the leaf node now has less than m nodes, then we need to condense the node by marking the leaf node to be deleted. And then we remove the leaf node from its parent's content as well as updating them. If the parent is now less than m nodes, we mark the parent to be deleted also and remove the parent from the parent's parent. At this point, the entire node that is marked delete is removed from the R-Tree.

Notice that the content with these delete nodes is not all garbage, since they still have some children that are valid nodes (but were removed from the tree). Now, you need to reinsert all these valid nodes back in the tree.

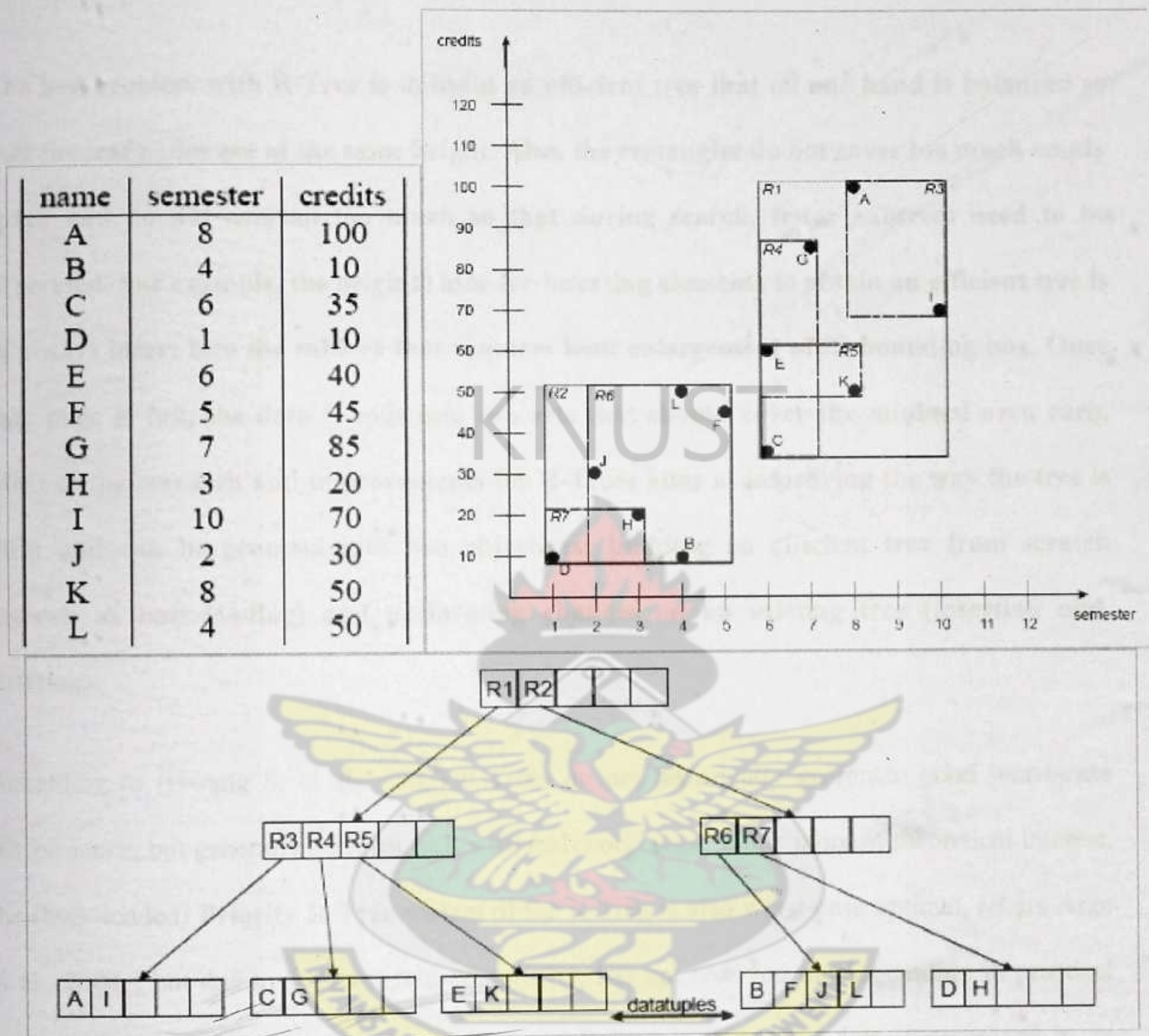
Finally, you check if the root node contains only one child, you throw away the original root and use its own child to become the new root.

Note: The only difference between R-Tree and B-Tree with respect to the algorithm used is that The insertion and deletion algorithms use the bounding boxes from the nodes to ensure that "nearby" elements are placed in the same leaf node (in particular, a new element will go into the leaf node that requires the least enlargement in its bounding box). Each entry within a leaf node stores two pieces of information; a way of identifying the actual data element (which, alternatively, may be placed directly in the node), and the bounding box of the data element.

Similarly, the searching algorithms (e.g., intersection, containment, nearest) use the bounding boxes to decide whether or not to search inside a child node. In this way, most of the nodes in the tree are never "touched" during a search. Like B-trees, this makes R-Trees suitable for large data sets and databases, where nodes can be paged to memory when needed, and the whole tree cannot be kept in main memory.

Update happens when an existing spatial node changes its dimension. One way is to just change the spatial node's MBR but not change the R-Tree. A better way (but more expensive) is to delete the node, modify its MBR and then insert it back to the R-Tree.

A typical example of R-Tree data structure is as shown in the **Fig 7** below:



Source: <http://lacot.org/public/enst/bda/img/schema1.gif>

R1 and R2 are the Root Nodes, R3, R4, R5 and R6 , R7 are also the Leaf nodes.

R1 covers leaf nodes R3, R4, R5 & R2 covers leaf nodes R6, R7. These comprise them with the Minimum Bounding Rectangle

R1 covers child nodes "A and I" and comprise them with the Minimum Bounding Rectangle

R4 covers child nodes “C and G” and comprise them with the Minimum Bounding Rectangle in that order. The records to be located are A, B, to L.

The key problem with R-Tree is to build an efficient tree that on one hand is balanced so that the leaf nodes are at the same height. Also, the rectangles do not cover too much empty space and do not overlap too much so that during search, fewer subtrees need to be processed. For example, the original idea for inserting elements to obtain an efficient tree is to always insert into the subtree that requires least enlargement of its bounding box. Once that page is full, the data is split into two sets that should cover the minimal area each. Most of the research and improvements for R-Trees aims at improving the way the tree is built and can be grouped into two objectives: building an efficient tree from scratch (known as bulk-loading) and performing changes on an existing tree (insertion and deletion).

According to (Hwang S. et al., 2003) R-Trees do not historically guarantee good worst-case performance, but generally perform well with real-world data. While more of theoretical interest, the (bulk-loaded) **Priority R-Tree** variant of the R-Tree is also worst-case optimal, (Lars Arge et al., 2004) but due to the increased complexity, has not received much attention in practical applications so far. However, ~~one~~ advantage of R-Tree is that when data is organized in an R-Tree, the **k nearest neighbours** (for any local space) of all points can efficiently be computed using a spatial join (Brinkhoff et al., 1993) This is beneficial for many algorithms based on the k nearest neighbours.

2.2.2. Visualizing R-Tree in 3-Dimension (3-D)

A three-dimensional (3-D) spatial index is required for real time applications of integrated organization and management in virtual geographic environments. Being one of the most promising methods, the R-Tree spatial index which adopted 2-D has paid increasing attention in 3-D, geospatial database management. Since the existing R-Tree methods are usually limited by their weakness of low efficiency, due to the critical overlap of sibling nodes and the uneven size of nodes. The 3-D R-Tree, proposed in (Y. Theodoridis et al., 1991) considers time as an extra dimension and represents 2-D rectangles with time intervals as three-dimensional boxes. This tree can be the original R-Tree or any of its (ephemeral) variants. The 3-D R-Tree was implemented and evaluated analytically and experimentally in (Y. Theodoridis et al., 1998), and it was compared with the alternative solution of maintaining a spatial index (e.g. a 2-D R-Tree) and a temporal index (e.g., a 1-D R-Tree or a segment tree). Synthetic (uniform-like) datasets were used, and the retrieval costs for pure temporal (during, before), pure spatial (overlap, above), and spatiotemporal operators (the four combinations) were measured, according to Y. Theodoridis . The results suggest that the unified scheme of a single 3-D R-Tree is obviously superior when spatiotemporal queries are posed, whereas for mixed workloads, the decision depends on the selectivity of the operators.

With the increase variations in R-Tree, a new spatial cluster grouping algorithm and R-Tree insertion algorithm is then proposed. Experimental analysis on comparative performance of spatial indexing shows that by the new method the overlap of R-Tree sibling nodes is minimized drastically and a balance in the volumes of the nodes is maintained, such as Virtual geographic environments; 3-D spatial index; R-Tree and Spatial cluster grouping.

2.2.3 Variations in R-Trees

A survey by Gaede and Guenther in (V. Gaede et al., 1998) annotates a vast list of citations related to multi-dimensional access methods and, in particular, refers to R-Trees to a significant extent. Just as in B-Tree various variations had brought an improvement in the original R-Tree proposed by Antoine Guttman. In this review, I am further focusing on the family of R-Trees by enlightening the similarities and differences, advantages and disadvantages of the variations in a more exhaustive manner. As the number of variants that have appeared in the literature is large, I group them according to the special characteristics of the assumed environment or application.

2.2.3.1 The R+-Tree

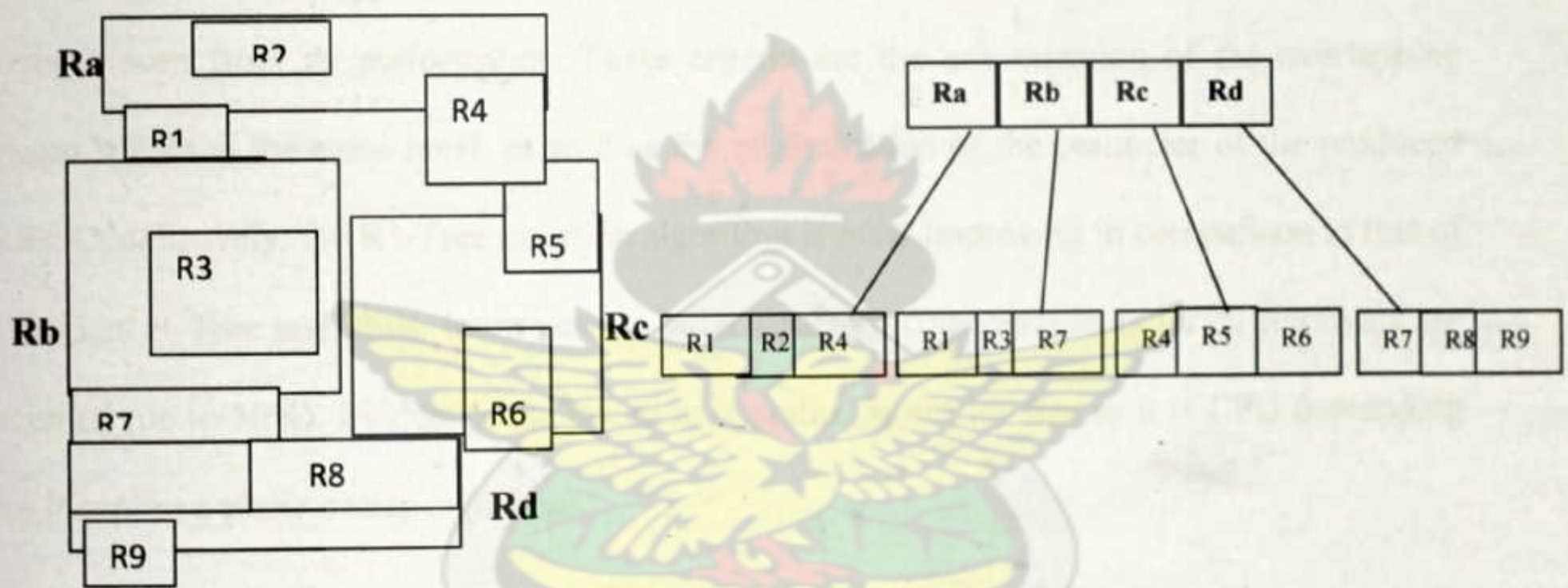
The original R-Tree has two important disadvantages that is:

- The execution of a point location query in an R-Tree may lead to the investigation of several paths from the root to the leaf level. This characteristic may lead to performance deterioration, specifically when the overlap of the MBRs is significant.
- A few large rectangles may increase the degree of overlap significantly, leading to performance degradation during range query execution, due to empty space.

