IMPLEMENTATION OF X-TREE WITH 3D SPATIAL INDEX AND FUZZY SECONDARY INDEX

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

SİNAN KESKİN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

DECEMBER 2010

Approval of the thesis:

IMPLEMENTATION OF X-TREE WITH 3D SPATIAL INDEX AND FUZZY SECONDARY INDEX

submitted by **SİNAN KESKİN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı _____
Head of Department, **Computer Engineering**

Prof. Dr. Adnan Yazıcı _____
Supervisor, **Computer Engineering Dept., METU**

Assoc. Prof. Dr. Halit Oğuztüzün _____
Co-Supervisor, **Computer Engineering Dept., METU**

Examining Committee Members:

Assoc. Prof. Dr. Ahmet Coşar _____
Computer Engineering Dept., METU

Prof. Dr. Adnan Yazıcı _____
Computer Engineering Dept., METU

Assoc. Prof. Dr. Halit Oğuztüzün _____
Computer Engineering Dept., METU

Assoc. Prof. Dr. Uğur Güdükbay _____
Computer Engineering Dept., Bilkent University

Asst. Prof. Dr. Pınar Şenkul _____
Computer Engineering Dept., METU

Date: 24 December 2010

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.

Name, Last name :

Signature :

ABSTRACT

IMPLEMENTATION OF X-TREE WITH 3D SPATIAL INDEX AND FUZZY SECONDARY INDEX

Keskin, Sinan

M.Sc., Department of Computer Engineering

Supervisor : Prof. Dr. Adnan Yazıcı

Co-Supervisor : Assoc. Prof. Dr. Halit Oğuztüzün

December 2010, 80 pages

Multidimensional datasets are getting more extensively used in Geographic Information Systems (GIS) applications in recent years. Due to large volume of these datasets efficient querying becomes a significant problem. For this purpose, before creating index structure with these enormous datasets, choosing an efficient index structure is an urgent necessity.

The aim of this thesis is to develop an efficient, flexible and extendible index structure which comprises 3D spatial data in primary index and fuzzy attributes in secondary index. These primary and secondary indexes are handled in a coupled structure. Firstly, a 3D spatial primary index is built by using X-tree structure, and then a fuzzy secondary index is overlaid over the X-tree structure. The coupled structure is shown more efficient on a certain class of queries than uncoupled index structures comprising 3D spatial data in primary index and fuzzy attributes in secondary index separately. In uncoupled index structure, we provided 3D spatial primary index by using X-tree index structure and fuzzy secondary index by using BPlusTree index structure.

Keywords: spatial indexing, multidimensional data indexing, fuzzy indexing, X-tree

ÖZ

3B UZAMSAL DİZİNLİ VE BULANIK İKİNCİL DİZİNLİ X-AĞACI GERÇEKLEŞTİRİMİ

Keskin, Sinan

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi : Prof. Dr. Adnan Yazıcı

Ortak Tez Yöneticisi : Yrd. Doç. Dr. Halit Oğuztüzün

Aralık 2010, 80 sayfa

Coğrafi Bilgi Sistemleri uygulamalarında çok boyutlu veri kümeleri son yıllarda daha yaygın kullanılmaktadır. Bu büyük veri kümeleri üzerinde etkili sorgulama yapmak önemli bir problem haline gelmiştir. Bu amaçla devasa veri kümeleri ile dizin oluşturmadan önce, etkili bir dizinleme yapısı seçmek kaçınılmaz bir gerekliliktir.

Bu tezin amacı, 3B uzamsal verileri birincil dizinde ve bulanık nitelikleri ikincil dizinde içerebilen etkili, esnek ve genişletilebilir bir dizin yapısı oluşturmaktır. Bu yapıdaki birincil ve ikincil dizinler tek bir yapı bünyesinde ele alınmıştır. İlk olarak X-ağacı yapısı kullanarak 3B uzamsal birincil dizin oluşturulmuş, sonrasında bulanık ikincil dizin X-ağacı yapısı üzerine kaplanmıştır. Bütünleşmiş yapının, 3B uzamsal birincil dizin ile bulanık ikincil dizinin ayrı ayrı ele alındığı ayrık dizin yapılarının kullanımından daha etkili olduğu belirli sorgular üzerinde gösterilmiştir. Ayrık dizin yapısında, 3B uzamsal birincil dizini X-ağacı dizin yapısını kullanarak ve bulanık ikincil dizini ise B+ ağacı dizin yapısını kullanarak sağladık.

Anahtar Kelimeler: uzamsal dizinleme, çok boyutlu veri dizinleme, bulanık dizinleme, X-ağacı

To My Father

ACKNOWLEDGEMENTS

First of all, I would like to thank to my thesis advisor, Prof. Dr. Adnan Yazıcı, and co-advisor, Assoc. Prof. Dr. Halit Oğuztüzün, for their guidance, support and motivation they provided throughout my research. I would like to thank to Dr. Aziz Sözer from METU for all his helps.

Also, I would like to thank to Dipl.-Inf. Daniel Schäfer from Philipps-Universität Marburg for his suggestions, comments and contributions.

And also I would like to thank to my elders for their best wishes.

My special thanks and appreciation goes to all my mates and friends who shared all the happiness and pains I felt.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

In spatial world we can define objects as points, lines and polygons. If we want to manage these objects in a database, we can use spatial database that is optimized to store and query data related to these objects. In general concept of database, operations are performed on various numeric and character types of data. However in spatial database concept, additional functionalities should be implemented for databases to process spatial data types.

In addition, numeric and character types are used for indexing to look up values in traditional database systems faster. On the other hand, these indexes are not optimal for spatial queries. Also, indexes used by non-spatial databases cannot provide efficient features such as how far two points differ or whether given points fall within a spatial area of interest. Therefore, spatial index is used in spatial databases to speed up database operations and also to optimize spatial queries.

Even though, the rectangular bounding boxes of huge multidimensional datasets lead to good performance by the derivatives of R-tree, such R*-tree, it has also been proved that other indexing structures based on R-tree, such as X-tree, perform very well for multidimensional data.

We not only studied with pure indexing structure of X-tree, but also added fuzzy secondary indexing on this extension of R-tree and performed for multidimensional meteorological datasets. After that, we tested its features and performance of X-tree having 3D spatial primary index and non-spatial fuzzy secondary index.

For this purpose firstly we developed our spatial index that could allocate 3D spatial data. So X-tree index structure was adopted to allocate 3D spatial data. After implementing this X-tree index structure, we overlaid fuzzy secondary index on our base index. Therefore our index structure became a coupled type that could store both primary and secondary indexes in single index structure.

Then we developed uncoupled index structure for handling 3D spatial data and fuzzy attributes in separate constructions. Storing 3D spatial data was supported by using X-tree index structure and fuzzy attributes were saved in BPlusTree index structure. We used X-tree as primary index and BPlusTree as secondary index.

In overview section we began to give information about spatial indexing. After the definition of spatial indexing, spatial indexing techniques were provided. Spatial indexing techniques were explained and example implemented techniques were given in expression.

Afterward in background section, we presented X-tree to understand its features and structure. XXL API that is one of the vital parts of our thesis work and has core structure of X-tree was particularly expressed. Basic components, function composition, query processing and advanced features of the XXL API were described.

Subsequently in implementation section, we explained implementation detail of the X-tree. We firstly informed about coupled index structure which allocates primary and secondary indexes in monolithic structure. And secondly we gave information about uncoupled index structure which allocates primary and secondary indexes separately. In this section, detail knowledge about creating primary and secondary indexes was expressed.

In performance tests section, we prepared test scenarios about X-tree index structure. We applied point, range and nearest neighbor queries over coupled and uncoupled index structures. We observed test result, draw graph of test results and made deductions about them.

CHAPTER 2

AN OVERVIEW ON SPATIAL INDEXING

2.1 About Spatial Indexing

A multidimensional or spatial index utilizes some kind of spatial relationship to organize data entries, with each key value seen as a point (or region, for region data) in a k-dimensional space, where k is the number of fields in the search key for the index. Spatial indexes are ideal for queries such as, "Find the 7 nearest neighbors of a given point" and, "Find all points within a certain distance of a given point".

2.2 Spatial Indexing Techniques

Extensive research has been carried out on multidimensional indexing structure for last twenty years, for enabling efficient range queries and nearest neighbor searches. Yet, the big percentage of recent studies have center upon high-dimensional feature-based similarity searches into a relatively small number of point data items [19]. Every day, lots of application can produce hundreds of gigabytes of spatio-temporal data daily, consisting of billions of individual data elements. Storing each data element in a big scientific dataset into a multidimensional indexing tree is unserviceable, due to the fact that the index size could be even bigger than the dataset, and the queries performance could be poor owing to the index size. We can use bounding box for each array elements, instead of using spatial information. In this array there are data elements which have similar spatial coordinates. Then we can store bounding box into indexing tree to decrease the size of the index and make index searches faster [19]. Storing bounding boxes for small subsets of datasets into

spatial indexing structures, such as R-trees [4] or its variants, allows for direct access to subsets of a dataset in order to improve data access performance.

2.2.1 Spatial Indexing Structures

There are two types of method for creating spatial indexing structure. One of them is space partitioning method and other one is data partitioning method [19]. Both of them are related to the performance and detail information about them is given in this section.

2.2.1.1 Space Partitioning Methods

Space partitioning is the process of dividing a space into two or more disjoint subsets. In other words, space partitioning divides a space into non-overlapping regions [22]. Any point in the space can then be identified to lie in exactly one of the regions. Space-partitioning systems are often hierarchical, meaning that a space (or a region of space) is divided into several regions, and then the same space-partitioning system is recursively applied to each of the regions thus created. The regions can be organized into a tree, called a space-partitioning tree [5].

The big percentage of space-partitioning systems use planes to divide space: Points are used to divide the regions. Partitioning space using planes in this way produces a BSP (Binary Space Partitioning) tree, one of the most common forms of space partitioning recursively. Most space-partitioning systems use planes (or, in higher dimensions, hyper planes) to divide space: points on one side of the plane form one region, and points on the other side form another. Points exactly on the plane are usually arbitrarily assigned to one or the other side. Recursively partitioning space using planes in this way produces a BSP tree, one of the most common forms of space partitioning [22].

Space partitioning is particularly important in computer graphics, where it is frequently used to organize the objects in a virtual scene. Storing objects in a space-partitioning data structure makes it easy and fast to perform certain kinds of geometry queries — for example, determining whether two objects are close to each other for collision detection, or for determining whether a ray intersects an object in ray tracing [1].

There are some examples of common space partitioning systems such as KDB-tree, Spatial KD-tree, Hybrid-tree, BSP trees, Quadtrees, Octrees [19].

2.2.1.2 Data Partitioning Methods

Data Partitioning is the formal process of determining which data belongs to which data site. It is an orderly process for allocating data to data sites that is done within the same common data architecture. Data partitioning methods, based on R-trees, store all spatial data in bounding box and perform well for hyper rectangular data.

R-tree and R-tree*: Instead of duplicating objects, spatial objects can be indexed by allowing overlapping regions, as in R-tree based index structures [4]. Although R-trees can be used for non-point data, a large amount of overlap between internal nodes in R-trees leads to search performance problems. To reduce overlapping regions for R-trees, Beckmann et al. proposed an improved version of R-trees, called R*-trees [2]. The R*-tree insertion algorithm reinserts elements from a node that overflows, instead of splitting the node. This forced reinsertion feature of R*-trees improves search performance, but node insertion can become very expensive.

X-tree: Another extension of the R-tree developed by Berchtold et al., called X-tree [3], which avoids highly overlap bounding boxes via the use of “supernodes”. A supernode is a tree node that spans multiple pages on disk, thus has a larger capacity than a normal node. When a node must be split and a large amount of overlap between sub-partitions is unavoidable, the X-tree algorithm increases the capacity of

the node instead of splitting it. If there would be a large amount of overlap between two nodes after a split, the probability that both nodes would be accessed by a search operation is high. Hence, sequential access to supernodes should be faster than random access to two separate nodes. However, supernodes have the overhead of additional disk management costs at index creation time. Therefore, before the X-tree insertion algorithm creates a supernode, it tries to find an overlap-free split based on past split history [3]. However, split history is not useful for non-point spatial objects, because an overlap-free split is not always possible for non-point data. Even if an overlap-free split can be found, in most cases it will not be acceptable since it will not meet minimum node utilization requirements.

2.3 Problems of (R-tree-based) Index Structures in High-Dimensional Space

Characteristics of the R*-tree were extensively studied for understanding the factors that cause performance problems. Query performance is directly related with the overlap in the directory since multiple paths have to be followed even for simple point queries [24]. There is no generally accepted definition for overlap in the directory especially for the high-dimensional case. In the following, we therefore provide an overview of overlap definitions.

2.3.1 Definition of Overlap

Intuitively, overlap is the percentage of the volume generated by more than one directory hyperrectangle. This intuitive definition of overlap is directly correlated to the query performance since in processing queries, overlap of directory nodes results in the necessity to follow multiple paths, even for point queries [23].

Definition 1a (Overlap)

The overlap of an R-tree node is the percentage of space covered by more than one hyperrectangle. If the R-tree node contains n hyperrectangles $\{R_1, ..., R_n\}$, the overlap may formally be defined as

$$Overlap = \frac{\|\cup_{i,j \in \{1,..,n\}, i \neq j} (R_i \cap R_j)\|}{\|\cup_{i \in \{1,..,n\}} R_i\|} \quad (1)$$

The amount of overlap measured in Definition 1a is related to the expected query performance only if the query objects (points, hyperrectangles) are distributed uniformly. A more accurate definition of overlap needs to take the actual distribution of queries into account [3]. Since it is impossible to determine the distribution of queries in advance, in the following we will use the distribution of the data as estimation for the query distribution. This seems to be reasonable for high-dimensional data since data and queries are often clustered in some areas, whereas other areas are virtually empty. Overlap in highly populated areas is much more critical than overlap in areas with a low population [24]. In second definition of overlap, the overlapping areas are therefore weighted with the number of data objects that are located in the area.

Definition 1b (Weighted Overlap)

The weighted overlap of an R-tree node is the percentage of data objects that fall in the overlapping portion of the space [3]. More formally,

$$WeightedOverlap = \frac{|\{ p | p \in \cup_{i,j \in \{1,..,n\}, i \neq j} (R_i \cap R_j) \}|}{|\{ p | p \in \cup_{i \in \{1,..,n\}} R_i \}|} \quad (2)$$

In Definition 1a, overlap occurring at any point of space equally contributes to the overall overlap even if only few data objects fall within the overlapping area. If the query points are expected to be uniformly distributed over the data space, Definition 1a is an appropriate measure which determines the expected query performance. If the distribution of queries corresponds to the distribution of the data and is no uniform, Definition 1b corresponds to the expected query performance and is

therefore more appropriate [3]. Depending on the query distribution, we have to choose the appropriate definition.

So far, we have only considered overlap to be any portion of space that is covered by more than one hyperrectangle. In practice however, it is very important how many hyperrectangles overlap at a certain portion of the space. The so-called multi-overlap of an R-tree node is defined as the sum of overlapping volumes multiplied by the number of overlapping hyperrectangles relative to the overall volume of the considered space [3].

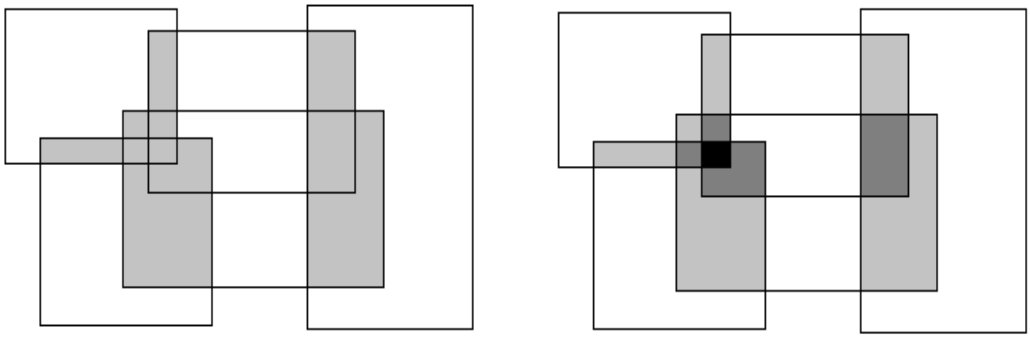

Figure 1: Overlap and Multi-Overlap of 2-dimensional data

In Figure 1 [3], we showed a two-dimensional example of the overlap according to Definition 1a and the corresponding multi-overlap. The weighted overlap and weighted multi-overlap (not shown in the figure) would correspond to the areas weighted by the number of data objects that fall within the areas [3].

CHAPTER 3

BACKGROUND

Preliminary concept and related work is presented in this section to provide a clear understanding of this thesis work.

We gave detailed information about X-tree, XXL API and FCM algorithm. Given information about these subjects are vital base of our work.

3.1 X-tree

The features of the X-tree (eXtended node tree) index structure can be summarized as follows [3]:

- Based on R-tree
- Used for storing data in many dimensions (not only point data but also extended spatial data)
- Supports efficient query processing of high-dimensional data
- Uses extended variable size(supernodes)
- Provides a directory organization and also uses the available main memory more

The X-tree may be seen as a hybrid of a linear array-like and a hierarchical R-tree-like directory as shown Figure 2 [3]. It is well established that in low dimensions the most efficient organization of the directory is a hierarchical organization. The reason is that the selectivity in the directory is very high which means that, e.g. for point

queries, the number of required page accesses directly corresponds to the height of the tree.

