University of Memphis
University of Memphis Digital Commons

Electronic Theses and Dissertations

7-24-2014

# R-Forest for Approximate Nearest Neighbor Queries in High Dimensional Space

Michael Charles Nolen

Follow this and additional works at: https://digitalcommons.memphis.edu/etd

Recommended Citation
Nolen, Michael Charles, "R-Forest for Approximate Nearest Neighbor Queries in High Dimensional Space" (2014). *Electronic Theses and Dissertations*. 1029.
https://digitalcommons.memphis.edu/etd/1029

This Dissertation is brought to you for free and open access by University of Memphis Digital Commons. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of University of Memphis Digital Commons. For more information, please contact khggerty@memphis.edu.

# R-FOREST FOR APPROXIMATE NEAREST NEIGHBOR QUERIES IN HIGH DIMENSIONAL SPACE

by

Michael Charles Nolen

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Major: Computer Science

The University of Memphis

August 2014

## Dedication

I wish to dedicate this work to my family.

**Acknowledgement**

I wish to thank my advisor and mentor Dr Lin for his guidance, support and encouragement during my PhD study and Masters degree. He has gone beyond the call of a teacher and professor during my studies at the University of Memphis just as he does with the rest of his students. I don't believe I would have finished this work without him. I would like to thank my committee members Dr. Dasguupta, Dr. Rus, Dr Phan and especially Dr Hnat who substituted at the last minute for Dr. Phan during my final defense. I would like to add that it has been a real honor working with these professors and the other professors during my studies. Finally I would like to thank my wife Daisy for her caring support, children Rebekah and Christopher for their patience with their father, my parents Anne and Charles Nolen for their support all of these years.

## Abstract

Nolen, Michael Charles. Ph.D. The University of Memphis. August 2014. R-Forest for Approximate Nearest Neighbor Queries in High Dimensional Space. Major Professor: King-Ip Lin Ph.D.

Searching high dimensional space has been a challenge and an area of intense research for many years. The dimensionality curse has rendered most existing index methods all but useless causing people to research other techniques. In my dissertation I will try to resurrect one of the best known index structures, R-Tree, which most have given up on as a viable method of answering high dimensional queries. I have pointed out the various advantages of R-Tree as a method for answering approximate nearest neighbor queries, and the advantages of locality sensitive hashing and locality sensitive B-Tree, which are the most successful methods today. I started by looking at improving the maintenance of R-Tree by the use of bulk loading and insertion. I proposed and implemented a new method that bulk loads the index which was an improvement of standard method. I then turned my attention to nearest neighbor queries, which is a much more challenging problem especially in high dimensional space. Initially I developed a set of heuristics, easily implemented in R-Tree, which improved the efficiency of high dimensional approximate nearest neighbor queries. To further refine my method I took another approach, by developing a new model, known as R-Forest, which takes advantage of space partitioning while still using R-Tree as its index structure. With this new approach I was able to implement new heuristics and can show that R-Forest, comprised of a set of R-Trees, is a viable solution to high dimensional approximate nearest neighbor queries when compared to established methods.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

## 1. Introduction

For many years researchers in different fields have been creating very large data sets, comprised of large numbers of attributes commonly known as high dimensional data. In many applications the need arises to determine the similarity of an object to one or more objects known as nearest neighbor (NN) [84], K-nearest neighbor (KNN) were as others wish to find objects in a range [60]. In other cases someone may ask the opposite question such that given an object, which objects in the set have this object as their nearest neighbor, known as reverse nearest neighbor [89, 92, 109]. Most cases the objects themselves are represented as a transformation of the attributes into a feature vector of high dimensionality, possibly numbering into the thousands of dimensions [54]. This set of objects which can be thought of as the search space (search domain), becomes a database. Real world applications can be found in molecular biology [88], gene identification, DNA sequencing and protein database sequence searching [2]. Other fields include image retrieval from multimedia databases [20, 33, 34, 54, 69, 74, 80, 96, 100, 114, 115]. More recently there has been an increasing interest into image and video recognition. Images are broken down and transformed into feature sets which become the objects in a database a similarity search (query) involves the same process an image is transformed into a feature vector which becomes the query object [72, 79, 82]. Google has taken this process a step further by attempting to develop processes to recognize video based on overlapping region and content requiring the search of high dimensional space [90]. Whereas other researchers in academia, have been focusing their research on video copy detection [32], and visual searching of objects in videos [91]. Others have applied this same research to the area of audio by attempting to use nearest neighbor

(NN) and approximant nearest neighbor (ANN) to find similar portions of music [19, 71]. Even in the field of computer learning which usually involves the transformation of objects into feature sets of high dimensionality, the need arises to search these spaces efficiently for NN and KNN [64]. All of these searches have the recurring theme of needing to deal with high dimensional data and the requirement to search for similar objects.

The common theme is the representation of an object as a set of high dimensional data, with the need to locate similar objects. When it comes to retrieval in any large data set it is usually done via a data structure known as an index. An index can be loosely defined as a method to locate a record or group of records in a collection of records (database) by using a sub-set of the data which is organized in such a way as to facilitate an efficient search. Here an index is used much the same way as a card catalog is used in a library. Without an index, a query would have to resort to a sequential scan just as a person without a card catalog would have to do in a library (look at every book).

One must ask, why, is this problem important? Researchers in the field of database systems have developed many indexing methods which are used in all major relational database management systems (RDBMS)[1, 52]. The most prevalent is the family of tree structures know as B-Tree [57], with B+Tree being the most well known and widely used. So it appears there is a simple solution to high dimensional NN and KNN searches, but there is not. B+Tree as well as most other indexing structure used for traditional RDBMS, are for exact matches and range queries not involving similarity of many attributes. Simply put, B+Tree is not well suited for high dimensionality similarity searches because it relies on the ordering of the attributes contained within the index.

Here the attributes are organized as a single dimension with each branch containing a distinct bound on the single dimension (see figure 1). For high dimensional data we are not concerned with ordering, but rather similarity of one object with another on multiple attributes (high dimensional) using some form of similarity computation such as a distance measure.

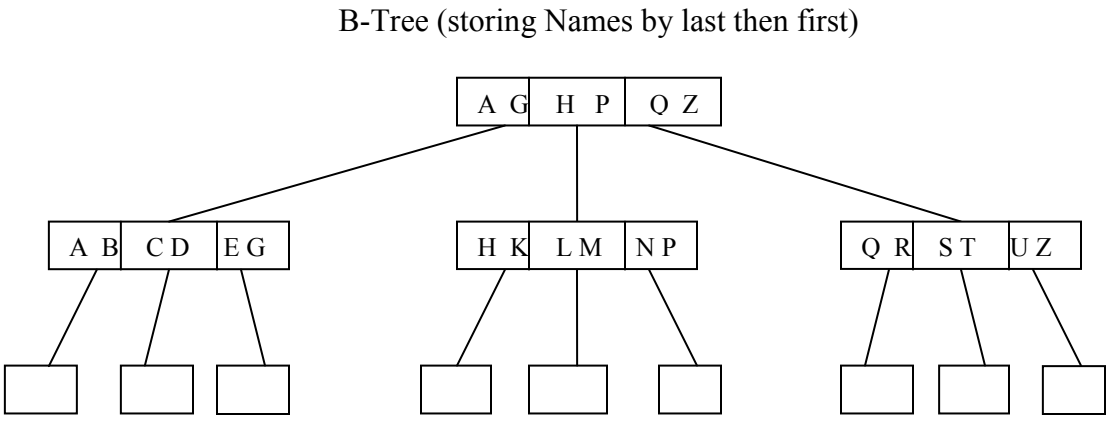



Figure 1. B+Tree structure, 3 levels each with 3 branches

Some may suggest the use of multiple B+Tree indexing a sub-set of the attributes, but this only complicates the issue. Rather than searching one index now, one has to search multiple indexes retrieving a group of objects based on a single dimension per B+Tree, yet the query is looking for similarity across multiple dimensions based on a distance measure. This sub-set may be closer in one dimension but not in another. Therefore, we may have a large sub-set (sum of all sets returned by the multiple B-Trees searches) with many points that may not be nearest neighbors when considering the entire set of attributes, here the emphasis is on similarity with all the attributes and a query point. The problem here can be loosely defined as follows, given a large set of objects, in

a high dimensional space, find NN or KNN (where K > 1) object(s) within the search space, to a given query object [31, 43] without having to compute the similarity (based on distance measure) between the query object and every object in the search space. What is needed is an indexing method that facilitates this type of search, which reduces the computation complexity of reading every object and computing the similarity based on all attributes.

Currently many high dimensional indexing methods exist, the most notable of which is the R-Tree [44]. It should be noted that this method is used in commercial relational database management systems (RDBMS), for example Oracle [51, 52], MS SQL Server [105], IBM Informix [45] as well as open source DB such as MySQL [73] and PostgreSQL [81] all use R-Tree. This data structure basically extends a B+Tree to include multiple dimensions, minimum bounding rectangles (MBR) and the addition of a similarity computation. In turn this tree structure can handle any number of points and a large number of dimensions, unlike the standard B+Tree which basically works by hieratically partitioning the objects based solely on the attributes arranged as a single dimension in the index.

The major drawback to R-Tree, as with other high dimensional indexing methods, is dimensionality. As the number of dimensions grows the ability to hieratically partition the objects becomes difficult if not impossible. This is caused by an unavoidable situation that occurs with the R-Tree whereby the minimum bounding rectangles overlap each other in one or more dimensions (boundaries are no longer distinct as they are in B+Trree) , details on this will be covered in chapter 2. Therefore, during a similarity search we have to traverse several branches covered by minimum bounding rectangles to

resolve a NN query, this is because a given object's NN could be in any MBR. Researchers have given this problem a name "The Curse of Dimensionality" [15] and have demonstrated that R-Tree, as well as most other high dimensional indexing methods degrades to no better than a sequential scan as the number of dimension grows, this will be discussed further in chapter 2.

Over the years researchers have struggled with this problem. They have attempted to solve it by using a whole array of indexing structures, and have even turned to alternate methods that do not rely on any indexing data structure. This other methods have had limited success when compared to a sequential scan. As yet no one has actually developed an efficient reliable method to effectively search high dimensional space. It has come a point where some researchers have all but given up. Some have even published paper indicating it can never be truly solved [41], yet every year several new papers are published by researchers who look at the problem from a different angle sometimes with fresh ideas [3, 4, 43, 48, 67, 68]. Still there is a real, ever growing need to solve this problem, as many fields of study would benefit from a solution.

Recently some have proposed a shift in the area of high dimensional similarity searching. It has been suggested that exact NN and KNN search may not always be relevant and that an approximant result may be sufficient [7, 13, 30, 39, 48, 56, 61, 63]. Thereby one may choose to sacrifice accuracy for efficiency and yet still being able to reliably use the results. Some of the current approximant nearest neighbor (ANN) methods throw out the concept of any data structure completely, and attempt to solve this problem by other techniques. One method for high dimensional ANN getting a lot of attention is Locality Sensitive Hashing (LSH) [3, 4, 30, 37, 43, 48, 67] and Locality

Sensitive Hashing B-Tree (LSB-Forest) [111, 112], which dismiss the notion of a data structure replacing it with hash functions. With this in mind some researchers have refocused their attention on ANN similarity search in high dimensional space, by applying heuristics to existing index structures [66], developing alternate methods that provide only approximant results. This new attention has renewed some of the research into index structures, such as R-Tree, that others have abandoned.

The main focus of my dissertation is on efficiency of high dimensional indexing with an emphasis on an index structure known as R-Tree. I will present my first work on construction cost that is improving the input/output (IO) cost of initially building an index from a data set. Then I will turn my attention to NN and ANN query efficiency attempting to maintain accuracy.

I started with research into index disk IO efficiency using Rdnn-Tree (modified version of R-Tree used to answer reverse nearest neighbor queries). This first involved implemented persistence storage for Rdnn-Tree then different buffer management methods, to determine which performed best for this index structure. The results from this led me to ask if I could improve the insertion efficiency by means of bulk loading the index. Rather than inserting points one at a time as normally done, I employed the use of an in-memory Rdnn-Tree built with small sets of the initial data. Points in the leaf nodes from in-memory Rdnn-Tree were than bulk loaded as a group with the thought that because they were in a leaf node together they would follow the same insertion path in the persistent Rdnn-Tree. This led to an improvement of 90% over single point insertion for Rdnn-Tree by use of an in-memory Rdnn-Tree built from a sub-set of the initial data. This method of bulk loading was shown to work very well and was applied to R-Tree,

although the improvement would not be as dramatic. I have dedicated Chapter 4 of my dissertation is to this particular work.

As already discussed high dimensional searching is a very difficult problem, as the number of dimensions increases most indexing methods degrade in term of efficiency to no better than a sequential scan. The problem stems from the fact that during a NN search we don't know whether the solution found so far is the exact NN, additional searching is really verifying that the remaining un-search space does not have a better solution (NN). One of the advantages of R-Tree is that it can handle multiple dimensions very easily, which means it handles high dimensional data. I decided to look at efficiency and accuracy of NN and later ANN queries using R-Tree indexing of low to high dimensional data. This work involved the application of a number of pruning heuristics applied to the search algorithm which attempts to improve efficiency of exact NN. My work involving the using pruning heuristics showed promise, in that I could reduce the amount of searching needed but could no longer maintain a guarantee on finding an exact NN. Basically there was a trade off in the amount of work (IO efficiency reading data from presenting storage) and query results, or to put simply, less IO returned lower accuracy in the results, yet as some have pointed out an ANN may be sufficient. With this in mind I focused on good IO efficiency with an attempt to maximize accuracy for ANN, no longer attempting to find exact NN for high dimensional data. This proved very promising as several of the tested heuristics have been shown to give better than expected results for ANN with very limited IO. My work on this, presented in chapter 5 shows that the heuristics worked for different dimensionality, data set sizes and

distributions without relying on any preprocessing or knowledge about the data, and returned accurate NN and ANN with limited IO.

While working on the pruning heuristics I started to reconsider the R-Trees strengths and weaknesses, first strengths is its ability to partition the space and apply branch and bound search, yet as the dimensionality of the data increases this strength, becomes its weakness. The question was asked "What if I use multiple R-Trees", each containing a sub-set of the data, disjointed on a limited number dimension, yet when taken as a whole covers the entire search space. My approach would be known as a forest of R-Tree or R-Forest for short. With this forest I can select a sub-set of data (R-Tree) to initially search and then select additional sub-sets of data (R-Trees) in an attempt to refine the ANN results. It is also possible to eliminate entire sub-sets (R-Trees) which would not contain an ANN. I will show that splitting the domain space on a relatively small number of dimensions using R-Forest yields better results than existing implementations for ANN queries, and I am able to implement an additional pruning heuristic with the ability to provide some feedback on the results returned. Details on R-Forest will be provided in chapter 6 with my results shown in chapter 7.

My research has focused on index structures for nearest neighbor queries in high dimensional space, it shares a common them with newer research in the area of "Big Data" which can be comprised of large volumes of structured or un-structured data [49]. This field which has been getting a lot of attention in the past few years, also involves large data sets often comprised of terabytes of data and beyond, which requires new file structure and storage methods, as exist systems do not handle gigabyte size files very well, much less terabyte size files. Given the large data sizes involved, research has

focused on distributed files systems, the best example is Hadoop [16], which is an open source distributed files system often used in cloud computing built on top MapReduce developed by Google [49, 101], used to process web pages daily. These methods address the processing, storage requirements, as well as fault tolerance (hardware failure), and parallel access using clustering and distributed files systems. One area that is being addressed, but may require additional research is indexing of large distributed file systems such as Hadoop, and the data produced by systems such as MapReduce [101, 102]. My research may not be directly applicable to "Big Data" in the since that it focused on NN and ANN, but indirectly has given me insight as to the problems of dealing with large complex data sets and how to approach indexing large files system.

The rest of this dissertation is organized as follows; Chapter 2, covers past and present research into different methods of querying high dimensional data. Chapter 3, my implementation as well as experimental procedures used throughout my research. Chapter 4 is a discussion on the application of my past work on bulk loading for R-Tree index maintenance. Chapter 5 my work on a number of heuristics which have been implemented and tested for ANN. Chapter 6, R-Forest implementation and querying for ANN. Chapter 7 discusses of a number of experiments with data distribution, dimensionality scaling and point scaling using R-Forest and LSB-Forest. Chapter 8 talks about some future work and direction.

## 2. Background

Research into high dimensional indexing has taken three different paths, the first has focused on solutions involving indexes based on data structures, whereas the second line of thought has been on solutions involving grid based indexes and the third being based on hash functions. Not only is this research following different paths, the search for a NN in high dimension space has moved toward finding an approximate solution. The following sections will present some of the predominant methods that have been studied.

### 2.1 Types of Searches

Typically high dimensional data is comprised of a very large set of objects. With each object being compressed of a large number of attributes which represent features of the data, all of which would be organized into a database. Users of this database are usually looking to perform specific types of queries (searches) such as NN, KNN, ANN or range queries even approximate range queries, [8] just to list a few. These types of queries don't involve searching for the existence of a specific object in the database but rather finding which objects are similar based on some form of similarity measure. All of the following queries involve a distance calculation as the method to determine similarity. For this research the focus is on the use of Euclidean distance, yet other distance measures such as Manhattan distance could be used, all one needs to do is modify the distance calculation used between points and MBR. In most cases Euclidean distance is used and can be formally defined by the following equation 1.

$$dist\ (q, p) = \sqrt{\sum_{i=1}^{d} (q_i - p_i)^2} \qquad (1)$$

### 2.1.1 Range Queries

A range query is a request for a single point or several points that fall within a user supplied range. This type of query can be defined by the following equation 2.

$$dist\ (Q,P) \leq R \qquad (2)$$

Where Q is the given query point and R is maximum distance as measured by some distance measure. Any point P who's distance from Q is less then or equal to R is considered solution to the query.

### 2.1.2 Nearest Neighbor Queries

As discussed earlier there is a need to search high dimensional space efficiently, with the predominant type of query being Nearest Neighbor search (NN). For this type of query the focus is with the similarity based solely on distance, not with the relationship between dimensions or importance of any one or more dimension, for the sake of this research, all dimensions are treated equally even though they may not be. Given a data set *s* of *d* dimensions with *n* points and a query point *q*, which point in *s* is nearest to *q* based on a distance measure. This definition for NN can be extended to define a variation known as KNN where by the goal is to find the K nearest neighbors (KNN) in some data set. That is a set of points in *p* of size K such that there are no other points in *p*

closer then this set. Formally defined in figure 2, also note that this definition also defines NN when K = 1:

Definition ($K^{th}$ Nearest Neighbor)

Given a set *S* of points in some *d* dimensional space and a query point *q*, the *$K^{th}$ Nearest Neighbor* of *q* (denoted $KNN_S(q)$) is a subset of S of size K defined as

$$KNNs(q) = \{p \in S \mid \forall p \in S \mid p \in KNN \mid p' \in S - KNN : D(q,p) \leq D(q,p')\}$$

where *D(q,p)* denote the distance between point q and p

Figure 2. Definition of K Nearest Neighbor

### 2.1.3 Approximate Nearest Neighbor

Another variation on nearest neighbor search is the Approximate Nearest Neighbor (ANN), this can be defined as a relaxation of the requirement that the exact nearest neighbor be found. This can be done in the definition for KNN (figure 2) by relaxing the requirement that a solution be the one and only. An approximate nearest neighbor search allows for some margin of error in that, of the set returned, there could be additional points that are closer but where not discovered during the search. These undiscovered points are usually defined as approximation within some value known as Epsilon, as defined in Figure 3.

Definition (Approximate Nearest Neighbor)

Given a set *S* of points in some *d* dimensional space and a query point *q*, the *Approximate Nearest Neighbor* of *q* (denoted $ANN_S(q)$) is a subset of *S* defined as $ANNs(q) = \{p' \in S \mid \forall p \in S : D(q,p) \leq (1+\varepsilon)D(q,p')\}$ where *D(q,p)* denote the distance between point q and p

Figure 3. Definition of Approximate Nearest Neighbor

### 2.1.4 Reverse Nearest Neighbor

Another type of search involves the reverse of the typical nearest neighbor search, known as reverse nearest neighbor (RNN). This type is defined as the opposite of NN in that given a query point *q* which point(s) in the set *s* have *q* as their NN formally defined in figure 4 as follows and illustrated in figure 5.

Definition (Reverse Nearest Neighbor)

Given a set *S* of points in some *d* dimensional space and a query point *q*, the *Reverse Nearest Neighbor* of *q* (denoted $RNN_S(q)$) is a subset of *S* defined as

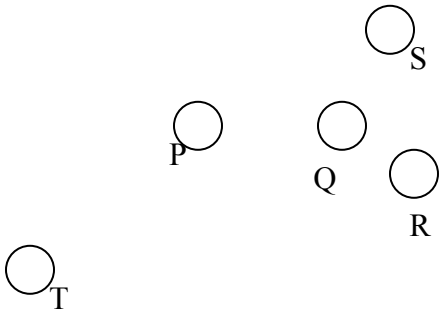
$RNNs(q) = \{r \in S \mid \forall p \in S : D(q,r) \leq D(r,p)\}$

where *D(p,q)* denote the distance between point *p* and *q*

Figure 4. Definition of Reverse Nearest Neighbor

Figure 5, shows an example for reverse nearest neighbors. Notice the contrast to the nearest neighbor, a point Q does not necessary have reverse nearest neighbors – or it

can have multiple reverse nearest neighbors. For instance, in 2-D Euclidean space, a point may have up to six reverse nearest neighbors.




| Point | Nearest Neighbor | Reverse Nearest Neighbor |
|---|---|---|
| P | Q | T |
| Q | R | P, R, S |
| R | Q | Q |
| S | Q | $\phi$ |
| T | P | $\phi$ |

Figure 5. Illustration of Reverse Nearest Neighbors

## 2.2 Search Methods for High Dimensional Data

Over the past 20 years many researchers have proposed, developed, tested, analyzed and presented results for different data structures in an attempt to solve the problem of high dimensional searching. Some have built on other's work by suggesting improvements, whereas others have taken a totally different approach in an attempt to solve this problem.

### 2.2.1 Indexing Structures

The following section describes some of the more predominate index data structures used to search low dimensional data. Some researchers have attempted to apply or extend some of these to NN searches in high dimensional data.

#### 2.2.1.1 R-Tree

The R-Tree proposed by Guttman [44] and later modified by other researchers including Roussopoulos [77, 84], is one of the primary data structures used for high dimensional indexes. An R-Tree is a balanced tree, with a root node at the top. The root is comprised of a number of branches each containing a minimum bounding rectangle (MBR) bounding all dimension and a pointer to the sub-tree bound by the MBR. The number of MBR is governed by the number of dimensions and amount of storage allocated for a node (size of a node in terms of bytes will be discussed in chapter 3). Each sub-tree is comprised of the same structure with the exception of the leaf nodes which stores the data points and a reference pointer to the actual data. Insertion involves choosing the branch starting at the root that minimizes overlapping with other branches. When a node is full it is split in such a way as to evenly distribute points into two nodes. The research White suggested splitting of nodes could be improved with the use of VAMSplit algorithm which he proposed in his paper [107]. He pointed out that VAMSplit would maintain the balance of the tree, should reduce storage and improve performance R-Tree. During a split most and sometimes all of the MBR traversed during the insertion will have to be updated as their dimensions (MBR) may have changed. Deletion involves, traversing the tree to locate the point, deleting it from the leaf, and updating the MBR as needed for each branch traversed. This basic data structure can be used to handle any number of data points having a larger number of dimensions (within a set all points must have the same number of dimensions).

Performing a NN query in R-Tree can be handled using a condition derived from query points's current best solution (initial ANN point found) and the distance from the

query point to a MBR. This method works by examining the minimum distance from Q to the closest edge of the MBR known as *Mindist (Q, MBR)* defined by [77, 84], if a point's current best solution is less then Mindist then there cannot be a solution in the MBR see formula 3 and figure 6.

$$Mindist(q,R) = \sqrt{\sum_{i=1}^{d} x_i^{\,2}}\ ,\ \mathrm{x_i} = \begin{cases} R_{low_i} - q_i & q_i < R_{low_i} \\ q_i - R_{high_i} & \text{if } q_i > R_{high_i} \\ 0 & otherwis \end{cases} \qquad (3)$$

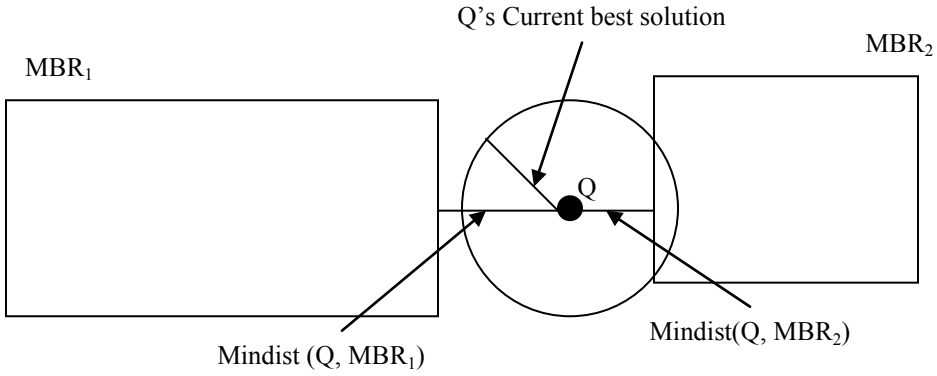



Figure 6 Illustration of Mindist and MBR pruning.

In figure 6, Q's Mindist (Q, $MBR_1$) is greater than Q's current best, therefore $MBR_1$ cannot contain a better solution, whereas Q's Mindist to $MBR_2$ is less then Q's current best solution, therefore $MBR_2$ could contain a better solution (this is further discussed in chapter 5).

Many variations exist like R+Tree developed by Sellis [87] and the dynamic R*Tree by Beckmann [9, 10]. Others continue to extend R-Tree by added additional

information to improve its efficiency at the leaf level [28], for example Mehdi developed VoR-Tree which they describe as R-Tree with Voronoi diagram [70].

#### 2.2.1.2 SS-Tree

One variation on the R-Tree is the SS-Tree developed by White [106], and extended by Kurnaiwati [62]. This structure is basically the same as R-Tree only that the MBR is replaced by a hyper-sphere, with the bounding region now based on a centroid (mean value of the child vectors) and a radius. Internal nodes store bounding regions bound by a radius rather bounding each dimension. This is illustrated by figure 7 below.

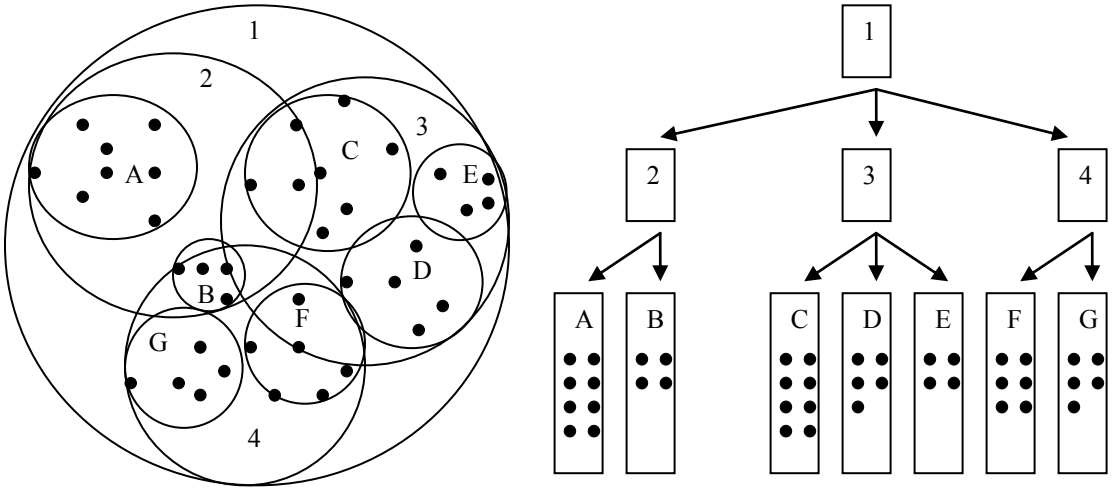



Figure 7. SS-Tree bounding regions.

Data points in SS-Tree are stored in the leaf nodes just as they are in R-Tree. Insertion proceeds in a similar manner to that of R-Tree with node splitting also choosing a dimension with highest variances. Deletions proceed by locating the point, removing it

from the leaf node, then adjusting the internal nodes traversed. Like the R-Tree the SS-Tree can handle high dimension data, having the same constraint on the nodes size based on the number of dimension of the centroid. One major advantage SS-Tree has over R-Tree concerns the storage of internal node. SS-Tree does not use a MBR, therefore internal nodes require less storage, thus a higher fan out can be achieved. One major difference between SS-Tree and R-Tree is in the name SS, which stands for similarity search, this implies a major difference in the two. The authors focused more on similarity search rather than on nearest neighbor. Their goal was to build a structure and search algorithm that met their needs, they were more concerned with similarity and in some cases dissimilarity. Hence, they made assumptions about the domain of the date which would be provided by an expert, who could provide domain knowledge used to refine the similarity search. This led to the use of a weighted Euclidean distance measure see formula 4.

$$D(\mathbf{x},\mathbf{y}) = \sqrt{(\mathbf{x}-\mathbf{y})^T \mathrm{diag}(\mathbf{w})(\mathbf{x}-\mathbf{y})} \qquad (4)$$

Here the authors have included weights on the feature vectors attributes based on the domain knowledge provided by an export. There is a second advantage to this structure, in that a variance is included that can be used to control the accuracy and similarity search. If the variance is zero then an exact search will be performed, the value of variance controls the level of similarity that will be accepted, hence this structure can also perform approximate searches.

#### 2.2.1.3 SR-Tree

Further work on both the R-Tree and SS-Tree involves a hybrid data structure which combines the best of both into one tree. This is known as the SR-Tree which uses both MBR regions and hyper-spheres bounding from both the R-Tree and SS-Tree respectively. Its focus is on the intersection of the two regions. This can best be seen in figure 8, where the intersection of a MBR and hyper-sphere are used as the bounding region [53].

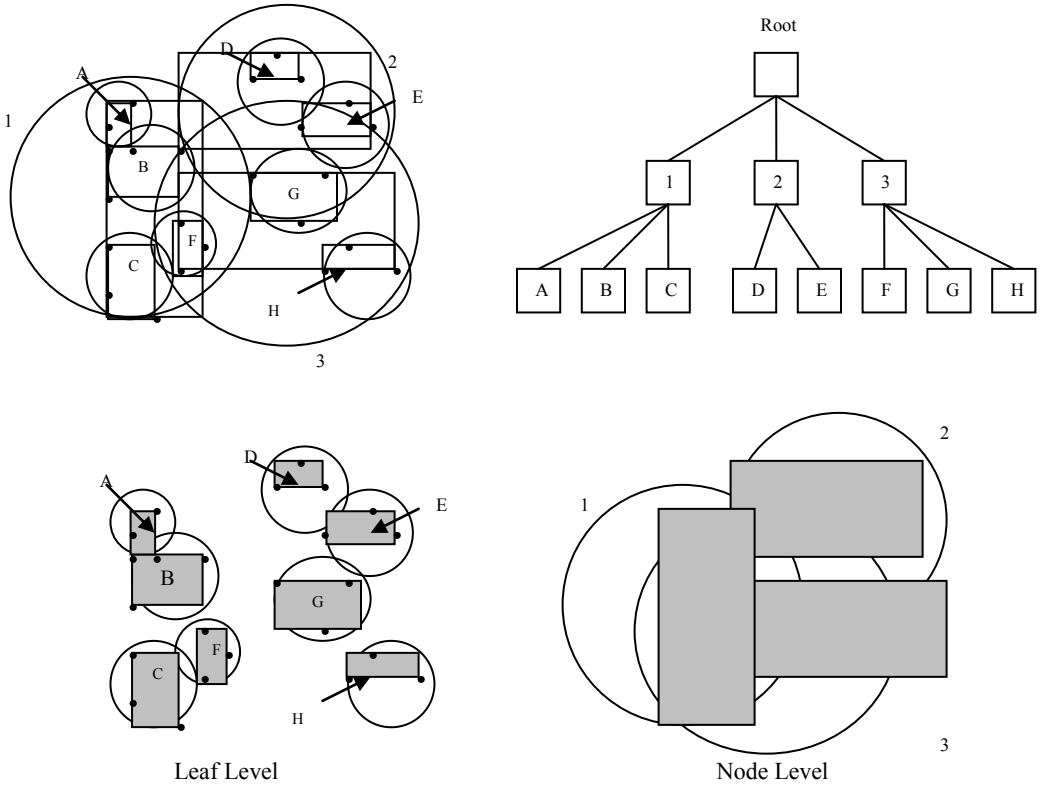



Figure 8. Showing intersection of MBR and hyper-sphere.

Insertion of new points and deletion of existing points are handled much the same way as other tree based data structures. One difference between the SR-Tree and other

tree based data structures has to do with storage requirements, now both the MBR and hyper-sphere bounding regions must be stored as well as updated during both insertion and deletion. As with all index structures the height and branch fan out is constrained by the size of a node and the number of dimensions.

#### 2.2.1.4 X-Tree

Some researchers have focused on attempts to improve existing index structures case in point is the X-Tree, which stands for eXtended node tree [17]. This is a hybrid R-Tree which attempts to deal with one of the main problems of boundary based indexes, the problem of overlapping bounding regions. The researchers give a good argument that by minimizing the bounding regions overlap should improved search performance. The author's main focus was on the selection of dimensions to split in an attempt to minimize or prevent overlapping which as they pointed out leads to a degradation of the indexes performance. They extended the (hence the name) R-Tree index by adding what they referred to as super nodes which contain information about node split history as an attempt to better handle high dimensionality. This history is accessed when a node has been filled and must be split; it is used to determine the best dimension to split. Of course this additional information comes at a price requiring additional storage of super nodes within the index structure.

#### 2.2.1.5 Generalized Search Tree

Other researchers have focused on building generalized indexing methods which can be used in a number of applications. The best example of this is GiST, known as the

Generalized Search Tree, which is implemented as a non-specific search tree. It can handle B-Tree, R-Tree and RD-Trees, by simply modifying not the structure but only the data objects stored and the search algorithm. Hence one does not have to have multiple indexing methods for different data sets [5, 40].

#### 2.2.1.6 Addition Search Trees

As with any other research topic, this topic also has many not as well known solutions usually developed to solve a specific version of the problem unique to a particular researcher. An example of this includes the LB-Tree (Lower Bound Tree) that tries to minimize searching by apply clustering techniques to the internal nodes in an attempt to maintain a tight lower bond on mean distances [21]. Others attempt to organize the data into buckets known as hyper-rectangle buckets based on dimensions for details see Bently's KD-Tree [11, 38]. Some have taken the approach of focusing on partitioning of the data using a rotary binary hyper-plane known as the $BM^{+}$-Tree [117] which is an extension of the $M^{+}$-Tree proposed by Zhou [116]. The VP-Tree proposed by Uhlmann [99] attempted to partition the data with respect to vantage points which are used as separators to partition the data into two balanced subsets. This list is by no means complete, as many variations exist which extend or provide slight modifications to these just presented.

### 2.2.2 Grid Based Indexes

As with most research in computer science alternative methods are sometimes developed either to satisfy a specific need or because researchers feel alternate methods

may perform better. With this in mind a number of researchers have embarked on different methods for solving this problem. These methods differ in several aspects from the methods listed in previous section. Rather than attempting to organize the data into a hieratical data structure these methods attempt to organize the data into a grid to facilitate improved searching. In turn these methods attempt to do away with concepts of a hieratical structure and bounding regions. The following sections give details on some of the better known grid space search methods.

#### 2.2.2.1 VA-File

One of the first proposed alternatives to tree based indexes was by Weber and his colleagues, known as VA-File [35, 104]. The authors presented arguments that any index structure which had to access more than 20% of the data (index) would be no better than a sequential scan given the extra overhead of the index structure and cost of random access. Therefore they proposed a relatively small approximation file that would be built from the original data, then sequentially scanned in much less time then scanning the original data file. This file known as the Vector Approximation File (VA-File) divides the data space into $2^b$ rectangular cells, b is given by the user and defines the number of bits used to represent the approximation of the cell for each dimension see figure 9.

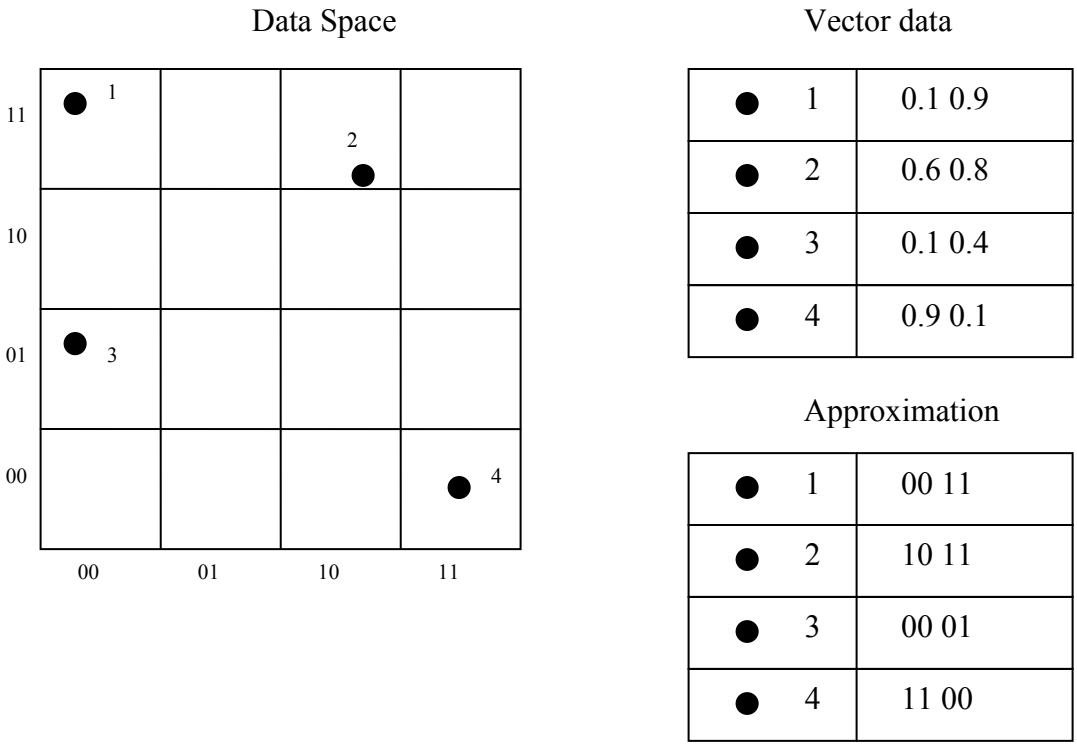



Figure 9. 2D data space representation of VA-File

The VA-File need only store the boundary information as well as the bit approximations for the data. Inserting and deleting become trivial in that points can either be added or removed from the VA-File. A search involves a sequential scan of the VA-File returning a set of approximation points that represents the possible solutions. Finally the original data must be read via random access to refine the approximations into the final nearest neighbor.