R+-Trees were proposed as a structure that avoids visiting multiple paths during point location queries, and thus the retrieval performance could be improved (M.Stonebraker et al.,1986). Moreover, MBR overlapping of internal nodes is avoided. This is achieved by using the clipping technique. In simple words, R+-Trees do not allow overlapping of MBRs at the same tree level. In turn, to achieve this, inserted objects have to be divided in two or more MBRs, which means that a specific object's entries may be duplicated and redundantly stored in several nodes

as shown below. Therefore, this redundancy works in the opposite direction of decreasing the retrieval performance in case of window queries. At the same time, another side effect of clipping is that during insertions, an MBR augmentation may lead to a series of update operations in a chain-reaction type. Also, under certain circumstances, the structure may lead to a deadlock, as, for example, when a split has to take place at a node with $M+1$ rectangles, where every rectangle encloses a smaller one.

Fig 8. An R+-Tree example



2.2.3.2 The R*-Tree

The R*-Tree. Although proposed in (N. Beckmann et al., 1990), are still very well received and widely accepted in the literature as a prevailing performance-wise structure that is often used as a basis for performance comparisons. The R*-Tree does not obey the limitation for the number of pairs per node and follows a sophisticated node split technique. More specifically, the technique of 'forced reinsertion' is applied, according to which, when a node overflows, p entries are extracted and reinserted in the tree (p being a parameter, with 30% a suggested optimal value). Other novel features of R*-Tree is that it takes into account additional criteria except the minimization of the sum of the areas of the produced MBRs. The benefit from involving these criteria is seen from its performance. These criteria are the minimization of the overlapping between MBRs at the same level, as well as the minimization of the perimeter of the produced MBRs. Conclusively, the R*-Tree insertion algorithm is quite improving in comparison to that of the original R-Tree and, thus, improves the latter structure considerable as far as retrievals are concerned (up to 50%). Evidently, the insertion operation is not for free as it is CPU demanding since it applies a plane-sweep algorithm.

2.2.3.3 The Hilbert R-Tree

The Hilbert R-Tree is a hybrid structure based on R-Trees and B+-Trees (I. Kamel et al., 1994). Actually, it is a B+-Tree with geometrical objects being characterized by the Hilbert value of their centroid. Thus, leaves and internal nodes are augmented by the largest Hilbert value of their contained objects or their descendants, respectively. For an insertion of a new object, at each level the Hilbert values of the alternative nodes are checked and the smallest one that is larger than the Hilbert value of the object under insertion is followed. In addition, another heuristic

used in case of overflow by Hilbert R-Trees is the redistribution of objects in sibling nodes. In other words, in such a case up to s siblings are checked in order to find available space and absorb the new object. A split takes place only if all s siblings are full and, thus, $s+1$ nodes are produced. This heuristic is similar to that applied in B*-Trees, where redistribution and 2-to-3 splits are performed during node overflows (Donald Knuth 1967). According to the authors' experimentation, Hilbert R-Trees were proven to be the best dynamic version of R-Trees. However, this variant is vulnerable performance-wise to large objects. Moreover, by increasing the space dimensionality, proximity is not preserved adequately by the Hilbert curve, leading to increased overlap of MBRs in internal tree nodes.

2.2.3.4 Linear Node Splitting

Linear Node Splitting in (Ang and Tan, op. cit. 1997) have proposed a linear algorithm to distribute the objects of an overflowing node in two sets. The primary criterion of this algorithm was to distribute the objects between the two nodes as evenly as possible, whereas the second criterion was the minimization of the overlapping between them. Finally, the third criterion was the minimization of the total coverage. Experiments using this algorithm have shown that it results in R-trees with better characteristics and better performance for window queries in comparison with the quadratic algorithm of the original R-tree.

2.2.3.5 Optimal Node Splitting

In an attempt to solve overflow of node, three node splitting algorithms were proposed by Guttman to handle a node overflow. The three algorithms have linear, quadratic, and exponential complexity, respectively. Among them, the exponential algorithm achieves the optimal

bi-partitioning of the rectangles, at the expense of increased splitting cost. On the other hand, the linear algorithm is more time efficient but fails to determine an optimal rectangle bipartition. Therefore, the best compromise between efficiency and bipartition optimality is the quadratic algorithm.

Y. Garcia et al., (1998) elaborated the optimal exponential algorithm of Guttman and reached a new optimal polynomial algorithm $O(nd)$ where d is the space dimensionality and $n = M + 1$ is the number of entries of the node that overflows. For n rectangles the number of possible bipartitions is exponential in n . Each bipartition is characterized by a pair of MBRs, one for each set of rectangles in each partition. The key issue, however, is that a large number of candidate bipartitions share the same pair of MBRs. This happens when we exchange rectangles that do not participate in the formulation of the MBRs. The authors show that if the cost function used depends only on the characteristics of the MBRs, then the number of different MBR pairs is polynomial. Therefore, the number of different bipartitions that must be evaluated to minimize the cost function can be determined in polynomial time.

The proposed optimal node splitting algorithm investigates each of the $O(n^2)$ pairs of MBRs and selects the one that minimizes the cost function. Then each one of the rectangles is assigned to the MBR that it is enclosed by. Rectangles that lie at the intersection of the two MBRs are assigned to one of them according to a selected criterion. In the same paper, the authors give another insertion heuristic, which is called *sibling-shift*. In particular, the objects of an overflowing node are optimally separated in two sets. Then one set is stored in the specific node,

whereas the other set is inserted in a sibling node that will depict the minimum increase of an objective function (e.g., expected number of disk accesses). If the latter node can accommodate the specific set, then the algorithm terminates. Otherwise, in a recursive manner the latter node is split.

Finally, the process terminates when either a sibling absorbs the insertion or this is not possible, in which case a new node is created to store the pending set. The authors reported that the combined use of the optimal partitioning algorithm and the sibling-shift policy improved the index quality (i.e., node utilization) and the retrieval performance in comparison to the Hilbert R-trees, at the cost of increased insertion time.

2.2.3.6 Branch Grafting

More recently, in (T. Schrek et al., 2000) an insertion heuristic was proposed to improve the shape of the R-tree so that the tree achieves a more elegant shape, with a smaller number of nodes and better storage utilization. In particular, this technique considers how to redistribute data among neighboring nodes, so as to reduce the total number of created nodes. The approach of **branch grafting** is motivated by the following observation. If, in the case of node overflow, you examined all other nodes to see if there is another node (at the same level) able to accommodate one of the overflowed node's rectangles, the split could be prevented. Evidently, in this case, a split is performed only when all nodes are completely full. Since the aforementioned procedure is clearly prohibitive as it would dramatically increase the insertion cost, the **branch grafting** method focuses only on the neighboring nodes to redistribute an entry from the overflowed node. Actually, the term *grafting* refers to the operation of moving a leaf or internal node (along with the corresponding subtree) from one part of the tree to another.

The objectives of branch grafting are to achieve better-shaped R-trees and to reduce the total number of nodes. Both these factors can improve performance during query processing.

To illustrate these issues, the following example is given in (T. Schrek et al., *ibid*). Assume that you are inserting eight rectangles (with the order given by their numbering), which are depicted in figure, below. Let the maximum (minimum) number of entries within a node be equal to 4. Therefore, the required result is shown in Figure 9 (a), because they clearly form two separate groups. However, by using the R-tree insertion algorithm, which invokes splitting after each overflow, the result in Figure 9. (b) would be produced. Using the branch and grafting method, the split after the insertion of rectangle *H* can be avoided.

Figure 9. (c) illustrates the resulted R-tree after the insertion of the first seven rectangles (i.e., *A* to *G*). When rectangle *H* has to be inserted, the branch grafting method finds out that rectangle 3 is covered by node *R1*, which has room for one extra rectangle. Therefore, rectangle *C* is moved from node *R2* to node *R1*, and rectangle *H* can be inserted in *R2* without causing an overflow. The resulted R-tree is depicted in Figure 9. (d).

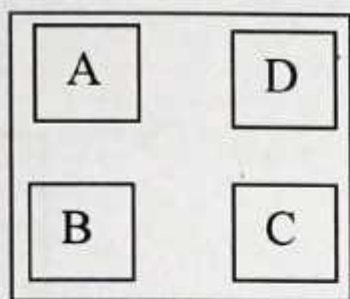


Fig. 9. (a)

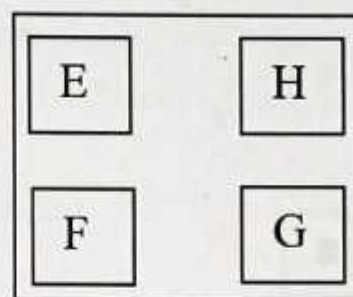


Fig. 9. (b)

Fig. 9. Branch grafting technique:

(a) Optimal result after inserting 8 rectangles;

(b) actual result produced after two R-tree splits.

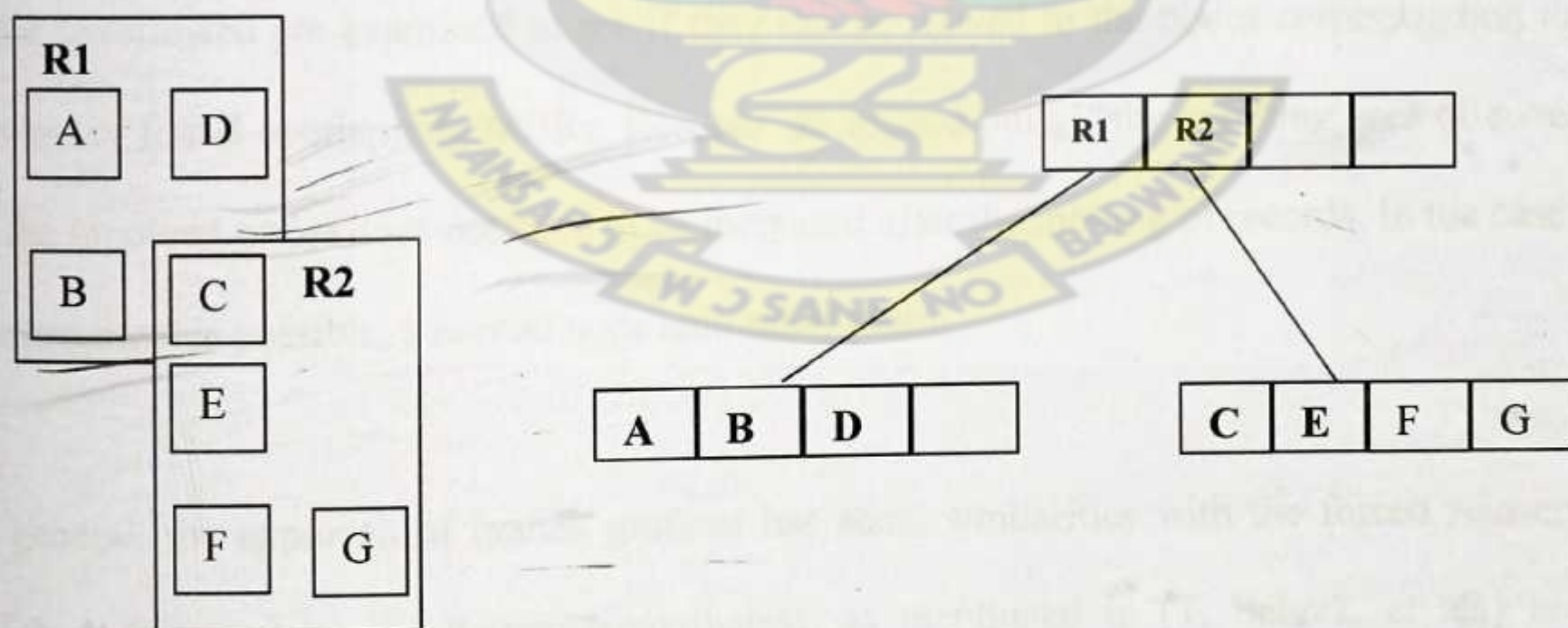


Fig. 9. (c)

