MULTIATTRIBUTE INDEXING
USING
MULTIDIMENSIONAL DATA STRUCTURES

By

YUSUF GARBA DAMBATTA

Submitted to the Institute of Graduate Studies in Science and Engineering

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Mevlana (Rumi) University

2014

MULTIATTRIBUTE INDEXING

USING

MULTIDIMENSIONAL DATA STRUCTURES

Submitted by **Yusuf Garba Dambatta** in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering, Mevlana (Rumi) University

APPROVED BY:

Examining Committee Members:

Asst. Prof. Dr. Armagan Ozkaya . . . . . .
Thesis Supervisor

Asst. Prof. Dr. Mustafa Kaiiali . . . . . .

Asst. Prof. Dr. Mesut Gündüz . . . . . .

Assoc. Prof. Dr. Halis Altun . . . . . .
Head, Department of Computer Engineering

Assoc. Prof. Dr. Ali Sebetci . . . . . .
Director, Institute of Graduate Studies in Science and Engineering

DATE OF APPROVAL (/ /2014):

DECLARATION

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Yusuf Garba DAMBATTA

Signature:

ABSTRACT

MULTIATTRIBUTE INDEXING
USING
MULTIDIMENSIONAL DATA STRUCTURES

By

Yusuf Garba Dambatta

M.Sc. Thesis, 2014

Thesis Supervisor: Asst. Prof. Dr. Armagan Ozkaya

Keywords: Relational Database, Query, Attribute, Composite Key, Index, Multidimensional Data Structure, R-tree.

Many applications involve searches using values of several of their attributes. Indexes are well-known data structures utilized to improve the performance of searches for data. A multiattribute index has advantages over several single-attribute indexes. First, the clustering of index terms reduces the number of I/O access needed for the search. Second, multiattribute index requires single update when new record is inserted.

Several data structures have been used for multiattribute key index. B+-tree is commonly used as multiattribute key index, but has the disadvantage that it does not allow search on some of the attributes from the multiattribute key. Grid File allows search on all the attributes from the multiattribute key while restricting the keys in the index to only contain uniform values. Insertion and deletion can also be difficult on grid files. There has been previous work that uses R-trees for indexing purposes where the emphasis was on spatial data. The research in this thesis focuses on the relational data which exploits relational databases and the use of multidimensional data structures to perform multiattribute key indexing. It examines how to employ R-trees to perform multiattribute indexing such that the order of attributes is no more important for queries. Data records with multiattribute keys are modeled as multidimensional

data to be indexed by means of a multidimensional data structure, specifically an R-tree. This will enable a relational database system to perform queries using any one of the attributes or any of their combination. It is hereby shown how a multiattribute key index implemented by an R-tree facilitates retrieval of records from database in response to search conditions based on any of the attributes forming the key or any combination thereof. An improvement on R-tree is then presented where regions do not overlap and n-dimensional signatures are incorporated into nodes of the tree for effective filtration of irrelevant tree nodes during searches. Algorithms for search (point, range and similarity), insertion, and deletion operations are also provided.

DEDICATION

I dedicate this thesis to my able Governor Engr. (Dr.) Rabi'u M. Kwankwaso whose contribution toward this academic achievement will remain in my memory forever, along with my encouraging parents, family and friends whose affection, encouragement and prayer led me to this success.

ACKNOWLEDGEMENT

All praises be to Allah, the exalted and the merciful. All His glory and blessing are upon our noble Prophet. I wish to express my sincere gratitude and appreciation to God for keeping me alive to attain this level in my professional career. Special appreciation goes to my supervisor Asst. Prof. Dr. Armagan Ozkaya whose suggestions, recommendations and criticisms played a vital role toward the successful completion of this thesis. I thank my parents for their continuous support and prayers. I am also grateful to all the members of Garba Dambatta's family, specifically my elder brother Dr. Auwal Garba. Finally, my special regards and thanks go to all my friends who stood by me at all the time especially my dear friend Muzammil Abdulrahman.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS/ABBREVIATIONS

Symbol	Explanation
δ	n-dimensional discrete space
α	Constant greater than or equal to min(domain M)
β	Constant less than or equal to max(domain M)
τ	Length of binary representation
n	Dimensions
CR	Relevance Ratio
N	Node
NIR	Number of Intersecting Regions
NRR	Number of Relevant Regions
M	Domain M
MBB	Minimum Bounding Box
P	Pointer
QB	Query Box
QH	Query High
QL	Query Low
S	Signature

CHAPTER ONE

1. INTRODUCTION

1.1 Overview

An index is any data structure that improves the search for a data. Indexing technique is associated with most database applications as it enables sub-linear lookup and improves performance because linear search is inefficient for large databases [1]. The other essential database operations are insertion of a new record and deletion of a possibly existing record. These operations also make use of searching as the insertion may require checking the presence of a record and the deletion may need to determine the location of a record as well as its presence. A cost of additional space overhead associated with using indexes.

There are several different data structures used for indexing purposes [2,3,4,5]. The design and analysis of such data structures has been a major subject of discussion in the field of computer science. B-tree and its B+-tree variation are among the most popular and efficient index structures proposed. Several access algorithms have been devised on these with the aim of increasing the concurrency in accessing the index and the data items and minimizing average access delay [6,7,8,9]. Though B+-tree index structure has been proved to be efficient on traditional database systems, it cannot be successfully applied to multiattribute keys. A multiattribute key index created on a B+-tree indexes the attributes forming the key as if it were a single attribute [10]. The order of attributes specified when creating such an index is important and needs to be considered when using this kind of index in queries. A B+-tree is then essentially a single attribute index. It indexes one attribute or concatenation of attributes in case of a multiattribute key index and that is why it can be thought of as a one dimensional index structure. The order of attributes in multiattribute key index is important because the data is sorted on a single dimension (i.e. the first attribute). If for instance a multiattribute key index has two attributes forming the key, a search can only be made based on the first attribute or on both attributes together. A search using the index based on the second attribute alone is impossible due to the ordering of the attributes in the B+-tree index structure.

The main objective of this research is to propose a multiattribute key index structure supporting not only simple but complex queries on any of the attributes forming the multiattribute key irrespective of their order in the key. To achieve the stated objective, a

multidimensional approach is introduced for data retrieval. The basic idea involves modeling the tuples of a key formed by multiple attributes as n-dimensional points. The multidimensional approach allows processing queries such as point and range queries regardless of the order of the individual attributes. Because multiattribute keys are to be represented by points in a multidimensional space in proposed scheme here, a multidimensional data structure will be used. A number of approaches to indexing data based on multiple attributes are reported in the literature and we shall review some of them.

The data structure chosen for the approach proposed herein is the R-tree which is nowadays used for multidimensional data in both theoretical and applied context [11]. The focus, however, is on the relational data which exploits relational databases. The R-tree structure will have non-overlapping minimum bounding boxes and will be augmented with the use of signatures created from key attributes in an effort to speed up the search procedure by prohibiting unnecessary visits to sub-trees, especially for processing of narrow range queries [12]. Hence, we end up with employing "Signature R-tree" data structure, whose regions do not overlap and which allows execution of multidimensional data queries, such as point and range queries, to index relational data with multiattribute keys.

1.2 Index

A database index is a data structure that speeds up retrieval of records from database at the cost of additional storage space for keeping the extra index file. Indexes are used to locate data quickly without scanning each and every row in the database table whenever query is executed. They are built from one or more table columns from the database which provides rooms for efficient access of records and fast random searches [10,13].

Indexes are generally stored in separate files on disk and records are accessed using index fields in the file. They do not alter the record placement in the main data file. Ultimately, an index can be created on any field of the data file while multiple indexes can also be created on multiple fields on the same file. Several indexes can be created for a single database table with each using a specific data structure to speed up searching for records. When a query is submitted to search for a record, first an index is searched based on the search condition on an indexing field. The index has pointers to one or more disk blocks in the data file where records are placed. The most common indexes are those based on the ordered file (also called single-level indexes) and tree data structures like B+-trees [13].

1.3 Multiattribute Key Index

While a relational database system can create index for a primary key automatically, it is also possible to change the keys manually when the key is made up from multiple columns. In such situations database creates a single index for all primary key columns which is also known as concatenated or multiattribute key index. A multiattribute key is a key that uniquely identifies occurrence of an entity including two or more attributes in a database. There are two types of multiattribute keys:

- Compound key where each and every attribute that makes up the key is a simple key in its own right.
- Composite key where at least one attribute that makes up the key is not a simple key in its own right.

A multiattribute key index created using B-tree structure indexes the attributes as if they were a single one. Indexed data are kept in a sorted list. The database uses the position of the attributes to categorize index entries. The first attribute is the main sort criterion while the second attribute is only used to determine the order when two entries have the same value on the first attribute and the so forth [13]. The ordering of two attributes in such a multiattribute index is like ordering of a telephone directory: It is first sorted by surname, then by first name which means that a two attribute index does not support search on the second attribute alone (i.e. searching telephone directory based on first name alone).

Whenever a query involving the complete primary key is executed, the database uses index scan to answer such queries. But, it refuses to use the index for queries involving some of the columns that formed the primary key. Instead, the database does a full table scan to answer the query. It reads the entire column evaluating each and every row against the "where" clause. The execution time for this type of query increases with an increase in table size. This kind of operation might be adequately fast in a small development environment, but becomes slow in a large environment which greatly affects system performance. The ability of an SQL statement to use a multiattribute index depends on the attributes contained in its "where" clause. A query uses a multiattribute index only if it references the leading portion of the index in its "where" clause. A leading portion of the index means the first attribute or attributes mentioned in the "create index" statement. The order of columns in such an index plays a vital role on its usage so it must not be chosen arbitrarily [14]. A multiattribute key index with three attributes can only be used to efficiently execute queries on the first attribute,

first two attributes together, or all the three attributes. A popular approach in many commercial database systems to handle general multiattribute search queries is by the consecutive application of single-attribute indexes. This approach, unfortunately, can be very inefficient. Since each index is traversed independently of the others, we cannot exploit the possibly high selectivity in one dimension for narrowing down the search in remaining dimensions. Several single-attribute indexes require many updates of its index when new record is inserted and also have higher numbers of I/O processes than a multiattribute index.

In general, there is no easy and obvious way to extend single key structures, such as B+-trees, to handle multidimensional data. A single index is preferable, however, because it will save space and result in less maintenance overhead. If a database table has fewer indexes on it, operations that change the data set such as insertion, deletion, update, etc. perform better because these operations may require reorganizing the index data structure [15].

1.4 Multiattribute Keys as n-dimensional Points

In the approach presented here, a multiattribute key obtained from the relational data is modeled as n-dimensional points in n-dimensional vector space, where n is the number of attributes forming the key. The n-dimensional vector space is called multiattribute key space where multiattribute keys are distinctively denoted by tuples of attributes as their co-ordinate values that may be obtained from a fixed alphabet like ASCII code character [16].

1.5 Query Types

The most common application of data structures in a database is the query, which enables the user to specify the desired data, leaving the database management system to perform the physical operations necessary to produce the result. Different data structures support different types of queries. In the case of R-tree data structure, it supports the following types of queries [16]:

- Point
- Range
- K-Nearest-Neighborhood (KNN)

1.6 Signature File Methods

This is a commonly used access method employed in applications requiring large storage space of document databases like medical information and office information systems [17, 18]. For that reason, signature file methods became famous for executing retrieval operations

on data files. It was later expanded to carry multi-media data like video and images [19]. Several of the DBMS that are used nowadays support multimedia data and hence need a dynamic data structure that supports insertion, deletion, update, and retrieval operations efficiently. Among the dynamic data structures proposed are the S-Tree [20] and quick filter [21].

The signature file is a generalized concept that is used as a filter to minimize disk block access and processing time when a query is executed. A signature is a bit string formed from multiattribute keys that are used to index records in a data file. Signature file methods mostly use a superimposed coding technique to create record signatures [22]. The method works by dividing n-dimensional space into regions with each region containing points or tuples representing multiattribute keys. Each point of the n-dimensional space produces bit strings that are OR'ed together to form block signatures. Assuming the record consists of *m* multiattribute keys and the multiattribute keys formed from *n* attributes, each attribute is converted into bit strings called the attributes' signatures that are combined to form the multiattribute keys signature. The record signature is formed by superimposing (inclusive OR'ing) the *n* multiattribute keys' signatures. The number of 1s in a signature is called the weight of the signature.

To process a query, we first examine the signature file instead of the data file in order to reject the non-qualifying records. In order to achieve that, a set of terms in a query are hashed in the same way we did for record signature to form a query signature. If a record signature contains ones in the same positions as query signature, it would be regarded as a possible match and can be used for the query. A bitwise AND operation is utilized for the above decision. Nevertheless, there can be a situation where a particular record qualifies for a query signature, but still does not satisfy the query. This condition is referred to as a *false drop*.

1.7 Problem Definition

Indexing is a method that involves the use of a particular data structure to improve the speed of data retrieval operations on database tables as the linear search is inefficient for large databases. Multiattribute key indexing is an indexing where multiple attributes that form the primary or a unique key of a table are used for indexing data records. Many data structures have been used for multiattribute key indexing, a popular one of which is the B^+-tree data structure. If a B+-tree index is formed on a multiattribute key, the database does not use the index to perform look up involving attributes arbitrarily. B^+ -tree keeps attributes in a sorted

list, thus the order of attributes is important and should be considered when writing queries. A B+-tree multiattribute key index does not allow search on some of the attributes from the multiattribute key. Grid File allows search on all the attributes from the multiattribute key, but it restricts the keys in the index to only contain uniform values. Insertion and deletion can also be difficult on grid files [25].

This research will focus on the use of R-tree data structure to perform multiattribute key indexing on relational data that is stored in relational database. A multiattribute key index based on the R-tree data structure allows retrieval of records from database in response to certain search conditions based on any of the attributes forming the primary key or any combination thereof. This R-tree structure will utilize non-overlapping minimum bounding boxes and signatures created from key attributes in an effort to speed up the search procedure by prohibiting unnecessary visits to sub-trees.

1.8 Thesis Organization

Chapter 1 introduces this research and contains the overview of the topic, explicit description of the problem, explanation of the purpose and types of indexing, and the proposed approach to multiattribute key indexing.

In Chapter 2, we review related literature and R-trees data structure is described together with one of its types called Signature R-trees which is the data structure selected for use in this research. Operations on these structures, query construction, and query types are also discussed in this chapter.

Chapter 3 describes multidimensional approach to multiattribute key indexing for query processing. It also contains information on indexing data as a multidimensional problem, multiattribute key as n-dimensional point, indexing and querying tuples in R-trees and cost analysis. It also contains algorithms to perform operations/queries such as search, insertion and deletion in the Signature R-trees.

Finally, Chapter 4 concludes the work by summarizing its contribution and provides possible areas for future work.

CHAPTER TWO

2. MULTIDIMENSIONAL DATA STRUCTURES

2.1 Overview

With an increase in computer applications that depend greatly on managing multidimensional data, spatial data management has now become an important research area among database community. The applications range from CAD, VLSI, geographical databases to multi media management system. Complete information about multidimensional indexing and data structures can be found in [23,24,25]. Bounding Universal B-Tree (BUB) offers good performance for indexing spatial data. It is a paged and balanced multidimensional index structure utilizing the Z-ordering and B+-Tree [26]. This study avoids using BUB-Tree in work, because it is inefficient for range queries. Instead, R-tree data structure augmented with signatures and non-overlapping regions to improve searches is used for indexing multidimensional data in this study. R-tree has become the most widely used data structure, since the time when Antonin Guttman proposed this dynamic index structure for spatial searching. This data structure supports the usual point queries, some form of spatial joins, and also KNN queries to some degree [27,28]. The focus was made on the appropriate placement of index records in the tree, in order to reduce the number of paths needed to be traversed when searching for or deleting data object. Later in [29,32], concurrency control is considered. These are variations of R-tree in which several parallel processes can run at the same time.

R-tree index structures improve access efficiency for spatial data. It is popularly employed nowadays as an index structure in many applications involving spatial data. For example, in [34], a new approach to creating a spatial index with R-tree was proposed, and in [35], an improvement of index method based on R-tree was proposed to boost query efficiency with the strategy of increasing the space to reduce the time. As it happens in most trees, the searching algorithms (e.g. intersection, nearest neighbor search, etc.) are considerably simple. The key idea is to use the minimum bounding boxes to decide whether or not to search inside a sub tree. In this way, unnecessary nodes in the tree are not visited during a search operation. Like B-trees, R-trees are suitable for large data sets and databases, where nodes can be paged to memory when needed, and the whole tree being kept in main memory.