#### 2.2.2.2 M-Grid

A variation developed by Digout [31] known as M-Grid, is built on the VA-File concept. This method uses the concept of triangular inequality [18, 36, 110] which can

be used to prune data points by means of a set of reference points known as pivot points. Triangular inequality basically says that given a pivot point and the distance to all data points in a data set, one can prune (eliminate) points that don't fall within a given range. This can best be seen in figure 10.

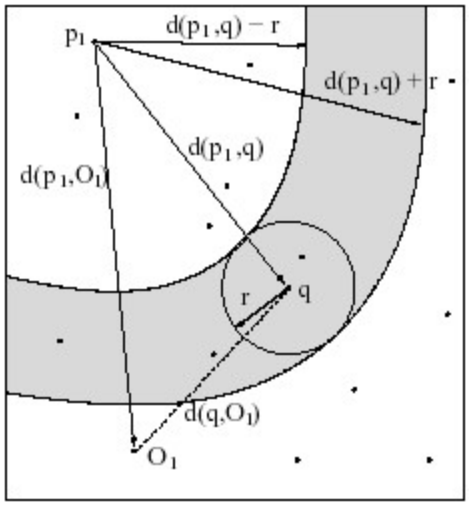



Figure 10. Pruning using triangle inequality

Here the pivot point P1 is given as well as distances to all other points. The distance from P1 to the query point Q is computed, a range R is also given, based on triangular inequality any point that falls out of the *Distance (P1,Q) + R* and *Distance (P1,Q) – R* cannot have a distance to Q less than R. With this concept in mind Digout devised a new method known as M-Grid, which first clustered the data set using any efficient data clustering method then computes and stores the distance from a given set of pivot points to all points in the data set. Finally a set of rings from each pivot point are defined and stored, each of which houses an equal number of objects (data points). The

intersection of these rings form grids or “cells” hence the name M-Grid. This entire process can be summed up in figure 11.

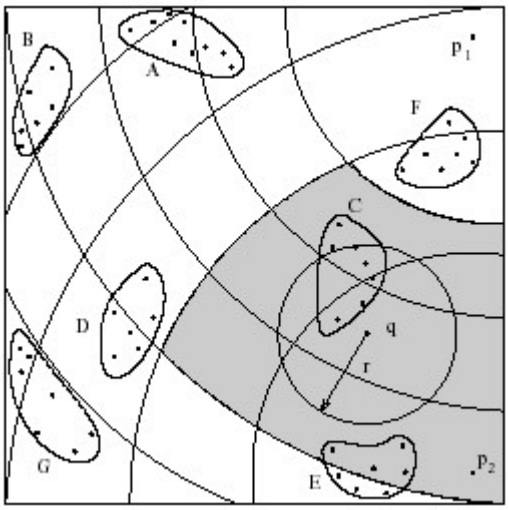



Figure 11. Illustrating a typical M-Grid implementation.

Based on Figure 11 one can easily see that clusters A,B,F,G,D cannot contain nearest neighbors to Q bound by region R and only clusters C and E need to be searched. The computed distances and pivot points as well as cluster information, is stored in a data file which in turn becomes the index for the data set. This method like other methods is dynamic such that insertion and deletion of points is relatively easy to implement. The authors suggest that their method is faster than VA-File which is in turn faster than a sequential scan, but the performance is dependent on the number of pivot points, number of rings for each pivot point, and the number of clusters, size and distance between the clusters.

#### 2.2.2.3 Addition NN and ANN Non-Data Structure Methods

Other researchers have embarked on alternate methods by proposing, implementing and testing different concepts in an attempt to solve NN, KNN, or ANN. On notable idea by Cui [29], involves bit mapping to represent each dimension. This method which the authors called Bit-Difference (BID), assigns a single bit to each dimension, with each object now being represented by this mapping. An object's attributes (dimensions) are mapped to 1 or 0 depending on how the value of the attributes falls in the full domain for the dimension. For example if a dimension domain is {0,1} then an attribute of 0.7 would be mapped to 1. This reduces all the attributes to single bits, which are organized into a structure. A query point is mapped the same way, using this new representation, than it is queried against the representation of the data set. The authors employed addition modification such as clustering, and dimensional weights to further improve their method. They pointed out this method appears to be very similar to VA-File but has several advantages in terms of computation and disk IO cost. Another method proposed by Volmer [103] known as Buoy Indexing, attempts to index high dimensional data by clustering methods. The data is clustered using any existing methods, next the median point for each cluster is located and stored, this becomes the buoy point (hence the name) also stored is the radius from the buoy point to the furthest point in the cluster in turn defining a hyper-sphere. A query proceeds by computing the distance to the buoy points, then ordering the clusters based on this distance. A parameter is used to control the number of clusters that are searched for ANN, this in turn controls the depth of the search of distance ordered clusters as well as the level of accuracy. The cluster's hyper-sphere can provide additional optimization if the query point's current

ANN is closer then the hyper-sphere of an unsearched cluster there is no reason to search that cluster. Volmer was able to show that his method worked reasonably well for his specific application of searching multimedia databases [103].

Others have continued to build on past research in attempts to improve the results or combine different techniques in an attempt to provide better results. For example, Ciaccia and Patella [26] extended the M-Tree with the addition of an algorithm known as PAC-NN (probably approximately correct nearest neighbor), which provides M-Tree with a search stopping criteria based on the distance distribution of the points. The authors showed there is no need to search nodes where the query point's current solutions overlapping volume is to small to contain a point. They were able to exploit the decrease in point density as the dimensionality increases. This was done by modifying C-NN (correct nearest neighbor) [14], by adding new parameters used to determine the probability that a branch will not have any additional NN because the overlap volume (volume of overlap between a query point's current solution and a nodes hyper-sphere) is relativity small and unlikely to contain a point. In turn their algorithm cannot guarantee an exact NN, rather it can only return ANN within some level of accuracy, based on the stopping parameters.

## 2.3 Curse of Dimensionality

Past research has shown that an index's performance degrades as the number of dimensions increases [104]. For the type of queries studied here (NN using R-Tree) this is also the case. Even for other index structures that attempt to solve the same problem (NN in high dimensional space) the same happens. This can best be seen in figure 11,

which shows the number of nodes accessed for nearest neighbor query using a SRTree. The query asks for the KNN with k equal to 100 from a data set of 70,000 points. The figures 12, 13, show that the number of nodes accessed grows exponentially with the number of dimensions. Typically the entire tree is read when the dimensionality reached 32 and above. Even if the data is clustered which should help the index structure, eventually the "Curse of Dimensionality" catches up with the data structure.

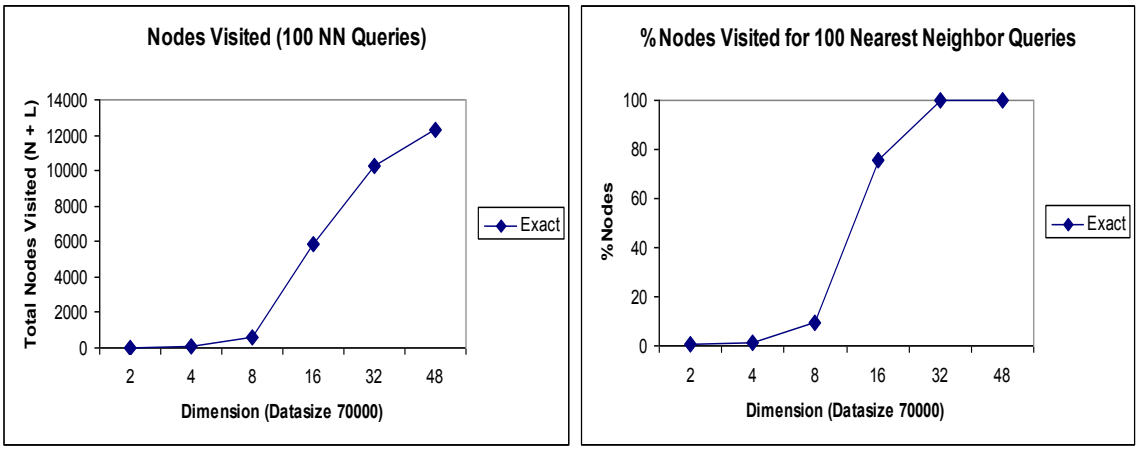



Figure 12, 13. Nearest neighbor query performance of SR-Tree

The paper by Böhm [15] contains a thorough discussion of the causes of this curse. The authors divided the causes into three categories:

- Geometric properties of high dimensional space. When the dimensionality increases, the volume of a sphere/rectangle (with fixed radius/length) increases exponentially. Consequently, the data in high dimensionality is more sparsely distributed.
- Index organization. Most indexes work by partitioning the data along different dimensions. Typically, a region for a leaf node is formed by splitting it *l* times,

where $l$ is the height of the tree. However, in a high dimensional index, $d >> l$, many dimensions are not split at all.

- Query properties. As a result of the geometric properties of the space, for any query point q, its nearest neighbors are farther apart from q when dimensionality increases. This, in turn, drives up the volume of the space to be searched (one has to examine the space between a point and its nearest neighbor to ensure there is nothing closer).

The above explanation for the "Curse of Dimensionality" provided by Böhm can easily be applied to R-Tree. All of the above cases have been verified in this research. Take the second property, index organization, in a typical R-Tree index with a height of 5 only 3 to 4 dimensions are split, this was verified by looking at the individual dimensions of a hyper-rectangle defining a single leaf node at the bottom level of the R-Tree. It was found that nearly all of the dimensions were close to full range of the entire data space as defined by the entire set of points except for the few that were split as the tree was built. It should be noted that even a split dimension can grow as new points are inserted. This is easily seen in table 1. This table shows the percentage of overlap for the 32 dimensions of a randomly select MBR within an R-Tree. The table lists the domain of each dimension and the ratio of overlap of the MBR with the domain of the dimension. It illustrates one of the fundamental problems of R-Tree with high dimensional data.

Table 1. 32 Dimensional MBR for a leaf selected at random for an R-Tree (data set contained 100,000 points with a uniform data distribution).

| Dim | MBR Lo | MBR Hi | Data Set Min (Root) | Data Set Max (Root) | Percent of Overlap | Split(Y/N) |
|---|---|---|---|---|---|---|
| **0** | 0.011590 | 0.996857 | 0.000019 | 0.999999 | 98.53% | |
| **1** | 0.060741 | 0.865796 | 0.000004 | 1.000000 | 80.51% | Y |
| **2** | 0.027376 | 0.727943 | 0.000004 | 0.999998 | 70.06% | |
| **3** | 0.222539 | 0.960931 | 0.000042 | 0.999978 | 73.84% | |
| **4** | 0.118394 | 0.956759 | 0.000002 | 0.999997 | 83.84% | |
| **5** | 0.063024 | 0.983117 | 0.000004 | 0.999983 | 92.01% | |
| **6** | 0.144777 | 0.964902 | 0.000004 | 0.999986 | 82.01% | |
| **7** | 0.000976 | 0.967537 | 0.000004 | 0.999987 | 96.66% | |
| **8** | 0.333247 | 0.997902 | 0.000001 | 0.999987 | 66.47% | |
| **9** | 0.062988 | 0.947717 | 0.000001 | 0.999992 | 88.47% | |
| **10** | 0.024613 | 0.986062 | 0.000004 | 0.999995 | 96.15% | |
| **11** | 0.017236 | 0.997224 | 0.000007 | 0.999977 | 98.00% | |
| **12** | 0.061829 | 0.914414 | 0.000000 | 0.999973 | 85.26% | |
| **13** | 0.060277 | 0.973872 | 0.000006 | 0.999987 | 91.36% | |
| **14** | 0.004876 | 0.803485 | 0.000005 | 0.999994 | 79.86% | |
| **15** | 0.007752 | 0.988437 | 0.000001 | 0.999951 | 98.07% | |
| **16** | 0.045223 | 0.870849 | 0.000026 | 0.999995 | 82.57% | |
| **17** | 0.446516 | 0.955342 | 0.000007 | 0.999988 | 50.88% | Y |
| **18** | 0.000699 | 0.933230 | 0.000025 | 0.999992 | 93.26% | |
| **19** | 0.021558 | 0.999661 | 0.000009 | 0.999999 | 97.81% | |
| **20** | 0.011833 | 0.985059 | 0.000004 | 0.999994 | 97.32% | |
| **21** | 0.000378 | 0.797043 | 0.000008 | 0.999999 | 79.67% | |
| **22** | 0.052105 | 0.999893 | 0.000006 | 0.999996 | 94.78% | |
| **23** | 0.011760 | 0.995374 | 0.000005 | 0.999991 | 98.36% | |
| **24** | 0.176440 | 0.912952 | 0.000000 | 0.999999 | 73.65% | |
| **25** | 0.046933 | 0.995643 | 0.000002 | 0.999988 | 94.87% | |
| **26** | 0.009195 | 0.999200 | 0.000002 | 0.999997 | 99.00% | |
| **27** | 0.003163 | 0.877452 | 0.000020 | 0.999999 | 87.43% | |
| **28** | 0.002473 | 0.865191 | 0.000002 | 0.999994 | 86.27% | |
| **29** | 0.058407 | 0.753878 | 0.000006 | 0.999999 | 69.55% | |
| **30** | 0.549941 | 0.999413 | 0.000005 | 0.999999 | 44.95% | Y |
| **31** | 0.014874 | 0.966914 | 0.000024 | 0.999991 | 95.21% | |

The columns MBR Lo and MBR Hi define the MBR for the branch containing a single leaf node with 23 data points. The columns "Data Set Min (Root)" and "Data Set Max (Root)" define the entire bounding space for the full data set also known as the Root MBR. The percent of overlap column is the ratio of root MBR to leaf MBR for each dimension. Finally the Split column indicates which dimensions have been split, meaning which columns don't cover the entire data space for that dimension. From table 1, one can see that most dimensions (all but 3) cover nearly the entire range of the data set. Only dimensions 1, 17 and 30 most likely have been split during the insertion process. This simple example illustrates the fundamental problem with high dimensional data.

This table also shows how the volume grows; take for example the first 16 dimensions and compute the volume of the hyper-rectangle then look at the volume of the entire 32 Dimensional hyper-rectangle. It is easy to see how the volume increases as the number of dimensions increase, this in turn illustrates the first property described by Böhm [15] and in turn the third property. Because of this growth in volume as the dimensionality increases most data structure such as R-Tree suffer from large overlap of MBR at the different levels.

## 2.4 Advantages of an Index

If one considers the "Curse of Dimensionality" described in the previous section, it might cause one to consider an index to be all but useless for high dimensional data. On the other hand if one takes into consideration the following advantages it is easy to

see the rational for continuing research into high dimensional indexes. The following is a list of advantages related to data structure based indexes:

1. Is generic in respect to the following:
    a. Number of dimensions
    b. Type of data (integer, floating point etc.)
    c. Size of data (number of points)
2. Can be used regardless on the data distribution.
3. Can be bulk loaded/inserted saving load time.
4. Insertion and deletion can be performed without requiring the total reconstruction of the index.
5. Search heuristic can be changed, modified or tuned (parameters adjusted) without requiring the complete reconstruction of the index.
6. Bounding region can be changed, which could require reconstruction of the index with only slight modification to the index storage.
7. All four searches discussed previously in section 2.1 can be implemented without changes to the index structure with the exception of RNN which requires only minor changes to the index structure [109].

## 2.5 Disadvantage of an Index

Of course a data structure used for indexing has overhead and some disadvantages, which are listed below:

1. The index structure (internal nodes) increase the space and time complexity.
2. Sometimes the index needs to be re-built to maintain efficiency when many points are inserted or deleted.

From this list of advantages one can make a good argument for using an index. Most of all it is the generality and lack of specialty that justifies the use of an index over other methods, for high dimensional data.

## 2.6 Approximate Nearest Neighbor Search

Because of the "Curse of Dimensionality" and the nature of the problem being solved many researchers have come to a conclusion that an exact nearest neighbor search need not always be found. In turn they have suggested that an approximate nearest neighbor (ANN) might be sufficient. When the number of dimensions is large and the data set is large, on the order of 100,000 points or more, with 32 dimensions or more, sometime on the order of hundreds, an ANN may be as relevant as an exact solution. In fact some have pointed out that an ANN also known as approximate similarity may provide a sufficient solution as to satisfy the user [47]. Others have suggested that a tradeoff between access time and approximate results, given that the approximate results may still be sufficient is worth it rather than having to wait for an exact result which in turn may not provide any more relevant information then the approximate results [27].

As an alternative Sakurai has developed a variation known A-Tree (approximation tree), which uses MBR and virtual bounding rectangles (approximate MBRs) in a tree structure, search this structure returns an approximate results which has been shown to outperform VA-File and SR-Tree [85].

### 2.6.1 Locality Sensitive Hashing

As with other proposes some researchers have ventured away from data structure based indexing solutions, in this case they have suggested a different approach known as Locality Sensitive Hashing (LSH) [43, 48]. This is an indexing technique based on random projections. Within LSH the projection is done on the Hamming space representation of the data, which corresponds to the combination of the unary representation of each dimension of the data. The method proposed here implements two levels each having its own hash function. The first maps the points into buckets based on the LSH function of a given size, the second level of hashing uses a standard hash function of a given size. Once the preprocessing is complete, all data points are placed into buckets according to a hash function, querying a new point becomes a simple process of applying the same hash functions (primary and second level) to the point. This returns an ANN with a guarantied computation complexity measured as disk access, with an error based on the parameters supplied during the pre-processing (primary and secondary hash table sizes). One drawback to this method is that it requires the data to be represented as a positive integer. The authors point out that a real number can be transformed into positive integers by shifting them with simple addition, and then transformed into integers by multiplying by a large enough integer. They point out that

this would lead to a small amount of rounding error and even gave an analysis as to the amount of error to expect.

A number of researchers have continued to explore this ANN method by expanding on the original work as well as applying it in a real world application. Work by, Lv [67] expands on the original LSH ANN method by attempting to improve its space efficiency using a method known as Multi-Probe LSH. This method expands LSH with the addition of a new concept known as multi-probe, which attempts to reduce the number of hash tables that have to be checked during a query. This method uses a derived probing sequence to determining the hash buckets with the highest probabilities of containing ANN. The authors showed that using multi-probe LSH fewer number of hash tables are required yet they were still able to maintain the quality of the query in comparison to standard LSH as well as another method known as Entropy-based LSH [78]. Multi-probe is very similar to a method proposed by Panigrahy [78] known as Entropy-based LSH, which uses sampling queries of a given distance to determine which hash buckets have the highest probabilities of containing ANN, thereby reducing the number of buckets that have to be searched. Recently Casey [19] has applied LSH to a real world research problem known as song intersection. While working on methods to determine if one song was derived from another, the authors used LSH in an attempt to find similarities in audio sequences. The authors pointed out that an ANN would be more then acceptable given the trade off required to find an exact NN which they suggested may not provide any more additional information given the nature of the problem.

### 2.6.2 Locality Sensitive Hashing B-Tree

In the past few years researches have focused on variations of LSH with the latest known as Locality Sensitive Hashing B-Tree. This latest method is based on the original proposal published by Indyk and Motwani in 1998 [48], which supports a variation on approximate NN similar to C-Approximate NN. This variation on ANN is defined as follows, a query point *q* has a point *o* who's distance to *q* is at most c times the distance from *q* to its exact NN *o*' where c is greater than or equal to 1, is the approximation ratio. This defines an approximate NN point *o* of *q* as having a distance ratio to *q's,* exact NN of no more than c times the distance. LSH relaxes this a little by defining ANN as Ball Cover *B(q,r)* and *B(q,cr)*. A Ball Cover is defined as query which must return a point if it falls within *B(q,r)* and at most *B(q,cr),* otherwise it should return nothing, as seen in figure 14 [48].

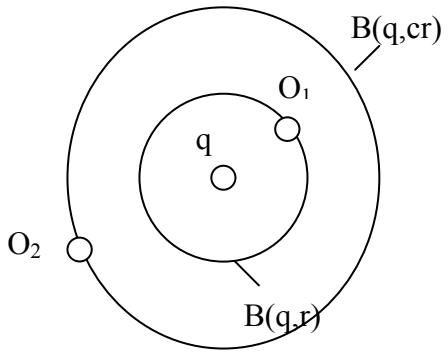



Figure 14. Illustration of C-Approximate Ball Cover query.

Given this approximation some researchers have concluded that an ANN queries can be reduced to a number of Ball Cover queries with different radii [39, 48] and that any point in *B(q,cr)* is therefore a c-approximate NN of *q*.

As described in previous discussion on Locality sensitive hashing, a set of hash functions to determine which points might be ANN to $q$. Points in the domain search space are first hashed into buckets based on a hash function H(o). These hash functions map d-dimensional points into buckets, which is locality sensitive if the chance of mapping two points to the same value (bucket) grows as their distance decreases [48, 112]. The basis is simple if two points are close to each other they are likely to fall into the same bucket and if faraway they are likely to fall into different buckets [30].

It can be shown that LSH provides a theoretic guarantee on quality but with several draw backs, very high cost in terms of space and query complexity. The original proposal for LSH (known as rigorous LSH) required numerous radii (buckets) to achieve its guarantee on quality but at a cost of space and query complexity. This high cost was resolved by Gionis using heuristics to find a pre-determined radius, this method became known as Adhoc LSH [43], but it sacrificed the quality of the results (guarantee) for reduced space and query complexity. Many researchers still continue to explore variations on LSH, for example Junhao [37] has focused on finding better hash functions known as Collision Counting LSH (C2LSH). Whereas, others have been applying LSH to other multiply dimensional search problems, for example, Venu has developed a variation for High Dimensional Similarity Searches [86].

Locality Sensitive B-Tree (LSB or LSB-Forest) takes LSH one step further by employing a forest of trees in an attempt to answer the short-comings of the different LSH methods, yet still guaranteeing the quality of the NN returned to be that of c-approximate ball cover. This new approach, takes a different path than the normal use of hash functions found in current implementations of LSH. LSB-Forest uses z-ordering of

d-dimensional points, pushing the z-order values which have been reduced to a single dimensional value into a B-Tree [112]. The authors proved that z-ordering is essentially locality sensitive and that by ordering the values into a B-Tree they can search much more efficiently. Queries are performed based on length of longest common prefix (LLCP) derived from the z-ordering of a query point $q$. As a further improvement they use a forest of LSB trees. The algorithm works by calculating the z-ordering value of a point $q$, searching each tree in the forest returning leaf nodes with the next greatest LLCP until one of two conditions is met. The first being a restriction on the number of pages read and the second being that the ANN returned so far has a distance at most $2^{u-(v/m)+1}$ to $q$ (this distance is defined by the authors as an upper bound [111] and is based on the LLCP and Z-ordering, not Euclidean distance). These conditions have the affect of controlling or reducing search complexity so that it meets the guarantee of rigorous LSH with improved space and query complexity and is equal to Adhoc in terms of space [112]. Current research from Junhao has offered a new LSH method known as C2LSH which has been shown to offer better guarantee on query results and outperforms LSB [37].

## 2.7 Conclusion

What has just been presented is a summary of past and current research into this problem, covering the basic types of searching, indexing methods and matrix space methods. This provides the foundations for the next sections as well as the basis for ANN using R-Tree.

## 3. Implementation and Experimental Procedures

This section provides a summary of my implementations and experimental procedures including experiment parameters, results and analysis techniques. When working with large index structures and complex data files it becomes necessary to describe in detail the storage requirements, in this case R-Tree, because different parameters, such as the page size or number of dimensions, can affect not only the storage but the results from different experiments involving different data sets.

### 3.1 Definition and Parameters

The following tables (2 and 3) contain lists of abbreviations and definitions, with a short description used throughout this dissertation, these will be defined further as they are used.

Table 2. List of Abbreviations used throughout this chapter and later chapters

| **Abbreviation** | **Description** |
|---|---|
| D | Number of Dimensions |
| Q | Query point |
| MBR | Minimum Bounding Rectangle of a set of points |
| CB | Current Best solution found during a search |
| NN | Nearest Neighbor |
| ANN | Approximate Nearest Neighbor |
| KNN | Kth Nearest Neighbor |
| RNN | Reverse Nearest Neighbor |
| KB | Kilo Byte |
| MB | Mega Byte |
| GB | Giga Byte |
| IO | Input Output (usually referring to Disk access) |

Table 3. List of definitions used throughout this chapter and later chapters

| **Definitions** | **Description** |
|---|---|
| Rank | Solution position in the $k^{th}$ nearest neighbor ordering |
| Leaf Node | Nodes containing data points |
| Internal Node | Internal nodes of the tree which contain MBR |
| Mindist | Minimum distance from Q to the closest edge of a given MBR |
| Maxdist | Maximum distance from Q to the furthest corner of a given MBR |
| Alpha | Parameter used by pruning heuristics |
| Cardinality | The amount or number of branches from one internal node to the next |
| Fan out | Same as Cardinality |
| Page Size | Size of a physical disk page in bytes or Kilo Bytes |

## 3.2 Experimental Data Sets

To test my implementation it was necessary to create testing data. For experimentation purposes a number of data sets was generated, I denoted the sets as uniformed, clustered, and Gaussian, which were based on the underlying probability distributions, that was used to generate the data. Most of the experiments in chapter 5 and 7 will be with sets having 32 or more dimensions. Past research [104] has shown that above 10 dimensions just about any index's performance drops to no better than a sequential scan, therefore focusing on at least 32 dimensions at this time should provide insight as to whether a new method is working well. The uniformed, clustered and Gaussian data sets were generated with a simple program that randomly selected points within a given range based on the desired data distribution. In the case of uniformed data range was 0.01 to 0.99999 for all dimensions, for clustered data 20 random clustered were specified, each having approximately the sample number of points (total did not exceed 100,000) again in the range of 0.01 to 0.99999 for all dimensions. Lastly for

Gaussian data the distributions were centered, around 0 with the range being from -4.0 to 4.0 for all dimensions. Finally a relatively large data set containing real world data was located at the UCI KDD Archive Information and Computer Science University of California, which contained 32 Dimensional color histograms for 66,000 images [97].

### 3.3 R-Tree Implementation

The R-Tree implementation used throughout this project follows the methods originally proposed by Guttman and later worked on by Roussopoulos [44, 84]. The basic structure is as follows, each internal node starting with the root node contains N number of branches known as its fan out, each of which contains an MBR and a pointer to the next level's internal node defined by the MBR. The maximum number of branches in an internal node is governed by the number of dimensions and page size. The overall tree is height balanced meaning the number of nodes from the root to any leaf node is the same. Data points are stored in the lowest levels called the leaf node, again the number of points stored is governed by the number of dimensions and page size. Points are inserted by traversing the tree selecting branches that would be least affected by the insertion of a new point. When the leaf node is reached the new point in inserted into the leaf. After insertion into the leaf the MBR of each node traversed must be updated, if any leaf or internal node becomes full it is split. Whereas deletion from the index is handled by simply locating the point and removing it from the leaf node then updating the MBR traversed.

Building on this method I added disk IO to the index structure described above by storing both the internal node and leaf nodes in a data file. This changed the index from

being a memory based structure to disk based, much like any data base system would be. I added buffer management functionality, which allows me to specify which type of buffering technique to be used. I implemented Least Recently Used (LRU), Most Recently Used (MRU), First In First Out (FIFO) and Last In First Out (LIFO). This was added to simulate a real world data base system, which uses available memory as data buffers to improve disk access to the data base. Rather then read and write every page of data during an IO request, the buffer management algorithm checks to see if the requested page is currently in memory and if so uses this buffered page thereby saving disk IO. This additional functionality allowed me to control the amount of memory allocated to the index. During any testing including building, maintaining, and search I specify the number of dimension, the page size in bytes, and the number of page buffers to allocate (main memory used by the index). Because the index is now disk based I only needed to build an index once for any given data set. This also gives me a real world basis for comparison, once built I can compare the physical size of the index to that of the data file, not only that I can examine the cost in terms of storage and IO, of the data structure, including the nodes used for searching (internal nodes) as well as the nodes used for storage (leaf nodes).

### 3.4 R-Tree Storage Cost

When looking at an index structure one must consider two cost measures, time and space. Storage cost (space) needs to be discussed first, because it directly effects the time cost. For an R-Tree, like any other tree based index, consideration has to be given to the cost of storing the bounding information as well as the final points. This can be easily

explained using points in a 2 dimensional space as seen in figure 15. R-Tree is a tree based data structure as the name suggests, with branches starting at the root, each bounding a sub-space of the overall space. Each branch can be thought of as R-Tree with branches bounding the sub-space of the parent branch. This repeats until a tree large enough to hold all the points is constructed, with the points being stored in the lowest level of the tree known as the leaf nodes. The height of the tree is inversely proportional to the branching, which is based on the number of dimensions and storage space allocated to each node. Figure 15 illustrates points in a 2 dimensional space and the corresponding R-Tree needed to store the data.

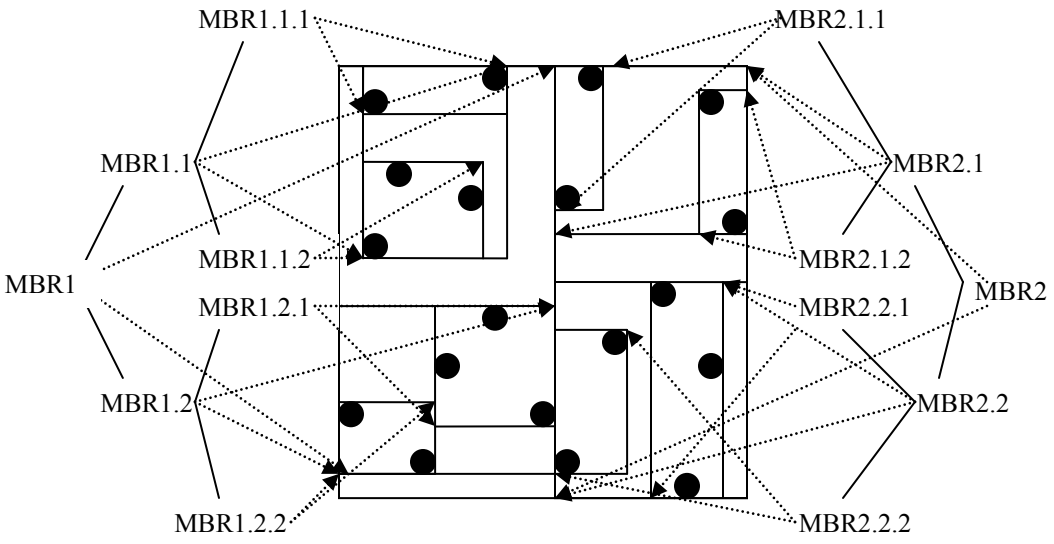



Figure 15. The root has 2 branches defined by MBR1 and MBR2, under each MBR are sub-trees rooted by the parent MBR. In this figure there are 4 levels each with 2 branches with the last being the lowest level which contains 2 or more points.

One can see from this figure that at the root of the tree, 2 bounding regions exist (MBR1 and MBR2), within each bounding region is a sub-tree, and within each sub-tree

is another sub-tree. The bounding regions are stored in the internal nodes of the tree and finally the raw points are stored in the lowest levels (leaf nodes).

With this index structure it is possible to answer a number of types of queries including approximate nearest neighbor, K nearest neighbor and even reverse nearest neighbor (with some slight modifications to the data structure) [109]. Solving both nearest neighbor and K nearest neighbor does not require any modifications to the data structure, for KNN all that needs to be done is to maintain a sorted list (by distance) of current best NN, updating the list as a new NN are found during a search.

The storage requirements for an R-Tree are based primarily on the number of dimensions and page size. Here page size is the physical storage of one node either an internal or leaf node (both of which require the same number of bytes). An internal node has the following fields; level number, branch count and set of MBR (branches). Each MBR is comprised of a high corner and a lower corner representing the minimum and maximum of each dimension, as well as a page pointer representing the pointer to the next node (branch pointer) which could be an internal or leaf node. The storage requirements of an internal node can be seen in figure 16 and 17.

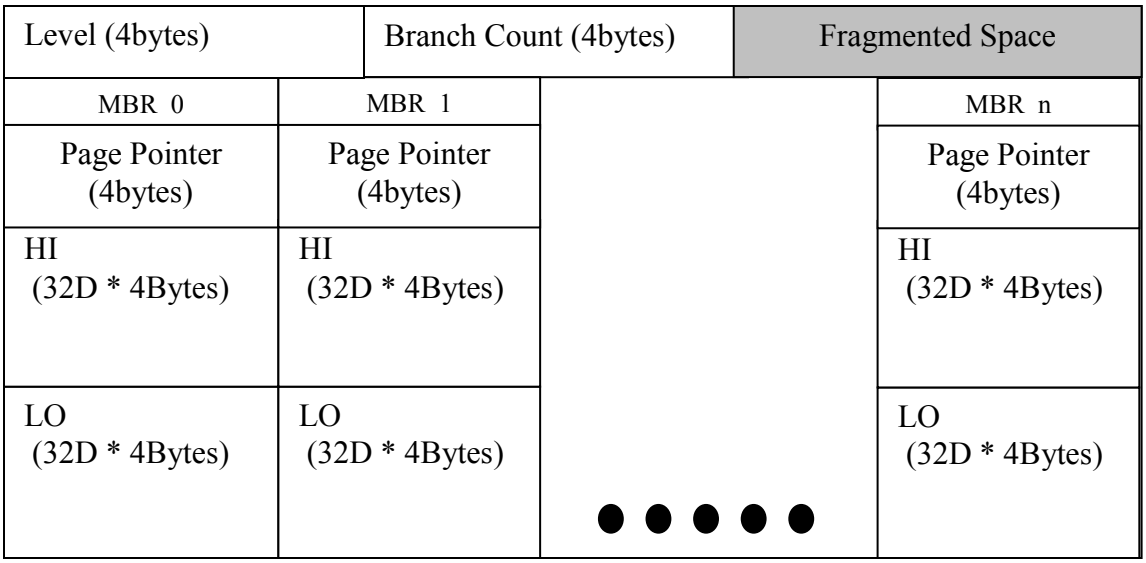



Figure 16. R-Tree Internal node structure with field names and sizes in bytes

Level – Internal Node level within R-Tree
Branch Count – Number of used branches in this node
MBR0 – MBRn – MBR defining a bounding region
MBR HI – upper bound corner
MBR LO – Lower bound corner
MBR Page Pinter – Page Pointer which points the node in the next level (sub tree)

Figure 17. Break down of the fields contained in a internal node

The total fan out or number of branches is calculated from the page size and number of dimensions. For a page size of 4KB with 32 dimensional points the fan out would be computed as seen in figure 18.

| | |
|---|---|
| Fan out | = (4KB – (4B + 4B)) / (4B + (4B * 32D) + (4B * 32D)) |
| | = 4088B / 260B |
| | = 15 |
| Fragmented space | = 4K – (4B + 4B) – (15 * 260B) |
| | = 188Bytes |

Figure 18. Computing the internal node fan out based on a 4KB page size

Based on these calculations an internal node would have a fan out of 15 (that is 15 MBR per page) with 188Bytes of fragmented space.

As with most index structure that use persistence storage the page size is fixed for all types of nodes. Therefore a leaf node will use the same amount of storage and would have the following fields; level number, point count and data points with a reference pointer. The storage requirements of a leaf node can be seen in figure 19 and 20.

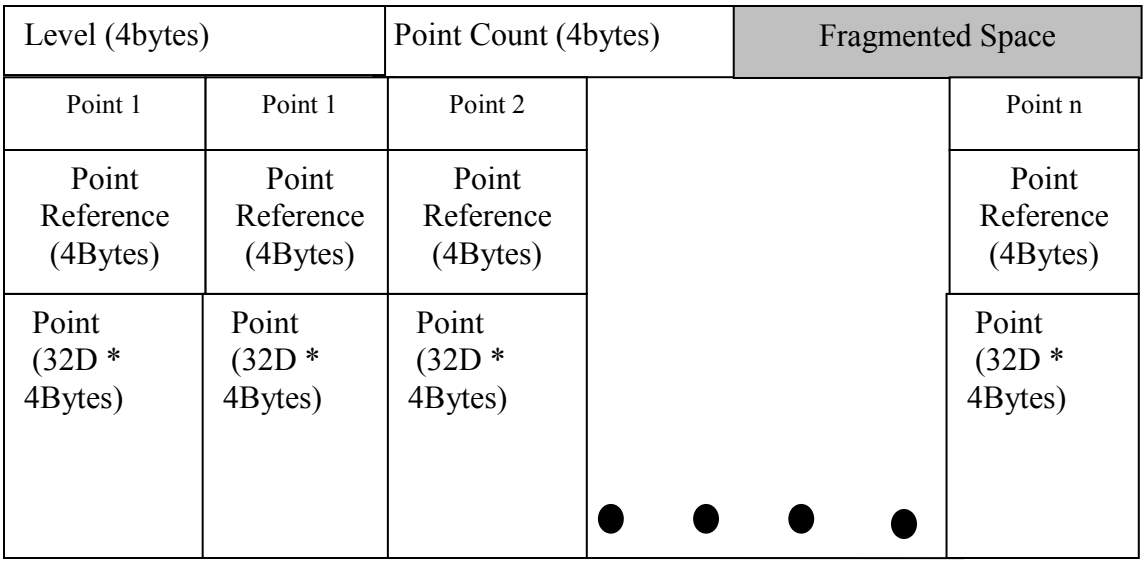



Figure 19. R-Tree Leaf node structure with field names and sizes in bytes

| |
|---|
| Level – Leaf Node level within R-Tree<br>Point Count – Number of points in this node<br>Point 0 to Point n – Point information<br>Point – The actual point (all dimensions)<br>Point Reference Pointer – Data reference pointer used to point to the data reference by this point |

Figure 20. Break down of the fields contained in a leaf node

The number of points that can be stored a leaf node is calculated from the page size and number of dimensions. An example how to compute the storage capacity of a leaf node with a page size of 4KB using 32 dimensional points can be seen in figure 21.

Number of points = (4KB – (4B + 4B)) / (4B + (4B * 32))

= 4088B / 132B

= 30

Fragmented space = 4K – (4B + 4B) – (30 * 132B)

= 128B

Figure 21. Computing the number of points that can be stored in a leaf node based on a 4KB page size.

Based on these calculations a leaf node would store a maximum of 30 points with 128B of fragmented space per node. This computation as with the fan out need only be computed once as all nodes have the same physical page size.

### 3.5 R-Tree Access Cost

Knowing these numbers it is now possible to calculate the approximate size of the R-Tree index, for a given number of points. Because the tree is balanced then the number of levels (height) is a logarithmic function of the number of leaf nodes needed and the fan out of the inter nodes. For example, 100,000 32 dimensional points, assuming each leaf node is 75% full then, 4,348 leaf nodes would have to be allocated to store just the points. This in turn would require at least 311 internal nodes (usually a little higher as not all internal nodes will be 100% full), having 3 levels as seen in the following figure 22.