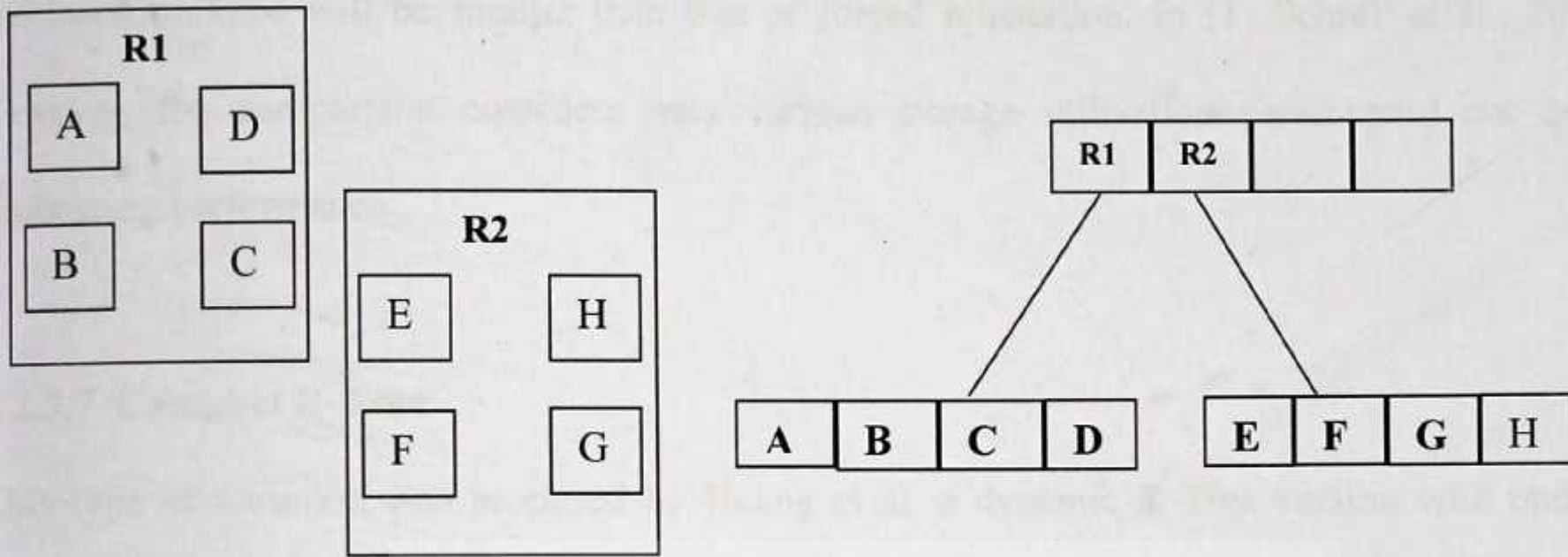


Fig. 9. (d)

Fig. 9. (c) Resulted R-tree after the first 7 insertions (rectangles *A* to *G*);

Fig. 9. (d) result of branch grafting after inserting the rectangle *H*.

In summary, in case of node overflow, the branch grafting algorithm first examines the parent node, to find the MBRs that overlap the MBR of the overflowed node. Next, individual records in the overflowed are examined to see if they can be moved to the nodes corresponding to the previously found overlapping MBRs. Records are moved only if the resulting area of coverage for the involved nodes does not have to be increased after the moving of records. In the case that no movement is possible, a normal node split takes place.

In general, the approach of branch grafting has some similarities with the forced reinsertion, which is followed by the R-tree. Nevertheless, as mentioned in (T. Schrek, et al.,) branch grafting is not expected to outperform forced reinsertion during query performance. However,

one may expect that, because branch grafting tries to locally handle the overflow, the overhead to the insertion time will be smaller than that of forced reinsertion. In (T. Schrek et al., 2000), however, the comparison considers only various storage utilization parameters, not query processing performance.

2.2.3.7 Compact R-Tree

This type of variation was proposed by Huang et al, a dynamic R-Tree version with optimal space overhead (P. W. Huang et al., 2001). The motivation behind the proposed approach is that R-Trees, R+-Trees, and R*-Trees suffer from the storage utilization problem, which is around 70% in the average case. Therefore, the authors improve the insertion mechanism of R-trees to a more compact R-tree structure, with no penalty on performance during queries. The heuristics applied are simple, meaning that no complex operations are required to significantly improve storage utilization. Among the $M+1$ entries of an overflowing node during insertions, a set of M entries is selected to remain in this node, under the constraint that the resulting MBR is the minimum possible. Then the remaining entry is inserted to a sibling that:

- has available space, and
- whose MBR is enlarged as little as possible.

Thus, a split takes place only if there is no available space in any of the sibling nodes.

Performance evaluation results reported in (P. W. Huang et al., 2001) have shown that the storage utilization of the new heuristic is between 97% and 99%, which is a great improvement.

A direct impact of the storage utilization improvement is the fact that fewer tree nodes are required to index a given dataset. Moreover, less time is required to build the tree by individual

insertions, because of the reduced number of split operations required. Finally, caching is improved because the buffer associated with the tree requires less space to accommodate tree nodes. It has been observed that the query performance of window queries is similar to that of Guttman's R-tree.

2.2.3.8. Priority R-Tree

The Priority R-tree (PR-R-tree for short) has been proposed in (Lars Arge et al., op. cit) and is a provably asymptotically optimal variation of the R-tree. The term *priority* in the name of PR-tree stems from the fact that its bulk-loading algorithm utilizes the "priority rectangles".

A PR-tree is a height-balanced tree, i.e., all its leaves are at the same level and in each node c entries are stored. According to Lars Arge et al., pseudo-PR-tree TS on S are defined recursively: if S contains fewer than c rectangles (c is the maximum number of rectangles that can fit in a disk page), then TS consists of a single leaf. Otherwise, TS consists of a node v . To derive a PR-tree from a pseudo-PR-tree, the PR-tree has to be built into stages, in a bottom-up fashion. First, the leaves V_0 are created and the construction proceeds to the root node. At stage i , first the pseudo-PR-tree TS_i is constructed from the rectangles S_i of this level. The nodes of level i in the PR-tree consist of all the leaves of TS_i , i.e., the internal nodes are discarded. The bulk-loading of a PR-tree is slower than the bulk-loading of a packed 4D Hilbert tree. However, regarding query performance, for nicely distributed real data, PR-trees perform similar to existing R-tree variants. In contrast, with extreme data (very skewed data, which contain rectangles with high differences in aspect ratios), PR-trees outperform all other variants, due to their guaranteed worst-case

performance, because any other R-tree variant may require (in the worst case) the retrieval of leaves, even for queries that are not satisfied by any rectangle.

2.2.3.9 Deviating Variations

Apart from the aforementioned R-Tree variations, a number of interesting extensions and adaptations have been proposed that in some sense deviate drastically from the original idea of R-trees. Among other efforts, include the following research works.

The **Sphere trees** by Oosterom use minimum bounding spheres instead of MBRs (P. Oosterom et al., 1990), whereas the **Cell trees** by Guenther use minimum bounding polygons designed to accommodate arbitrary

shape objects (O.Guenther 1989). The **Cell tree** is a clipping-based structure and, thus, a variant of **Cell trees** has been proposed to overcome the disadvantages of clipping. The latter variant uses 'oversize shelves', i.e., special nodes attached to internal ones, which contain objects that normally should cause considerable splits (O.Guenther et al 1991).

Similarly to Cell trees, Jagadish and Schiwietz proposed independently the structure of **Polyhedral trees or P-trees**, which use minimum bounding polygons instead of MBRs (H. V Jagadish 1990).

The **X-tree** by Berchtold et al. uses the notion of 'supernodes' (i.e., nodes of greater size) to handle overflows and avoid splits (Berchtold et al. op. cit. pp.28-39).

The **S-tree** by (Aggrawal et al., 1997). relaxes the rule that the R-tree is a balanced structure and may have leaves at different tree levels. However, S-trees are static structures in the sense that they demand the data to be known in advance.

Another recent effort by (C. H Ang et al., 2000), is the **Bitmap R-tree** where each node contains bitmap descriptions of the internal and external object regions except the MBRs of the objects. Thus, the extra space demand is paid off by savings in retrieval performance due to better tree pruning. The same trade-off holds for the **RS-tree**, which is proposed by (J. D. Park et al., 2001) and connects an R*-tree with a signature tree with an one-to-one node correspondence.

P. K. Agarwal et al., (2001) proposed the **Box-tree**, that is, a bounding volume hierarchy that uses axis-aligned boxes as bounding volumes. They provide worst-case lower bounds on query complexity, showing that box-trees are close to optimal, and they present algorithms to convert box-trees to R-trees, resulting in R-trees with (almost) optimal query complexity.

Y. J. Lee et al., (2001) developed the **DR-tree**, which is a main memory structure for multi-dimensional objects. They couple the R*-tree with this structure to improve the spatial query performance.

Finally, Bozanis et al. have partitioned the R-tree in a number of smaller R-trees (P. Bozanis et al., 1999), along the lines of the binomial queues that are an efficient variation of heaps.

2.3 Static Versions of R-Trees

There are common applications that use static data. For instance, insertions and deletions in census, cartographic and environmental databases are rare or even they are not performed at all. For such applications, special attention should be paid in order to construct an optimal structure with regards to some tree characteristics, such as storage overhead minimization, storage utilization maximization, minimization of overlap or cover between tree nodes, or combinations of the above. Therefore, it is anticipated that query processing performance will be improved. These methods are well known in the literature as 'packing' or 'bulk loading'. Thus, such methods that require the data to be known in advance are also reviewed.

2.3.1 The Packed R-Trees

The first packing algorithm was proposed by (N. Roussopoulos et al., 1985), soon after the proposal of the original R-tree. This first effort basically suggests ordering the objects according to some spatial criterion (e.g., according to ascending x-coordinate) and then grouping them in leaf pages. No experimental work is presented to compare the performance of this method to that of the original R-tree. However, based on this simple inspiration a number of other efforts have been proposed in the literature.

2.3.2 The Hilbert Packed R-Trees

I. Kamel and Faloutsos (1993) proposed an elaborated method to construct a static R-tree with 100% storage utilization based on sorting the objects according to the Hilbert value of their centroids and then build the tree in a bottom-up manner. Experiments showed that the latter method achieves significantly better performance than the original R-tree with quadratic split,

the R*-tree and the Packed R-tree by (N. Roussopoulos et al., 1985) in point and window queries. Moreover, Kamel and Faloutsos proposed a formula to estimate the average number of node access, which is independent of the details of the R-tree maintenance algorithms and can be applied to any R-tree variant.

2.3.3 Small-Tree-Large-Tree

The previous packing algorithms build an R-tree access method from a set of spatial objects. The small-tree-large-tree method (STLT) (L. Chen et al., 1998) performs efficient bulk insertions into an existing R-tree structure. And they proposed this: Let R be a set of spatial objects indexed by an already existing R-tree and N a set of new objects that must be inserted. Instead of inserting the objects in the R-tree one-by-one, the STLT method constructs a small R-tree for N and then inserts the small R-tree into the large R-tree. Obviously, the efficiency of the resulting index depends on the data distribution of the small R-tree. If the objects in N cover a large part of the data space, then using the STLT approach will result in increasing overlap in the resulting index. Therefore, the method is best suited for skewed data distributions. STLT is extended, where the Generalized R-tree Bulk-Insertion Strategy (GBI) is proposed. GBI inserts new incoming data sets into active R-trees as follows: it first partitions the data sets into a set of clusters and outliers, then it constructs a small R-tree for each cluster, finding suitable places in the original R-tree to insert the newly created R-trees, and finally it bulk-inserts the new R-trees and the outliers in the original R-tree.

2.3.4 Buffer R-Trees

According to (L. Arge et al., 2002), the Buffer R-tree (BR) for performing bulk update and queries. BR is based on the buffer tree lazy buffering technique and exploits the available main memory. Analytical results in L. Arge et al show the efficiency of BR, whereas experimental results illustrates its superiority over the other methods. BR requires smaller execution times to perform bulk updates and produces a better quality index in terms of query performance. Moreover, BR (differently from other methods) allows for simultaneous batch updates and queries.

2.3.5 Trajectory Bundle-Tree (TB-Tree)

The TB-tree (D. Pfoser et al., 2000) relaxes a fundamental R-tree property, i.e., keeping neighboring entries together in a node, and strictly preserves trajectories (paths) such that a leaf node only contains segments belonging to the same trajectory. This is achieved by giving up on space discrimination. The TB-tree indexes past locations of objects and supports continuous changes.