Though good worst-case performance is not assured in R-trees, it usually performs well with real-world data [36]. The Priority R-tree, which is a variant of the R-tree, is worst-case optimal, but due to the increased complexity it has not attracted much attention in practical applications up to now [37].

When the multidimensional data structures are applied for multiattribute index, they suffer from one problem or the other. A B+-tree multiattribute index does not allow search on some of the attributes forming the key. Even though Grid File allows search on all attributes from the multiattribute key, it restricts attribute values to contain only uniform values. Insertion and deletion can be also difficult in this kind of structure [25]. A KD-tree requires special procedure for insertion and deletion. It is also an unbalanced tree structure [30]. Bitmap Index has advantage of space and performance, but not efficient for columns whose data is frequently updated [31]. Partitioned Hashing Index cannot be used for range queries [33].

2.2 Multidimensional Data

This is any data broken down into dimensions. The dimensions are broken down into categories that may be composed of points in space. In this work, the multiattribute key to be indexed is considered as multidimensional data represented by points in multidimensional space. That is why multidimensional data structures are used for indexing data with multiple attributes in this work.

2.3 Multidimensional Access Methods

The basic aim of a database management system is to return a set of records from the database as requested by the user. Some systems try to read the database into the memory in order to find the data that is requested. Most systems try to read directly from the disk and how they achieve this is referred to as their access method. Multidimensional access method deals with reading multidimensional data off disk. The essential part of access methods is the data structure and design for indexes that point to the data structure.

Requirements of multidimensional access methods are summarized as follows [39]:

- Time and space efficiency (performance).
- Support for dynamically changing data.
- Integration with secondary and tertiary storage as it is not possible to store the complete data in main memory.
- Support for broad range of operations.

- Efficiency independent of the nature of input data and insertion sequence.
- Simplicity and scalability.
- Support for concurrent users and transactions.
- Minimum impact on integration to an existing system.

2.4 Multidimensional Data Structures

These are data structures specifically designed to store and manage multidimensional data. They are best suited for applications involving multidimensional data such as those used in areas like robotics, medical imaging, geoscience, etc. [39] and used for indexing such data. There are many variations of such specialized data structures as KD-Trees, UB-trees, BUB-trees, R-trees, etc. In this thesis, we chose R-tree to use for the proposed indexing scheme.

2.5 Z-ordering and Z-regions

Using a mathematical transformation, multidimensional data spaces can be made linear. Some of the methods used for the transformations are space filling curves, and the linear order of the multidimensional keys is the order of the points on that curve. An example of this curve is the Z-curve. The order defined by the Z-curve is called Z-order. Points in multidimensional space are transformed to their one dimensional Z-coordinates. The position of a tuple in the Z-ordering is called Z-address. An interval of the Z-curve covers a multidimensional subspace, a so called Z-region [16] (Figure 2.1).

2.5.1 Z-address Definition

Let δ be a discrete finite n-dimensional vector space, $\delta = Mn$, where $M = \{0, 1, \ldots, 2^{\tau D} - 1\}$, $|M| = 2^{\tau D}$. For a tuple $T \in \delta$ of the length n, $T = (a_1, a_2, \ldots, a_n)$ and a binary representation of the coordinate (attribute) value $a_i = a_{i,\tau D-1} a_{i,\tau D-2} \ldots a_{i,0}$, where $a_i \in D$, is the bit-length of the value a_i, $a_{i,j}$ is j-th bit value of a_i, $1 \leq i \leq n$, $0 \leq j < \tau D$, the function Z(T) (Z-address) is defined by:

$$Z(T) = \sum_{j=0}^{\tau D-1} a_{i,j} \sum_{i=1}^{n} 2^{j*n+i-1}$$

If we calculate the Z-addresses for all the points of n-dimensional space δ, we will get a Z-curve filling the entire space δ [16].

2.5.2 Z-region Definition

A Z-region [x:y] is the space covered by an interval of the Z-curve and is defined by two

Z-addresses x and y, $x \leq y$.

Figure 2.1 shows the interval [4:20] of the Z-curve (numbering starts with 0 in the left upper corner) in a two-dimensional 8x8 space and the region covered by it with its characteristic shape [40]. In Figure 2.2(b), four Z-regions in 2-dimensional space 8×8 are depicted.

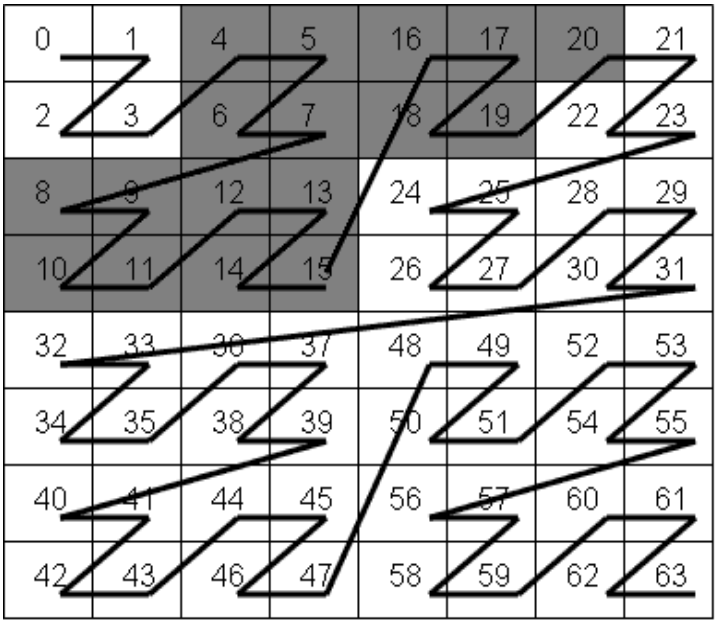
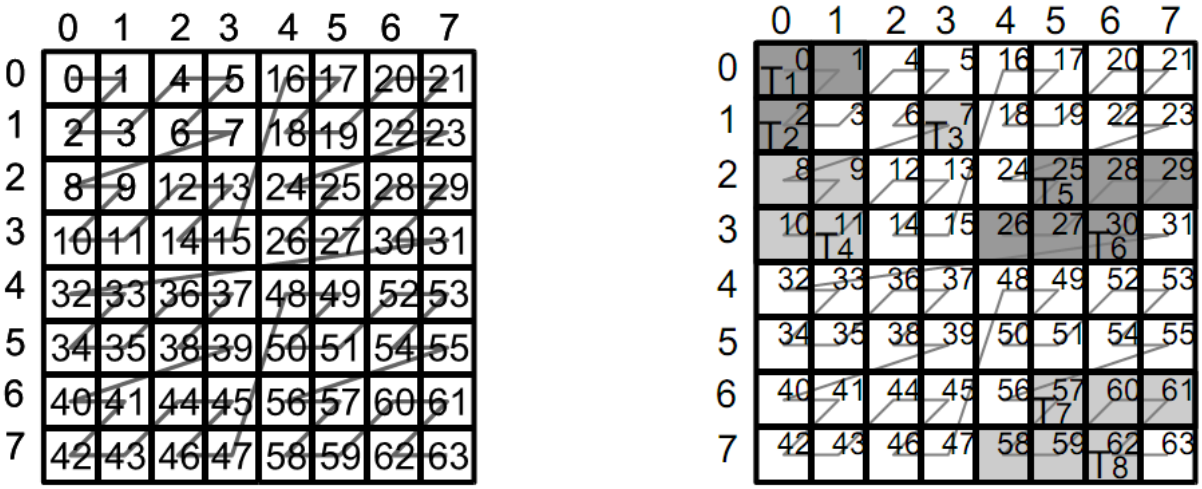


Figure 2.1 The interval of Z-curve and its corresponding regions [44].



(a) (b)

Figure 2.2 (a) The Z-curve filling the entire 2-dimensional space 8 × 8.
(b) 2-dimensional space 8 × 8 with tuples T1 – T8.

2.6 (B)UB-trees

The B-tree is a dynamic high performance tree data structure to organize and manage collections of data which are stored on random access devices like disks. The UB-tree is a multidimensional generalization of the B-tree [40]. Traditional B-trees were designed for one dimensional, linearly ordered key spaces. In this case, B-trees perform well for one dimensional point queries and interval queries. But, there are many applications using multidimensional data such as geographic maps. Range queries in such situations resemble multidimensional rectangles, and the multidimensional points in those rectangles must also be read.

(B)UB-trees exploit the Z-ordering. When a multidimensional data space is made linear using Z-ordering and is then represented in ordinary B-trees, the resulting data structure is called UB-tree (Universal B-tree). Points in a node of a UB-tree correspond to an interval on the Z-curve, hence to a Z-region.

UB-tree and BUB-tree establish Z-regions for clustering spatial neighbors onto disk pages. Tuples from regions are placed into a single B-tree page. The regions are mapped onto disk pages. Thus, it ensures minimal disk accesses upon getting spatial neighbors.

Bounding Universal B-tree (BUB-tree) is also a paged and balanced multidimensional index structure. It uses the Z-ordering and B+-tree. Each node in the BUB-tree contains hierarchy of the Z-region stored in a single disk page. The BUB-tree hierarchy is depicted in Figure 2.3. The leafs contain indexed tuples while the inner nodes contain Z-regions. In the case of UB-tree, the Z-regions define an ordered disjunctive partitioning of the entire n-dimensional space. The BUB-tree does not index the ”dead space” (contiguous empty space) due to the shapes of Z-regions evolving during the tuples insertion. This is an improvement over the UB-tree which indexes the entire space, and it makes the range query processing more efficient in the BUB-tree [41].

For example, the tuples in Figure 2.2(b) define the BUB-tree Z-regions partitioning [0:2], [7:11], [25:30] and [57:62] with the node capacity of two. The empty interval (31:56) between Z-regions [25:30] and [57:62] is not indexed by the BUB-tree as shown in Figure 2.3.

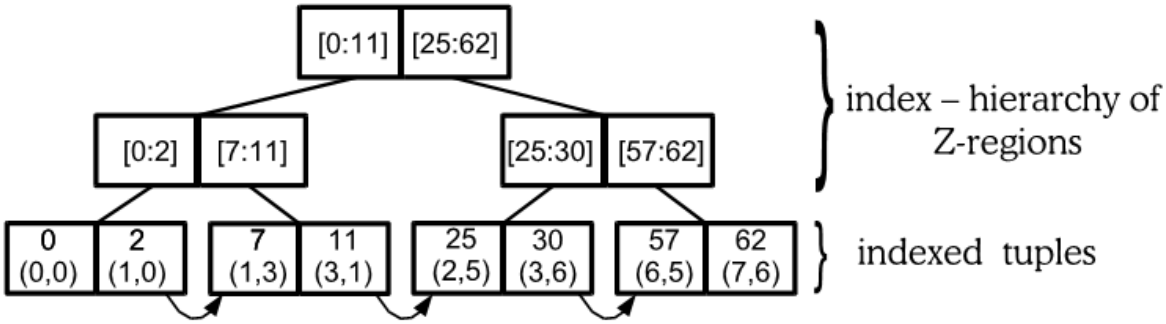


Figure 2.3 BUB-tree indexing tuples presented in Figure 2.2 [16].

2.7 R-trees

R-tree is a straightforward extension of B-trees in n-dimensional space. This data structure is balanced in terms of depth and made up of intermediate and leaf nodes. The data objects (tuples in the case of this research) are stored in leaf nodes. Intermediate nodes are formed by grouping lower level nodes, hence its associated rectangles also called minimum bounding boxes (MBB) enclose associated rectangles for lower level nodes [42].

R-tree resembles a hierarchy of two n-dimensional points formed from minimum bounding boxes that are grouped together. If N is interior node of the R-tree, it contains pairs (R_i, P_i) where P_i represents a pointer to the child of node N. If R is the MBB of N, then box R_i encloses children N_i of N. If N is a leaf node, the P_i represents pointer to a disk block containing data object (e.g. tuples). In Figure 2.4, a general structure of R-tree for indexing point data from n-dimensional space is shown.

Properties of the R-tree are as follows [42]:

- It is height balanced.
- The root node has at least two children unless it is a leaf.
- Every non-leaf node has between m and M children unless it is the root node.
- In each non-leaf node, the entries are in the form of <MBB, Pc> where Pc is the pointer to the child node.
- Each leaf node contain between m and M entries if it is not the root node.
- In each leaf node, the entries are in the form of <T, Pd> where T is the tuple identifying the record and Pd is a pointer to the record in disk.

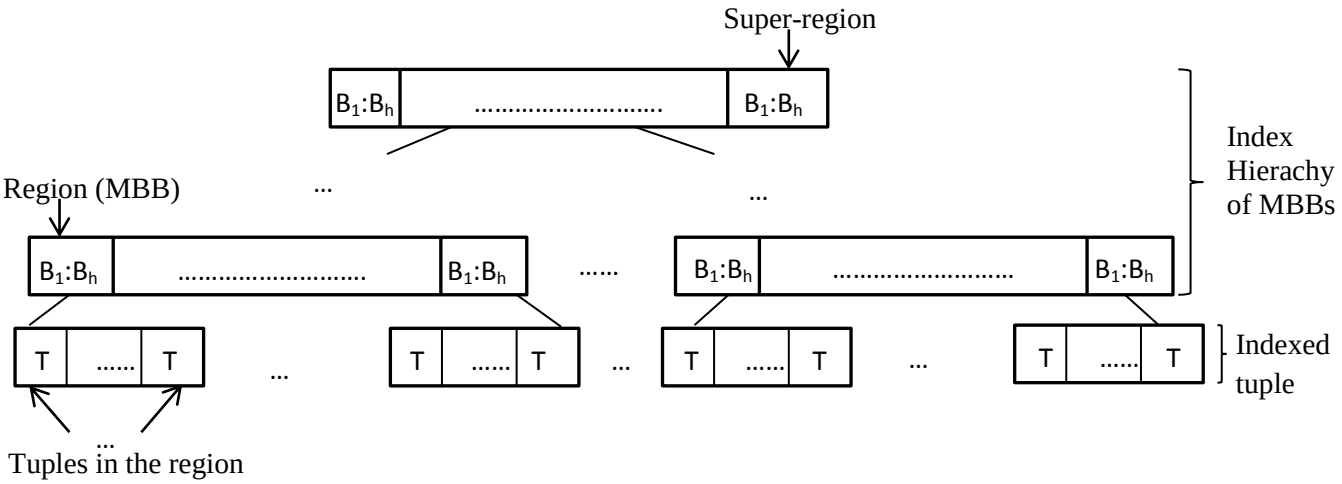


Figure 2.4 General structure of R-tree.

2.8 Minimum Bounding Box (MBB)

A minimum bounding box (MBB) also known as minimum bounding rectangle (MBR) is the least surrounding rectangle of points associated with an object. It is denoted by two (x, y) coordinates in space where x defines lower left and y defines upper right corners of the rectangle. The MBB is commonly used as an approximation item in spatial indexes like R-trees and its variations and in areas of object intersection.

MBB was first seen in Guttman's work [42] as an approximation tool to perform spatial indexing. His idea was to have each index record in leaf node of R-tree represented by the smallest enclosing rectangle that contains n-dimensional data points in n-dimensional space and a pointer to a file containing the actual object's representation. Since then, MBB becomes one of the most widely used approximation technique for spatial indexes.

It is also one of the most common methods used for approximation in spatial access methods. Its simple representation is the main reason behind its popularity. It needs just 2 points to denote the minimum bounding box, while the data object it is representing may be of many orders of magnitude which are more complicated. The advantage of minimum bounding box has resulted in the development of many data structures among which are the R-trees and its variations.