Number of leaf nodes = (75% * Number of points per leaf) / 100,000

= (75% * 30) / 100,000

= 23/100,000

= 4,348 Leaf Nodes required

Number of levels = $\log_{\text{fan out}}$ (LeafNodes)

= $\log_{15}$ (4,348)

= 3.25 rounded down to 3

= 3 + 1 (1 more level for leaf nodes)

= 3 levels

Number of Internal nodes level 1 = (4,348 Leaf nodes ) / 15

= 290 Level 1 nodes

Number of Internal nodes level 2 = (290 Level 1 nodes ) / 15

= 20 Level 2 nodes

Number of Internal nodes level 3 = (20 Level 2 nodes ) / 15

= 1 Level 3 node (known as the root)

which needs 2 branches

Total number of internal nodes needed = 290 + 20 + 1

= 311 internal nodes

Figure 22. Computing the height and storage for an R-Tree housing 100,000 32 dimensional points

To compute the access cost during a search (traversal) simply count the number of internal and leaf nodes visited, since each is 4KB the total space read is (Internal + Leaf) * 4KB = KB read. This becomes a simple measure of how efficient a particular search method is. One conclusion that can be drawn from the R-Tree index is that it has a higher storage cost when compared to other high dimension search methods such as the SS-Tree, VA-File and M-Grid, the MBR, counters, page pointer and tree level indicators, all must be stored requiring extra space in the index. Therefore, it is very important to count the total number of nodes accessed, which translates directly to the amount of disk IO performed and the number of bytes read. If for example during an experiment 300 internal nodes are read and 110 leaf nodes then the true cost is (300 + 110) * 4KB (assuming 4KB page) = 1,679,360 Bytes read, which might be plotted against the total cost of the R-Tree index, and in turn compared to sequential scan. It also becomes important to try and reduce not only the number of leaf nodes but also the number of internal nodes read.

As talked about earlier, results from any method need to be an improvement over a typical sequential scan. This involves reading the entire data set, including a reference point (to be fair), computing the distance from Q to every point in the data set and returning Q's NN at the end of the process. Therefore, to compare R-Tree access to a sequential scan, the storage requirements need to be considered. In the previous section it was mentioned that a point required 4 Bytes per dimension, and the reference pointer also required 4Bytes. For the sake of comparison this can be considered the lower bound on the data storage requirements. Therefore, a 100,000 point data set having 32 dimensions would require (32D * 4Bytes + 4Bytes) * 100,000 = 13,200,000 Bytes and the

corresponding R-Tree would require approximately 19,083,264 Bytes (note this additional space is not fully utilized in that the leaf nodes could store an additional 30,000 points). This makes presentation of results a function of the original data set if a result in terms of disk IO is greater than the size of the raw data file then the method did not have any improvement.

There is also another item to consider, CPU time. During a sequential scan of 100,000 data points, exactly 100,000 NN distance calculations will be performed. Any method that accesses less of the data set will also reduce the number of distance calculations. This is usually not presented as it is assumed that reduced disk IO also equates to reduce CPU time (I assume CPU time is much smaller then disk access time).

### 3.6 Maintenance Cost

As with any index structure there is a cost associated with building and maintaining the index. For this work I am not as concerned with that as I am with searching, still maintenance needs to be discussed. Typically an index would be built by repeated insertion of single points, Chapter 4 has a detailed discussion of an alternate method to initial building of an index known as bulk loading/inserting. Once the index is built additional points can be added by either insertion via single point insertion or by bulk insertion.

### 3.7 Accuracy Measures

In some cases people are willing to accept a less then accurate result this requires one to consider a second measure, accuracy. If the search is for an exact result and a

pruning heuristic is employed then the possibility exists that the results returned will not be exact. The best way to think of accuracy is that the results returned are not the exact NN but rather an approximate solution, meaning that points might exist that are closer but were missed during the search (pruned by some heuristic) not only that, but this lack of accuracy increases as the amount of pruning increases which equates to less searching. If the user is willing to accept some inaccuracy, then any high dimensional ANN search method should be able to provide some level of performance or quality in terms of accuracy. One way to provide a level of performance is to measure the accuracy during development and through experimentation. Currently there are two methods that can be used to measure the quality (accuracy) of an ANN result returned, with the first being rank and the second distance ratio.

The first method known as the rank of the solution is defined as the ordering of a query point's solutions based on distance with the NN having a rank of one and the $2^{nd}$ NN having a rank of two etc. For testing purposes the actual rank of a given solution is computed via a sequential scan of the test data computing the distance from the query point to every point. By checking the query results ordering one can get its actual rank (correct). This is used to report results and analyze the results, as it gives an idea as to the accuracy of the results. In a real world implementation the actual rank of the solution would not be computed or known, yet based on the analysis done during development and testing it should be possible to provide some idea as to the accuracy.

The second method is the ratio of the query points ANN and to its exact NN, also known as the distance ratio (Formula 5). This method, like the rank measure requires the query point's exact NN therefore it is only computed as a means to determine the

accuracy of the results returned during an experiment and ultimately how well an implementation for ANN is performing.

$$distRatio = \frac{Dist(q, ANN)}{Dist(q, NN)} \qquad (5)$$

Throughout most of the literature on the subject of ANN in high dimensional space, distance ratio is used as the measure of accuracy, because of this I will also use distance ratio to report my results, so that my results can be compared to other research.

### 3.8 Conclusion

This section has described the parameters that must be considered when analyzing the results of any query using R-Tree index. If these parameters are not taken into consideration when testing and running experiments then the performance results could be skewed and may not provide a fair comparison.

## 4. Bulk Loading R-Tree

With any index structure there are additional costs associated with maintaining the index. This includes initial building the index, inserting and deleting points. These are not usually included in any experimental results studying high-dimension queries because the experiments are usually focused on searching not insertions and deletions. Still these costs need to be discussed. When initially building R-Tree index the primary method used is single point insertion. One point is read from the input, the tree is

traversed according to an insertion algorithm when the best leaf node is located the data point is added to the node then all parent nodes affected are updated. The cost of this type of insertion is basically the cost of reading one internal node for each level of the tree, (height) and the cost of reading one leaf node, as well as the cost of possible updating and re-writing each node read. As new points are inserted they may traverse the path causing the same internal nodes to be updated numerous times as their MBR needs to be updated, this leads to a lot of redundant IO.

My previous work can be directly applied to R-Tree [65] mainly involving work done on bulk loading/insertion algorithm for the Rdnn-tree [109], an indexing structure developed to efficiently answer nearest neighbor and reverse nearest neighbor (RNN) queries (R-Tree extended to support reverse nearest neighbor queries). The algorithm developed utilizes the notion of in-memory organization of the data to efficiently insert multiple data points, with the following advantages: it was independent of the order of input the same cannot be said of many bulk insertion algorithms; it does not require any preprocessing of the entire data set; the algorithm was orthogonal to the buffer management that it used; and it was applicable to both bulk loading (initializing the index) and bulk insertion (addition of new points).

### 4.1 Previous Work on RNN

Many have developed methods to find RNN efficiently; a good example proposed by Korn and Muthukrishnan [59], is to preprocess the data set by finding the nearest neighbor of every point of the data set and then store them. This process although efficient from one aspect still required additional work when the index is built. The work

from Yang and Lin improved on this by combining both NN and RNN queries into one index [109]. Other work on algorithms for reverse nearest neighbors can be found in following papers [12, 92, 93, 94, 95].

The method given developed by Yang, was an extension of the basic R-Tree known as the Rdnn-tree index. The major modification to the R-Tree was the addition of the nearest neighbor distance for each point stored in the leaf nodes and for each MBR (branch), the additional storage of the maximum of all the nearest neighbor distances contained in the MBR. This extra information allows a single index to handle both nearest neighbor and reverse nearest neighbor queries efficiently, thus enabling quicker queries as well as effective insertion/deletion. Yang's work showed that the Rdnn-tree outperforms the original two trees approach by a factor four or more requiring only a single index. He was also able to show that, single point insertion also has far better performance as compared to the standard approach.

## 4.2 Previous Work on Bulk Loading R-Tree and Other Index

Many other researchers have also discovered the need to find alternate methods to efficiently insert and update tree based indexes. Some of the methods proposed for other index structures may also be applied to R-Tree

One approach to bulk loading taken by Ciaccia [25] is to partition the new data, then build a tree for each partition, and finally combine the trees together forming a single tree. The authors point out that care must be taken in how to partition the data; otherwise it might be necessary to perform a reorganization of the new tree. The method for bulk loading M-Tree, proceeds by drawing samples from the data set, then each data

point is assigned to the sample that it is closest to. Using this approach still requires a reorganization of the final tree. The authors point out that this concept can be applied to bulk insertion.

A second approach given by Choubey, [22] proposes to bulk insert points into an R-Tree by first using clustering techniques to partition the points into spatially-close clusters, second build a small R-Tree for each partition and third grafting it into the original R-Tree.

A third approach by Arge [6] involves building the index by buffering the points to be inserted temporary. For example, Arge proposed a method whereby the internal nodes have buffers associated with them. When an insertion occurs, rather than traversing down the whole path, the insertion stops at a certain level, and the points to be inserted are stored in the buffer associated with the internal node. When enough points are accumulated in the buffers, they are pushed down to the next level. This process is repeated until the points eventually reach the leaf nodes.

The previous works on bulk loading R-Trees as well as other tree based indexes works well, but for the Rdnn-Tree these methods may not be efficient because of the additional work needed to maintain the index. My test showed that, for a typical 2,000,000-point data set, inserting the points one at a time requires on average of 17 page requests.

### 4.3 Rdnn-tree Bulk Loading/Insertion Method

For the Rdnn-tree, I incorporate bulk loading and bulk insertion into one algorithm. Bulk loading is done by first reading a sub-set of points into main memory and

preprocessing them, and then inserting those points into the current Rdnn-tree. This approach enables this method to treat both bulk loading and bulk insertion uniformly. Moreover, the algorithm I developed was flexible enough to deal with variable size of main memory allocated to the program.

The goal of the preprocessing was to enable multiple points to be inserted into the tree without duplicating effort in the form of additional page requests. The most intuitive way was to organize the points such that I can determine which points are going to be in the same node, or traverse a very similar path during insertion. Then multiple points can be inserted only by one transversal of the tree, saving a significant amount of page access, equating to reduced disk IO.

An additional point to consider involving the Rdnn-Tree is that inserting a point not only requires finding the correct leaf node, but also requires finding its nearest neighbor to be stored as well as its reverse nearest neighbors (so other points can update their nearest neighbor distances). This has two implications. First it makes the grouping of points even more important. One can surmise that points that are to be inserted into the same node are likely to be close together, thus likely to share the same nearest neighbors. Thus inserting them at the same time will save effort in finding nearest neighbors. Moreover, the points may share the same reverse nearest neighbors. This suggests that inserting them together can reduce the number of updates of nearest neighbor distances. This is best illustrated by the following figure 23:

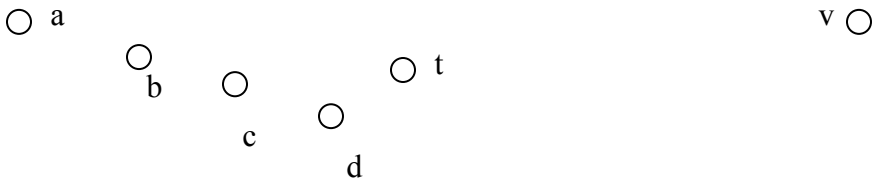



Figure 23. Illustration of wasted effort in single point insertion. *v* is *t*'s original nearest neighbor. Insertion of *a, b, c, d* will cause successive updates of *t*'s nearest neighbors.

In figure 23, assume point *t* is a point that is already in the Rdnn-Tree. Point *a, b, c, d* are new points to be inserted into the existing Rdnn-Tree. Notice that they are inserted one at a time in the order of *a, b, c* and *d*, then *t* is going to be the reverse nearest neighbor of all 4 points at the instance when each of the points is inserted. Based on this, I need to update the nearest neighbor distance of *t* 4 times as well as access the node 4 times. This can lead to a large amount of wasted effort in terms of IO if *t* is actually in a different node then the new points. However, if I can insert all four points at once, then *t*'s nearest neighbor distance only needs to be updated once, thus saving a lot of disk access.

A second implication of the extra work needed is that I cannot directly apply any of the standard bulk loading/insertion algorithms that currently exist in literature, therefore a new method needed to be developed.

All of the discussion about this suggested that if I organize the points in such a way that I insert the closest points together I should be able to save disk IO. However, one needs to be cautious about this. As I mentioned earlier, during insertion one needs to find the nearest neighbors as well as the reverse nearest neighbors of the new point to be inserted. However, in the case of bulk insertion, the nearest neighbor/reverse nearest

neighbor of a point to be inserted may actually be some other point(s) that are to be inserted. This suggests that care must be taken when determining those values.

Taken all of this into consideration, I proposed building an in-memory Rdnn-Tree for the points that are to be bulk loaded/inserted. This essentially kills two birds with one stone: the in-memory Rdnn-Tree automatically clusters the data points, so that one can determine which points can be inserted simultaneously; also the in-memory Rdnn-Tree provides the nearest neighbor distance of the points to be inserted (relative to other points in the input set). This can help to speed up finding the real nearest neighbors and reverse nearest neighbors of the points to be inserted. Once the in-memory tree is built, I pick up each leaf node and insert the points together. If there are too many points to be loaded/inserted, then I first read a portion or sub-set of the data points to be inserted. After that, I insert all the points in the in-memory tree, and then build a new in-memory tree for the next portion of points. The overall pseudo-code is listed below in figure 24.

```
Algorithm 1: Bulk Insertion/Loading an Rdnn-tree

BulkInsert (Rdnn-tree T, PointSet S)

While not all points inserted do
        Read the next portion of S and build an in-memory Rdnn-tree T'
        For each leaf node lnode in T' do
                BulkInsertNode (T, lnode)
        End
End
```

Figure 24. Bulk Insertion/Loading

The *BulkInsertNode* procedure takes a leaf node (*lnode*) of the in-memory Rdnn-Tree and inserts the points into the main disk based Rdnn-Tree. The insertion procedure is the same as the original tree: first, the nearest neighbors and reverse nearest neighbors of the points to be inserted are located ([109] showed that these two searches usually travel down the same path, so they can be performed in one traversal). After that the nearest neighbor distance of the reverse nearest neighbors are updated. Finally, the points in *lnode* are inserted by traversing the tree again. Since there is a possibility that the points in *lnode* will eventually reside in different leaf nodes in the main disk based Rdnn-Tree, for the last traversal I apply a depth first traversal (as multiple branches of the tree may have to be traversed). Depth first search has an advantage of a fixed upper bound on the memory required. The pseudo code is listed below in figure 25.

Algorithm 2: Insert Individual node

*BulkInsertNode* (Rdnn-tree *T*, LeafNode *N*)

1. Let *P* = the set of points in *N* (For each point p, there is a *dnn (p)* denoting the current distance from its nearest neighbor)
2. Apply *Batch-NN-Search* [YAG01] to find the nearest neighbor for each point in *P* (call this set $NN_P$)
3. Update dnn (p) for each point in P if necessary
4. *BatchUpdateCurrentTree* (*T*, *P*)
5. *BatchInsertPoints* (*T*, *N*, $NN_P$)

*BatchUpdateCurrentTree* (Node *N*, set < Points > *P*)

If *N* is not a leaf
    For each branch *B*
        1. Let *R* be the bounding rectangle for *B*, and *max_dnn* as described in section 2
        2. Find the set $P' \in P$ such that $\forall\ p \in P'$, *dist (p' R) < max_dnn*
        3. *BatchUpdateCurrentTree* (*B.child, P'*)
else
    For each point q in Node
        If $\exists$ p' in P' s.t. Dist (p', q) < dnn (q).
            Find the minimum of such distance and update dnn (q)

Figure 25. Insert and update algorithms

I have omitted details of *BatchInsertPoints ()*. It is basically the depth first traversal of the tree while input each point in P in the appropriate node, and apply the R-Tree insert procedure accordingly. One final item to point out has to do with the number of points being inserted (contained in an in-memory leaf node), which is less then or equal to the number contained in a leaf node in the main index. Because of this, there is

no danger that a single leaf node will overflow into 3 nodes, this means the standard R-Tree algorithms that handle overflow (branch splits) can be used.

### 4.4 Bulking Loading Results

The following section will present some of the results from experiments involving 2 dimensional uniformed, clustered and real data. Tests were run using synthetic data, involving five different test sets of one million 2 dimensional non-clustered points as well as 5 different test sets of one million 2 dimensional clustered points, all randomly generated. For the real world data, a single test set was retrieved from the US government web site: [98]. This test set contained 288,000 interesting locations in the southeastern United States represented as latitude and longitude points. I first apply individual point insertion to create the index; then apply my algorithm, varying the number of points in the in-memory Rdnn-Tree (denoted by *Rsize*) from 1,000 to 10,000. For each value I apply the 5 different test sets and present the averages in the plots.

I have presented comparison of cost of building the index by measuring the cost in two ways. First I measured the number of read and write requests made. This gives me an overview of the cost. However, in case of bulk operations, buffering is an important aspect, as the same pages may be read and written repeatedly. Therefore, I maintained a consistent buffer count of 1,000 4K pages (4K is the size of a tree node in this set of experiment), with LRU as the page replacement policy, to make a fair comparison. Next I measured the actual number of page requests from the buffer pool giving me realistic numbers.

### 4.4.1 Bulking Loading Non-Clustered Results

Figure 26 shows the results for page request, while Figure 27 shows the results for read and write requests.

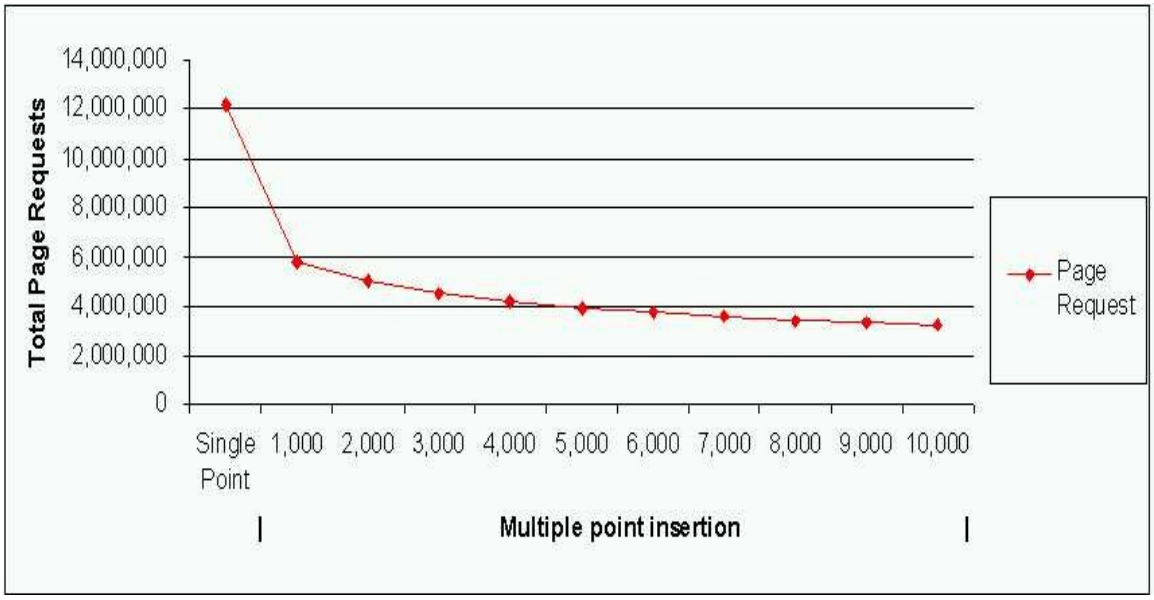



Figure 26. Total page requests for non-clustered data showing both single point and multiple point insertion.

The results clearly show that as *Rsize* increases there is a reduction in the number of page requests, as compared to single point insertion. There is an immediate reduction in page requests when *Rsize* = 1,000 and a continued improvement as it approaches 10,000. Figure 27 shows the change in pages read and written for the same set of tests.

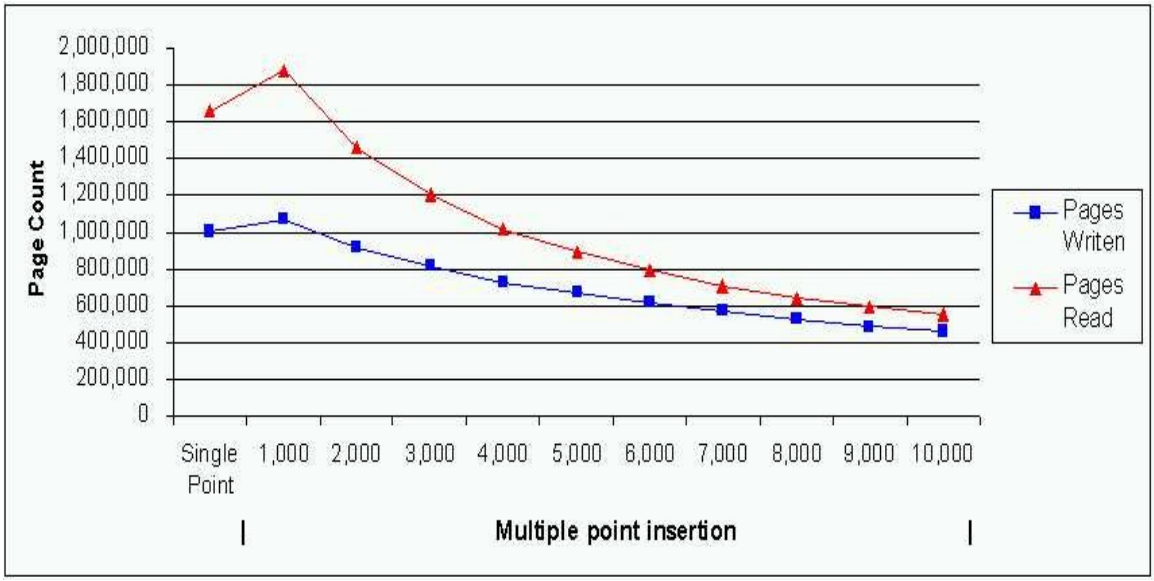



Figure 27. Read/Write page counts for non-clustered data showing both single point and multiple point insertion.

Figure 27 (above) shows that the number of read requests is more then half that of single point insertion, while the number of write requests is also close to being half that of single point insertion. Therefore, we can see that this method provides an inherit advantage. One other thing to note is that a small *Rsize* actually put the bulk insertion algorithm at a disadvantage. This is because it is less likely to locate points in the in-memory Rdnn-Tree that are actually close together and will benefit from bulk insertion. However, this situation is quickly remedied even with a modest *Rsize*.

### 4.4.2 Bulking Loading Clustered Results

Identical experiments are run again but this time with the data being clustered. Figure 28 shows the results.

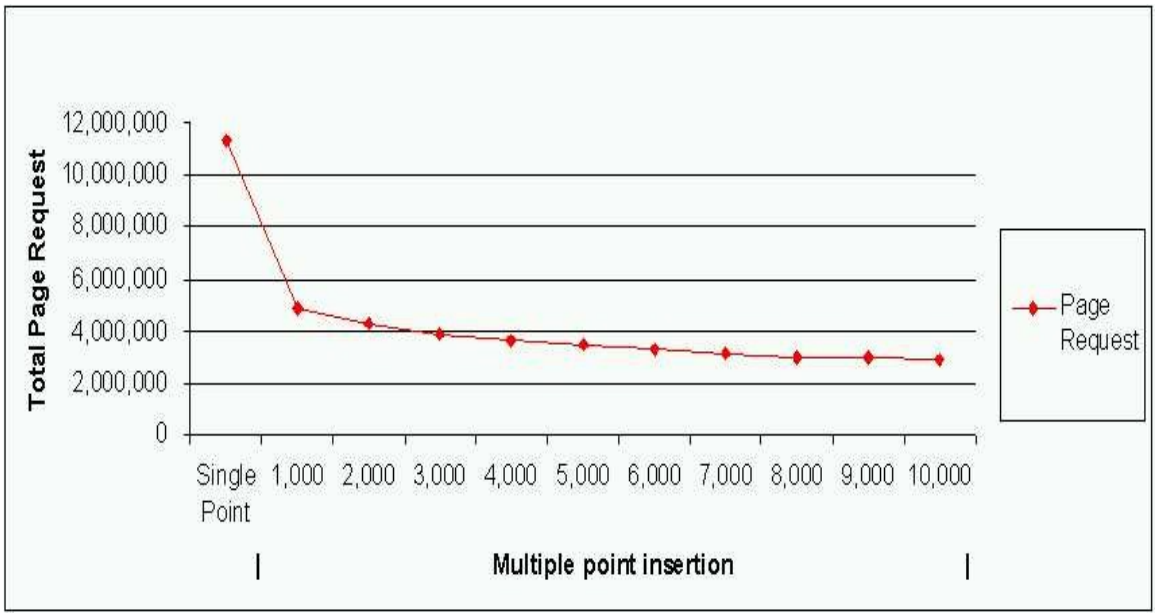



Figure 28. Total page requests for clustered data showing both single point and multiple point insertion.

Just as with the non-clustered data figure 28 clearly shows that as *Rsize* increases the number of page requests decreases, as compared to single point insertion. The improvement is as much as 4 times. Figure 29 shows the change in pages read and written.

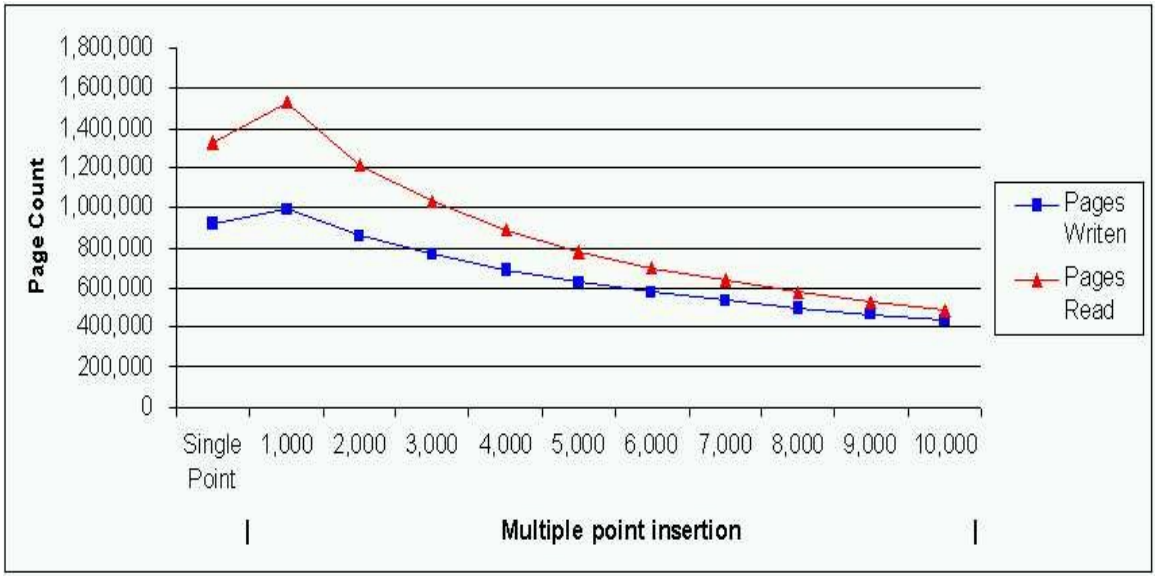



Figure 29. Read/Write page counts for clustered data showing both single point and multiple point insertion.

### 4.4.3 Bulking Loading Real Results

For the real world data I have one test set for each instance. Figures 30 and 31 highlight the results which show significant improvement for the bulk loading/insertion algorithms. Notice that the improvement is bigger than the case of synthetic data.

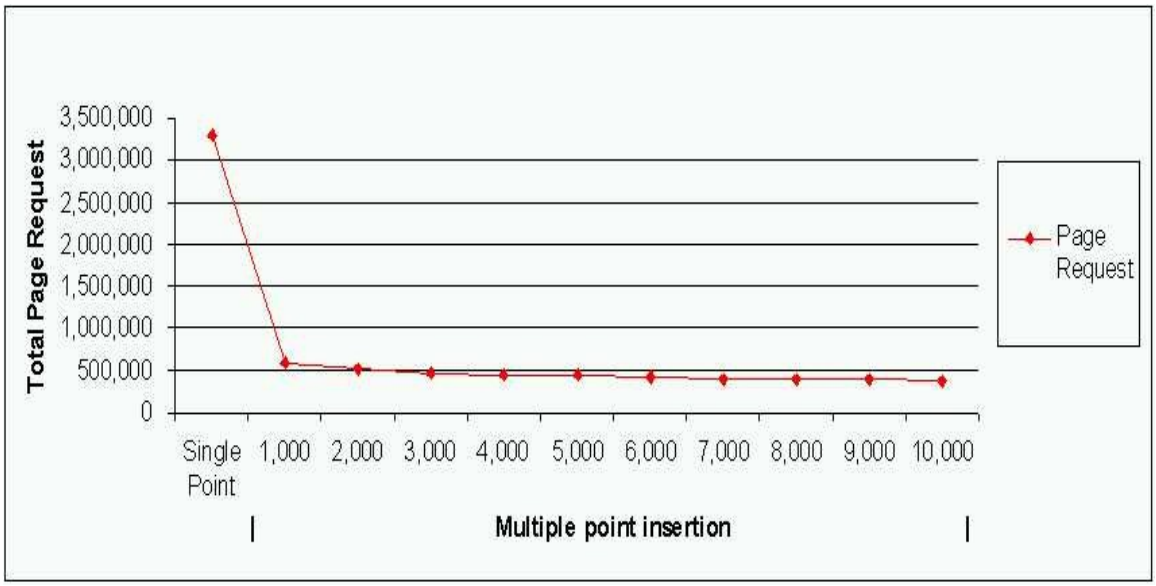



Figure 30. Read/Write page counts for real data showing both single point and multiple point insertion.

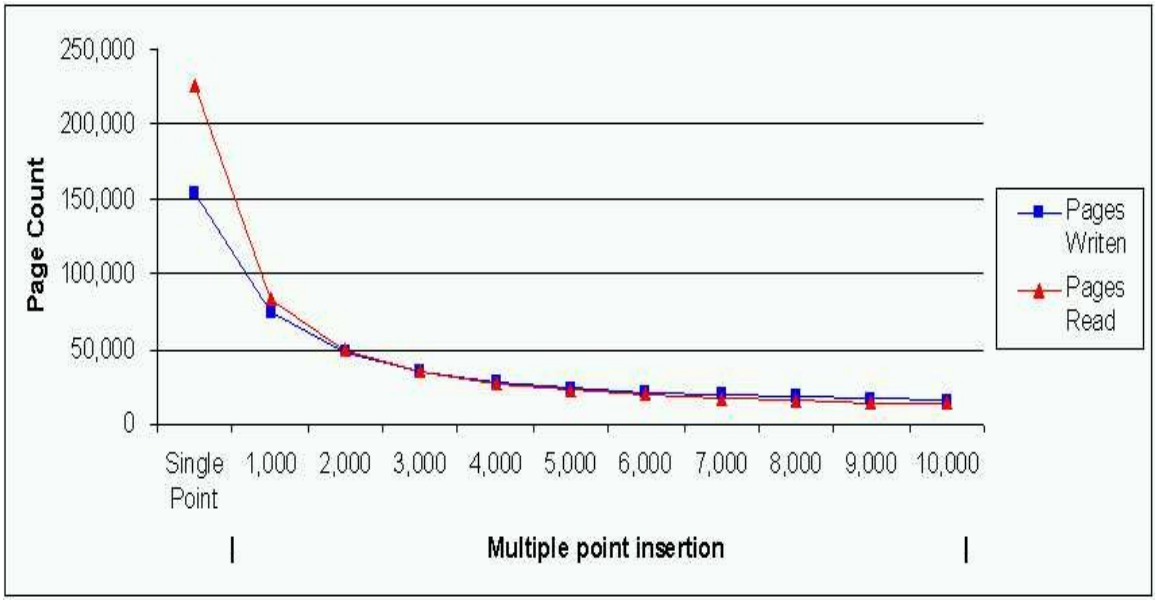



Figure 31. Total page requests for real data showing both single point and multiple point insertion.

### 4.4.4 Bulking Loading Robustness Results

One concern for this algorithm is that whether the performance of the bulk insertion will degenerate as the database size becomes large. To test this, I keep the running average of the page requests for every 10,000 points inserted. From figure 32, one can see that the number of insertions grows at a slow pace. Moreover, the growth rate for the bulk insertion is similar to that of single point insertion. Thus we can see that the bulk insertion algorithm is as robust as single point insertion.

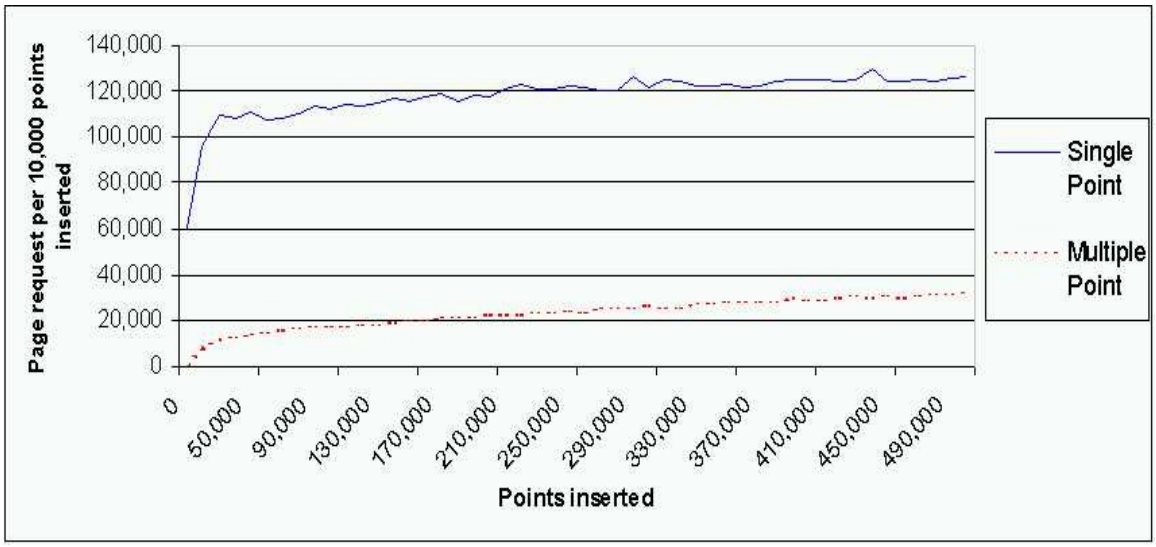



Figure 32. Comparison of average page requests and leaf requests for indexes created with both single point and multiple point insertion.

As I mentioned earlier, I need to check whether the index created by my bulk loading algorithm has similar quality to the one using single point insertion. To verify this, I execute a series of 100 reverse nearest neighbor queries for each of the indexes produced from the same test data set. The same queries are posed to all indexes. I measure the cost of the query by the average number of nodes visited as well as the average number of leaves visited – as in some cases, all but the leaf nodes of an index are

stored in main memory. Figure 33 shows the results of these tests. We can see that as *Rsize* varies, the average page request remains stable. This shows that the index built by bulk insertion is of good quality.

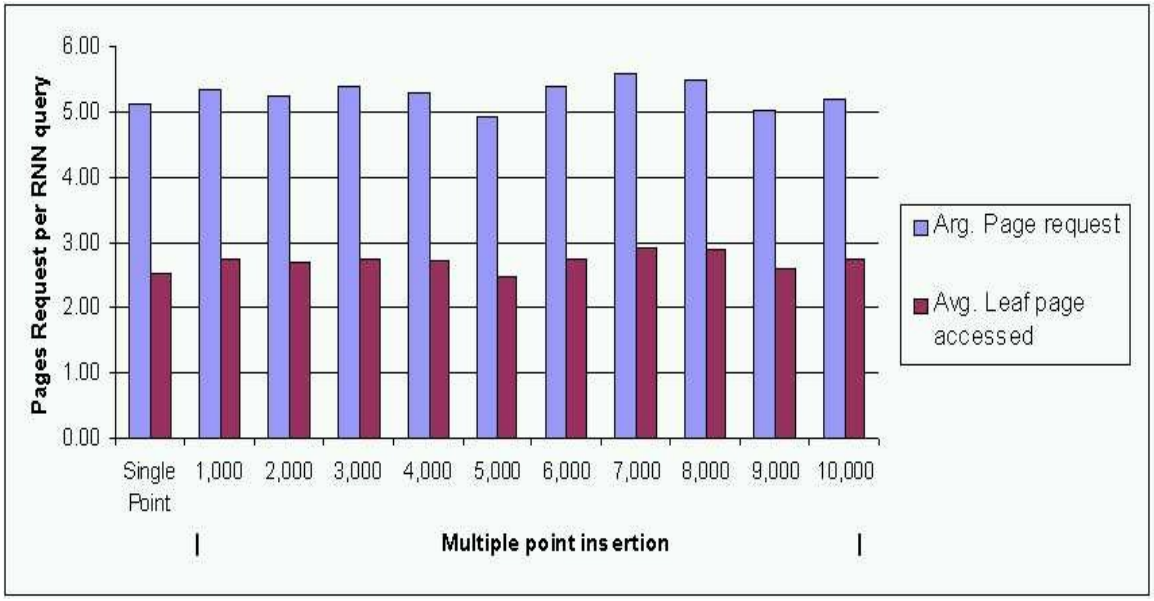



Figure 33. Comparison of page request per 10,000 points inserted for both single point and multiple point insertion.

### 4.5 Application of Rdnn-Tree Bulk Loading to R-Tree

This work on bulk loading showed that methods could be designed that would improve the maintenance cost by outperforming single point insertion, without any affect on efficiency or quality of the index. One of the goals of this work was to extend this particular method to other indexes including the general R-Tree. After publishing this work [65] I re-implemented my method in existing R-Tree code and ran some initial tests. Figures 34, 35 and 36 show the cost in pages accessed, written and read respectively for single point verses bulk insertion of data into a R-Tree. These tests show the cost of inserting 100,000 points for the four different data distribution, having 32

dimensions with R-Tree pages size set to 4K pages and data buffers limited to 1,000. For bulk insertion in-memory multiple point insertion was set to 5,000 (*Rsize*).

