Perfect Hashing Base R-tree for Multiple Queries

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science and Engineering

Submitted By
Parth Patel
(Roll No. 801032018)

Under the supervision of:
Dr. Deepak Garg
Associate Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2012

Certificate

I hereby certify that the work which is being presented in the thesis entitled, "**Perfect Hashing Base R-tree for Multiple Queries**", in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Dr. Deepak Garg and refers other researcher's work which are duly listed in the reference section.

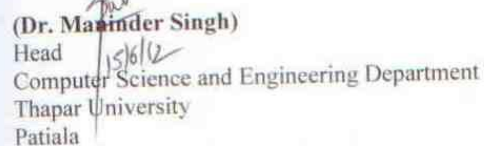
The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

(Parth Patel)

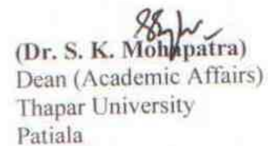
This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Dr. Deepak Garg)
Associate Professor,
Computer Science and Engineering Department

Countersigned by



(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala



(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

First of all, I am thankful to God for his blessings and showing me the right direction. With His mercy, it has been made possible for me to reach so far.

It gives me great pleasure to express my gratitude towards the guidance and help I have received from **Dr. Deepak Garg**. I am thankful for her continual support, encouragement, and invaluable suggestion. She not only provided me help whenever needed, but also the resources required to complete this thesis report on time.

I am also thankful to **Dr. Maninder Singh**, Head, Computer Science and Engineering Department and **Mr. Karun Verma** PG coordinator for his kind help and cooperation. I express my gratitude to all the staff members of Computer Science and Engineering Department for providing me all the facilities required for the completion of my thesis work.

I would like to say thanks to all my friends especially Megha, Dharya, Ashish, Nishant for their support. I want to express my appreciation to every person who contributed with either inspirational or actual work to this thesis.

Last but not the least I am highly grateful to all my family members for their inspiration and ever encouraging moral support, which enables me to purse my studies.

Parth Patel

Abstract

Information retrieval and Data base management system are one of the important filed of computer science. Popularity of internet is increased, number of people use internet are increased daily. Information on the web and database of information are becoming larger and larger. Information retrieval and database management system both need efficient query processing for storing, deleting and retrieval of data (information). In the field of computer science efficient query processing on the large data is challenging task because data is in various form and there are many types of data. Different type of data requires different type of query to process them. To store the data efficiently system need efficient data structure. So, there is a need of data structure that store data and process queries efficiently. Indexing structures are one of the data structures use for this purpose. Many indexing structure has been developed and used for the various application. But there is still need of improve that index structure for more efficient query processing. One of the major challenge is to support multiple query on the same index structure. In this thesis different index structures are discussed and one new index structure is proposed for the efficient point query and range processing on multidimensional data.

This thesis comprise of 5 chapters, chapter 1 describes information retrieval, basic index structures and hashing technics. Chapter 2 describes various index structure and comparison between them. Chapter 3 specifies the problem statement, objective defined for this thesis and methodology to achieve these objectives. Chapter 4 describes new index structure and its performance. Chapter 5 concludes the overall thesis with overall observations and future work is highlighted.

Table of Contents

Certificate........**Error! Bookmark not defined.**

Acknowledgement ....**Error! Bookmark not defined.**

Abstract iv

Table of Contents v

List of Figures vii

List of Tables viii

Chapter 1 INTRODUCTION 1

1.1 Introduction to Different Index Structure 1

1.1.1 Database Index 2

1.1.2 Index for Search Engine 2

1.2 Basic Index Structures 4

1.2.1 B-tree 4

1.2.2 R-tree 4

1.3 Different Type of Query 6

1.3.1 Point Query 6

1.3.2 Range Queries 7

1.4 Universal Hashing and Perfect Hashing 7

1.4.1 Universal Hashing 7

1.4.2 Perfect Hashing 9

Chapter 2 Different Index Structures 11

2.1 Different Index Structures 11

2.1.1 Variants of B-tree 11

2.1.2 Variants of R-tree 18

2.2 Index Structure for Different Query and Different Data Types 27
2.5 What is achieved to support Different queries and Different data types 32
2.3.1 Different types of Queries 32
2.3.2 Different Data Types 33
Chapter 3 Problem Statement 34
3.1 Objectives.. 34
3.2 Methodology 35
Chapter 4 PHR – Tree Structure and Query Performance 36
4.1 PHR-tree structure... 36
4.2 Comparison of PHR-tree structure and other state of art structure 38
4.3 Example of Multiple Queries 39
4.4 Local Operations on PHR-tree 41
4.4.1 Item Insertion 41
4.5. Performance of Multiple Queries 43
4.5.1 Point Query 43
4.5.2 Range Query 44
4.5.3 Performance analysis of multiple queries 44
Chapter 5 Conclusion and Future Scope 45
5.1 Conclusion.. 45
5.2 Future Scope... 46
References 47
List of Publications 51

List of Figures

Figure 1.1 Structure of R-tree …………………………………………………………… 4
Figure 1.2 Overlapping Relation Between Rectangles ….…………………………….. 5
Figure 2.1 Conventional B+-tree ……………………………………………………….. 15
Figure 2.2 Compact B+-tree …………………………………………………………… 17
Figure 2.3 R+-tree …………………………………………….………………………… 19
Figure 2.4 Hilbert Curve of order 1, 2 and 3……………………………………… 21
Figure 2.5 Flat chart of QR+-tree ……………………………………………………… 21
Figure 2.6 Structure chart of QR+-tree ………………………………………………… 22
Figure 2.7 The R-tree before inserting data entry C ……………………………… 24
Figure 2.8 The R-tree after inserting data entry C………………………………… 24
Figure 2.9 The new tree before inserting data entry C …………………………… 24
Figure 2.10 The new tree after inserting data entry C ……………………………… 25
Figure 2.11 BR+-tree Example …………………………….………………………….. 26
Figure 2.12 Example of Multiple Queries on BR-tree ……………………………... 26
Figure 4.1 Figure 4.1 PHR-tree Example …………………………………………… 37
Figure 4.2 Example of Multiple Queries on PHR-tree …………………………… 38

List of Tables

Table 1.1 Comparison between Index Structures …………………………………. 28
Table 4.1 Complexity Comparison of Index Structures with PHR-tree …………... 42

Chapter 1
INTRODUCTION

In computer science, index is a data structure that enables sub linear time lookup. An index is a data structure that improves performance of searching. A data store contains N objects and it is required to retrieve one of them based on value. For this number of operation in worst case is $\Omega(n)$. Since data store contain millions of object and searching is common operation, so, performance should improve. Index is one of the data structures that can be used to improve this performance. Many index design exhibit logarithmic $O(\log N)$ and in some applications it is possible to achieve $O(1)$. There are many different index structures used for this purpose. Database software and information retrieval are major application of indexes [11].

The field of the Computer Science has been devoted for the design and analysis of the Index Data Structures. Research is going on to optimize the space and time for the indexing of the various data and many new index structure are proposed and used in the indexing for efficient query processing.

Main goal of indexing is to optimize the speed of query. For any type of search or retrieval of information we ask a query and query is processed by database system or search engine internally process query on database of different content. For optimization of query, many other functions and technics are also used to improve index structure like hashing, improvement in the insert and search algorithm, combine of two index structure etc. In this chapter, introduction to hashing, basic index structures and different type of queries have been discussed.

1.1 Introduction to different Index structures

In this world of data and lots of application, mainly two type of index are there. One that is used for the storing large data with many attributes and other information for this database index and for the search engine data in the particular form is stored and search engine need fast query processing to retrieve the data in no time [12].

1.1.1 Database Index

In databases, software database index structure is used that improves speed of data retrieval operation on database table. Index can be created using one or more attributes (column) of a database table [24]. They provide with multiple type of indexes to improve performance across different application. Different types of data in the database, to construct index on these data different structure is required. All database software use indexing that enables sub-linear time look up to improve speed for the large databases. In linear time *N* objects are stored in the database need *O(N)* look up operation. Here, value of *N* is not small it's in millions. So, it is necessary to improve the performance of lookup operation. There are many index structures that are used for this purpose. Major applications are reservation system, billing system, university management, banking *etc*. Like for multidimensional data R-tree [1] and its variants are used and for simple data B-tree and its variants are used.

1.1.2 Index for Search Engine

Index for Search Engine is same as database index but it can be define like "A database where information is collected, parsed and processed then stored to allow quick retrieval." Search engine need indexes to collect, parses and store data to fast and accurate information retrieval [12]. Using indexes, different types of contents can be searched like different type of document in different languages and different media types. For indexing of the different type of data, different index structures are used. Most search engine use inverted index that is an index structure storing a mapping from content. It is the most popular data structure used in document retrieval systems. Major applications are search engine like Google, Yahoo *etc*. Inverted index is typically in form of binary trees [12].

1.2 Index structure for Information retrieval in Search engine

Meaning of Information Retrieval can be very broad. However, as a field of study information retrieval might be define thus:

Information Retrieval (IR) is finding material of an unstructured nature that satisfies an information need from within large collections (usually stored on computers) [12].

Now days user of internet are increased and hundreds and millions of people engage in information retrieval every day when they use a web search engine. Information retrieval is fast becoming the dominant form of information access, overtaking traditional database style searching.

Information retrieval systems can also be distinguished by the scale at which they operate, and it is useful to distinguish three prominent scales. In web search, the system has to provide search over billions of documents stored on millions of computers. Distinctive issues are needing to gather documents for indexing, being able to build system that work efficiently at this enormous scale. Inverted index is one of the data structures that have been used for the information retrieval.

Inverted Index: This idea is central to the first major concept in information retrieval, the inverted index. The name is actually redundant: an index always maps back from terms to the parts of a document where they occur.

The inverted index data structure is a central component of a typical search engine indexing algorithm. A goal of search engine implementation is to optimize the speed of the query: find the documents where word occurs.

Example:

Given 3 sentences as below

S_1: It is a apple.

S_2: What is it?

S_3: How it is?

Inverted index for the word occurs in these 3 statements is given below:

"a" : {1}

"apple": {1}

"how": {3}

"is": {1, 2, 3}

"it": {1, 2, 3}

"what": {2}

1.3 Basic Index Structures

B-tree and R-tree are two basic index structure used for the indexing. B-tree used for single dimensional data and R-tree used for multidimensional data. Many index structures has been developed using these index structures.

1.2.1 B-tree

The data structure which was proposed by Rudolf Bayer for the Organization and Maintenance of large ordered database was B-tree [14]. B-tree has variable number of child node with predefined range. Each time node is inserted or deleted, internal nodes may join and split because range is fixed. Each internal node of B-tree contains number of keys. Number of keys chosen between d and $2d$, d is order of B-tree. Number of child node of any node is $d+1$ to $2d+1$. B-tree keeps record in sorted order for traversing. The index is adjusted with recursive algorithm. It can handle any no of insertion and deletion. After insertion and deletion it may require rebalancing of tree.

As per Knuth's definition [15], B-tree of order n (maximum number of children for each node) is satisfied following properties:

1. Every node has at most n children.
2. Every node has at least $n/2$ children.
3. The root has at least two children if it is not a child node.
4. All leaf node at the same level.
5. A non-leaf node has n children contains $n-1$ keys.

The best case height of B-tree is $log\ mn$ and worst case height is $log\ m/2n$. Searching in B-tree is similar to the binary search tree. Root is starting then search recursively from top to bottom. Within node, binary search is typically used. Apple's file system HFS+, Microsoft's NTFS [8] and some Linux file systems, such as btrfs and Ext4, use B-trees. B+-tree, B* tree and many other improved variants of B-tree have also been proposed for specific application or data types. B-tree is efficient for the point query but not for range query and multi-dimensional data [26].

1.2.2 R-tree

Spatial data cover space in multidimension not presented properly by point. One dimensional index structure B-tree does not work well with spatial data because search

space is multidimensional. R-tree was proposed in 1982 by Antonin Guttman. It is dynamic index structure for the spatial searching [1]. It represents data object in several dimension. It is height balanced tree like B-tree. Index structure is dynamic. Operation like insertion and deletion cam be intermixed with searching.

Let *M* be the maximum number of entries in one node and minimum number of entries in a node is $m \leq M/2$. R-tree satisfies following properties [1]:

1. Each leaf node(Unless it is root) have index record between *m* and *M*.
2. Each index record (*I*, tuple- identifier) in a leaf node. *I* is smallest rectangle represented by the indicated tuple and contains the n- dimensional data object.
3. Each non-leaf (unless it is root) has children between m and *M*.
4. Each entry in non-leaf node (*I*, child pointer), *I* contain the rectangle in the child node is the smallest rectangle.
5. The root node (unless it is children) at least two children.
6. All leaves appear on the same level

Figure 1.1 and figure 1.2 show structure of R-tree and relation exist between its rectangles [1].