MBR0	SplitHistory0	Ptr0		MBRn-1	SplitHistoryn-1	Ptrn-1

Figure 2: Structure of a Directory Node

3.1.1 Structure of X-tree

The heterogeneous structure of X-tree is illustrated in Figure 3 [3]. There are three different types of nodes in X-tree and they can be given as in the following way;

- Data Nodes: These nodes contain rectilinear Minimum Bounding Rectangles (MBRs) together with pointers to the actual data objects,
- Normal Directory Nodes: These nodes contain MBRs together with pointers to sub-MBRs.
- Supernodes: They are large forms of directory nodes and they have variable sizes. They are used mainly to avoid splits in the directory.

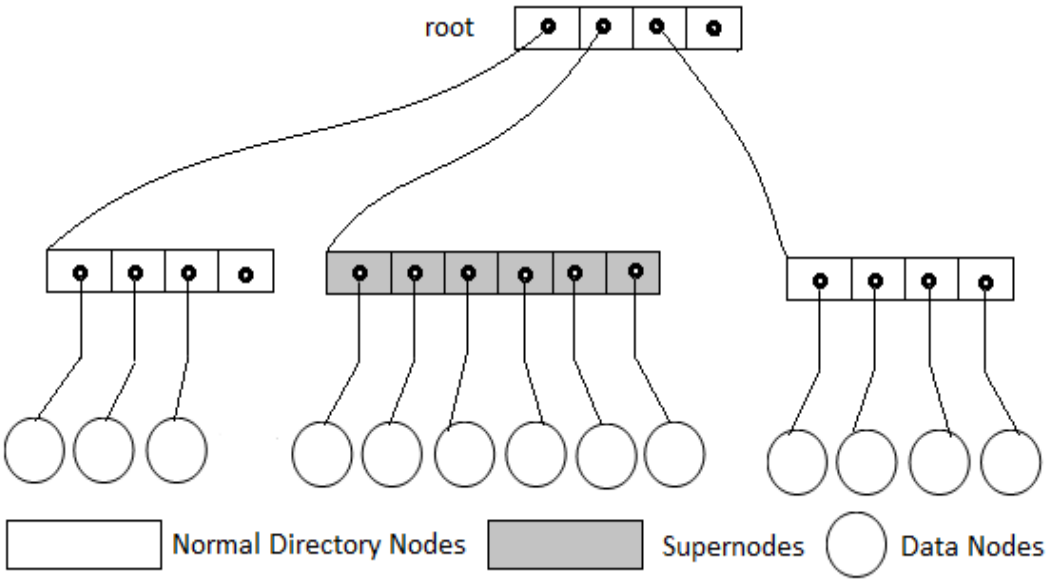

Figure 3: Structure of the X-tree

The block sizes for X-tree are different from those for R-tree and X-tree contains larger nodes when it is actually necessary. In Figure 4 [7], there are three examples of X-tree with having data in different dimensions. For X-trees, when the dimension increases the number and size of supernodes also increases and the height of X-tree which indicates the number of page accesses for point queries decreases.

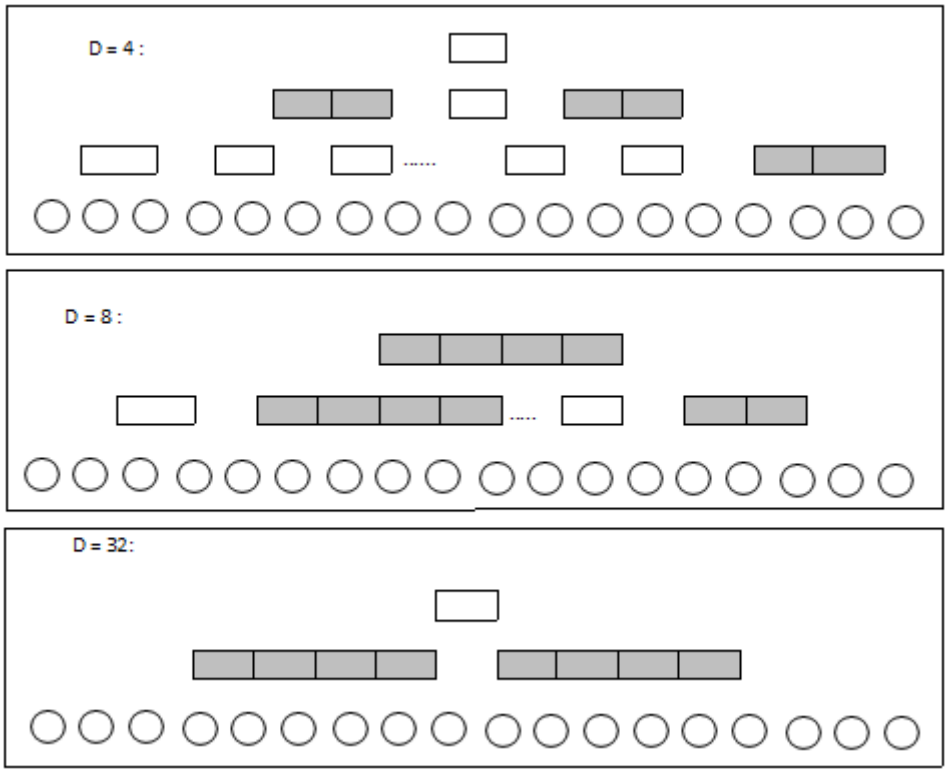

Figure 4: Various Shapes of the X-tree in different dimensions

In an X-tree, creating supernodes is mostly avoided, but when it is not possible to avoid overlap they are created. For many cases, this creation can be avoided by choosing an overlap-minimal split axis. If there is enough memory available, supernodes are kept in main memory [3]. Otherwise, the nodes to be replaces are selected by using a priority function and the following priority function [7] can be a proper one according to the experience.

$$c_t.type + c_l.level + c_s.size \text{ with } c_t \gg c_l \gg c_s \quad (3)$$

The storage utilization for uniformly distributed data is about 66% and this ratio is higher for supernodes [7]. It can be calculated for supernodes of size m.BlockSize with the following two extreme cases:

Assuming a certain amount of data occupies $X.m$ blocks for a maximally filled node. Then the same amount of data requires $X.\frac{m^2}{m-1}$ blocks when using a minimally filled node [7]. On the average, a supernode storing the same amount of data requires $\frac{\left(X.m+X.\frac{m^2}{m-1}\right)}{2} = X\left(\frac{m(2m-1)}{2m-2}\right)$ blocks. From that, it is obtained a storage utilization of $m/\left(\frac{m(2m-1)}{2m-2}\right) = \frac{2m-1}{2m-2}$ which for large m is considerably higher than 66%. For $m = 5$, for example, the storage utilization is about 88%.

In Figure 4 [7], it is shown 3 different shapes of X-tree. There are two extreme cases of the X-tree:

- None of the directory nodes is a supernode
- The directory consists of only one large supernode (root)

X-tree has two special cases and first one is that none of the directory nodes is a supernode. Here, the directory organization of the X-tree is completely hierarchical and it is similar to an R-tree. The height and size of the directory basically correspond to that of an R-tree [6]. Low dimensional and non-overlapping data may cause such an extreme case.

In the second case, X-tree is basically one root-supernode containing the lowest directory level of the corresponding R-tree. High dimensional or highly overlapping data can cause this situation and the size of the directory linearly depends on the dimension [3].

$$DirSize(D) = \frac{DatabaseSize}{BlockSize\ .StorageUtil}\ .2\ .BytesFloat\ .D \qquad (4)$$

3.1.2 Algorithms for the X-tree

In this section we explained the algorithms for the X-tree. Insertion, split, query, delete and update algorithms were represented below.

1. Insertion Algorithm: The main goal of this algorithm is to avoid splits producing overlap. The algorithm is given in Figure 5 [7] and its steps can be summarized as in the following way;

- The algorithm firstly determines the MBR and recursively calls the insertion algorithm.
- If there is no split in the recursive insert, only the size of corresponding MBRs is updated. Otherwise, a new MBR is added to the current node and the current node calls the split algorithm which is presented in Figure 6 [7].

```
int X_DirectoryNode::insert(DataObject obj, X_Node **new_node) {
   SET_OF_MBR *s1, *s2;
   X_Node *follow, *new_son;
   int return_value;
   follow = choose_subtree(obj); // choose a son node to insert obj into
   return_value = follow->insert(obj, &new_son); // insert obj into subtree
   update_mbr(follow->calc_mbr()); // update MBR of old son node
   if (return_value == SPLIT){
          // insert mbr of new son node into current node
         add_mbr(new_son->calc_mbr());
          if (num_of_mbrs() > CAPACITY) { // overflow occurs
                   if (split(mbrs, s1, s2) == TRUE) {
                     // topological or overlap-minimal split was successfull
                     set_mbrs(s1);
                     *new_node = new X_DirectoryNode(s2);
                      return SPLIT;
                    }
                    else { // there is no good split
                      *new_node = new X_SuperNode();
                      (*new_node)->set_mbrs(mbrs);
                      return SUPERNODE;
                    }
              }
     } // node 'follow' becomes a supernode
     else if (return_value == SUPERNODE) {
                   remove_son(follow);
                   insert_son(new_son);
     }
     return NO_SPLIT;
}
```

Figure 5: X-tree Insertion Algorithm for Directory Nodes

2. Split Algorithm: Split algorithm first tries to find a split of the node based on the topological and geometric properties of the MBRs. If the topological split however results in high overlap, the split algorithm tries next to find an overlap-minimal split [3].

- Partitioning MBRs may result in under filled nodes and this causes a degeneration of the tree. Also this is not good for the space utilization.
- If the number of MBRs in one of the partitions is below a given threshold, the split algorithm terminates without a split.

- In this situation, the current node becomes a supernode and it is extended to twice of the standard block size. If this occurs for a node which is already a supernode, it is extended by one additional block [7].
- If there is not enough space on disk to store the newly created or extended supernode, disk manager will perform a local reorganization. This will be necessary only when writing back the supernodes on secondary storage. However, this is a rare operation.
- The only condition that an overlap in the X-tree directory may occur is an overlap caused by the topological split below a threshold value which can be shown as MAX_OVERLAP [7]. The maximum overlap value can be estimated as in the following formula :

$$Max0.2.(T_{IO} + T_{Tr} + T_{CPU}) + (1 - Max0).(T_{IO} + T_{Tr} + T_{CPU}) = T_{IO} + 2.(T_{Tr} + T_{CPU})$$

$$=> Max0 = \frac{TTr + TCPU}{TIO + TTr + TCPU} \quad (5)$$

T_{I0} = Page access time,

T_{Tr} = The time to transfer a block from disk into main memory,

T_{CPU} = CPU time necessary to process a block

MIN_FANOUT is the usual minimum fan-out value of a node. Its values may be between 35% and 45% appropriately.

```
bool X_DirectoryNode::split(SET_OF_MBR *in, SET_OF_MBR *out1,
                            SET_OF_MBR *out2) {
        SET_OF_MBR t1, t2;
        MBR r1, r2;
        //first try topological split, resulting in two sets of MBRs t1 and t2
        topological_split(in, t1, t2);
        r1 = t1->calc_mbr(); r2 = t2->calc_mbr();
        // test for overlap
        if (overlap(r1, r2) > MAX_OVERLAP)
        {
            // topological split fails -> try overlap minimal split
            overlap_minimal_split(in, t1, t2);
            // test for unbalanced nodes
          if (t1->num_of_mbrs() < MIN_FANOUT ||
                t2->num_of_mbrs() < MIN_FANOUT)
            // overlap-minimal split also fails
            // (-> caller has to create supernode)
            return FALSE;
          }
         *out1 = t1; *out2 = t2;
         return TRUE;
}
```

Figure 6: X-tree Split Algorithm for Directory Nodes

3. Query Algorithms: This algorithm is about searching MBR object in X-tree. Query algorithm starts from root of X-tree and searches relevant MBR object in normal directory nodes. At the end, query algorithm returns a result set of MBR objects which are included by the parameter MBR object. Point, range and nearest neighbor queries can be given as example of X-tree queries.

4. Delete-Update Operations: Delete and update operations on X-tree may change the structure of tree. If the supernode consists of two blocks, it is converted to a normal directory node. Otherwise, that is if the supernode consists of more than two blocks, it is reduced the size of the supernode by one block [3].

3.2 XXL (eXtensible and fleXible Library) API

In this section, we gave detailed information about XXL API that is one of the vital parts of our thesis work. The core structure of the X-tree is particularly expressed.

3.2.1 Definition of XXL API

XXL (extensible and flexible Library) is a Java library providing advanced query processing functionality and also low-level components such as accessing to raw disks and high-level components such as a query optimizer. Its main features can be given as in the following way;

- Easy to use
- High-level
- Platform independent

XXL has a demand-driven cursor algebra which can be considered as a framework for indexing. When we use cursor in querying, we create a demand-driven cursor. At this point we only have a cursor object which points to root node of X-tree. And when we want to take an object from cursor, search algorithm runs and finds wanted object and returns it. For this reason cursor is demand-driven [17].

XXL is freely available under the terms of the GNU Lesser General Public License. The components of this library can be listed as in the following way;

1. The cursor package has algebra of the most important query operators. These operators have demand-driven implementations and it is required by the operators that both input and output satisfy an iterator interface. XXL also contains an algebra based on Java's ResultSet Interface to support the import of external database sources.

2. A rich infrastructure of external data structures is included in XXL and this enables you to implement new database functionality. Also, there is a very flexible buffer mechanism in XXL and you can use this with no need to know the other parts of XXL.
3. XXL has default implementations such as R-tree, M-tree, X-tree and also it is easy to implement new structures in XXL. All of these implementations exist as a framework in XXL and it is fully embedded into the library.

XXL is proposed as a library for conducting experiments. The results of experiments are used for evaluating the new query processing techniques' performance. With the idea of improving the quality of experimental work, it is thought to publish the results and the code. The code is considered to be poor and so XXL would be an ideal infrastructure for experiments as it is a well-documented and open source library [8, 9, 10].

The reasons behind the design and implementation of XXL can be given as in the following way:

- If platforms, programming languages or compilers are not same, it becomes difficult to compare two access methods. As a result, the number of I/Os becomes the only criterion for the comparison. Therefore, XXL should support experimental evaluations in standard forms.
- One of the aims can be given as that XXL could serve as a repository for algorithms. It can be considered as a good study to collect the implementations of existing algorithms under a framework structure.
- Many of the existing implementations of query processing algorithms have an ad-hoc manner and they have poor software designs. Also there are limitations about specific operating systems and they have incomplete or in available documentations.

When the XXL project started eleven years ago, the first choice for the platform to develop was Java. The functionalities of other Java libraries such as API (Application Programming Interface) of the SDK (Software Development Kit) [11]

and Colt [12], benefits of using design patterns like factory, iterator and decorator leads the developers to choose Java [13].

3.2.2 Basic Components

3.2.2.1 Functions and Predicates

As a general principle, functions are remarkably important to encapsulate abstractions but this concept has received little attention in database community especially due to the implementation problems of query processing algorithms [14].

XXL has an abstract class named as Function and you can implement a new function by defining a sub-class of Function. To make the number of explicit classes smaller Function's sub-classes are generally implemented in the form of anonymous classes. In Java, if the occurrence of the implementation of a sub-class and creation of an object of the class are in the same place of the code, this can be considered as a specific concept. Here, after the method of the object is invoked the evaluation of the target function occurs eventually.

With the help of compose function in the Function class, new functions as compositions of other functions could be created at runtime. This can be provided easily with the help of anonymous classes in Java. In XXL, this feature is used, for instance, to implement aggregate functions incrementally.

Function class is implemented by a functional class in XXL and a new functional object can be declared in the following ways;

1. Function class is extended and an anonymous class is implemented. Here, invoke method containing the executable code is overridden.
2. Composition of functional objects can call the compose method of a functional object.

When the invoke method is called with the appropriate parameters, the code of a functional object is executed and in XXL functional objects may have a status unlike the pure mathematical functions.

3.2.2.2 Containers

A container can be defined as an implementation of a map providing an abstraction from the physical storage. An object is inserted into a container with a new ID and after that this object can be retrieved with this generated ID. Containers include mechanisms for buffer management to act as bridge between levels of a storage hierarchy.

The container implementations of XXL can be given as in the following way;

- *MapContainer*: In MapContainer, the set of objects is kept in main memory and with the help of this container queries can be run fast in memory and also debugging is supported.
- *BlockFileContainer*: Here, block is an array of bytes having a fixed length and BlockFileContainer is a file of blocks. This container is useful for the implementation of index-structures such as R-trees.
- *ConverterContainer*: It is a decorated container which is designed to overcome the deficiencies of inappropriate serialization mechanism of Java.
- *BufferedContainer*: It is another decorator to support object buffering in XXL.

XXL supports accesses to raw devices with the help of some Java classes. Therefore, it is enabled to run experiments on external storage without any interaction with the underlying operating systems. This can be operated in two ways as in the following;

- Native methods provided by the *NativeRawAccess* class can be used.
- An implementation of an entire file system running on a raw device which is provided by XXL can be used.

The abstract class Container is specifically designed to provide buffered access to objects. Thus, access methods have a flag to fix and unfix the accessed object to be used in the insertion and the removal of the objects.

The package xxl.collections contains a few of the implementations of the Container interface such as Map-Container. Many of the containers like BlockFileContainer and BufferedContainer are designed for external data management.