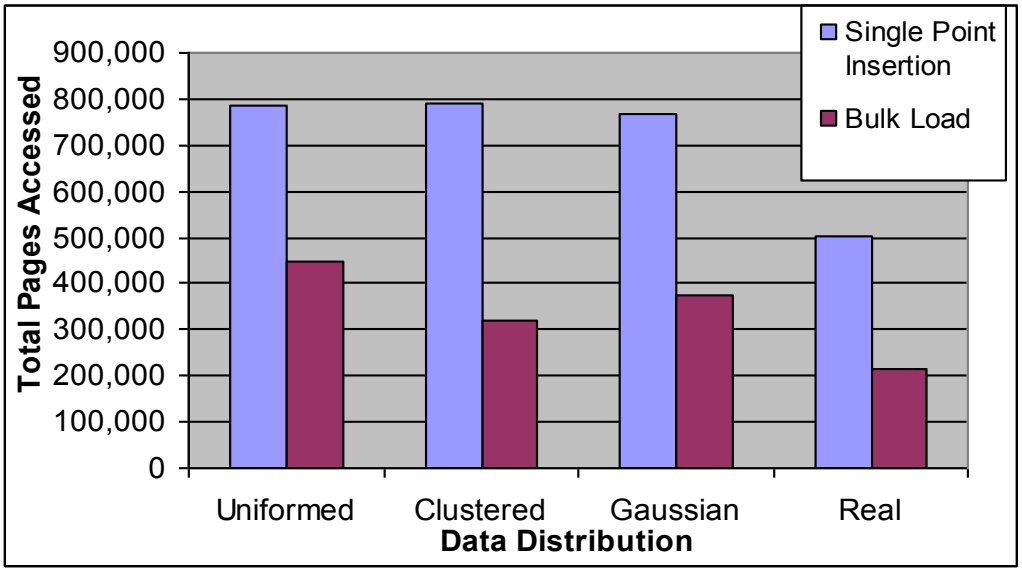



Figure 34. Total Pages accessed for single point verse bulk insertion, for all four data distributions

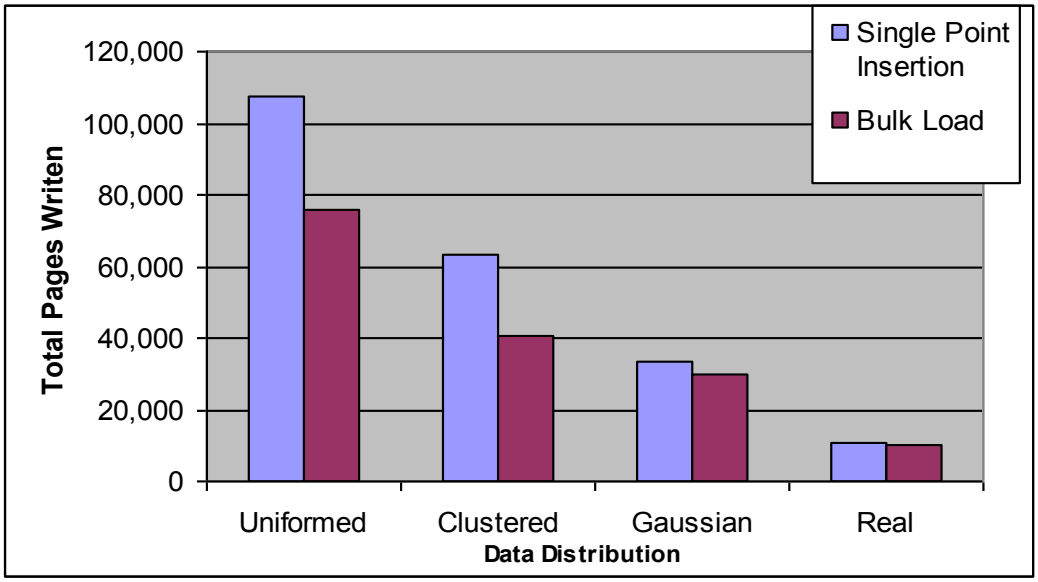



Figure 35. Total Pages written for single point verse bulk insertion, for all four data distributions

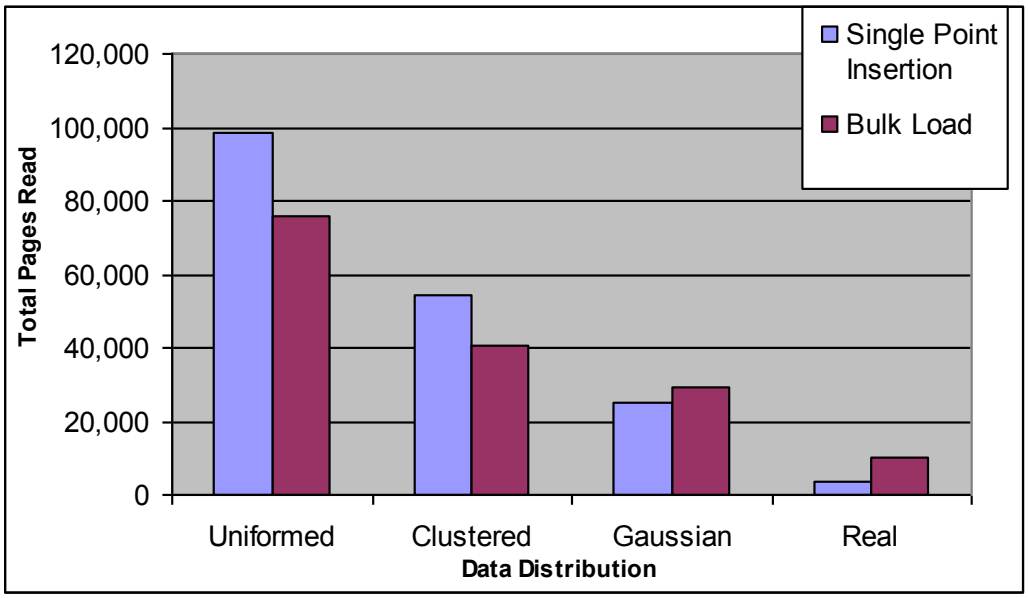



Figure 36. Total Pages read for single point verse bulk insertion, for all four data distributions

The figures show that for pages accessed bulk insertion showed an improvement of around 45%-65% over single point insertion. When broken down for pages written and read the improvement was less dramatic, in two cases (read results figure 36) Gaussian and real, bulk insertion did worse than single point insertion.

### 4.6 Bulk Loading Conclusion

In conclusion I have shown that the use of an in-memory Rdnn-Tree to process sub-sets of the input data, which is then used to bulk load a disk based Rdnn-Tree reduces disk IO by a significant amount. And when compared to an index built with single point insertion, the bulk loaded index has the same level of performance in terms of query IO. This means the bulk loaded index is of the same quality as one built with single point insertion. Further work shows this same method can be applied to R-Tree with similar

results, thereby improving the IO efficiency during construction of an R-Tree index. Finally my method can be used for both bulk loading of the initial data and bulk insertion of larger sets of new data.

## 5. Approximate Nearest Neighbor Search

As some researchers have already suggested, an approximate solution may be acceptable in many circumstances. Because of R-Tree's many advantages, it may be possible to apply heuristics for NN queries that may eliminate branches that are less likely to improve the current solution, thereby returning an approximate result. If this is possible then it should reduce the amount of searching hence reducing disk IO and CPU computations. Since this pruning would return the likelihood of an improvement (probability) I might prune a branch that could have a better NN, therefore I can no longer guarantee exactness. If the search heuristic is working correctly then the number of nodes searched will be reduced and the accuracy of the solution will still be acceptable.

Before developing any R-Tree ANN heuristics it is necessary to define the standard search method as a baseline, (other than a sequential scan) to be used as a comparison for improvement. The basic goal is straight forward, provide a simple but effective means to eliminate unnecessary searching during a query that wastes disk IO and CPU time. This will be done by means of a search heuristic applied to each MBR (branch) in each internal node during a query.

## 5.1 Standard Search

As described earlier R-Tree is basically a tree data structure comprised of bounding regions. Therefore, a basic straight forward NN search, also known as a standard search, should take advantage of this feature. During a search, traversal involves picking a branch based on the bounding region, which should lead to a better solution (closer NN). When a leaf node is reached the distance from Q to each point is computed, any point(s) that are closer to Q are now considered the current best solution and become Q's new nearest neighbor(s).

Because the branches in R-Tree index uses all dimensions in the similarity calculation rather than using them as a discrete bound as in a B-Tree index, it is possible that any given MBR may or may not contain a new solution (new NN). Therefore, to avoid searching the entire space some method of pruning must be used. If the method is optimal then only the branches that might lead to a solution are searched whereas branches that can never lead to a solution are pruned. One optimal method of pruning a branch (MBR) involves making a decision based on the minimum distance from Q to the closest edge of the MBR known as *mindist (Q, MBR)* defined by [77, 84]. This method is considered optimal in that it will only select branches that could contain a solution, and prune branches that can never have a solution. This simple method can be seen in formula 6 below.

$$Mindist(q,R) = \sqrt{\sum_{i=1}^{d} x_i^{\,2}}\ ,\ \mathrm{x_i} = \begin{cases} R_{low_i} - q_i & q_i < R_{low_i} \\ q_i - R_{high_i} & \text{if } q_i > R_{high_i} \\ 0 & otherwis \end{cases} \qquad (6)$$

By computing *Mindist (Q, MBR)* and comparing it to the current best solution (current NN), one can see that if Mindist is less than the distance from Q to its current best solution then the given MBR can potentially contain a new better solution. On the other hand if Mindist is greater than the distance from Q to its current best solution then the given MBR cannot contain a better solution. With this in mind, a sub-tree of the R-Tree can potentially contain the solution to Q if, and only if, the Mindist between Q and the MBR of that node is smaller than the current best solution. This is used as the main pruning heuristic, and is illustrated in figure 37.

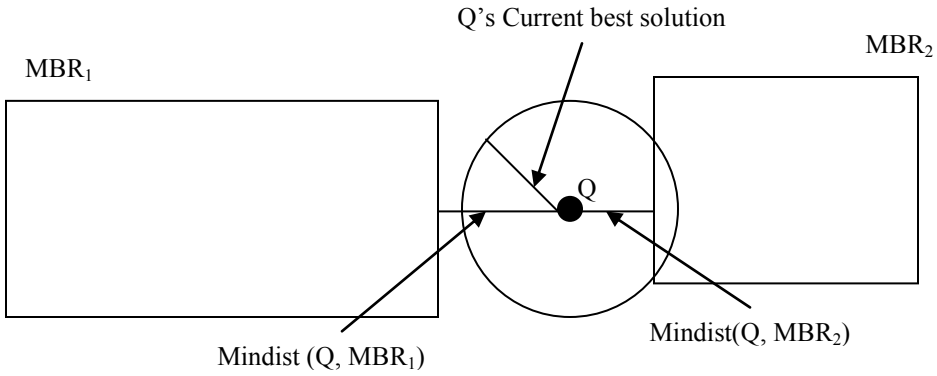



Figure 37. Illustration of Mindist and MBR pruning.

In figure 37 Q's current best solution is less then *Mindist (Q, $MBR_1$)* therefore $MBR_1$ cannot contain a better solution, whereas Q's Mindist to $MBR_2$ is less then Q's current best solution therefore $MBR_2$ could contain a better solution.

The only other item to consider is the order of MBR to search, known as traversal order. Controlling the search order may direct the search to a solution faster, as Q's nearest neighbors distance improves "gets smaller" then in theory more MBR should be pruned. Therefore, the search order should improve performance. In the past two

methods have been proposed: in [84] a depth-first approach is used; while [42] uses a ``best-first'' method involving the use of a priority queue to store all nodes not yet traversed then choosing the best one to traverse next. A slight improvement "best-first" involves using Mindist but also the distance to center of the MBR (hyper-rectangle) as search order.

This standard approach has an advantage in that it guarantees an exact match for KNN, but with a high penalty. As discussed in previous sections, R-Tree built with data in high dimensional space (above 16 dimensions) suffers from a high degree of overlap in the MBR. It can be shown that at the top level (root) of the tree all of the MBR overlaps each dimension by nearly 99% and even at the level above the leaf nodes the overlap is still 95%, therefore this approach ends up searching nearly 100% of the tree, an example can be seen in table 1 section 2.3.

When examining the definition of Mindist, it appears to be a very good solution to pruning branches within R-Tree, yet because of the nature of high dimensional data and the "curse of dimensionality" this method fails to prove very useful as was just discussed. Before this method is totally discarded it should be noted that Mindist does provide some useful information, such as a guaranteed bound of the approximate solution. For instance, if during a traversal a node with given MBR is pruned then *Mindist (Q, MBR)* will provide a lower bound for the actual solution. Therefore, it is possible to calculate the lower bound of the actual solution as formula 7, note that MBR is the bounding region of a node that has been pruned or not yet visited [66].

$$\text{Min}_{\text{MBR}}\ \textit{MinDist(q, MBR)} \qquad (7)$$

This guaranteed bound can be very useful for on-line query processing. For example, as each node is traversed calculate the Mindist to each MBR (for each branch) before they are traversed, thereby providing a bound of the solution during query execution in return giving feedback to the user. Of course, this online query with guaranteed bounds works best with "best-first" traversal. Because the traversal order is based on increasing Mindist of the MBR, the lower bound will steadily increase therefore the bound will continue to get tighter. The drawback to on-line queries using guaranteed bounds has to do with the fact that it works best with the "best-first" traversal method and does not work very well with "depth-first" traversal. To use "best-first" would require enumerating all branches therefore requiring a much larger amount of main memory then "depth-first". Because of this a query uses "depth-first" traversal, but with the addition modification of involving the center distance to the MBR.

Because the traversal is "depth-first", some children of the root node will not be visited until very late in the search. It should be pointed out that the root node tends to have a large MBR, and therefore they usually have a Mindist closer to or more commonly equal to zero which indicates that a query point is fully contained in the MBR. This is because at the root all MBR usually overlap each other by 99%, therefore any query point will be contained in all the MBR. Also, since the bounding region of a node is always enclosed in the bounding region of its parent node, it can be shown that Mindist of the parent's bounding region is never going to be larger than that of the current node. Therefore, the lower bound only changes when the next child of the root node is visited. For high dimensional queries, it may take a long time before the lower bound changes, as

a large portion of the tree is traversed. This can be seen in figure 38, which shows two examples of changes in the guaranteed bound (as well as the current best solution) during a NN search for a 150,000 point, 32 dimension data set, using best-first traversal [66].

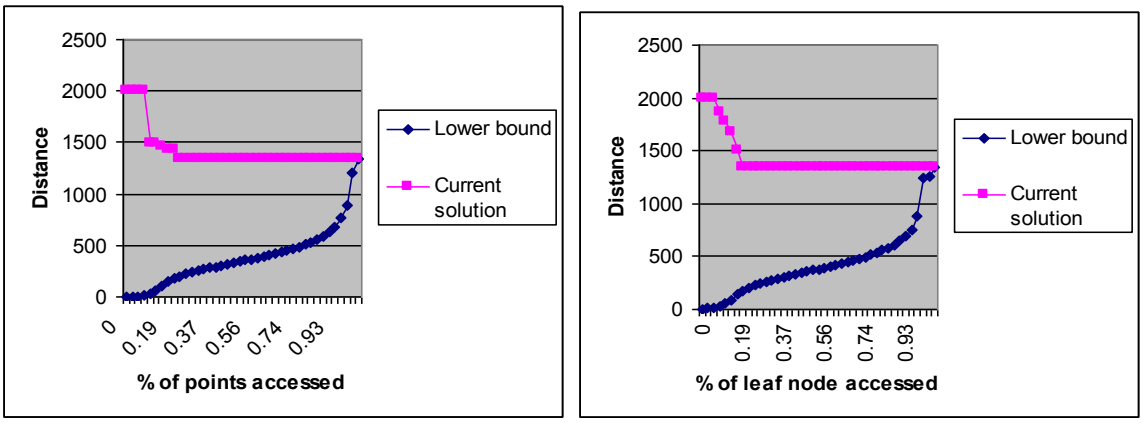



Figure 38. Illustrates lower bounds for on-line NN query

Based on the standard NN search and the "Curse of Dimensionality" one can easily surmise that a standard search of high dimensional space will lead to a complete search of the space. Yet from the previous observations it is possible to derive new heuristics which could be used for ANN queries. These new pruning heuristics sacrifice the NN exactness guarantee for efficiency.

## 5.2 ANN Search Heuristics

Approximate nearest neighbor search involves pruning based on some approximation or estimate as to the likelihood that a better solution exists. By relaxing the requirement of an exact solution and being willing to accept an approximant solution,

it may be possible to apply a new heuristic to the tree traversal in an attempt to prune branches that will have less of a chance at improving the solution.

### 5.2.1 Methods

The following sections describe several heuristics that can be used during the traversal to prune branches which may be less likely to improve the current solution. Each of these heuristics attempts to estimate the likelihood of an improvement if a node is visited.

#### 5.2.1.1 Heuristics Using Mindist

The following heuristics are built on the concept of Mindist in an attempt to prune branches based on an estimate as to whether a given MBR will improve the current solution.

##### 5.2.1.1.1 Heuristic R_MCB

Heuristic MCB (Mindist/currentbest) looks at an estimate of how much the radius (hyper-sphere) as defined by Q and its current solution extends into the MBR (hyper-rectangle) see figure 39. Pruning accrues based on the following rule, a branch is pruned from the search if *Mindist (Q, MBR) / CurrentBest* < $\alpha_{R\text{-}MCB}$ where $\alpha_{\text{R-MCB}}$ is a user defined parameter. Small $\alpha_{\text{R-MCB}}$ has the effect of pruning very few branches leading to a search of the majority of the tree, improving accuracy, but still with the possibility of not finding the exact NN. A larger $\alpha_{\text{R-MCB}}$ has the effect of pruning more of the branches decreasing the search space and decreasing the accuracy.

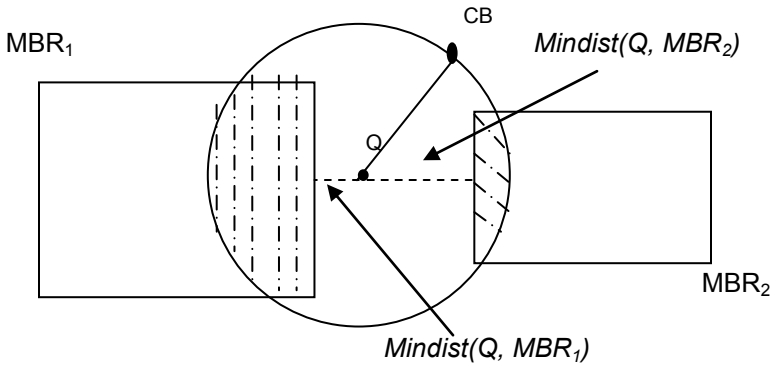



Figure 39. Example of Heuristic R_MCB

This heuristic guarantees a solution within the user defined value of $\alpha_{R\text{-}MCB}$ of the true solution, yet it can be shown that the heuristic is not very useful in high dimensional space.

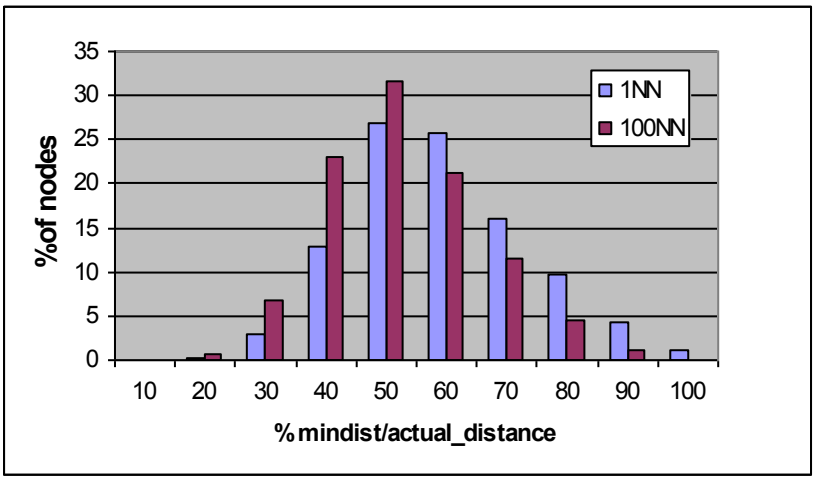



Figure 40. Ratio of Mindist/actual NN distance

In Figure 40, which plots the ratio between Mindist and the actual NN distance, notice that if $\alpha_{R\text{-}MCB}$ is small, then only a relatively small number of branches are pruned. Therefore, if this approximate algorithm needs to be efficient (high degree of branch pruning), the guaranteed bound will be very low. For example, consider a single NN query for a given query Q, looking at the numbers from figure 40 if the user wants half of the tree to be pruned (not searched), then $\alpha_{R\text{-}MCB}$ will have to be set at a minimum of 0.5. This can be seen in figure 40, as around 50% of the nodes have the ratio *Mindist / ActualSolution > 50%.* It should also be pointed out that the actual solution is always closer than the current solution, which implies $\alpha_{R\text{-}MCB}$ may have to be set even higher [66].

#### 5.2.1.1.2 Heuristic R_MAX

A variation on Heuristic R_MCB known as Heuristic R_MAX, takes into consideration two aspects of query point Q and a given MBR, involving Mindist and the approximate size of the MBR. This heuristic computes the Mindist, and a new variable known as Maxdist, which is defined as the maximum distance from Q to the furthest corner of the hyper-rectangle see formula 8.

$$Maxdist(q, R) = \sqrt{\sum_{i=1}^{d} \max\left(\left(q_i - R_{low_i}\right)^2, \left(q_i - R_{high_i}\right)^2\right)} \qquad (8)$$

This heuristic considers how far Q is from the closest hyper-surface of the MBR (Mindist) and how much the hyper-sphere defined by Q and its current solution extends into the hyper-rectangle (MBR) as well as the size of the hyper-rectangle. This can be summarized in figure 41, where $\alpha_{R\_MAX}$ is a user defined parameter.

$$(\text{CurrentBest} - \text{MinDist}) / (\text{MaxDist} - \text{MinDist}) < \alpha_{R\_MAX}$$

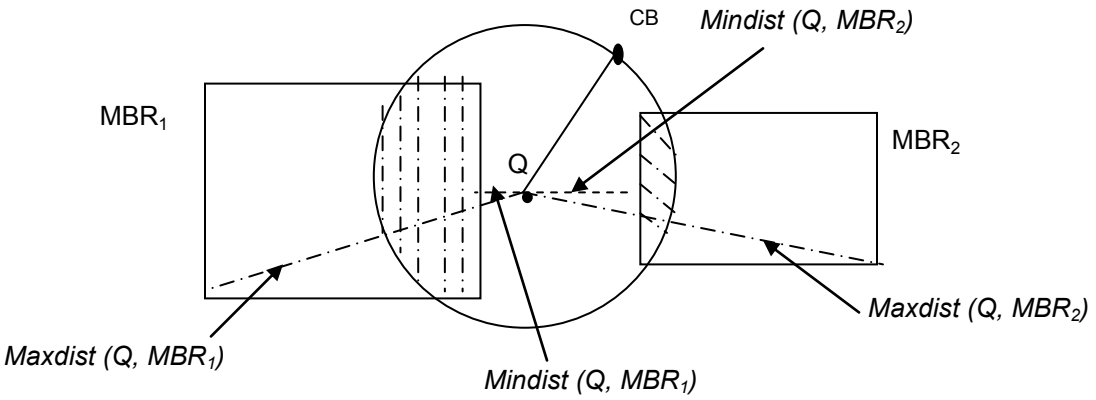



Figure 41. Illustrating Heuristic R_MAX

Like the previous heuristic a branch is pruned based on whether the *(CurrentBest – MinDist) / (MaxDist – MinDist)* is less then $\alpha_{R\_MAX}$. One can see from figure 41 that if Q is further from a given MBR (see $MBR_2$) then the value returned from *(CurrentBest – MinDist) / (MaxDist – MinDist)* will be small and therefore not as likely to contain a better solution (improvement). On the other hand if Q is very close to a given MBR such as $MBR_1$ in figure 41, then the distance from Q to the current best solution becomes more important in deciding whether to prune this branch (MBR). One can draw a simple conclusion that this formula is an attempt to estimate the overlap of the hyper-sphere

defined by Q and its current best solution and a given hyper-rectangle defined by a MBR. With this heuristic one can see, that the larger the intersection between a given MBR hyper-rectangle and Q's hyper-sphere, the more likely that an improved NN might exist in a given MBR. Still this heuristic like its predecessor does not guarantee that a better solution does not exist in $MBR_2$.

#### 5.2.1.2 Estimation of Probability of Improvement

If it were possible to better estimate how much the hyper-sphere defined by Q and its current best solution, overlaps a given MBR, then it might be a better estimate as to the likelihood of a better solution. One can easily see that simply traversing throughout the tree during a search is not useful in the sense that it does not improve the solution. A better estimate of which of the nodes is not likely to be helpful (most likely do not contain a better solution) could help reduce the search space. To put it in another way, if it was possible to make a prediction as to the probability that in a given node (with MBR) contains a point P such that *dist (p, q) < CurrentBest* then pruning could be based on this prediction. Any point that satisfies this condition lies in a region represented by the intersection of the hyper-rectangle and the hyper-sphere. This is shown in figure 42, where Q is the query point, CB is the current best solution, and the shaded intersection is the only region where a better solution may be found. Therefore, accurate estimates of this intersection (volume) could be used as an estimated probability of improvement.

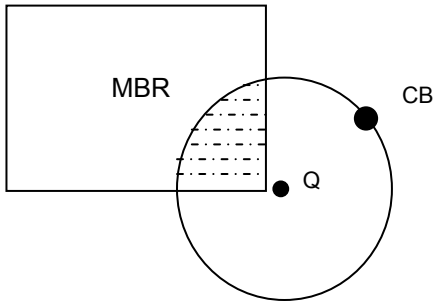



Figure 42. Estimation of volume for probability measures

One can easily conclude within reason that if it were possible to estimate the volume of this intersection then an estimate to the likelihood that the MBR contains a better solution than the current best (CB) solution.

#### 5.2.1.2.1 Heuristic MC_R

It is possible in theory to calculate the volume of the region defined by the intersection of the hyper-sphere and a given hyper-rectangle by integration, yet as the dimensionality increases this becomes intractable. To estimate this volume another method that does not involve complex integration needs to be used, this can be found in the area of statistics, known as Monte Carlo approximation [50, 83]. To summarize this method, one generates K random points contained in the given MBR then computes the distance from Q to each of the K points, counting the number that are less then the current solution. This provides an estimation of the volume (intersection of the hyper-rectangle and hyper-sphere defined by Q and its current best solution) which can be used as a probability for the likelihood that the given MBR contains an improvement. This

can serve as a means to prune branches if the probability that a better solution exists in its descendents is too small. For example, a given MBR is pruned if after generating K uniformly distribution random points, contained within the MBR, $\alpha_{MC\_R}$ / K points generated are closer to Q than the current best solution. In this case $\alpha_{MC\_R}$ becomes the estimate of the intersection.

This method does have a drawback in that it incurs extra CPU cost as points need to be generated and the distances from these points to Q need to be computed. If K is not too large, then the extra CPU cost is still likely to be small compared to the disk access cost. On the other hand, if K is too small, then the error in estimation can be significant enough, leading to very little pruning [66].

#### 5.2.1.2.2 Heuristic MC_H

Notice that in R-Tree based structures the bounding regions are minimum bounding rectangles. Each branch of the tree is in turn bounding a sub-tree, which in turn bounds a set of sub-trees until the number points is less than or equal to that which can be stored in a leaf node is reached. If one considers the "Curse of Dimensionality" and the number of points that are stored in a leaf node, and applies some simple logic, namely the pigeon hole principle, one realizes that all the points contained in the lowest level MBR contribute to the overall definition of the MBR. That is nearly every point contained in a leaf node helps to define one or more edges of the MBR. This means that most likely all the points must be on the hyper-surface of the MBR. This was verified by running a simple test counting the number of leaf nodes that have 1 or more points that do not lie on the MBR's surface (meaning the point does not contribute to any edge of the MBR).

Table 4 shows the results for this test using a tree with a 4K page with a fan out 15 (originally 15 was used as it corresponded to the fan out of the internal nodes), which was used for earlier testing. This test showed that better than 99% of the points lie on the MBR's surface with only around 7.5% of the leaf nodes having at least one point not on the surface.

Table 4. Showing results for different data distributions, showing the number of leaf nodes with all points on the MBR, (fan out set to 15).

| Data Distribution | No. of Points | Number of Leaf Nodes in Tree | Number of Leaf Nodes with all points on MBR | Percentage of Leaf Node with all Points on MBR | Number of Points Contributing to a MBR | Percentage of Points Contributing to a MBR |
|---|---|---|---|---|---|---|
| Uniformed | 100,000 | 8,229 | 7,804 | 94.84% | 99,566 | 99.57% |
| Clustered | 100,000 | 8,244 | 7,663 | 92.95% | 99,378 | 99.38% |
| Gaussian | 100,000 | 7,971 | 7,374 | 92.51% | 99,377 | 99.38% |
| Real | 66,000 | 6,077 | 5,612 | 92.35% | 65,459 | 99.18% |

Later it was realized that considerable space was being wasted in the leaf nodes (at least 50%), so to use this wasted the space, the leaf nodes were modified to store twice as many points, as described in Chapter 3. Because of this change, the same tests were run using a fan out of 30, results of which can be seen in table 5. This was done to verify that the hypotheses that most of the points were still on the MBR (hyper-surface). Table 5 shows that between 92.4% - 96.3% of the points lie on the MBR's surface. This test shows that 22% or more of the leaf nodes now have at least one point not on the surface as compared to table 4, with only 7.5% of the leaf nodes had at least one point not on the surface.

Table 5. Showing results for different data distributions, showing the number of leaf nodes with all points on the MBR, (fan out set to 30).

| Data Distribution | No. of Points | Number of Leaf Nodes in Tree | Number of Leaf Nodes with all points on MBR | Percentage of Leaf Node with all Points on MBR | Number of Points Contributing to a MBR | Percentage of Points Contributing to a MBR |
|---|---|---|---|---|---|---|
| Uniformed | 100,000 | 4,266 | 1,465 | 34.34% | 94,772 | 94.77% |
| Clustered | 100,000 | 4,277 | 1,187 | 27.75% | 92,729 | 92.73% |
| Gaussian | 100,000 | 4,084 | 902 | 22.09% | 92,461 | 92.46% |
| Real | 66,000 | 2,982 | 1,726 | 57.88% | 63,512 | 96.23% |

Closer examination of these two tables reveals that as the fan out increases the number of points that do not lie on the surface increases, this is supported by the pigeonhole principle, in that we have more points so each point will be less likely to contribute to an edge of the MBR.

Based on the results from these tests it was realized that a modified version of heuristic MC_R could be derived known as heuristic MC_H. Because 92% to 99% of the points, most likely lie on the hyper-surface of the leaf MBR one realizes that a volume estimate is not needed, but rather that an estimate of the size of the hyper-surface may suffice. If one can locate the hyper-surface closest to the query point and apply the Monte Carlo technique on the hyper-surface it would be possible to have an estimate of the area of the surface see figure 43. With this in mind the following heuristic was defined, a MBR is pruned if after generating K random points from a uniform distribution from within the *d-1* dimension hyper-surface closest to Q, less than $\alpha_{MC\_H}$ / K points generated are closer to Q than the current best solution (where $\alpha_{MC\_H}$ is a user-defined parameter).

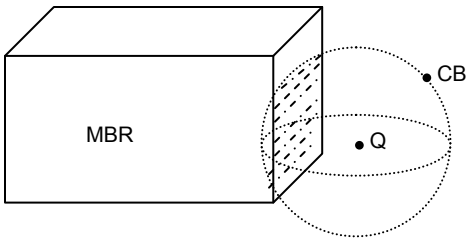



Figure 43. Illustration of heuristic MC_H

Because the bounding regions in R-Tree are hyper-rectangles, finding the closest hyper-surface for a query point is straightforward. Since each *d*-dimensional MBR is represented by the two points at the opposite corner of the diagonal, a *(d – l)* - dimension hyper-plane is represented by equating the value of one of the d dimensions for the two corner points [66].

As with heuristic MC_R, this method also suffers from the high cost of generating K random points, now on the surface, closest to Q and again computing the distance between the random points and Q. Again if K is small then the error estimate will be high and little pruning will occur, whereas if K is too large then the CPU cost becomes an issue.

#### 5.2.1.3 Estimation using Maxdist and Mindist

The main drawback of the heuristics that rely on Monte Carlo approach is the extra CPU cost that is needed to generate test points (especially when K is large – which is necessary to make the estimation more accurate) and the extra distance computation needed. Therefore, I looked at alternate heuristics in an attempt to use other means to determine whether a certain node is worth visiting or not.

#### 5.2.1.3.1 Alternate Heuristics Involving Maxdist and Mindist

One additional approach would be to look at the range of possible distances between Q and all MBR contained in an internal node as well as minimum Mindist and Maxdist. Given Mindist and Maxdist, it may be possible to evaluate how likely there will be an improvement by visiting a given node. If the current solution is much closer than Mindist then Maxdist, it implies a large portion of the bounding region is further away from the query point than the current best solution which is illustrated in figure 44.

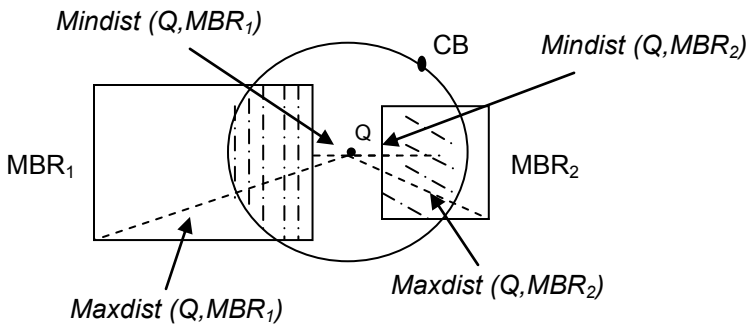



Figure 44. Illustration of heuristic using Maxdist and Mindist

Here in figure 44 the hyper-sphere defined by Q and CB intersects both $MBR_1$ and $MBR_2$, denoted by the shaded regions in figure 37 these are where a potential solution possibly exists. Notice that while the Mindist between Q and $MBR_1$ and $MBR_2$ are the same, the proportion of the volume of the shaded intersection regions as compared to the volume of $MBR_1$ and $MBR_2$ is very different. In $MBR_2$, the shaded region is much greater than half of the total volume, while in $MBR_1$ it is less than 1/3.

#### 5.2.1.3.2 Heuristic R_MINMAX

Building on previous heuristics R_MAX (section 5.2.1.1.2) and the understanding of high dimensional space another heuristic can be derived. If we take the previous heuristic R_MAX and the discussion above as well as part of the discussion from heuristic MC_H concerning the points being on the surface of the hyper-rectangle, and then consider the minimum Maxdist (known as *minMaxdist),* defined as the upper bound of the minimum distance between a point in the node and the query point together it may be possible to derive a new heuristic. This makes use of the notion that the region is minimum bounding and that at least one point must be on the boundary. Therefore instead of comparing Mindist and Maxdist, now compare Mindist and *minMaxdist*. This leads to the following heuristic known as RMINMAX illustrated in figure 45 and formula 9. A node is pruned if the value from formula 9 is smaller than a certain fraction $\alpha_{R\_MINMAX}$ again defined by the user

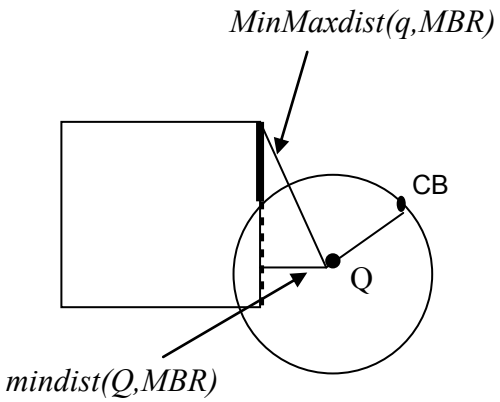



Figure 45. Illustration of heuristic R_MINMAX

$$\frac{CurrentBest - mindist}{minmaxdist - mindist} < \alpha_{R\_MINMAX} \qquad (9)$$

This heuristic looks at the surface closest to the query point Q, intersected by the hyper-sphere defined by Q and CB, this is seen in figure 45 (in 2-D) by the dashed line. It attempts to approximate the ratio of the length of the line that is outside the circle (denoting the current best solution) to that of the overall solution [66]. This heuristic, in turn, prunes only those branches that would not be as likely to improve the current solution.

#### 5.2.1.3.3 Heuristic R2_MINMAX

Building on the previous heuristics involving the ratio of Mindist and Maxdist with the current best solution, a second heuristic may prove to be useful. Rather then attempting to factor in Mindist, consider only the ratio of current best to the minimum Maxdist. Simply put, prune a node if *CurrentBest / (minMaxdist)* is less than a user defined parameter known as $\alpha_{R2\_MINMAX}$. As seen in formula 10.

$$\frac{CurrentBest}{minmaxdist} < \alpha_{R2\_MINMAX} \qquad (10)$$

This is basically a simplified version of heuristic R_MINMAX only the distance from Q to given MBR is no longer considered. While experimenting with this heuristic, I realized that Mindist was relatively small, most likely due to the high overlapping of

MBR, it was surmised the Mindist may have little effect on the pruning. Most likely this heuristic will not perform very well.

### 5.3 Search Heuristic Experiments

After implementing the above heuristics, I ran a number of experiments involving different dimensions, data set sizes and distributions. These were run with randomly generated data sets for 4, 8, 16, and 32 dimensions respectively, of data set sizes ranging 50,000 point, up to 250,000 points in increments of 50,000. The data sets followed the distributions as described in chapter 3 Experimental/Implementation Results. For all of the experiments the page size was fixed at 4KB with minimum space utilization (leaf, internal node storage) set to 40%. For test data, I generated 50 query points from the appropriate dimensionality. Finally, I run the various algorithm with different parameters (for Monte Carlo heuristics, $k$ = 2,000). For evaluation purpose, the averages of the performance data will be presented.

In terms of results, I wanted to measure both efficiency and accuracy. For efficiency, I measured the number of leaf nodes visited (as it dominates the running time). For accuracy, I have three different ways of measuring accuracy:

- Recall. This represents the proportion of the actual result that where actually found. For example, for a 100-NN query, it may be that an approximation algorithm returns 100 points, on which 45 actually belongs to the true 100-NN. Then the recall in this case is 45% [67]
- Distance ratio. I measured the ratio of the distance between the solution found and the actual solution. For multiple NN queries, I return the result of the solution that

is furthest away from the query point. For example a 100-NN query, the distance ratio returns the ratio between following two numbers: the distance from $Q$ to the $100^{th}$ nearest neighbor that was located by the algorithm, and the distance from $Q$ to the actual $100^{th}$ nearest neighbor.

- Rank. The actual rank of the solution. For example in a 100-NN query, the $100^{th}$ nearest neighbor returned from the approximate algorithm may actually be the $127^{th}$ true nearest neighbor. Rank is a useful measure as it is less affected by the data distribution.

I first examined the tradeoffs between efficiency and accuracy in the nearest neighbor search. It is very easy to see that the more pruning that is performed, the more likely a good solution is missed and thus accuracy decreases. The level of pruning is determined by the parameters set in the algorithms (namely $\alpha$). So I vary $\alpha$ specific to each heuristic and repeated the experiments to see its effect. For this set of experiments, I will show only results for 8 dimensional and 32 dimensional data. Results obtained in 4 dimensional and 16 dimensional are similar to that of 8 dimensional and 32 dimensional respectively.