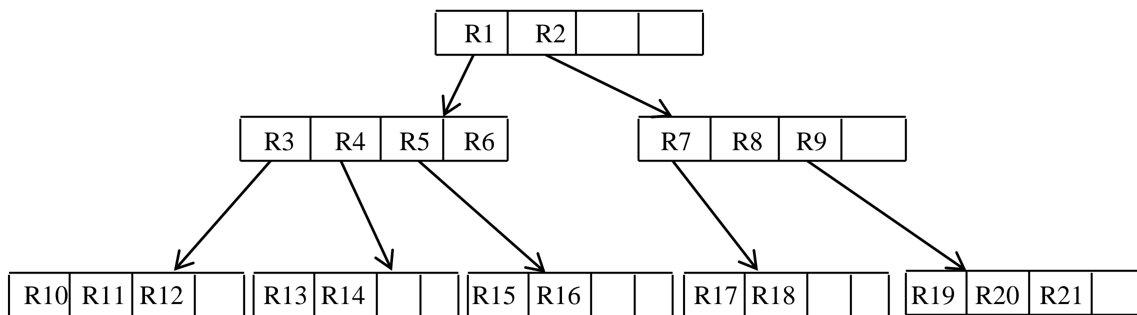


Figure 1.1: Structure of R-tree

The searching is similar to the B-tree. More than one sub tree under a node may need to be searched; hence there is not guarantee of worst-case performance. Insertion of records is similar to insertion in B-tree. New records are added and overflowed result lead to split and splits propagate up the tree. Relational database systems that have conventional access method, R-tree is easy to add. R-tree give the best performance when it is 30-40 %

full because more complex balancing is require for spatial data. Disadvantage of space wastage in R-tree variant of R-tree were also proposed. R+-tree, R*-tree, Priority R-tree, Hilbert tree, X-tree etc.

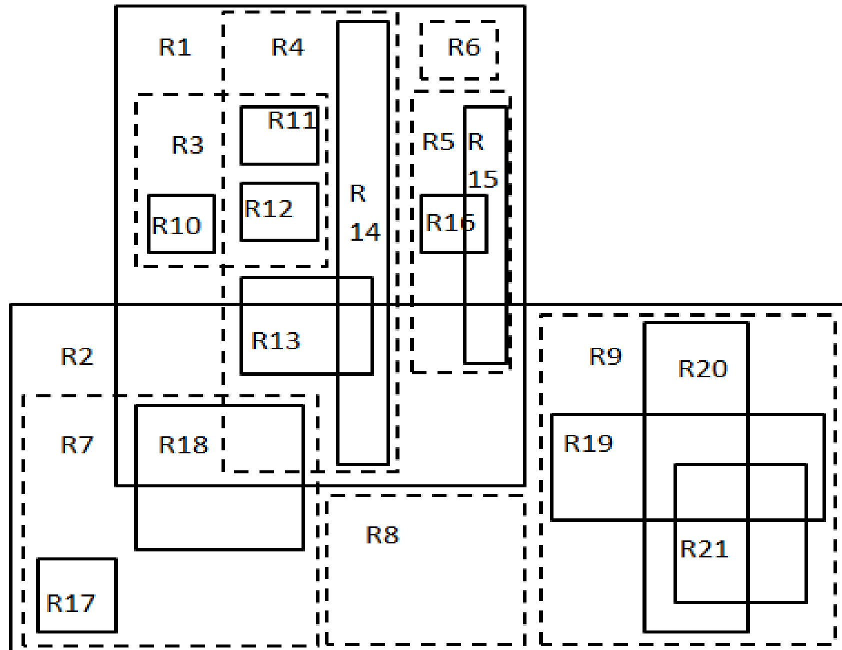


Figure 1.2: Overlapping Relation Between Rectangles

1.3 Different types of Queries

In the field of the information retrieval, query plays an important role. Different index structure support different types of data to retrieve information different type of query required like point query, range query and cover query.

1.3.1 Point Query

Support by many index structure which support single dimensional data [6]. Variants of B-tree support point query. Point query is used when there is a need to retrieve just one value or one particular set of data from the database. Point query time complexity is *O(N)* in worst case, if linear look up is done. Major index structures that support point query have *O(log N)* time complexity in the worst case because they use B-tree or its variants which support binary searching. Major issue with point query is index structures that support multidimensional data and do not support point query.

1.3.2 Range Queries

Range queries are defined in two contexts. One is range query in single dimensional data and other is range query in multidimensional data. Range query in single dimensional data is like many point query to the database [6]. Range query in multidimensional data is like point query in single dimensional data. Multidimensional data single attribute of data is stored in many dimensions and range query retrieve all the dimension of that particular attribute.

Range query does not work like point query when only one dimensional information of multidimensional data is required. Point query is needed at that time and index structure that support multidimensional data does not support point query. To support point query in the index structure that support range query, it needs some improvements like use of two index structure, use of other data structure *etc*. One such type of technique is hashing. Hashing use with index structure improves the performance and support point and range query.

1.4 Universal Hashing and Perfect Hashing

Hashing is a great practical tool and use as a dictionary data structure. The basic idea of hashing is not to search for the position of a record by comparison but to compute the position within the index. Here describe two important hashing Universal Hashing and Perfect Hashing.

1.4.1 Universal Hashing

No matter how we choose our hash function, it is always possible to devise a set of key that will hash to the same index value, that make hash function poor. To overcome this, randomization is used to choose a hash function from a carefully designed set of functions [36].

Let U be the set of universe keys and H be a finite collection of hash functions mapping U into {0, 1, …. , m-1}. Then H is called Universal if for $x, y \in U, (x \neq y)$.

$$|\{h \in H : h(x) = h(y)\}| = \frac{|H|}{m}$$

In other words, the probability of a collision for two different keys *x* and *y* given a hash function randomly choose from *H* is *1/m.*

To create a set of universal hash functions we can follow steps as following:

1. Choose the table size *m* to be prime.
2. Decompose the key *x* into $r + 1$ "bytes" so that $x = \{x_0, x_1, \ldots, x_r\}$, where the maximal value of any x_i is less than *m.*
3. Let $a = \{a_0, a_1, \ldots, a_r\}I$ denote a sequence of $r + 1$ elements chosen randomly such that $a_i \in \{0, 1, \ldots, m - 1\}$. There are m^{r+1} possible such sequences.
4. Define a hash function h_a with

$$h_a(x) = \sum_{i-0}^{r} a_i x_i \; mod \; m.$$

5. $H = \bigcup_a \{h_a\}$ with m^{r+1} members, one for each possible sequence of a.

Theorem: Class *H* defined above defines a universal class of hash functions.

Proof: Consider any pair of distinct keys *x* and *y* and assume $h(x) = h(y)$ as well as $x_0 \neq y_0$. Then for any fixed $\{a_1, a_2, \ldots, a_r\}$ it holds:

$$\sum_{i=0}^{r} a_i x_i \; mod \; m = \sum_{i=0}^{r} a_i y_i \; mod \; m$$

Hence,

$$\sum_{i=0}^{r} a_i \, (x_i - \; y_i) \; mod \; m = 0$$

Hence,

$$a_0(x_0 - \; y_0) = \; - \sum_{i=1}^{r} a_i x_i \; mod \; m$$

Note that *m* is prime and $(x_0 - y_0)$ is non-zero, hence it has a (unique) multiplicative inverse modulo *m.* Multiplying both sides of the equation with this inverse yields,

$$a_0 = \; - \sum_{i=1}^{r} (a_i x_i) \cdot \; (x_0 - y_0)^{-1} \; mod \; m.$$

and there is a unique $a_0 \; mod \; m$ which allows $h(x) = h(y)$.

Each pair of keys x and y collides for exactly m^r values of a, once for each possible value of $\{a_1, a_2, \ldots, a_r\}$. So, out of m^{r+1} combinations of $a_0, a_1, \ldots, a_r$, there are exactly m^r collisions of x and y, and hence the probability that x and y collide is $m^r / m^{r+1} = 1/m$. Hence H is universal [36].

1.4.2 Perfect Hashing

The idea present above leads to *perfect hashing*. In perfect hashing, worst case search time of $O(1)$ while using, it is only $O(n)$ space. This is achieved by a clever two step hashing scheme similar to the double hashing scheme and universal hashing.

The idea is as follows:

One uses a first hash function chosen from a family of hash function H which is universal. To hash the n keys to a table of size $O(n)$, and then hashes all elements n_j that are in the same table slot to a secondary hash table of size $O(n_j^2)$. Allocating enough space this scheme guarantees, that using linear space find constant number of hash function without collision [35].

Theorem: If we store n keys in a hash table of size $m = n^2$ using a hash function h randomly chosen from a universal class of hash functions, then the probability of there being any collisions is less than $1/2$.

Proof: There are $\binom{n}{2}$ pairs that could collide, each with probability

$$\frac{1}{m} = \frac{1}{n^2}.$$

The probability of having at least one collision is bounded by the sum of the probabilities of those collisions. Hence n

$$P_r\ (any\ collision) \leq \binom{n}{2}\frac{1}{n^2} = \frac{n(n-1)}{2n^2} \leq \frac{1}{2}$$

So, randomly and repeatedly pick a hash function until we find one without collisions. The expected number of times we need to test is a small constant.

The hash function use in perfect hashing is of the form

$$h_k(x) = (kx \; mod \; p) mod \; s$$

-where p is prime.

Use of simple doubling technique junction with static perfect hashing, such that a dynamic hash table that support insertion, deletion and lookup time in expected, amortized time *O(1)*.

Chapter 2
Different Index Structures

Index structure analysis start after Rudolf Bayer and Ed McCreight invented the B-tree while working at Boeing Research Labs in 1971 [14]. Then survey is done on B-tree shows that it also covers B+-trees [3]. These trees are widely used for indexing purpose and become most popular but having some disadvantage further analysis is required. R-tree is proposed in 1984 by Antonin Guttman. Index structure is become most immerging area of research for computer scientists as it employs different concepts on different index structure and different sorting techniques for the index generation to make more efficient search and optimize the query performance. A number of indexing structure are proposed for various application.

2.1 Different Index Structures

Some of the index structures that are widely used and some are more application or query type specific. First discuss variants of B-tree and then discuss variants of R-tree.

2.1.1 Variants of B-tree

B-tree is important index structure that is used in many applications where single dimensional data has been used, and also important to developed new index structure. Many variants of B-tree had been proposed and used in real time here few of them are discussed.

- **B+-tree**

B+-tree is similar to the B-tree the difference is all records are stored at leaf level and only keys stored in non-leaf nodes. Order of B+-tree b is capacity of node, number of children to a node referred as m, constrained for internal node that $([b)/2] \leq m \leq b$. The root allowed having as few as 2 children; the numbers of keys are at least $b-1$ and at most b. No paper on B+-tree but a survey of B-tree also covering B+-tree [6]. Figure 2.1 shows B+-tree example**.**

B+-tree is widely use in most of the rational database system for metadata indexing and also useful for the data stored in the RAM.

- **B*-tree**

To keep internal node more densely packed B*-tree balance more internal neighbor nodes [6]. This require non-root node to be at least 2/3 fill. When both nodes are full they split into three, single node gets full then it shares with the next node.

- **UB-tree**

UB-tree [20] is proposed for storing and retrieving the multidimensional data. It is like B+-tree but records are stored according to Z-order or called Morton order. The algorithm provided for the range search in the multidimensional point data is exponential with dimension so not feasible.

- **H-tree**

H-tree is a special index structure similar to B-tree but use for directory indexing. It has constant depth of one or two levels and do not require balancing, use a hash of a file name. It is use in Linux file system *ext3* and *ext4*.

- **ST2B-tree**

A Self-Tunable Spatio-Temporal B+-tree Index for Moving Objects [17]. It is built on B+-tree without change in insertion and deletion algorithms. It index moving objects as *1d* data points. *1d* key has two components: KEYtime and KEYspace. Object is updated once in a time ST2B-tree splits tree into two sub trees. Logically it splits B+-tree and each sub tree assign a range. A moving object is a spatial temporal point in natural space. For index in the space data space is partitioned into the disjoint regions in terms of reference point's distance. In this structure reference point and grid granularity are tunable. ST2B-tree meets two requirements:

1. Discriminate between regions of different densities.

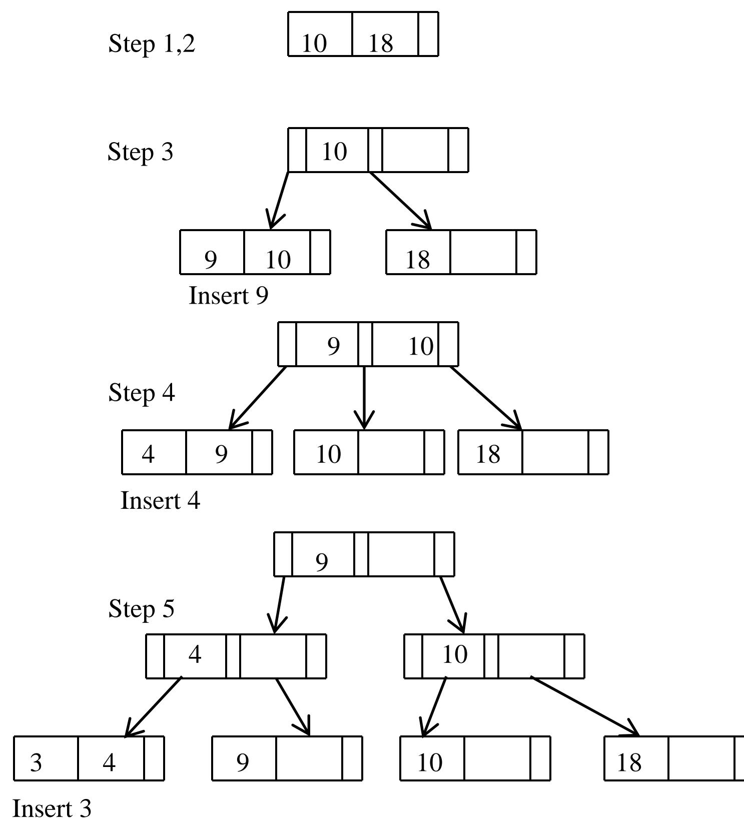
2. Adapt to density and distribution changes with time.