3.2.2.3 Cursors

A cursor can be considered as an abstract mechanism and it is used to access objects within a stream. In XXL, cursors are not dependent to specific type of underlying objects. A cursor's interface that shown in Figure 7 can be given as in the following way;

```
interface Cursor extends java.util.Iterator {
        Object peek();
        void update(Object o);
        void reset();
        void close();
}
```

Figure 7: Interface of cursor

The functions of a cursor are similar to the functions of the iterator placed into the java.util package. These functions can be given as in the following way;

- *Peek*: Reports the next object of the iteration
- *Reset*: Sets the cursor the beginning of the iteration

- *Close*: Stops the iteration and releases resources
- *Update*: Updates the current object of the iteration

There is algebra in XXL for processing cursors and this algebra contains operations which get cursors input and produce outputs as cursors.

The types of cursors can be given as in the following way;

- *Input cursors*: These cursors are used to transform a data source into a cursor.
- *Processing cursors*: They modify the input cursor.
- *Flow cursors*: They change the underlying data flow without changing the objects within the input stream.

3.2.3 Query Processing

3.2.3.1 Indexing

The package xxl.indexStructures is a framework containing tree-based index structures and it has a skeleton implementation for so-called grow-and-post trees [17] such as R-trees, B-trees and X-trees. Also, this package facilitates implementing new index-structures.

An example for using an index-structure like an M-tree [15] can be given as in the following way;

```
MTree mTree = new MTree(MTree.HYPERPLANE_SPLIT);
mTree.initialize(getDescriptor, container, minCap, maxCap);
```

Here, first of all a constructor is called and in this example the constructor has single parameter which is used to specify the split strategy. After construction, M-tree is initialized and here the first parameter shows a functional object that computes a so-

called descriptor for a given data item. The second parameter is responsible for managing the nodes of the tree and the last two parameters specify the minimum and maximum number of items within a node.

In order to implement a new index-structure, firstly the framework which is the implementation of grow-and-post trees [16] must be understood. The inner class Node determines the index-structure and coding a specialized class for nodes can be considered as the basic part of implementing a new index-structure.

3.2.3.2 The Lower Interface of Index Structures

The data of tree-based index structures are kept in nodes referring to blocks having a fixed size. A user may implement own container class to manage these blocks. A sample usage of containers is illustrated in Figure 8 [3] and top container is a BufferedContainer where a large number of nodes are kept in buffer.

If the requested node does not exist in the buffer, the request for the node is passed to the ConverterContainer and this container converts a node into its block and vice versa. After that, the request is redirected to a FileContainer object which reads the desired block from disk.

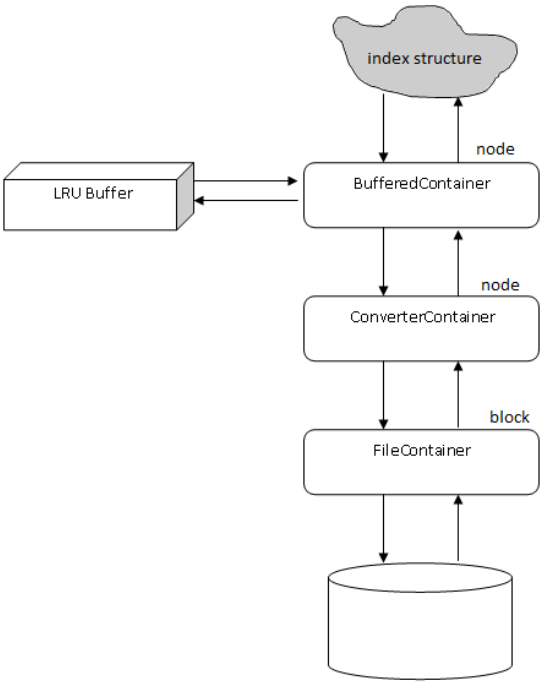

Figure 8: An example for using containers

3.2.3.3 Join Processing

It can be said that the most important operations in a database system are joins and new join types are required for new applications like spatial databases. XXL join processing is a framework designed to obtain a single implementation to support different kinds of join operations efficiently. Also, XXL is enabled to cover sort-merge and hash-based joins.

While processing join operations a sweep-area that is a small subset is kept for each input source and input's elements are inserted into the related sweep-area. After that, the other sweep-area is checked for join partners and to remove the elements not producing join results anymore a sweep-area is reorganized periodically.

The most important functions of the interface SweetArea as shown in Figure 9 can be given as in the following way;

```
public interface SweepArea {
        public void insert(Object o);
        public void reorganize(Object curStatus, int id);
        public Iterator query(Object o);
        ...
}
```

Figure 9: Interface of SweepArea

Here, each of the inputs has a unique identifier and the operations show the basic steps of join processing.

In XXL a join can be called by the following statement;

```
Iterator it = new Join (input1, input2,
                HashBagSweepArea.FACTORY_METHOD,
                Tuplify.DEFAULT_INSTANCE);
```

Here, two input sources are specified by the first two parameters and the third parameter indicates a factory method in order to generate a sweep-area which is organized as a hash-table [18]. The last parameter is a functional object describing how to construct the output tuple of the join. For a user to implement a new join type, it is necessary to implement an appropriate class satisfying the interface SweepArea.

3.2.3.4 Aggregation

Aggregate operations are important in large database systems to deliver a quick overview of the response set. In contrast to a relational DBMS, XXL supports functions as results of aggregate operations. This allows returning a histogram or

other more advanced statistical data structures directly to the user (without producing an intermediate relation). In the following, it is briefly described the basic structures of XXL package statistics.

This package is based on a generic aggregator cursor that applies user-defined functions to aggregate the objects of a given iterator. This cursor returns the intermediate value of the aggregate among the input that has been consumed so far. The final value can be reported by a call to aggregator.last(), which consumes the entire iterator. An example of such an aggregator as shown in Figure 10 is given below:

```
Aggregator aggregator = new Aggregator(

        new RandomIntegers(100, 50),
              new Function () { // the aggregation function
                   public Object invoke (Object agg, Object next) {
                     return (agg == null) ? next : maxComp.invoke(agg, next);
                   }
             }
);
```

Figure 10: Definition of Aggregator

3.2.4 Summary of XXL API

XXL is a query processing library implemented in Java that includes the most important ingredients for efficient query processing. In addition XXL, a well documented Java library, suitable for rapid implementation of advanced query processing techniques. The software is freely available under the terms of the GNU Lesser General Public License. The design of the library was determined by two goals: the functionality of XXL should be extended easily and XXL should be flexible enough for being customized fast to specific problems.

Key components of XXL are powerful cursor algebra, a framework for a broad class of index structures and a toolbox of I/O data structures. Due to its powerful methods, XXL is also an excellent platform for experimental work. Coding of new algorithms and data structures requires substantial less time than beginning from scratch. XXL is not in competition to the popular Java API, but it provides seamless enhancements.

3.3 Fuzzy C-Means Clustering

In our thesis work, FCM algorithm is used for generating fuzzy membership value of meteorological attributes. Algorithm is applied for each attribute in meteorological attributes. For this purpose an adaptor structure is developed to handle getting fuzzy values from given input value of each meteorological attribute. After we calculated the fuzzy membership value of meteorological attribute, we inserted this calculated value to fuzzy secondary index.

3.3.1 About FCM

Fuzzy C-Means (FCM) is a method of clustering which allows one piece of data to belong to two or more clusters. FCM clustering was first reported in the literature for a special case (m=2) by Joe Dunn in 1974 [20]. The general case (for any m greater than 1) was developed by Jim Bezdek in his PhD thesis at Cornell University in 1973 [21].

In FCM clustering, each point has a degree of belonging to clusters, as in fuzzy logic, rather than belonging completely to just one cluster. Thus, points on the edge of a cluster may be in the cluster to a lesser degree than points in the center of cluster [20].

FCM clustering processes n vectors in p-space as data input, and uses them, in conjunction with first order necessary conditions for minimizing the FCM objective functional, to obtain estimates for two sets of unknowns.

The unknowns in FCM clustering are:

- A fuzzy c-partition of the data, which is a c*n membership matrix $U=[u(i_k)]$ with c rows and n columns. The values in row i give the membership of all n input data in cluster i for $k=1$ to n; the k-th column of U gives the membership of vector k (which represents some object k) in all c clusters for $i=1$ to c. Each of the entries in U lies in [0,1] ; each row sum is greater than zero; and each column sum equals 1.
- The other set of unknowns in the original FCM model is a set of c cluster centers or prototypes, arrayed as the c columns of a p*c matrix V. These prototypes are vectors (points) in the input space of p-tuples. Pairs (U, V) of coupled estimates are found by alternating optimization through the first order necessary conditions for U and V. The objective function minimized in the original version measured distances between data points and prototypes in any inner product norm, and memberships were weighted with an exponent $m > 1$ [21].

3.3.2 The Algorithm of FCM

FCM is based on minimization of the following objective function:

$$J_m = \sum_{i=1}^{N}\sum_{j=1}^{C} u_{ij}^{m}\left\|x_i - c_j\right\|^2, 1 \le m < \infty \qquad (6)$$

where m is any real number greater than 1, u_{ij} is the degree of membership of x_i in the cluster j, x_i is the ith of d-dimensional measured data, c_j is the d-dimension center of the cluster, and ||*|| is any norm expressing the similarity between any measured data and the center [20].

Fuzzy partitioning is carried out through an iterative optimization of the objective function shown above, with the update of membership u_{ij} and the cluster centers c_j by:

$$u_{ij} = \frac{1}{\sum_{k=1}^{c} \left(\frac{\|x_i - c_j\|}{\|x_i - c_k\|} \right)^{\frac{2}{m-1}}} \tag{7}$$

$$c_j = \frac{\sum_{i=1}^{N} u_{ij}^m . x_i}{\sum_{i=1}^{N} u_{ij}^m} \tag{8}$$

This iteration will stop when $max_{ij}\{|u_{ij}^{(k+1)} - u_{ij}^{(k)}|\} < \delta$, where δ is a termination criterion between 0 and 1, whereas k are the iteration steps. This procedure converges to a local minimum or a saddle point of J_m [21].

The algorithm is composed of the following steps:

1. Initialize $U=[u_{ij}]$ matrix, $U^{(0)}$
2. At k-step: calculate the centers vectors $C^{(k)}=[c_j]$ with $U^{(k)}$

$$c_j = \frac{\sum_{i=1}^{N} u_{ij}^m . x_i}{\sum_{i=1}^{N} u_{ij}^m} \tag{9}$$

3. Update $U^{(k)}$, $U^{(k+1)}$

$$u_{ij} = \frac{1}{\sum_{k=1}^{c} \left(\frac{\|x_i - c_j\|}{\|x_i - c_k\|} \right)^{\frac{2}{m-1}}} \tag{10}$$

4. If $\|U^{(k+1)} - U^{(k)}\| < \delta$ then STOP; otherwise return to step 2.

And some more clear explanation can be shown as in below steps. For each point x we have a coefficient giving the degree of being in the kth cluster $u_k(x)$. Usually, the sum of those coefficients for any given x is defined to be 1:

$$\forall x \left(\sum_{k=1}^{num.\ cluster} u_k(x) = 1 \right) \tag{11}$$

With fuzzy c-means, the centroid of a cluster is the mean of all points, weighted by their degree of belonging to the cluster:

$$center_k = \frac{\sum_x u_k(x)^m x}{\sum_x u_k(x)^m} \tag{12}$$

The degree of belonging is related to the inverse of the distance to the cluster center:

$$u_k(x) = \frac{1}{d(center_k, x)} \tag{13}$$

Then the coefficients are normalized and fuzzified with a real parameter $m > 1$ so that their sum is 1 [20]. So,

$$u_k(x) = \frac{1}{\sum_j \left(\frac{d(center_k, x)}{d(center_j, x)}\right)^{\frac{2}{(m-1)}}} \tag{14}$$

For m equal to 2, this is equivalent to normalizing the coefficient linearly to make their sum 1. When m is close to 1, then cluster center closest to the point is given much more weight than the others, and the algorithm is similar to k-means [21].

CHAPTER 4

IMPLEMENTATION OF X-TREE STRUCTURES

In this section, we explained spending efforts to develop X-tree with new specialties. We described step by step how to build X-tree node that contains both 3D spatial data and fuzzy data. Then we represented the structure of uncoupled index. For these purposes, some part of implemented code and figure of structure are given to provide explanatory expression.

In performance tests section, we prepared class of queries for test scenarios about coupled and uncoupled index structures. And we applied on both indexes and observed results as table. We wrote our deductions according to the test results.

4.1 General Overview

In this section, efforts and studies on developing new specialties of X-tree are explained in details. Our main goals in this thesis can be summarized as follow;

- Providing 3D spatial primary indexing on X-tree by using point coordinates
- Supporting non-spatial fuzzy secondary indexing on X-tree by using meteorological attributes to create coupled index structure
- Building uncoupled index structure that handles 3D primary index and fuzzy secondary index separately
- Comparing the performance of coupled and uncoupled index structure

First of all, implementation details about development of XXL API by using meteorological attributes are given step by step in order to explain the details of studies.

After that, the development of 3D spatial primary index is explained particularly. Also, implementation of X-tree index structure is given and operations on X-tree structure such as making tree-traversals on X-tree, showing the structure of created index as rectangle boxes view and using X-tree index in querying are focused on.

4.2 Handling 3D Rectangle in X-tree Node Structure

In this work, it is planned to have a node containing a 3D rectangle and meteorological attributes having values in a specified [min, max] range. Our X-tree is designed to have nodes containing both 3D rectangle data for spatial index and meteorological attributes for fuzzy secondary index.

Our node structure can be given as in the following way:

```
Node {
        Rectangle [coor: x, coor:y, coor:z],

        Meteorological_attributes [hot, cold, snowy, ...]
}
```

Figure 11: New Node Structure of X-tree

Here, the meteorological attributes such as temperature, wind speed humidity and pressure have numerical values in [min, max] interval. These values are allocated in map structure to enable dynamical access.

In spatial side, we wanted to create a primary index with our data-points. For this purpose firstly we created MBR objects by using data-points, and then we inserted

them to the X-tree. Creating MBR object by using data-points was needed a converting operation. For conversion we made modification on converter functions. Before the modification, converter functions are shown in Figure 12 in the following code:

```
Converter dataConverter = new ConvertableConverter(
                                    new Function(){
                                        public DoublePoint invoke(){
                                          return new DoublePoint(dim);
                                        }
                                    });

Converter idConverter = new ConvertableConverter(
                                    new Function(){
                                        public DoublePointRectangle invoke(){
                                          return new DoublePointRectangle(dim);
                                        }
                                    });
```

Figure 12: Previous Structure of Converter

The converter in Figure 12 was used before inserting data to the X-tree. It is usage is demonstrated in Figure 13;

```
for(...) { //reading data from any source, such as input file
   double data[] = ...; //read coordinates as double
   //create point object that contains double type point
   DoublePoint p = new DoublePoint(data);
   //create MBR object by using point data
   DoublePointRectangle mbr = new DoublePointRectangle(p,p);
   //KPE object can be inserted to the X-tree so convension is needed

   KPE k = new KPE(new Object[]{data, id},
                        new Converter[]{dataConverter, idConverter});

   tree.insert(k); //insert the KPE data
}
```

Figure 13: Usage of Previous Structure of Converter in Insertion

However the code in Figure 13 is not applicable for our insertion operation because the converter was not suitable for our 3D rectangular data. We adapted converter functions to support 3D spatial data insertion. We implemented abstract functions for conversion. The modified structure of converter functions code is shown in Figure 14;

```
Converter dataConverter = new ConvertableConverter(
        new AbstractFunction<Object, DoublePoint>() {
            public DoublePoint invoke(){
                return new DoublePoint(DIMENSION_OF_X_TREE);
            }
});

Converter idConverter = new ConvertableConverter(
        new AbstractFunction<Object, DoublePointRectangle>() {
             public DoublePointRectangle invoke(){
                 return new DoublePointRectangle(DIMENSION_OF_X_TREE);
        }
});
```

Figure 14: New Structure of Converter

After we modified converter functions we created MBR objects that has 3 dimensional rectangle by using data-points and inserted them to the X-tree successfully.

4.3 Obtaining Supernode

As we mentioned in section 3, the X-tree consists of three different types of nodes: data nodes, normal directory nodes, and supernodes.

In here supernodes are large directory nodes of variable size. The aim of using supernodes in the X-tree is to avoid splits in the directory that would result in an inefficient directory structure [3].

In insertion steps if there is no other possibility to avoid overlap, supernodes are created during insertion only. After we created X-tree by using different number of records, we have two different cases:

- When there is not any supernode among the directory nodes
- When there is at least one supernode among the directory nodes

In the first case as shown in the Figure 15, the X-tree has a completely hierarchical organization of the directory and is therefore similar to an R-tree. This case may occur for low dimensional and non-overlapping data.

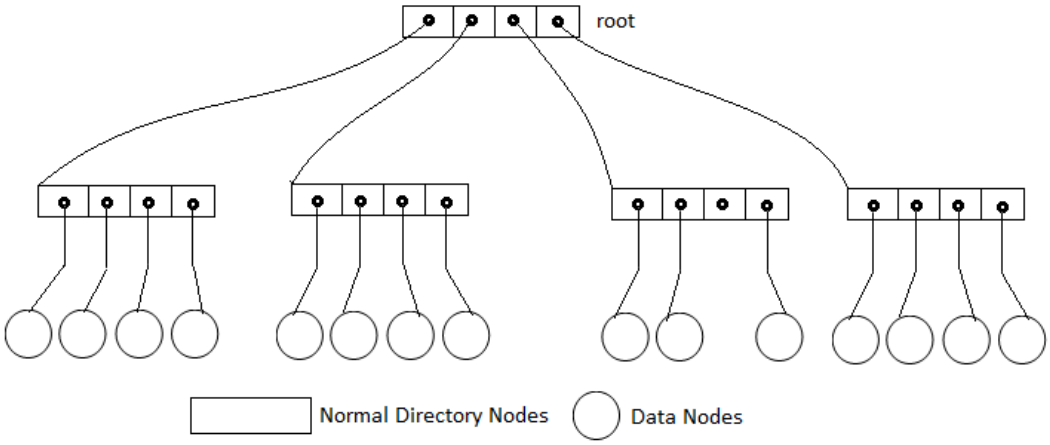

Figure 15: X-tree that has only normal directory nodes

In the second case as shown in the Figure 16, the directory of the X-tree has a supernode so that the performance corresponds to the performance of a linear directory scan according to the number of supernode.