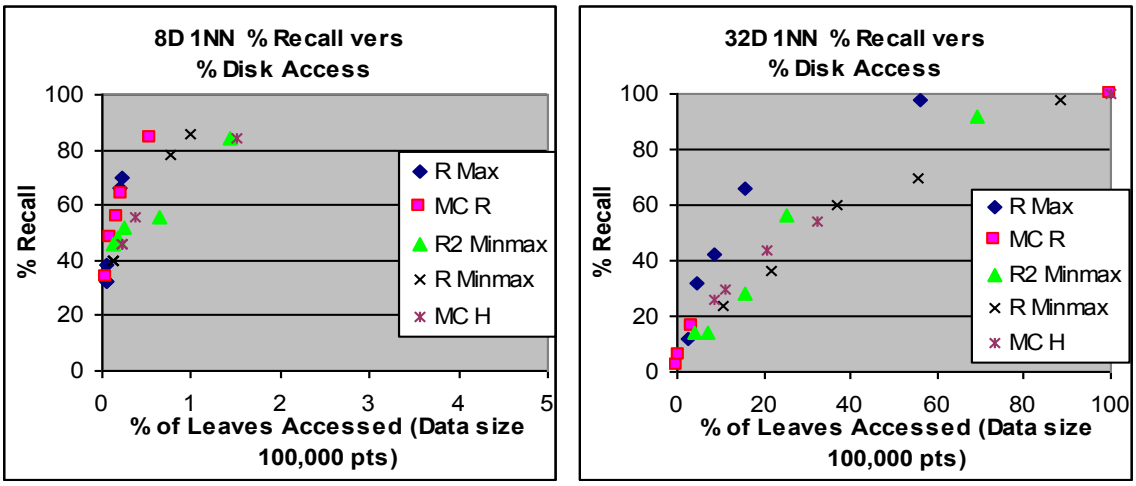



Figure 46. Recall vs. Leaf access, 8 dimensional / 32 dimensional data, 1 NN query

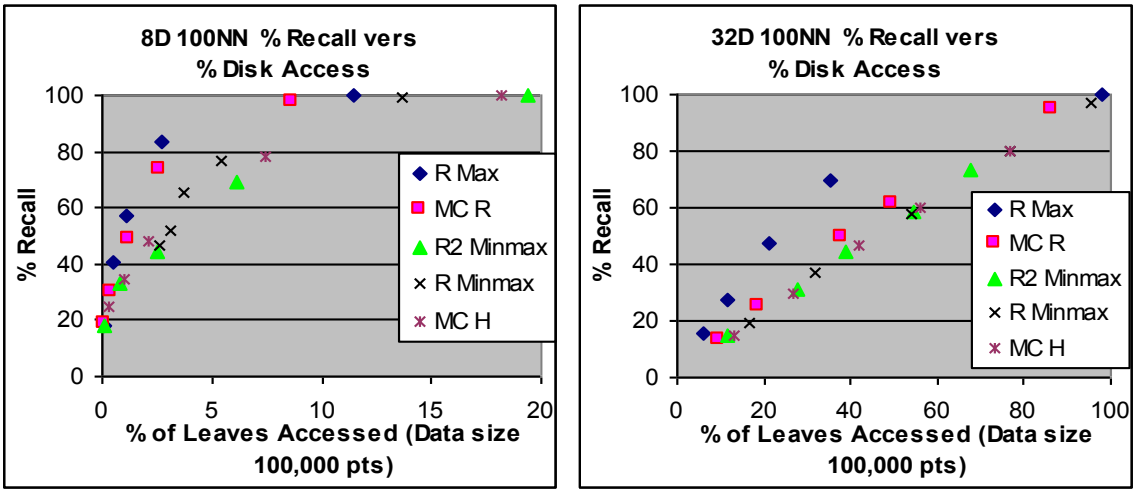



Figure 47. Recall vs. Leaf access, 8 dimensional / 32 dimensional data, 100 NN query

Figures 46 and 47 show the results of 1ANN and 100ANN queries for 8 dimensional and 32 dimensional data sets (100,000 data points). Ideally, we would like to have points on the graph that is on the top left corner. This represents higher level of recall with few leaf accesses. As seen from the figure, all algorithms perform well at 8 dimensional. However, in 32 dimensional we can see the trade-off between recall and

efficiency is more on a linear level. In this aspect, the algorithms do not provide significant advantages.

In terms of comparing between the heuristics, we see that R_MAX performs the best. For instance, in 32 dimensional and 100NN query, the algorithm needs to access less than 40% of the data to reach about 70% recall. This is quite a reasonable result, especially if at times the algorithm goes wrong, it is not too far off. I examined this notion in the next set of results.

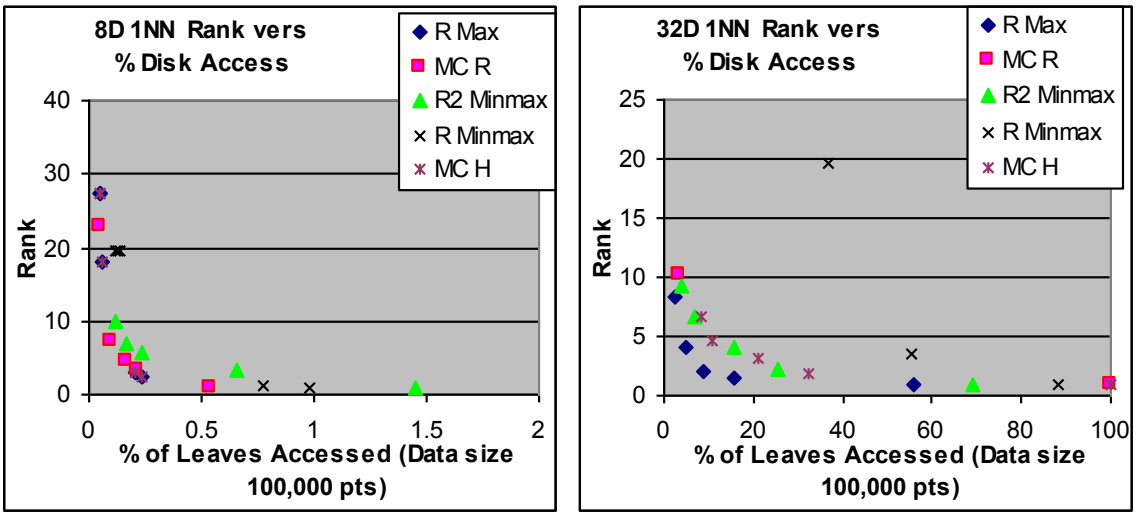



Figure 48. Rank vs. Leaf Access, 8 dimensional / 32 dimensional data, 1NN query

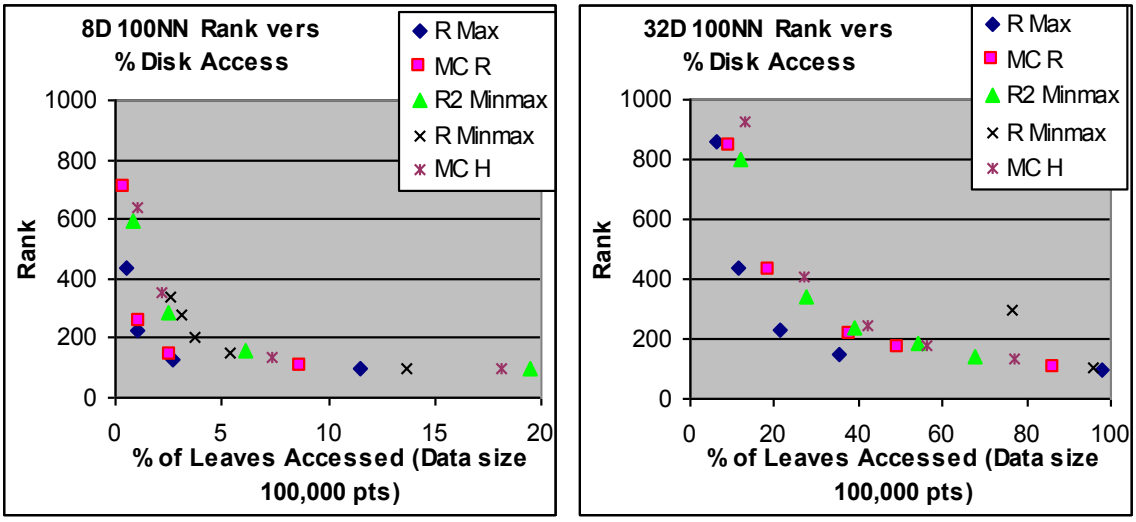



Figure 49. Rank vs. Leaf Access, 8 dimensional / 32 dimensional data, 100NN query

Figure 48 shows the relationship between the actual ranks for the approximate nearest neighbor respectively found by the algorithm and the efficiency. Similar results are shown in figure 49, only that it looks at the approximate $100^{th}$ nearest neighbor instead. Notice that the rank drops very fast (improves) when the number of leaf nodes accessed increases. Thus, a little bit more effort in accessing data can reap much better results. As the query is a 100-ANN query, the rank of the $100^{th}$ approximate nearest neighbor cannot be less than 100. Given this fact, the figures show that even with not too much disk IO, most heuristics returns objects that are less than twice the rank of the solution. These plots indicate that the heuristics performed fairly well. From the figures, again it can be seen that the R_MAX heuristic performs better than the others, returning objects with rank less than 200 with less than 40% of data accessed.

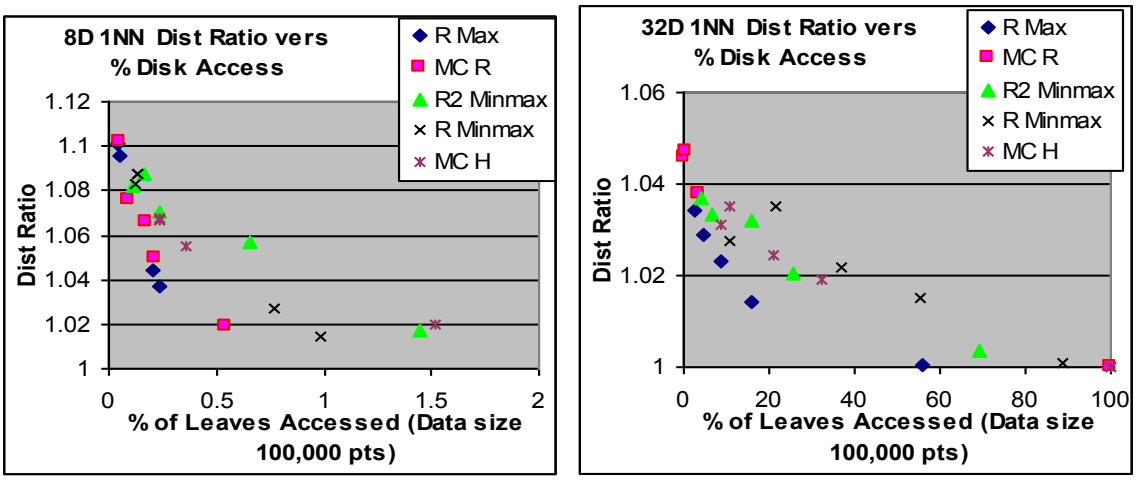



Figure 50. Distance ratio vs. leaf access, 8 dimensional / 32 dimensional data, 1NN query

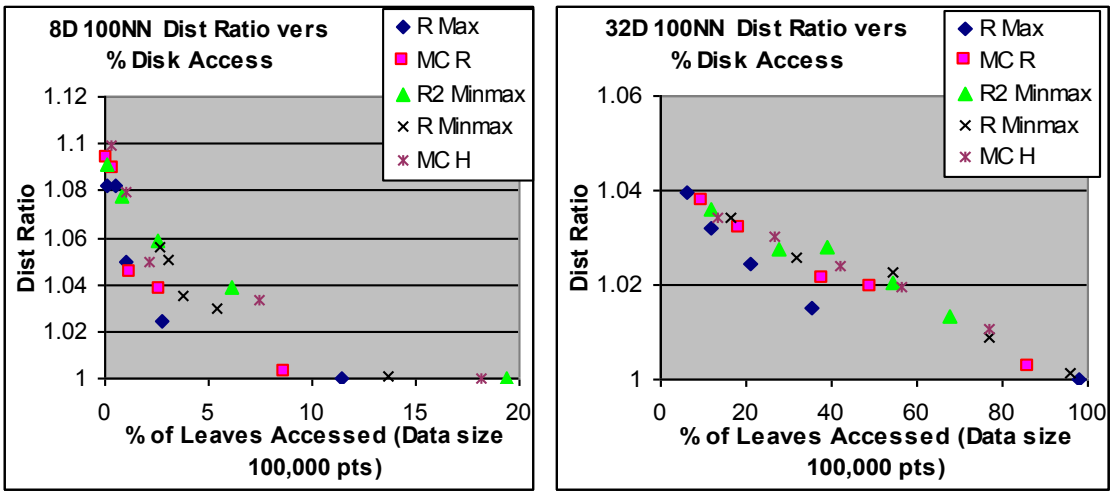



Figure 51. Distance ratio vs. leaf access, 8 dimensional / 32 dimensional data, 100NN query

The third measure for the performance of the approximate nearest neighbor algorithm is the distance ratio. This corresponds to the $(1 + \varepsilon)$ approximate nearest neighbor queries, where one would like to return an object within a certain given bound. Figure 49 and 50 shows the result for 8 dimensional / 32 dimensional data with 1NN and 100NN queries. Notice the difference in the result between 8 and 32 dimensional data,

especially in the case of 100NN. In 8 dimensional, one can see the performance increase very quickly with increases in the number of leaf nodes accessed. This means that a little bit more effort will pay much more in the end. However, in 32 dimensional the figure shows the decrease in the distance ratio comes at a price of linear increase in the number of pages accessed. Thus in terms of distance ratio, the R-Tree based methods do not provide a good solution to drastically improve performance. Compare the change in distance ratio against the rank. It can be seen that the rank decreases much faster than distance ratio. This suggests that while many algorithms are looking for points within a certain bound, it may turn out that we can have much more efficient algorithms finding points within a certain rank of the result. This is also feasible as the distance ratio is very dependent on the dimensionality of data and data distribution.

Another interesting observation is that algorithms that use *minmaxdist* (R_MINMAX, R2_MINMAX) do not perform well. It is a bit surprising, as one may think *minmaxdist* is a tighter bound and thus can prune more useful data and keep the useful ones.

### 5.4 Initial Conclusion of Using Heuristics with Alpha

A review of these experiments indicates that the heuristics given in this section perform very well given the nature of the problem. Some, of course, better than others and still some with additional costs that have to be taken into consideration. These experiments also show that these heuristics work for not only NN but also for KNN be it that they return approximate results, without any additional modifications.

## 5.5 Heuristic Parameter Selection

One common feature of all the heuristic discussed in the previous sections is the parameter $\alpha$, specific to each heuristic, which was described as user defined. Yet, this is misleading as to which user will know enough in advance to actually specify this value. In reality $\alpha$ is a parameter that would need to be specified to the heuristic at query time to get a desired performance result.

### 5.5.1 Deriving Alpha for Heuristics and Data Distributions

It should also be pointed out that the selection of $\alpha$ is dependent on three different factors common to all heuristics, these are the number of dimensions, number of points (size of the tree), and data distributions. These, dependencies can be seen in the following figures, found during past research [66]. Figure 52 shows how $\alpha$ has to adjust for the different heuristics as the number of dimensions increases from 4 dimensional to 32 dimensional for R-Trees with 100,000 points with uniformed data distribution. Here $\alpha$ for each heuristic was chosen in such a way as to provide the same efficiency and accuracy for different dimensionality. Figure 52 shows how $\alpha$ had to be varied for each heuristic, based on the number of dimensions.

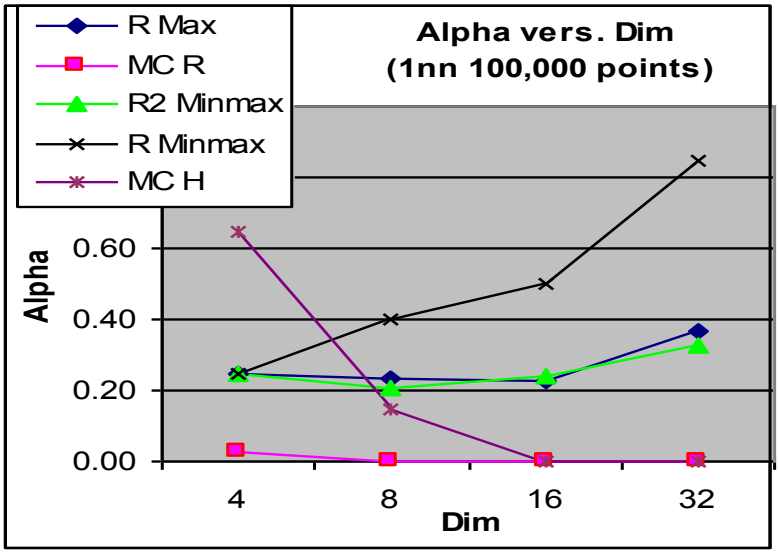



Figure 52. Effect of increasing dimensionality on α selection.

Figures 53, 54, 55 and 56 show how $\alpha_{R\_MAX}$ for heuristic R_MAX changes for increasing number of data points, this change is less dramatic as one can see, but the effect is there. To illustrate this, R-Trees with 5,000, 10,000, 20,000, 50,000 and 100,000 32 dimensional points, were built, $\alpha_{R\_MAX}$ was varied over a wide range for the different data distributions. Then the percent of tree access was plotted against $\alpha_{R\_MAX}$ for the different sizes of R-Tree. In these figures 53, 54, 55 and 56 the effect data set size has on the selection of $\alpha_{R\_MAX}$ can be seen, although the effect is small, it is there (a large difference in data set sizes had to be used to see it).

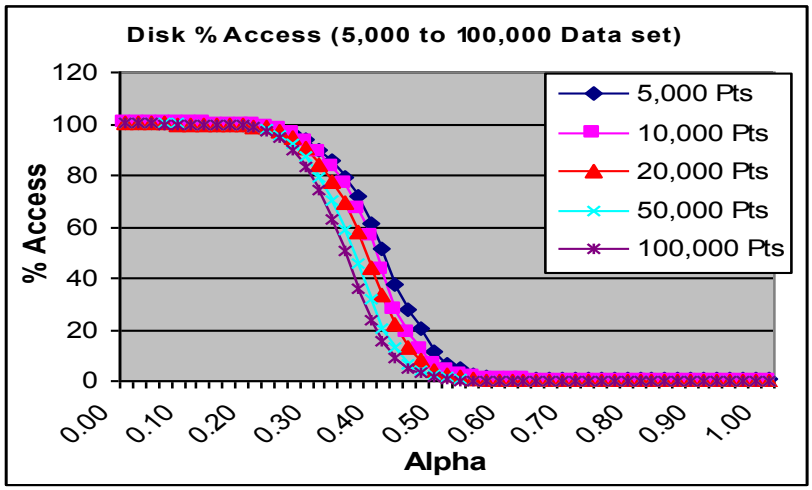



Figure 53. Uniformed data ranging from 5,000 to 100,000 data set sizes, plot of percent of access against Alpha.

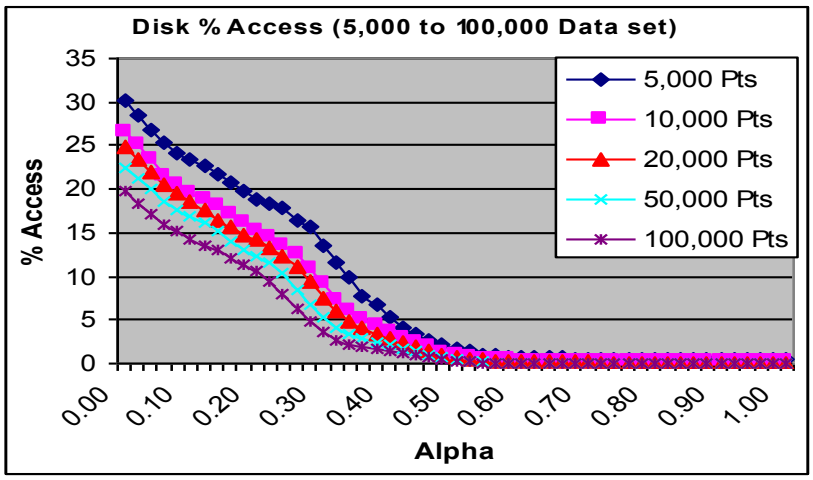



Figure 54. Clustered data ranging from 5,000 to 100,000 data set sizes, plot of percent of access against Alpha.

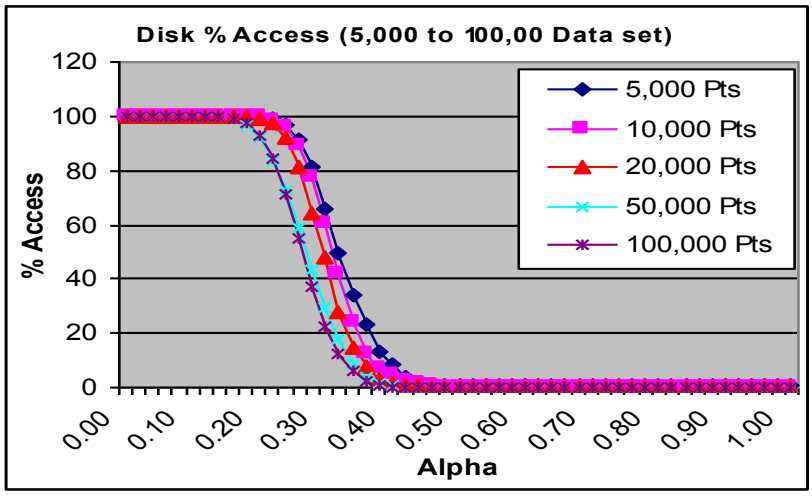



Figure 55. Gaussian data ranging from 5,000 to 100,000 data set sizes, plot of percent of access against Alpha.

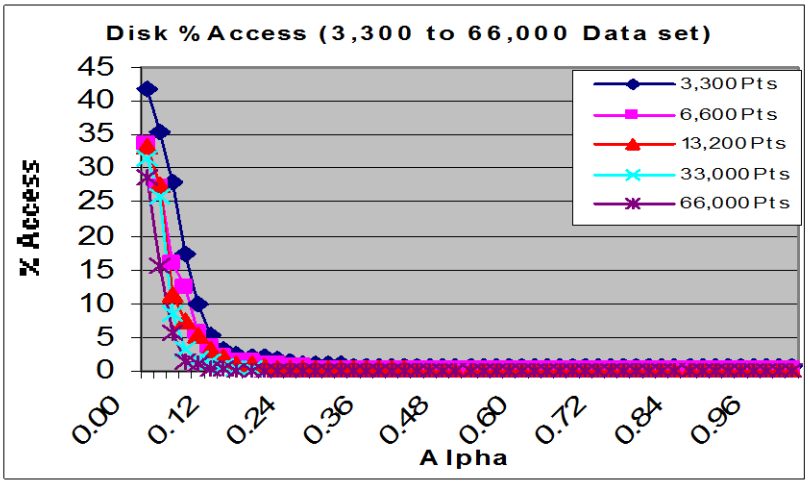



Figure 56. Real data ranging from 3,300 to 66,000 data set sizes, plot of percent of access against Alpha.

Another conclusion that one can draw from these simple tests illustrated in figures 53, 54, 55 and 56 is the effect of data distribution on the selection of $\alpha_{R\_MAX}$. Based on these figures one can plainly see that $\alpha_{R\_MAX}$ for Heuristic R_MAX must be adjusted for each data distribution, simply put, $\alpha_{R\_MAX}$ cannot be constant for all data set sizes of

varying dimension and data distributions. This leads to a question, how to determine $\alpha_{R\_MAX}$ for a given query, even though the number of dimensions is known in advance, as well as, the initial data set size. The main issue involves the data distribution which may not be known or worst could be a combination of distributions. Not only that but due to the dynamic nature of R-Tree the size could grow and shrink over time. Deriving $\alpha_{R\_MAX}$ becomes a very daunting task for heuristic R_MAX as well as the other heuristics.

### 5.5.2 Accuracy vs. Efficiency

It can be seen from previous sections that the selection of $\alpha$ for a given heuristic is dependent on the many factors, but what is not evident is the actual effect of $\alpha$ on the accuracy and efficiency. In all of the heuristics discussed in the previous section, $\alpha$ is the main factor in determining the accuracy and efficiency of a given R-Tree comprised of a given data set. The parameter $\alpha$ can be tuned or adjusted to a particular degree in turn trading efficiency for accuracy. The more aggressive the pruning the more likely better solutions will be missed. This can be seen in the following figure 57. Ideally, we would like to see results in the lower left corner of figure 57, basically low percent of accesses with low rank (accurate NN).

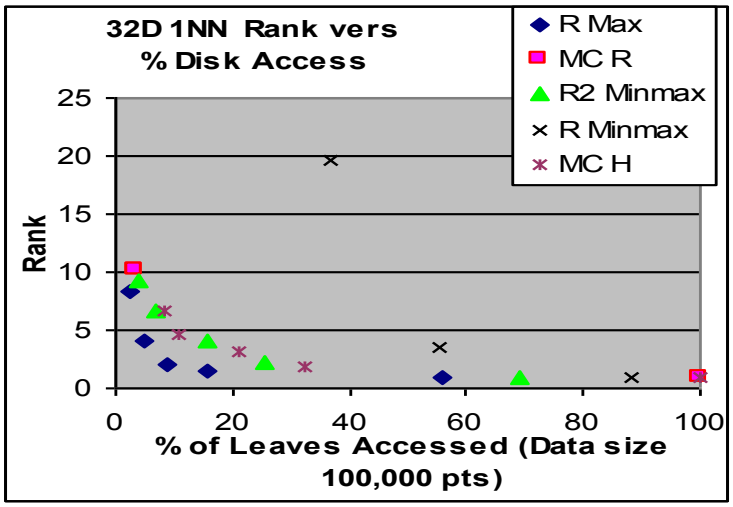



Figure 57. Rank vs. Leaf Access, uniformed 32 dimensional data, 1NN query

Figure 57 shows the relationship between the actual ranks for an approximate nearest neighbor found by the different heuristic and the efficiency. One can see that as the percent of access (leaf nodes read) decreases then the accuracy (rank) of the solution increases. At first glance one wants to draw a conclusion that $\alpha$ could be arithmetically derived but recall the previous sections figures 53 through 56 using different data sizes and distributions, a comparison of all of these figures contradicts this conclusion. Ideally one wants a low level of access with an exact NN solution, or an ANN solution within some given guarantee of accuracy. Therefore, $\alpha$ needs to be selected to provide both of these requirements [66].

## 5.6 ANN Search Heuristic Conclusion

In conclusion I have shown that relatively simple heuristics can be derived that will estimate the probability of a given branch improving the current solution found. Some of the heuristics when applied and tested on different data sets with increasing

dimensionality show an improvement that is reduced IO yet returned an ANN that was very close to exact NN. The drawback to the uses of heuristics for ANN is the parameter $\alpha$ which needs to be provided to each heuristic but is difficult to derive.

## 6. ANN R-Forest

Even though R-Tree suffers from the "Curse of Dimensionality" and many other researchers have all but given up on it, I believe that it may still be useful and given it's afore mentioned advantages (chapter 2 section 4) worth further research. Looking at the work in the previous chapter, leads one to believe that R-Tree still has potential. Using pruning heuristics I was able to get reasonable good results [66], which supported the premises that R-Tree is still a viable solution to ANN in high dimensional space. But the previous work on pruning heuristics has its own complexity, for example how does one choose a value for $\alpha$ that provides both accuracy and efficiency, but also works well under all data distributions, increasing dimension and increasing number of points. While researching methods to derive $\alpha$, it was realized that R-Trees problems stem from its inability to partition the space even though it is a branch and bound data structure.

The weakness of R-Tree comes from the "Curse of Dimensionality" as the number of dimensions increase, the ability of a single R-Tree to sub-divide the search space becomes nearly impossible. For example, take an R-Tree being built with data points having 32 dimensions, since its leaf nodes hold a fixed number of points, they will need to be split into two nodes as they fill up when new points are inserted. When a leaf node must be split an additional leaf node will be created typically having half the points of the original node. The two nodes will be split along a single dimension chosen by an

algorithm that attempts to evenly distribute the points [44]. This node splitting can trigger future splitting of parent MBR which will propagate up the tree to the root. Now as more points are inserted, split dimensions formed by a node split are allowed to grow, which means that any given MBR at the top of the tree can cover nearly the entire domain of the search space. In figure 58 on the left, I have illustrated a search domain represented by the outer most edges of the three MBR rectangles, (represented by different dashed lines). These three MBR are at the highest levels on the R-Tree and are the results of inserting the points into R-Tree (on the right). They overlap each other and in this case each nearly covers the entire domain, thereby showing the limitations of R-Tree.

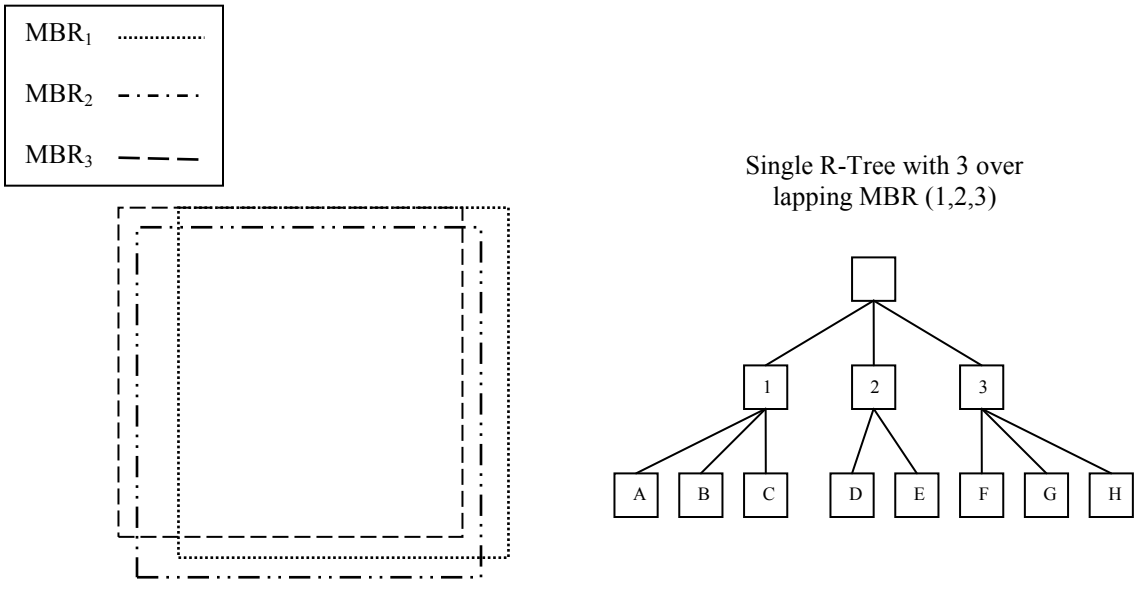



Figure 58. Illustration of a search domain, on the left and the corresponding R-Tree on the right

## 6.1 Forest of R-Trees

The model I will present chapter continues to use R-Tree even with this apparent limitation, by using a set of indices each of which contains a sub-set of the points in the domain space. Rather than building a single R-Tree index over the entire domain space I will now build a set of disjoint R-Trees each having a sub-set of points, which when taken as a whole (the union) covers the entire domain space. Now given a 32 dimension example, with any number of points, the union of the root's MBRs from all the R-Tree indices will cover the entire domain space. This set of disjointed R-Trees will be known as a Forest of R-Trees or R-Forest, which is illustrated in figures 59 and 60 where labels 0 to N, represent $N^{th}$ R-Trees. With this model, each R-Tree is now guaranteed to not overlap any other R-Tree in a few dimensions but not all. In figure 59, I am showing the entire search domain which is sub-divided on one dimension into three sub-sets, with the points in each sub-set being stored in their respective R-Tree indices (labeled 0 to N).

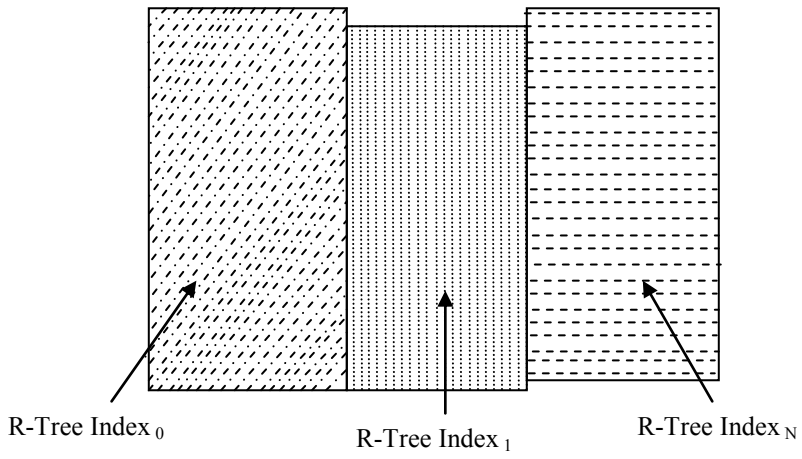



Figure 59. Search domain now divided into 3 regions disjointed on one dimension.

In figure 60 below, I now have three R-Trees each with a sub-set of points disjointed on the split dimension.

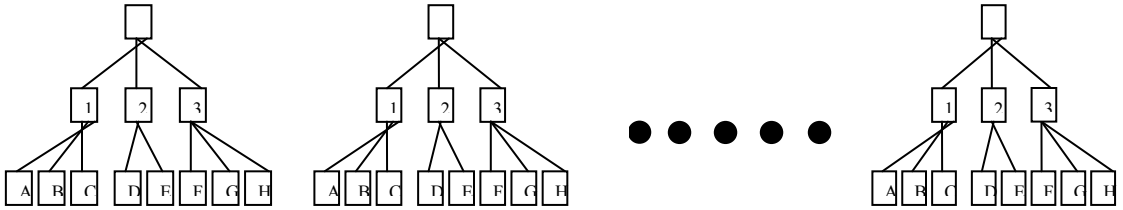



Figure 60. R-Forest over search domain, show splits on a single dimension

By using this, it is highly likely that for any query point, at least one or more of the bounding regions at the top will have a "large" distance along the splitting dimension, making it more likely that pruning can take place, and giving me the ability to derive additional pruning heuristics and even provide feedback, with this I have the following advantages:

- By having non-overlapping bounding regions at the top levels and internal to each R-Tree, pruning becomes feasible
- Traversal of the indices is improved, as I can order the MBR based on the query point distance to the MBR.
- I will be able to provide a bound on the quality of the ANN returned.

### 6.2 Disjoint Sub-Set

To build R-Forest, a set of R-Tree indices needs to be defined each over a sub-space of the overall domain. The first choice would be to split all dimensions but this would require $2^d$ R-Tree indices, which is exponential in the number of *d* dimensions. I therefore must choose a sub-set of the dimensions and split each into *B* non-overlapping

boundaries to keep the R-Tree indices growth rate manageable I chose a maximum of *log(d)* dimensions, which leads to a total of $O(B^{log(d)}) = O(d)$ number of R-Trees. For testing purposes the actual number of dimensions to split on is set as a parameter at R-Forest creation, this gave me the ability to test different configurations without having to re-write the implementation.

### 6.3 Construction of Forest

Construction an R-Forest involves the following steps to be discussed in further detail in the next sections.

1. Dimension Selection
2. Boundary Selection
3. Maintaining the Disjoint Set
4. Index Selection
5. Insertion of points into an R-Tree index

Figure 61. Steps to Build R-Forest

The first four are newly introduced to the R-Forest, their purpose is to define the disjointed R-Tree indices, select which index will house a given point and maintain the disjointed indices. It should be noted that the first three steps need only be preformed once at the beginning of the initialization of the R-Forest, with the last two steps being repeated as new points are inserted.

### 6.3.1 Dimension Selection

The first step in building R-Forest is selecting the set of dimensions to split on. If I have some domain knowledge I could select dimensions in such a way as to evenly distribute points between the indices. However it is better to have a generic system so I assume no domain knowledge is present. At first thought, one could randomly choose the dimensions to split on, which under some distributions may work well, but I will be working toward a generic approach to this problem, therefore the dimension selection process needs to not depend on chance. Because of this I choose to use correlation analysis to find a set of dimensions to split on. To do this I use the following method. First, select a random sample (for example 1,000 points) set form *S* the initial data set being loaded. Then find the correlation of every pair of dimensions in the sample set and attempt to choose pairs of dimensions that are weakly correlated. It seems reasonable to use weak correlation, as it should provide a means to evenly distribute points between indices in R-Forest, where as strongly correlated might have a tendency to unbalance the tree by pushing points into a few indices. In the end I want at most *log(d)* dimensions that are pair wise weakly correlated, which will become the bases for the disjoint sub-sets.

### 6.3.2 Boundary Selection

In the previous section I discussed selecting the number of dimensions to split on, the next question is how many splits per dimension, in other words do I split each dimension into two or more regions with each region having a fixed boundary. For testing purposes I decided that the number of boundaries would not be fixed, meaning I

would not hard code this to say two regions per dimension, rather I choose to implement variable number of regions as a parameter passed at the time the R-Forest is being created. This parameter defines how many times I will split within each dimension, it should be noted all split dimensions have the same number of regions. There is support for using more than one boundary, for example if I use two boundaries then I have three regions, a point in one region will in most cases have one other region further away, as illustrated in figure 62 therefore increasing the distance between a point and neighboring MBR. This same concept bisecting versus trisecting can be found in mathematics and computer science [24, 46, 108] hence I chose to usually trisect split dimension. To define this boundary(s) per region, I use the same sample sub-set used to find the split dimensions, and within each split dimensions I locate B boundaries that evenly distributes points into the regions, in turn these become the fixed boundaries covering the entire domain of the split dimension. In figure 61 dimension $D_1$would be split into three regions ($Region_1$ $-\infty$ > and <= $B_1$, $Region_2$ $B_1$ > and <= $B_2$ , $Region_3$ $B_2$ > and <= $\infty$) having boundaries $B_1$ and $B_2$ which stored in R-Tree $Index_0$, $Index_1$ and $Index_2$.