2.4 Summary and Conclusion

Evidently, the original R-tree, proposed by Guttman, has influenced all the forthcoming variations of static and dynamic R-tree structures. The R*-tree followed an engineering approach and evaluated several factors that affect the performance of the R-tree. For this reason, it is considered the most robust variant and has found numerous applications, in both research and commercial systems. However, the empirical study has shown that the Hilbert R-tree can perform better than the other variants in some cases. It is worth mentioning that the PR-tree,

although a variant that deviates from other existing ones, is the first approach that offers guaranteed worst-case performance and overcomes the degenerated cases when almost the entire tree has to be traversed. Therefore, despite its more complex building algorithm, it has to be considered the best variant reported so far.

KNUST



CHAPTER THREE

METHODOLOGY

3.0 Introduction

In recent years, there has been an upsurge of interest in spatial databases. A major issue is how to efficiently manipulate massive amounts of spatial data stored on disk in multidimensional spatial indexes (data structures). Construction of spatial indexes has been studied intensively in the database community. The continuous arrival of massive amounts of new data makes it important to efficiently update existing indexes. The R-tree, one of the most popular access methods for rectangles, is based on the heuristic optimization of the area of the enclosing rectangle in each inner node. A lot of variations has emerged since its inception, all in an effort to improve the original R-tree proposed by Antoine Gutman. In this research, a new R-tree variant “an optimized R-tree” is designed which is expected to improve upon the original R-tree. This section covers the study area, design of study, design of algorithms and constraints/problems.

3.1 Study Area

There has been major concern of interest in spatial databases in the commercial and research database communities. Spatial databases are systems designed to store, manage, and manipulate spatial data like points, polyclinic, polygons, and surfaces. Geographic information systems (GIS) are a popular incarnation. Spatial database applications often involve massive data sets, thus the need for efficient handling of massive spatial data sets has become a major issue, and a large number of disk based multidimensional index structures (data structures) have been proposed in the database literature. Typically, multidimensional index structures support

insertions, deletions, and updates, as well as a number of proximity queries like window or nearest-neighbor queries. Recent research in the database community has focused on supporting bulk operations, in which a large number of operations are performed on the index at the same time. The increased interest in bulk operations is a result of the ever-increasing size of the manipulated spatial data sets and the fact that performing a large number of single operations one at a time is simply too inefficient to be of practical use. The scope is limited to memory utilization and the use of directory pages in memory.

3.2 Design of Study

In this design study, I consider all approaches of optimizing the retrieval performance that have to be applied during the insertion of a new data rectangle. The two basic design for this approach are:

- the Leaf Node and
- the Non Leaf Node

A leaf node contains entries of the form (Dataobject, Rectangle) where Dataobject refers to a record in the database, describing a spatial object and Rectangle is the enclosing rectangle of that spatial object.

A non-leaf node contains entries of the form (cp, Rectangle) where cp is the address of a child node in the R-tree and Rectangle is the minimum bounding rectangle of all rectangles which are entries in that child node.

As I said in the previous chapters, the main problem with R-tree are:

- For an arbitrary set of rectangles, dynamically build up bounding boxes from subsets of between m and M rectangles, in a way that, arbitrary retrieval operations with query rectangles of arbitrary size are supported efficiently. The known parameters of good retrieval performance affect each other in a very complex way, such that it is impossible to optimize one of them without influencing other parameters which may cause a deterioration of the overall performance.
- Moreover, since the data rectangles may have very different size and shape and the minimum bonding boxes grow and shrink dynamically, the success of methods which will optimize one parameter is very uncertain.

In this approach, some of the parameters which are essential for the retrieval performance are considered. Furthermore, interdependences between different parameters are analyzed. The design seek to consider the following parameters:

1. The area covered by a minimum bounding rectangle should be minimized, i.e. the area covered by the bounding rectangle but not covered by the enclosed rectangles, the dead space, should be minimized. This will improve performance since decisions which paths have to be traversed, can be taken on higher levels.
2. The overlap between minimum bounding rectangle should be minimized. That's also decreases the number of paths to be traversed.

The method will consider the principle of R-tree under Antoine Guttman with respect to inserting new data rectangle and its effects as compared with the design in detail an improved R-tree (an Optimized R-tree).

3.3 The Principles of R-Tree

The R-tree is a dynamic structure, thus all approaches of optimizing the retrieval performance have to be applied during the insertion of a new data rectangle. The insertion algorithm calls two more algorithms in which the crucial decisions for good retrieval performance are made.

The first is the algorithm ChooseSubtree Beginning in the root, descending to a leaf, it finds on every level the most suitable subtree to accommodate the new entry.

The second is the algorithm Split and it's always called, If ChooseSubtree ends on a node filled with the maximum number of entries M Split should distribute $M+1$ rectangles into two nodes in the most appropriate manner.

In the following, the ChooseSubtree- and Split-algorithms, suggested in available R-tree variants are analyzed and discussed. I will first consider the original R-tree as proposed by Guttman in [Gut 84].

Algorithm ChooseSubtree (Select a leaf node in which to place a new index entry E)

CS 1 Set N to be the root

CS 2 If N is a leaf

Return N

Else

Choose the entry in N whose rectangle needs least area enlargement to include the new data. Resolve ties by choosing the entry with the rectangle of smallest area
end

CS 3 *Set N to be the childnode pointed to by the childpointer of the chosen entry and repeat from CS2*

Obviously, the method of optimization is to minimize the area covered by a directory rectangle, this may also reduce the overlap and the CPU cost will be relatively low.

Guttman discusses split-algorithms with exponential, quadratic and linear cost with respect to the number of entries of a node. All of them are designed to minimize the area, covered by the two rectangles resulting from the split. The exponential split finds the area with the global minimum, but the cpu cost is too high. The others try to find approximations. In his experiments, Guttman obtains nearly the same retrieval performance but I will only discuss the quadratic algorithms in details.

Algorithm QuadraticSplit

[Divide a set of $M+1$ entries into two groups]

QS 1 *Invoke PickSeeds to choose two entries to be the first entries of the groups*

QS 2 Repeat

DistributeEntry

until

all entries are distributed or one of the two groups has $M-m+1$ entries

QS 3 If entries remain, assign them to the other group such that it has the minimum number m

Algorithms PickSeeds

PS 1 For each pair of entries $E1$ and $E2$, compose a rectangle R including $E1$ rectangle and $E2$ rectangle

Calculate $d = \text{area}(R) - \text{area}(E1 \text{ rectangle}) - \text{area}(E2 \text{ rectangle})$

PS 2 Choose the pair with the largest d

Algorithms DistributeEntry

DE 1 Invoke *PickNext* to choose the next entry to be assigned.

DE 2 Add it to the group whose covering rectangle will have to be enlarged least to accommodate it. Resolve ties by adding the entry to the group with the smallest area, then to the one with the fewer entries, then to either.

Algorithms PickNext

PN 1 For each entry E not yet in a group, calculate d_1 = the area increase required in the covering rectangle of Group 1 to include E Rectangle

Calculate d_2 analogously for Group 2

PN 2 Choose the entry with the maximum difference

between d_1 and d_2

The algorithm PickSeeds finds the two rectangles which would waste the largest area put in one group. In this sense the two rectangles are the most distant ones. It is important to mention that the seeds will tend to be small too, if the rectangles to be distributed are of very different size and or the overlap between them is high.

The algorithm DistributeEntry assigns the remaining entries by the criterion of minimum area. PickNext chooses the entry with the best area and the value in every situation.

If this algorithm starts with small seeds, problems may occur if in $d-1$ (the area required in the covering rectangle) of the d axes a far away rectangle has nearly the same coordinates as one of the seeds, it will be distributed first. Indeed, the area and the area enlargement of the created needle-like bounding rectangle will be very small, but the distance is very large. This may initiate a very bad split. Moreover, the algorithm tends to prefer the bounding rectangle, created from the first assignment of a rectangle to one seed. Since it was enlarged, it will be larger than others. Thus it needs less area enlargement to include the next entry, it will be enlarged again, and so on.

Another problem is that, if one group has reached the maximum number of entries $M-m+1$, all remaining entries are assigned to the other group without considering geometric properties. Figure 10 (as shown below) gives an example showing all these problems. The result is either a split with much overlap (fig 10b) or a split with uneven distribution of the entries reducing the storage utilization (fig 10a).

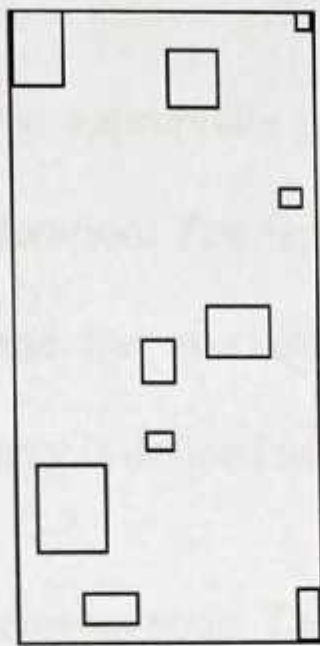


fig 10. Overfilled Node

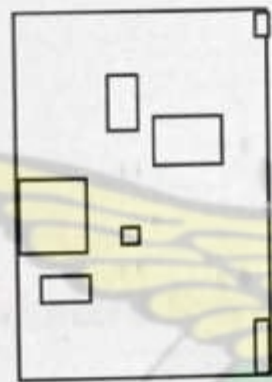
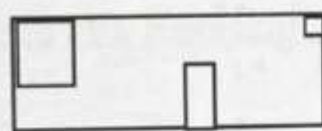


fig. 10a

Split of the
quadratic R-tree



fig. 10b

Split of the
quadratic R-tree

3.4 The Optimized R-Tree

An Optimized R-Tree is completely dynamic, insertions and deletions can be intermixed with queries and no periodic global reorganization is required. Obviously, the structure must allow overlapping directory rectangles. Thus, it cannot guarantee that only one search path is required for an exact match query as in the original R-Tree by Guttman.

To solve the problem of choosing an appropriate insertion path, previous R-tree versions take only the area parameter into consideration. The Optimized R-tree will take into considerations the area perimeters, the margins and the overlaps in different combinations with respect to insertion, where the overlap of an entry is defined as

Let E_1 , & E_p be the entries in the current node. Then

$$\text{overlap}(E_k) = \sum_{p=1, p \neq k}^p \text{Area}(E_k \text{ Rectangle} \cap E_p \text{ Rectangle}), \quad 1 \leq p \leq k$$

3.4.1 Algorithms for Optimized R-tree

The best insertion and retrieval method is described in the following algorithms using the relation above.

Algorithm ChooseSubtree (Select a leaf node in which to place a new index entry E)

CS 1 Set N to be the root

CS 2 If N is a leaf

Return N

Else

if the childpointer in N points to leaves (determine the minimum overlap cost) Choose the entry in N whose rectangle needs least overlap enlargement to include the new data. Resolve ties by choosing the entry whose rectangle needs least area enlargement

then

the entry with the rectangle of smallest area if the childpointer in N do not point to leaves(determine the minimum area cost),choose the entry in N whose rectangle needs least area enlargement to include the new data rectangle. Resolve ties by choosing the entry with the rectangle of smallest area

end

CS 3 *Set N to be the childnode pointed to by the childpointer of the chosen entry and repeat from CS2*

3.4.2 Node Splitting

The Optimized R-tree will use the following method to find good splits along each axis. The entries are first sorted by the lower value, then sorted by the upper value of their rectangles.

For each sort $M-2m+2$ distributions of the $M+1$ entries into two groups are determined, where the k -th distribution ($k = 1, (M-2m+2)$) is described as follows The first group contains the first $(m-1)+k$ entries, the second group contains the remaining entries. Three different approaches are considered to have a good split value for the node. These are: the area, the margin and the overlap and are defined as follows:

- **area value** $\text{area}[\text{bb}(\text{first group})] + \text{area}[\text{bb}(\text{second group})]$
- **margin value** $\text{margin}[\text{bb}(\text{first group})] + \text{margin}[\text{bb}(\text{second group})]$
- **overlap value** $\text{area}[\text{bb}(\text{first group}) \cap \text{bb}(\text{second group})]$

Where **bb** is the bounding box of a set of rectangles

The obtained values may be applied to determine a split axis or the final distribution (on a chosen split axis).