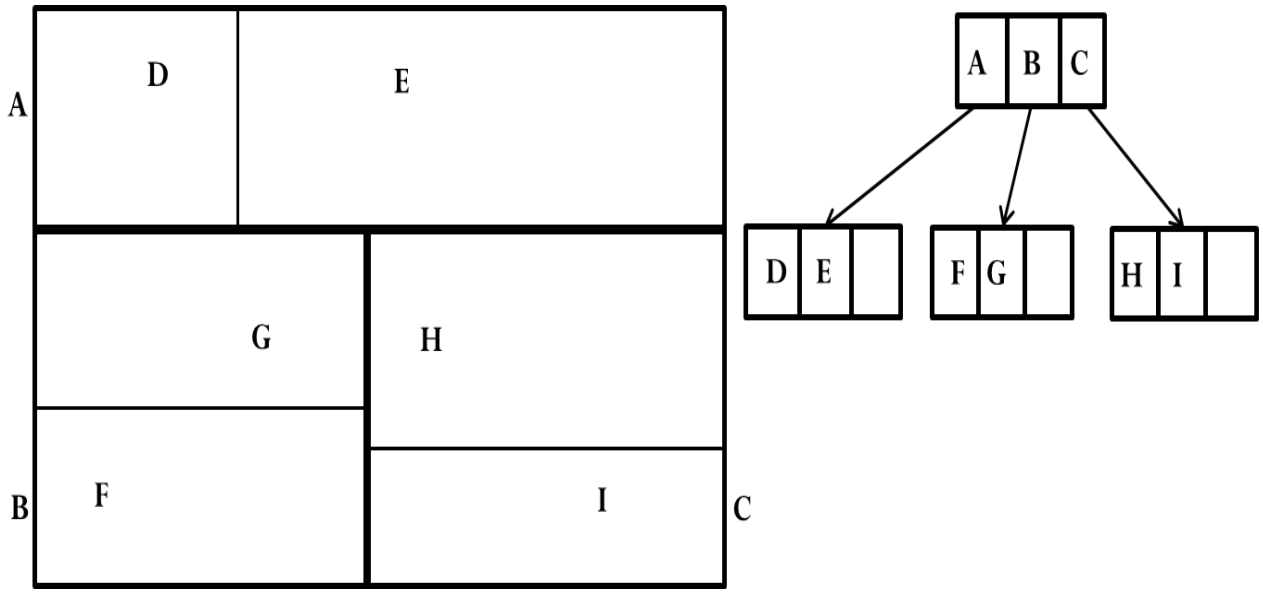


Figure 2.5 Example of MBBs and R-tree created from them.

R-tree is concerned more with reducing the overall area of the minimum bounding boxes of the parent nodes, it is not concerned with reducing the overlap of parent nodes. To insert a new node into R-tree, the tree is traversed to find the leaf node requiring the least amount of expansion to accommodate new node. This action provides us with a tree of minimum bounding box nested together.

A search algorithm is provided to perform a search on the tree, with a box representing the range to be searched. Boxes that are completely outside the query rectangle are eliminated to shorten the result space while those that are completely inside the query rectangle are automatically submitted. Boxes that intersect the query box, i.e. those that are neither inside nor outside the query rectangle, are studied more using the actual object representation and those that are discovered to intersect the query box are later submitted. Several spatial access techniques have been created on this structure put down by Guttman, which increase the popularity of the minimum bounding box as approximation tool.

2.9 Node Splitting Techniques

There are various techniques for redistributing tuples of a node into two nodes. In the traditional R-tree, Guttman proposed two such techniques: Quadratic Split and Linear Split [27]. The goal of the split is how to partition the M entries already contained in a node and one new entry to be inserted into two nodes such that the total area and the overlapping of the two resulting nodes is minimized. In quadratic split which tries to minimize the area of the two nodes, the algorithm searches the pair of rectangles that is the worst combination to have

in the same node, and puts them into the two new nodes. It then searches the entry that would bring the smallest area increase into one of two nodes and assigns it to this node until all objects are assigned satisfying the minimum fill [43].

2.10 R-tree Index Structure

R-tree index structure contains pairs of the form (N, P) representing regular nodes where N corresponds to MBB and P represents pointers to a child node. In the case of leaf nodes, the pointer P points to the tuples [11].

Important Parameters for R-tree index are:

- M is the maximum number of entries in one node.
- Parameter $m \leq M/2$ specifies the minimum number of entries in a node.

2.11 R-tree Operations

Below is the description of operations that can be performed on R-trees.

2.11.1 Search

A search operation in R-tree resembles B-tree search. It is simple and straightforward and it involves traversing the whole tree, beginning from the root node.

The complexity of the R-tree search algorithm is O(n) under the worst case when the query box overlaps all MBBs in the tree (i.e. all MBBs overlapping the search area), where n is the number of nodes overlapping the search area. The best case occurs when there is at most one overlap at each tree level. Here, the complexity becomes $O(\log_M n)$, where M is the number of entries in a node and n is the number of nodes overlapping search area.

Example 2.1: In the Figure 2.6(a) below, the long narrow rectangle shows the search rectangle (query box). The algorithm is looking for qualified records in the query box's area. In Figure 2.6(b), the paths chosen by the algorithm can be seen [44]. The long narrow MBB representing the QB intersects the root entries R1 and R2, so the algorithm checks these entries. In R1 there is just R4 which intersects the query MBB. Its entries are also checked. The algorithm arrives at the leaf node level. The entries of the leaf node are checked for qualifying records. R11 is the only one and so the first search result. In R2, there are two rectangles that overlap with the query MBB: R5 and R6. Both of them are checked and the algorithm recognizes that in the leaf node level the entries R13, R15 and R16 overlap with the search rectangle. Therefore, the search result is R11, R13, R15 and R16.

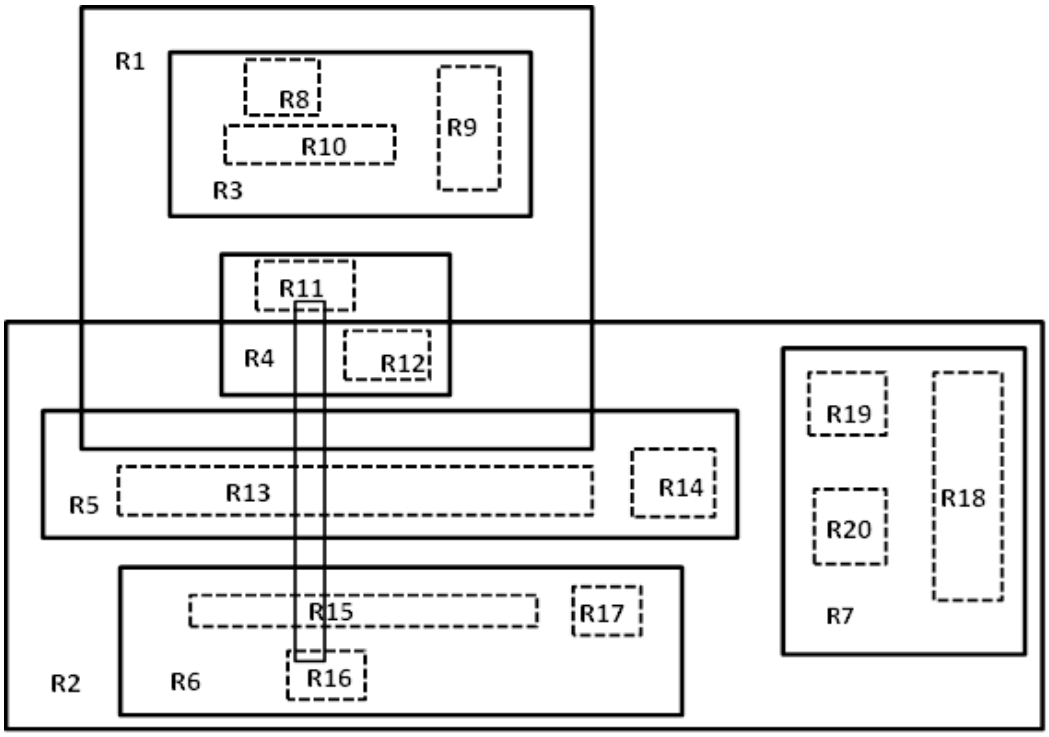


Figure 2.6 (a) MBB search example.

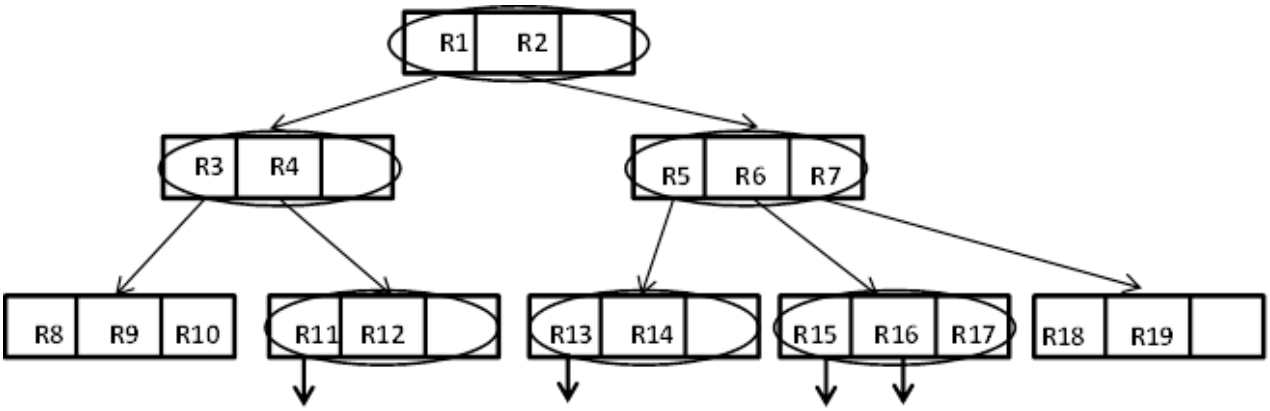


Figure 2.6 (b) R-tree search example

2.11.2 Insertion

This operation is similar to insertion in B-Trees. It involves adding new data into the leaves. Nodes that overflow are split and the split causes adjustment to the nodes which is propagated up to the root. It basically consists of 3 stages [11]:

1. Choose Leaf: Insert operation initially traverses the tree from root until leaf node is reached where data is to be placed. It selects the leaf node to place the new data entry.
2. Insert: Insert the new data object into the leaf node.
3. Adjust Tree: If the leaf node has an overflow after insert operation, the tree should be adjusted by splitting the affected node into two and the split propagated upward.

The complexity of the R-tree insert algorithm is O(n) under the worst case when the query box overlaps all MBBs in the tree, where n is the number of nodes overlapping the search area. It is 2nh in all other cases, where n and h are the number of entries in each node and the tree height, respectively.

Example 2.2: Supposing a rectangle R21 is to be inserted into the tree shown in Figure 2.8. To find the best position for the new rectangle, the algorithm starts with *Choose Leaf*. Figure 2.8 also shows the path of *Choose Leaf*. The first step is clear because R21 is in R1. Next, R3 is chosen because this rectangle needs fewer enlargements than R4. At the last step, the algorithm finds the leaf node, however, all entries are full. Thus, it comes to a node split which is also shown in Figure 2.9 and 2.10 below. Node splitting tries to minimize rectangles as much as possible. That is the reason why the algorithm puts R21 and R9 in rectangle R3. R8 and R10 are put in the new parent rectangle $R3^+$. $R3^+$ is conveyed to *Adjust Tree* where it is propagated upward. Since there is enough room to include $R3^+$, it's not necessary to split this node again. R3 must be adjusted as well because it only points to R9 and to the new rectangle R21. At last, root node R1 is also adjusted because it includes a new entry $R3^+$. So the structure of the tree is saved. The insertion is now finished and Figure 3.8 shows the newly included rectangle.

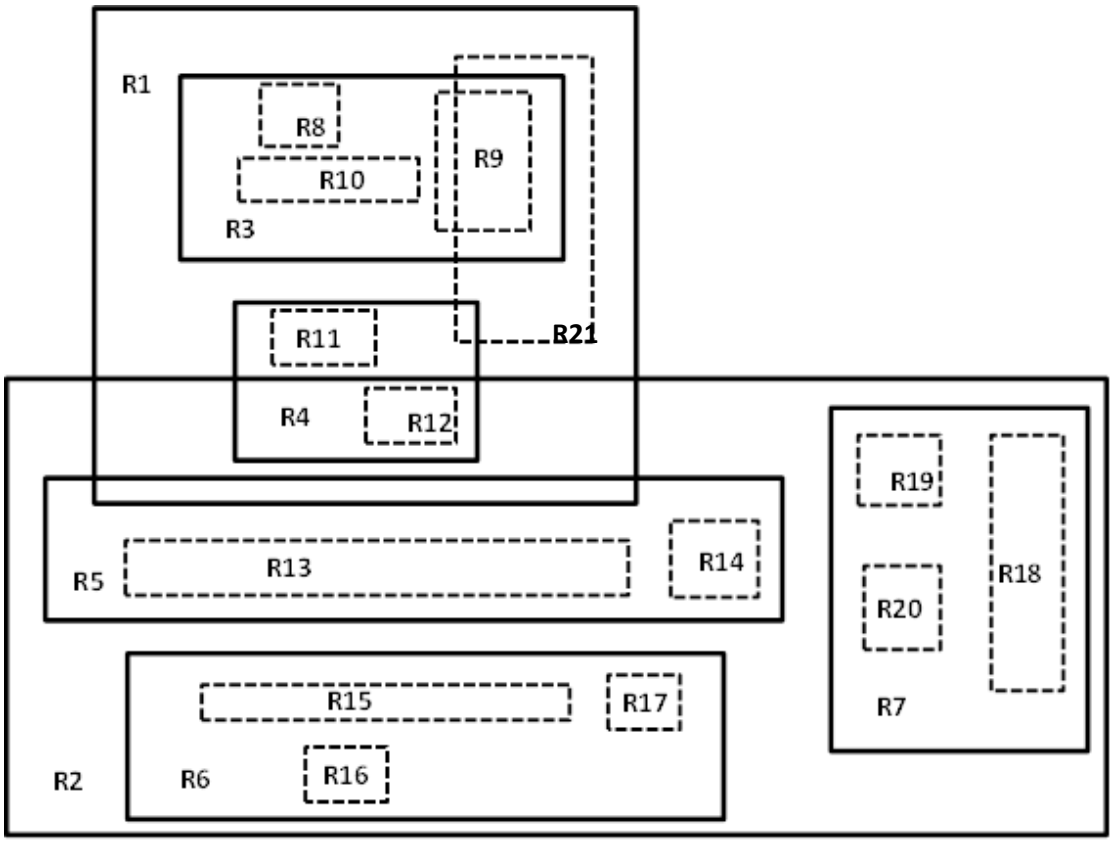


Figure 2.7 Inserting a new rectangle.

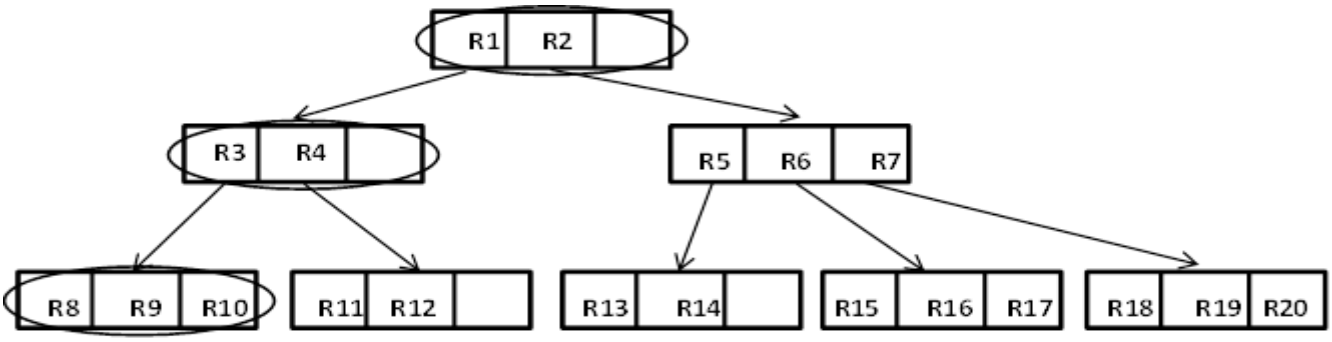


Figure 2.8 R-tree insert example.