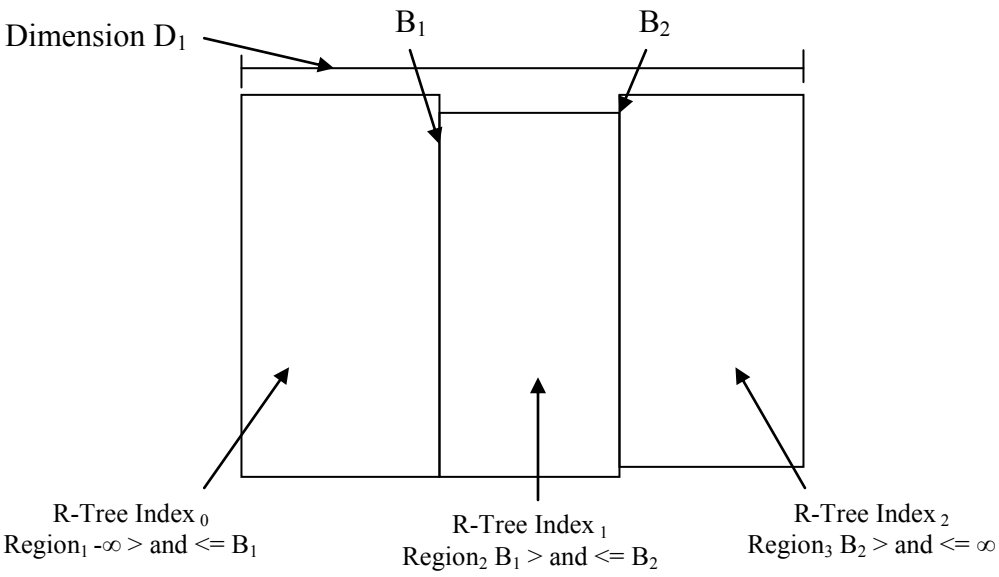



Figure 62. Dimension $D_1$ split into three regions defined by $B_1$ and $B_2$

With $B_1$ and $B_2$ being fixed for the given dimension for the lifetime of the R-Forest, also note the outer bounds of $Region_1$ and $Region_3$ extend to +/- ∞ even though the domain for the dimension may not. For my implantation I fix the inner most boundaries but allow the outer most to grow as R-Tree must handle domains that may grow or shrink as points are inserted or deleted. This leads to $O(B^{log(d)}) = O(d)$ number of R-Trees.

### 6.3.3 Maintaining Disjoint Set

These split dimensions and bounds are stored within the R-Forest as hard bounds for each R-Tree. This in turn becomes part of the MBR for each R-Tree index. At this point I have a set of R-Trees that are guaranteed not to overlap in maximum of $D = O(log\ d)$ dimensions and $B$ boundaries which means that each R-Tree has a root MBR that is disjointed on several dimensions. Because the boundaries are fixed, the split dimensions

boundaries will remain the same the disjoint set is maintained. Finally the union of all R-Tree indices will be the domain the entire data set.

### 6.3.4 Space Requirements

This approach using R-Forest does incur an increase in storage as compared to a single R-Tree. In the R-Tree implementation used each index has a header node occupying one page (typically 4KB). This header node contains basic index information, such as a root page pointer, deleted page pointer and free space pointer. By using multiple R-Trees I now have multiple header nodes, as each R-Tree is an independent index, also the split dimensions and boundary values are now stored in the unused space in the header nodes. Unlike other ANN methods (example LSB-Forest) points are not duplicated. R-Forest with 4 split dimensions and 3 boundaries will require $3^4 = 81$ R-Tree indices meaning I have an additional over head of 80 4KB pages for the header nodes versus a single R-Tree. Whereas, the total number of internal nodes and leaf nodes in the R-Forest will remain approximately the same as compared to a single R-Tree over the same data (that is without the inclusion of MBR-Center point, which will be discussed shortly). R-Forest increased storage cost will be covered in section 6.6.2 with comparisons given in chapter 7.

### 6.3.5 Inserting Points into R-Forest

When inserting a new point, I must now choose which R-Tree index to insert into. This is done by choosing the index by which the new point would be within the boundary limits of the split dimensions. Meaning the new point would belong to the sub-set of the

indices domain, because the boundaries within a split dimensions are fixed and never overlap, any given point will be a member of one and only one R-Tree index in the R-Forest. After the index is chosen, insertion into the R-Tree processed via the normal insertion algorithm used by R-Tree [44].

### 6.3.6 Index Maintenance

Like all other index based search methods maintenance must be addressed, in the case of R-Forest I have a set of R-Trees with no duplicate points. Insertion of new points proceeds as previously described. Whereas, updating an existing point would require locating the point in the appropriate index, traversing the index to locate the point and either deleting it and re-inserting the updated point or simply updating it, then updating parent MBRs as they may have changed. As for deletion of a point it would again require locating the point in the appropriate index, traversing the index to locate the point and deleting the point, then updating parent MBRs. In all cases, the maintenance cost has only increased by the cost of locating the appropriate R-Tree in R-Forest which is basically the cost of determining which index would contain the point, once the index is located the maintenance cost is no different than that of a single R-Tree.

## 6.4 Query Process with R-Forest

With R-Forest built and properly being maintained, attention now turns to how I can take advantage of the disjoint sub-sets when performing ANN queries. With the set of disjoint R-Trees, I now know that any query point $Q$ is within the subset of exactly one R-Tree based on the split dimensions and boundaries and using the same split dimensions

and boundaries, is not a sub-set of any other R-Trees. This means I can now take advantage of what was originally a pruning heuristic illustrated in figure 37 (chapter 5 section 5.1), only that it can be applied to an entire R-Tree, giving me the following advantages:

- Initial R-Tree selection for ANN query, which most likely will contain a reasonably good result
- Better ability to order of remaining R-Trees for searching
- Ability to prune off an entire R-Tree

These will be discussed in more detail in the next few sections, and summarized in the last section of this chapter with pseudo code for performing ANN queries on R-Forest and for ANN query in R-Tree.

### 6.4.1 Initial R-Forest Index Selection

The first step to perform an ANN query using R-Forest involves selection of the first R-Tree index to search. Because of the insertion method used it seems reasonable that searching should follow the same logic. In this case I want to select the index by which the query point Q is in the set of split dimension's boundaries, as seen in figure 63.

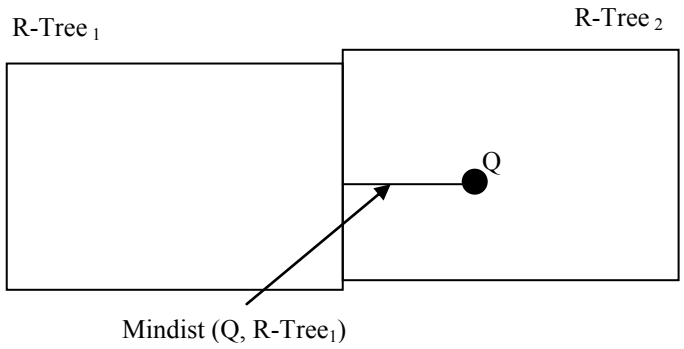



Figure 63. Illustration of use of Disjoint R-Trees

In Figure 63, point Q belongs to the set contained in R-Tree$_2$ using the split dimensions and boundaries, therefore this index will be searched first. Once the initial search index is selected I processed to search the R-Tree, using pruning and search heuristics applied to a single R-Tree.

### 6.4.2 Ordering Remaining R-Forest Indices

After the first index is searched I have an ANN which I refer to as Q's current best solution, the remaining R-Tree indices in the forest may contain better solutions. Therefore, I need to consider searching them as well. At this point the query point Q is not in the sub-sets of any of the remaining R-Tree indices, because of this I use it to my advantage, by ordering the remaining indices based on mindist. An important aspect of any branch-and-bound search is the order of how the nodes or in this case R-Trees are accessed. If the search can reach a good solution quickly then subsequent pruning will have very little negative effect on the quality of the ANN and may even eliminate entire R-Trees from the current search. Looking at figure 64, one can see that based on

Mindist, R-Tree$_4$ is closer to Q then R-Tree$_3$ and may be a better choice for improving Q's ANN.

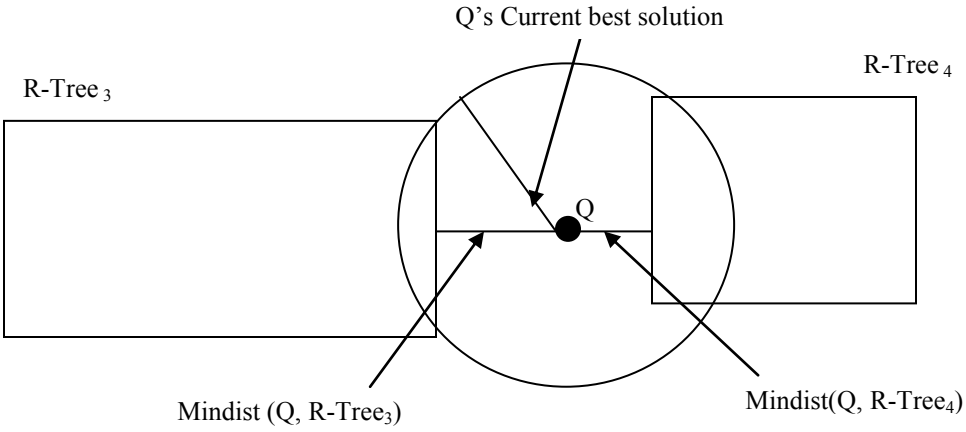



Figure 64. Illustrates R-Tree index ordering based on MinDist.

I should also point out that since Q is not in the subset of any R-Tree index other then the first index searched, it is also not in any of the MBR of any subsequent R-Tree indices to be searched, as seen in figure 65.

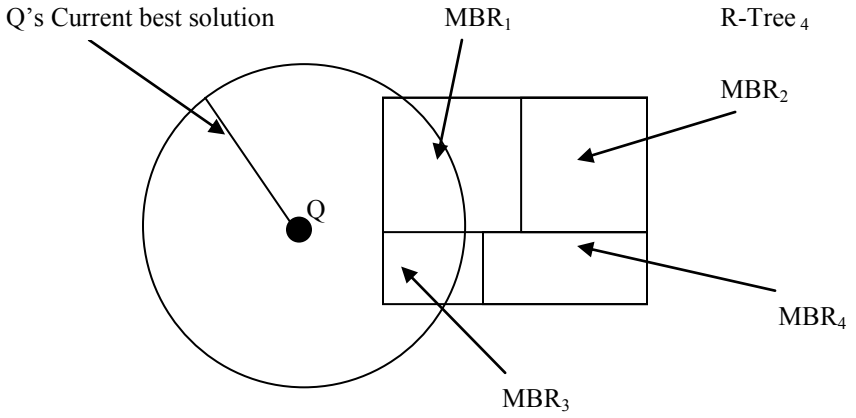



Figure 65. Illustrates R-Tree MBR ordering

Therefore, it will be possible to order MBR based on the same *mindist* to each MBR in an R-Tree and its nodes. In figure 65, R-Tree$_4$ has four MBR at its root, of these the initial search order would be MBR$_3$, MBR$_1$ and MBR$_4$ with MBR$_2$ being pruned because its *Mindist (Q, MBR$_2$) > Q' s* current best solution (ANN) distance.

### 6.4.3 Pruning of R-Trees

Besides being able to order index selection and node selection it is also possible to prune entire indices because of the disjoint sub-sets. As a search proceeds, Q's current best ANN should improve, which means that in some cases I should be able to eliminate an entire R-Tree indices from a given search based on mindist from Q to the indices MBR as seen in figure 66.

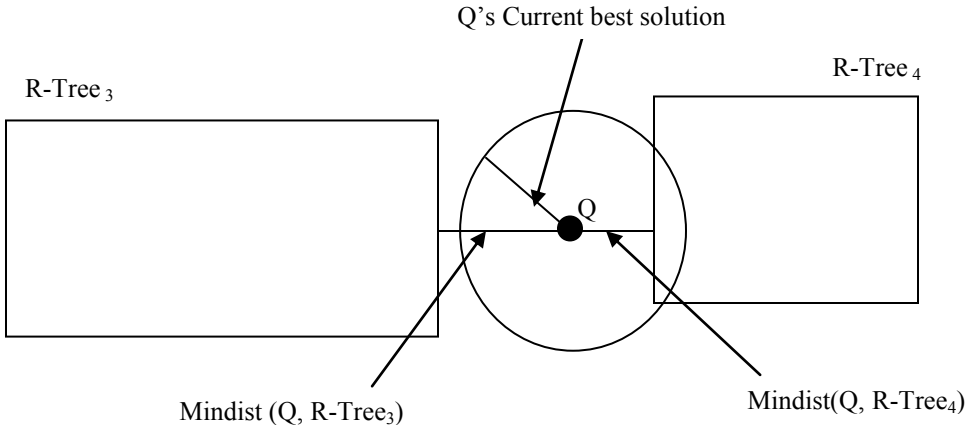



Figure 66. Illustrates R-Tree index pruning based on MinDist.

In figure 66 as an example, R-Tree$_4$ was selected and traversed, during the search Q's current best ANN improved when I completed the search of R-Tree$_4$, Q's current best

ANN is now less then Mindist of Q to R-Tree$_3$, therefore R-Tree$_3$ need no longer be considered thereby pruning the entire index.

## 6.5 K-Factor and Feedback

The use of disjoint sub-sets has allowed, for the introduction of a new pruning parameter and feedback on the quality of the results returned. It attempts to minimize the error if a MBR is not searched. For any MBR best possible ANN is Mindist therefore this is the worst possible error if I choose not to search a MBR. This new pruning parameter known as K-Factor (*K* for short), works by pruning away nodes (MBR) and their sub trees, if the distance between the query point and the current solution is less than *K* times the minimum distance between the query point and the MBR of a node (MBR). This parameter will prune MBR with large Mindist (*mindist (Q, MBR)*) as it is thought these are much less likely to contain a better solution. Not only that, when pruning occurs with this parameter, it is possible to provide some feedback in terms of the quality of the results. Details on K-Factor and its feedback will be discussed in the next few sub-sections.

### 6.5.1 K-Factor for Pruning

This new pruning parameter K-Factor, takes advantage of the fact that for all but one R-Tree index in R-Forest, a point Q is not a sub-set of every MBR. K-Factor works by pruning away a node, (and its sub tree) if the minimum distance between the query point and the current solution is less than K times the minimum distance between the query point and the MBR of that node, (K is a user controlled parameter). This can be

illustrated in figure 67 where query point Q has an ANN represented by CB, and K-Factory pruning will be applied to one of the two MBR.

If K-Factor * Mindist(Q, $MBR_2$) > CB then prune $MBR_2$

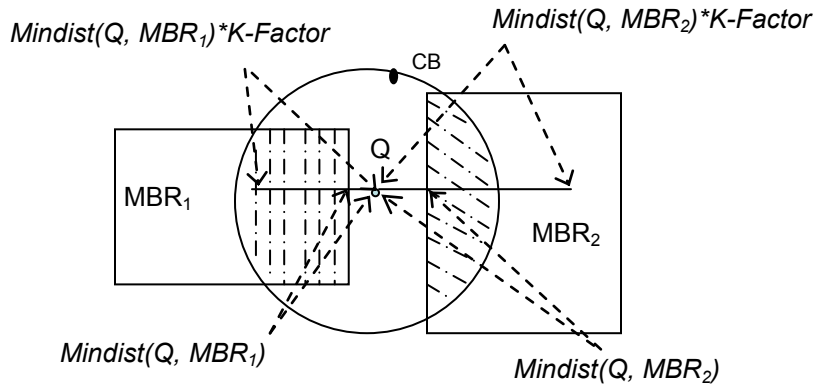



Figure 67. Illustration of *K-Factor* pruning where *K* is 4.0

In Figure 67 K-factor is used to examine two $MBR_1$ and $MBR_2$ and decide if either needs to be pruned or not leading to a search for an improved ANN. In this case *Mindist (Q, $MBR_2$) * K-Factor* is greater than current best ANN (*CB*) so prune $MBR_2$ where as *Mindist (Q, $MBR_1$) * K-Factor* is less then *CB* therefore search $MBR_1$. The larger the pruning factor (K), the more aggressive the heuristic will prune during the search, because *Mindist (Q, $MBR_1$) * K-Factor* is more likely to be larger than CB. Whereas the smaller the pruning factor the less likely *Mindist (Q, $MBR_1$) * K-Factor* will be larger than CB, therefore less pruning which will led to a more accurate solution but at higher access cost. This heuristic should also return an ANN no more than K-Factor times further then Q's exact NN.

### 6.5.2 Providing Feedback on Lower Bound

One advantage of using this pruning with the disjoint R-Forest is that it can provide a lower bound on the space not searched, in some cases. When a MBR such as $MBR_2$ (figure 67) is pruned then *Mindist (Q, $MBR_2$)* becomes lower bound as it could have a better solution than Q's current best solution which at best could be *Mindist (Q, $MBR_2$)*. This value is preserved during a search and updated whenever a pruned MBR has a smaller *Mindist (Q, MBR)*.

This lower bound comes from the use of R-Forest whereby any given point is within the sub-set of exactly one R-Tree in R-Forest and is not in the sub-set of the rest. This fact is supported by the implementation and the maintenance of hard boundaries on the split dimension throughout the life time of all the indices. Now that I can guarantee that a point is outside of all but one R-Tree index, I can define the lower bound on MBR not searched. Mindist, defined as the minimum distance from Q to a MBR, is now the lower bound on the ANN results, but only for MBR pruned (not searched). In figure 68, if a point in $MBR_2$ were at Mindist (on the surface of the MBR) then it would be the new ANN of a Q if $MBR_2$ were searched, were as if $MBR_2$ is not searched then MinDist is the best possible ANN that could exist in $MBR_2$ hence its lower bound. This can best be seen in figure 68 whereby, for a given point Q the formula *Mindist (Q, $MBR_2$)* K-Factor > CB* is true thereby pruning $MBR_2$. Because, the heuristic pruned $MBR_2$ its lower bound is *Mindist (Q, $MBR_2$)*. To provide feedback, if this value is less than any other lower bound from already pruned MBR than I would update the lower bound on the results already discovered to this new value.

If Mindist (Q, $MBR_2$) * K-Factor > CB then prune $MBR_2$

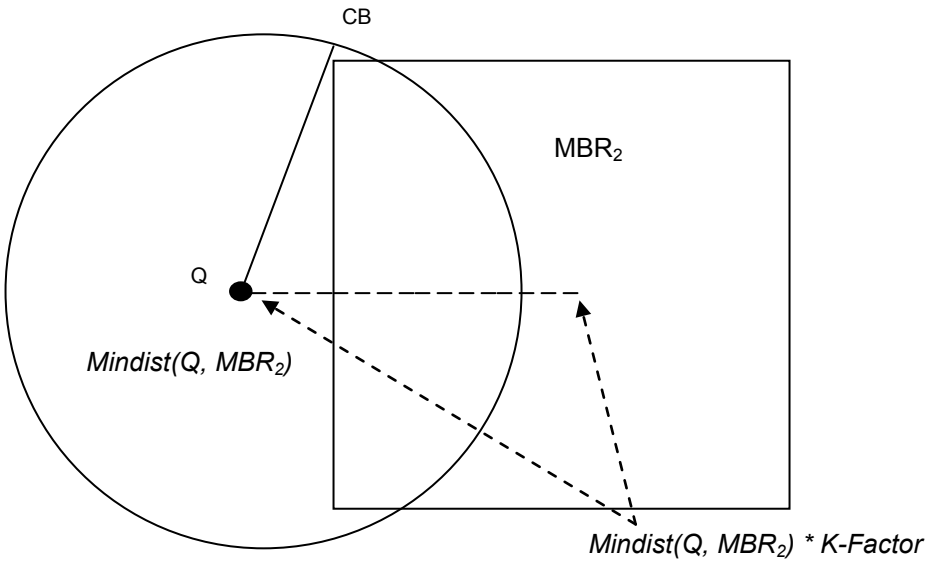



Figure 68. Illustration of Mindist as lower bound where K-Factor = 4.

This lower bound gives the user an idea of the best possible ANN distance that could exist in the MBR not searched because of K-Factor and the disjoint nature of R-Forest. If during a search K-Factor does not prune any MBR then the lower bound is undefined. This also assumes no restricted access or hard termination on the number of pages (to be discussed shortly), if I allow for restricted access then it is possible that a smaller lower bound on a future MBR may exist but never be accessed.

### 6.6 MBR-Center Point

As a further enhancement to R-Tree I decided to add an additional piece of information known as MBR-Center point. This point which will be maintained for each MBR in each internal node is loosely defined as follows: for a leaf node, it corresponds to

the data point in the node that is closest to the center of the MBR and for internal MBR at higher levels it is defined as the children MBR-Center point closest to the MBR center. An example, seen in figure 69 below, $MBR_1$ which has 6 points $P_1$ to $P_6$, would have $P_3$ as its MBR-Center point, this is due to its location, as it is closest to the center of $MBR_1$.

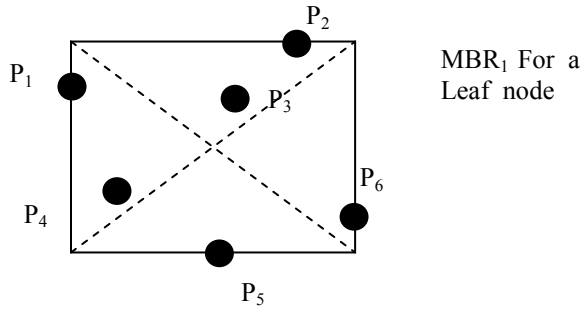



Figure 69. Illustrating MBR-Center point in a given MBR (point $P_3$ in this example)

For the internal nodes higher in the tree, I find the MBR-Center point of the sub MBR (children) closest to the center current MBR and store that as the MBR-Center point. This can be seen in figure 69, where $MBR_1$ to $MBR_4$ comprise the parent MBR and for these four sub MBR $C_4$ would become the Parent MBR's new MBR-Center point.

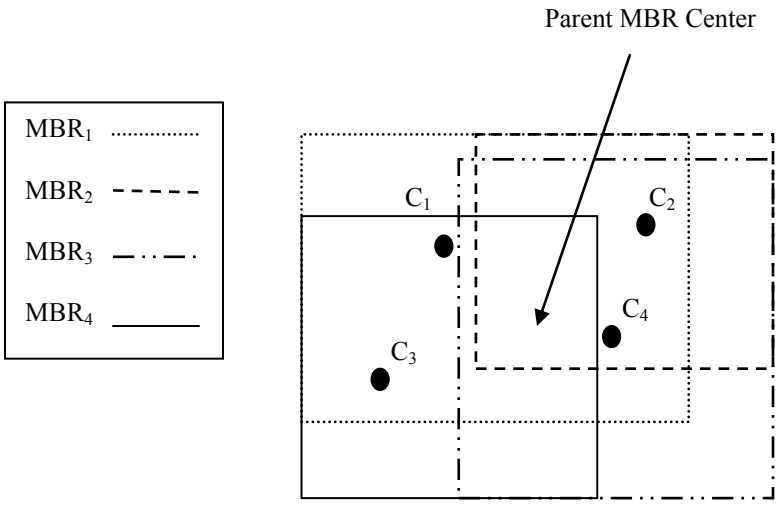



Figure 70. Showing Parent MBR's MBR-Center point selection.

With this new point I have the following possibilities:

- If the MBR's MBR-Center of the node is closer to the query point than the current solution, then the node should be accessed because there is at least a point in the sub-tree rooted there that has a better solution.
- It is possible to use the distance from a query point Q to a MBR's MBR-Center point as a means to order which MBR are searched.

### 6.6.1 MBR-Center Point Selection

To maintain the MBR-Center point I need only add it to the index structure of the internal nodes and then update it as points are inserted, updated or deleted. During a point Q insertion into an R-Tree leaf node, check the Qs distance to the center of the MBR and if closer than the existing MBR-Center point than Q becomes the new MBR-Center point. When this occurs I must also propagate the MBR-Center point change up the trees in the same fashion R-Tree updates parent MBR after an insertion. For the

MBR-Center point to be useful it must be maintained for each parent MBR as changes in dimension domain will also cause the MBR-Center to change and thereby the MBR-Center point closest to the center might change.

### 6.6.2 Increased Storage Cost

Adding a MBR-Center point to each MBR does incur an additional cost, in terms of storing the MBR-Center point per branch but with no additional cost in the leaf nodes. This means that the storage of each branch housing the MBR now has increased by 1/3, meaning that I now store the lower and higher corner of the MBR with the additional MBR-Center point. This has the effect of reducing the fan out of the R-Tree by 1/3, which will cause the R-Tree's height to increase. In chapter 3 section 3.4, there was given an explanation on how to compute the R-Tree fan out, this now has to be extended to cover the addition of the MBR-Center point (seen in the following figure 71 page size of 4KB with 32 dimensional points). Now the total fan out or number of branches is calculated from the page size and number of dimensions.

Fan out = (4KB – (4B + 4B)) / (4B + (4B * 32D) + (4B * 32D) + (4B * 32D))

= 4088B / 388B

= 10

Fragmented space = 4KB – (4B + 4B) – (10 * 388B)

= 208Bytes

Figure 71. Computing the internal node fan out based on a 4KB page size with additional MBR-Center point.

I should reiterate that the leaf node storage cost stays the same as MBR-Center points are stored only in the internal nodes. I should also point out that with a 4KB page size, R-Tree would reach its fan out lower limit sooner. Given the example in figure 70, using a 4KB page and 171 dimensions, R-Tree would have fan out of 1 meaning it is no longer a tree it is a linked list. To resolve this, the index page size would need to be increased so that more than one branch would exist per internal node.

### 6.6.3 Index Ordering

As discussed before, with R-Forest I must have a method to select the search order of the R-Tree indices. Typically the Mindist of the query point Q to the root MBR of each R-Tree index is used to order the remaining indices to search. Each R-Tree index has a root MBR (comprised of the union of the root nodes branch MBR) that covers the

sub-set of points. The root node's branch MBR each have a MBR-Center point as seen in figure 72, which can be examined at little IO cost as the root nodes are already in memory.

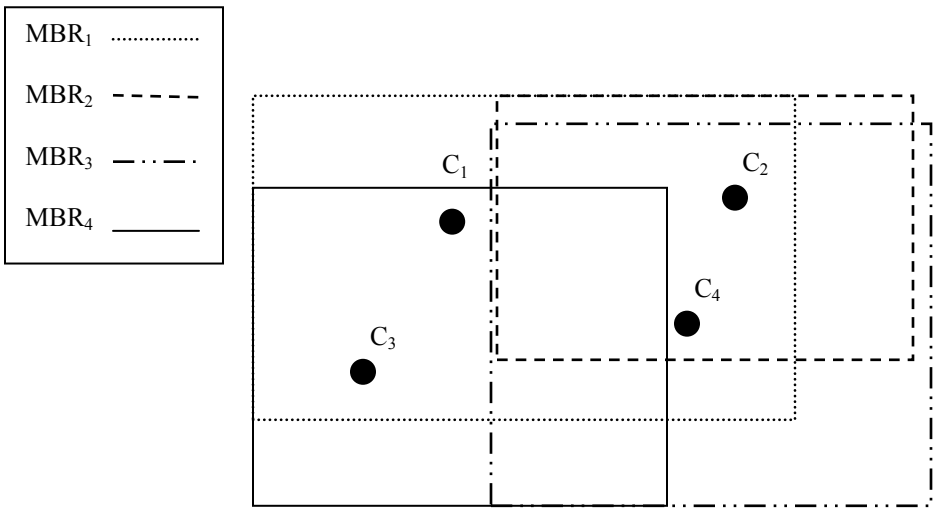



Figure 72. Example of Root MBR and 4 branch MBR-Center points

In figure 72 see an example of four MBR ($MBR_1$ to $MBR_4$) each with a MBR-Center point $C_1$ to $C_4$, by computing the distance from the query point Q to $C_1$ to $C_4$, I have alternate method of ordering which MBR or R-Tree to search next as I can now order either MBR or entire R-Trees based on a distance to a physical point.

### 6.6.4 Branch Selection

This new MBR-Center point, which is a physical point, gives me the ability to select some branches to search which will have a guaranteed better solution. If I find that a MBR-Center point of any MBR is closer than our current best ANN then I know that at least one point in that MBR is a better than the ANN discovered so far. This condition

will not be met all the time but when it does, I know how to guide the search through the tree to an improved ANN and better solution.

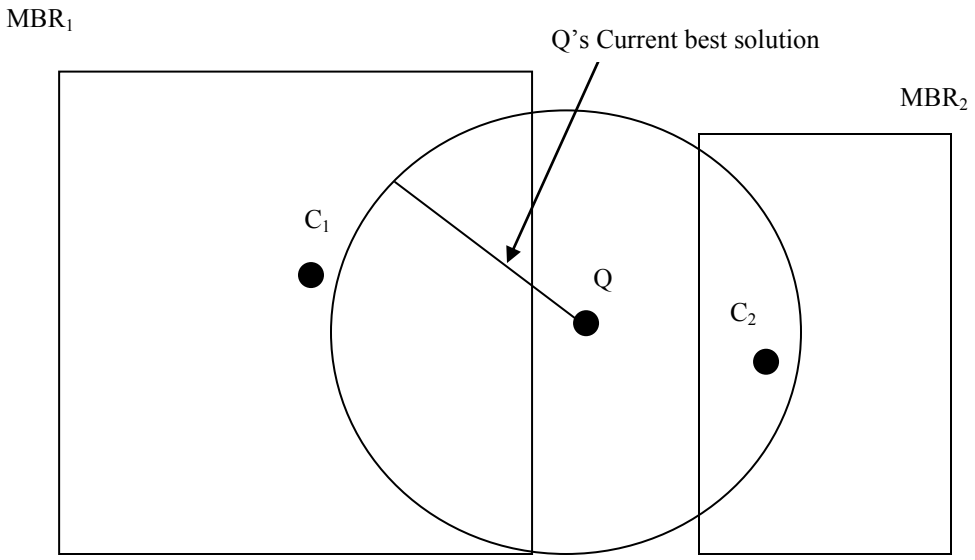



Figure 73. Showing example of one MBR with a MBR-Center Point that would be a better ANN.

An example of this can be seen in figure 73, where Q is closer to $MBR_1$ but the MBR-Center point $C_2$ in $MBR_2$ is within the radius defined by Q's current best ANN which means $C_2$ (which is a point in $MBR_2$) is closer then Q's current best ANN, therefore we should search $MBR_2$ before $MBR_1$. It should be noted that when this condition is met as in figure 73, a MBR will be searched even if another condition would prune the MBR, and that I am not using the MBR-Center point to pruning any other MBR.

### 6.6.5 Branch Ordering

During a query within R-Tree I need to choose the best branch (MBR) to traverse, this is known as branch ordering. The algorithm typically employed involves computing Mindist from Q to each MBR and ordering the search based on this distance. This algorithm has one flaw in that Mindist is zero when the query point Q is inside a given MBR. This occurs on the first R-Tree searched in R-Forest as the query point is in the sub-set of the first R-Tree index, based on the split dimensions and boundaries, and is usually inside most of the MBR of the first R-Tree Index searched. Because of this I chose a second method of ordering by using the distance to the MBR-Center point which is already being computed. Therefore, if *Mindist (Q, MBR) = 0* then use Q's distance to the MBR's MBR-Center point as a means to order the branches. For the remaining R-Trees in the forest Q will not be inside any MBR, so ordering by Mindist is used.

## 6.7 Search Termination

While working on this research and to compare to others it was decided to add a search termination parameter which the user can specify. This parameter simply terminates the search when the number of pages read reaches the value set by this parameter. It works by counting the total number of pages requested for all trees traversed and when the total exceeds the parameter by 1, searching is halted and results (ANN) found so far are returned. During previous research, it was observed that during an exact nearest neighbor search, most of the nodes visited do not improve the current best solution found before visiting them. Past experiments with a standard branch-and-bound approach shows that during an (exact) NN search, on average 0.14% (with a max

0.33%) of the leaf nodes visited leads to an improvement over the current nearest neighbor found. That is, instead of just pruning away nodes that cannot contain the nearest neighbor, I prune away nodes that are unlikely to contain the nearest neighbor. Thus I believe a more aggressive pruning scheme will lead to high efficiency of the search while maintaining a good solution. Past and present research shows that a relatively good solution is arrived at very early on during a search and that traversing additional indices may lead to a better solution but at higher cost in terms of IO. Therefore, terminating the search with a hard cut off on the number of pages saves IO with only a small quality performance hit. Since this method in my dissertation is only looking for an ANN, and I am willing to trade accuracy for efficiency, adding a hard termination condition on the IO will only affect the accuracy.

### 6.8 Summary of R-Forest Construction and ANN Queries

All the changes to R-Tree covered throughout this chapter are summarized below starting with the construction of R-Forest from a given data set, and ending with ANN queries of R-Forest.

I. Initialization of R-Forest and construction.

1. Initial R-Forest by random sampling of initial data set, find a sub-set of dimension to split then split each dimension into 2 to 3 regions (figure 74 below).

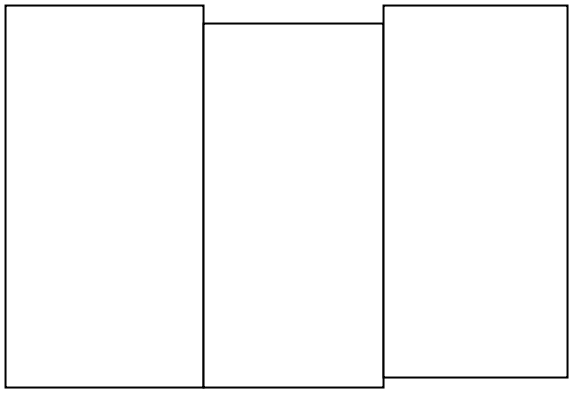


Figure 74. Defined Regions after single dimension is split and boundaries are defined

2. Define N number of R-Trees with hard boundaries on the split dimension and boundaries based on the number of split dimensions and boundaries.
3. Insert points into the appropriate R-Tree index within the R-Forest by locating which index the point is a set of, use standard insertion originally defined for R-Tree figure 75.

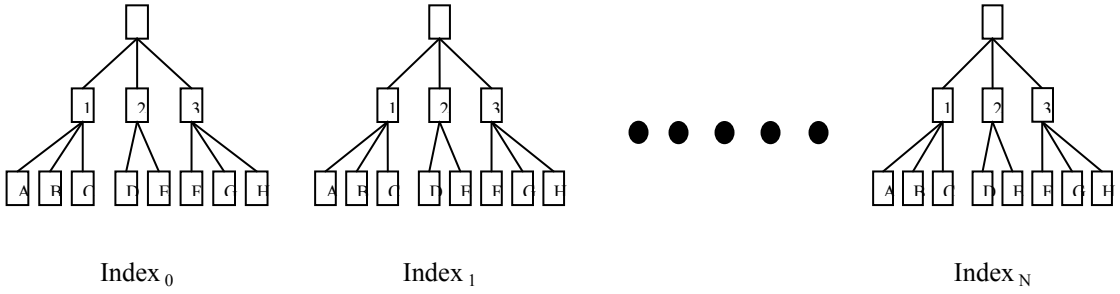



Figure 75. R-Tree indices that comprise R-Forest

4. During the insertion of points into leaf nodes update the MBR-Center point and prorogate the changes up the tree as needed figure 75.

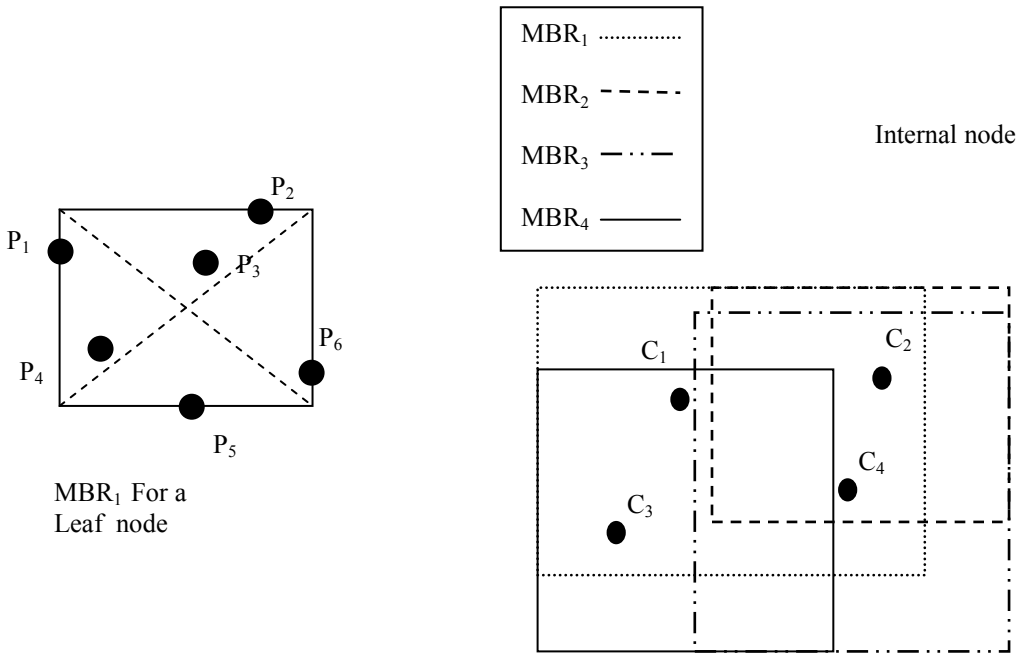



Figure 76. MBR-Center point discovery in a leaf node and internal node

5. Maintain the split dimensions and boundaries when updating and deleting points.

II. Querying R-Forest – Given a query point Q using the following steps (Pseudo code in Figure 77 and 78)

```
R-Forest_ANN_Query (Point Q)
        Select Initial R-Tree where Q is a SubSet of Boundaries
        Initial R-Tree ANN_Query (R-Tree.Root, Q)
        If Page Access Count > Termination count then
                Goto Terminate_Search

        For each R-Tree in R-Forest
                Compute Mindist (Q, MBR)
        Sort R-Trees Based on Mindist to All MBR

        For Each R-Tree in Sorted List
                R-Tree ANN_Query (Q)
                If Page Access Count > Termination count then
                        Goto Terminate_Search

        Terminate_Search
        return Q's CB ANN found and Lower Bound
```

Figure 77. Pseudo code for R-Forest ANN query with R-Tree Ordering and termination condition

```
ANN_Query (Parent.Node, Point Q)

        If Node is Internal
              { /* Node is Internal Node */

                For each MBR in Parent.Node
                        Compute Mindist (Q, MBR)
                        If Mindist (Q, MBR) = 0
                                Compute Dist(Q, MBR-Center Point)

                        Sort MBR List on Mindist or MBR-Center Point Dist

                For each Sorted MBR in Partent.Node


                        If Q's CB ANN distance  < Mindist (Q, MBR) then
                                Prune MBR = True
                        If K-Factor * Mindist (Q, MBR) > CB distance  then
                                Prune MBR = True
                                lower bound = Mindist (Q, MBR)
                        If Q's CB distance > MBR-Center Point then
                                Prune MBR = False
                        If Page Access Count > Termination count then
                                Goto Terminate_Search

                        If Prune MBR = False then
                                ANN_Query (MBR.Node, Q)
              }
        Else
              {  /* Node is Leaf Node */

                For each Point in Leaf Node
                        If Q's CB ANN > Dist (Q, Point) then
                                Q's CB ANN = Dist (Q, Point)
              }

        Terminate_Search
        Return Q's CB ANN found and Lower Bound
```

Figure 78. Pseudo code for R-Tree ANN query (with K-Factor, MBR-Center Point and termination condition)

1. Determine which R-Tree index in the R-Forest, which Q is a set of, and select it as the initial tree to search.
2. Starting at the Root.
3. Within an internal node order the MBR based on Mindist and distance MBR-Center distance. Choose the MBR Q is closest too, based on Mindist or MBR-Center distance, access that node.
4. Repeat Step 3 until I reach a leaf node then search the leaf establishing an ANN and updating this as I check all points in the Leaf.
5. Once visit the first leaf the following pruning heuristics can be applied to leaf nodes and internal nodes figure 79

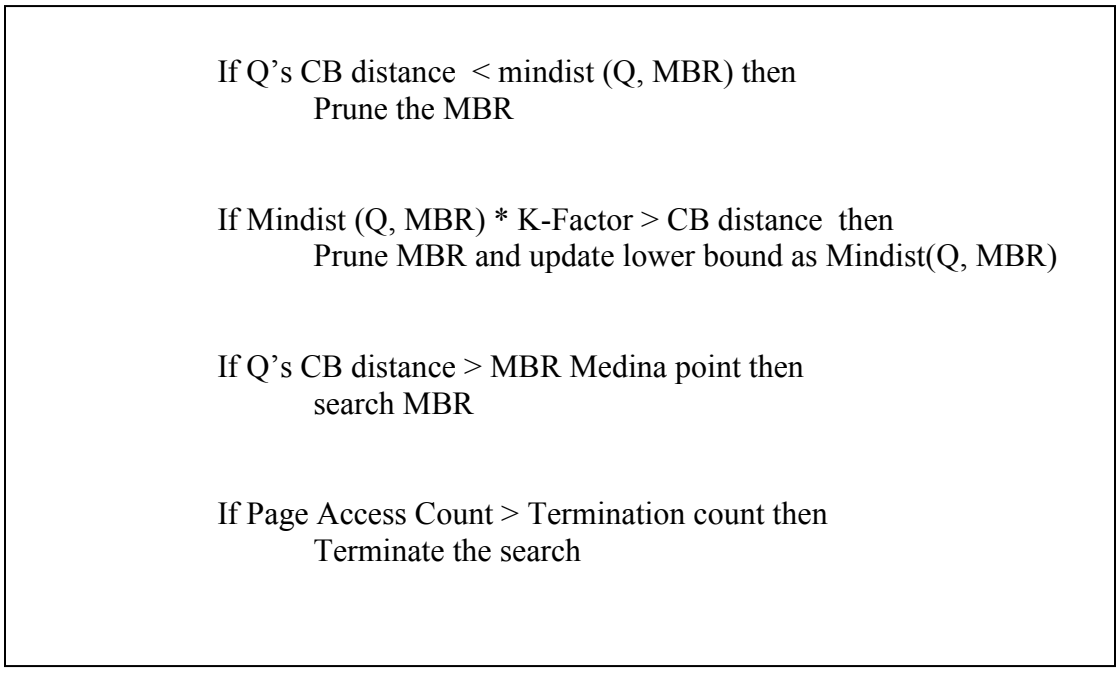



Figure 79. Search heuristics used in R-Tree, including new conditions available with R-Forest

6. Repeat step five for all MBR in the given nodes.
7. Move up the tree and again repeat step 5.
8. When finished searching the first R-Tree order the remaining based on Mindist and MBR-Center point, select the next closest and loop to step 3.
9. If search termination is not used then terminate when all R-Trees have been accessed.
10. The results returned at the end of a query would include the closest ANN point found and a lower bound on all the MBR pruned by K-Factor, whether the search was terminated with restricted access or not.