Node Splitting Algorithm

- S1 *Invoke ChooseSplitAxis to determine the axis, perpendicular to which the spit is performed.*
- S2 *Invoke ChooseSplitIndex to determine the best distribution into two groups along that axis.*
- S3 *Distribute the entries into two groups*

Algorithm ChooseSplitAxis

- CSA1 *For each axis*
- Sort the entries by the lower then by the upper value of their rectangles and determine all distributions as described above Compute S. the sum of all margin values of the different distributions*
- end*
- CSA2 *Choose the axis with the minimum S as split axis*

Algorithm ChooseSplitIndex

CS11 Along the chosen split axis, choose the distribution with the minimum overlap value

Resolve ties by choosing the distribution with minimum area value

Algorithm Insert

(starting with the leaf level as a parameter, to insert a new data rectangle)

- I1 Invoke ChooseSubtree. with the level as a parameter, to find an appropriate node N , in which to place the new entry E
- I2 If N has less than M entries, accommodate E in N
If N has M entries. invoke OverflowTreatment with the level of N as a parameter (for reinsertion or split)
- I3 If OverflowTreatment was called and a split was performed, propagate OverflowTreatment upwards if necessary
If OverflowTreatment caused a split of the root, create a new root
- I4 Adjust all covering rectangles in the insertion path such that they are minimum bounding boxes enclosing their children rectangles

Algorithm OverflowTreatment

OT1 ~~If the~~ level is not the root level and this is the first call of OverflowTreatment in the given level during the insertion of one data rectangle, then

 invoke ReInsert

else

 invoke Split

end

Algorithm ReInsert

- RI1 For all $M+1$ entries of a node N , compute the distance between the centers of their rectangles and the center of the bounding rectangle of N
- RI2 Sort the entries in decreasing order of their distances computed in RI1
- RI3 Remove the first p entries from N and adjust the bounding rectangle of N
- RI4 In the sort, defined in RI2, starting with the maximum distance equal (far reinsert) or minimum distance equal (close reinsert), invoke Insert to reinsert the entries

If a new data rectangle is inserted, each first overflow treatment on each level will be a reinsertion of p entries. This may cause a split in the node which caused the overflow if all entries are reinserted in the same location. Otherwise, splits may occur in one or more other nodes, but in many situations splits are completely prevented. The parameter p can be varied independently for leaf nodes and non-leaf nodes as part of performance tuning.

3.5 Constraints/problems

Designing an algorithm that could be best fitted for spatial data is a bit difficult because spatial data is static or dynamic in nature especially, working on overfull node. Much attention is needed to split the node and also maintain the data structure.

Another constraint could be algorithm **ChooseSubtree**, determining the minimum area cost and the minimum overlap cost to include new data. However, the success of these algorithms would be felt when it is put to test.

CHAPTER FOUR

ANALYSIS OF FINDINGS

4.0 Introduction

The R-tree is a dynamic structure. Thus, all approaches of optimizing the retrieval performance have to be applied during the insertion of a new data rectangle. The insertion algorithm calls two more algorithms in which the crucial decisions for good retrieval performance are made. The first is the algorithm ChooseSubtree, beginning in the root, descending to a leaf, it finds on every level the most suitable subtree to accommodate the new entry. The second is the algorithm Split it is called, If ChooseSubtree ends in a node filled with the maximum number of entries, M Split should distribute $M+1$ rectangles into two nodes in the most appropriate manner. In the following, the ChooseSubtree and Split-algorithms, suggested in available R-tree variants are analyzed and discussed.

In this section, the implementation of the algorithms presented in the last section will be discussed and give empirical evidence for their efficiency when compared to existing methods used. The description of implementation and the experimental setup is dedicated to an empirical analysis of the effects of insertion, splitting and deletion and how to improve the algorithms using heuristics similar to the ones used in other methods. Finally, the I/O (input/output) cost and query performance of insertion and deletion algorithms with respect to the control processing unit (cpu) time of execution proposed by Guttman in the original R-tree and other R-tree variants will also be compare. The analysis will preview how the Optimized R-tree will perform.

4.1 Variable m and M

M is the maximum number of entries which is usually given and m is the minimum number of entries in one node.

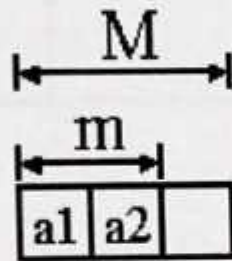


Figure 11: Representation of M and m

The minimum number of entries in a node is dependent on M with $M/2 \geq m$.

The maximum number of nodes is $\lceil N/m + N/m^2 \rceil + 1$

Here N stands for the number of index records of the R-Tree. m is jointly responsible for the height of an R-Tree and the speed of the algorithm. The choice of M depends on the hardware, especially on hard disk properties such as capacity and sector size. If nodes have more than 3 or 4 entries, the tree is very wide, and almost all the space is used for leaf nodes containing index records.

If m has a small value, it doesn't come so quickly to an overflow or an underflow so that the tree structure does not have to be reorganized. In the following example of deleting it comes to an underflow because of deleting $a3$. Here $M = 5$ and $m = 3$.

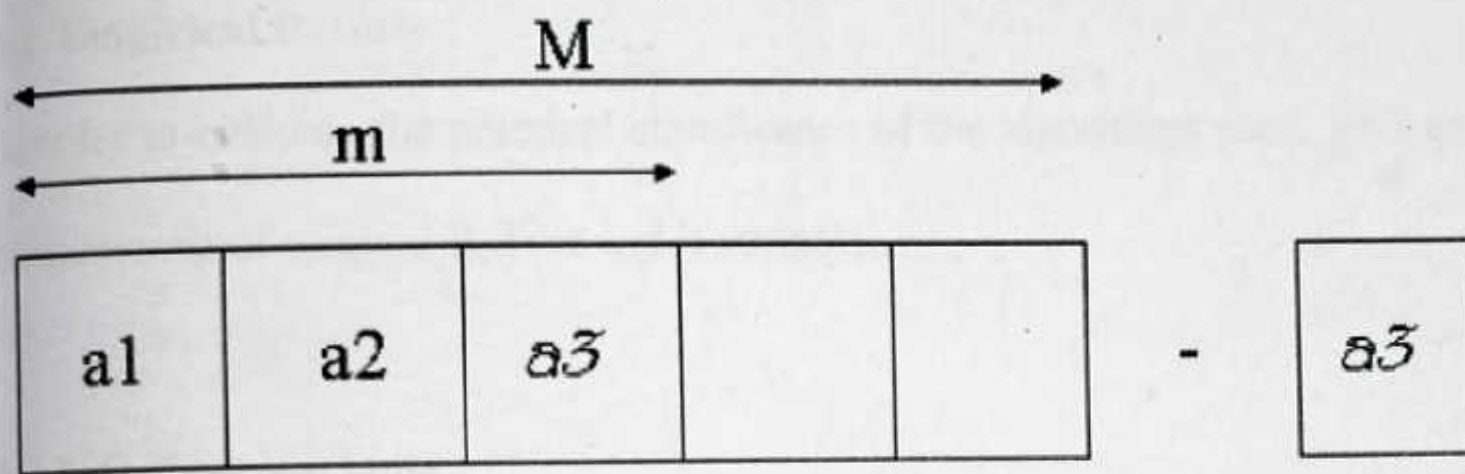


Figure 12: Underflow

There are now less than $m = 3$ entries in the node. Thus the tree has to be reorganized.

The other example explains the overflow through inserting.

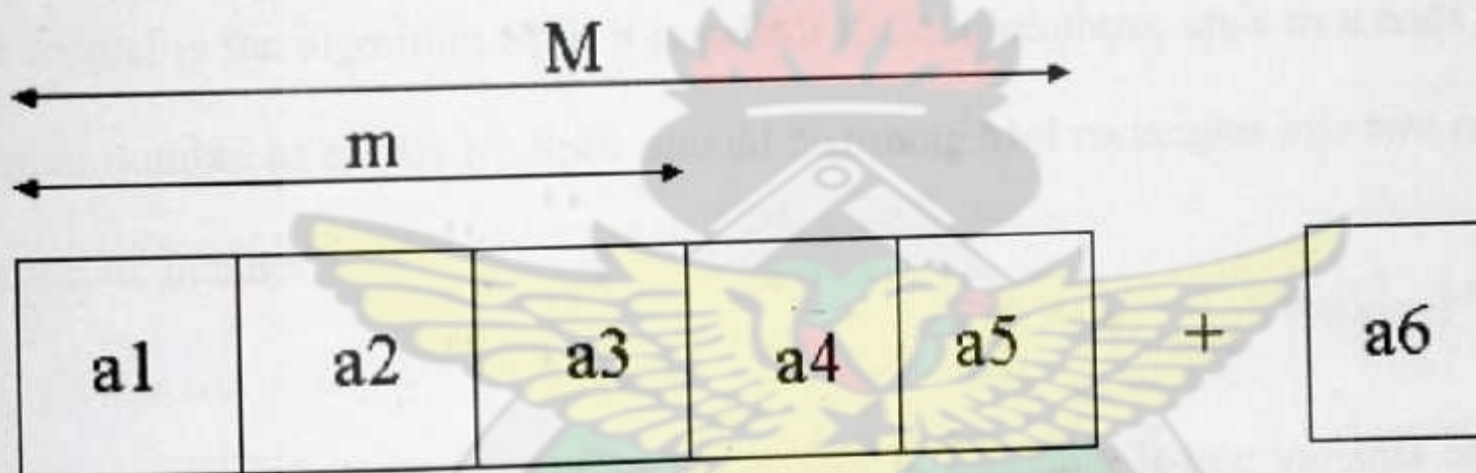


Figure 13: Overflow

Adding 1 to $M = 5$ entries thus result in reorganization.

4.2 Empirical Results

In order to evaluate the practical significance of the algorithms used, let's examine in detail the performance of original R-Tree and its variants.

4.2.1 R-Tree Variants

As already stated, the R-tree is a dynamic structure. Thus all approaches of optimizing the retrieval performance have to be applied during the insertion of a new data rectangle. The insertion algorithm calls two more algorithms in which the crucial decisions for good retrieval performance are made. The first is the algorithm **ChooseSubtree**, beginning in the root, descending to a leaf, it finds on every level the most suitable subtree to accommodate the new entry. The second is the algorithm **Split** it is called, If **ChooseSubtree** ends in a node filled with the maximum number of entries M , **Split** should distribute $M+1$ rectangles into two nodes in the most appropriate manner.

The **ChooseSubtree**- and **Split**-algorithms, suggested in available R-tree variants are analyzed and discussed. First, let's consider the original R-tree as proposed by Guttman in (Gut 84) and that of Greene's **Split** algorithms.

Algorithm ChooseSubtree

CS1: Set N to be the root

CS2: If N is a leaf,

return N

else

Choose the entry in N whose rectangle needs least area enlargement to include the new data. Resolve ties by choosing the entry with the rectangle of smallest area.

end

CS3: Set N to be the childnode pointed to by the childpointer of the chosen entry and repeat from CS2.

In an attempt to choose sub tree to insert a record starting from the root node, you may encounter a node that is full which will necessitate for a split in the overflow node. This will invoke split algorithms to accommodate the new record.

Obviously, the method of optimization in this approach was to minimize the area covered by a directory rectangle so that the cpu cost will be relatively low and the overlap of rectangles are reduced. Guttman discusses split-algorithms with exponential, quadratic and linear cost with respect to the number of entries of a node. All of them are designed to minimize the area, covered by the two rectangles resulting from the split.

The exponential split finds the area with the global minimum, but the cpu cost is too high. The others try to find approximations. In his experiments, Guttman obtains nearly the same retrieval performance for the linear and for the quadratic version. However, in implementing the optimized R-tree with different distributions, different overlap, variable numbers of data-entries and different combinations of M and m , the quadratic R-tree yielded much better performance than the linear version.

Algorithm Quadratic Split

This algorithm divide a set of $M+1$ entries into two groups.

QS 1: Invoke PickSeeds to choose two entries to be the first entries of the groups

QS 2: Repeat

DistributeEntry

until

all entries are distributed or one of the two groups has $M-m+1$ entries

QS 3: If entries remain, assign them to the other group such that it has the minimum number of m

Algorithm PickSeeds

PS 1: For each pair of entries $E1$ and $E2$, compose a rectangle R including $E1$ rectangle and $E2$ rectangle

Calculate $d = \text{area}(R) - \text{area}(E1 \text{ rectangle}) - \text{area}(E2 \text{ rectangle})$

PS 2: Choose the pair with the largest d

Algorithm DistributeEntry

DE 1: Invoke PickNext to choose the next entry to be assigned

DE:2 Add it to the group whose covering rectangle will have to be enlarged least to accommodate it. Resolve ties by adding the entry to the group with the smallest area, then to the one with the fewer entries, then to either

Algorithm PickNext

PN 1: For each entry E not yet in a group, calculate d_1 = the area increase required in the covering rectangle of Group 1 to include E Rectangle

Calculate d_2 analogously for Group 2

PN 2: Choose the entry with the maximum difference between d_1 and d_2

4.2.2 Analyzing the Quadratic Split Algorithm

The algorithm PickSeeds finds the two rectangles which would waste the largest area put in one group. In this sense the two rectangles are the most distant ones. It is important to mention that the seeds will tend to be small too, if the rectangles to be distributed are of very different size and or the overlap between them is high, the algorithm DistributeEntry assigns the remaining entries by the criterion of minimum area.