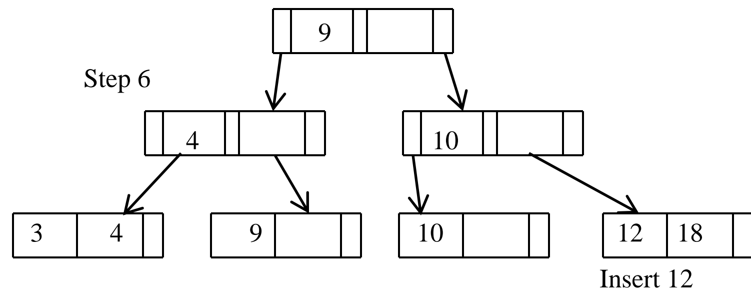
Use B+-tree for the multinational data need to reduce dimension and data density and granularity of space partition wield a joint effect on the index performance.

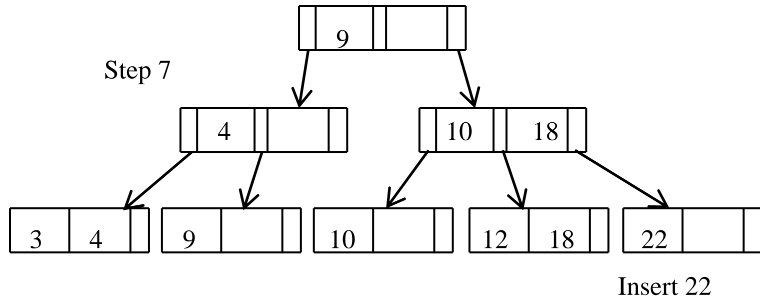
- **Compact B+-tree**

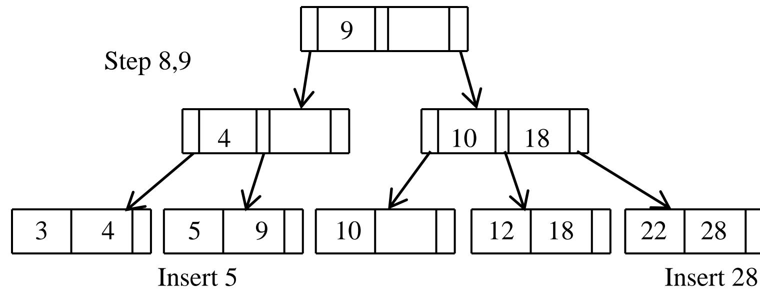
Compact B+-tree [4] is a variant of B+-tree which organizes data in more compact way via better policy. The basic idea is to use vacant space of the siblings before the overflow happen in the node. Base on this data can accommodate in external structure before splitting operation is require. Figure 2.1 and figure 2.2 shows presentation of data sequence {10, 18, 9, 4, 3, 12, 22, 28, 5, 2, 17, 11} for comparison. The result compact B+-tree requires only 6 split and 9 nodes and space utilization is (17/18). On the other hand our conventional B-tree required 9 split and 12 nodes and space utilization is (19/24). This is better policy for the insertion and split operation in traditional index eliminate.

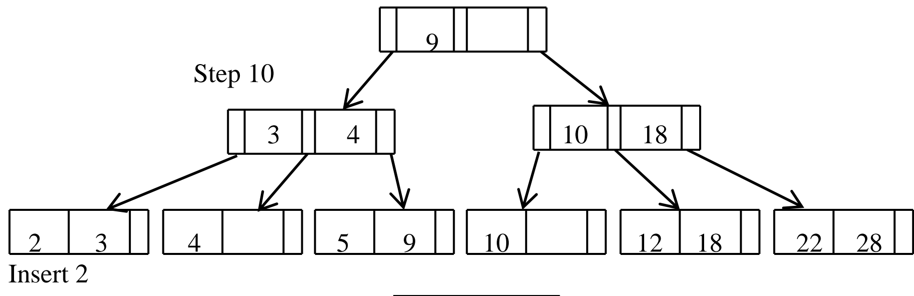
Many other variant of B-tree is also there which are not discussed in this. They are either application specific or data specific.

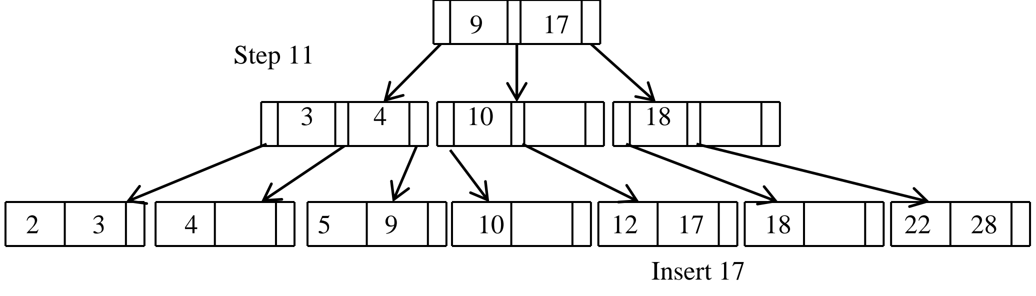



9
Step 6
4
10
3
4
9
10
12
18
Insert 12


9
Step 7
4
10
18
3
4
9
10
12
18
22
Insert 22


9
Step 8,9
4
10
18
3
4
5
9
10
12
18
22
28
Insert 5
Insert 28


9
Step 10
3
4
10
18
2
3
4
5
9
10
12
18
22
28
Insert 2


9
17
Step 11
3
4
10
18
2
3
4
5
9
10
12
17
18
22
28
Insert 17

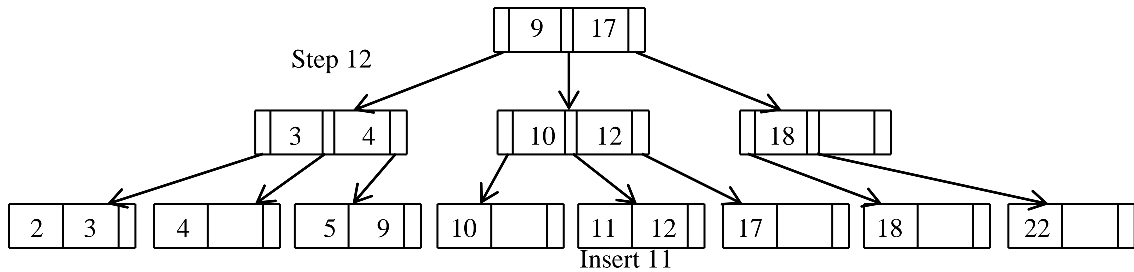


Figure 2.1: Conventional B+-tree

Step by step construction of compact B+-tree:

The basic idea of our design is to take the vacant space of the siblings around one target for dispersing the forthcoming overflow happened in the target. Data can be accommodated as many as possible into the external structure before splitting operation is required. The utilization of free space at leaf level low the frequency of node splitting. A data sequence of {10, 18, 9, 4, 3, 12, 22, 28, 5, 2, 17, 11} is respectively adopted to build a compact B+-tree and a conventional B+-tree for comparison. Node capacity and fan-out is supposed to be 2 and 3, respectively. Figure 5.1 and figure 5.2 shows the procedure step by step in constructing a conventional B+-tree and compact B+-tree [13].

Step 1: Initially, the root has accommodated data 10 and 18. Data item 9 is going to join the full root. The root must be split.

Step 2: The root splitting is done and structure is grow with one more level. Then, data item 4 is going to join the left leaf and target overflowed.

Step 3: Data item 18 is selected and migrated to the right leaf because the rest data can achieve the better aggregation in the target. As a result, data item 4 can join the target without involving any splits. The key in the root is replaced and data item 3 is going to join the left leaf again, which is already full. Because all generated leaves are full, this insertion result into split and generate new leaf.

Step 4: Data item 12 is going to join the third leaf which is full. The target fails to find its right sibling and then tries to appeal to its left sibling. The left sibling has free space. Key in the root is replaced. The entire structure is fully saturated with data and keys. The space utilization is 100% at this moment.

Step 5: Then, data item 22 is going to join the third leaf. A leaf split is needed and propagate upward to the root. The compact B+-tree grows with one more level and the space utilization down to 71.4% (10/14). Datum 7 is going to join the third leaf. The target is full but its right sibling has free space.

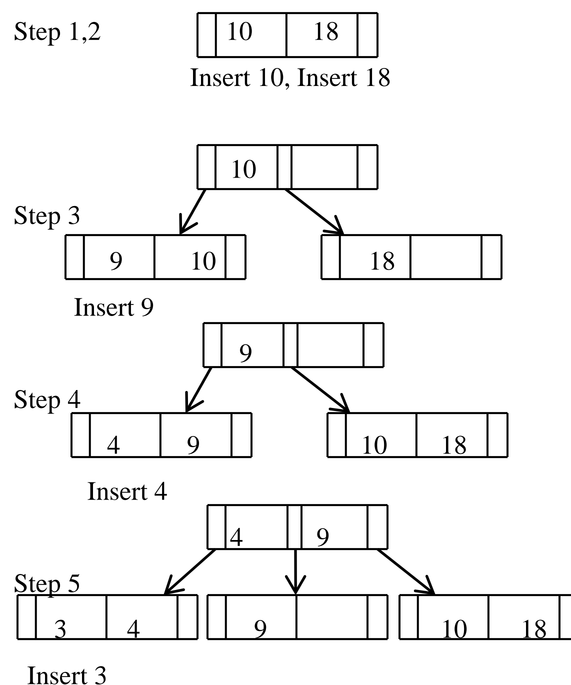
Step 6: Data item 28 going to join the last leaf and it has free space.

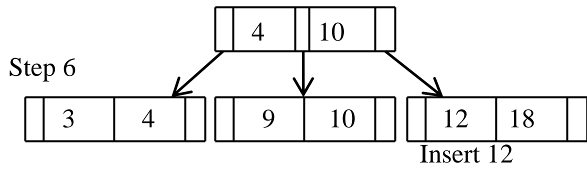
Step 7: Data item 5 is going to join the second leaf nod which is full. That result into split and generate a new leaf. Key in the parent is inserted.

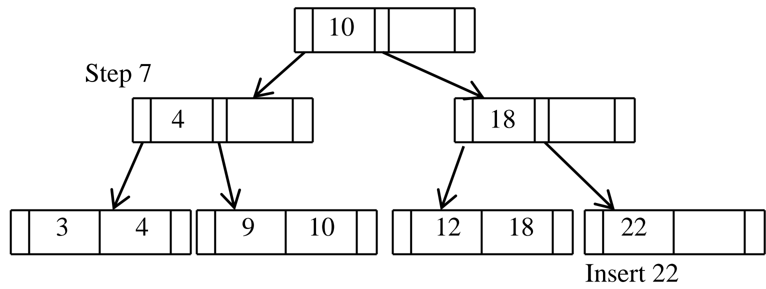
Step 8: Data item 9 is migrated to the forth leaf and then data item 2 joins the target. Meanwhile, the related key in the parents is replaced.

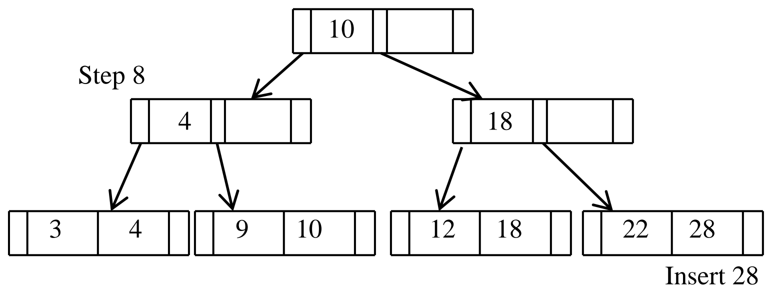
Step 9: Data item 17 is going to join the fifth leaf node which is full. That result into split and generate a new leaf. Key in the parent is inserted and data item 17 join the target.

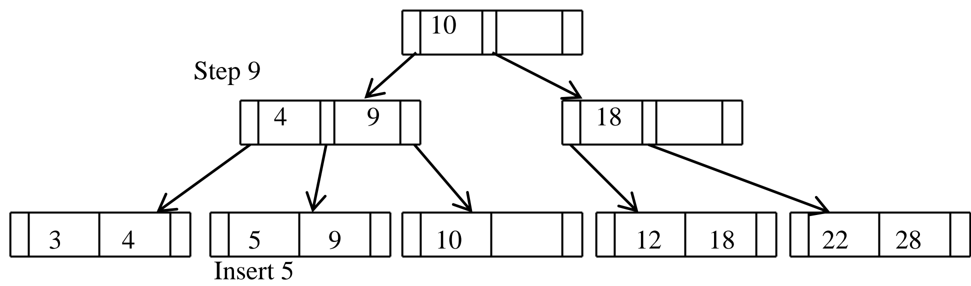
Step 10: Data item 11 is going to insert in the fourth leaf but it is full. Data item 18 migrates to its right sibling and 11 join the target. At this time space utilization is 94.44% (17/18).