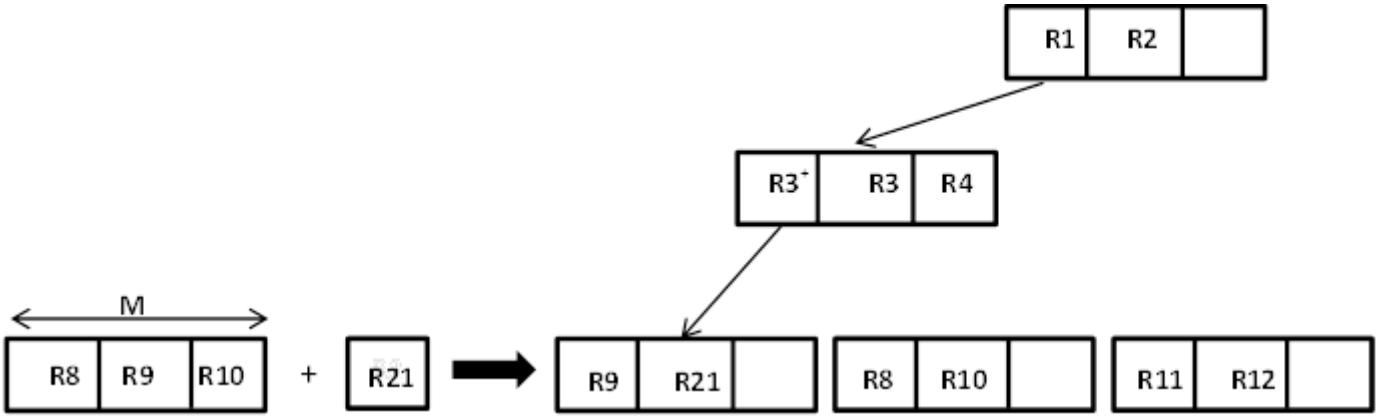


Figure 2.9 Splitting of a node.

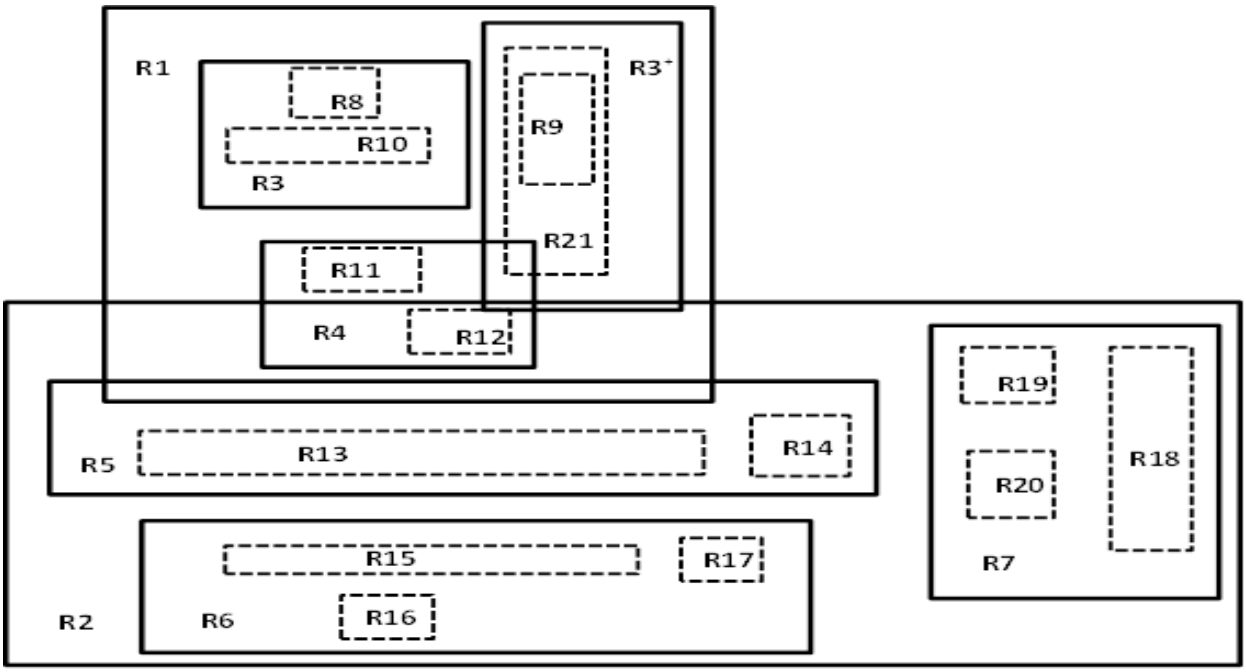


Figure 2.10 MBBs after insertion.

2.11.3 Deletion

The Deletion operation is unlike deletion in B-Tree. It is complicated because it needs to treat the underflow. It consists of the following 3 basic steps [11]:

1. Choose Leaf: Delete operation first traverses the tree to find the leaf node containing the data to be deleted.
2. Delete: Delete the entry from node.
3. Condense Tree: If a node has its entry removed and there is an underflow, the few remaining entries are reallocated while recursively checking its parent until the root is reached. All the MBBs are updated to remove all underflow nodes. All entries deleted from the removed nodes are reinserted according to the INSERT algorithm.

The complexity of the R-tree deletion algorithm is O(n) under the worst case when the query box overlap all MBBs in the tree, where n is the number of nodes overlapping the search area. It is 2nh in all other cases, where n is the number of entries in each node and h is the tree height.

Example 2.3: In the following example we describe deletion of R11 from a tree with m = 2 and M = 3. At first, the delete algorithm starts *Choose Leaf* to get the position of R11. R11 is returned as result of the query. After that R11 is removed from the tree. Now, *Condense Tree* is started. Figure 2.11 shows the first procedures of this algorithm. With the new value m = 1, R4 has an underflow. It is eliminated from the tree but the last entry R12 of R4 is saved in list Q. Through the deletion of R4, R1 has an underflow as well and thus R1 is eliminated. Its entry R3 is saved in the list. Next, all entries of the list Q (highlighted in red) are reinserted in the tree (Figure 2.12).

Firstly, the node R3 has to be placed in the same level again where it was before having been set in Q. After that leaf node R11 is reinserted in R5 supposing R5 is the nearest rectangle which has to enlarge least. *Condense Tree* is finished. The root node has only one child and thus the child is the new root. The following Figure 2.12 shows the new structure of the tree after deletion.

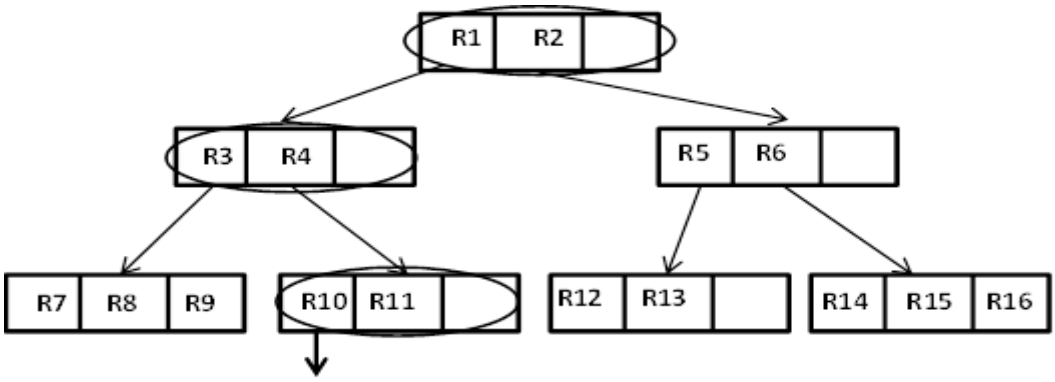


Figure 2.11 R-tree delete example

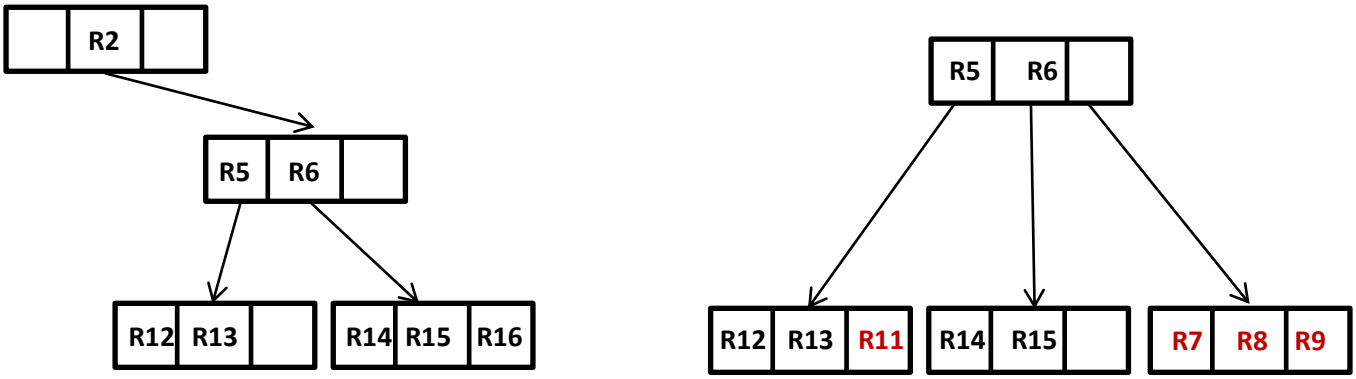


Figure 2.12 R-tree after deletion of R10 (left) and reinsertion of R11, R3, R7, R8, R9 (right)

2.11.4 Split Node

This operation is necessary to save the tree structure whenever there is an underflow as a result of deletion or overflow as a result of insertion. Insertion uses this method to divide entries into two nodes when adding a new entry into a full node [11]. The two new nodes obtained as a result of split should be distinct to allow them to be checked on subsequent searches. The ultimate goal of split operation is to minimize the resulting node's MBBs. Figure 2.11 shows an example of good and bad splits.

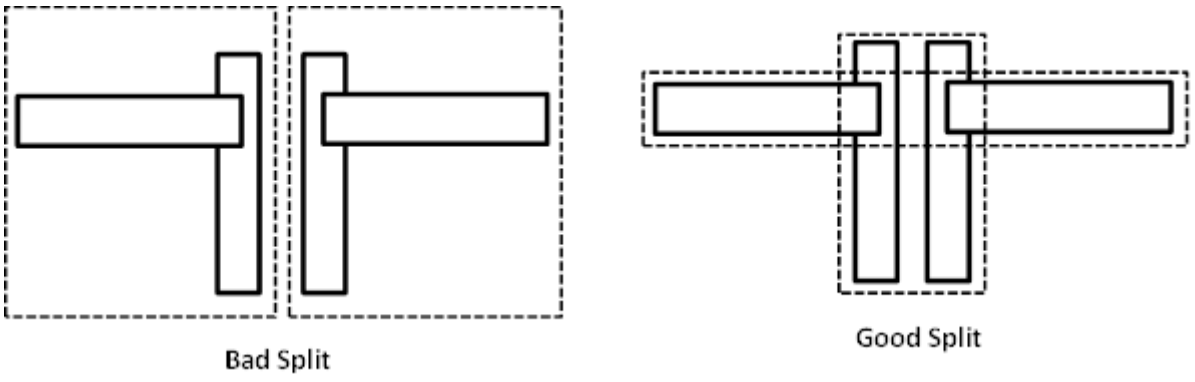


Figure 2.13 Types of split

2.12 Signature R-trees

Signature R-tree is the combination of R-tree with n-dimensional signatures for effective filtration of irrelevant tree nodes. Figure 2.9 shows the general structure of Signature R-tree. Leaf nodes contain indexed tuples that are clustered into regions, also called minimum bounding boxes (MBBs). The MBBs can be arranged to form another MBBs again called super regions.

The description of regions and super regions (formed in the n-dimensional space) are stored in the tree nodes. An n-dimensional signature is assigned to each region (MBB). Any node containing a super region also holds an n-dimensional signature, superimposed with signatures of immediate children of the node. As a result, Signature R-tree has two hierarchies: hierarchy of MBBs and hierarchy of n-dimensional signatures [16].

The operations of R-tree data structures are maintained and n-dimensional signatures are applied for effective filtration of irrelevant nodes during narrow range query processing. With the help of signatures, intersection algorithm determines which among the nodes are relevant or not to the user's query. Enlargement of this data structure is small, because the n-dimensional signatures are only inserted into inner tree nodes.

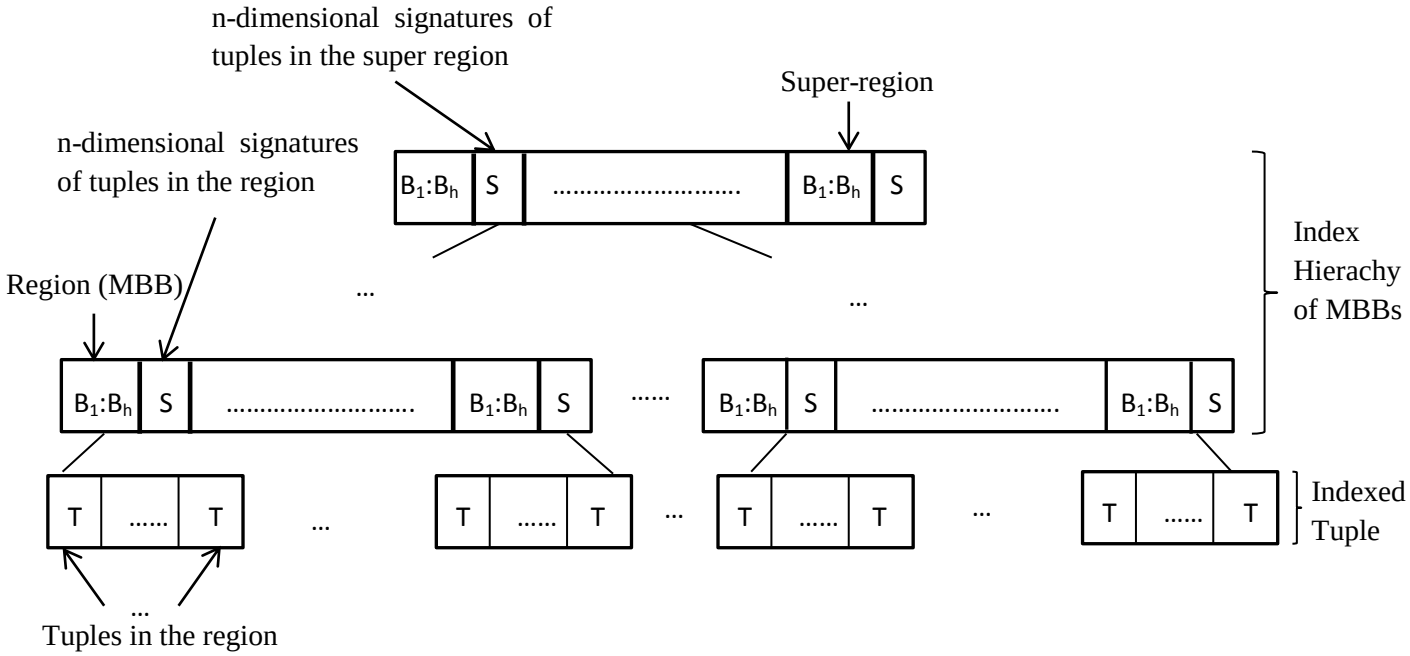


Figure 2.14 Structure of the Signature R-tree

2.13 Query Creation

Regular expression queries are created by a combination of numerous range queries. The maximal number of attributes in the multiattribute key indexing is used during query construction [16].

Let us now demonstrate how to create 3 forms of regular expression queries. An additional regular expression query is based on these 3 forms of regular expression queries. Let k be the length of string, n be the dimension of the attribute space δ, and max M_i be the maximal value of the domain M_i.

A right extension query (expression <string>*) is achieved by a single range query:

$$QB = (c_1 , c_2 , \ldots , c_k , 0, \ldots , 0) : (c_1 , c_2 , \ldots , c_k , \max M_{k+1} , \ldots , \max M_n).$$

A left extension query (expression *<string>) is achieved by a single range query:

$$QB = (0, \ldots , 0, c_1 , c_2 , \ldots , c_k) : (M_{k+1}, \ldots , \max M_n, c_1 , c_2 , \ldots , c_k).$$

A left-right extension query (expression *<string>*) is processed by a range query of the form:

$$QB = (0,\ldots 0 ,c_1 , c_2 , . ., c_k , 0, . ., 0) : (M_{k+1}, .. , \max M_n, c1 , c2 ,. . , ck , \max M_{k+1},\ldots,\max M_n).$$

In [12], range query processing is described. The result of these queries is made up of all relevant tuples. Tuples retrieved from multiattribute key index are returned to the user as a query result.