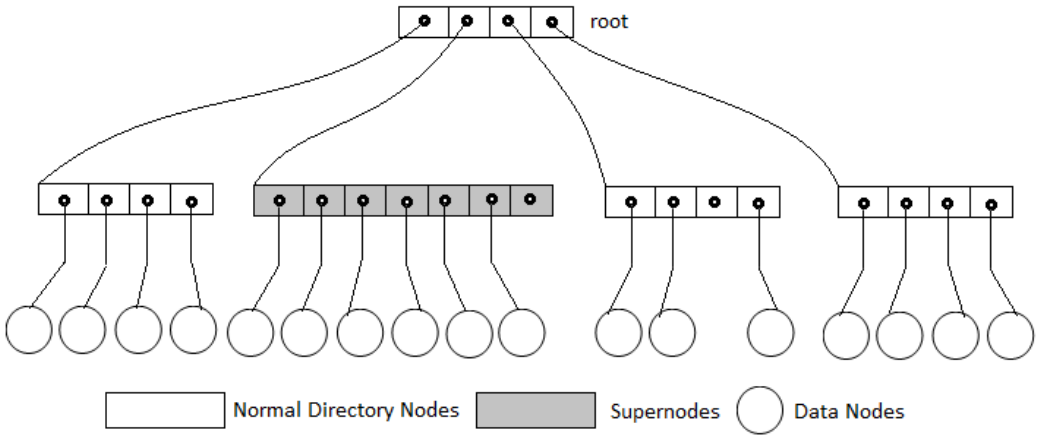

Figure 16: X-tree that has both normal directory nodes and supernode

In our work, we wanted to observe the both special cases. So that, firstly we tried to create an X-tree structure which has no supernode. Then we attempted to create X-tree structure having at least one supernode among the directory. We made modification on our API to generate supernode. In this operation, we created *minMaxFactor* value in API source that is a factor where the minimum capacity of nodes was smaller than the maximum capacity. It means *minMaxFactor* is the quotient between minimum and maximum number of entries in a node, e.g. 0.5.

We set *minMaxFactor* of the X-tree when the time of X-tree initializing. In insertion operation if split is necessary, *minMaxFactor* value is considered. Therefore by changing this value of X-tree, we could generate supernode easily.

4.4 Drawing Tree View of X-trees

Drawing the structure of X-tree as topological view is not a major issue of our work but it is useful for us to show the proof of concept. For this purpose the topological view of the X-tree was drawn. We developed a recursive algorithm to draw the tree view. Algorithm starts from the root node and draws each node of X-tree when the cursor on it and moves the cursor to the child node from the current parent node. The pseudecode can be given as in the following way:

- if current node level is greater than zero
    - draw the rectangle of the current node and calculate the position of the drawer cursor on the screen
    - get all the child node of the current node
        - call this function for each child node of the current node
- if current node level is equal to zero, this is leaf node
    - draw the rectangle of this leaf node and calculate the position of the drawer cursor on the screen

Figure 17: Drawing algorithm for X-tree

We started this function by giving root node as a parameter. After the execution is completed, output of the execution is shown in Figure 18.

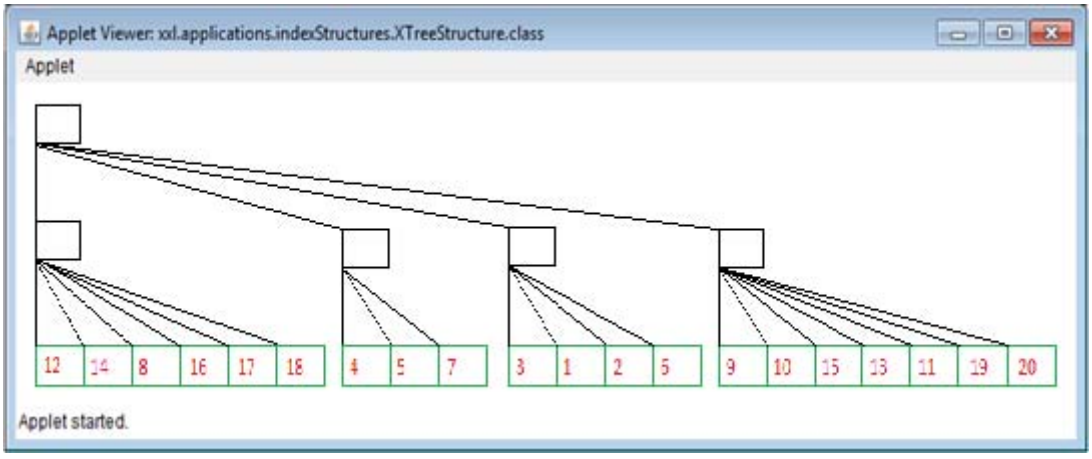

Figure 18: Tree view of an X-tree

4.5 Rectangular View of X-trees

For proof of concept and also to show that X-tree index structure had been successfully built, we also implemented another drawing method. We showed the node elements of the X-tree as rectangular view. In this view, it can be seen that parent node covers its children nodes, as shown in Figure 19. This appearance claims

that the X-tree structure was correctly created. The outermost rectangle is the root node of the X-tree and the innermost rectangles are the data elements that are in leaf nodes. Their id numbers can also be seen in the view.

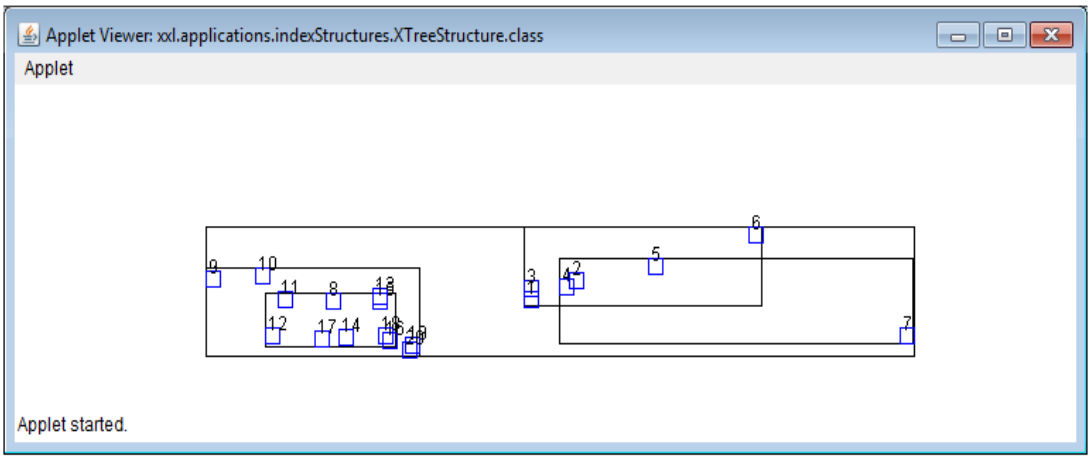

Figure 19: Rectangular view of an X-tree

We drew the X-tree structure by coloring the rectangles according to their levels to show the structure more clearly. In Figure 20, it is shown that the nodes, which are in the same level, are colored by the same color.

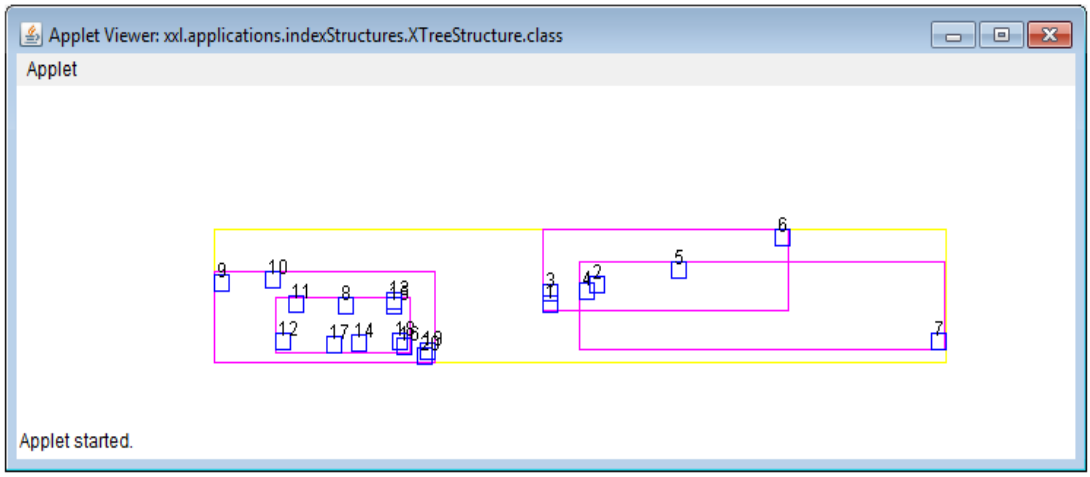

Figure 20: Colored rectangular view of an X-tree

4.6 Overlaying Secondary Index to the X-tree Structure

In this section, we gave detailed information about overlaying operation on building secondary index. In coupled index structure, we handled primary and secondary indexes in monolithic structure. For this purpose, we firstly built 3D spatial primary index by using X-tree structure, then we overlaid fuzzy secondary index over created X-tree structure.

We have three major parts of overlaying operation. These are allocation 3D spatial data and fuzzy data in node, applying FCM algorithm on meteorological data to calculate fuzzy membership value for fuzzy secondary index and traversing over X-tree primary index to build secondary index.

4.6.1 Allocating Meteorological Attributes

The base implementation of X-tree structure done by XXL API only contains rectangular object in creating X-tree nodes. As in the structure of X-tree which is presented in Figure 3, the X-tree has three different types of nodes: data nodes, normal directory nodes, and supernodes. The data nodes of the X-tree contain rectilinear minimum bounding rectangles (MBRs) together with pointers to the actual data objects, and the directory nodes contain MBRs together with pointers to sub-MBRs.

According to the aim of our thesis work, we developed X-tree nodes containing both MBRs and meteorological attributes. For this purpose, we made some changes on the MBRs object that allocated these attributes. Not only MBRs allocate these attributes but also directory nodes and supernodes should allocate them. Therefore secondary index on X-tree could be achieved by using meteorological attributes on the tree structure.

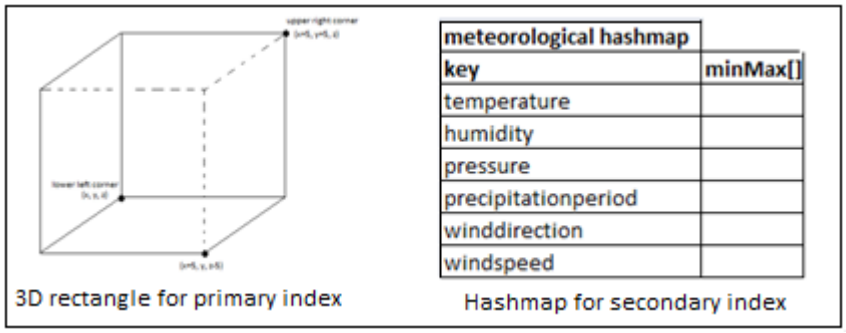

Figure 21: 3D rectangle for spatial index and meteorological attribute for secondary index

In detail, X-tree node structure was changed with a new hash map which was used for meteorological attributes. The structure of this hash map containing a key for each attribute and two variables for max-min values of each attributes is shown in Figure 21. By this definition, lots of meteorological attributes can be allocated in the nodes of the X-tree. A meteorological attribute in the data node, has only one value. Normally in the data node max-min value has no meaning and it has only value that points its meteorological value about meteorological attribute.

To allocate both spatial data and meteorological data, we need a specialized Descriptor-class. We used this class for allocating data in the node and getting data from node. The Descriptor in the X-tree is just n-dimensional DoublePointRectangle. We used 3-dimesional DoublePointRectangle class to store both spatial data and meteorological attributes as seen in Figure 22. Note that this is a simplified view of the class and it is not the actual class implementation.

```
public static class DoublePointRectangle implements Rectangle {
   // default definitions in here
   .....
   // allocates <temperature, humidity,
   // pressure, ..., windspeed> attributes
   protected HashMap<String, double[]>
         meteorologicalAttributes = new HashMap<String, double[]>();

  public DoublePointRectangle(double[] leftCorner,
         double[] rightCorner, HashMap meteorologicalAttributes)  {

         super(leftCorner, rightCorner);
         this.meteorologicalAttributes = meteorologicalAttributes;
  }

  //provide setter and getter methods for meteorologicalAttributes
  getMeteorologicalAttributes();
  setMeteorologicalAttributes(String attributeName, double values[]) ;
  // provide reasonable implementations...
  public boolean overlaps (Descriptor descriptor){}
  public boolean contains (Descriptor descriptor){}
  public void union (Descriptor descriptor){}
  public boolean equals (Object object){}
}
```

Figure 22: Basic Structure of DoublePointRectangle Class

After we adapted DoublePointRectangle class, we worked on insertion to X-tree. In insertion, when we were inserting new data to the X-tree, its structure was changing. In this operation some of nodes' position was changing in tree arrangement process, such as move node from one branch of tree to another branch according to the split algorithm. In arrangement operation node is read from object buffer or written to object buffer or cloned. We also adapted these operations to handle our new *DoublePointRectangle* class.

We adapted the *read* and *write* methods of DoublePointRectangle to make sure that meteorological attributes get serialized to disk properly.

The *read* method reads the state (the attributes) for an object of this class from the specified data input and restores the calling object. The *read* method, as shown in

Figure 23, must read the values in the same sequence and with the same types as were written by write.

```
public void read(DataInput dataInput) throws IOException {
        for(int i=0; i< leftCorner.length; i++)
                leftCorner[i] = dataInput.readDouble();
        for(int i=0; i< rightCorner.length; i++)
                rightCorner[i] = dataInput.readDouble();

        double [] meteorologicalAtt = new double[16];
        for(int i=0; i< meteorologicalAtt.length; i++)
                meteorologicalAtt[i] = dataInput.readDouble();
        addMeteorologicalAttributes(meteorologicalAtt);
}
```

Figure 23: Implemented read Method

And the *write* method, as shown in Figure 24, writes the state (the attributes) of the calling object to the specified data output. This method should serialize the state of this object without calling another write method in order to prevent recursions.

```
public void write(DataOutput dataOutput) throws IOException {
        for(int i=0; i< leftCorner.length; i++)
                dataOutput.writeDouble(leftCorner[i]);
        for(int i=0; i< rightCorner.length; i++)
                dataOutput.writeDouble(rightCorner[i]);

        Set<String> keys=meteorologicalAttributes.keySet();
        Iterator<String> it = keys.iterator();
        while(it.hasNext()) {
          String keyName = (String)it.next();
          double [] d = meteorologicalAttributes.get(keyName);
          dataOutput.writeDouble(d[0]);
          dataOutput.writeDouble(d[1]);
        }
}
```

Figure 24: Implemented write Method

The *addMeteorologicalAttributes* puts the meteorological attributes' values to the meteorological attribute hash map, as shown in Figure 25. This method is used in the insertion section of application after the 3D rectangle created. We firstly created *DoublePointRectangle* with spatial data, and then we add meteorological attributes.

```
private void addMeteorologicalAttributes(double[] meteorologicalAtt)  {
        double [] minMax = null;
        for(int i=0; i< meteorologicalAtt.length; ++i) {

                switch (i) {
                case 0://temporal
                        minMax = new double[2];
                        minMax[0]=meteorologicalAtt[i];
                        break;
                case 1://temporal
                        minMax[1]=meteorologicalAtt[i];
                        meteorologicalAttributes.put("tm", minMax);
                        break;
                .......
}
```

Figure 25: Implemented addMeteorologicalAttributes Method for meteorological attributes

The constructor of *DoublePointRectangle* class is used with given rectangle parameter and it creates a new *DoublePointRectangle* as a copy of the given rectangle as shown in Figure 26. This is generally used by copy operation on the node objects.

```
public DoublePointRectangle(Rectangle rectangle)  {
        DoublePointRectangle rect = (DoublePointRectangle)rectangle;
        leftCorner = new double[rect.leftCorner.length];
        rightCorner = new double[rect.rightCorner.length];
        System.arraycopy (rect.leftCorner, 0, leftCorner, 0,
        rect.leftCorner.length);
        System.arraycopy (rect.rightCorner, 0, rightCorner, 0,
        rect.rightCorner.length);
        meteorologicalAttributes = new HashMap<String, double[]>();

        copyMeteorologicalAttributes(rect.meteorologicalAttributes,
        meteorologicalAttributes);

}
```

Figure 26: Implemented Constructor of DoublePointRectangle

And the final method of this method group is *copyMeteorologicalAttributes* method which is shown in Figure 27 and copies meteorological attributes' values from source hash map to the destination hash map. It is useful for clone operation.

```
private void copyMeteorologicalAttributes( HashMap<String,
double[]>  source, HashMap<String, double[]> destination)  {
                if(source!=null && source.size()>0) {
                        Set<String> keys=source.keySet();
                        Iterator<String> it = keys.iterator();
                        while(it.hasNext()){
                                String keyName = (String)it.next();
                                double [] d = source.get(keyName);

                                double [] minMax = new double[2];
                                minMax[0]=d[0];
                                minMax[1]=d[1];

                                destination.put(keyName, minMax);
                        }
                }
}
```

Figure 27: Implemented copyMeteorologicalAttributes Method

Once we adapted the read/write method, constructor of *DoublePointRectangle* and developed new function for our purpose, we created X-tree containing both spatial and meteorological data in its nodes.