In summary this section has covered the details on how to construct R-Forest and the modifications to searching R-Tree that take advantage of the forest and it's disjoint nature. I now have a new method for ordering searching the R-Trees, pruning heuristics and a new search termination function. The results returned at the end of a query would include the closest ANN point found and a lower bound on all the MBR pruned by K-Factor. It should be pointed out that a search terminated with restricted access or without, but with the use of K-Factor now has an additional termination function which eliminates MBR or entire R-Trees because they meet the condition *Mindist (Q, MBR) * K-Factor > C.*

## 7. Results

In this section I present my testing data, experiment methods and results for both R-Forest and LSB-Forest. These results include a general set of tests using three data distributions with 99,000 to 2,500,000 points and seven real world sets, these were used to show how well my approach works with different types of data. Next I will show how my approach scales as the number of dimensions increases. Finally, I will show how my method scales as the number of points increases from 500,000 points to 2,500,000 points. In all three sets of experiments I will compare my results with LSB-Forest [111] as this has become one of the predominant ANN solutions.

### 7.1 Data Sets

For my experiments I tested my approach with different data distributions and include a real world data set. To set up these experiments, I used a program that generates random data sets of any size (number of points), with different data distributes with varying number of dimensions. Most of my work has focused on 32 dimensional data and above, as past research has shown this is where most high-dimension NN an ANN methods including single R-Tree, fails to provide anything better then a sequential scan. Using the data generation program, I generated multiple data sets with the following data distributions, uniform, clustered and Gaussian. Details on these data sets will be given in the following sections.

### 7.1.1 Uniform Data Set

For uniform data, where each dimension is a random number with uniform probability distribution over domain of [0.0, 1.0], I generated 100,000 point set having 32, 48, 64, 80 and 96 dimensions. These sets were used for both general testing and dimensional scaling testing. I also generated five additional sets of 32 dimensional data having point counts of 501,000, 1,001,000, 1,501,000, 2,001,000 and 2,501,000 points. The additional 1,000 points in each set were used as test data, to ensure that the test data follows the actual distribution of the data.

### 7.1.2 Clustered Data Set

For clustered data, I generated 100,000 point set having 32, 48, 64, 80 and 96 dimensions and five 32 dimensional sets points ranging from 501,000 to 2,501,000 sets as discussed under the section for uniform data, over the domain of [0.0, 1.0] for each dimension. It should be noted, this clustered data has 20 random clusters, build by randomly selecting 20 points as the cluster's centroid and generating the remaining points around the centroids with a radius of 0.1, and attempting to make sure the clusters do not overlap in the dimensions but not guaranteed with each cluster have approximately the same number of points.

### 7.1.3 Gaussian Data Set

For Gaussian data, I generated 100,000 point set having 32, 48, 64, 80 and 96 dimensions and five 32 dimensional sets points ranging from 501,000 to 2,501,000 sets as discussed under the section for uniform data, but over the domain of [-4.0, 4.0] where

each dimension is a random number with Gaussian probability distribution, with mean 0.0 and standard deviation of 1.0.

### 7.1.4 Real Data Sets

For Real data sets I was able to locate seven different sets on the internet with different number of dimensions and number of points. The first was based on Corel images histograms from http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.html made up of 68,040 points (it should be noted I used the first 66,000 points in my test to compare to my previous research) [97]. For this data set each dimension has a domain [0.0 to 1.0] of unknown distribution and will be referred to as Corel. At http://lms.comp.nus.edu.sg/research/NUS-WIDE.htm I located sets which were compressed of 269,648 images analyzed with five different methods giving me five real data sets with different number of dimensions to experiment with (minor details to be provided in later sections including data set names) for the time being this entire set will be known as NUSWIDE [76]. Finally I used a set from http://osmot.cs.cornell.edu/kddcup/datasets.html which was used Knowledge Discovery and Data (KDD) Cup 2004 conference competitions, which had 145,751 points comprised of 74 dimension [55]. The sets from [76] and [55] had attributes in different ranges, because I am comparing my results to LSB-Forest which only works with integers, I normalized all the real sets to domain of [0.0 to 1.0] across all dimensions.

## 7.2 Experimental Setup

In all of my data sets no matter the size or number of dimensions, I chose to split on 4 dimensions with 3 regions. Notice that I chose a number of split dimensions of 4 instead of 5 = $log_2\ 32$ because this provides me a reasonable number of trees $3^4 = 81$ with each tree having a height of 1-5 levels depending on the data set size. With larger data sets, the choice of 5 may yet become reasonable, for my initial testing of 100,000 points splitting on 5 dimensions with 3 regions would have led to $3^5 = 241$ R-Trees indices, which would be flattened out, leaving them comprised mostly of leaf nodes. Listed below in table 6 are the parameters used for testing both R-Forest and LSB-Forest. Please note LSB-Forest parameters came directly from the authors paper [111]. For some of the data sets I tested with the page size had to be increased from 4KB to 12KB, which is noted in the results when need be.

Table 6. Parameters for R-Forest and LSB-Forest

| **Parameter** | **R-Forest** | **LSB-Forest** |
|---|---|---|
| **Page size (KB)** | 4KB or 12KB | 4KB or 12KB |
| **No. of DIM** | 32 – 225 | 32 – 225 |
| **Split Dimension** | 4 | N/A |
| **Regions per Dim** | 3 | N/A |
| **K-Factor (query parameter)** | range 1 to 4 | N/A |
| **R (restricted access, discussed in 6.7 Search Termination)** | 90 pages | 90 – 110 pages |
| **Number of Indices** | 81 | 20 |
| **No. of Points stored** | N | N * Number of Indexes |
| **Largest Integer T (specific to LSB-Forest)** | N/A | 10,000 |

### 7.2.1 Index Construction

In the previous section I listed multiple data sets with different data distributions, these include three 99,000 points sets, seven real sets, twelve sets with dimension from 48 to 96, and fifteen sets points from 500,000 to 2,500,000. This gives me 37 data sets to work with. For each, I used my R-Forest code to construct forests of R-Trees using the parameters given in table 6 (section 7.2). As a comparison I used the code from Yufei web site http://www.cse.cuhk.edu.hk/~taoyf/paper/tods10-lsb.html [113] to build LSB-Forest forest with the same 37 data sets using the parameters given in their paper, also summarized in table 6 (section 7.2).

### 7.2.2 Test Points Used

For testing I selected the last 1,000 points from each set (including the Real data sets) and used them as my test points. By using the last 1,000 points I know the test points follows the same distribution as the data sets, which is important for Gaussian and clustered data. Since Gaussian and clustered data have specific distribution it seems best that the test points have basically the same distribution otherwise the results might be skewed.

## 7.3 Experimental Results with Comparison to LSB-Forest

In my first set of experiments, I tested different data distributions and the affect of varying K-Factor from 1 to 4 to see how it affects the search. I am comparing my results with LSB-Forest. For LSB-Forest the page size was set to 4K (unless the number of dimensions required a larger page size), and the number of trees in LSB-Forest was set to

20, research clearly shows that at around 20 trees the results level off and do not improve much as the number of trees increases [111]. Also their research paper points out that they have a hard termination point at 90 pages (actually according to their code it is the number of trees multiplied by the height of a tree). So for my method I set the access restriction to 90 pages as a comparison to LSB-Forest, therefore both methods will be using the same amount of IO, and both implementation count cost in terms of pages accessed, by keeping the page size the same I am comparing actual cost of both methods.

For all of the experiments, I record the number of leaf node accessed, as well as the quality of the result by means of a sequential scan against the full set of points, and will measure the quality of the results using the following:

- Distance ratio: Ratio of distance of retrieved point (ANN) from $q$ to actual NN distance or ratio of how close the solution found is to the actual solution.
- Percentage of exact matches: How often does the query search return the actual nearest neighbor, even though both methods are seeking only an ANN.
- Percentage of better ANN: What percentage of the test results returned by R-Forest had an ANN with a distance ratio less than the same results from LSB-Forest.
- Lower bounds: the value of the lower bound distance returned, and the ratio of it to the actual distance to the nearest neighbor.

For each set of experiments, I present my results in a table that contains all the measures, as well as a histogram showing the distribution of distance ratios. The tables will show the best, worst, and average case distance ratio. The histogram will provide a more detail picture of the performance. To make the histogram more presentable, I only

included the results where K equal 4. Finally I wanted to add some details on the lower bound feedback, this was done by presenting the maximum ratio of K times minimum Mindist to Q's actual NN for all 1,000 test. This shows that minimum Mindist found so far could be used as feedback as to the bound on the ANN found when the query terminates.

### 7.3.1 Uniform Data Set

Starting with 32 dimensional uniform data distribution in table 7, I have summarized the parameters used by both methods, as well as the total size in MB of indices built.

Table 7. Summary of Parameters (for 32 dimensional uniform data)

| | **Page size (KB)** | **# of pages accessed** | **Number of indices** | **Number of points** | **Total size of index (in MB)** |
|---|---|---|---|---|---|
| **R-Forest** | 4K | 90 | 81 | 99,000 | 18MB |
| **LSB-Forest** | 4K | 90 | 20 | 99,000 | 592MB |

In table 8 I have listed the results for R-Forest and LSB-Forest with increasing values for K-Factor (K) followed by figure 80, which shows a histogram for the test points distributions by increasing distance ratios.

Table 8. Comparison of R-Forest and LSB-Forest for uniform data

| Method | k | Distance Ratio | | | | % of exact match | % of cases where R-Forest outperforms LSB-Forest | Lower bound ratio (exclude outliers -- see below) |
|---|---|---|---|---|---|---|---|---|
| | | Min | Mean | Standard Deviation | Maximum | | | |
| LSB-Forest | N/A | 1 | 1.263103 | 0.12813 | 1.860324 | 1.3 | N/A | 4 (guar.) |
| R-Forest | 1 | 1 | 1.098378 | 0.07802 | 1.484131 | 12.5 | 91.8 | ∞ |
| R-Forest | 2 | 1 | 1.098011 | 0.07788 | 1.484131 | 12.6 | 91.8 | 1.487 - ∞ |
| R-Forest | 3 | 1 | 1.091692 | 0.07673 | 1.484131 | 14.4 | 92.6 | 2.600 – 3.000 |
| R-Forest | 4 | 1 | 1.087586 | 0.07604 | 1.430819 | 16.8 | 93.4 | 3.591-3.969 |

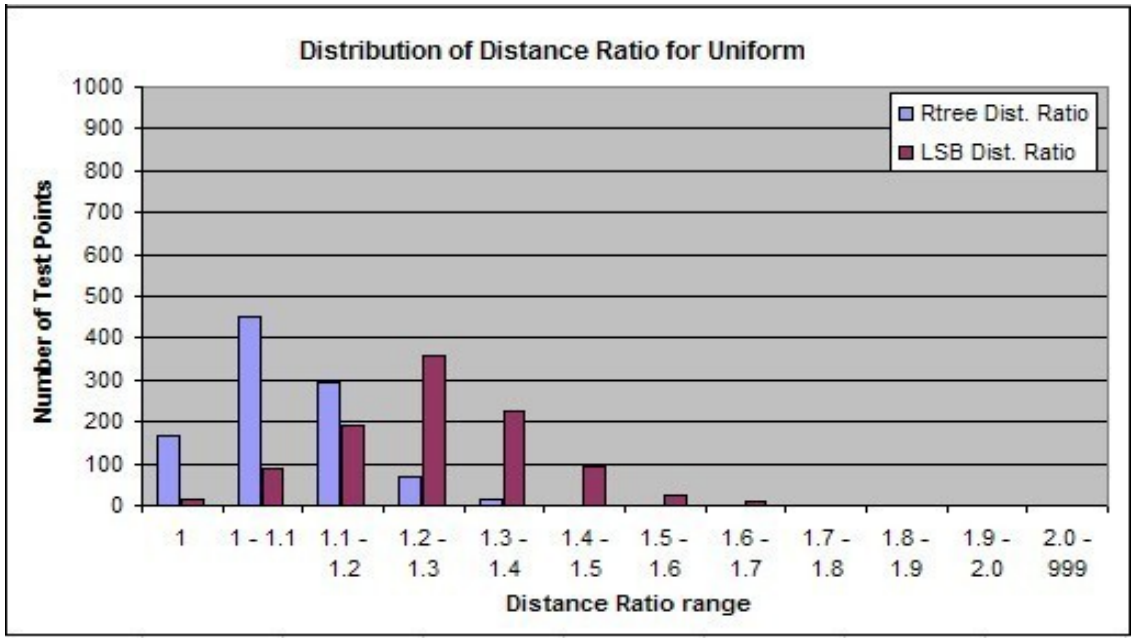



Figure 80. Distance Ratio distribution for R-Forest and LSB-Forest using uniform data set, with K = 4

From Table 8, one can see that the R-Forest performs favorably over the LSB-Forest, given the same amount of access as listed in Table 8. The R-Forest locates significantly more exact nearest neighbor than the LSB-Forest (12.5-16.8% vs. 1.3%). Moreover, in more than 90% of the cases, R-Forest returns a better solution than the LSB-Forest in terms of distance ratio for the same test point. There is further support in figure 80 which shows distribution of the solutions found by both methods. For R-Forest, most of the solutions (nearly 85%) have a distance ratio less than 1.2 as compared to LSB-Forest.

The authors of LSB-Forest offers a "guarantee" on the bounds in that the point returned will be C-Approximate of the NN. For R-Forest (uniform data) I was able obtain better results but without "guarantee" bounds. However, in the last column of table 8 (cases were K equal 3 or 4) shows the returned value of lower bound for the R-Forest will be within 3 to 4 of the actual solution, achieving the same lower bound as the LSB-Forest.

Looking at the results for varying the prune factor (K) shows some counter-intuitive results. First, the quality of the results actually improves with larger value of K-Factor – i.e. the more aggressive the pruning, the more likely the R-Forest returns a better solution. This actually echoes back to my discussion earlier that very few nodes examined, actually improve the solution during the branch-and-bound search. If K is too small, then very few nodes will be pruned in the beginning and I reach the limit set by restricted access, which terminates the search. If the NN solution to $q$, is somewhere later in the search path (which, interestingly, is quite often the case during a depth-first traversal), and I terminate too early, then the search will never reach NN. However, with

more aggressive pruning I am more likely to prune away nodes that do not improve on the solution, thus enable further searching and increasing the chances of reaching a better or actual solution.

Secondly, the lower bound returned shows two interesting results that warrants explanation. For K equal 3 and 4, I notice that the actual lower bound ratio is slightly less than *K* in my experiments, even though there is no guarantee. However, when K is 2 or smaller, there are cases where R-Forest fails to return a lower bound. This is due to the fact that when K is small, very little pruning is done, and thus the search may never finish searching the first R-Tree – which contains overlapping regions within its sub-trees (access restriction terminates the search). Thus it is possible there are many nodes in the first R-Tree where their MBR contains q that the search does not reach (few nodes are pruned). This leads to a lower bound of 0 (and a lower bound ratio of infinity). However, for K equal 3 or 4, the search is able to prune enough nodes such that the search of the first tree is complete. Thus all nodes which MBR contains q have been searched, so a lower bound can be established, and in all of those cases, the lower bound ratio is actually less than K [75].

### 7.3.2 Clustered Data Set

For 32 dimensional clustered data results, table 9 lists the parameters used by both methods, as well as the total size in MB of indices built.

Table 9. Summary of Parameters (for 32 dimensional clustered data)

| | Page size (KB) | # of pages accessed | Number of indices | Number of points | Total size of index (in MB) |
|---|---|---|---|---|---|
| **R-Forest** | 4K | 90 | 81 | 99,000 | 18MB |
| **LSB-Forest** | 4K | 90 | 20 | 99,000 | 591MB |

In table 10, I have listed the results for R-Forest and LSB-Forest with increasing values for K-Factor followed by figure 81, which shows a histogram for the test points distributions by increasing distance ratios.

Table 10. Comparison of R-Forest and LSB-Forest for Clustered data.

| Method | k | Distance Ratio | | | | % of exact match | % of cases where R-Forest out performs LSB-Forest | Lower bound ratio |
|---|---|---|---|---|---|---|---|---|
| | | Min | Mean | Standard Deviation | Maximum | | | |
| LSB-Forest | N/A | 1 | 1.30401 | 0.13723 | 2.01258 | 0.8 | N/A | 4 (guar.) |
| R-Forest | 1 | 1 | 1.611245 | 2.23787 | 14.14206 | 48.8 | 92.1 | ∞ |
| R-Forest | 2 | 1 | 1.029058 | 0.04833 | 1.368558 | 51.8 | 98.6 | 1.105 - ∞ |
| R-Forest | 3 | 1 | 1.026273 | 0.04593 | 1.319137 | 55.2 | 98.7 | 1.961-3.000 |
| R-Forest | 4 | 1 | 1.022165 | 0.04246 | 1.319137 | 60.9 | 99.0 | 3.293 – 4.000 |

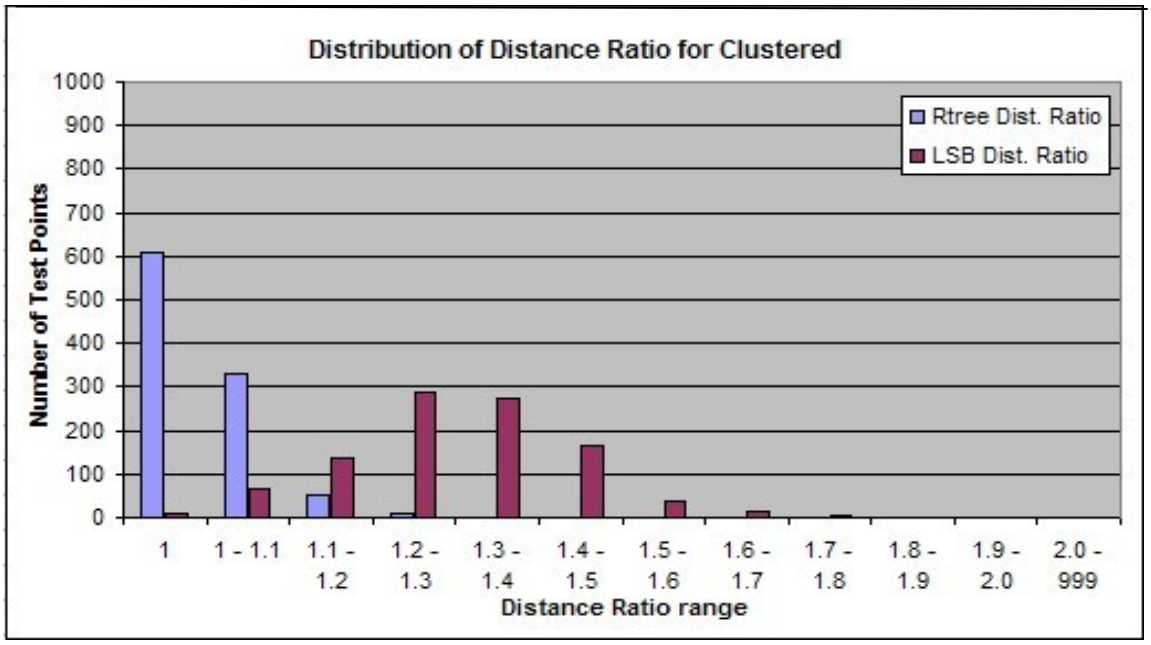



Figure 81. Distance Ratio distribution for R-Forest and LSB-Forest using clustered data set, $K = 4$

Looking at Table 10 and Figure 81 one can see that when K-Factor pruning is used, at K equal 2 and above, I had over 500 correct NN returned out of a 1,000 tests, and as K increased I continued to see more correct NN returned, and at *K* equal 4, I reached over 600 tests (60%) with correct NN. Also, out of all 1,000 clustered point test 98% - 99% of the results R-Forest returned had a better distance ratio then the same 1,000 test results from LSB-Forest, see Figure 81. Most likely this was due to the nature of the data being clustered, it is thought that the R-Forest found the clusters and organized the data accordingly; hence R-Forest was able to take advantage of the clustered data without any modification of the parameters, or prior knowledge of the data distribution. As for the lower bound on the results when K equal 3 and 4, all test cases were at or less then the desired bound. For cases where K equal 2, the upper range of the lower bound was

infinity this is due to the situation where K-Factor is not as aggressive at pruning and in some test case not being used to prune any MBR [75].

### 7.3.3 Gaussian Data Set

Next I will present my experiment results for 32 dimensional Gaussian data results. In table 11, the parameters used by both methods, as well as the total size in MB of indices built are listed.

Table 11. Summary of Parameters (for 32 dimensional Gaussian data)

| | **Page size (KB)** | **# of pages accessed** | **Number of indices** | **Number of points** | **Total size of index (in MB)** |
|---|---|---|---|---|---|
| **R-Forest** | 4K | 90 | 81 | 99,000 | 18MB |
| **LSB-Forest** | 4K | 90 | 20 | 99,000 | 591MB |

In table 12 I have listed the results for R-Forest and LSB-Forest with increasing values for K-Factor (K) followed by figure 82, which shows a histogram for the test points distributions by increasing distance ratios.

Table 12, Comparison of R-Forest and LSB-Forest for Gaussian data.

| Method | k | Distance Ratio | | | | % of exact match | % of cases where R-Forest out performs LSB-Forest | Lower bound ratio |
|---|---|---|---|---|---|---|---|---|
| | | Min | Mean | Standard Deviation | Maximum | | | |
| LSB-Forest | N/A | 1 | 1.458489 | 0.19257 | 2.089663 | 0.2 | N/A | 4 (guar.) |
| R-Forest | 1 | 1 | 1.098768 | 0.07186 | 1.381195 | 9.0 | 97.7 | ∞ |
| R-Forest | 2 | 1 | 1.09835 | 0.07198 | 1.381195 | 9.4 | 97.7 | 1.407 - ∞ |
| R-Forest | 3 | 1 | 1.092139 | 0.07069 | 1.360345 | 11.6 | 98.1 | 2.724 – 3.000 |
| R-Forest | 4 | 1 | 1.08777 | 0.07218 | 1.365311 | 13.4 | 98.2 | 3.684 – 4.000 |

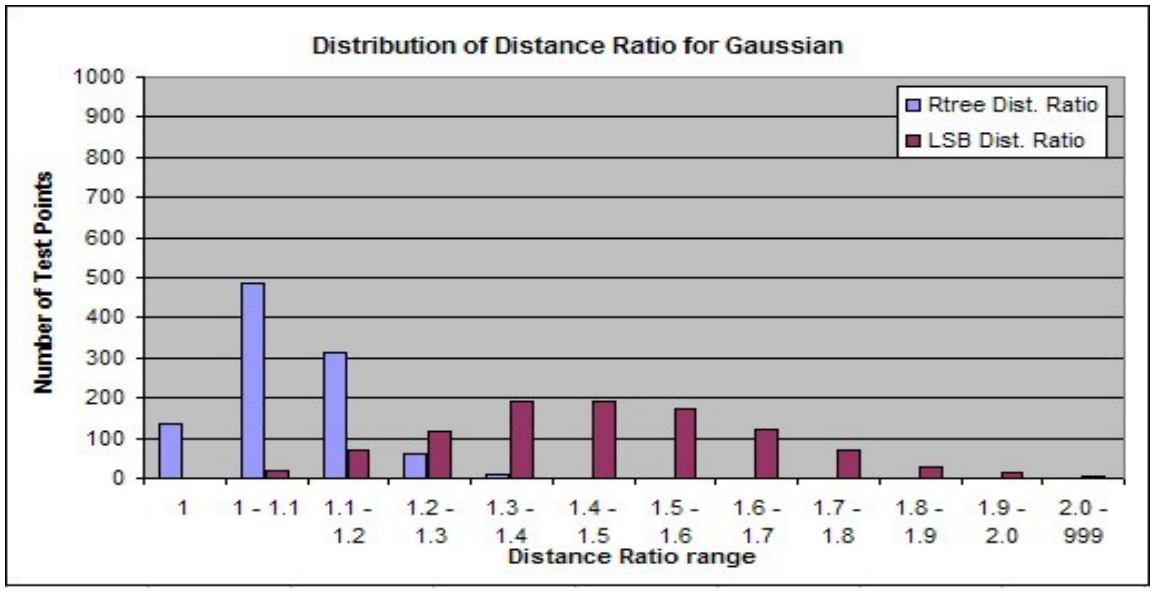



Figure 82. Distance Ratio distribution for R-Forest and LSB-Forest using Gaussian data set and $K = 4$.

Just as with the clustered data results, Gaussian data results fared just as good. As K increases, seen in Table 12, one can see an increase in the number of correct NN returned (9% for K equal 2 to 13.4% for K equal 4). Of course this was not as dramatic as it was for clustered, but still an improvement over LSB-Forest. When I review the percentage of test results where my distance ratio was better than that of LSB-Forest, I see that 97.7% to 98.2% my results are better. The lower bound was again held for Gaussian just as it was for clustered and uniformed when K was 3 or 4. Finally for Gaussian at K equal to 2, I also saw cases where the upper range on the lower bound was infinity because K-Factor prune was not always used in every test point, hence infinity on the lower bound [75].

### 7.3.4 Real Data Set Corel

Next I will present my experiment results for 32 dimensional Corel real data set results [97]. In table 13, the parameters used by both methods, as well as the total size in MB of indices built are listed (note the number of points is only 65,000 versus 99,000 as in previous experiments).

Table 13. Summary of Parameters (for 32 dimensional Corel real data)

| | **Page size (KB)** | **# of pages accessed** | **Number of indices** | **Number of points** | **Total size of index (in MB)** |
|---|---|---|---|---|---|
| **R-Forest** | 4K | 90 | 81 | 65,000 | 15MB |
| **LSB-Forest** | 4K | 90 | 20 | 65,000 | 370MB |

In table 14 I have listed the results for R-Forest and LSB-Forest with increasing values for K-Factor followed by figure 83, which shows a histogram for the test points distributions by increasing distance ratios.

Table 14. Comparison of R-Forest and LSB-Forest for real data set Corel.

| **Method** | **k** | **Distance Ratio** | | | | **% of exact match** | **% of cases where R-Forest out performs LSB-Forest** | **Lower bound ratio (exclude outliers -- see below)** |
|---|---|---|---|---|---|---|---|---|
| | | **Min** | **Mean** | **Standard Deviation** | **Maximum** | | | |
| LSB-Forest | N/A | 1 | 1.450399 | 0.40365 | 3.294505 | 7.9 | N/A | 4 (guar.) |
| R-Forest | 1 | 1 | 1.264915 | 0.29098 | 3.245898 | 23.1 | 68.3 | ∞ |
| R-Forest | 2 | 1 | 1.200835 | 0.24498 | 2.766588 | 30.9 | 73.0 | 1.067 – 2.000 |
| R-Forest | 3 | 1 | 1.189478 | 0.24496 | 2.808308 | 31.8 | 74.8 | 1.191 – 3.000 |
| R-Forest | 4 | 1 | 1.190283 | 0.2466 | 2.766588 | 31.1 | 74.5 | 1.148 – 4.000 |

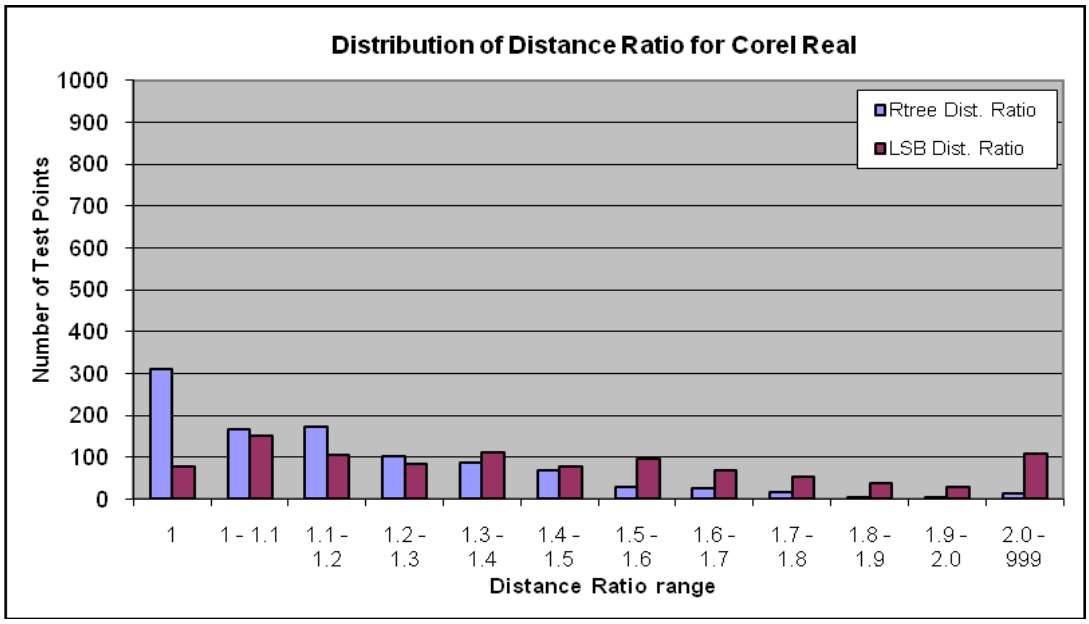



Figure 83. Distance Ratio distribution for R-Forest and LSB-Forest using real data set Corel, and $K = 4$.

For real data set Corel Table 14 and Figure 83, again one can see a slight improvement in the average distance ratio, yet both R-Forest and LSB-Forest did have numerous cases where the maximum distance ratio was over 2.5. For my method around 70% of the test fell into the distance ratio range of 1 to 1.3 where as 70% of the LSB-Forest tests fell into the range from 1 to 1.7. The noticeable improvement comes from the number of correct NN returned, for my approach as I increased prune factor (as seen in Table 14) I see improvement in the number of correct NN from 200 to 300 tests. Note, that in my tests, there are 4 cases that the actual nearest neighbor distance is extremely small. This tends to skew the lower bound (to nearly 0 if there is enough pruning, and to $\infty$ if there is not). Thus, for the lower bound I remove those points to allow one to see the range more clearly but just for the lower bound column [75].

### 7.3.5 Real Data Sets NUSWIDE Research Site

This data set was comprised of 269,648 images, for each image 5 different analysis where run, generating 5 data sets with different number of dimensions but with no specific data distribution. These sets where located at http://lms.comp.nus.edu.sg/research/NUS-WIDE.htm used as research data by researchers who maintained the site, this set again will be referred to as the NUSWIDE Set. In Table 15, I have summarize the number of dimensions, analysis and assigned a name to each sub set that will be used though this paper further details can be found in [23].

Table 15. NUSWIDE Data sets listing name, dimension and a short description

| Data Set Name | Number of Dimensions | Short Description |
|---|---|---|
| CH_64D | 64 | Color Histogram |
| CORR_144D | 144 | Color Correlogram |
| EDH_75D | 75 | Edge direction histogram |
| WT_128D | 128 | Wavelet texture |
| CM55_225D | 225 | Block-wise color moments |

All 5 data set were normalized to vales between 0.0 and 1.0, so it could be converted to integers as required by LSB-Forest, and so that accurate test point comparisons could be made between R-Forest and LSB-Forest. For testing the last 1,000 points of each set were taken as test points, leaving 268,648 points in each data set. Though out this chapter I have always used 4K pages for all of my testing, because three of the data sets have more than 120 dimensions the page size was increase to 12K, this was done because R-Tree needed a larger page size to increase fan out otherwise it would

be 2 to 4 depending on the number of dimensions. When I adjusted the page size for R-Forest the same adjustment was made for LSB-Forest so IO would be a fare comparison.

#### 7.3.5.1 Real CH_64D Data Set

Next I will present my experiment results for 64 dimensional color histogram real set results referred to as CH_64D [76]. In table 16, the parameters used by both methods, as well as the total size in MB of indices built are listed.

Table 16. Summary of Parameters (for 64 dimensional CH_64D real data)

| | Page size (KB) | # of pages accessed | Number of indices | Number of points | Total size of index (in MB) |
|---|---|---|---|---|---|
| **R-Forest** | 4K | 90 | 81 | 268,648 | 169MB |
| **LSB-Forest** | 4K | 90 | 20 | 268,648 | 2,980MB |

In table 17 I have listed the results for R-Forest and LSB-Forest with increasing values for K-Factor followed by figure 84, which shows a histogram for the test points distributions by increasing distance ratios.

Table 17. Comparison of R-Forest and LSB-Forest for real data set CH_64D.

| Method | K | Distance Ratio | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Min | Mean | Median | Stand. Dev. | Max. | % of exact match | % of cases where R-Forest out performs LSB-Forest | Lower bound ratio |
| LSB-Forest | N/A | 1 | 1.51277 | 1.46135 | 0.743195 | 13.7169 | 5.6 | N/A | 4.0 |
| R-Forest | 1 | 1 | 2.95651 | 1.44126 | 28.26309 | 885.7069 | 9.5 | 54.9 | ∞ |
| R-Forest | 2 | 1 | 2.87701 | 1.38330 | 28.25219 | 885.7069 | 10.5 | 89.3 | 0.0 – 1.998 |
| R-Forest | 3 | 1 | 2.87888 | 1.38059 | 28.26095 | 885.7069 | 10.4 | 89.3 | 0.0 – 2.999 |
| R-Forest | 4 | 1 | 2.90062 | 1.39476 | 28.26003 | 885.7069 | 10.0 | 57.1 | 0.0 – 3.997 |

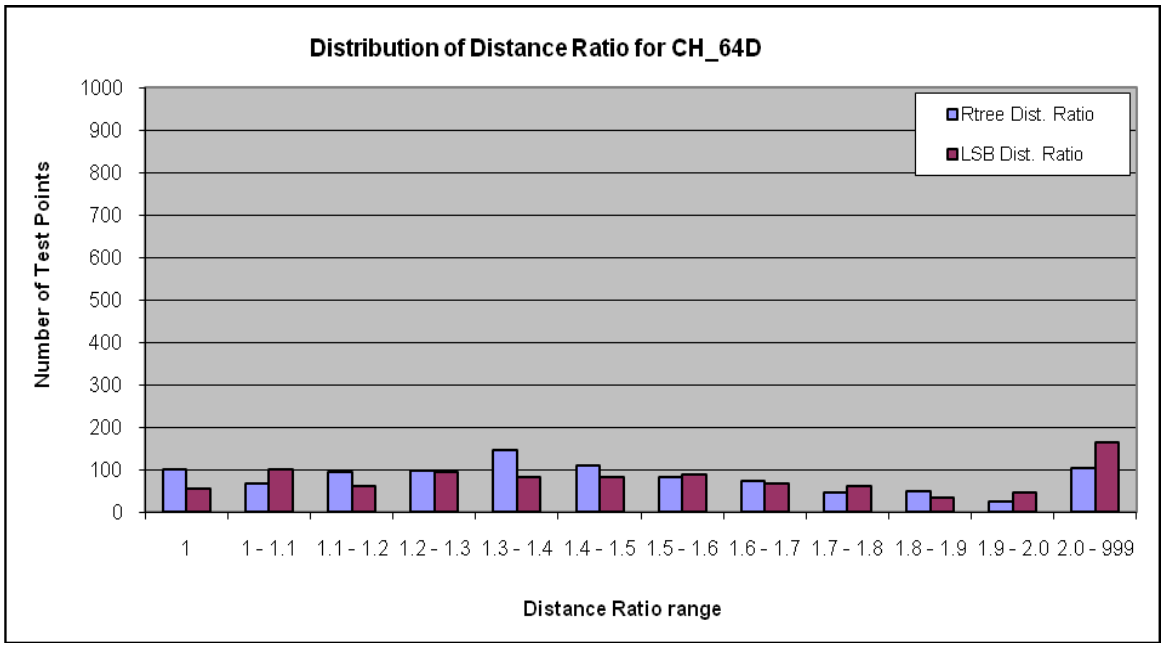



Figure 84. Distance Ratio distribution for R-Forest and LSB-Forest using real CH_64D data set, and $K = 4$.