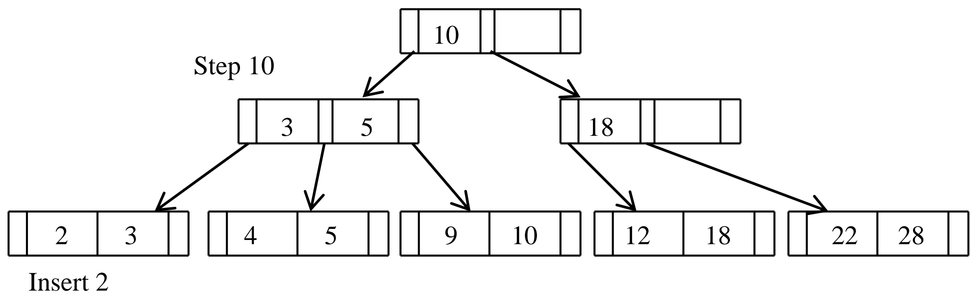



4
10
Step 6
3
4
9
10
12
18
Insert 12


10
Step 7
4
18
3
4
9
10
12
18
22
Insert 22


10
Step 8
4
18
3
4
9
10
12
18
22
28
Insert 28


10
Step 9
4
9
18
3
4
5
9
10
12
18
22
28
Insert 5


10
Step 10
3
5
18
2
3
4
5
9
10
12
18
22
28
Insert 2

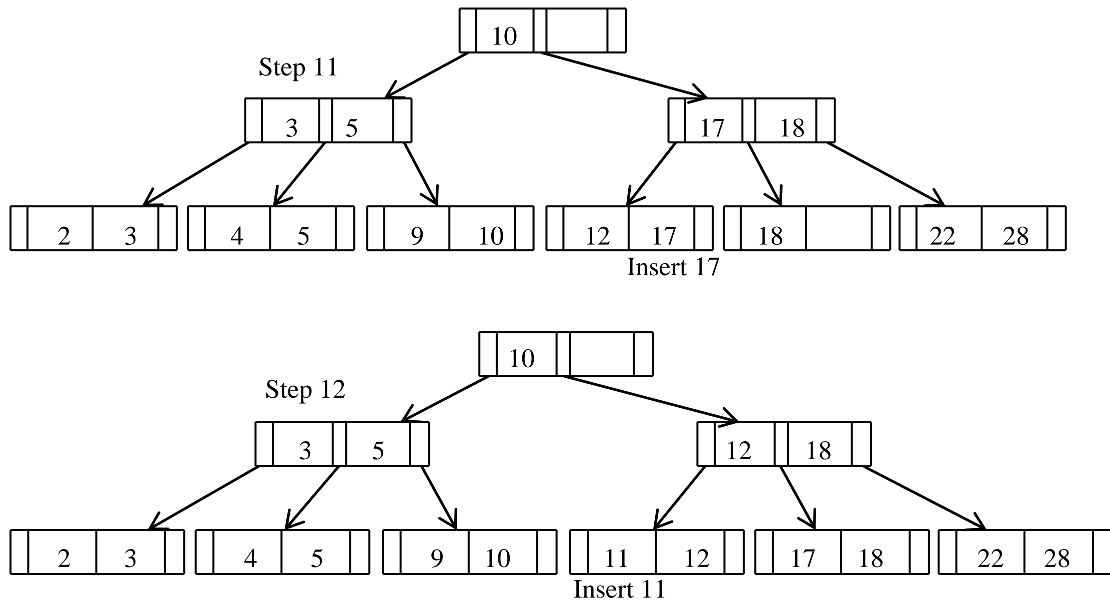


Figure 2.2: Compact B+-tree

2.1.2 Variants of R-tree

R-tree is having same importance for multidimensional data same as B-tree for single dimensional data. Many variants of R-tree had been proposed for efficient and accurate query processing and to support point query on multidimensional data. Here few variants of R-tree has been discussed.

- **R+-tree**

R+ tree is a variant of the R-tree proposed by Timos K. Sellis, Nick Roussopoulos, Christos Faloutsos in 1987 [18].

R+-tree is a variant of R-tree differs from it in

1. Nodes are not guaranteed to be at least half filled.
2. Entries of internal node do not overlap.
3. Object id may be stored in more than one leaf node.

R+-tree searching follows single path fewer nodes are visited than R-tree. But data are duplicated over many leaf node structure of R+-tree can be larger than R-tree. Figure 2.3 show R+-tree and its relation between rectangles.

Division of R+ tree remove the "Dead Region" created by overlap or R-tree nodes, it reduces individual enquiries and improve the efficiency of the space index.

Advantage: R+ tree are nodes are not overlapped with each other improve performance of point query and a single path is followed in searching and fewer nodes are visited than R-tree.

Disadvantage: R+ tree are Rectangles can be duplicated so size of an R+ tree can be larger than an R-tree build on same data set. The division of the minimum bounding rectangles may lead to the redivision of other related sub-tree nodes. Splitting operations might cause the redivision of the nodes. After the delete operation, the R+ Tree should be re-constructed [18].

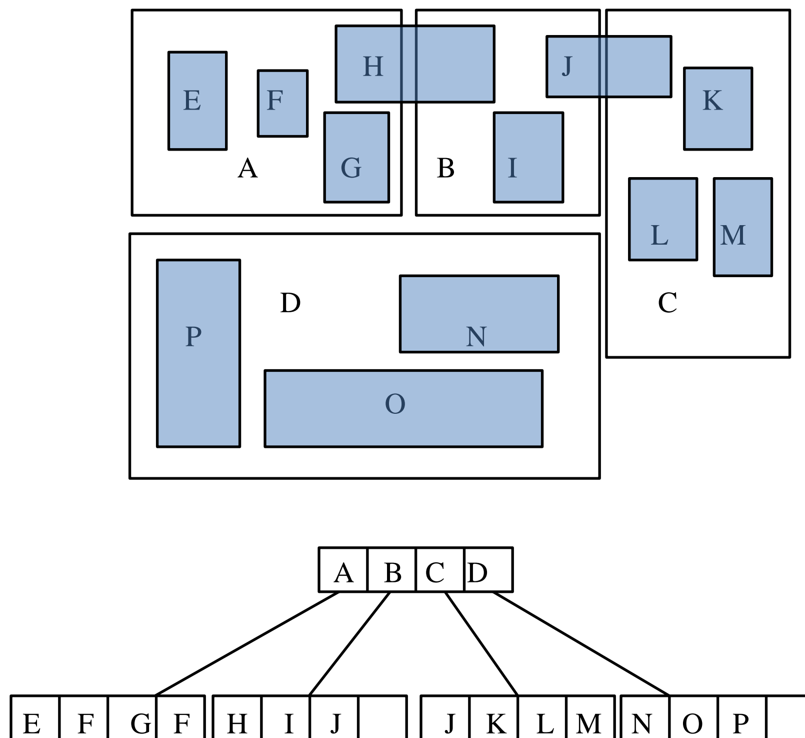


Figure 2.3: R+ tree [18]

- **R*-tree**

R*-tree [9] is also variant of R-tree its results shows that it outperform the traditional R-tree in query processing and performance. It tested parameter area, margin and overlap in different combination. To calculate overlap at each entry and with very distance rectangles probability of overlap is very small. For splitting of node R*-tree first sort lower values and then upper values of the rectangles then two group are determined.

Choose goodness of value for the final distribution of entries. Three goodness value and different approaches using them in different combination are tested.

1. Area-value,
2. Margin-value,
3. Overlap-value.

R*-tree is very robust in compare to other ugly data distribution. It's one of costly operation is reinsertion but it reduce the split operation. Storage utilization is higher than variants of R-tree but implementation cost is higher than R-tree.

- **X-tree and M-tree**

X-tree [16] and M-tree [10] are other variants of R-tree use for the same multidimensional data. Construction of M-tree is fully parametric on distance function *d* and triangle inequality for efficient queries. It has overlap of regions and no strategy to avid overlap. Each node there is radios *r*, every node *n* and leaf node l residing in node *N* is at most distance *r* from *N*. It is balanced tree and not requires periodical reorganization. X-tree prevents overlapping of bounding boxes. Which is problem in high dimension, node not split will be result into super-nodes and in some extreme cases tree will linearize.

- **Hilbert R-tree**

Hilbert R-tree [5], R-tree variant is used for indexing of object like line, curve, 3-D object and high dimension future based parametric objects. It use quad tree and z-ordering, quad tree divides object into quad tree blocks and increase no of item. It use space filling curves and specifically the Hilbert curve achieve best clustering figure 2.4 [5] show Hilbert curve. These goals can achieve for every node (a) store MBR (minimum bounding rectangle), (b) the Largest Hilbert Value of the data rectangles that being to the sub tree with root [5]. Leaf node entries of the form (R, obj_id) where R is MBR of real object and obj_id is pointer to object record. A non- leaf node entries of the form (R, ptr, LHV) where R is MBR, ptr is pointer to child node and LHV is Largest Hilbert value among data rectangle enclose by R. It give 285 of the saving over the best competitor R*-tree on Real data.

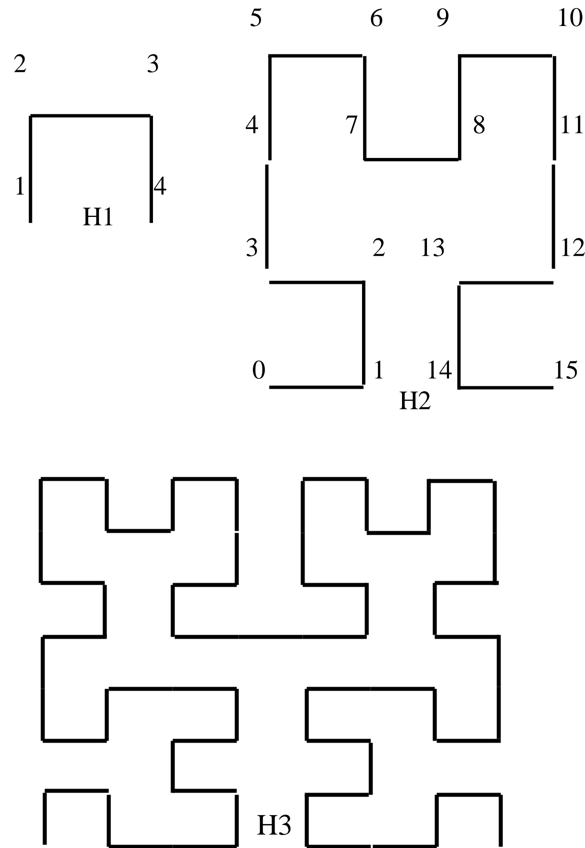


Figure 2.4: Hilbert curves of order 1,2 and 3 [5].

- **QR+-tree**

QR+-tree [7] is hybrid structure of Quad tree (Q-tree) [13] and R-tree. This structure first step is rough level partition of index space using Q-tree and second step is by using R-tree index space object. QR+-tree subdivides the spatial area and constructs the first level index. Construction algorithm of second level is improvement splitting algorithm on R-tree. Each quad has a pointer refer to the root and if quad does not have R-tree then pointer will be null. Figure 2.5 [7] shows the flat chart of QR+-tree and figure 2.6 [7] shows the structure chart of QR+-tree.

QR+-tree does not have the redundant index information that allows index to store the data directly and save the storage space. Fast and adjustable index makes query processing efficient.