4.6.2 Implementing FCM Algorithm

In our work, FCM algorithm is used to generate fuzzy secondary index by using meteorological attributes. Algorithm is applied for each attribute in meteorological attributes. For this purpose an adapter structure is developed to handle getting fuzzy values from the given input value of each meteorological attribute. In detail, following steps were implemented;

- Choose the number of clusters, *numberOfCluster*. In this step clusters are chosen by using input values threshold. After reading records from input file, read values are sorted ascending order and clusters are defined on this sorted array.
- After generating *numberOfCluster* clusters, then determine the cluster centers. Sorted array that generated in the previous step is used for this aim and the threshold of elements in array determines the exact cluster centroids.
- Assign each point to the nearest cluster center, where "nearest" is defined with respect to one of the distance measures discussed above.
- Determine the fuzzy membership of each cluster centroid for each point.
- Repeat the two previous steps until some convergence criterion is met (usually that the assignment hasn't changed).

The main advantages of this algorithm are its simplicity and speed which allow it to run on large datasets.

In our work, firstly we read whole data from input file and then each meteorological attribute was put on a set. Then ascending ordering operation was applied to data in each set and then fixed centroids were determined for each set. For example

temperature values were ordered by ascending order than fixed centroids in the name of cold, warm and hot were determined by using the following formulas:

$$Centroid_{warm} = point_{min} + \frac{point_{max} - point_{min}}{2} \quad (15)$$

$$Centroid_{cold} = point_{min} + \frac{Centroid_{warm} - point_{min}}{2} \quad (16)$$

$$Centroid_{hot} = Centroid_{warm} + \frac{point_{max} - Centroid_{warm}}{2} \quad (17)$$

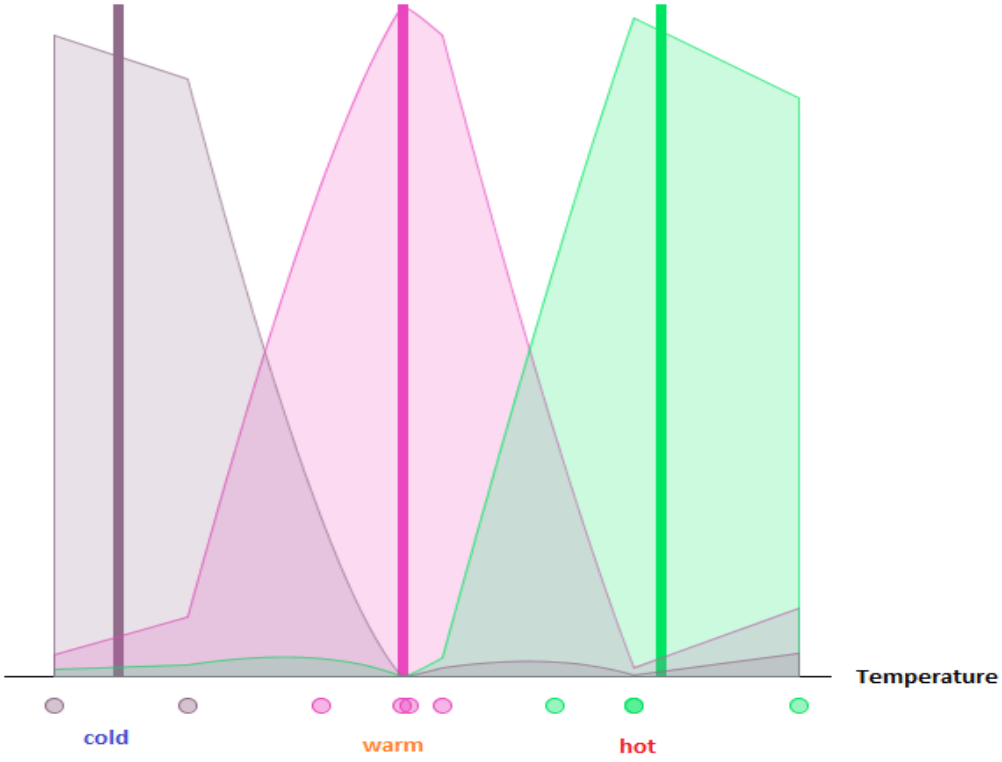

Figure 28: Fuzzy C-Means clustering

After determining of these three centroids, the algorithm of Fuzzy-C Means algorithm was applied as mentioned above.

When algorithm is completed, FCM algorithm generates fuzzy membership values for each centroid that is about the distance from centroid. We do not need distance of

fuzzy values of each centroid; we need membership of fuzzy values. Calculating membership from distance values is explained below.

Let a, b and c values are the fuzzy membership distance values of each centroid. At the beginning we have fuzzy membership [a, b, c] that are about distance. When the algorithm completed we get these [a, b, c] fuzzy membership of each values in set about the distance of each centroid as shown in Figure 28. At this point we applied reverse ratio formula as shown below.

$$\frac{x}{a} + \frac{x}{b} + \frac{x}{c} = 1 => x = \frac{(a.b.c)}{(a.b + b.c + a.c)} \tag{18}$$

And by using calculated *x* we can determine the fuzzy membership value for each as follows;

$$val_1 = \frac{x}{a}, val_2 = \frac{x}{b}, val_3 = \frac{x}{c} \tag{19}$$

Therefore we get fuzzy membership as $[\frac{x}{a}, \frac{x}{b}, \frac{x}{c}]$. These calculated fuzzy membership values are used in meteorological attributes fuzzification that comprises the process of transforming crisp values of meteorological attribute into grades of membership for linguistic terms of fuzzy sets.

4.6.3 Setting Meteorological Attributes of Each Node by Traversing in X-tree

In the structure of the X-tree, there are rectangles and meteorological attributes in each node. Meteorological attributes are used for secondary index on X-tree. In the creation step of the X-tree, only primary index that consists of spatial data is normally created by the insertion operation of the X-tree. After the primary index of the X-tree is created, there are no meteorological attributes in directory and supernode. Only data nodes have meteorological attributes. In these leaf nodes each meteorological attribute has value of its owner data that is observed in measurement and written on the input file's records. But these meteorological attributes have no

relation each other at this point. It means parent nodes of the data nodes, simply directory nodes, have no meaningful information about their child nodes' meteorological attributes. So these meteorological attribute values should be arranged for creating our secondary index structure. At this work, a complete navigation should be done over all the data node, normal directory nodes or super nodes and also root node. After this traverse is done, each node should have values which has min-max interval of child nodes' values.

The suitable way of doing this traverse is making recursive call. It means operation starts from the root node and traveling the X-tree's all nodes and finally finishes the travel on the root node by setting the min-max interval of all data used in the creation of the X-tree index. At starting point of algorithm about this recursive iteration begins with root node. Then it continues with handling all of its child nodes and doing the same for each child node until it reaches the data nodes. In recursive procedure, when it reaches the data node, it returns value to the parent node. In this progress every node has min-max interval of meteorological attributes that are set by using their child nodes' values. Finally after the traversing is completed, root node has min-max meteorological attribute interval of the whole data set. The pseudecode of this recursive procedure is given in Figure 29;

- if current node level is greater than zero
    - this node is not data node so that create dummy meteorological attribute values for each node for initialization.
    - get all the child node of the current node
        - call this function for each child node of the current node
        - set min max values of meteorological attributes
            - if parent min is higher than child min, set child value to parent
            - if parent max is less than child max, set child value to parent
- if current node level is equal to zero, this is leaf/data node
    - return the meteorological attribute values for this data node to the parent node

Figure 29: Algorithm of traversing in X-tree

This function starts executing from root node. Executing steps of this algorithm is shown in Figure 30.

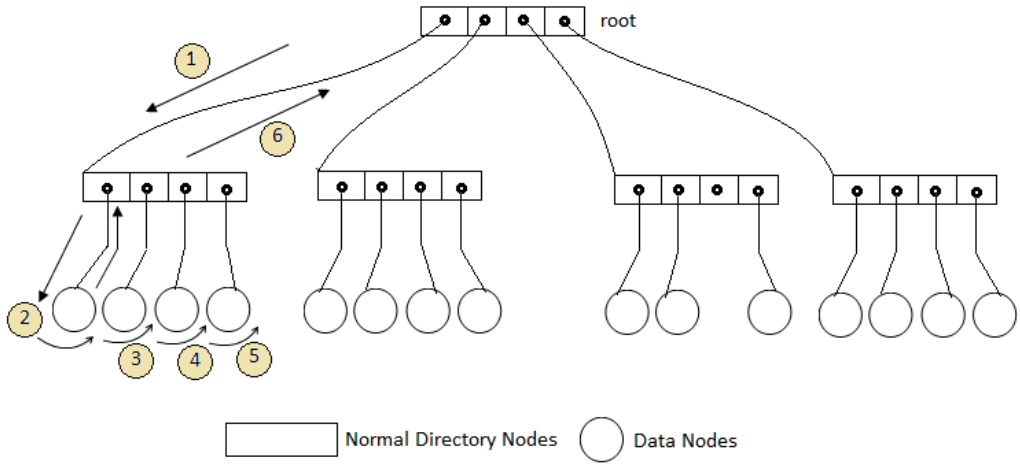

Figure 30: Recursive traverse on X-tree

4.7 Reading Records from File and Objectifying Them

In our work, we have text file that has meteorology stations and each station's meteorological values in each line record. The columns of a line record of this text file are shown below;

latitude	longitude	station	temperature	humidity	pressure	Precipitation period	Wind direction	Wind speed	altitude

The meanings of these columns can be given as in the following way;

latitude : latitude coordinate of the station

longitude : longitude coordinate of the station

station : identification number of the station

temperature : temperature value of the meteorological observation

humidity : humidity value of the meteorological observation

pressure : pressure value of the meteorological observation

precipitation period : precipitation period value of the meteorological observation

wind direction : wind direction value of the meteorological observation

wind speed : wind speed value of the meteorological observation

altitude : altitude coordinate of the station

In insertion, we could create X-tree index by using these values from text file. Latitude, longitude and altitude variables were used for spatial indexing that indicate the point of a rectangle for MBRs. These three values made our rectangle 3D form containing one for coordinate x, one for y and one for z. For generating a 3D rectangle from these three values we put this point on the lower left corner of the rectangle then calculated the other corner of the rectangle by adding static variable 10 to the latitude, longitude, altitude variables. We used 10 for static variable

because we wanted to show MBR object as cubic form which had 10 units each dimension length. Therefore we have got a 3D rectangle as shown in Figure 31.

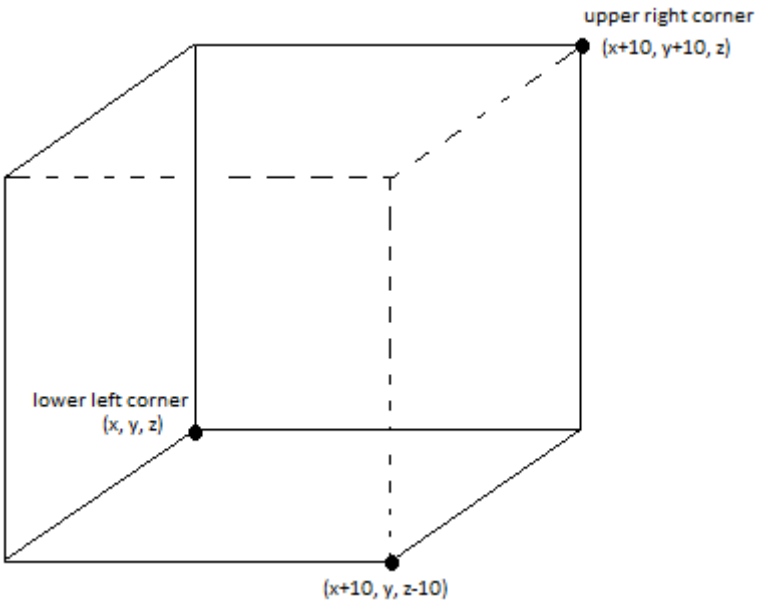

Figure 31: 3D rectangle for insertion to X-tree

So that three variables are used for spatial indexing and other variables of meteorological attributes are used for secondary index. In other words, three variables used for primary index and other six variables are used for secondary index that indicate meteorological attributes as shown in Figure 32.

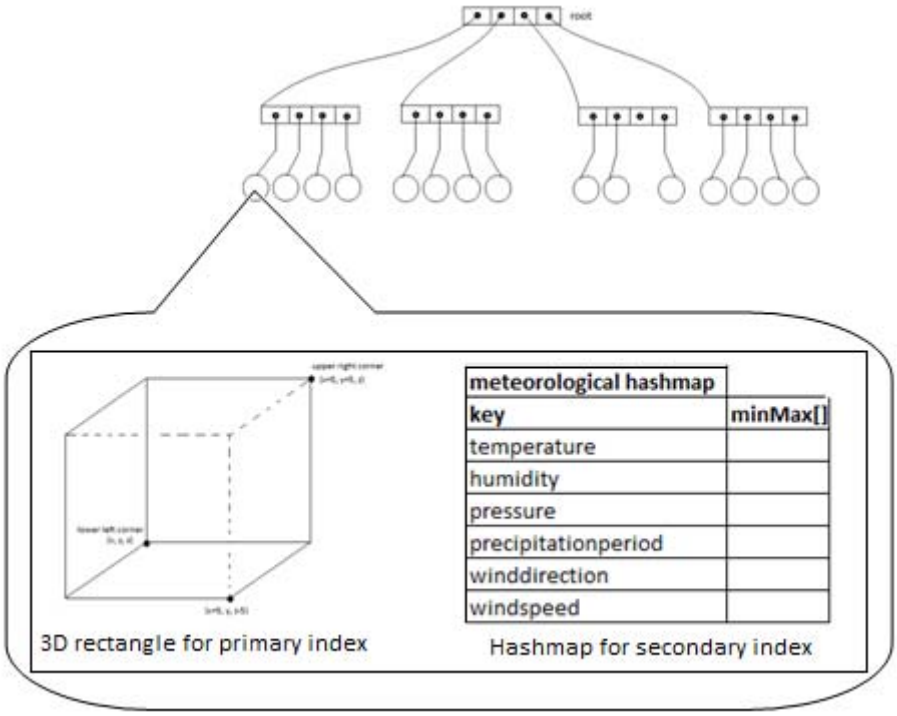

Figure 32: Detail view of node element in X-tree

In implementation details, we read the lines in the text file one by one then parse each line to identify the variables. Then we created 3D rectangle that was 3-dimensional Rectangle and extended the abstract class Rectangle the value at each dimension should be of the type double. After that, hash map for allocating meteorological attributes was created by using six meteorological variables. This hash map allocates <*temperature, humidity, pressure, precipitation period, wind direction, wind speed*> attributes. Then hash map of the meteorological attributes was put to the MBR object containing our 3D rectangle.

4.8 A Simple Example of X-tree Index Creation

We demonstrate the execution of X-tree index creation by using ten input data.

- Firstly we read data from input file. Example of input file is shown in Table 1.

Table 1: A simple view of input file

Latitude	Longitude	Station	Temperature	Humidity	Pressure	Precipitation Period	Wind Direction	Wind Speed	Altitude
41	10	17015	-4.70	97	1028	1	0	0	248.0
45	17	17016	1.80	94	1029	1	0	0	189.0
43	9	17017	3.30	72	1029	1	140	15	118.0
42	32	17018	1.60	99	1028	1	140	5	324.0
44	15	17019	2.60	91	1028	1	190	25	1041.0
40	22	17020	4.40	85	1028	1	240	10	36.0
36	28	17021	4.20	80	1027	1	230	5	1295.0
25	14	17022	0.50	100	1025	1	0	20	42.0
14	19	17023	-2.30	98	1029	1	0	10	60.0
35	30	17024	-1.40	94	1029	1	50	10	77.0

- Then we created 3D spatial primary index by using read data from input file.

We read records from input file line by line and executed FCM algorithm to calculate fuzzy membership for each record attributes. Then we applied algorithm as shown in Figure 33;

- for each line record in input file
  - read <latitude, longitude, altitude> values for 3D points
  - read <temperature, humidity, pressure, precipitation period, wind direction, wind speed> for meteorological attributes
  - create DoublePointRectangle object with these read values and apply conversion to created rectangle object
  - insert created rectangle object to the X-tree

Figure 33: Algorithm of reading line from input file and insertion to X-tree

After we read all records from input file and inserted them to the X-tree, we had 3D spatial index X-tree as shown in Figure 34. In our index we had data nodes that have meteorological attributes. On the other hand, in directory nodes we had no information about meteorological attributes.