Algorithm PickNext chooses the entry with the best area value in every situation, if this algorithm starts with small seeds, problems may occur, if in $d-1$ of the d axes a far away rectangle has nearly the same coordinates as one of the seeds, it will be distributed first. Indeed, the area and the area enlargement of the created needle-like bounding rectangle will be very small, but the distance is very large. This may initiate a very bad split. Moreover, the algorithm tends to prefer the bounding rectangle created from the first assignment of a rectangle to one seed. Since it was enlarged, it will be larger than others. Thus, it needs less area enlargement to include the next entry, it will be enlarged again and so on. Another problem is that, if one group has reached the maximum number of entries $M-m+1$, all remaining entries are assigned to the other group without considering geometric properties.

The result is either a split with much overlap or a split with uneven distribution of the entries reducing the storage utilization.

In an attempt to comparing the R-tree with other structures storing rectangles, Greene proposed the following alternative split-algorithm (Greene 1989) To determine the appropriate path to insert a new entry. She uses Guttman's original ChooseSubtree-algorithm. Greene's algorithm is analyzed and discussed.

Algorithm Greene's-Split

This algorithm also divide a set of $M+1$ entries into two groups.

GS 1: Invoke ChooseAxis to determine the axis perpendicular to which the split is to be performed.

GS 2: Invoke Distribute.

Algorithm ChooseAxis

CA 1: Invoke PickSeeds as used by Gutman to find the two most distant rectangles of the current node.

CA 2: For each axis record the separation of the two seeds.

CA 3: Normalize the separations by dividing them by the length of the nodes enclosing rectangle along the appropriate axis.

CA 4: Return the axis with the greatest normalized separation.

Algorithm Distribute

- D 1: Sort the entries by the low value of then rectangles along the chosen axis
- D 2: Assign the first $(M+1) \div 2$ entries to one group, the last $(M+1) \div 2$ entries to the other.
- D 3: If $M+1$ is odd, then assign the remaining entry to the group whose enclosing rectangle will be increased least by its addition.

4.2.3 Analyzing Greene's Split Algorithm

Greene's split algorithm is the choice of geometric criterion to split axis. Although choosing a suitable split axis is important, it is anticipated that more geometric optimization criteria have to be applied to considerably improve the retrieval performance of the R-tree. Notably among them is the number of coordinates and the direction with respect to the topology. In spite of a well clustering, in some situations Greene's split method cannot find the "right" axis and thus a very bad split may result.

4.2.4 Analyzing the Optimized R-Tree algorithms

The optimized R-tree which is the basis for the subject of this research has the potential to perform better and improve upon the original R-tree proposed by Antoine Gutman. Making references to the algorithms designed in chapter three above, choosing the best non-leaf node, alternative methods did not outperform Guttman's original algorithm. For the leaf nodes, minimizing the overlap performed slightly better. In this algorithms the cpu cost of determining the overlap is quadratic when the number of rectangles are sorted in an increasing order to include a new data rectangle of their area enlargement. Because for each entry the overlap with all other entries of the node has to be calculated. However, for large node sizes you can reduce

the number of entries for which the calculation has to be done, since for very distant rectangles the probability to yield the minimum overlap is very small.

4.3 Performance Tests

The implementations of the Optimized R-tree was done in Java under windows in an Intel Pentium (R) dual core processors machine using two dimensional data. The purpose of this implementation was to verify the practicality of the structure and storage utilization. Some values were chosen for **M** and **m**, and to evaluate different node-splitting algorithms and organization of data in memory (that is, number of data stored, number of stored data pages in memory and the storage utilization of cpu cost). To this, six page sizes were chosen as shown in the table below.

Bytes per page	Maximum entry per page (M)
128	6
256	20
512	30
1024	40
2048	50
3096	55

Table : 1 Page sizes

For the values that was tested for **m**, the minimum number of entries in a node, were $M/2$, $M/3$, and 2. The algorithms described earlier were implemented in different versions of the program

varying the cache size. All the tests used two-dimensional data, though the structure and algorithms may work for any number of dimensions it was limited to two for the purposes of this research.

During the first part of each test run the program read geometry data from files using parameters area, margins and overlaps in different combinations and constructed an index tree, beginning with an empty tree and calling the function Insert with each new index record. The insertion performance was measured for the last 10% of the records, when the tree was nearly its final size.

During the second phase, the program called the function Search with search rectangles made up using **random numbers**. Queries with small query rectangles on datafiles with non-uniformly distributed small rectangles or points per test run, each retrieving about 5% of the data.

Finally the program read the input files a second time and called the function Delete to remove the index record for every fifth data item, so that measurements were taken for scattered deletion of 5% of the index records.

The split algorithm is tested with several values for M and m , and the results of the test is shown in the following pages.

4.3.1 Results of Inserting Data

The diagram below shows the cost in CPU time for inserting the last 10% of the records as a function of page size.

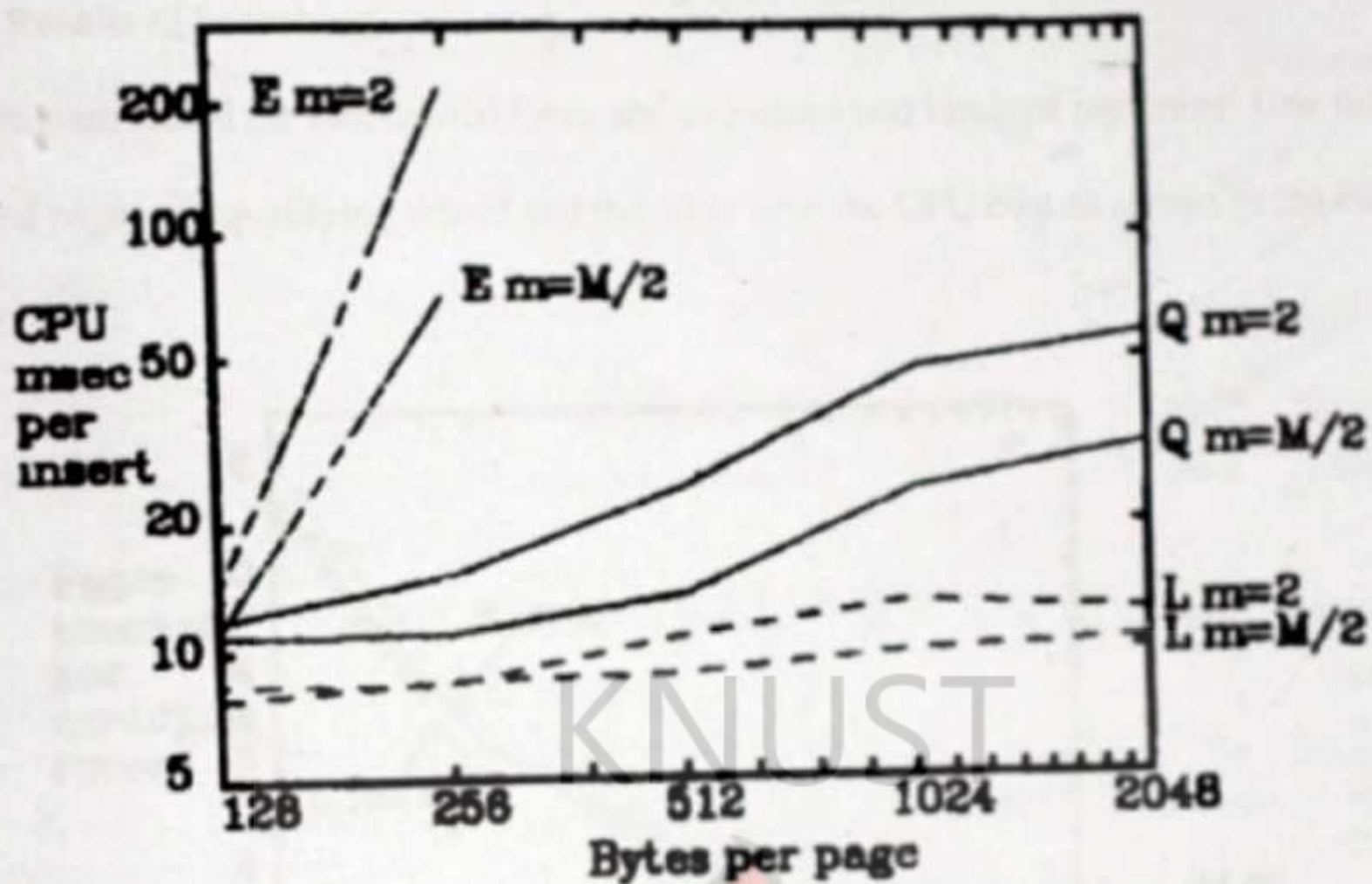


Figure 14: CPU cost of inserting data

The CPU time for inserting the last 10% of the records as a function of page size is as shown above. The quadratic (Q) algorithm, whose cost increases exponentially with page size, is seen to be very slow for larger page sizes. The linear (L) algorithm is fastest, as expected. With this algorithm CPU time hardly increased with page size at all, which suggests that node splitting was responsible for only a small part of the cost of inserting data. The decreased cost of insertion with a stricter node balance requirement reflects the fact that when one group becomes too full, all split algorithms simply put the remaining elements in the other group without further comparisons.

4.3.2 Results of Searching

The program called the function 10 times and examined two kinds of searching. One time the touched pages per qualifying record and the other time the CPU cost as shown in the diagram below

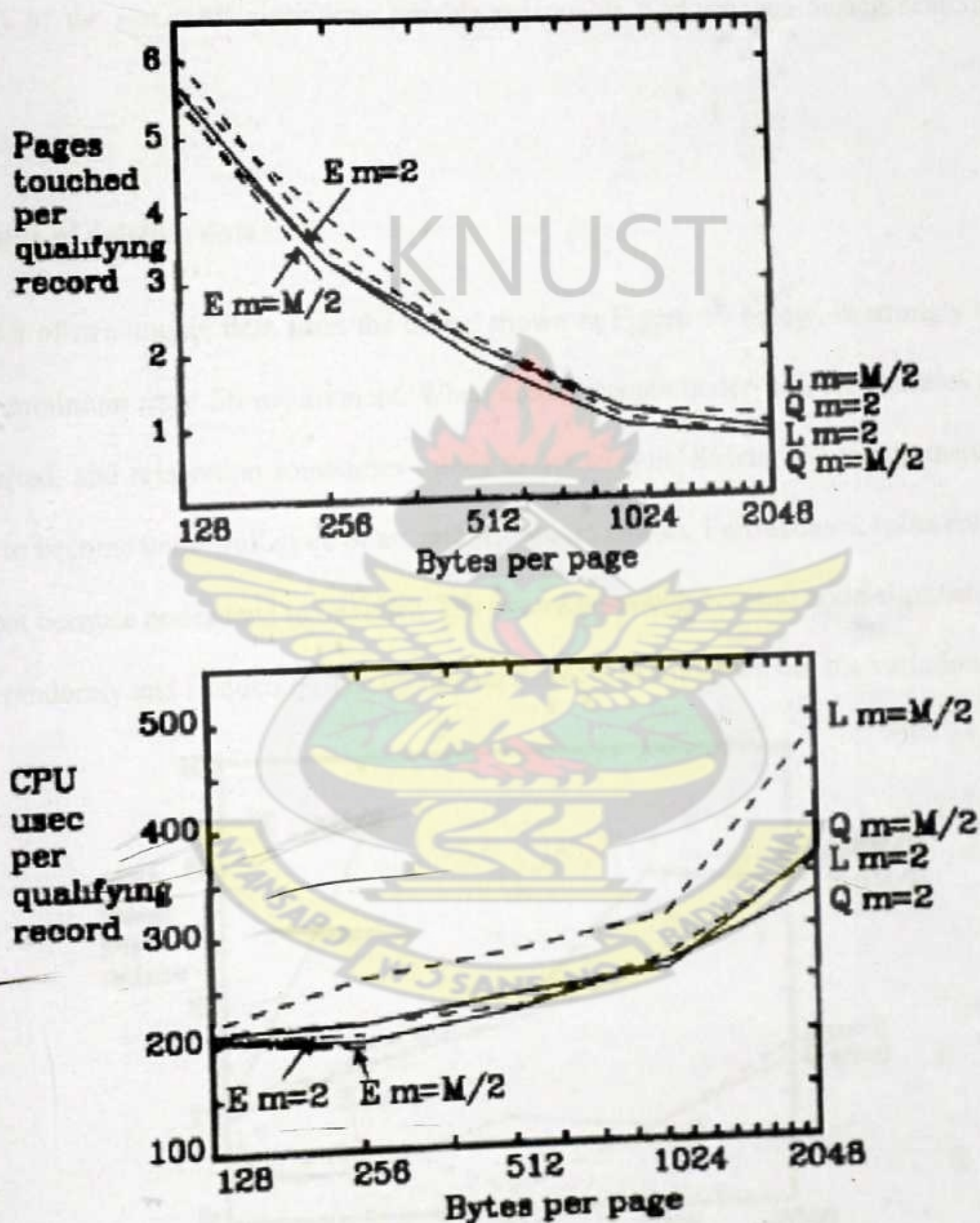


Figure 15: CPU Cost of Search performance

The Search algorithms as demonstrated in the two diagrams above show that the search performance of the index is very insensitive to the use of different node split algorithms and fill requirements. The algorithm produces a slightly better index structure, resulting in fewer pages touched and less CPU cost, but most combinations of algorithms and fill requirement come within 10% of the test. All algorithms provide reasonable performance during searching for data.