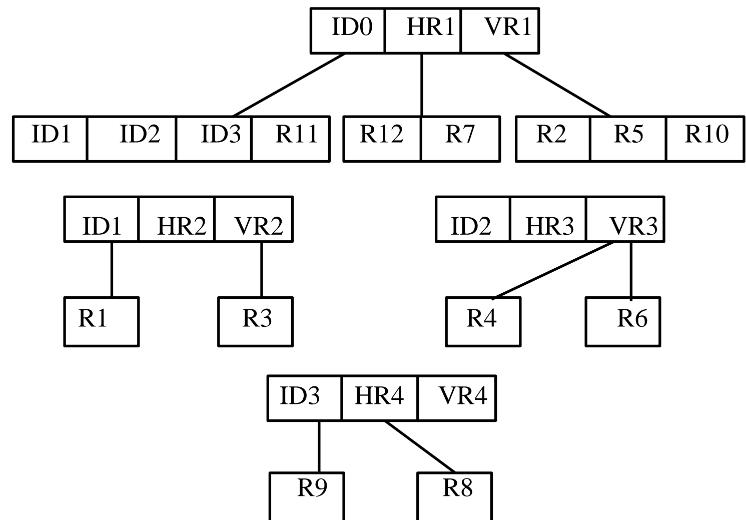


Figure 2.5: Flat chart of QR+-tree [7]

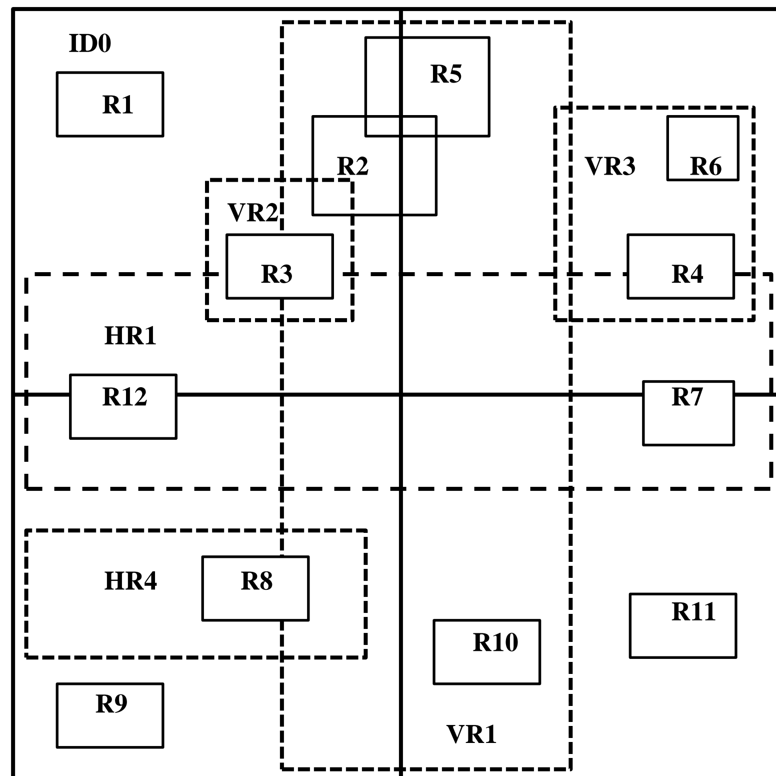


Figure 2.6: Structure chart of QR+-tree [7]

- **Improved Index Structure based on R-tree**

A spatial data object may be composed of a single point or several thousands of polygons. A large number of index structures for multi-dimensional data have been proposed to solve the problem in previous years, such as the R-tree, R*-tree , R+-tree,

RD-tree, BSP-tree, grid index and so on, which the spatial data object is conventionally represented in by external approximation, such as the minimum bounding rectangle (MBR) and minimum bounding sphere (MBS). But no access method has proven itself superior to all its competitors in whatever sense. So this shows that conventional spatial index with external approximate expression such as R-tree or R*-tree and so on, have little effect on improving access efficiency only by algorithm optimization technology. A new index structure and method based on R-tree in spatial database is proposed to improve query efficiency with the strategy of increasing the space to reduce the time. It is a variant of the current R-tree index structure, which aims to avoid the expansion of MBRs in the tree [29].

Index structure of New Index

The index structure illustrate that the data entry *C* is to be inserted into the R-tree in figure 2.7.

According to the R-tree's algorithm, the data entry *C* is stored in the leaf node and the MBR *R1* is expanded as shown in figure 2.8. However, the MBR and maximum enclosed circle (MEC) of spatial data object of data entries are allowed in a node of new tree in figure 2.9. If the data entry *C* is stored in the available space in the interior node containing *R1* and *R2*, the MBR *R1* need not be expanded and the unused space can be utilized profitably as shown in figure 2.10.

The new index structure can be defined as follows. A data entry is a 3-tuple *(IR, IC, D)*, where *IR* is an MBR which spatially contains the data object *D*, and *IC* is an MEC which spatially is contained by the data object *D*. An index entry is a 2-tuple *(B, P)*, where *B* is an MBR which spatially contains MBRs in the child node which is pointed to by *P*. A leaf contains only data entries. An interior node contains index entries, and may also contain data entries [29].

Insertion Algorithm:

Given a node *N* and a data entry *E*,

1. If *N* has enough space for *E*, then accommodate *E* in *N* and return.
2. If *N* is a leaf, then invoke R-tree Split *(N, E)*.
3. If *N* is an interior node, Let *X* be a list of index entries in *N*, and *D* be a list of data entries in *N* together with *E*. Choose, among all possible pairs of entries from *X* and

D, IE from *X* and DE from *D* such that *IE* requires the least area enlargement to accommodate *DE*. Resolve ties by choosing the pair with the smallest area of the MBR of the index entry and then the lowest marginal length of the MBR of the data entry. Remove *DE* and invoke Algorithm-Insert (Child node pointed by *IE, DE*).

4. If R-tree Split was invoked, then propagate a new index entry to the parent of *N*. If the parent of *N* has no space and contains no data entry, then invoke Algorithm-Split. If the parent of *N* has no space and contains data entries, then remove a data entry *DE* according to the lowest marginal length of the MBR of the data entry, and accommodate the parent of *N*, then choose the smallest area of the MBR of the index entry *IE* in the parent of *N*, then invoke Algorithm- Insert(*IE, DE*).

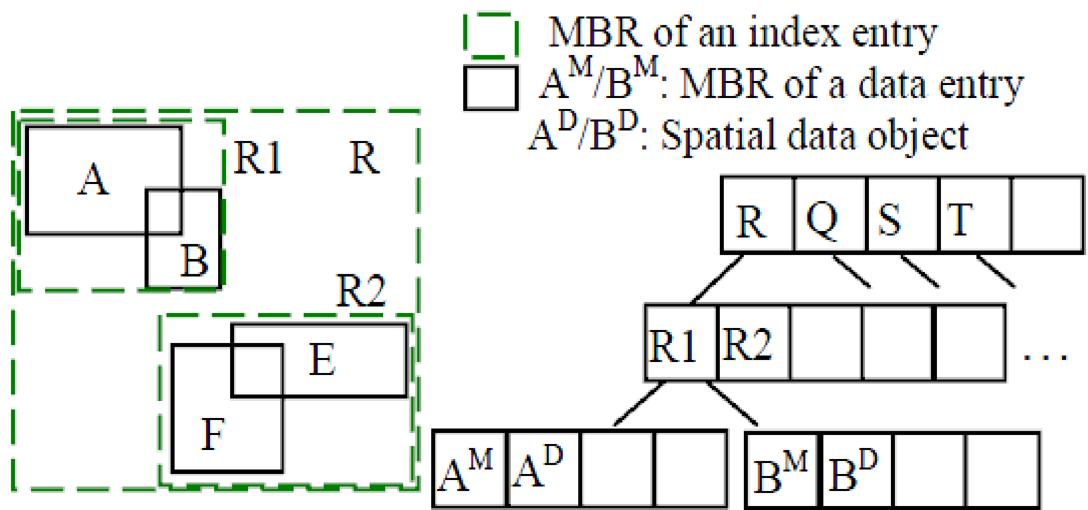


Figure 2.7: The R-tree before inserting data entry *C* [29].

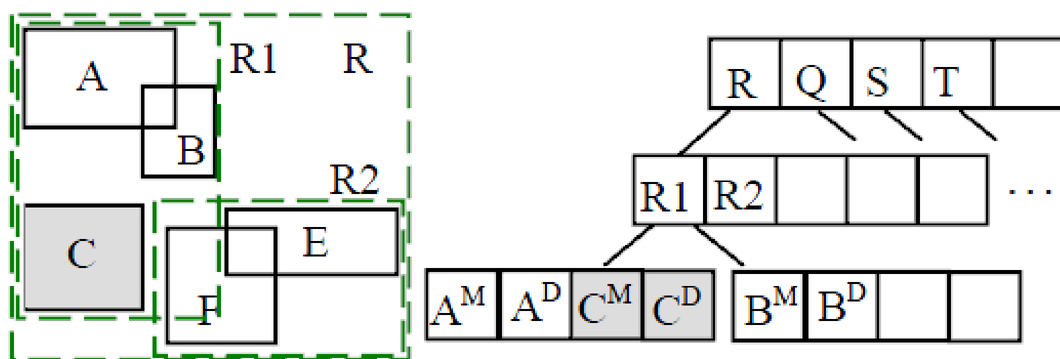


Figure 2.8: The R-tree after inserting data entry *C* [29].

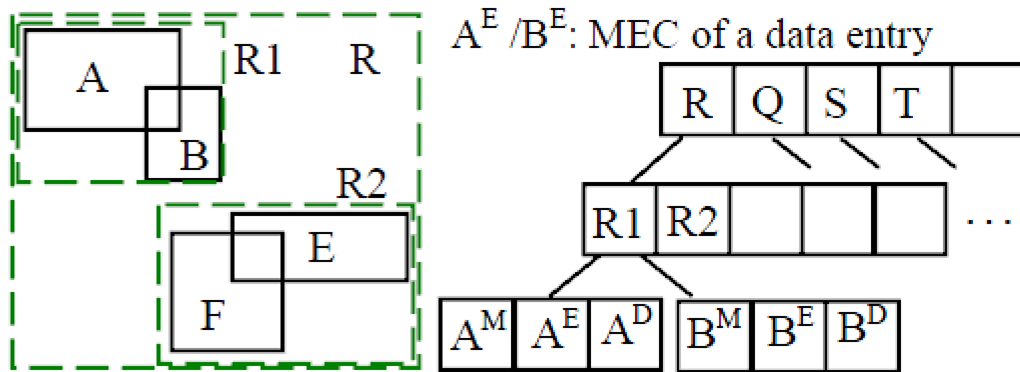


Figure 2.9: The new tree before inserting data entry *C* [29].

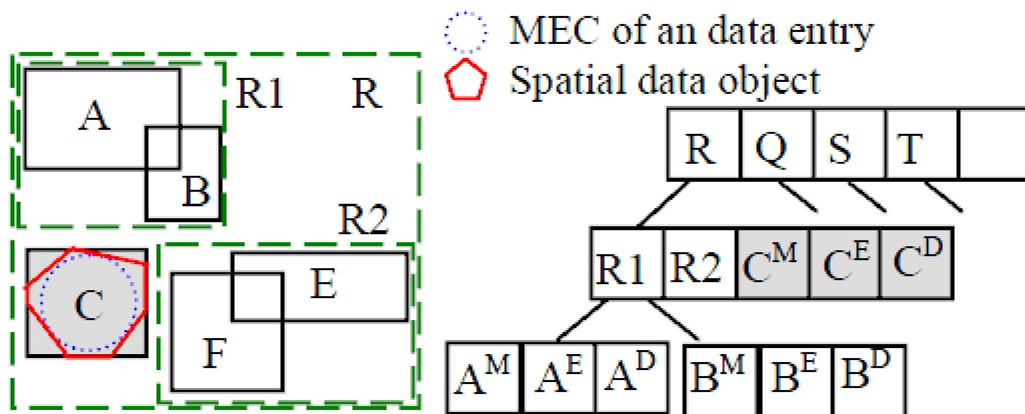


Figure 2.10: The new tree after inserting data entry *C* [29].

- **BR-tree**

Bloom filter base R-tree (BR-tree) [21] in which bloom filter is integrated to R-tree node. BR-tree is basically R-tree structure for supporting dynamic indexing. In it each node maintains range index to indicate attribute of existing item. Range query and cover query supported because it store item and range of it together. A Bloom filter is a space-efficient data structure to store an index of an item and can represent a set of items as a bit array using several independent hash functions [23]. Figure 2.11 show proposed BR-tree structure. BR-tree node is combination of R-tree node and Bloom filter.

BR-tree is proposed in 2009 use properties of R-tree and bloom filter structure. BR-tree, however, enhances query functions to efficiently support four types of queries for items with multiple attributes in *O(log n)* time complexity.

R-tree aims to be a load-balanced tree such that the nodes in the same level have approximately the same number of items. A BR-tree based on an R-tree reconfigures the segments of a multidimensional range after using bounding boxes to cover items. This guarantees that the BR-tree nodes in the same level contain approximately the same number of items [19].

BR-tree is also load balanced tree. Overloaded bloom filter produce high false positive probabilities. It reconfigures the multidimensional range using bounding boxes to cover item. BR-tree support Bound query the first index structure to talk about the bound query. Bound query result into range information of multidimensional attribute of a queried item. It is not trivial because BR-tree maintains advantage of Bloom filter and R-tree both. It mixes the queries like bound query and range query after point query result is positive. BR-tree keep consistency between queried data and the attribute bound in an integrated structure so that fast point query and accurate bound query possible. Figure 2.12 shows example of multiple queries on BR-tree.

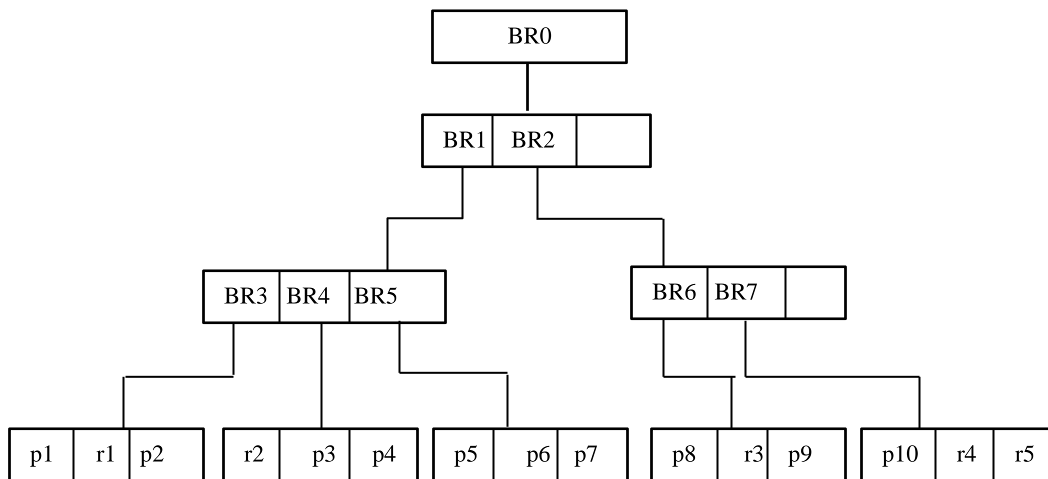


Figure 2.11: BR-tree Example [19].