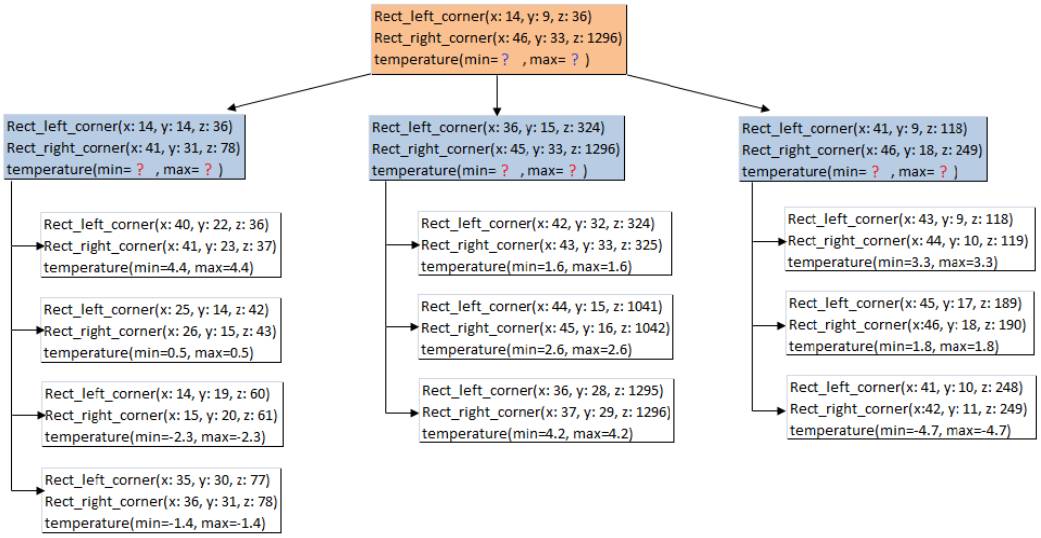

Figure 34: View of X-tree that has only 3D primary index

- Then we built fuzzy secondary index by overlaying created primary X-tree index.

Meteorological attributes of data node have no relation with each other at this point. It means parent nodes of the data nodes, simply directory nodes, have no meaningful information about their child nodes' meteorological attributes. So these meteorological attribute values should be arranged for creating our secondary index structure. Therefore a complete navigation should be done to travel all the data node indexes and their normal directory nodes or super nodes and also root node. After this traverse each node should have values which has min max interval of child nodes as shown in Figure 35.

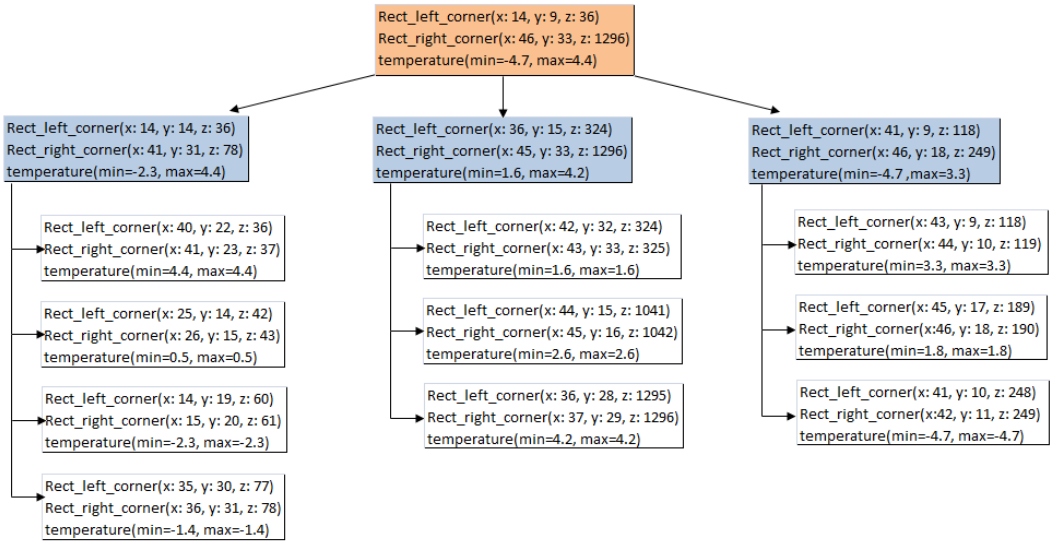

Figure 35: View of X-tree that has primary and secondary indexes

4.9 Building the Uncoupled Index Structure

We developed uncoupled index structure for handling 3D spatial data and fuzzy attributes in discrete constructions. Storing 3D spatial data was supported by using X-tree index structure and fuzzy attributes were saved in BPlusTree index structure.

For this purpose we created both 3D spatial index and secondary index by using each record in input file while we were performing insertion operation. In detail, firstly we read a line from input file and created a 3D spatial MBR object and stored it in X-tree. And then we created a fuzzy attribute object and inserted it in BPlusTree index.

Finally we had two separate indexes, one of them is X-tree as primary index and the other one is BPlusTree as secondary index as shown in Figure 36.

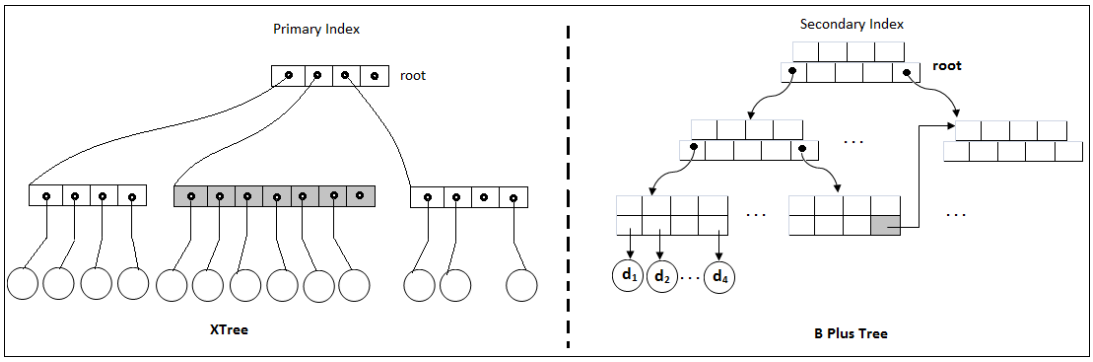

Figure 36: View of uncoupled index: X-tree for primary and BPlusTree for secondary index

4.9.1 X-tree Index Creation for 3D Spatial Primary Index

In this case we had simple X-tree that only contains 3D spatial MBR objects as shown in Figure 37. To create X-tree we just used <*latitude, longitude, altitude*> data.

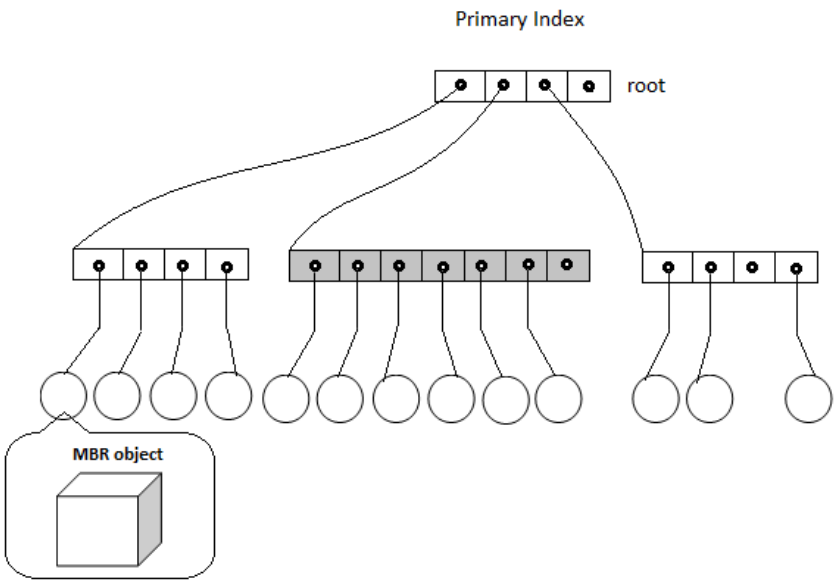

Figure 37: View of X-tree for primary index

4.9.2 BPlusTree Index Creation for Fuzzy Secondary Index

For secondary indexing we created BPlusTree structure as shown in Figure 38. We inserted <*temperature*> data to the index. Before we inserted this data to BPlusTree index, we applied FCM algorithm to get the fuzzy membership values of related temperature data. Therefore we inserted fuzzy membership value to the BPlusTree index.

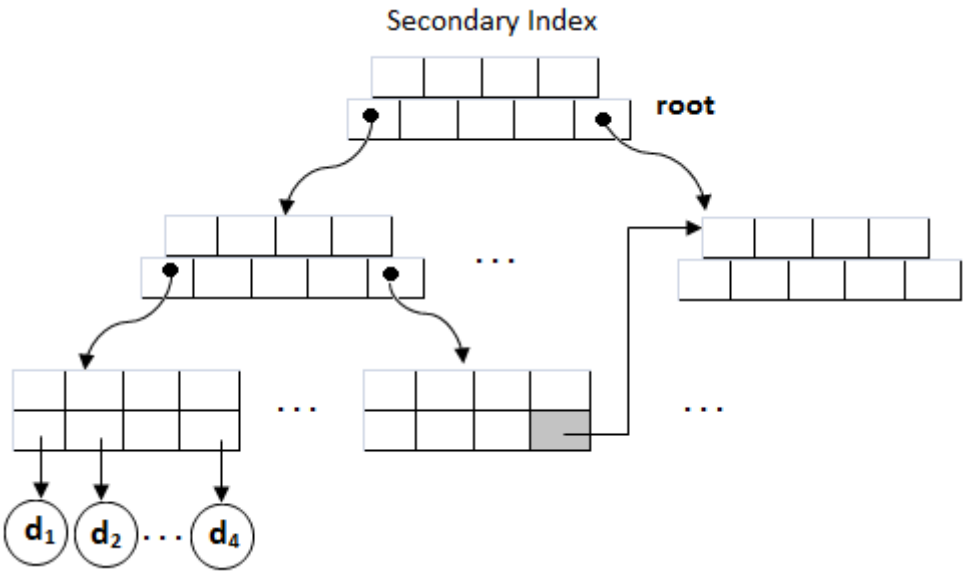

Figure 38: View of BPlusTree for secondary index

4.9.3 Management of 3D Spatial Index and Fuzzy Secondary Index

We implemented an algorithm for search operation by using both indexes as shown in Figure 39.

In here, we firstly used X-tree primary index in search operation by using 3D spatial parameters and we stored the results in a list. Then BPlusTree secondary index was used by using meteorological attributes and the results were also stored in another list.

Search Algorithm of Uncoupled Index Structures

1. Search on Xtree primary index and put results to spatialResultList
2. Search on BPulsTree secondary index and put results to fuzzyResultList
3. For each element in spatialResultList
   3.1. For each element in fuzzyResultList
      3.1.1.If spatialResultList element is member of fuzzyResultList
      3.1.1.1. Put element to finalResultList

Figure 39: Search algorithm for using both primary index and secondary index

Finally we tried to match the results if any element existed in both list or not. In matching operation we used station id for each records. If the element was in both result lists, we put it in final result list containing the entire searched element according to the query. If the element was not in both result lists, we could say that this element was not the exact element which we searched. Finally we had final result list that contains all the elements for which we queried.

CHAPTER 5

PERFORMANCE TESTS

5.1 Test Inputs and Queries

In performance tests section, different test cases were performed to get idea about the performance and features of both coupled and uncoupled index implementation approaches.

We noted elapsed time and iteration count values as a result of testing measurements. Time values were collected for giving idea about time performance of each index. And iteration count values were observed to show the I/O performance of each index. In each iteration process we fetched a node of index to analyze if it was proper for query or not. This fetch operation refers one I/O cost for index organization. By this way iteration count becomes meaningful for performance evaluation over two different index approaches.

In the beginning, insertion operation was afforded by using different number of input records. Then different types of queries were done to perform the structure.

5.1.1 Insertion Tests

In this step firstly coupled index creation was tested and then uncoupled index creation was observed. At test operation 1000-5000-10000-20000-40000-60000-80000 numbers of records were inserted by descending order in each step.

In insertion tests, we firstly tested data insertion to coupled index structure. And we observed insert operation results as seen in Table 2. In Table 2, number of inserted records, creation time of primary and secondary indexes, insertion total duration, supernode and normal node count for each level were given in detail. Then we secondly tested data insertion to uncoupled index structure. Results of this test were written in Table 3. We gave the same detail information as Table 2. However, there is no information about creation time of secondary index. The reason is that in uncoupled index structure primary index and secondary index was created individually. Therefore in the same time we could build both of them. It means both indexes creation was done during the same time period.

Table 2: Observed values of coupled index structure insertion

Number of Inserted Records	Primary Index Duration (ms)	Secondary Index Duration (ms)	Total Time for Insertion (ms)	Number of super nodes per level	Number of normal nodes per level	Height
1000	557	59	616	[0, 0, 0, 0, 0]	[0, 52, 12, 2, 1]	5
5000	2841	998	3839	[0, 11, 5, 1, 0, 0]	[0, 302, 51, 10, 2, 1]	6
10000	6693	3696	10389	[0, 25, 10, 1, 0, 0, 0]	[0, 651, 137, 28, 8, 2, 1]	7
20000	17802	15846	33648	[0, 56, 28, 12, 1, 0, 0]	[0, 1373, 282, 44, 10, 3, 1]	7
40000	47314	72010	119324	[0, 104, 48, 27, 5, 0, 0]	[0, 2976, 694, 107, 19, 4, 1]	7
60000	89340	185920	275260	[0, 159, 53, 38, 8, 0, 0, 0]	[0, 4547, 1126, 187, 24, 7, 2, 1]	8
80000	149271	330094	479365	[0, 219, 61, 50, 12, 0, 0, 0]	[0, 6034, 1532, 266, 33, 10, 2, 1]	8

Table 3: Observed values of uncoupled index structure insertion

Number of Inserted Records	Primary & Secondary Index Duration (ms)	Total Time for Insertion (ms)	Number of super nodes per level	Number of normal nodes per level	Height
1000	610	610	[0, 0, 0, 0, 0]	[0, 52, 12, 2, 1]	5
5000	2295	2295	[0, 11, 5, 1, 0, 0]	[0, 302, 51, 10, 2, 1]	6
10000	4974	4974	[0, 25, 10, 1, 0, 0, 0]	[0, 651, 137, 28, 8, 2, 1]	7
20000	11289	11289	[0, 56, 28, 12, 1, 0, 0]	[0, 1373, 282, 44, 10, 3, 1]	7
40000	24126	24126	[0, 104, 48, 27, 5, 0, 0]	[0, 2976, 694, 107, 19, 4, 1]	7
60000	37636	37636	[0, 159, 53, 38, 8, 0, 0, 0]	[0, 4547, 1126, 187, 24, 7, 2, 1]	8
80000	52904	52904	[0, 219, 61, 50, 12, 0, 0, 0]	[0, 6034, 1532, 266, 33, 10, 2, 1]	8

We combined total duration and total iteration results of Table 2 and Table 3 to demonstrate better view of performance difference in Table 4.

Table 4: Coupled and uncoupled index structures insertion comparison table

Number of Inserted Records	Total Time for Coupled Index Insertion (ms)	Total Time for Uncoupled Index Insertion (ms)	Difference (ms)
1000	616	610	6
5000	3839	2295	1544
10000	10389	4974	5415
20000	33648	11289	22359
40000	119324	24126	95198
60000	275260	37636	237624
80000	479365	52904	426461

By using results in Table 4, we generated comparison chart, shown in Figure 40.

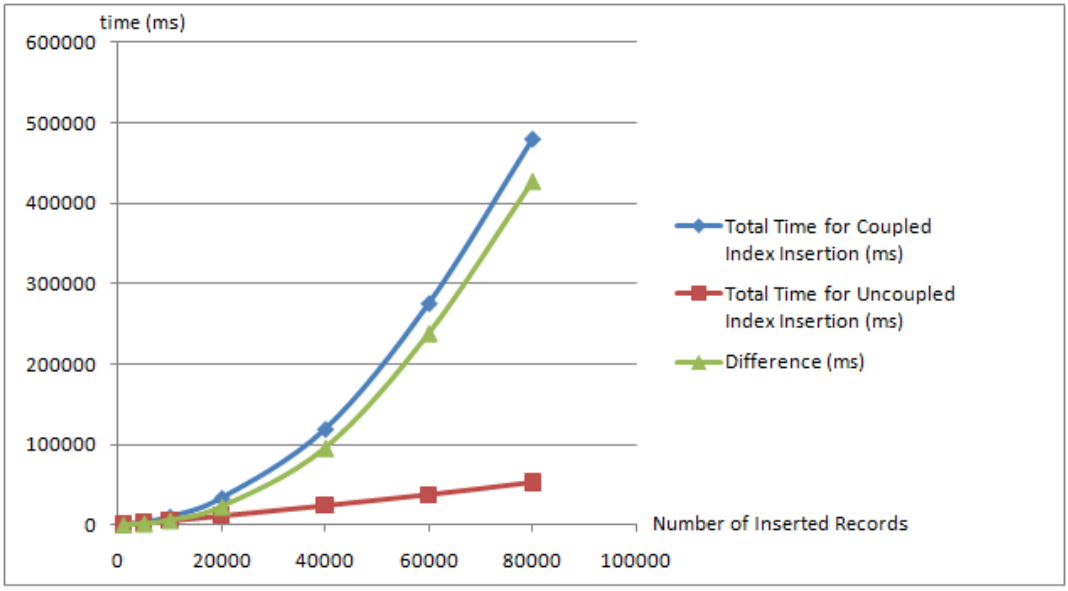

Figure 40: Coupled and uncoupled index structures insertion comparison chart

With respect to our experiment it can be observed that;

- The behavior of insertion on coupled index structure over growing number of record is exponential. And insertion duration difference between these index

structures is also exponential. In coupled index structure, we should traverse whole of X-tree to overlay the secondary index on this index. When we inserted more records to X-tree, we got larger X-tree that contained more nodes. Therefore traversing on this X-tree took longer time and behavior of this relation became exponential.