4.3.3 Results of deleting data

The cost of deleting an item from the index, shown in Figure 15 below, is strongly affected by the minimum node fill requirement. When nodes become under-full, their entries must be re-inserted, and reinsertion sometimes causes nodes to split. Stricter fill requirements cause nodes to become under-full more often, and with more entries. Furthermore, splits are more frequent because nodes tend to be fuller. The curves are rough because node eliminations occur randomly and frequently. The optimized R-tree will smoothen out the variations

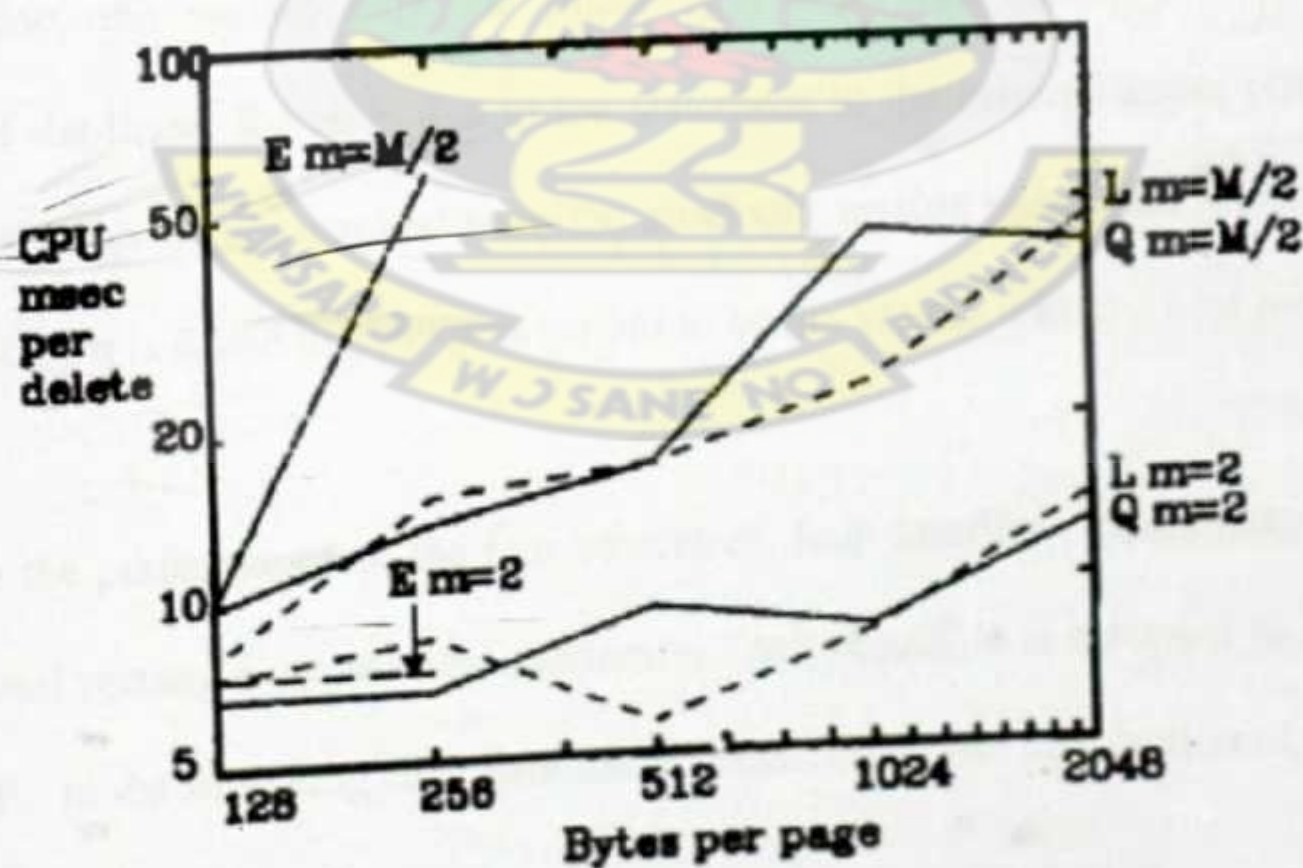


Figure 16: CPU cost of deleting records

4.4 Performance Comparisons

As stated earlier the implementations of the Optimized R-tree was done in Java under windows in an Intel Pentium (R) dual core processors machine using two dimensional data to achieve its objectives. In order to keep the performance comparison manageable, I have chosen the page size for data and directory pages to be 512 bytes which is at the lower end of realistic page sizes. Using smaller page sizes, I obtain similar performance results as for much larger file sizes.

From the chosen page size, the maximum number of entries in directory pages is 40. According to my standardized test, I have restricted the maximum number of entries in a data page to 50.

The R-tree with quadratic split algorithm is named as (qua Gut), Greene's variant of the R-tree (Greene) and the optimized R-tree as Optz where the parameters of the different structures are set to the best values as described in the previous sections (**that's the area value, the margin value and the overlap value**). Additionally, the most popular R-trees implementation, the variant with the linear split algorithm (in Gut) is analyzed. The popularity of the linear R-tree is due to the statement in the original paper (Gut84) that no essential performance gain resulted from the quadratic version versus the linear version. For the linear R-tree it is found that $m=20\%$ (of M) to be the variant with the best performance.

To compare the performance of the four structures, four datafiles containing about 200 in 2- dimensional rectangle are selected randomly. Each rectangle is assumed to be in the unit square $(0,1)^2$. In the following each data file is described by the distribution of the centers of the rectangles and by the tuple

$$(\mathfrak{n}, \mu_{\text{area}}, \mathfrak{nV}_{\text{area}})$$

Where, \mathfrak{n} denotes the number of rectangles,

μ_{area} is the mean value of the area of a rectangle and

$\mathfrak{nV}_{\text{area}} = \sigma_{\text{area}} / \mu_{\text{area}}$ is the normalized variance where

σ_{area} denotes the variance of the areas of the rectangles. Obviously, the parameter $\mathfrak{nV}_{\text{area}}$ increases independently of the distribution the more the areas of the rectangles differ from

the mean value and the average overlap is simply obtained by $\mathfrak{n} * \mu_{\text{area}}$,

The above scenarios are used to generate the following:

1. The centers of the rectangles follow a 2-dimensional independent uniform distribution ($\mathfrak{n} = 1,00$, $\mu_{\text{area}} = 0.01$, $\mathfrak{nV}_{\text{area}} = 9.52$) - "Uniform"
2. The centers follow a distribution with 20 clusters, each cluster contains about 6 Objects ($\mathfrak{n} = 99$, $\mu_{\text{area}} = 0.02$, $\mathfrak{nV}_{\text{area}} = 1.53$) - "Cluster"
3. The centers of the rectangles follow a 2-dimensional independent uniform distribution. First I take 100 small rectangles with $\mu_{\text{area}} = 0.0101$. Then 10 large rectangles are added with $\mu_{\text{area}} = 0.1$. Finally, these two datafiles are merged to one. ($\mathfrak{n} = 110$, $\mu_{\text{area}} = 0.002$, $\mathfrak{nV}_{\text{area}} = 6.77$) - "Mixed Uniform"

For each of the processes (1) - (3) queries are generated for the following three types of rectangles:

- **Rectangle intersection query:** Given a rectangle S , find all rectangles R in the file with $R \cap S \neq \emptyset$.
- **Point query:** Given a point P , find all rectangles R in the file with $P \in R$
- **Rectangle enclosure query:** Given a rectangle S , find all rectangles R in the file with $R \supseteq S$

Each of these files were used to performed 100 rectangle intersection queries where the ratio of the x-extension to the y-extension uniformly varies from 0.25 to 2.25 and the centers of the query rectangles themselves are uniformly distributed in the unit square.

In addition, four query files made of 50 rectangle intersection queries each were considered. The area of the query rectangles of each query file varies from 1%, 0.1% to 0.01% relatively to the area of the data space used.

For the range enclosure query, two query files where the corresponding rectangles are the same as in the query files were considered. Additionally, I analyzed a query file of 50 point queries where the query points are uniformly distributed.

Each of the query files were measured to determine the average number of disc accesses per query. The performance comparison used in the optimized R-tree as a measuring stick for the other access methods was to standardize the number of page accesses for the queries of the optimized R-tree to say 100%.

Thus, the performance of the other R-tree variants can be observed relative to the 100% performance of the optimized R-tree.

To analyze the performance for building up the different R-tree variants, the parameters considered were insert and storage (stor) utilization.

Where insert denotes the average number of disc accesses per insertion and **stor** denotes the storage utilization after completely building up the files.

The following table presents the results of the experiments depending on the different distributions (data files).

Uniform

	Point	Intersection			Enclosure		Stor	Insert
		0.01	0.1	1.0	0.01	0.1		
Lin Gut	25.8	88.7	83.0	55.5	78.7	88.3	65.0	7.43
Qua Gut	14.8	79.4	74.1	57.2	66.7	41.1	70.0	4.27
Greene	16.0	56.4	60.1	59.1	52.8	53.8	71.3	4.67
Optz R-tree	100	100	100.	100.	100	100	75.0	4.20
# accesses	5.26	7.63	13.29	53.42	4.85	3.66		

Table 2

Cluster

	Point	Intersection			Enclosure		Stor	Insert
		0.01	0.1	1.0	0.01	0.1		
Lin Gut	35.8	85.8	76.0	52.1	68.7	90.3	61.0	6.43
Qua Gut	66.8	64.4	58.1	45.2	66.4	84.1	66.0	4.87
Greene	60.0	51.3	44.1	51.1	42.8	53.2	69.3	4.47
Optz R-tree	100	100	100.	100.	100	100	76.0	3.66
# accesses	2.20	3.63	7.29	34.42	1.85	1.46		

Table 3

Mixed Uniform

	Point	Intersection			Enclosure		Stor	Insert
		0.01	0.1	1.0	0.01	0.1		
Lin Gut	87.7	82.8	72.8	62.3	79.7	92.3	62.4	14.63
Qua Gut	56.5	55.4	53.1	45.5	59.4	68.1	66.0	4.90
Greene	58.2	51.6	48.2	42.3	54.6	56.3	72.1	4.49
Optz R-tree	100	100	100.	100.	100	100	74.2	4.66
# accesses	4.75	7.25	15.29	53.58	4.65	3.61		

Table 4

The results shown for performance of building up different R-tree variants above indicate that given the storage utilization, the optimized R-tree yielded a better result in all the three tables.



CHAPTER FIVE

CONCLUSION AND RECOMMENDATION

5.0 Summary

The optimized R-tree is the most vigorous method which is undersigned by the fact that for every query file and every data file less disk accesses are required than by any other variants.

The gain in efficiency for smaller query rectangles is higher than for larger query rectangles, because storage utilization gets more important for larger query rectangles. This emphasizes the goodness of the order of preservation of the Tree (i.e. rectangles close to each other are more likely stored together in one page). The maximum performance gained taken over all query and data files in comparison is about 90%, and in spite of it shortfalls the cost of storage is low.

Concerning the cost of the split algorithm, the optimized R-tree requires a running time of $O(M \log(M))$ because for each axis (dimension) the entries have to be sorted two times. The margin in each axis as well as the rectangles and the overlap of the distributions have to be calculated.