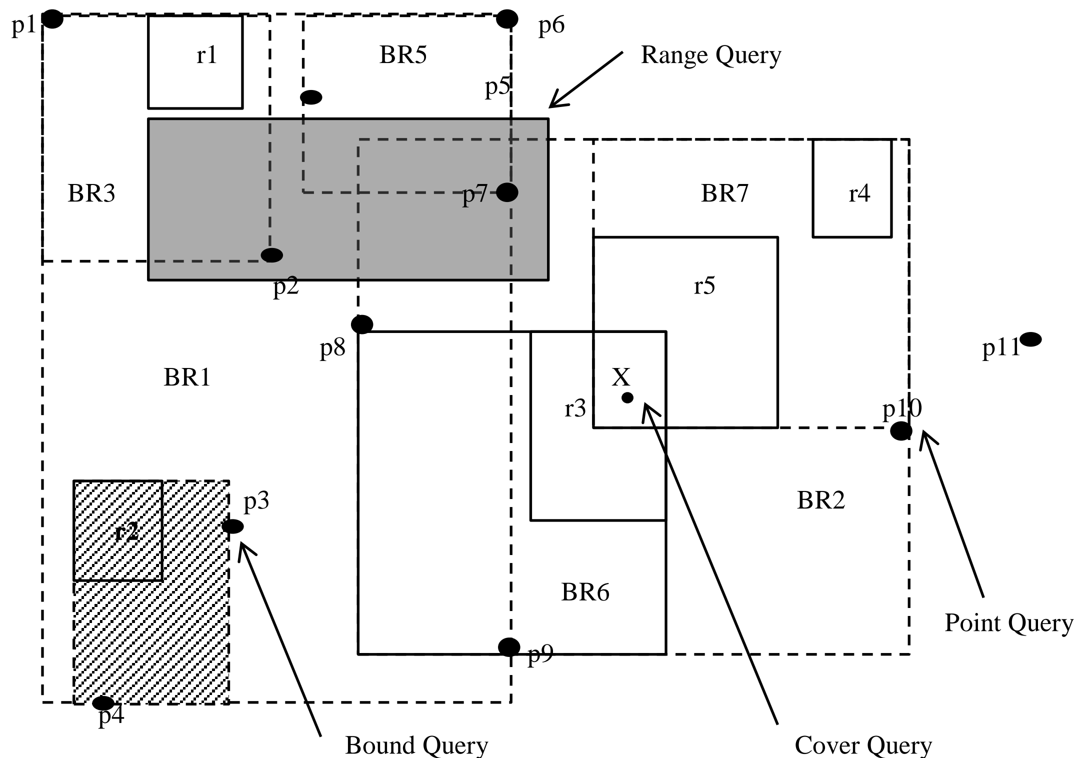


Figure 2.12: Example of multiple queries in BR-tree [19].

2.2 Index Structure for Different Query and Different Data Types

The R-tree structure [19] can efficiently support range query by maintaining index records in its leaf nodes containing pointers to their data. The completely dynamic index structure is able to provide efficient query service by visiting only a small amount of nodes in a spatial search. The index structure is height balanced. The path length from the root to any leaf node is identical, which is called the R-tree height. In essence, the family of R-tree index structures, including R+-tree [4] and R*-tree [5], uses solid Minimum Bounding Rectangles (MBRs), i.e., bounding boxes, to indicate the queried regions. The MBR in each dimension denotes an interval of the enclosed data with a lower and an upper bound [6].

A lot of work which aims to support range query efficiently has been done. In essence, existing index structures for range query often hierarchically divide data space into smaller subspaces, such that the higher level data subspace contains the lower level

subspaces and acts as a guide in the range query. Such work, however, cannot efficiently support both range query and point query.

One of the benefits using tree-based structures is to efficiently support range-based queries, such as range query and cover query, which cannot be supported by conventional hash-based schemes. VBI-tree [4] provides point and range query services and supports multiple index methods in a peer-to-peer network, which, however, is unable to support bound query. BATON [5], a balanced binary tree, can support both exact match and range queries in *O(log n)* steps in a network with n nodes. It requires certain messages to provide load balance and fault tolerance. Distributed segment tree (DST) [9] focuses on the structural consistency between range query and cover query. It needs to, respectively, insert keys and segments to support these two queries. SD-Rtree [8] intends to support point and window (range) queries over large spatial data sets distributed at interconnected servers by using a distributed balanced binary spatial tree. In addition, the main difference between BR-tree and RBF [5] is that the latter only hashes the content of the root into its correlated Bloom filter which is then replicated to other servers. Though RBF can achieve significant space savings, it cannot provide exact-matching services or support complex queries in a distributed environment.

2.3 Comparison between Index structures

In above section many index structures are discussed, here comparison between them is done with below parameters:

2.3.1 Query type

Basically 4 types of query are there Point query, range query, bound query and cover query.

2.3.2 Data type

Two types of data are there linear and multidimensional. Multidimensional data represent the object like curves, rectangles, 3-D objects. Spatial data and high dimensional data are part of multidimensional data.

2.3.3 Complexity

Each and every data structure has complexity in terms of space and time. Most of the index structures have time complexity in terms of *O(log n)*. But different index structures have different factor, terms and condition on algorithm.

2.3.4 Application

Different index structures are used for the different application for the efficient performance and some structures are introduced for the specific application only.

Table 2.1 Comparison between Index Structures

Index Structure	Query type	Data type	Complexity	Application
B-tree	Point query [1]	Linear data [1]	O(log n)	Apple's file system HFS+, Microsoft's NTFS and some Linux file systems, such as btrfs and Ext4.
B+-tree	Point Query [3]	Linear Data [3]	O(log n)	Most of the database management systems like IBM DB2, Microsoft My Sql, Oracle 8, Sybase ASE etc.
B*-tree	Point query [3]	Linear data [3]	O(log n) use space more efficiently than B+-tree	HFS and Reiser4 file systems
UB-tree	Point query, Range query [18]	Linear data, multidimensional data [18]	O(log n) but not feasible for multidimensio	Multidimensional range search.

			nal data	
H-tree	Point query	Linear data	O(log n) utilize space more efficiently.	Ext3, ext4 Linux file systems.
ST2B-tree	Range query, k-NN query [15]	Multidimensional data [15]	Work more efficiently for the moving object data.	Application with multidimensional data but now not use because other data structure outperform it.
Compact B-tree	Point query [4]	Linear data [4]	O(log n) but use space more efficiently than B-tree	In place of B-tree.
R-tree	Range query [1]	Multidimensional data [1]	Not utilize space more efficiently, not have worst case time complexity.	Real world application like navigation system etc.
R+-tree	Range query [16]	Multidimensional data [16]	Non overlapping data utilize space efficiently than R-tree	Multidimensional data object
R*-tree	Point query, Range query [9]	Spatial data, multidimensional data [9]	Implementation cost is more than other R-	Application with data in form of points and rectangles

			tree variants but robust in data distribution than other ugly structures.	
X-tree	Range query [14]	Multidimensional data, High dimensional data [14]	In some extreme cases tree become linear and time complexity O(n)	High dimension data
M-tree	Range query, k-NN query [10]	Multidimensional data [10]	Not require periodic reorganization , time is less in construction.	k-NN query, application use multidimensional (spatial) access methods [10]
Hilbert R-tree	Range queries [5]	Multidimensional data [5]	Search cost give 28% saving above R*-tree.	Cartography, Computer Aided Design(CAD), computer vision and robotics etc. [5]
BR-tree	Point query, Range query, Cover query, Bound query [19]	Linear data, multidimensional data [19]	O(≤ log n)	Application require all four type of query and also use in distributed environment [19].

QR+-tree	Range query [7]	Large scale spatial data [7]	No redundant information make query processing more efficient.	Large scale GIS database [7].

2.5 What is achieved to support different Queries and different Data types

Different types of query like point query and range query are there and different type of data like single dimensional and multidimensional data are there. To support these two things efficiently many efforts have been made. Below section describes this:

2.3.1 Different types of Queries

Point Query determines whether given item is in the data set or not and Range Query finds all the possible items whose range value is in the query. Performance of point query and range query heavily on the reliable, scalable and efficient system. Improve performance of the query and scalability of the system many structures are studied, e.g. Hash-based distributed structures like hash table [22], many tree structure like and variants of tree structures were studied and implemented like Compact B-tree, B+-tree, UB-tree, 2-3-tree etc. This type of single dimensional data structures can only support exact matching point query.

Some efforts done for fast point query, e.g. Group-Hierarchical Bloom filter Array (G-HBA) and RBF [19] but they failed to provide multiple-query services. When we use bloom filter inaccurate query results may be returned due to false positives. A perfect hash function is a function that maps a set of *n* keys into a set of *n* integer numbers without collisions so no false positives but fail to support the range query. Single dimensional data structures are space inefficient and their query result depend upon the attribute value and attribute identifier storage structure.

2.3.2 Different Data Types

Data structures for the multidimensional data like R-tree, R+-tree, R*-tree, Hilbert R-tree, Priority R-tree etc. are studies over the years. Research is going on to facilitate data storage, management and manipulation. Although R-tree structure [1] uses for multidimensional data and support range query quite efficiently. It cannot support point query efficiently because R-tree only maintains the bounding boxes of multidimensional data and the pointers to actual data. To get the point query result need to store the item identities in the leaf node but this requires large storage space when use for the real world data. Perfect hashing is an efficient design for point query but not support range query, cover query. It use hashing and not store multidimensional range information of items.

Chapter 3
Problem Statement

Traditionally, the data structures for indexing a data file are for one-dimensional space only. This one-dimensional space is either a single attribute or composite attributes with specific order. As prominent as B-tree family (B-tree, B+-tree and B*-tree), this type of data structures is not only efficient but also popular for query in one-dimensional data space. A multi-dimensional query is more complicated like range query on R-tree. Since the traditional index data structures are not effective enough for indexing multi-dimensional data, many index structures were proposed to improve the index technique. One such structure is Bloom filter Base R-tree (BR-tree). BR-tree supports range query, point query, cover query and bound query. It use Bloom filter to support and improve the query performance of point query. Bloom filter is space efficient structure to check weather an element is a member of set or not. Bloom filter return the false positive for the query result. False positive mean an element is not available in the index but result is positive. In bloom filter number of elements are more, larger probability of false positives. Problem of false positive exist with BR-tree also, due to which accuracy of BR-tree is challenged. So, there is a need to create index structure which resolves the problem of the false positive to improve the accuracy of query processing but not at the cost of time and space complexity.

3.1 Objectives

In this thesis following objectives have been defined.

- To study index structures.
- To find hashing technique that overcome problem of existing index structure.
- To propose new index structure.
- To compare existing and proposed index structure.

3.2 Methodology

Following methodology has been defined to achieve the objectives.

- To analyze efficiency and accuracy of different of different index structures.
- Comparison of different index structure based on query type, data type and complexity.
- Propose new index structure to overcome drawback of existing index structure.
- Performance analysis of new index structure in comparison with existing index structure.

Chapter 4
PHR – Tree Structure and Query Performance

4.1 PHR-tree structure

In order to achieve deterministic lookup times, a perfect hashing scheme, as shown by Kumar et al. in ([23],[33]), is very effective. These works propose the adoption of a small fast table of "discriminator" values which, together with the key, are fed to a regular hash function thus removing collisions and achieving perfect hashing.

A PHR-tree is composed of root, internal, and leaf nodes. Figure 4.1 shows an example of the proposed PHR-tree structure. A PHR-tree node combines an R-tree node with an extra Perfect Hash Index where a Perfect Hash Table is an *m-bit* array representing a set with n items by applying perfect hash function on the item set. Because an R-tree node can exhibit a series of multidimensional attribute ranges and a Perfect Hash can display items in those ranges, the combined structure encompasses multidimensional ranges to cover an item's attributes (e.g., *p* attributes) in the R-tree node and stores the hashed value of an item identifier in the Perfect Hash Index.

PHR-tree structure we use the Dynamic Perfect Hash Function for the Perfect Hash Index. A node in the Index is created whenever new value is come and node is deleted along with deletion of the item. Perfect Hash Function given below is Dynamic Perfect Hash Function which has set of Hash function and two level hash tables for the Perfect Hashing Index [37], [34].