- The behavior of insertion on uncoupled index structure over growing number of record is linear.

5.1.2 Query Tests

After insertion, we tested query operations over both coupled and uncoupled index structures. For this purpose in coupled structure, 3D spatial locations were queried on primary index of X-tree and also fuzzy attributes were queried on secondary index of X-tree. In uncoupled structure, 3D spatial locations were queried on primary index of X-tree as in the coupled structure, but fuzzy attributes were searched on BPlusTree secondary index. We explained this operation details in Section 4.9.3 where we also expressed the management algorithm of both indexes.

In coupled index structure tests we only observed total duration and total iteration count on X-tree. Because of this structure was monolithic, we could only observe these results. However, in uncoupled index structure, we could observe query time and iteration count on primary and secondary indexes individually.

5.1.2.1 Point Query

This query is used for finding exact point in our spatial index as shown in Figure 41. The query has capability of searching fuzzy attribute values on secondary index also. As we know that our X-tree contains rectangles, not points. So in point query, we generated 3D rectangle by using 3D point as like insertion operation that mentioned previous section.

A simple usage of Point Query can be given as in the following way;

"Find the station which is based in <10, 20, 30> coordinates and it is temperature is higher than 0,01 hot"

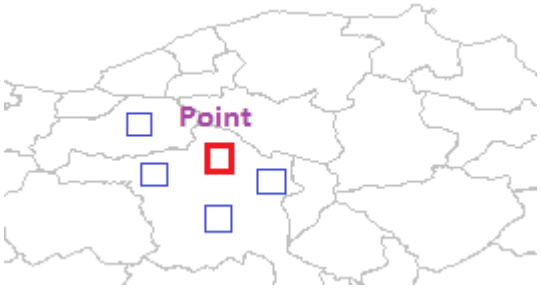

Figure 41: Point query

We firstly run point query on coupled index and noted duration, iteration results in Table 5.

Table 5: Observed values about point query on coupled index structure

Number of Inserted Records	Primary&Secondary Index Query Duration (ms)	Primary&Secondary Index Query Iteration
1000	2	20
5000	3	27
10000	4	28
20000	4	30
40000	4	30
60000	5	31
80000	5	31

Then we run the same query on uncoupled index and wrote the results in Table 6. In this table primary and secondary index duration, iteration and their sums are available.

Table 6: Observed values of point query on uncoupled index structure

Number of Inserted Records	Primary Index Query Duration (ms)	Secondary Index Query Duration (ms)	Total Time for query (ms)	Primary Index Iteration	Secondary Index Iteration	Iteration for match operation	Total Iteration
1000	2	7	9	20	37	167	224
5000	3	12	15	27	180	3669	3876
10000	3	25	28	28	424	15883	16335
20000	4	71	75	30	1024	47371	48425
40000	4	641	645	30	2087	83314	85431
60000	5	781	786	31	3241	124807	128079
80000	7	968	975	31	4357	160360	164748

We created Table 7 by using total duration and total iteration results of Table 5 and Table 6 to demonstrate better view of performance difference.

Table 7: Coupled and uncoupled index structures point query comparison table

Number of Inserted Records	Coupled Index Structure		Uncoupled Index Structure	
	Total Time for query (ms)	Total Iteration	Total Time for query (ms)	Total Iteration
1000	2	20	9	224
5000	3	27	15	3876
10000	4	28	28	16335
20000	4	30	75	48425
40000	4	30	645	85431
60000	5	31	386	128079
80000	5	31	475	164748

By using iteration results in Table 7, we generated iteration comparison charts between coupled and uncoupled indexes, shown in Figure 42. In this figure, (a) shows total iteration count for coupled index structure, (b) shows total iteration count for uncoupled index structure and (c) shows both of them in a chart.

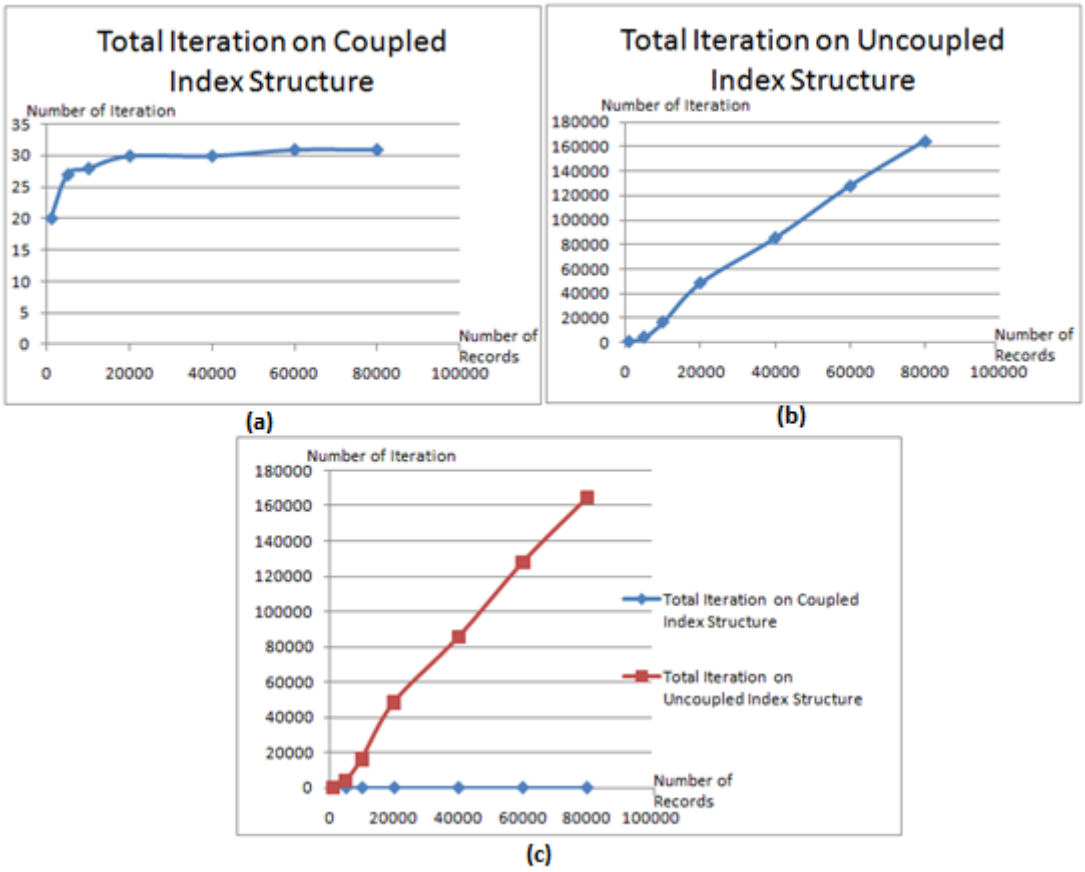

Figure 42: Point query total iteration charts

By analyzing these charts we can argue that;

- In iteration perspective, behavior of coupled index structure is logarithmic in point query. This situation is derived from height of balanced X-tree. X-tree is a balanced tree, so searching performance on this structure is related to the height of the tree.

- Again in iteration view, behavior of uncoupled index structure is linear in point query. In Table 6 we can see that primary index iteration numbers are equal to the same values in Table 5. But secondary index iteration numbers are always increasing according to the rise of inserted record count in uncoupled index structure. In addition iterations that were done for match operation is also increasing. Therefore these increases made the behavior linear.

We also drew duration comparison charts between coupled and uncoupled indexes by using elapsed time results in Table 7, shown in Figure 43. In this figure, (a) shows total time for coupled index structure, (b) shows total time for uncoupled index structure and (c) shows both of them in a chart.

Figure 43: Point query total execution time charts

Coupled and uncoupled indexes' elapsed time charts are similar to their iteration charts. So that;

- Coupled index structure's elapsed time chart has logarithmic behavior as its iteration charts.
- Uncoupled index structure's elapsed time chart has linear behavior as its iteration charts and it has the same reasons about this situation.

5.1.2.2 Range Query

Range query is a common database operation that retrieves all records where some value is between an upper and lower boundary. In here as shown in Figure 44, range query is used for searching spatial elements in the given rectangular range that covers upper and lower boundary. In addition we also checked the rectangle's fuzzy attribute by using fuzzy secondary index. If the input rectangle contains any spatial element and also this covered element provides wanted fuzzy condition, it will be shown in result set.

An example usage of Range Query can be given as in the following way:

"Find the stations that are covered by rectangle with lower boundary <10, 20, 30> and upper boundary <20, 30, 40> coordinates and it is temperature is higher than 0,01 hot"

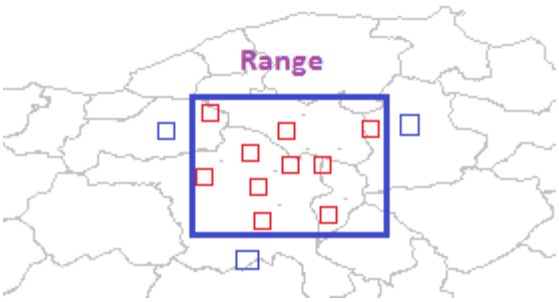

Figure 44: Range query

Range query test results over different number of inserted record were collected for coupled index structure in Table 8. In this table we also gave the found element count according to each query.

Table 8: Observed values of range query on coupled index structure

Number of Inserted Records	Primary&Secondary Index Query Duration (ms)	Primary&Secondary Index Query Iteration	Found Element Count
1000	10	32	8
5000	15	59	17
10000	17	62	19
20000	22	62	32
40000	25	62	32
60000	27	65	32
80000	30	65	32

Then we run the same range query on uncoupled index and wrote the results in Table 9. In this table primary and secondary index duration, iteration and their sums are available.

Table 9: Observed values of range query on uncoupled index structure

Number of Inserted Records	Primary Index Query Duration (ms)	Secondary Index Query Duration (ms)	Total Time for query (ms)	Primary Index Iteration	Secondary Index Iteration	Iteration for Match Operation	Total Iteration	Found Element Count
1000	4	8	12	32	37	3374	3443	8
5000	7	57	64	59	180	70118	70357	17
10000	8	63	71	62	424	146852	147338	19
20000	8	179	187	62	1024	392695	393781	32
40000	9	281	290	62	2087	615202	617351	32
60000	10	418	428	65	3241	909867	913173	32
80000	12	1022	1034	65	4357	1172061	1176483	32

We joined total duration and total iteration results of Table 8 and Table 9 in Table 10 to demonstrate better view of performance difference.

Table 10: Coupled and uncoupled index structures range query comparison table

Number of Inserted Records	Coupled Index Structure		Uncoupled Index Structure	
	Total Time for query (ms)	Total Iteration	Total Time for query (ms)	Total Iteration
1000	10	32	12	3443
5000	15	59	64	70357
10000	17	62	71	147338
20000	22	62	187	393781
40000	25	62	290	617351
60000	27	65	428	913173
80000	30	65	1034	1176483

We used iteration results in Table 10 to show iteration comparison charts between coupled and uncoupled indexes, shown in Figure 45. In this figure, (a) shows total iteration count for coupled index structure, (b) shows total iteration count for uncoupled index structure and (c) shows both of them in a chart.

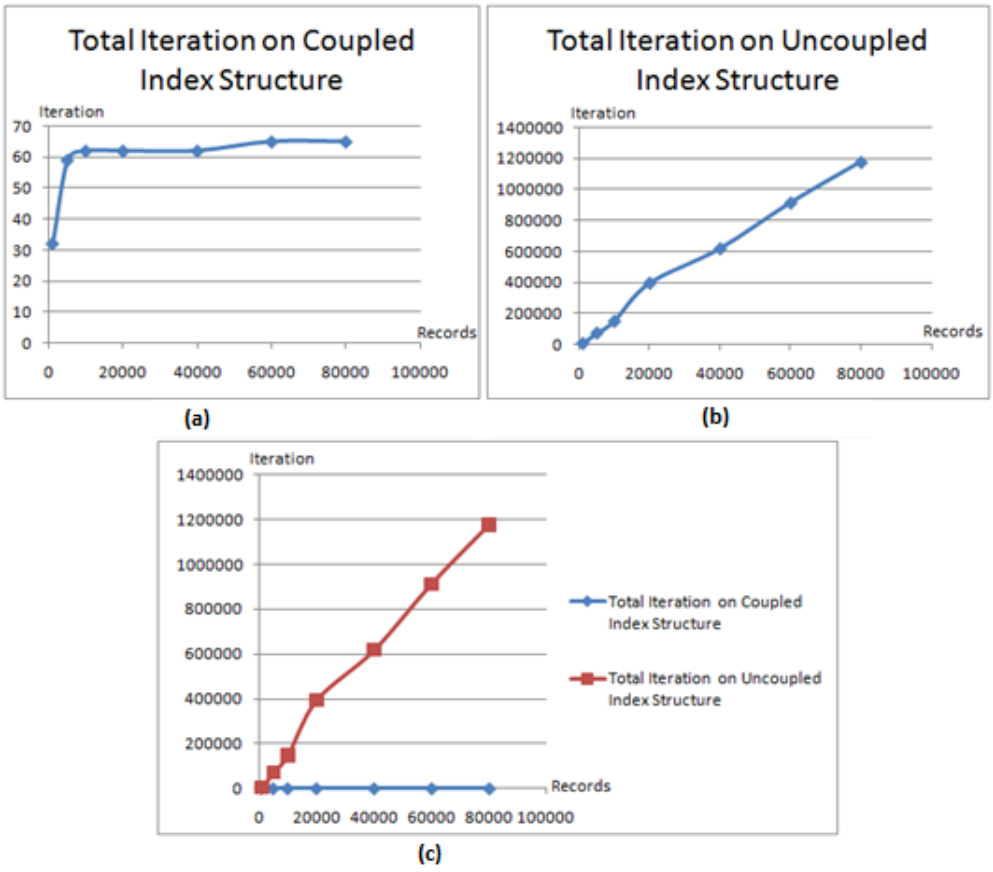

Figure 45: Range query total iteration charts

By the side of these charts analysis we can argue that;

- Behavior of coupled index structure is logarithmic in range query according to iteration count. X-tree is a balanced tree, so searching performance on this structure is related to the height of tree.
- Uncoupled index structure side, behavior of chart is linear in range query. In Table 9 we can see that primary index iteration numbers are equal to the same values in Table 8. But secondary index and match operation iteration numbers are always increasing according to the rise of inserted record count in uncoupled index structure. As a result these increases made the behavior linear.

Comparison charts between coupled and uncoupled indexes are shown in Figure 46. In this figure, (a) shows total time for coupled index structure, (b) shows total time for uncoupled index structure and (c) shows both of them in a chart.

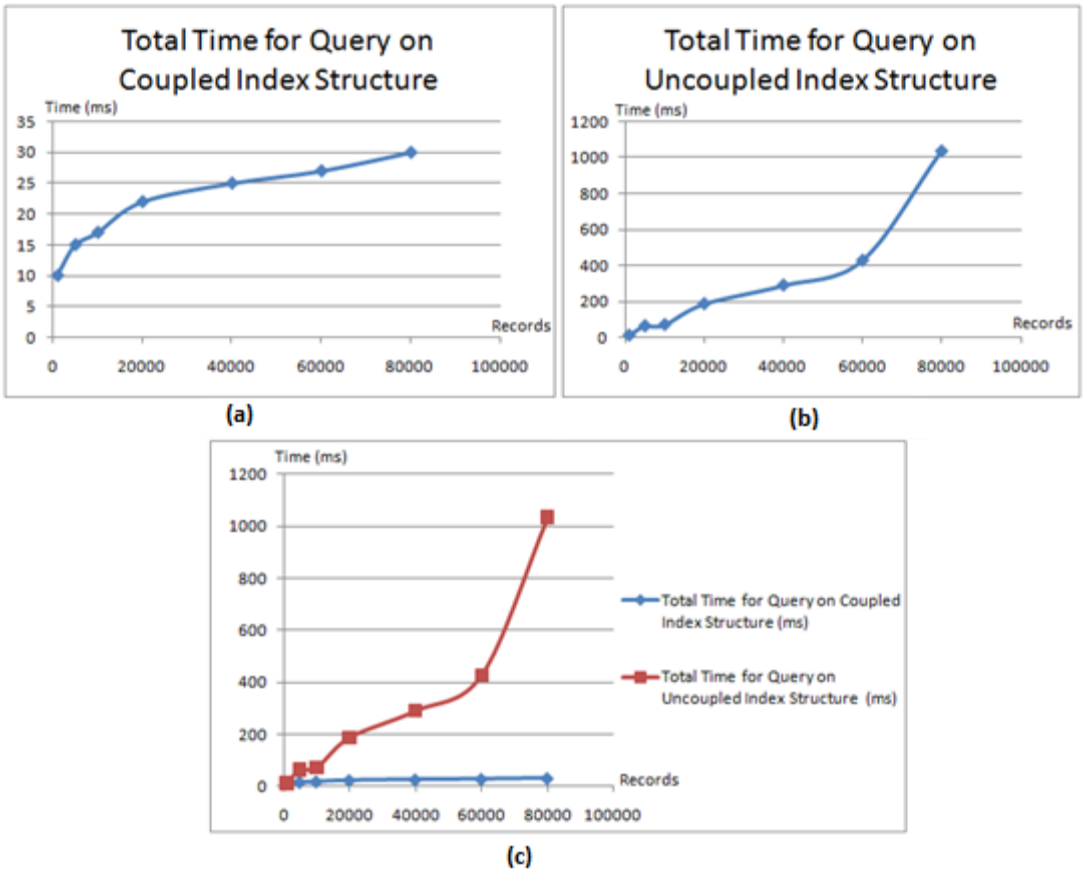

Figure 46: Range query total execution time charts

The element count in the result list on range query and total iteration on querying process are graphed as seen in Figure 46. The chart shows that the behavior of range query about iteration over growing number of inserted record is linear that both of them are rising with nearly the same ratio.