2.14 Query Types

The most common application of data structures is the query, which enables the user to specify the desired data, leaving the database management system to perform the physical operations necessary to produce the desired result. Different data structures support different types of queries. In the case of Signature R-trees data structures, it supports the following types of queries [16]:

- Point
- Range
- K-Nearest-Neighbor

2.14.1 Point Query

This is one of the query types supported by R-trees. It involves searching the tree's MBBs intersecting the point to find a tuple corresponding to this point.
Given a point p ϵ δ, find a tuple corresponding to P.

2.14.2 Range Query

R-trees support this kind of query that is sometimes called window or box query. It is processed from the given query window. It involves searching the tree to find MBBs intersecting the QB. The result of this query is all tuples that falls within the query box.

Let δ be an n-dimensional discrete space, $\delta = M_n$, $D = \{0, 1, \ldots, 2^{\tau D} - 1\}$, and points (tuples) $T_1, T_2, \ldots, T_m \in \delta$. $T_i = (t_1, t_2. \ldots t_n)$, τD is the chosen length of a binary representation of a number t_i from domain M. The range query RQ is defined by a query box QB which is determined by two points $QL = (ql_1, \ldots, ql_n)$ and $QH = (qh_1, \ldots, qh_n)$, QL and QH $\in \delta$, ql_i and $qh_i \in M$, where $\forall i \in \{1, \ldots, n\} : ql_i \leq qh_i$ [16].
Given a query box QB = QL x QH where $QL = (ql_1, \ldots, ql_n)$ and $QH = (qh_1, \ldots, qh_n)$, find all tuples intersecting QB.

2.14.3 KNN Query

R-trees support this type of query also. It involves finding k-most similar objects/points to the objects/points obtained from the query definition.

Given a tuple T, find all tuples having a minimum distance from T. Distance function like Euclidean distance is used to determine the nearness of points [42].

2.14.4 Narrow Range Query

Let δ be an n-dimensional discrete space, $\delta = M_n$. The query hyper-box is defined by two points $QL = (ql_1, \ldots, ql_n)$ and $QH = (qh_1, \ldots, qh_n)$, where $\forall i : ql_i \leq qh_i$. Let α and β be constants: $\min(M) \leq \alpha \vee \beta \leq \max(M)$. The range query is called the narrow one if:

1. $\forall i : qh_i - ql_i \leq \alpha \vee qh_i - ql_i \geq \beta$.
2. Let $n\alpha$ and $n\beta$ be the number of dimensions for which formulas $qh_i - ql_i \leq \alpha$ and $qh_i - ql_i \geq \beta$, respectively, hold. Furthermore, in the case of the narrow range query it holds $1 < n\alpha < n \wedge 1 < n\beta < n$ [16].

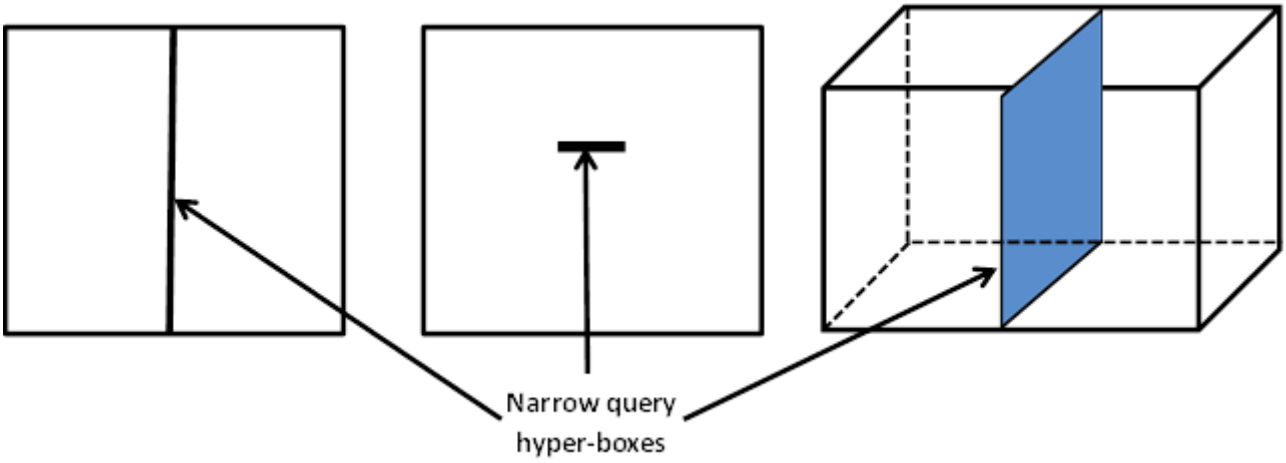


Figure 2.15 Examples of the narrow range queries in two and three dimensional spaces [12].

2.14.5 Multiattribute Keys as n-dimensional Points

Supposing we have a domain, M = {0, 1, 2…….$2^{\tau d}$-1} and a discrete finite n-dimensional vector space, $\delta = M_n$. Let C represents ASCII code character and tuple, T = c_1, c_2, c_3 ……c_n, of length n, where $c_i \in C$ with $1 \leq i \leq n$. Now, the n-dimensional tuple representing the multiattribute key is defined as T_s = (code(c_1),code(c_2), code(c_3),…… code(c_n)), $T_s \in \delta$, code(c_i) ∈ M, where code : C → M is a function that encodes character c_i into binary of length τd [16].

When the tuple length is not complete, blank values (zero in this case) are used to complete it. The tuples, as a set of multidimensional points, are indexed using a spatial access method. R-tree supports range query algorithms that are applicable for regular expression queries. This proposed approach, therefore, allows such queries.

Supposing we have a regular expression query p*, its corresponding range query in 2 dimensional space will be (code(p),0) × (code(p), max M) and in 3-dimensional space will be (code(p),0,0) × (code(p), max M, max M). The execution of this query will retrieve all tuples beginning with p. These queries marked as 1 and 2 are shown in the Figures 2.14 and 2.15 below. The first query will retrieve tuple (P, 2) while the second query will retrieve tuples (P, A, 2) and (P, K, 8).

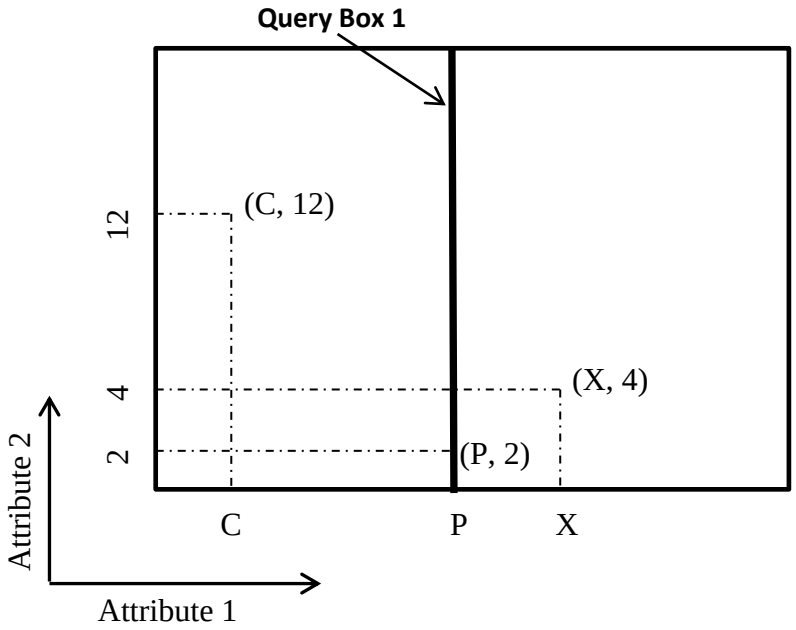


Figure 2.16 Two-attribute tuples modeled in 2D space and a query box of (P,*).

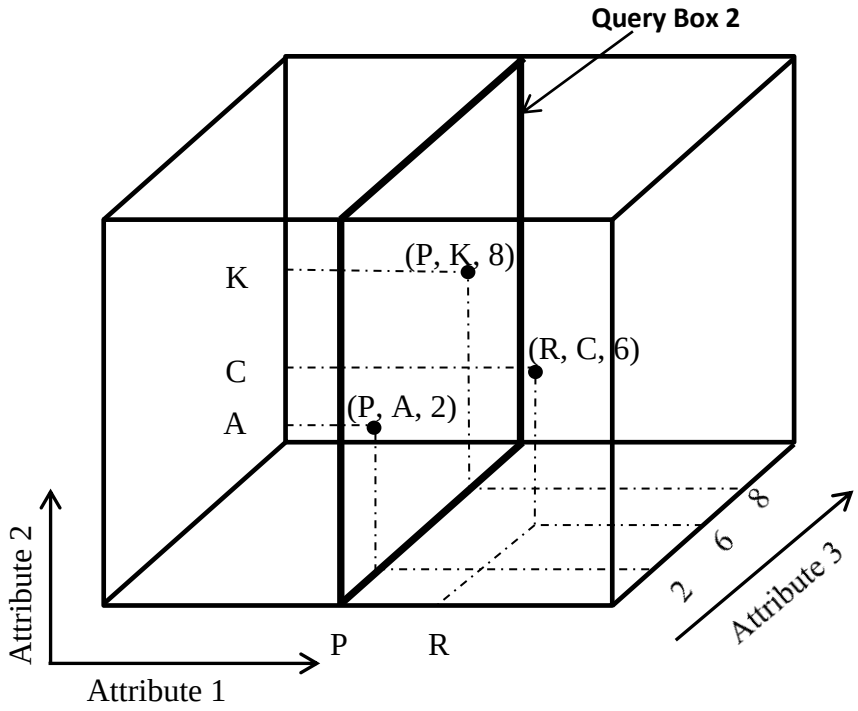


Figure 2.17 Three-attribute tuples modeled in 3D space and a query box of (P*).

2.15 Narrow Range Query Processing in Multidimensional Data Structures

Usually, multidimensional data structures like R-trees divide n-dimensional space into sub-spaces called regions. The R-trees used for the sake of this research cluster tuples into MBBs accordingly. The index is formed from regions (also called super regions) with each region

containing tuples stored in one leaf node. The inner nodes define the outer super regions, also known as MBB in R-trees. A range query algorithm filters out the irrelevant tree nodes (regions) by searching only the leaf nodes intersecting the query box [12].

Example 2.3: This example illustrates the reason for inefficiency of the narrow range query in R-tree. Supposing we have a two dimensional space that contains 3 points: (4, 7), (4, 4) and (5, 7). These points define MBB of (3, 3): (5, 7) as shown in Figure 2.16 below. A range query defined by (1, 5): (4, 5) query box would lead to searching of the above MBB. Though the region is relevant to the query box, because it has intersected the query box, it contains no point of the query box.

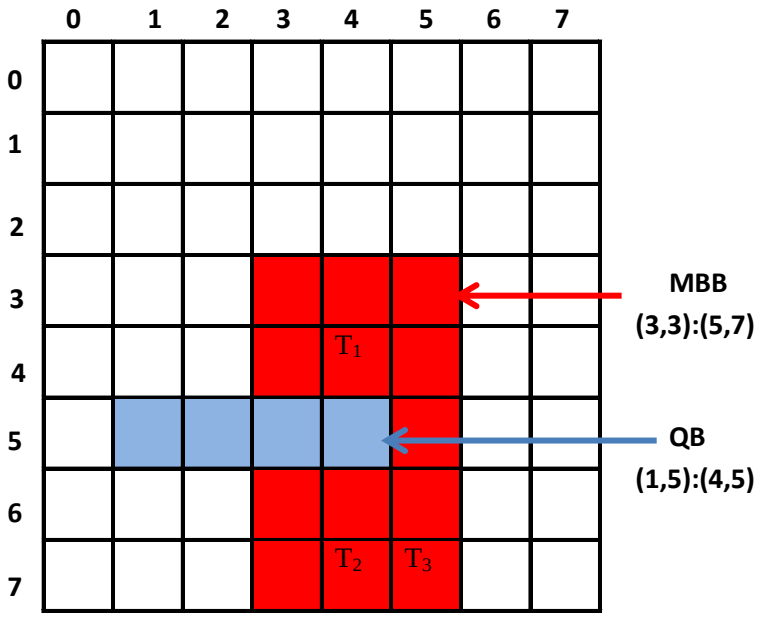


Figure 2.18 Points T_1, T_2, and T_3 in MBB and the narrow range query.

2.15.1 Intersect, Relevant Regions and Relevance Ratio

Let RQ be the range query defined by the box QB. Regions that intersect a query box during the processing of a range query are called intersect regions and regions that contain at least one point of the query box are called relevant regions. We denote their number by NIR and NRR, respectively [16]. The relevance ratio is CR= $\frac{\text{NRR}}{\text{NIR}}$

Example 2.4: This example illustrates the use of n-dimensional signatures for filtration of irrelevant nodes. Let us express how to create and apply a simple n-dimensional signature for better filtration of irrelevant tree nodes in the R-tree, using points from the previous example. The first coordinate of the n-dimensional signature contains superimposed first coordinates of the points: 4 (100) OR 4 (100) OR 5 (101). The second coordinate equals: 7 (110) OR 4 (100) OR 7 (100). In this way, the n-dimensional signature (101,110) is created. Since the second coordinates of both query box points contain the same values (the number 5) then all relevant points contain value 5 in their second coordinate. Consequently, the n-dimensional signature of the query hyper box is (101,101). The region (MBB (3, 3):(5, 7)) is recognized as irrelevant by the signature operation (101,110) AND (101,101). Since (101) AND (110) ≠ (101) then the region is irrelevant, in spite of the query box intersecting the region which is searched during the narrow range query processing in the classical R-tree.

2.15.2 Range Query processing with the n-dimensional Signature

Let us take the range query defined by two points of an n-dimensional space QL = (ql_1, . . . , ql_n) and QH = (qh_1, . . . , qh_n).

Then, let us create the n-dimensional signature of the query box S^n_{QB} = (S_{QB1}, . . . ,S_{QBn}):
If $ql_i = qh_i$, and $qh_i - ql_i \geq \beta$, then $S_{QBi} = F(ql_i) = F(qh_i)$ and $S_{QBi} = F(ql_i)$ OR $F(qh_i)$, respectively.
If $ql_i \neq qh_i$ and $qh_i - ql_i \leq \alpha$, then $S_{QBi} = 2^{lS} - 1$ (the number with only true bits).
Let us take the n-dimensional signature S^n = (S_1 . . . S_n) of points T^1, T^2… T^m. The points generating the n-dimensional signature can belong to the query box if all partial signatures S_i and S_{QBi} , $1 \leq i \leq n$, are matched by the AND operation [16]. A partial signatures S_i and S_{QBi} are matched if:

- For $ql_i = qh_i$, and $qh_i - ql_i \geq \beta$ it holds S_i and $S_{QBi} = S_{QBi}$.
- For $ql_i \neq qh_i$ and $qh_i - ql_i \leq \alpha$, it holds γ (S_i and S_{QBi}) ≥ 1.

As a result of which we can conclude that the n-dimensional signatures S^n_i and S^n_{QB} are matched by the AND operation if all partial signatures S_i and S_{QBi}, $1 \leq i \leq n$, are matched [16]. Certainly, if S_{QBi} contains only ones, then the AND operation can be omitted. If $\gamma(S_{QBi}) \rightarrow lS$ a probability of the false drop is close to one.

CHAPTER THREE

3. MULTIATTRIBUTE KEY INDEXING

3.1 Multidimensional Approach to Multiattribute Key Indexing

The idea of multidimensional approach to multiattribute keys indexing is that attributes can be represented as points of multidimensional space. Multidimensional data structures are employed to index these points. In this research, attributes forming the multiattribute keys are represented by points in n-dimensional space, and BUB-tree and Signature R-tree are used to indexing such points. R-tree and variations have proven to work perfectly at a space with low dimensionality but their performance decrease when the number of dimensions is greater than 6. In practice, however, multiattribute keys made up of 6 attributes are very rare. In OLTP applications, multiattribute indexes usually contain 2 or 3, at most 4 attributes.