- Set H of Hash Function h is called perfect if

$$|hH : h(x) = h(y)| = \frac{|H|}{m} \; for \; all \; x, y \in U, x \neq y.$$

- Then the Hash Function

$$h_a = \sum_{i=0}^{r} a_i k_i \; mod \; m.$$

- Size of the second level table is m_i^2, that is equal to

$$\binom{m_i}{2}\frac{1}{m_i^2} < \frac{1}{2}$$

- Expected size of all tables are sum of first level and second level tables, this sum equals to

$$n + 2E\left[\sum_{i=0}^{n-1}\binom{m_i}{2}\right] = n + \frac{2\binom{n}{2}}{n} < 2n$$

The root node (e.g., PHR0) represents domain ranges of all possible attributes. Let *R* be the maximum number of children of a node. Each internal node can contain *r* $(R/2 \leq r \leq R)$ child nodes. We set a lower bound on *r* to prevent tree degeneration and to ensure efficient storage utilization. Whenever the number of children drops below *r*, the node will be deleted and its children will be redistributed among sibling nodes. The upper bound *R* can guarantee that each tree node, in fact, can be stored exactly on one disk page. Each internal node contains entries in the form of (*I*; Perfect hash table; Pointer) where *I* = *(I0; I1; . . . ; Ip-1)* is a p-dimensional bounding box, representing an MBR as shown in figure.4.1. *Ii* is a bounded interval, which can cover items in the *ith* dimensional space. Pointer is the address of a child node. Perfect Hash Index stores all hashed values of item identities, whose multidimensional attributes are covered by the bounding box *I*. An internal node (e.g., *PHR1*) can illustrate the boundaries of a p-dimensional bounding box and the pointer to the addresses of its child nodes, and represent item identities covered by the bounding box. All leaf nodes (e.g., *PHR3*, *etc*.) appear at the bottom level and differ from internal nodes with the form (*I*; Perfect Hash Index; item-pointer) where item-pointer stores item identities and their pointer addresses. PHR-tree allows the stored item to be either a point item or range for multiple queries. From the union of child nodes, we get the bounding range of the parent node in each dimension. The range union of siblings from the same level spans the whole range as the root does. This guarantees the data integrity in the PHR-tree.

4.2 Comparison of PHR-tree structure and other state of art structure

PHR-tree is different from other state-of-the-art structures, including Bloom filter [2], baseline R-tree [1], BATON [28], VBI-tree [27], DST [30], SD-Rtree [24], and RBF [19]. PHR-tree can achieve comprehensive advantages. PHR-tree has a bounded *O(log n)*

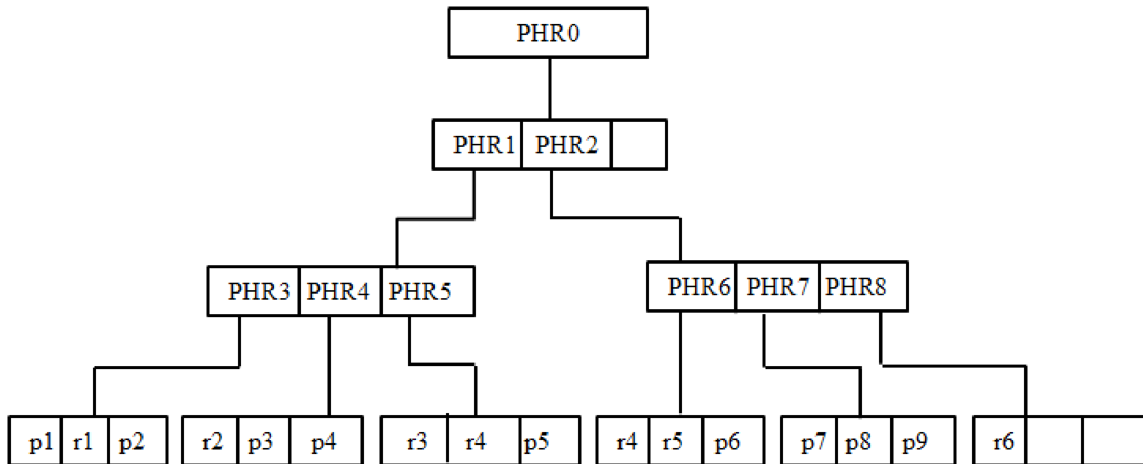


Figure 4.1 PHR-tree Example

complexity for point query. The Perfect Hash Index in the root of PHR-tree can provide fast query result with *O(1)* complexity. In applications requiring exact query results, we can follow the Perfect Hash Index branch of PHR-tree to a leaf node to verify the presence of the queried item, with the searching complexity of *O(log n)*. In such *O(log n)* complexity for point query, the real query latency is very small. The Bloom filter in the root of BR-tree can provide fast query result with *O(1)* complexity. However, the result may not be accurate due to false positive. Since Bloom filters have the same number of hash functions and counters, we need to carry out the hash-based computations for a queried item only once. Because we mainly follow the R-tree part in BR-tree to obtain range and cover query services, these queries have the same complexity as R-tree to be *O(log n)*. Meanwhile, PHR-tree structure can support bound query by checking the Perfect Hash Index along query path from the root to a leaf node, achieving *O(log n)* complexity. Perfect Hashing Index structure is an efficient design, which is also adopted in the PHR-tree. The PHR-tree structure is also able to support distributed queries as

described in the next section in detail. No existing architectures provide the aforementioned four types of queries for multidimensional data. BATON and VBI-tree aim to provide virtual indexing frameworks. Their practical performance heavily depends on the underlying used structures, not themselves. Although RBF is able to support point query, its query result is probabilistic (not exact matching). Normally, the baseline R-tree cannot support point query. However, it can do so if we specially concatenate multidimensional attributes of an item as its identity. As a result, we can compare our PHR-tree with baseline R-tree in terms of point query, with SD-R tree in terms of point and range queries, and DST in terms of range and cover queries.

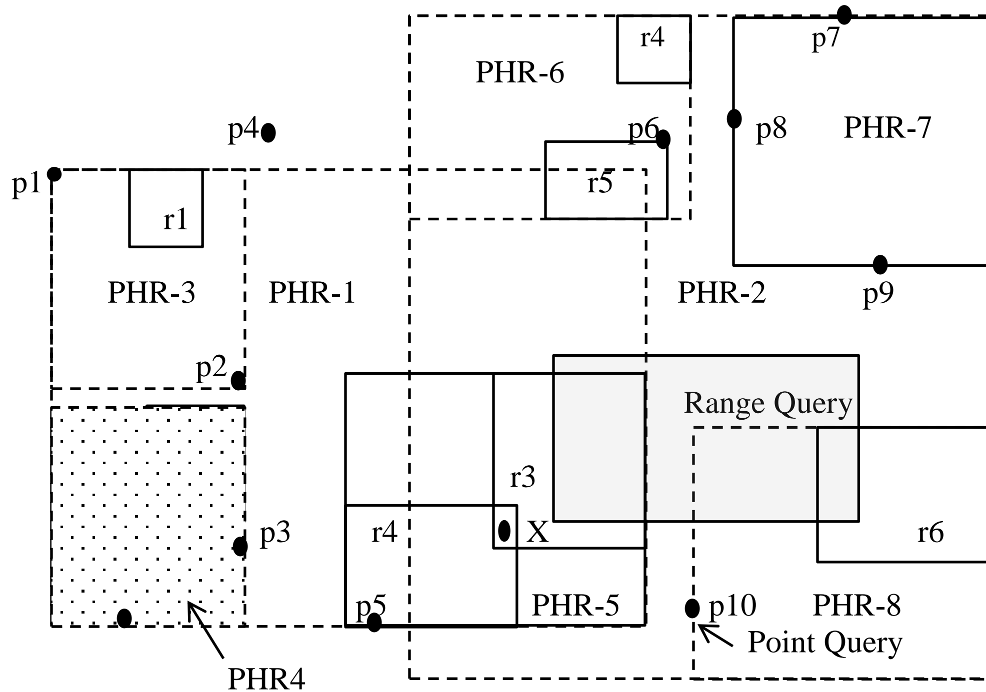


Figure 4.2: Example of Multiple Queries on PHR tree

4.3 Example of Multiple Queries

Figure 4.1 exhibits an example of PHR-tree structure. Our current data set, represented as a PHR-tree with root node *PHR0*, contains two subsets, *PHR1* and *PHR2,* respectively, having subsets, *PHR3, PHR4, PHR5* and *PHR6, PHR7, PHR8* We store data objects (represented as points *p*) and ranges (represented as ranges *r*) into our PHR-tree structure. Figure 4.2 explicitly describes multiple operations, including point and range queries, for items with two attributes, i.e., *(x, y),* in a two-dimensional space. The operations of point

query using PHR-tree become very simple and can be fast implemented compared with previous R-tree structures. For example, if we need to know whether item point belongs to our data set, we need to check the Perfect Hash Index along the query path from node *PHR0, PHR2*, to *PHR7* by computing the hash values of item point. Perfect Hash Index will return positive to the existence of point item in this example. Given the outside point in figure 4.2, the Perfect Hash Function will return negative of its existence after the computation of hash functions for item point. Note that the point query in BR-tree actually can be executed with the complexity of *O(1)* only in the root that will cause a small false positive originated from Bloom filters, but with perfect hashing no false positive. It always results positive if the item is in the Index. Bloom Filter takes *O(log n)* time to eliminate the false positive by multistep verifications on Bloom filters along the query path in the BR-tree. The processing of a range query starts from the root. If there is a node entry whose MBR intersects the query region, its sub tree is recursively explored. When the range query encounters a leaf node, we get all items whose bounding rectangles intersect the query region. For example, the shaded region in figure 4.2 intersects MBRs of both leaf nodes, *PHR5* and *PHR8*. As a result, items points will be returned for the range query. A cover query is to obtain all multidimensional ranges. Cover a given item, For example, given an item *X* in figure 4.2, a cover query can determine that the two-dimensional bounding ranges *r3* and *r4* can cover it after query operations along the path from *PHR0, PHR1* to *PHR5* and *PHR2* to *PHR 6* that contain r3 and r5. The operations of bound query are similar to those of point query. Given an item represented as a point, we need to check Perfect Hash Index along the query path from the root to a leaf node. When a leaf node containing the queried item is found, the multidimensional ranges linked to the leaf node are the queried bounds. For example, given an existed item *p3* in figure 4.1, we know that *p3* is contained in the leaf node *PHR4*. Thus, the shaded area, i.e., *PHR4*, denotes the multidimensional bounds on item *p3* that will be the bound query result. In this way, we can quickly obtain approximate multidimensional attribute ranges of an item without querying its explicit attributes. In practice, the space-efficient index structure of PHR-tree can be fully deployed into high-speed memory to provide fast query services. Although we can get tighter bounds of

items for bound queries by setting tighter MBRs on leaf nodes, the PHR-tree depth will become larger and more storage space will be required.

4.4 Local Operations on PHR-tree

The basic operations performed on every index structure are insertion and deletion. These operations define how the elements are store in structure and shape of index structure. To make any index structure efficient insertion and deletion should take minimum time.

4.4.1 Item Insertion

Insert(Data d, PHR-tree)

Leaf Node = Choose_leaf (Data D, PHR-tree);
if Count(leaf Node) ≥ R **then**
  Leaf Node = Quad_Split (Leaf Node);
end if
Insert (Data d, Leaf Node);
Current Node = Leaf Node;
while (Current node ≠ NULL) **do**
  if (D ∉ MBR (Current Node)) **then**
    ExMBR(Current Node)
  end if
  Insert (Data D, **Perfect Hashing** (Current Node));
  Current Node = Parent Node (Current Node);
end while

Insertion of an item into a PHR-tree includes operations on the R-tree and corresponding Perfect Hashing Index. Since an inserted item needs to be placed in a leaf node, we need to first locate the leaf node and then insert it. As per the algorithm to locate a leaf node for a new arrival item *a*. It use CurNode to denote a currently checked PHR-tree node. The suitable leaf node for the item can be found in *O(log n)* time, by examining a single

path as shown in the R-tree design [1]. In above insertion algorithm when adding an item *a* into our PHR-tree structure, after locating the leaf node for the new item, we can carry out node insertion. If the leaf node has room for the new item, i.e., the number of entries is less than *R*, we can execute direct insertion operations by adding item pointer into the leaf node, hashing the item into Perfect Hashing in the leaf node and all its ancestors till the root. This process is in *O(log n)* time complexity. Otherwise, we need to split the leaf node by utilizing the quadratic-cost algorithm into two leaf nodes, i.e., the old one containing old entries and the new one containing item *a*. The insertion algorithm can be applied to insert a point (or a range) object, while taking its identity as the input to an associated Perfect Hash Index.

4.4.2 Item Deletion

Data Delete (Data D, PHR-tree)

LeafNode = Choose_leaf (Data D, PHR-tree);
DeletePointer (Data D, LeafNode);
Current Node = Leaf Node;
while Count(Current Node) < r **do**
    SibNode = argmin $_{node \in Sib\ (Current\ Node)}$ Count (Node);
    if (Count(Current Node) + Count(Sib Node) < R **then**
        Merge Node (Current Node, Sib Node);
        Current Node = PatentNode (Current Node);
    end if
end while

The item deletion to be conducted in a PHR-tree node includes both deletion operations on its R-tree node and Perfect Hash Index. The item deletion operation using Bloom-filter-based structure is deemed as a difficult problem, though some possible solutions exist. Unlike the standard Bloom filter that cannot support the deletion operation because a bit *1* is likely to be set by multiple items, a counting Bloom filter is the one that effectively supports inserting, deleting, and querying items by replacing a bit in a standard Bloom filter with a counter. But in our case Perfect Hashing support the deletion

and so no need to use extra counter for the store the count. But in the perfect hashing there is no need to update counter whenever an item is inserted or deleted [32]. We first find the leaf node that contains the item to be deleted by using Choose Leaf function. PHR-tree further deletes the pointer to item a in the leaf node. Due to the item deletion, the number of items at the current leaf node may be smaller than a predefined minimum threshold *r*. Consequently, PHR-tree will proceed with the node merging operation, which combines two nodes that have fewer entries into a new one.