Both, R-tree and the optimized R-tree are nondeterministic in allocating the entries onto the nodes i.e. different sequences of insertions will build up different trees. For this reason the R-tree suffers from its old node entries. Data rectangles inserted during the early growth of the structure may have introduced directory rectangles, which are not suitable to guarantee a good retrieval performance in the current situation. A very local reorganization of the directory rectangles is performed during a split. But this is rather poor and therefore it is desirable to have a more powerful and less local instrument to reorganize the structure. The discussed problem would be maintained or even worsened, if underfilled nodes, resulting from deletion of records would be

merged under the old parent. Thus, the known approach of treating underfilled nodes in an R-tree is to delete the node and to reinsert the orphaned entries in the corresponding level. Therefore to delete randomly half of the data and then to insert it again seems to be a very simple way of tuning existing R-tree datafiles, but this is a static situation and for nearly static datafiles the pack algorithm is a more sophisticated approach. To achieve dynamic reorganizations, the optimized R-tree forces entries to be reinserted during the insertion routine. A better way to address this is the ChooseSubtree algorithm that has a new chance of distributing entries into different nodes.

The result was a performance improvement of 20% up to 40% depending on the types of the queries that is performed. Obviously, the cpu cost will be higher now since the insertion routine is called more often. This is alleviated, because less splits have to be performed. The tests show that the average number of disc accesses for insertions increases only about 4% and remains the lowest of all R-tree variants. The usage of memory organization is particularly due to the structure improving properties of the insertion algorithm.

5.1 Conclusion

The optimized R-tree structure has been shown to be useful for indexing spatial data objects and multi dimensional points in database systems organizing. Although the optimized R-tree outperforms its competitors, the cost for the implementation of the structure is only slightly higher than the other R-trees. Nodes corresponding to disk pages of reasonable size have values of M that decides the maximum number of entries and the variable m the minimum number of entries in a node produce good performance in memory organization. The maximum performance gained, taken over all query and data files into consideration is due to the smaller

query rectangles that become formatted. The concepts are based on the reduction of area, margin and overlap of the directory rectangles. In addition, the split algorithm is very rigorous against unsightly data distributions in particular ChooseSubtree. This make the structure reorganizes dynamically and the storage utilization is slightly higher than other R-tree variants. Notwithstanding the benefit of this research, there are still more to be done in topology because in real world spatial objects are more static.

5.2 Problems

The implementations was done using machine with a limited memory size which does not allow over 200 rectangles to be queried and so updating the structure is limited. What happens is that for rectangles exceeding 200, the system turns out to be slower.

5.3. Recommendations

I wish to recommend the following for consideration:

- Acquire large machine with big memory size for research.
- Need a machine with a memory size of one terabyte and cpu processing speed also one terabyte.
- Many new variants of R-tree must follow engineering approach to improve complex applications.

Bibliography

- A Practical Introduction to Data Structures and Algorithm Analysis Clifford A. Shaffer. Prentice-Hall (1997).
- Ang, C. H.; Tan, T. C. (1997). "New linear node splitting algorithm for R-Trees". In Scholl, Michel; Voisard, Agnès. *Proceedings of the 5th International Symposium on Advances in Spatial Databases (SSD '97), Berlin, Germany, July 15–18, 1997*. Lecture Notes in Computer Science. 1262. Springer. pp. 337–349. doi:10.1007/3-540-63238-7-38
- Bayer, R., M. Schkolnick. Concurrency of Operations on B-Trees. In *Readings in Database Systems* (ed. Michael Stonebraker), pages 216-226, 1994.
- Brinkhoff, T.; Kriegel, H. P.; Seeger, B. (1993). "Efficient processing of spatial joins using R-Trees". *ACM SIGMOD Record* 22: 237. doi:10.1145/170036.170075
- C. Aggarwal, J. Wolf, P. Wu and M. Epelman: "The S-tree - an Efficient Index for Multidimensional Objects", *Proceedings 5th SSD Conference*, pp.350-373, Berlin, Germany, 1997.
- Comer, Douglas (June 1979), "The Ubiquitous B-Tree", *Computing Surveys* 11 (2): 123–137, doi:10.1145/356770.356776, ISSN 0360-0300
- C.H. Ang and T.C. Tan: "Bitmap R-trees", *Informatica*, Vol.24, No.2, 2000.
- Data Structures & Program Design, 2nd ed. Robert L. Kruse. Prentice-Hall (1987). There is also a third edition (1994) as well as a new text, *Data Structures and Program Design in C++*, authored by Kruse and Alexander J. Ryba (1999).
- D.J. Park, S. Heu and H.J. Kim: "The RS-tree - an Efficient Data Structure for Distance Browsing Queries", *Information Processing Letters*, Vol.80, pp.195-203, 2001.

- D. Knuth: "The Art of Computer Programming: Sorting and Searching", Vol.3, Addison-Wesley, 1967.
- D. Pfoser, C.S. Jensen and Y. Theodoridis: "Novel Approaches to the Indexing of Moving Object Trajectories", *Proceedings 26th VLDB Conference*, pp.395-406, Cairo, Egypt, 2000.
- Folk, Michael J.; Zoellick, Bill (1992), *File Structures* (2nd ed.), Addison-Wesley, ISBN 0-201-55713-4
- Gray, J. N., R. A. Lorie, G. R. Putzolu, I. L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *Readings in Database Systems* (ed. Michael Stonebraker), pages 181-208, 1994.
- (Gre 89) D Greene 'An Implementation and Performance Analysis of Spatial Data Access Methods', Proc 5th Int. Conf. on Data Engineering. 606-615, 1989
- Guttman: R-Trees: A dynamic index structure for spatial searching, Proceedings of the 1984 ACM SIGMOD international conference on Management of data – SIGMOD doi:10.1145/602259.602266. ISBN 0897911288. <http://www-db.deis.unibo.it/courses/SL-LS/papers/Gut84.pdf>. edit.
- (Hm 85) K Hinrichs 'The grid file system implementation and case studies for applications', Dissertation No 7734, Eidgenossische Technische Hochschule (ETH), Zuerich. 1985.
- Hwang, S.; Kwon, K.; Cha, S. K.; Lee, B. S. (2003). "Performance Evaluation of Main-Memory R-Tree Variants". *Advances in Spatial and Temporal Databases*. Lecture Notes in Computer Science. 2750. pp. 10. doi:10.1007/978-3-540-45072-6_2. ISBN 978-3-540-40535-1.

- H.V. Jagadish: "Spatial Search with Polyhedra", *Proceedings 6th IEEE ICDE Conference*, pp.311-319, Orlando, FL, 1990.
- Introduction to Algorithms, Ronald L. Rivest, Thomas H. Cormen, Charles E. Leiserson, and McGraw-Hill (1990). See chapter 19.
- I. Kamel and C. Faloutsos: "Hilbert R-tree - an Improved R-tree Using Fractals", *Proceedings 20th VLDB Conference*, pp.500-509, Santiago, Chile, 1994.
- I. Kamel and C. Faloutsos: "On Packing R-trees", *Proceedings 2nd CIKM Conference*, pp.490-499, Washington, DC, 1993.
- Kemper/A. Eickler: *Datenbanksysteme, Eine Einfuehrung*, 4. Au-age, S, 2001
- Knuth Donald (1997), *Sorting and Searching*, The Art of Computer Programming, Volume 3 (Third ed.), Addison-Wesley, ISBN 0-201-89685-0. Section 6.2.4: Multiway Trees, pp. 481–491. Also, pp. 476–477 of section 6.2.3 (Balanced Trees) discusses 2-3 trees.
- Lars Arge, Mark de Berg, Herman J. Haverkort, Ke Yi: "The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R Tree", *SIGMOD 2004*, June 13–18, Paris, France
- Lars Arge, Mark De Berg, M.; Haverkort, H. J.; Yi, K. (2004). "The Priority R-Tree". *Proceedings of the 2004 ACM SIGMOD international conference o Management of data - SIGMOD '04*. pp. 347. doi:10.1145/1007568.1007608. ISBN 1581138598.
<http://www.win.tue.nl/~mdberg/Papers/prtree.pdf>.
- L. Arge, K. Hinrichs, J. Vahrenhold and J.S. Vitter: "Efficient Bulk Operations on Dynamic R-trees", *Algorithmica*, Vol.33, No.1, pp.104-128, 2002.

- L. Chen, R. Choubey and E.A. Rundensteiner: "Bulk-Insertions into R-trees Using the Small-Tree-Large-Tree Approach", *Proceedings 6th ACM GIS Conference*, pp.161-162, Washington, DC, 1998.
- M. Astrahan, et al, System R* Relational Approach to Database Management, *ACM Transaction on Database Systems* 1, 2 (June 1976)
- M. Stonebraker, T. Sellis and E. Hanson: "An Analysis of Rule Indexing Implementations in Data Base Systems", *Proceedings 1st Conference on Expert Database Systems*, pp.465-476, Charleston, SC, 1986.
- M. Vazirgiannis, Y. Theodoridis and T. Sellis: "Spatio-temporal Composition and Indexing in Large Multimedia Applications", *Multimedia Systems*, Vol.6, No.4, pp.284-298, 1998.
- N. Beckmann, H. P. Kriegel and B. Seeger: The R*-Tree: An efficient and robust method for points and rectangles, *Proceedings of the 1990 ACM SIG-MOND Conference*, Atlantic City, NJ, 1990
- N. Roussopoulos and D. Leifker: "Direct Spatial Search on Pictorial Databases Using Packed R-trees", *Proceedings ACM SIGMOD Conference*, pp.17-31, Austin, TX, 1985.
- O. Guenther: "The Cell Tree - an Object Oriented Index Structure for Geometric Databases", *Proceedings 5th IEEE ICDE Conference*, pp.598-605, Los Angeles, CA, 1989.
- O. Guenther and H. Noltemeier: "Spatial Database Indices for Large Extended Objects", *Proceedings 7th IEEE ICDE Conference*, pp.520-526, Kobe, Japan, 1991.
- P. Bozanis, A. Nanopoulos and Y. Manolopoulos: "LR-tree - a Logarithmic Spatial Index Method", *The Computer Journal*, accepted.

- P. Oosterom: "Reactive Data Structures for Geographic Information Systems", Ph.D. Dissertation, University of Leiden, 1990.
- P.K. Agarwal, M. deBerg, J. Gudmundsson, M. Hammar and H.J. Haverkort: "Box-trees and R-trees with Near Optimal Query Time", *Proceedings Symposium on Computational Geometry*, pp.124-133, Medford, MA, 2001.
- P.W. Huang, P.L. Lin and H.Y. Lin: "Optimizing Storage Utilization in R-tree Dynamic Index Structure for Spatial Databases", *Journal of Systems and Software*, Vol.55, pp.291-299, 2001.
- S. Berchtold, D.A. Keim, H.-P. Kriegel: The X-Tree: An index structure for high-dimension data, Proceedings of the 22nd International Conference on Very Large Databases, Bombay, India, 1996
- (SK 88) B Seeger, H P Kriegel. 'Design and implementation of spatial access methods', Proc 14th Int. Conf. on Very Large Databases, 360-371, 1988
- (SK 90) B Seeger. H P Kriegel 'The design and implementation of the buddy tree', Computer Science Technical Report 3/90, University of Bremen. submitted for publication, 1990.
- Scott T. Leutenegger, Jeffrey M. Edgington and Mario A. Lopez: STR: A Simple and Efficient Algorithm for R-Tree Packing
- T. Schrek and Z. Chen: "Branch Grafting Method for R-tree Implementation", *Journal of Systems and Software*, Vol.53, pp.83-93, 2000.
- T. Sellis, N. Roussopoulos and C. Faloutsos: The R+-Tree: A dynamic index for multidimensional objects, Proceedings of the 13th VLDB Conference, p.507-518, UK, 1987

- V. Gaede and O. Gunther. Multidimensional access methods. ACM Computing Surveys, 30(2):170{231, June 1998.
- Y. Garcia, M. Lopez and S. Leutenegger: "On Optimal Node Splitting for R-trees", Proceedings 24th VLDB Conference, pp.334-344, New York, NY, 1998.
- Y.J. Lee and C.W. Chung: "The DR-tree - a Main Memory Data Structure for Complex Multidimensional Objects", *Geoinformatica*, Vol.5, No.2, pp.181-207, 2001.
- Y. Manolopoulos; A. Nanopoulos; Y. Theodoridis (2006). *R-Trees: Theory and Applications*. Springer. ISBN 978-1-85233-977-<http://books.google.com/books>