5.1.2.3 Nearest Neighbor Query

NN (Nearest Neighbor) query, also called closest point query, is an optimization problem for finding closest points in metric spaces. The aim of query is about given a set S of points in a metric space M and a query point q that is an element of M; find the closest point in S to q.

In our tests, performing a nearest neighbor query against the tree determining the 10 nearest neighbor entries at target level concerning the input rectangle is applied as seen in Figure 47. In addition we also wanted fuzzy condition in querying.

An example usage of Nearest Neighbor Query can be given as in the following way.

"Find 10 nearest stations to the given station which is based in <10, 20, 30> coordinates and it is temperature is higher than 0,01 hot"

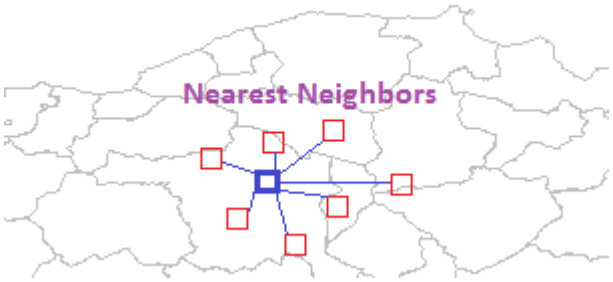

Figure 47: Nearest Neighbor Query

Nearest neighbor query results over different number of inserted record were collected for coupled index structure in Table 11. In this table we noted duration, number of distance based iterations for finding closest points and also found element

count. Naturally we always got ten result elements after querying on coupled index structure.

Table 11: Observed values of NN query on coupled index structure

Number of Inserted Records	Primary&Secondary Index Query Duration (ms)	Primary&Secondary Index Distance Based Query Iteration	Found Element Count
1000	38	494	10
5000	45	459	10
10000	98	1107	10
20000	102	846	10
40000	79	466	10
60000	83	497	10
80000	86	606	10

The same query was executed on uncoupled index and the results were written in Table 12. In this table primary and secondary index duration, iteration and their sums are available. In detail, found element count is not always ten on uncoupled structure. Because in uncoupled index structure we firstly run query on primary index and we got ten nearest neighbor elements, then we run query on secondary index for fuzzy attributes, and finally we verified match operation between these two result sets. Sometimes the elements that were in primary index result set could not provide fuzzy condition so it could not be a member of final result set. Therefore founded element count could be less than ten. To support ten results we may need to query more than ten elements on primary index and make match operation over these elements. Therefore in this situation query operation cost is become higher.

Table 12: Observed values of nearest NN on uncoupled index structure

Number of Inserted Records	Primary Index Query Duration (ms)	Secondary Index Query Duration (ms)	Total Time for query (ms)	Primary Index Distance Based Iteration	Secondary Index Iteration	Iteration for match operation	Total Iteration	Found Element Count
1000	8	45	53	486	37	5122	5645	10
5000	14	57	71	927	180	36781	37888	10
10000	20	126	146	1121	424	34914	36459	10
20000	37	109	146	846	1024	70204	72074	10
40000	40	202	242	466	2087	169102	171655	10
60000	43	282	325	497	3241	362006	365744	10
80000	54	447	501	606	4357	463529	468492	10

Table 13 was created by using total duration and total iteration results of Table 11 and Table 12 to demonstrate better view of performance difference.

Table 13: Coupled and uncoupled index structures NN query comparison table

Number of Inserted Records	Coupled Index Structure		Uncoupled Index Structure	
	Total Time for query (ms)	Total Iteration	Total Time for query (ms)	Total Iteration
1000	38	494	53	5645
5000	45	459	71	37888
10000	98	1107	146	36459
20000	102	846	146	72074
40000	79	466	242	171655
60000	83	497	325	365744
80000	86	606	501	468492

Then we generated iteration comparison graphs between coupled and uncoupled indexes by using iteration results in Table 13. Number of total iteration over growing number of inserted record charts are shown in Figure 48. In this figure, (a) shows total iteration count for coupled index structure, (b) shows total iteration count for uncoupled index structure and (c) shows both of them in a chart.

By analyzing these charts we can say that;

- Coupled index structure has no meaningful behavior in nearest neighbor query. Because observed iteration count is related to distance based search. Sometimes this operation can need lots of dual distance comparison to find closest neighbor or sometimes not need lots of comparison. Comparison is relevant to the point density in searched boundary.

- On the other hand uncoupled index structure has linear behavior in nearest neighbor query because of the match operation between primary and secondary indexes' results.

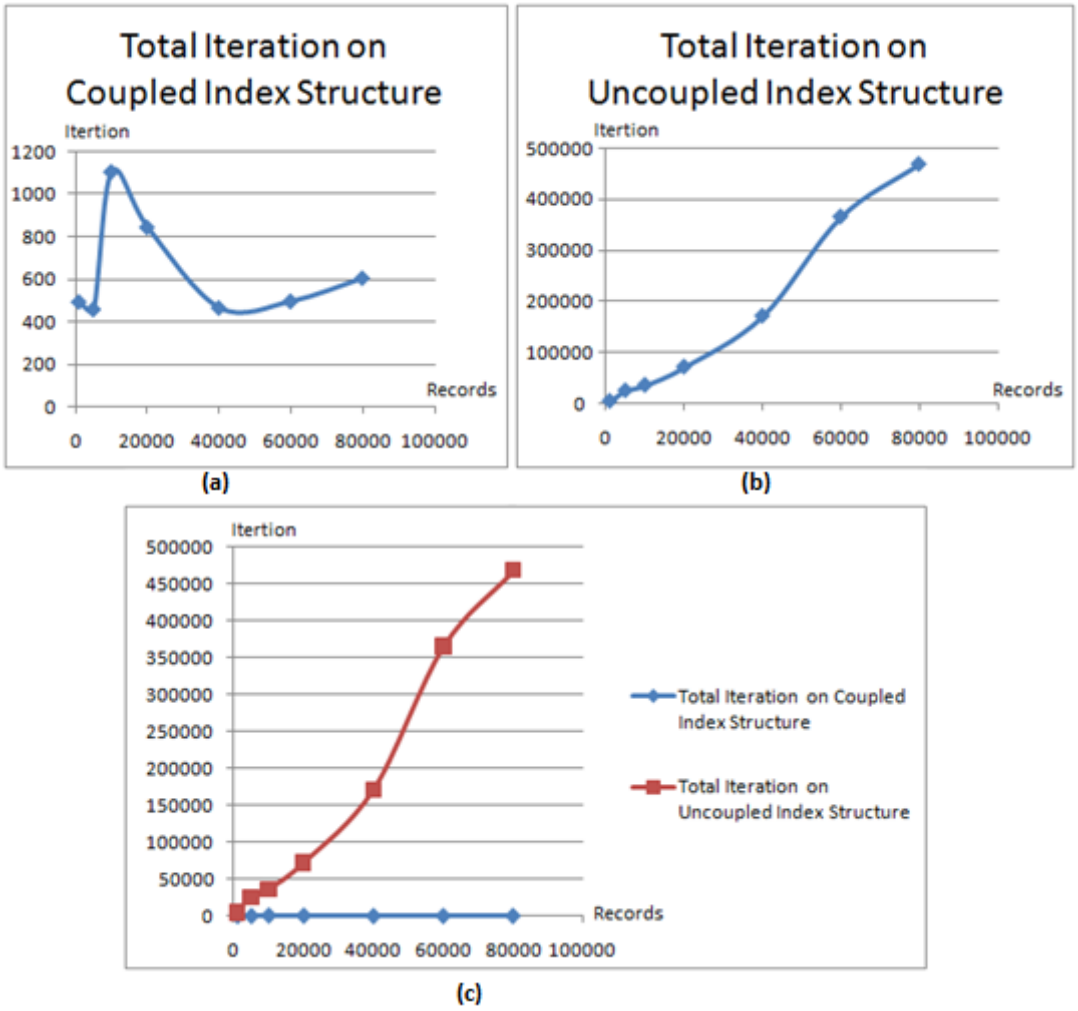

Figure 48: Nearest Neighbor query total iteration charts

For nearest neighbor query result, we also drew duration comparison charts between coupled and uncoupled indexes by using duration results in Table 13, shown in

Figure 49. In this figure, (a) shows total time for coupled index structure, (b) shows total time for uncoupled index structure and (c) shows both of them in a chart.

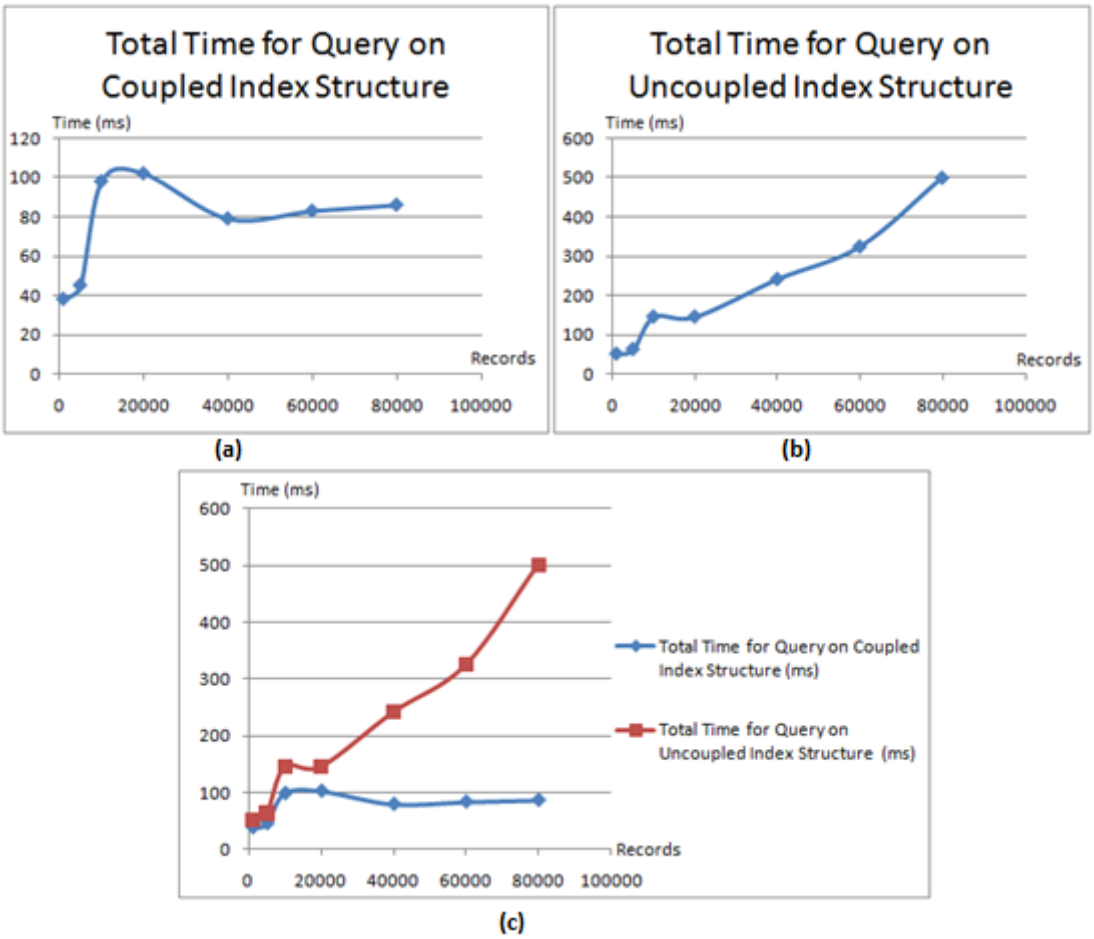

Figure 49: Nearest Neighbor query total execution time charts

Time comparison charts' behaviors are the same as iteration comparison charts. Again coupled index structure has no relation for different number of inserted records and also uncoupled index structure behavior is linear because of match operation between primary and secondary indexes' results.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis work, we worked on 3D spatial primary index and fuzzy secondary index. Firstly we implemented a coupled index structure that comprises both primary index and secondary index in monolithic X-tree index structure. Secondly we developed separated index structures as uncoupled approach. In this approach, we implemented X-tree index for 3D spatial objects and BPlusTree for fuzzy meteorological attributes individually. Then we performed these index structures with insertion and some basic queries to demonstrate the performance difference between them.

For this aim a java based library which is called XXL API is used for base structure of X-tree. We also used XXL API for BPlusTree index structure. The source code of this project is adapted and modified for our purpose to handle both primary and secondary indexes. For fuzzy part Fuzzy-C Means algorithm is applied for fuzzification of meteorological values and also modifications are done on calculated membership values.

In performance test section, according to the insertion operation uncoupled index structure was more efficient than coupled index structure because we did overlaying on coupled index structure to build fuzzy secondary index. So this operation badly affected insertion performance of coupled index structure. On the other hand, in querying coupled index structure was more efficient than uncoupled index structure because we run query in two different indexes and verified match operation over two result sets. For querying, we proved that coupled index structure more efficient than uncoupled index structure.

For future works, our coupled X-tree index structure can be compared with other R-tree based index structure, such as R*-tree, in a suitable platform. Especially insertion operation and querying on fuzzy secondary index can be observed via testing with different number of record and different test cases. After testing special X-tree and special R*-tree structure, deductions can be made over their performance and features.

REFERENCES

[1] http://undergraduate.csse.uwa.edu.au/units/CITS4241/Handouts/Lecture14.html, 12.07.2010

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD90), pages 322-331, May 1990.

[3] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In Proceedings of the 22th International Conference on Very Large Data Bases (VLDB96), pages 28-39, 1996.

[4] A. Guttman. R-trees: A dynamic index structure for spatial searching. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD84), pages 47-57, 1984.

[5] Antonin Guttman: R-Trees: A Dynamic Index Structure for Spatial Searching, Proc. 1984 ACM SIGMOD International Conference on Management of Data, pp. 47–52.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger: The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles, SIGMOD Conference, 322-327, 1990

[7] S. Berchtold,D. A. Keim, and H. P. Kriegel. 'The X-tree: An Index Structure for High-Dimensional Data', Proc. of VLDB, Institute for Computer Science, University of Munich, Munich, Germany, 1996, pp. 4-8.

[8] J. Bercken, B. Blohsfeld, J.-P. Dittrich, J. Kramer, T. Schafer, M. Schneider, and B. Seeger. XXL - A Library Approach to Supporting Efficient Implementations of

Advanced Database Queries. In Proc. of the Conf. on Very Large Databases (VLDB), pages 39-48, 2001.

[9] M. Cammert, C. Heinz, J. Kramer, M. Schneider, and B. Seeger. A Status Report on XXL - a Software Infrastructure for Efficient Query Processing. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 26(2):12-18, 2003.

[10] The XXL Project, http://www.mathematik.uni-marburg.de/dbs/xxl, 14.07.2010

[11] JavaTM 2 Platform, Standard Edition, v 1.4.1 API Specification,

http://java.sun.com/j2se/1.4.1/docs/api/, 15.07.2010

[12] The Colt Distribution - Open Source Libraries for High Performance Scientific and Technical Computing in Java,

http://hoschek.home.cern.ch/hoschek/colt/V1.0.3/doc/index.html, 19.07.2010

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley. October 1994.

[14] J. Hughes. Why Functional Programming Matters. Computer Journal, 32(2):98-107, 1989.

[15] P. Ciaccia, M. Patella, P. Zezula: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. VLDB Conf. 1997: 426-435

[16] D. Lomet: Grow and Post Index Trees: Roles, Techniques and Future Potential. Proc. Symp. on Spatial Databases 1991: 183-206

[17] D. Lomet. Grow and Post Index Trees: Role, Techniques and Future Potential. In SSD, 1991.

[18] A. Wilschut, P. Apers: Dataflow Query Execution in a Parallel Main-Memory Environment. PDIS 1991: 68-77

[19] Nam B., Sussman A., A Comparative Study of Spatial Indexing Techniques for Multidimensional Scientific Datasets , UMIACS and Dept. of Computer Science, University of Maryland, 2002, pp.3-4

[20] J. C. Dunn, "A fuzzy relative of the ISODATA process and its use in detecting compact well-separated clusters", J. Cybern., vol. 3, pp. 32–57, 1973.

[21] J. Bezdek, "A convergence theorem for the fuzzy ISODATA clustering algorithms", IEEE Trans. Pattern Anal. Machine Intell., vol. PAMI-2, pp. 1–8, 1980.

[22] Yang G., Zhang J. 'A Dynamic Index Structure for Spatial Database Querying Based on R-trees', Institute Remote Sensing Applications, Chinese Academy of Sciences, Beijing, P.R.China, Chinese Academy of Surveying and Mapping, Beijing, P.R.China, pp. 1-3.

[23] Zhang Y., Zhou L., Chen J. 'Layered R-tree: An Index Structure for Three Dimensional Points and Lines', Department of Computer Science and Technology, Tsinghua University, Beijing, China, pp. 1-2.

[24] Bliujute R., Jensen C.S., Saltenis S., and Slivinskas G. 'R-tree Based Indexing of Now-Relative Bitemporal Data', A TIMECENTER Technical Report, 1998, pp. 4-8.