3.2 Indexing Multiattribute Key as a Multidimensional Problem

In order to index multiattribute keys, the attributes forming the multiattribute keys are first modeled as points in n-dimensional space where n represents number of attributes in the key. If, for example, the multiattribute keys are formed from three attributes, it would be represented by a 2-dimensional space. Let us demonstrate this method with examples.

Attribute 1	**Attribute 2**
P	2
X	4
C	12

Figure 3.1 Multiattribute key formed from two attributes.

Attribute 1	Attribute 2	Attribute 3
A	P	2
C	R	6
K	P	8

Figure 3.2 Multiattribute key formed from three attributes.

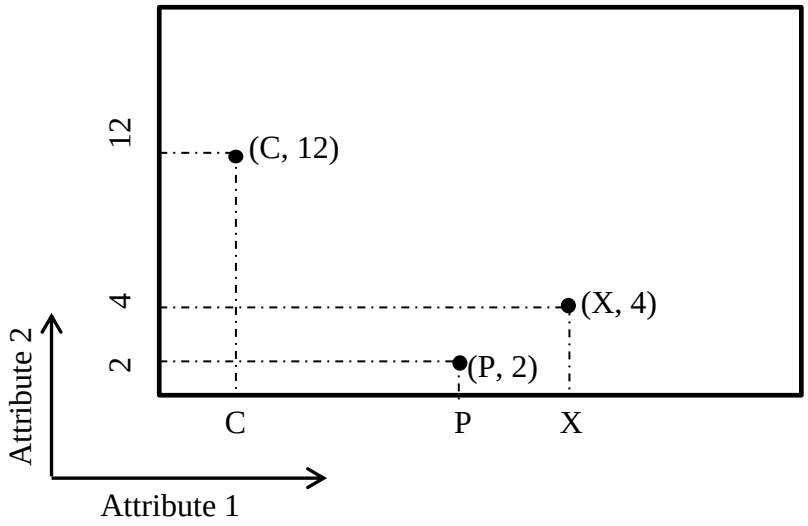


Figure 3.3 Two-attribute tuples represented in two dimensional space.

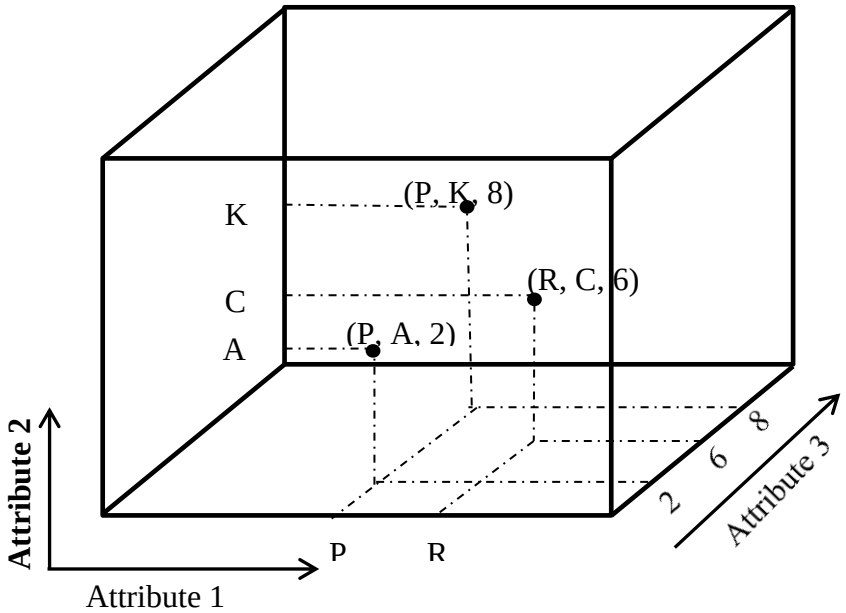


Figure 3.4 Three-attribute tuples represented in three dimensional space.

3.3 Minimum Bounding Boxes Creation

We choose bisection method to build the spatial R-tree. The method works by bisecting any box B containing d points into smaller boxes containing at most (d/2) points using a straight line.

The idea is as follows: Let B correspond to the root of the tree and let the 2 halves correspond to the children of the root after bisection. Thus, the constructed tree has 2 children corresponding to non-overlapping subsets of B. The bisection is recursively repeated on the individual children nodes taking into account the threshold which is the number of points allowable within a partition. The example shown in the following figures, demonstrate how to create MBB from points in n-dimensional space. We use a threshold of 1 (1 point per partition) in this example.

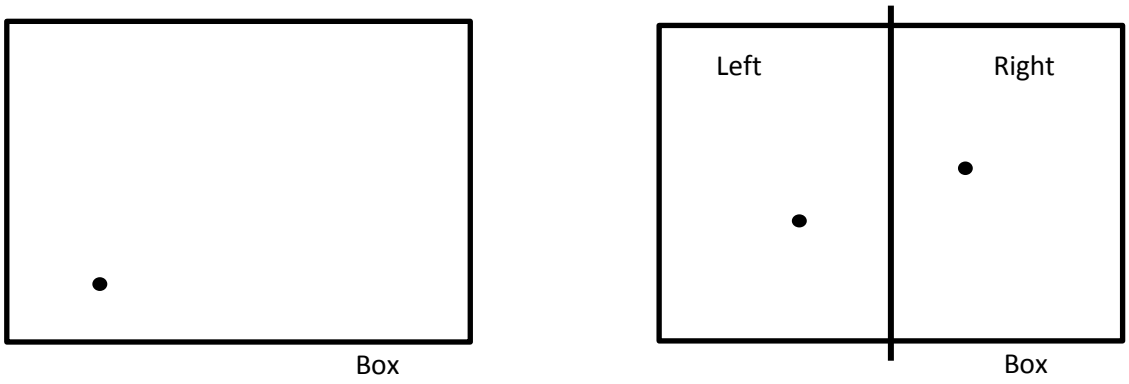


Figure 3.5 How to create MBBs.

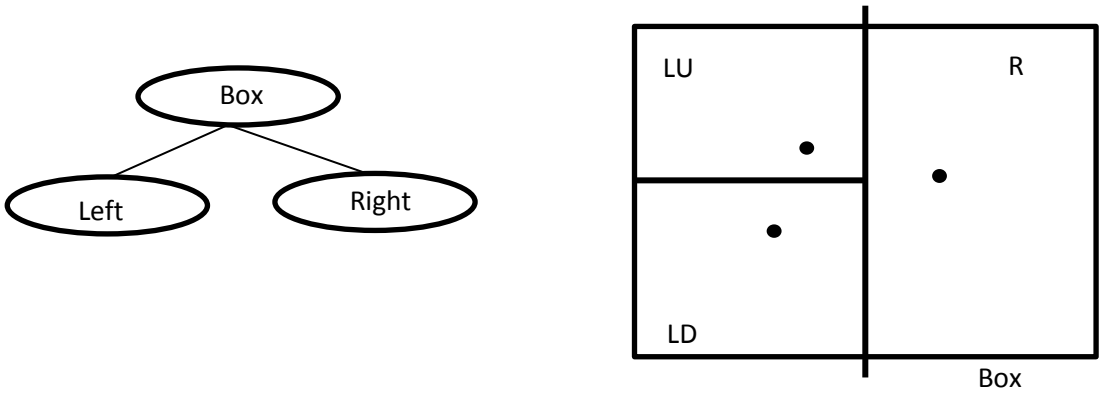


Figure 3.6 R-tree built from previous MBBs and how MBBs change after insertion.

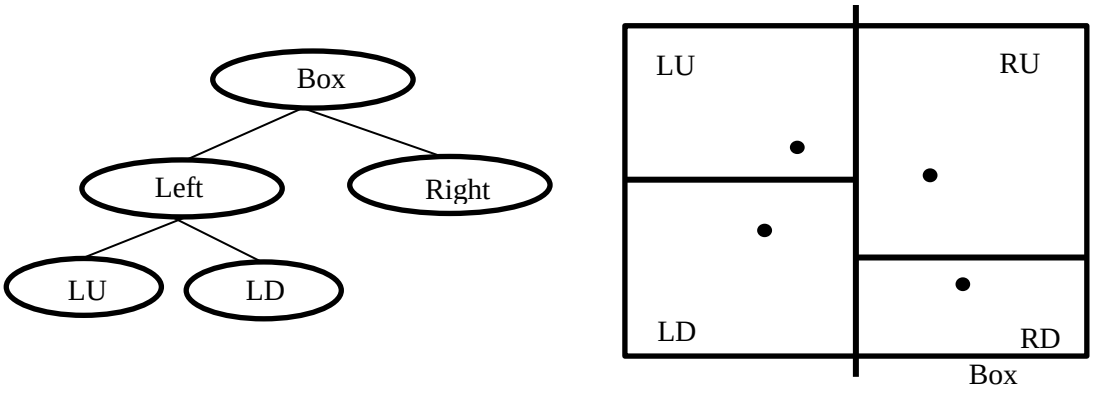


Figure 3.7 How MBBs change after insertion.

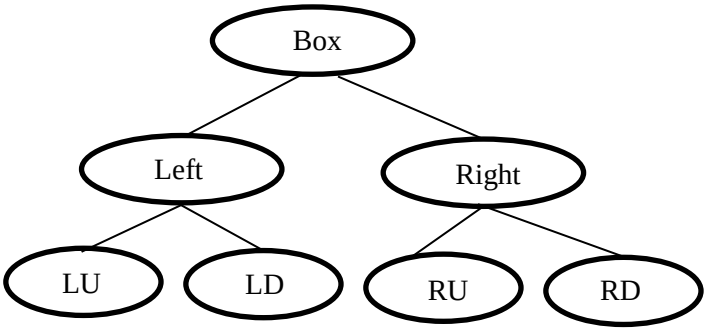


Figure 3.8 R-tree built from MBBs.

3.4 Description of the Multidimensional Approach

Using Figure 3.9 below, we will show how a multiattribute key index can be implemented on BUB-tree and how queries are processed on the attributes forming the key.

Attribute 1	**Attribute 2**
A	2
B	5
C	6
D	4
E	7
F	3
G	6
H	5

Figure 3.9 Multiattribute keys formed from two attributes.

Supposing we have the following tuples of the form T = (Y, X):

T_1= (a, 2), T_2 = (b, 5), T_3 = (c, 6), T_4 = (d, 4), T_5 = (e, 7), T_6 = (f, 3), T_7 = (g, 6), T_8 = (h, 5).

T_1 = (a, 2), S/T_1 = (000, 001)

T_2 = (b, 5) S/T_2 = (001, 100)

T_3 = (c, 6), S/T_3 = (010, 101)

T_4 = (d, 4), S/T_4 = (011, 011)

T_5 = (e, 7), S/T_5 = (100, 110)

T_6 = (f, 3), S/T_6 = (101, 010)

T_7 = (g, 6), S/T_7 = (110, 101)

T_8 = (h, 5), S/T_8 = (111, 100)

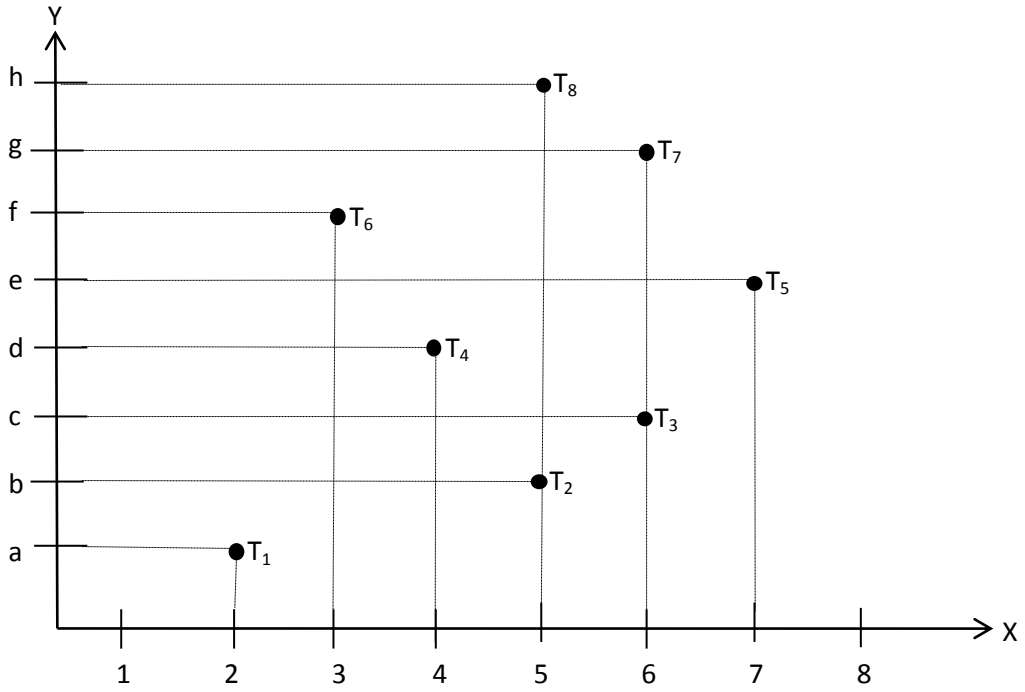


Figure 3.10 Multiattribute tuples represented in a 2D space

Let us now calculate the Z-addresses of the above tuples which serve as a clustering technique for BUB-tree.

000001

$T_1 = (000001)_2 = 1$ $T_5 = (110100)_2 = 52$

$T_2 = (010010)_2 = 18$ $T_6 = (100110)_2 = 38$

$T_3 = (011001)_2 = 25$ $T_7 = (111001)_2 = 57$

$T_4 = (001111)_2 = 15$ $T_8 = (111010)_2 = 58$

These tuples define the BUB-tree Z-regions partitioning [1:15], [18:25],[38:52] and [57:58] for a BUB-tree with the node capacity of 2.

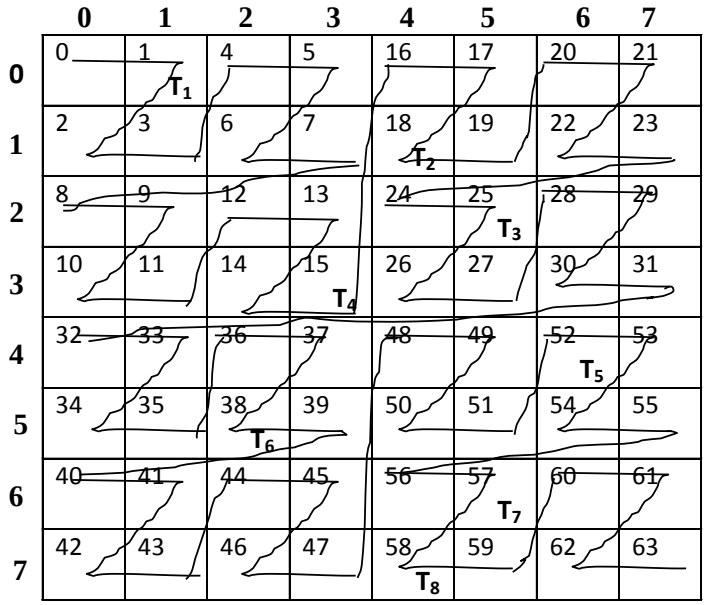


Figure 3.11 2D 8×8 space with tuples T1–T8. These tuples define the BUB-tree Z-regions partitioning.

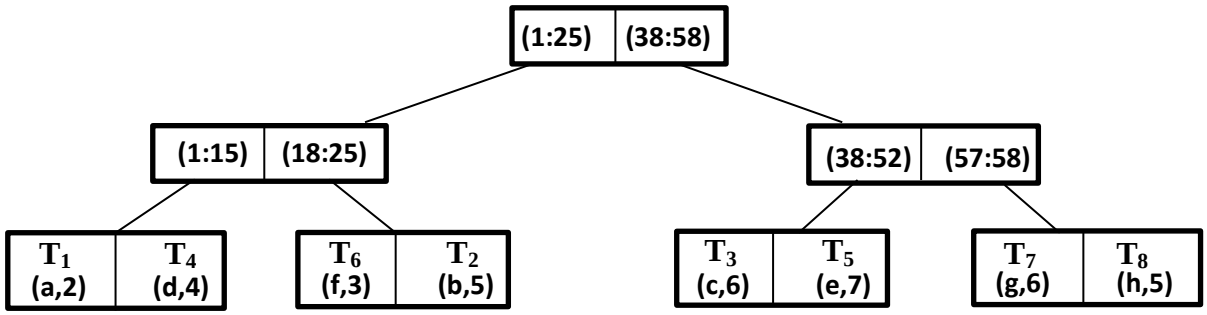


Figure 3.12 BUB-tree created from Z-regions.