4.5. Performance of Multiple Queries

Table 4.1: Complexity Comparison of Index Structures with PHR-tree

Index Structure	Time Complexity (Insertion)	Space Complexity	Search Time
R-tree	O(nlog n)	O(n)	O(log n)
SD-tree	O(nm)	O(n)	O(logp f+a)
BR-tree	O(nlog n)	O(n)	O(log n)
PHR-tree	O(nlog n)	O(n)	O(≤log n)

4.5.1 Point Query

Point query allows us to determine whether a queried item *a* is a member in a given PHR-tree structure. The query result can guide us to obtain actual data-related information from pointer address in a leaf node. We can carry out point query with *O(1)* complexity only in the root, which can generate an immediate result with a relatively higher probability of false positives inherently originated from Bloom filters. In contrast, performing a query with *O(logN)* complexity in the critical path from the root to a leaf node can ensure membership presence of an item. Using the computation of hash functions, we can check the counters of the corresponding Perfect Hash Index shows the point query algorithm for an item with multidimensional attributes. If we keep the instruction in the dashed box, the algorithm complexity is *O(1)* by only checking the Perfect Hash Index of the root for item a. Since the root in a PHR-tree structure takes the union operation of its descendants in Perfect Hash Index, the union operations usually produce extra computation time. To get an exact query result, algorithm complexity

becomes *O(logN)* since we need to check nodes in a path from the root to a leaf node in the worst case [31].

4.5.2 Range Query

The main function of this algorithm is to provide item identities whose attributes fall into the request bounds of a range query. All qualified items will be included in an item set Result that is initialized to be [25]. We start the algorithm from the root of PHR-tree. Given a PHR-tree, we carry out a two-step process to implement the range query. In the first step, we search sub trees that intersect the queried range *Q* with *p*-dimensional attributes. If a CurNode has intersection with *Q,* it implies that its children may intersect *Q* as well. Thus, its child nodes will be recursively checked in the branch. Otherwise, we continue the check operation on its sibling nodes [31]. The second step is linked to the leaf nodes whose MBRs intersect request *Q*.

4.5.3 Performance analysis of multiple queries

PHR-tree support range query as well as point query. Other index structures that support range query and point query SD-tree [24], BR-tree and Distributed segment tree(DST). In SD-tree range query search time is *O(log p f+a),* while BR-tree and DST both takes *O(log n).* PHR - tree outperform all other index structure with *O(≤ log n)* time. While searching tree for the range query our structure direct search for the value and apply hash function for the search of range time is *O(1)* and height of tree is *O(log n)* time to search the node is *O(≤ log n).* Our query result come in *O(≤ log n)* time. For the point query R-tree not support point query. SD-tree and DST takes *O(log n)* time. BR-tree takes *O(≤ log n)* because it have bloom filters who reduce the cost of the searching. PHR –tree use of perfect hashing and direct use hash function for the search of point data. PHR –tree preforms better than all other structure that support the point query. While insertion takes same time in all index structures and space complexity is also almost same in all *O (n).*

PHR-tree structure performs well in both range query and point query compare to the other index structure. Here aim is to support point query on the R-tree mean structure for multidimensional data support range query as well as point query efficiently.

Chapter 5
Conclusion and Future Scope

5.1 Conclusion

Many variants of B-tree and R-tree are proposed and some of them are used in the real world for the query and performance optimization. Some index structure have less space complexity, some have less time complexity and support different data types. Most of them support point query and single dimensional data efficiently but for range query and multidimensional data specific structure is required and support specific type of data. B-tree and its variants are support point query and single dimensional data efficiently while R-tree and its variants support multidimensional data and range query efficiently. BR-tree support single dimensional, multi-dimensional and all four type of query. New index structure is proposed by making change in previous structure with use of some other data structure like hash function or use two good property of two different structure. Like BR-tree use hash function and QR+-tree use of Q-tree and R-tree. For optimize space complexity change in existing algorithm is made, like in Compact B-tree. In future take idea from this and change existing index structure. For new index structure change can be made in algorithm, use two different index structure or use data structure or use of data structure like hash in index construction.

Here we discussed PHR-tree (Perfect Hash Base R tree) which supports multiple queries on the multidimensional data. It efficiently support Point query on the multidimensional data, and range query.

PHR-tree, however, enhances query functions to efficiently support four types of queries for items with multiple attributes in *O(log n)* time complexity. Moreover, our proposed PHR-tree utilizes space efficient storage design and deviates internal nodes routing (i.e., hash result probing on the same positions), providing fast response to user queries. Use of perfect hashing instead of bloom filter gives more efficient result for the point query and also supports range query efficiently. No possibility of collision and return of false

positive unlike in the bloom filter. PHR-tree keeps consistency between the data and attribute bound in the structure.

5.2 Future Scope

- For future PHR-tree can be extended for the various data types and various application. Like for large data like video, audio, images and other multidimensional data like GIS data.
- PHR-tree will be used for the distributed data over the network. Like for Peer-to-Peer networks with use of Distributed Segment Tree (DST) [30] and Distribute Dynamic Hashing [22].
- It will be useful for Cover Query and Bound Query also.

References

[1] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching" *Proc. in ACM SIGMOD*, pp. 47–57, 1984.

[2] B. Bloom, "Space/Time Trade Offs in Hash Coding with Allowable Errors", *Communication of ACM*, vol. 13, no. 7, pp. 422-426, 1970.

[3] Douglas Comer, "The Ubiquitous B-Tree", *ACM Computing Surveys*, Vol 11, Fasc. 2, pp. 121–137, 1979.

[4] Hung-Yi Lin, "A Compact Index Structure with High Data Retrieval Efficiency", *International Conference on Service Systems and Service Management*, pp. 1–5, 2008.

[5] I. Kamel, and C. Faloutsos, "Hilbert R-tree: An improved R-tree using fractals", *Proc. in 20th International Conference on Very Large Data Bases*, pp. 500–509, 1994.

[6] Knuth, Donald, "Sorting and Searching, The Art of Computer Programming", *Addison-Wesley*, ISBN 0-201-89685-0, Vol. 3, Issue 2, Section 6.2.4, pp. 481–491, 1998.

[7] Mao Huaqing and Bian Fuling, "Design and Implementation of QR+Tree Index Algorithms", *International Conference on Digital Object Identifier*, pp. 5987 - 5990, 2007.

[8] Mark Russinovich, "Inside Win2K NTFS, Part 1", *Microsoft Developer Network*, Retrieved 2008-04-18.

[9] N. Beckmann, H. P. Kriegel, R. Schneider and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles", *Proc. in ACM SIGMOD International Conference on Management of data*, pp. 322-331, 1990.

[10] Paolo Ciaccia, Marco Patella and Pavel Zezula, "M-tree An Efficient Access Method for Similarity Search in Metric Spaces", *Proc. in 13th International Conference on Very Large Data Bases*, 1997.

[11] Maron, Melvin E.: "An Historical Note on the Origins of Probabilistic Indexing",

Information Processing and Management, Vol 44, Issue 2, pp. 971–972, 2008.

[12] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, “An introduction to information retrieval online edition”, *Cambridge university press*, 2009.

[13] R. A. Finkel and J. L. Bentley, “Quad trees: a data structure for retrieval on composite keys”, *Acta Informatica*, vol 4, pp. ll-9, 1974.

[14] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes", *Acta Informatica*, Vol. 1, Fasc. 3, pp. 173–189, 1972.

[15] Ramakrishnan Raghu and Gehrke Johannes, “Database Management Systems”, *McGraw-Hill Higher Education*, edi. 2nd, pp. 267, 2000.

[16] Stefan Berchtold, D. A. Keim and Hans-Peter Kriegel, "The X-Tree: An Index Structure for High-Dimensional Data", *Proc. In 22th International Conference on Very Large Data Bases*, pp. 28–39, 1996.

[17] Su Chen, Beng Chin Ooi, Kian-Lee Tan and M. A. Nascimento, “ST2B-tree: A Self-Tunable Spatio-Temporal B+-tree Index for Moving Objects”, *Proc. in ACM SIGMOD international conference on Management of data*, 2008.

[18] Timos K. Sellis, Nick Roussopoulos and Christos Faloutsos, “The R+-Tree: A Dynamic Index for Multi-Dimensional Objects”, *Proc. in VLDB 13th International Conference on Very Large Data Bases*, pp. 507-518, 1987.

[19] Y. Hua, D. Feng, H. Jiang, and L. Tian, “RBF: A New Storage Structure for Space-Efficient Queries for Multidimensional Metadata in OSS,” *File and Storage Technologies (FAST) Work-in- Progress Reports*, 2007.

[20] V. Markl, “MISTRAL: Processing Relational Queries using a Multidimensional Access Technique”, *Infix Verlag*, ISBN 3-89601-459-5, 1999.

[21] Yu Hua, Bin Xiao and Jianping Wang, “BR-Tree: A Scalable Prototype for supporting Multiple Queries of Multidimensional Data”, *IEEE Transactions on Computers*, vol. 58, Issue 12, pp. 1585–1598, 2009.

[22] Devine, "Design and implementation of DDH: A Distributed Dynamic Hashing Algorithm", *Proc. In 4th International conference Foundation of Data Organizations and Algorithms*, pp. 101 – 114, 1993.

[23] D. Ficara, S. Giordano, S. Kumar and B.Lynch, "Divide and Discriminate: Algorithms for fast and deterministic hash lookups", *Proc. In ACM symposium on Architecture for Networking and Communication Systems*, 2002.

[24] C. Du Mouza, W. Litwin and P. Rigaux, "SD-tree: A Scalable Distributed R-tree", *Proc. In International Conference of Data Engineering*, pp. 296 – 305, 2007.

[25] V. Gaede and O. Gunther, "Multidimensional Access Methods", *ACM Computing Surveys*, Vol. 30, issue 2, pp. 101-114.

[26] E. Bertion, B. C. Ooi, R. sacks-Davis, K. L. Tan, J. Zobel, B. Shidlovsky and B. Cantania, "Indexing techniques for Advanced Database Applications", *Kluwer Academics*, 1997.

[27] H. V. Jagadish, B. C. Ooi, Q. H. Vu, R. Zang and A. Zhou, "VBI-Tree: A Peer-to-Peer Framework for Supporting Multi-Dimensional Indexing Schemes", *Proc. In International Conference on Data Engineering*, 2006.

[28] H. V. Jagadish, B. C. Ooi, Q. H. Vu, R. Zang and A. Zhou, "BATON: A Balanced Tree Structure for Peer-to-Peer Networks", *Proc. In International Conference on Very Large Data Bases*", pp. 661-672, 2005.

[29] Weihua Lin, Yonggang Wu, Xiaojun Tan and Yan Yu, "An Improvement of Index Method and Structure Based on R-tree", *International Conference on Digital Object Identifier*, Vol. 4, pp. 607 – 610, 2008.

[30] C. Zheng, G. Shen, S. Li and S. Shenker, "Distributed Segment Tree: Support of Range Query and Cover Query Over DHT", *Proc. In International Workshop on Peer-to-Peer Systems (IPTPS),* 2006.

[31] C. Bohm, S. Berchtold and D. A. Keim, "Searching in High- Dimensional Spaces Index Structures for Improving The Performance of Multimedia Databases.", *ACM Computing Surveys*, Vol. 33, Issue 3, pp. 322 – 373, 2001.

[32] A. Gionis, P. Indyk and R. Motwani, “Similarity Search in High Dimensions via Hashing”, *Proc. In 25th International Conference on Very Large Data Bases*, pp. 518 – 529, 1999.

[33] A. Fiat and M. Naor, “Implicit O(1) Probe Search”, *SIAM Journal of Computing*, Vol. 22, Issue 1, pp. 1 – 10.

[34] E. A. Fox, Q. F. Chen, A. M. Daoud and L.S. Heath, “Order Preserving Minimal Perfect Hash Functions and Information Retrieval.”, *ACM Transactions on Information Systems*, Vol. 9, Issue 3, pp 281 – 308, 1991.

[35] B. S. Majewski, N. C. Wormald, G. Havas and Z. J. Czech, “A Family of Perfect Hashing Methods”, *Computer Journal*, Vol. 39, Issue 6, pp. 547-554, 1996.

[36] Z. J. Czech, G. Havas and B. S. Majewski, “Fundamental Study – Perfect Hashing”, *Theoretical Computer Science*, Vol. 182, Issue 1, 1997.

[37] F. C. Bothelho, R. Pagh and N. Ziviani, “Simple and Space- Efficient Mininal Perfect Hash Functions”, *Springer-Verlag Lecture Notes in Computer Science*, Vol. 4619, 2007.

List of Publications

Published:

Parth Patel, Deepak Garg, “Comparison of Advance tree data structure.” International Journal of Computer Applications, March 2012, Vol. 41, No.2, pp. 11-21, Foundation of Computer Science, New York, USA.

To Be Communicated:

Parth Patel, Deepak Garg, “Perfect Hashing Base R-tree for Multiple Queries.”