Sample queries:

Q.1 Select from T where Y = '4'

The query is defined by the query box QB = (a,4) X (h,4) Z (a,4) = 5 and Z (h,4) = 47. Select x and y such that: $x \leq Z(T) \leq y$. This corresponds to [1:15] and [38:52]

Q.2 Select from T where X = g

The query is defined by the query box QB = (g,1) X (g,7) Z (g,1) = 40 and Z (g,7) = 61. Select x and y such that: $x \leq Z(T) \leq y$. This corresponds to [38:52] and [57:58].

Even though the multiattribute key index on BUB-tree supports queries on both attributes from the key regardless of their order, it suffers from one problem which is visiting nodes that

are not relevant to the query. This increases the cost of operations in the BUB-tree. To solve this problem, we employ Signature R-tree.

Again using Figure 3.13 below, we will now demonstrate how to use multidimensional approach to perform indexing on two attributes and how queries get processed on Signature R-trees.

Attribute 1	Attribute 2
A	5
B	2
C	5
D	3
e	6
f	4
g	6
h	3

Figure 3.13 Multiattribute keys formed from two attributes

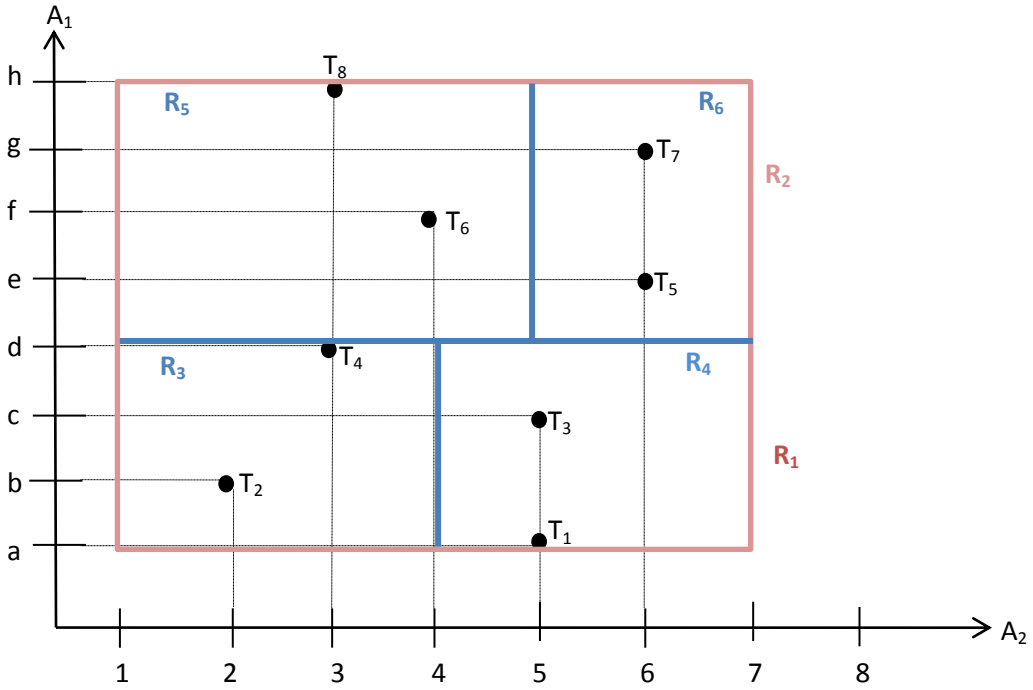


Figure 3.14 Multiattribute tuples represented in a 2D space and MBBs created on it.
Supposing we have the following tuples of the form T = (A_1, A_2):

T_1 = (a, 5), T_2 = (b, 2), T_3 = (c, 5), T_4 = (d, 3), T_5 = (e, 6), T_6 = (f, 4), T_7 = (g, 6), T_8 = (h, 3).

Let us now express the creation and application of n-dimensional signature for filtering irrelevant tree nodes. Transforming tuples (points) from the n-dimensional space into points and assuming we have 3-bit long binary characters as signatures of these tuples. We obtain:

T_1 = (a, 5), S/T_1 = (000, 100)

T_2 = (b, 2) S/T_2 = (001, 001)

T_3 = (c, 5), S/T_3 = (010, 100)

T_4 = (d, 3), S/T_4 = (011, 010)

T_5 = (e, 6), S/T_5 = (100, 101)

T_6 = (f, 4), S/T_6 = (101, 011)

T_7 = (g, 6), S/T_7 = (110, 101)

T_8 = (h, 3), S/T_8 = (111, 010)

The signatures of regions are obtained by ORing signatures of the tuples contained in the region.

$S/R_3 = S/T_2 + S/T_4$ = (001, 001) + (011, 010) = (011, 011)

$S/R_4 = S/T_1 + S/T_3$ = (000, 100) + (010, 100) = (010, 100)

$S/R_5 = S/T_6 + S/T_8$ = (101, 011) + (111, 010) = (111, 011)

$S/R_6 = S/T_5 + S/T_7$ = (100, 101) + (110, 101) = (110, 101)

Whenever S_{QB} AND $S_{REGION} = S_{QB}$, then that region is relevant to the query (i.e. it contains tuple relevant to that query), unless if it is a false drop.

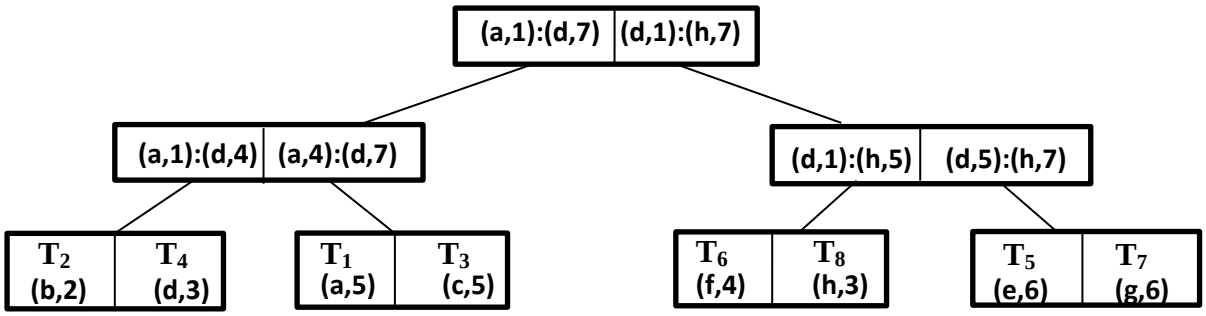


Figure 3.15 R-tree created from the MBBs

Sample queries:

Q.1 Select from T where A_1 = e

The query is defined by the query box QB = (e, 1) X (e, 7) and S_{QB} = (100, 110)

Use S_{QB} AND $S_{REGION} = S_{QB}$ to determine regions that are relevant to the query.

S_{QB} AND S_{R3} = 100 AND 011 ≠ 100 (Irrelevant to the query)

S_{QB} AND S_{R4} = 100 AND 010 ≠ 100 (Irrelevant to the query)

S_{QB} AND S_{R5} = 100 AND 111 = 100 (Relevant to the query)

S_{QB} AND S_{R6} = 100 AND 110 = 100 (Relevant to the query)

Thus, region R_5 and R_6 are relevant to the query based on the above signature operation. The tuple satisfying this query is in R_6, thus R_5 is a false drop.

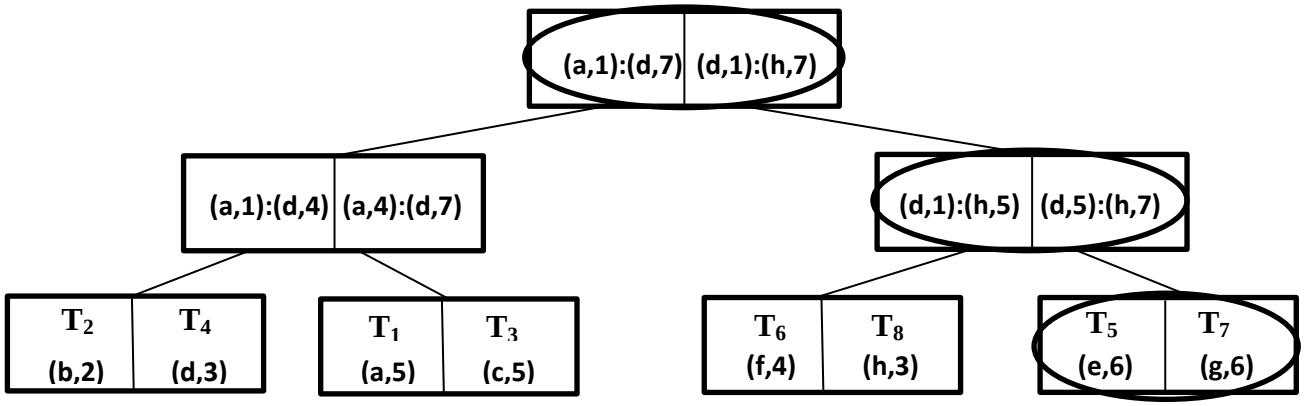


Figure 3.16 Querying for A_1 = e

Q.2 Select from T where A_2 = 5

The query is defined by the query box QB = (a, 5) X (h, 5) and S_{QB} = (111, 100)

Use S_{QB} AND S_{REGION} = S_{QB} to determine regions that are relevant to the query.

S_{QB} AND S_{R3} = 100 AND 011 ≠ 100 (Irrelevant to the query)

S_{QB} AND S_{R4} = 100 AND 100 = 100 (Relevant to the query)

S_{QB} AND S_{R5} = 100 AND 011 ≠ 100 (Irrelevant to the query)

S_{QB} AND S_{R6} = 100 AND 101 = 100 (Relevant to the query)

Thus, region R_4 and R_6 are relevant to the query based on the above signature operation. The tuples satisfying the query are in R_4 while R_6 is a false drop.

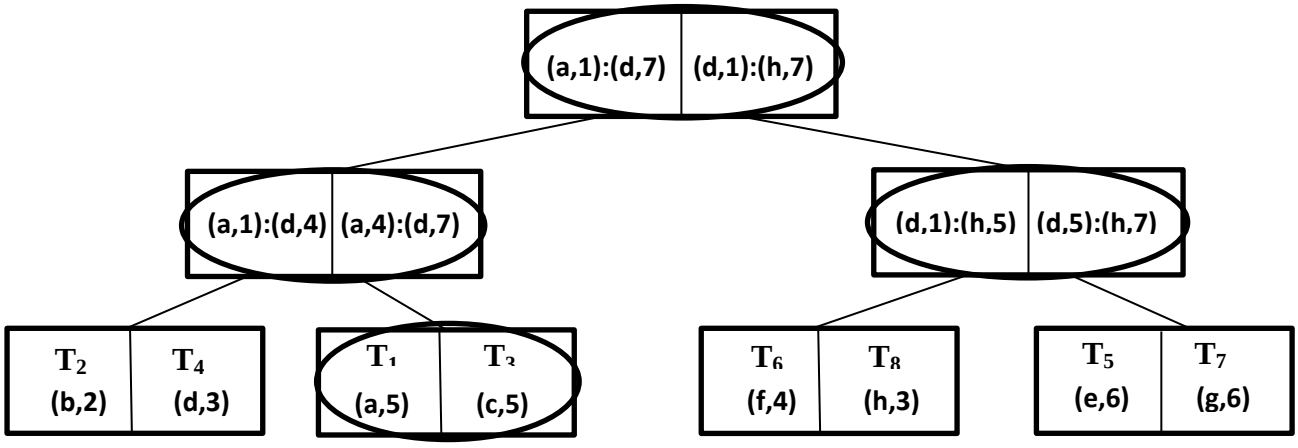


Figure 3.17 Querying for A_2 = 5

Q.3 Select from T where A_2 = 6 AND A_1 = g

The query is defined by the query box QB = (g, 6) which is a point query; it would only intersect the region that is relevant to the query. Region R_6 is the one intersected and also relevant to the query.

Q.4 Select from T where A_2 is BETWEEN 3 AND 5

The query is defined by the query box QB = (a, 3) X (h, 5) and S_{QB} = (111, 110).

Use γ (S_{QB} AND S_{REGION}) $\geq$ 1 to determine regions that are relevant to the query.

S_{QB} AND S_{R3} = 110 AND 011 = 010 γ = 1 (Relevant to the query)

S_{QB} AND S_{R4} = 110 AND 100 = 100 γ_1 = 1 (Relevant to the query)

S_{QB} AND S_{R5} = 110 AND 011 = 010 γ = 1 (Relevant to the query)

S_{QB} AND S_{R6} = 110 AND 101 = 100 γ = 1 (Relevant to the query)

Thus, the regions R_3, R_4 and R_5 are relevant to the query based on the above signature operation. R_6 is a false drop.

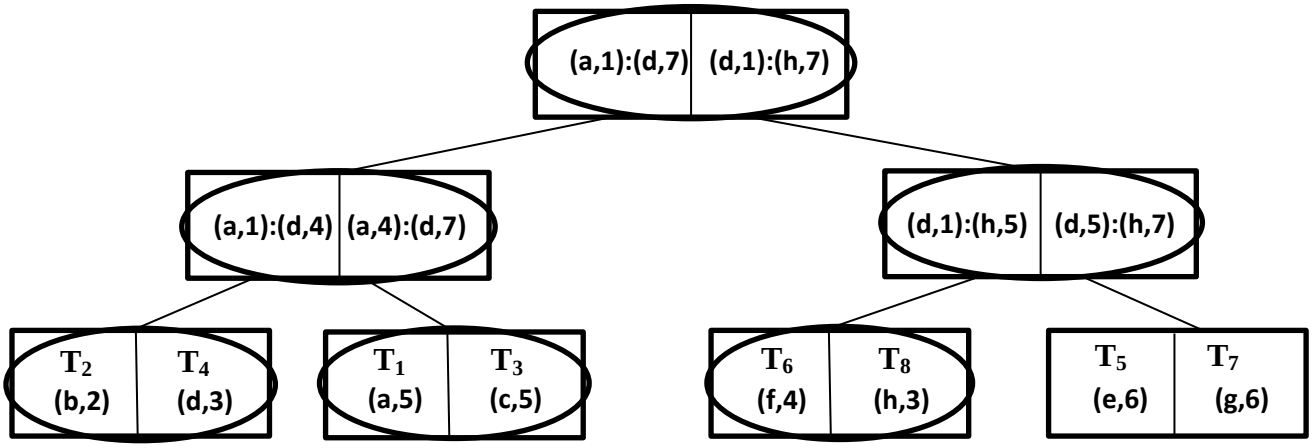


Figure 3.18 Querying for A_2 between 3 and 5

Q.5 Select from T where A_1 is BETWEEN 'b' AND 'd'

The query is defined by the query box QB = (b, 1) X (d, 7) and S_{QB} = (011, 110).

Use γ (S_{QB} AND S_{REGION}) $\geq$ 1 to determine regions that are relevant to the query.

S_{QB} AND S_{R3} = 011 AND 011 = 011 γ = 2 (Relevant to the query)

S_{QB} AND S_{R4} = 011 AND 010 = 010 γ = 1 (Relevant to the query)

S_{QB} AND S_{R5} = 011 AND 111 = 011 γ = 2 (Relevant to the query)

S_{QB} AND S_{R6} = 011 AND 110 = 010 γ = 1 (Relevant to the query)

Thus, all the regions R_3, R_4, R_5 and R_6 are relevant to the query based on the above signature operation. R_5 and R_6 are false drops because they do not contain tuples related to the query. Any co-ordinate that has all ones as its value cannot be used to filter irrelevant nodes, because it would always give a false drop.

3.5 Signature Creation

Let us now describe a method to generate n-dimensional signature responsible for narrow range query processing.

In this research, we select ASCII character encoding scheme to obtain the signatures used to filter irrelevant tree nodes. Suppose we choose to use C programming language for the implementation of algorithms, 64bits *long* data type in C would be used to represent the attribute values forming the multiattribute key. Because ASCII has 7bits per character, the maximum number of characters for an attribute would be 9 (from 64/7). The signature of an attribute is determined by its encoded ASCII equivalent. For any attribute that is shorter than the specified character (i.e. 9), it would be appended with zero (s) to ensure uniformity in the length of the bits.

3.6 Tuples Indexing and Querying in R-trees

Tuples are indexed in leaf nodes which are also clustered into regions called minimum bounding box (MBB). The MBBs can be made hierarchical again to create super regions (two points in n-dimensional space). The definitions of regions and super regions are stored in inner tree nodes with n-dimensional signature assigned to each region. As a result of which, the tree contains 2 hierarchies: the MBB hierarchy and n-dimensional signature hierarchy. A query is done on this tree by using the definition of the query box to traverse the tree looking for regions intersecting the query box and at the same time making use of signatures to reject irrelevant regions. A query is processed by retrieving those tree regions that intersect a query box. Tuples relevant to the query box are found in the disk page pointed to by R-tree leaf nodes.

3.7 Cost Analysis

The Signature R-tree complexity for basic operations like find, insert and delete is not modified. A node splitting policy depends on the type of R-tree chosen while the complexity of the splitting algorithm depends on the complexity of the selected splitting algorithm. In Signature R-trees, a change of tuples in leaf nodes transcends to changes of the signatures in the existing path, hence the complexity is maintained.

Assuming NIR to be the number of regions that intersect a given query box (regions to be retrieved when a range query is submitted) and m to be the number of indexed tuples, then the range query complexity is O ($\log_c$ (m) x NIR), where c is the fixed node capacity and NIR is the number of regions intersecting the i-th query box.

3.8 Searching

The search algorithm in Signature R-trees is almost similar to the one in B-Trees. The input is the MBB representing the query box. The searching starts from the root node downward to the leaf nodes. The query box is compared with the nodes' MBBs, if there is intersection, the search proceeds to its children nodes, otherwise it stops there. Each node's MBB corresponds to a rectangle of points in the n-dimensional space representing multi-attribute keys. Each node in the tree has 2 co-ordinates, one for the MBB and the other for the pointer to the node's children except for the leaf nodes that have pointers to the tuples of attributes from the n-dimensional space.

Algorithm Search(N, QB)

Input: Tuple, T which defines the query box QB.
Output: a set of tree tuples in the query box stored in an array A

/ Finds all MBBs that are stored in the Signature R-tree with root node N, which are intersected by a query box QB. Resulting tuples are stored in an array A */*

If N! = L
then for each entry (P, MBB) of N, check if MBB∩QB
If S_{QB} *AND* $S_{MBB} = S_{QB}$
Search (CHILD, MBB∩QB), where CHILD is the node pointed to by P
End if
Else if N = L
Check all tuples in leaf's MBBs
Return $T_n \cap QB$
End if

3.9 Insertion

To insert a new MBB into the Signature R-tree, the tree is searched and the new MBB is added to the leaf nodes. The MBB is added to the leaf node requiring least expansion. If it overflows, it would be split and the split propagated to the parent nodes up to the root node. Overflowing nodes are split and the splits are broadcasted to both their parents and children nodes. Any root node that overflows would lead to the creation of a new root which makes the previous root a child to the newly created node. In this way, the height of the tree is increased. The discussion on the split algorithm comes up in the subsequent section.

Algorithm Insert(N, QB)

Input: A tuple, T defining the query box QB
Output: The resulting Signature R-tree after the insertion of T

/ Inserts a new Tuple, T in Signature R-tree with root node N */*

```
If N! = L
        then for each entry (P,MBB) of N, check if MBB∩QB
        If S_QB AND S_MBB = S_QB
                Insert (CHILD,QB)where CHILD is a node pointed to by P
                Update all MBBs in the path up to N
        End if
        If new MBB > M
                Invoke the split algorithm
        End if
Else if N = L
        Add T into N
        If new MBB > M
                Invoke the split algorithm
        End if
End if
```

Algorithm SplitNode(N)

Input: a node N (leaf or intermediate)
Output: The resulting new Signature R-Tree

```
If N! = R
     Create n1 = (P1, MBB1) and n2 = (P2, MBB2) where MBBi = MBB ∩ ni, for  i=1,2.
     Assign all entries (Pk, MBBk) of N into ni, for i=1,2.
     For worst pairs of MBBk
     If N = L
          place each MBBk into n1 and n2.
          place the remaining entries to least enlarge ni
     Else
           use SplitNode to repeatedly split the children nodes along the partition
Else if N = R
     Create a new root NP with two children, n1 and n2.
     Replace N in PN with n1 and n2
      If PN > M, invoke SplitNode on PN again
End if
```

3.10 Deletion

To delete any MBB form Signature R-tree, the tree is searched first to find the MBB that should be deleted and removed from the leaf nodes. When node underflows as a result of

MBB's deletion, the node would be dissolved and all its children would be reinserted. In this way, the tree height decreased.

Algorithm Delete(N, T)

Input: A tuple, T defining the query box QB
Output: The resulting Signature R-tree after the deletion of T

/ Delete a tuple, T from Signature R-tree with root node N */*

If N! = L
then for each entry (P,MBB) of N, check if MBB∩QB
If S_{QB} AND S_{MBB} = S_{QB}
Delete(CHILD, T), where CHILD is the node pointed to by P
End if
If new MBB < m
Invoke the merge algorithm
End if
Else if N = L
Delete T from N
If new MBB < m
Invoke the merge algorithm
End if
End if

3.11 Range Search

Range search in Signature R-tree is almost similar to the search algorithm described in section 3.8. While the query box defines a point in the search algorithm, it defines a range or a box in the range search. Range search involves searching the Signature R-tree to determine those tuples or points which intersect with query box defining the range. The query box is compared with the nodes' MBBs, if there is intersection, the search proceeds to its children nodes, otherwise it stops there. The result of this query is the set of tuples that intersect our range.

Algorithm Merge

Input: a node N (leaf or intermediate)
Output: The resulting new Signature R-tree

/ Given L from which a tuple T has been deleted, if after the deletion of T, L has fewer than m entries, then remove the entire node and reinsert all its entries. Updates are propagated upwards and all MBBs along the path up to the root (R) are modified */*
Y = Node that underflow
Let M be the set of nodes that are going to be removed from the tree (Initially empty)
While Y! = R
    Let Parent y be the father node of Y
    Let MBB_Y be the entry of Y in Parent y
        If Y < m
            Remove MBB_Y from Parent y
            Insert Y into M
        End if
        If Y has not been removed
            Adjust its corresponding MBB, so as to enclose all MBBs in Y
        End if
        Set Y = Parent y
End while
Reinsert all tuples of nodes that are in the set M

Algorithm RangeSearch

Input: tuples A_1,A_2 which define the query box
Output: a set of tree tuples in the query box stored in an array A
Variables: a node N, a stack S which contains a current path in the tree

while (!S.Empty())
if $N \neq L$
    if there is the next MBB, in N that MBB $\cap$ QB
        Check S_{QB} AND $S_{MBB} = S_{QB}$
        if it is matched
            S.Push(N)
        else
            N = S.Pop()
else if N = L
  for any point overlapping query box
 add such points into A
N = S.Pop()

CHAPTER FOUR

4. CONCLUSION AND FUTURE WORK

4.1 Conclusion

In this thesis, a multidimensional approach to multiattribute key indexing was described. Traditional approach employs B+-tree data structure to perform multiattribute key indexing whereby queries for values of attributes are processed taking into consideration the order of the attributes. For instance, if the multiattribute key has two attributes, only queries on the first attribute or first and second attributes are processed efficiently. In proposed approach, queries on first attribute second attribute or their combination (i.e. 1^{st} and 2^{nd} attributes) can be processed on such a multiattribute key.

The multidimensional approach represents multi-dimensional keys with points in n-dimensional space. Such points are indexed by multidimensional data structures, specifically the Signature R-tree that was chosen for the purpose of this work. Various query types, such as the point, range, and similarity queries, using the chosen data structure are applied to process queries on the multiattribute keys. Queries on the first attribute, on the second attribute or on both attributes were described for a two-attribute key. Techniques for creating signatures from the attributes in addition to the techniques for creating the minimum bounding boxes were explained. Consequently, the approach is hopeful for implementation on databases. An important problem of multiattribute key indexing is the order of attributes considered during query processing, as a result of which this novel approach was proposed. This approach makes it possible to execute queries on attributes regardless of their order.

Initially, we tried to use multidimensional lattice on this problem. It was not successful, because in lattice the attributes are sorted based on the first attribute which makes it impossible to search on the second attribute. Consequently, it has failed the main objective of this thesis. A straightforward application of multidimensional data structures like R-tree is not too effective because query processing in R-trees has large overhead. A specific type of range query encountered in the multidimensional approach to indexing multiattribute key is called the narrow range query. Processing this query in the existing multidimensional data structures is inefficient as well. As a result, signature multidimensional data structures (e.g. Signature R-tree) were employed for better processing of narrow range queries.

4.2 Future Work

In our future work, we would especially like to implement the algorithm in code and run simulations to make comparisons with multiattribute keys index on B-Trees. In addition to main database operations, we desire to make the execution of other types of queries such as "like" queries and logical operators' queries possible and to improve more on the efficiency of this multidimensional approach. Finally, we would like to compare the performance of this approach using different split algorithms for MBBs and memory requirement of this approach to that of B+-trees.

REFERENCES

[1] "Database Index" from Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Database_index, March. 9, 2014 [May 12, 2014].

[2] H. Lu, Y. Yeung and Z. Tian. "T-tree or B-tree- main memory database index structure revisited," in *11th Proceedings of the Database Conference*, 2000.

[3] W. Lin, X. Tan and Y. Yu. "An Improvement of Index Method and Structure Based on R-tree," *International Conference on Computer Science and Software Engineering*, 2008.

[4] S. Harikumar and A. Vinay. "NSB-TREE for an efficient multidimensional indexing in non-spatial databases," *Recent Advances in Intelligent Computational Systems (RAICS), IEEE*, 2013.

[5] L. Tao, F.R. Xie and Y. Jia. "HVA-Index: An efficient indexing method for similarity search in high-dimensional vector spaces," *International Conference on Information Networking and Automation (ICINA)*, 2010.

[6] R. Bayer and M. Schkolnick. "Concurrency of Operations on B-Trees," *Acta Informatica*, 1977.

[7] P. Lehman and S. Yao. "Efficient Locking for Concurrent Operations on B-trees," *ACM Transactions on Database Systems*, 1981.

[8] C. Mohan and F. Levine. "ARIES/IM- An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," *ACM SIGMOD Conference*, 1992.

[9] Y. Mond and Y. Raz. "Concurrency Control in B+-trees Databases Using Preparatory Operations," *VLDB*, 1985.

[10] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*, Pearson Education, Limited, 2013.

[11] Y. Manolopoulos, A. Nanopoulos and Y. Theodoridis. *R-trees: Theory and Applications.* Springer, 2012.

[12] M. Kratky, V. Snasel, J. Pokorny, P. Zezula and T. Skopal. "Efficient processing of narrow range queries in R-tree," *ARG Technical report*, 2004.

[13] A. Silberschatz and H. Korth, *Database System Concept*, McGraw-Hill Companies, 2011.

[14] IBM Solid DB Information Centre Home, Concatenated Index, http://publib.boulder.ibm.com/infocenter/soliddb/v6r3/index.jsp?topic=/com.ibm.swg.im.soliddb.sql.doc/doc/concatenated.indexes.html, Nov. 01, 2013 [May. 01, 2014].

[15] Oracle Documentation on Indexes, http://docs.oracle.com/cd/B19306/server.102/b14231/indexes.html, Mar. 04, 2013 [May. 01, 2014].

[16] M. Karatky. "Multidimensional Approach to Indexing XML Data," PhD. Thesis, Technical University of Ostrava, 2004.

[17] J. Chang, J. H. Lee and Y. Lee. "Multi-key access methods based on term discrimination and signature clustering," *Proceedings of 12th International Conference on Research and Development in Information Retrieval,* 1989, pp.176–185.

[18] W. Walter Chang and H. Schek. "A signature access method for the starburst database system," *Proceedings of the 15th International Conference on Very Large Data Bases,* 1989, pp. 145–153.

[19] N. Beckmann, H. Kriegel, R. Schneider,and B. Seeger. "The R-tree: An efficient and robust access method for points and rectangles." *Proceedings of the International Conference on Management of Data*, 1990, pp. 322–331.

[20] U. Deppisch. "S-tree- A dynamic balanced signature index for office retrieval." *Proceedings of 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1986, pp. 77–87.

[21] P. Zezula, F. Rabitti and P. Tiberio. Dynamic Partitioning of Signature Files. ACM Transactions on Information Systems, 1991, pp. 336–369.

[22] C. Faloutsos and S. Christodoulakis. "Signature Files- An Access Method for Documents and its Analytic Performance Evaluation." *ACM Transactions on Information Systems,* 1984.

[23] B. Christian, B. Berchtold and A. Daniel. "Searching in High dimensional Spaces Index Structures for Improving the Performance of Multimedia Databases," *ACM Computing Surveys,* 2001.

[24] Y. Manolopoulos, Y. Theodoridis and J. Vassilis. *Advanced Database Indexing*. Kluwer Academic Publisher, 2001.

[25] C. Yu. *High-Dimensional Indexing,* Lecture Notes in Computer Science, Springer–Verlag, 2002.

[26] F. Robert. "The BUB-Tree," *Proceedings of 28rd VLDB International Conference on Very Large Data Bases*, 2002.

[27] R-Tree, from Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/R-Tree

[28] A. Guttman. “R-trees- A Dynamic Index Structure for Spatial Searching,” *Proceedings of the International Conference on Management of Data*, 1984, pp. 47–57.

[29] M. Kornacker and D. Banks. “High-Concurrency Locking in R-trees,” *Proceedings of the 21st International Conference on Very Large Data Bases*, 1995, pp. 34-145.

[30] T. John "The KDB-tree: a search structure for large multidimensional dynamic indexes." *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. ACM, 1981.

[31] C. Chee-Yong and E. Ioannidis. "Bitmap index design and evaluation." *ACM SIGMOD Record*. vol. 27. No. 2. ACM, 1998.

[32] K.V. Ravi Kanth, D. Serena and A.K. Singh. “Improved Concurrency Control Techniques for Multidimensional Index Structures,” *Proceedings of the 12th International Parallel Processing Symposium / Ninth Symposium Parallel and Distributed Processing*, 1998, pp. 580-586.

[33] D. Mun-Hien and C. Mohan. "Method for generating a multi-tiered index for partitioned data." U.S. Patent No.5,960,194. 28 Sep. 1999.

[34] N. Vincent and T. Kamada. “The R-Link Tree- A Recoverable Index Structure for Spatial Data,” *Proceedings of the Fifth International Conference on Database and Expert Systems Applications*, 1994, pp. 163-172.

[35] Z. Zhange, J. Yang and Y. Yang. “A new approach to creating spatial index with R-tree,”*IEEE Proceedings of the International Conference on Machine Learning and Cybernetics*, 2007.

[36] W. Lin, X. Tan, and Y. Yu. “An improvement of index method and structure based on R-tree” *IEEE International Conference on Computer Science and Software Engineering*, 2008.

[37] S. Hwang, K. Cha Kwon and B.S. Lee. “Performance Evaluation of Main Memory R-tree Variants,” Advances in Spatial and Temporal Databases, 2003.

[38] L. Arge, M. De Berg, H.J. Haverkort and K. Yi. “The Priority R-tree,” *Proceedings of the International Conference on Management of Data*, 2004, pp. 347.

[39] Advanced Database Concept: http://www.smckearney.com/adb/notes/lecture.multi.attribute.pdf [02,04,2014]

[40] V. Gaede and O. Gunther. “Multidimensional Access Methods,” ACM Computing Survey, Vol. 30 pp. 170-231, June 1998.

[41] The Wikipedia: http://en.wikipedia.org/wiki/UB-tree,Jun. 17, 2013 [May. 11, 2014].

[42] The scholarpedia: http://www.scholarpedia.org/article/B-tree_and_UB-tree, Jul. 11, 2011 [May. 11, 2014].

[43] T. Sellis, R. Nick and F. Christos. "R+ tree- A dynamic index for multidimensional trees," *Proceedings of the 13th International Conference on Very Large Database*, 1987.

[44] A. Kemper. "Pro seminar: Algorithms and Data Structures for Database Systems." University of Passau, 2003.