A first look at table 17 and figure 84 it appears nether method did all that well, turns out this data set was a little unique. First there were 20-30 identical points, second the last 1,000 test points had several points that were identical to the existing points, and third several of these test points were very close to their nearest neighbor. A review of table 17 maximum distance ratio shows that LSB-Forest had a case were the distance ratio was 13.71691, actually there were 7 case were the distance ratio was greater the 4.0 which shows LSB-Forest appears to have broken its guarantee of retuning a point that was c-approximant. In these cases the test point's NN was very close and the ANN that LSB-Forest returned was far enough away that the distance ratio was greater than 4.0, this usually occurred when the distance from the test point to its NN was very small. An example, the distance from test point 76 to its NN is 0.00002364 but LSB-Forest returned an ANN point who's distance was 0.0001, yielding a distance ratio of 4.230119. R-Forest suffered from the same problem, there were numerous cases were the test point's NN was very close and the ANN returned was far enough away that the distance ratio became large, with the worst case example being test point 215 who's NN distance was 0.000000101 and R-Forest returned a point who's distance to test point 215 was 0.00089747 leading to a distance ratio of 885.706738 (it should be pointed out that LSB-Forest retuned a point who's distance was 0.0 in this case possible indicating some rounding error occurred).

Because my average and maximum distance ratio were skewed do to a hand full of points, I added an additional column to table 17, the median distance ratio. This was done to give a better picture of the distance ratio from the different sets of experiments.

From the median one can see that R-Forest did better in that for all four sets of experiments the median distance ratio was lower than LSB-Forest.

A further review of the test data showed that for LSB-Forest had 106 cause were it returned a ANN distance less than the test points true NN, in turn making the distance ratio less than 1, this was most likely due to rounding error as the data set must converted from floating point values to integers, before building an LSB-Forest. Finally LSB-Forest also had numerous cases were a test points ANN returned had a distance of 0.0 when its true NN was greater than 0.0, again most likely caused by rounding when the distance was very small.

Looking at all of these details it appears that R-Forest did not do that well when compared to LSB-Forest for data set CH_64D, but looking at table 17 and 84 one can see that 57% - 89% of the time the ANN returned was better than results returned from LSB-Forest. Also as I examined the data from the 1,000 test I see that both methods did not do well when the distance from the test point to its NN where really small even 0.0. One item I should point out is that with R-Forest I have better control of the amount of access, so I ran additional test on this data set and found out that my poor performance was due to restricted access, and as I increased the amount of allowed access the quality of the results improved.

#### 7.3.5.2 Real CORR_144D Data Set

Next I will present my experiment results for144 dimensional color correlogram real data set [76]. In table 18, the parameters used by both methods, as well as the total

size in MB of indices built are listed, it should be noted that I set the page size 12KB for both methods because of the number of dimensions.

Table 18. Summary of Parameters (for 144 dimensional CORR_144D real data)

| | **Page size (KB)** | **# of pages accessed** | **Number of indices** | **Number of points** | **Total size of index (in MB)** |
|---|---|---|---|---|---|
| **R-Forest** | 12K | 90 | 81 | 268,648 | 302MB |
| **LSB-Forest** | 12K | 90 | 20 | 268,648 | 5,510MB |

In table 19 I have listed the results for R-Forest and LSB-Forest with increasing values for K-Factor followed by figure 85, which shows a histogram for the test points distributions by increasing distance ratios.

Table 19. Comparison of R-Forest and LSB-Forest for real data set CORR_144D.

| **Method** | **k** | **Distance Ratio** | | | | **% of exact match** | **% of cases where R-Forest out performs LSB-Forest** | **Lower bound ratio** |
|---|---|---|---|---|---|---|---|---|
| | | **Min** | **Mean** | **Standard Deviation** | **Maximum** | | | |
| LSB-Forest | N/A | 1 | 1.524554 | 0.365476 | 3.23035 | 1.9 | N/A | N/A |
| R-Forest | 1 | 1 | 1.43036 | 0.32154 | 4.761805 | 1.8 | 62.3 | ∞ |
| R-Forest | 2 | 1 | 1.41324 | 0.32778 | 4.761805 | 1.9 | 63.4 | 1.006 – 1.997 |
| R-Forest | 3 | 1 | 1.408319 | 0.33408 | 4.761805 | 2.1 | 64.5 | 1.062 – 2.999 |
| R-Forest | 4 | 1 | 1.409956 | 0.32538 | 4.761805 | 2.3 | 64.6 | 1.055-3.997 |

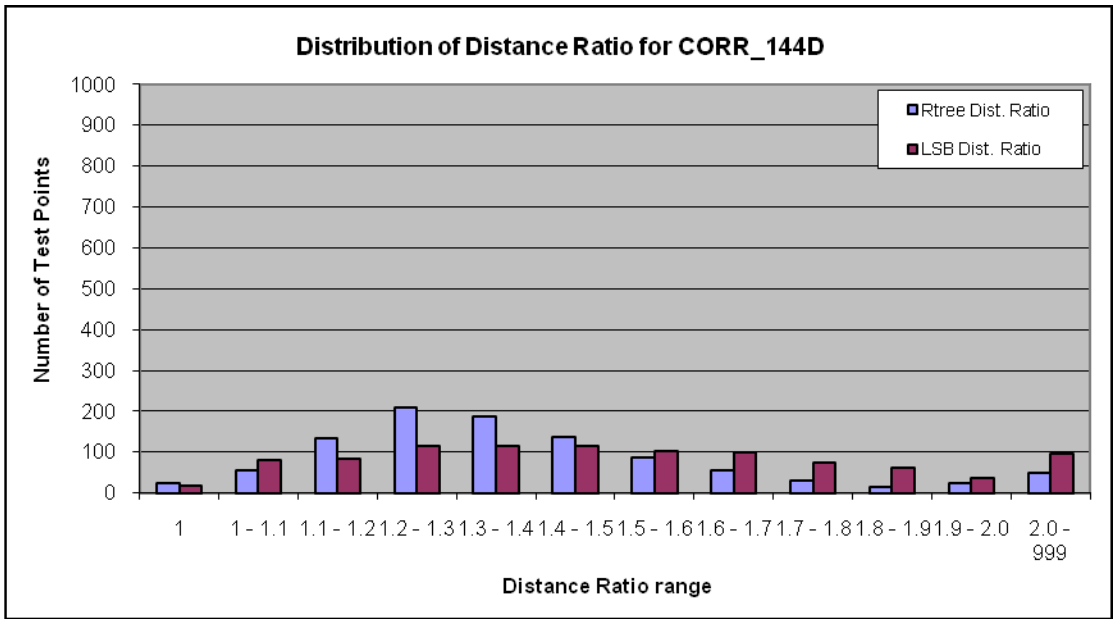



Figure 85. Distance Ratio distribution for R-Forest and LSB-Forest using real data set CORR_144D, and $K = 4$.

For this data set LSB-Forest did reasonable well, in that no test points had a distance ratio greater than 4.0. Were as R-Forest had a maximum distance ratio of 4.761805, turns out that of the 1,000 test points exactly 2 test points had distance ratio greater then 4.0, in both cases like the previous data set the true NN was relatively close so being off by a little amount created a large distance ratio. Again I ran a second test with restricted access increased and found that the quality of these two points improved which means that limited access was the reason why R-Forest missed the better ANN causing a larger distance ratio.

Looking at figure 85, we see that for R-Forest nearly 50% of the test points had a distance ratio of less than 1.5, where as for LSB-Forest 70% of the points fell between 1.0 and 1.7. Finally in table 20 62% to 64% of the time R-Forest retuned an ANN point closer to the test point than the results returned by LSB-Forest.

#### 7.3.5.3 Real EDH_75D Data Set

Next I will present my experimental results for 75 dimensional EDH_75D real data set [76]. In table 20, the parameters used by both methods, as well as the total size in MB of indices built are listed.

Table 20. Summary of Parameters (for 75 dimensional EDH_75D real data)

| | Page size (KB) | # of pages accessed | Number of indices | Number of points | Total size of index (in MB) |
|---|---|---|---|---|---|
| **R-Forest** | 4K | 90 | 81 | 268,468 | 222MB |
| **LSB-Forest** | 4K | 90 | 20 | 268,468 | 3,680MB |

In table 21 I have listed the results for R-Forest and LSB-Forest with increasing values for K-Factor followed by figure 86, which shows a histogram for the test points distributions by increasing distance ratios.

Table 21. Comparison of R-Forest and LSB-Forest for real data set EDH_75D.

| Method | k | Distance Ratio | | | | % of exact match | % of cases where R-Forest out performs LSB-Forest | Lower bound ratio |
|---|---|---|---|---|---|---|---|---|
| | | Min | Mean | Standard Deviation | Maximum | | | |
| LSB-Forest | N/A | 1 | 1.43604 | 0.291533 | 3.400322 | 1.5 | N/A | N/A |
| R-Forest | 1 | 1 | 1.22230 | 0.157674 | 2.341839 | 4.4 | 79.3 | ∞ |
| R-Forest | 2 | 1 | 1.197081 | 0.132343 | 2.049986 | 5.7 | 82.7 | 1.096 – 1.999 |
| R-Forest | 3 | 1 | 1.214195 | 0.145069 | 2.11797 | 4.9 | 80.0 | 1.394 – 2.996 |
| R-Forest | 4 | 1 | 1.24519 | 0.161797 | 2.37321 | 4.1 | 74.8 | 1.405 – 3.992 |

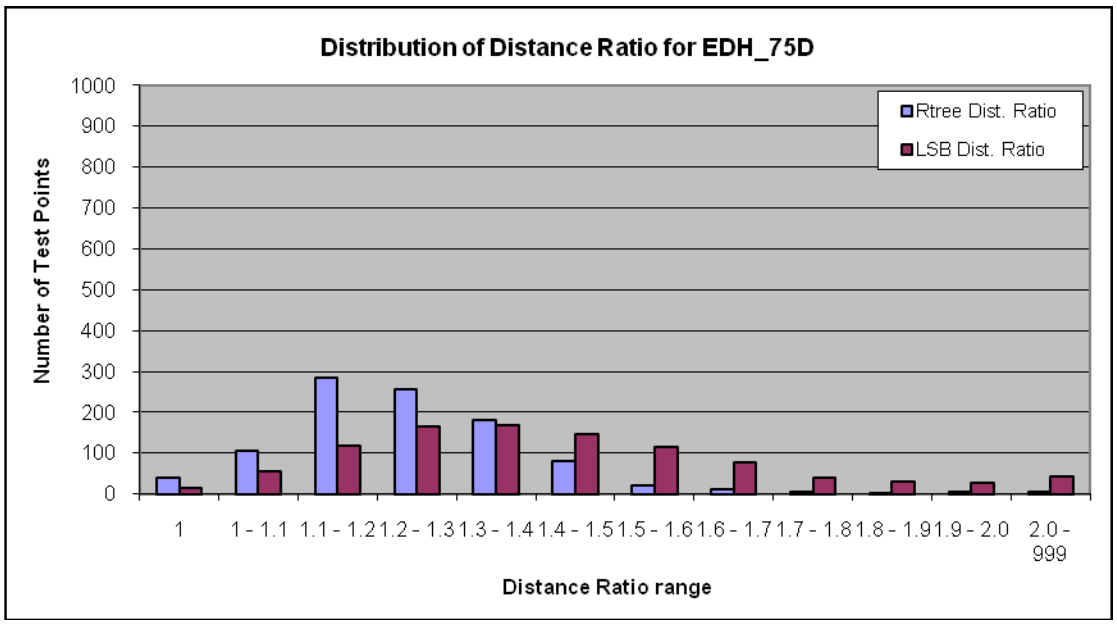



Figure 86. Distance Ratio distribution for R-Forest and LSB-Forest using real data set EDH_75D, and $K = 4$.

Like the previous data set R-Forest preformed very well on EDH_75D when compared to LSB-Forest. From the table 21 one can see that the average distance ratio was better and the maximum distance ratio was less than any test from LSB-Forest. Looking at the number of ANN points returned by R-Forest I find that 74% to 82% of the tests returned a better ANN then that returned by LSB-Forest. This is further supported by the distance ratio histogram in figure 86 whereby approximately 90% of the ANN returned had a distance ratio less than 1.5 as compared to LSB-Forest.

#### 7.3.5.4 Real WT_128D Data Set

Next I will present my experimental results for 128 dimensional WT_128D real data set [76]. In table 22, the parameters used by both methods, as well as the total size

in MB of indices built are listed, also note the use of 12KB page versus 4KB used for other data sets.

Table 22. Summary of Parameters (for 128 dimensional WT_128D real data)

| | **Page size (KB)** | **# of pages accessed** | **Number of indices** | **Number of points** | **Total size of index (in MB)** |
|---|---|---|---|---|---|
| **R-Forest** | 12K | 90 | 81 | 268,648 | 313MB |
| **LSB-Forest** | 12K | 90 | 20 | 268,648 | 4,640MB |

In table 23 I have listed the results for R-Forest and LSB-Forest with increasing values for K-Factor followed by figure 87, which shows a histogram for the test points distributions by increasing distance ratios.

Table 23. Comparison of R-Forest and LSB-Forest for real data set WT_128D.

| **Method** | **k** | **Distance Ratio** | | | | **% of exact match** | **% of cases where R-Forest out performs LSB-Forest** | **Lower bound ratio** |
|---|---|---|---|---|---|---|---|---|
| | | **Min** | **Mean** | **Standard Deviation** | **Maximum** | | | |
| LSB-Forest | N/A | 1 | 1.467389 | 0.329996 | 3.384279 | 2.6 | N/A | N/A |
| R-Forest | 1 | 1 | 1.219113 | 0.183563 | 3.634738 | 10.7 | 77.3 | ∞ |
| R-Forest | 2 | 1 | 1.191621 | 0.176986 | 3.634738 | 13.9 | 80.4 | 1.009 – 1.999 |
| R-Forest | 3 | 1 | 1.19140 | 0.17599 | 3.634738 | 13.8 | 81.1 | 1.073 – 2.998 |
| R-Forest | 4 | 1 | 1.209311 | 0.187608 | 3.684613 | 12.3 | 80.0 | 1.080 – 3.999 |

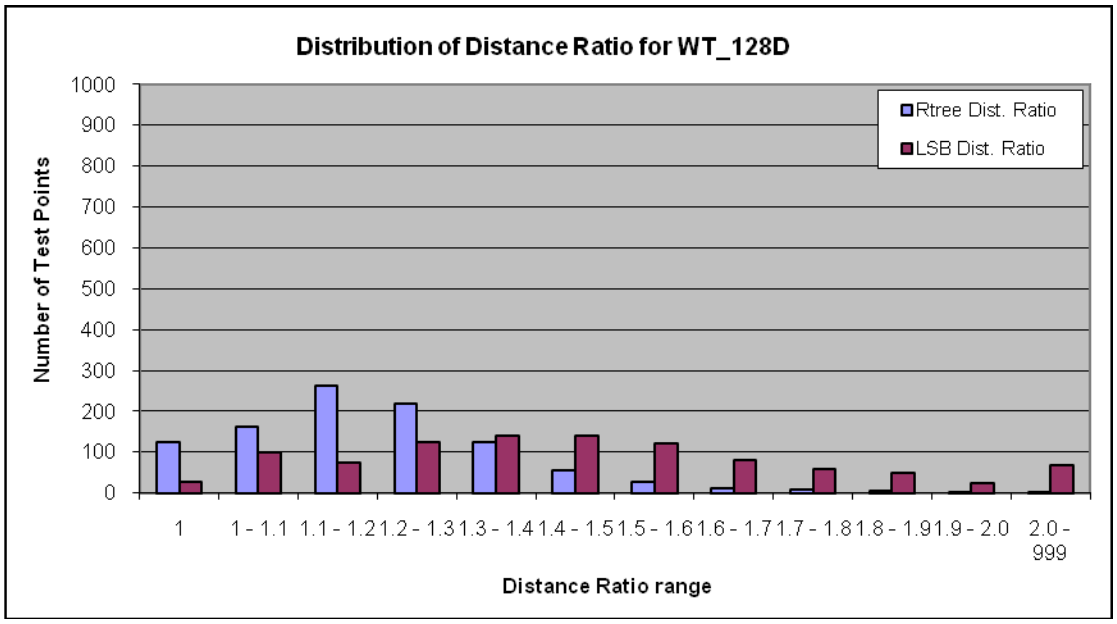



Figure 87. Distance Ratio distribution for R-Forest and LSB-Forest using real data set WT_128D, and $K = 4$.

For WT_128D we can see from table 23, that the average distance ratio was better in all cases (K = 1 to K = 4) although the max distance ratio was higher. After checking the test data in more detail it was found that only 1 point (test point 255) had a distance ratio greater than 3.6 (for all test K = 1 to K = 4), with 2 points above 2.0 and remaining 997 points were below 2.0. Were as LSB-Forest had over 50 points with distance ratio above 2.0, I only point this out because R-Forest maximum distance ratio was larger but for only 1 case. Looking at table 23 and figure 87 we can see that 77% to 81% of the time R-Forest returned an ANN point closer to the test point's NN as compared to the results returned by LSB-Forest.

#### 7.3.5.5 Real CM55_225D Data Set

Next I will present my experimental results for 225 dimensional CM55_225D real data set [76]. In table 24, the parameters used by both methods, as well as the total size in MB of indices built are listed, note the use of 12K page, versus 4K pages used in other test. I should also point out that of all experiments in this paper this set had the largest number of dimensions.

Table 24. Summary of Parameters (for 225 dimensional CM55_225D real data)

| | **Page size (KB)** | **# of pages accessed** | **Number of indices** | **Number of points** | **Total size of index (in MB)** |
|---|---|---|---|---|---|
| **R-Forest** | 12K | 90 | 81 | 268,648 | 655MB |
| **LSB-Forest** | 12K | 90 | 20 | 268,648 | 8,730MB |

In table 25 I have listed the results for R-Forest and LSB-Forest with increasing values for K-Factor followed by figure 88, which shows a histogram for the test points distributions by increasing distance ratios.

Table 25. Comparison of R-Forest and LSB-Forest for real data set CM55_225D.

| Method | K | Distance Ratio | | | | % of exact match | % of cases where R-Forest out performs LSB-Forest | Lower bound ratio |
|---|---|---|---|---|---|---|---|---|
| | | Min | Mean | Standard Deviation | Maximum | | | |
| LSB-Forest | N/A | 1 | 1.396671 | 0.223590 | 2.97950 | 0.5 | N/A | N/A |
| R-Forest | 1 | 1 | 1.181420 | 0.115906 | 2.224905 | 2.1 | 88.4 | ∞ |
| R-Forest | 2 | 1 | 1.174161 | 0.111315 | 2.268678 | 2.9 | 88.5 | 1.005 – 1.995 |
| R-Forest | 3 | 1 | 1.183311 | 0.156851 | 1.918851 | 2.9 | 87.0 | 1.415 – 2.999 |
| R-Forest | 4 | 1 | 1.205510 | 0.118839 | 2.415298 | 2.2 | 84.2 | 1.643 – 3.999 |

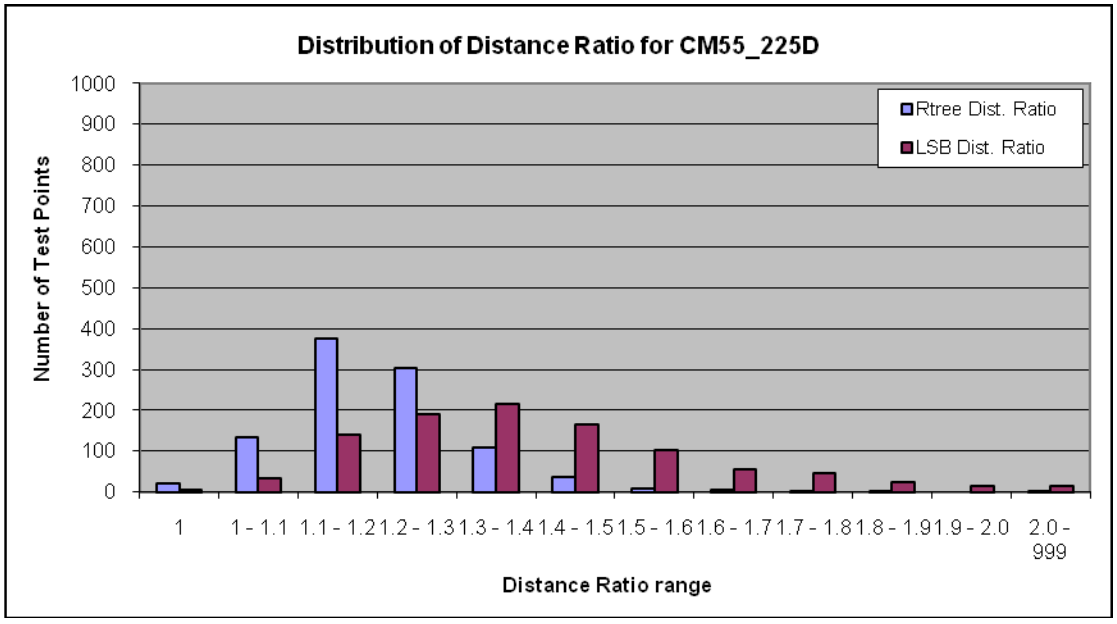



Figure 88. Distance Ratio distribution for R-Forest and LSB-Forest using real data CM55_225D set, and $K = 4$.

Finally of all of the NUSWIDE real data sets CM55_225D performed the best, from figure 88 and table 25 we can see that around 900 points had a distance ratio less 1.4 and that 84% to 88% of the ANN returned were closer to the correct NN than the results returned by LSB-Forest. The average and maximum distance ratio for R-Forest were better than that of LSB-Forest.

#### 7.3.5.6 Conclusion on NUSWIDE Sets

In conclusion these five real data sets derived from 269,648 images performed reasonably well when compared to LSB-Forest. Looking at the data I see that 50% to 88% of the experiments R-Forest returned an ANN point that was closer to the exact NN then results returned by LSB-Forest. Both methods had trouble with duplicate points and points who's exact NN distance was very small, in some cases causing the distance ratio to be skewed (mostly this happened in CH_64D data set). One item that should be pointed out was the storage cost, in all cases LSB-Forest required 13 to 18 times as much storage as R-Forest for the same number of points and dimension. In fact the total storage of all five R-Forests for all NUSWIDE sets was 2.29 GB which was less than the LSB-Forest storage requirements for the set with the fewest number of dimensions CH_64D which required 2.9GB.

### 7.3.6 Real KDDCUP04BIO_74D Data Set

Next I will present my experiment results for 74 dimensional KDDCUP04BIO_74D real data set [55]. In table 26, the parameters used by both methods, as well as the total size in MB of indices built are listed. This set was used for

Knowledge Discovery and Data Cup competition in 2004, it is comprised of 145,751 points which contain data about protein homology. Like the other experiments the first 144,751 points were used as data points and the last 1,000 for test points. This set was also normalized to a range of 0 to 1.0 so that comparisons could be run with LSB-Forest.

Table 26. Summary of Parameters (for 74 dimensional KDDCUP04BIO_74D real data)

| | **Page size (KB)** | **# of pages accessed** | **Number of indices** | **Number of points** | **Total size of index (in MB)** |
|---|---|---|---|---|---|
| **R-Forest** | 4K | 90 | 81 | 144,751 | 118MB |
| **LSB-Forest** | 4K | 90 | 20 | 144,751 | 1,750MB |

In table 27 I have listed the results for R-Forest and LSB-Forest with increasing values for K-Factor followed by figure 89, which shows a histogram for the test points distributions by increasing distance ratios.

Table 27. Comparison of R-Forest and LSB-Forest for real data set KDDCUP04BIO_74D.

| Method | K | Distance Ratio | | | | % of exact match | % of cases where R-Forest out performs LSB-Forest | Lower bound ratio |
|---|---|---|---|---|---|---|---|---|
| | | Min | Mean | Standard Deviation | Maximum | | | |
| LSB-Forest | N/A | 1 | 1.46818 | 0.28308 | 3.050342 | 1.0 | N/A | N/A |
| R-Forest | 1 | 1 | 1.23911 | 0.156667 | 2.151665 | 5.2 | 81.2 | ∞ |
| R-Forest | 2 | 1 | 1.212235 | 0.141057 | 2.052110 | 7.1 | 84.1 | 1.096 – 1.999 |
| R-Forest | 3 | 1 | 1.216396 | 0.144005 | 1.998548 | 7.7 | 84.4 | 1.241 – 2.999 |
| R-Forest | 4 | 1 | 1.241632 | 0.152028 | 2.064364 | 7.0 | 80.2 | 1.241 – 3.998 |

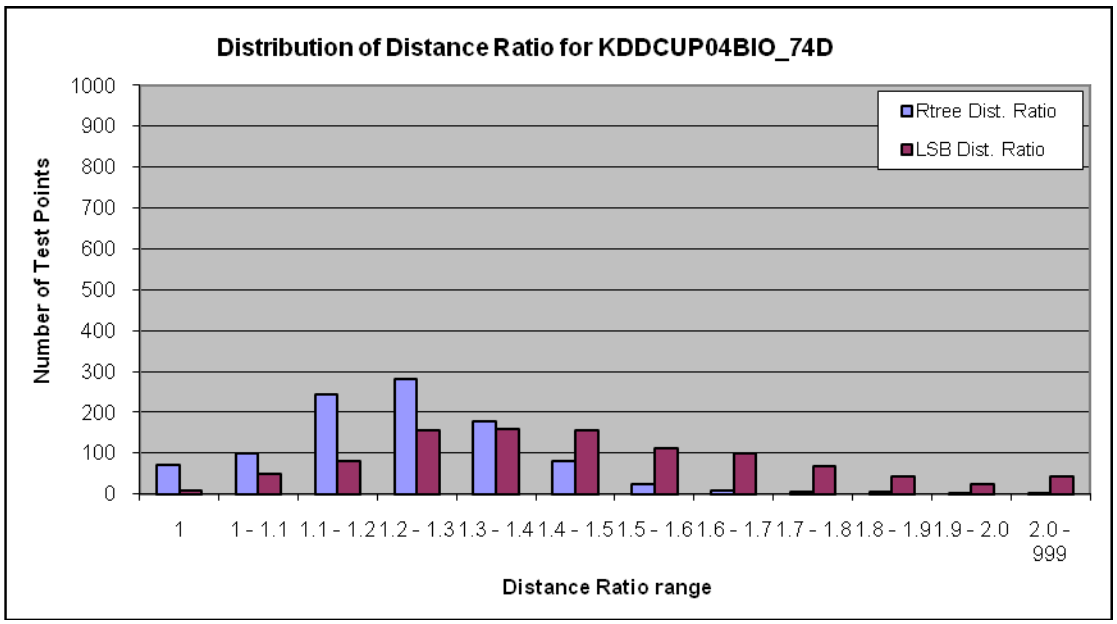



Figure 89. Distance Ratio distribution for R-Forest and LSB-Forest using real data set KDDCUP04BIO_74D, and $K = 4$.

For this data set again from tables 27 and figure 89 one case see that for R-Forest the average distance ratio and maximum distance ratio were better than results returned by LSB-Forest. In fact a more detailed review of the data reviled that R-Forest had 2 points with a distance ratio greater than 2.0 were as LSB-Forest over 40 points above 2.0. Again the majority of the ANN points returned by R-Forest 81% to 84% were closer to the test points exact NN then the results returned by LSB-Forest.

## 7.4 Scaling

In this section I will present numerous experimental results to show how well R-Forest scales in relation to increasing dimensionality and points. These results will be compared to LSB-Forest using the same data sets and test points.

### 7.4.1 Dimensional Scaling Results

An additional question needed to be answered: does my approach to ANN scale with an increase in the number of dimensions. To test this, I need to look at how the distance ratio changes as the number of dimensions increase. I generated four additional uniform, clustered and Gaussian data sets with increasing dimensionality (32, 48, 64, 80 and 96 dimensions), each having 100,000 points (again I used the last 1,000 as test points and the first 99,000 as my data sets). Using the same procedures and parameters as in my previous test, I generated results for each of the data sets, and then plotted the average distance ratio for each data set as I am only interested in how the distance ratio changes as the number of dimensions increase. The results can be seen in Figure 90 to 92. What I am looking for is the effect on the average distance ratio as the number of dimensions

increases. As a comparison I ran the same test using LSB-Forest to see how my average distance ratio scaled with LSB-Forest distance ratio as the number of dimensions increased.

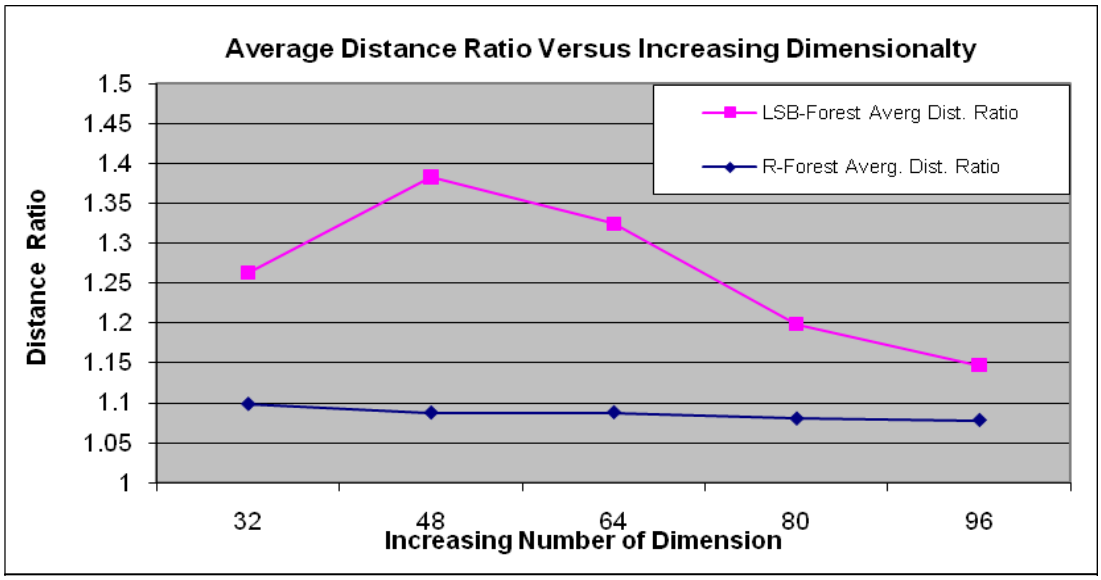



Figure 90. Average distance ratio for uniform data versus increasing number of dimensions.

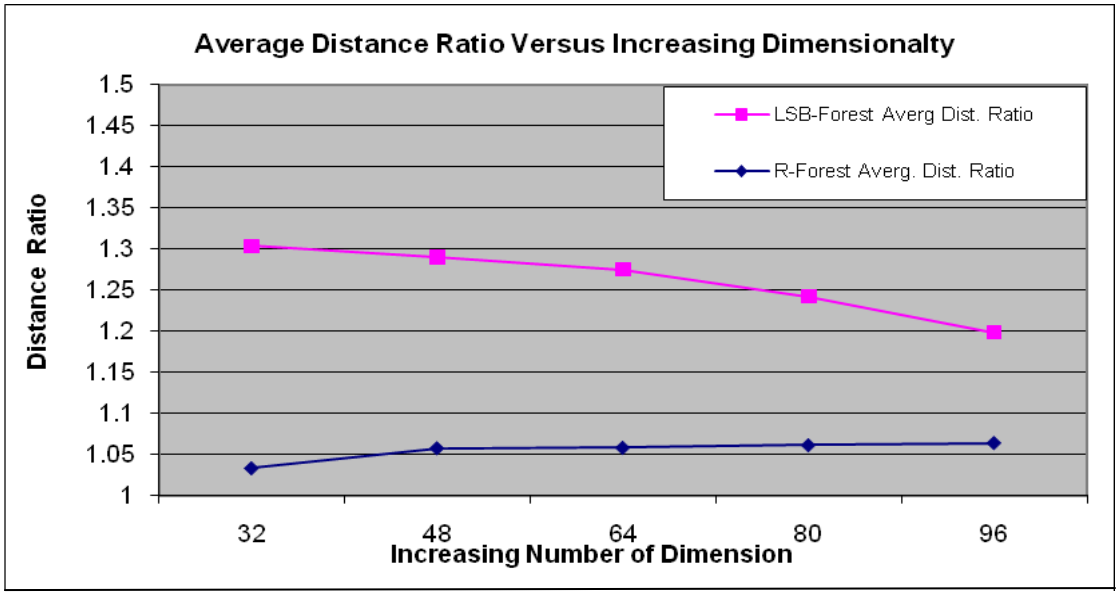



Figure 91. Average distance ratio for clustered data versus increasing number of dimensions.

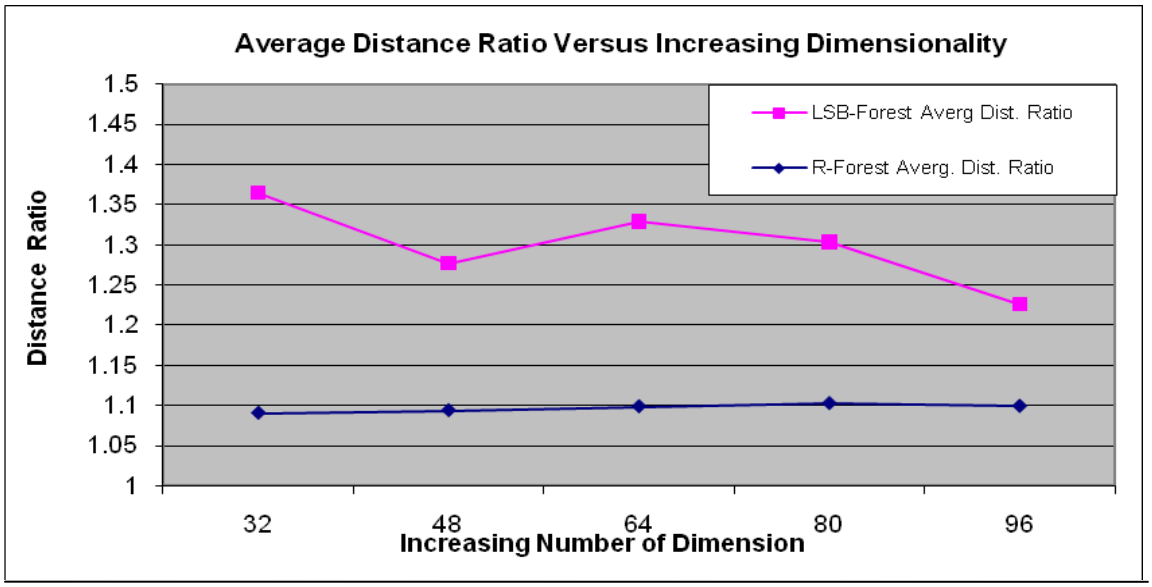



Figure 92. Average distance ratio for Gaussian data versus increasing number of dimensions.

In Figure 90 to 92 above, one can see a change in the average distance ratio for R-Forest for all three data distributions does not change significant, which indicates that my method scales with an increase in the number of dimensions over different data distributions. Overall I can say from this test that this method remains consistent and does not appear to degrade as the dimensionality increases. From these experiments one can see the LSB-Forest also performs relatively well as the number of dimension increases.

### 7.4.2 Point Scaling Results

The next I will present how well my method scales as the number of points increase. To show this, five data sets for each of the three data distributions were created. These sets were of increasing size starting at 500,000 points through 2,500,000 points in increments of 500,000 points. For each set, I ran R-Forest and LSB-Forest using the same parameters used in previous experiments, and again used 1,000 test points of the same distribution. I am looking for is any significant change in the minimum, average and maximum distance ratios for the test points as the number of data points increases. Does the distance ratio increase rapidly as the number of data points increase which would indicate that my method degrades or not as the number of points increases? Finally, as a comparison I wanted to look at how well Single R-Tree performs against R-Forest, by comparing how well R-Forest performs with the minimal amount of work and access restriction as a percentage of the number of pages. Details on these will be given in the experimental set up section.

#### 7.4.2.1 Point Scale Experiment Set up

For point scale test I ran the four different tests on R-Forest, and one set of tests on LSB-Forest all using the same construction parameter, data sets and test points. For R-Forest the four different tests are defined below, each is meant to test a different condition over an increasing number of points.

- Single R-Tree – This is defined as exactly one R-Tree with no additional pruning, MBR-Center points, or disjoint sets. It is basically an unmodified R-Tree, with one exception, code was added to restrict access, much as I do with R-Forest (R = 90 pages)
- Single Depth – This is defined as the minimal amount of work possible, it consists of picking the first R-Tree based on whether Q (our test point) is in the subset of split dimension and boundaries then traversing this index to a leaf node, search the points, return the best ANN in the leaf, then terminating the search. The total cost in pages accessed will be the height of the R-Tree index searched (typically 4-7 pages in these experiments).
- R = 90 - Restrict access to 90 pages as was done in previous experiments. It should be noted that page access will be fixed at 90 pages across all experiments even as the number of points increases.
- Access as a Percentage of R-Forest (Access = 1.79%) – This is a variation on R limited to 90 pages. Rather than limiting access to a fixed number as the number of points increases, increase the access as a percentage of the number of pages in each R-forest. The percentage amount chosen was derived from previous

experiments whereby, 90 pages was equal to 1.79% access when R-Forest has 99,000 points which requires around 5,000 pages.

For LSB-Forest (labeled as LSB-Forest (20) in the following figures), I will be using the same parameters as other experiments, a forest of 20 trees for each set of test data and 4K page size.

#### 7.4.2.2 Point Scale Results

In the next three sections broken down by data distribution, I will provide a tables showing fixed, maximum and average number of pages accessed, the amount of storage required for each Forest (R-Forest and LSB-Forest), the minimum, average and maximum distance ratio, and the number of correct NN returned. This table will be followed by a figure plotting the average distance ratio for five sets of experiments (four for R-Forest and one for LSB-Forest).

##### 7.4.2.2.1 Uniform Results

Starting with uniform data distribution, table 28 compares the storage requirements in megabytes (MB) for the raw input data files, a single R-Tree index (without any MBR-Center point storage), R-Forest comprised of 81 R-Tree indices with MBR-Center points, and LSB-Forest with 20 indices.

Table 28. Storage requirements for Uniform data

| No. Data Points | Original file Size (MB) | Single R-Tree (MB) | R-Forest 81 R-Tree Indices (MB) | LSB-Forest with 20 trees (MB) |
|---|---|---|---|---|
| 500,000 | 138.28 | 91.30 | 97.50 | 3,164.16 |
| 1,000,000 | 276.56 | 177.34 | 194.00 | 6,338.56 |
| 1,500,000 | 414.85 | 273.43 | 291.00 | 9,553.92 |
| 2,000,000 | 553.13 | 365.95 | 389.00 | 13,414.40 |
| 2,500,000 | 691.41 | 455.00 | 487.00 | 16,793.60 |

From table 28 one can see that single R-Tree and R-Forest require less storage then the original uniform input data file. This seems counter intuitive as both data structures are comprised of leaf nodes (4K pages) storing points and internal nodes (4K pages) storing MBR, none of which are 100% full. The reason for the difference has to do with the storage of each attribute for each point in the input data file. The attributes are real point, carried to at least 7 decimal places and 1 digit to the left of the decimal, with each digit and even the decimal requiring 1 byte also, finally all attributes are delimited by a single space. Therefore, in the raw data file a single attribute can require up to 10 bytes, where as in single R-Tree and R-Forest the attributes are floating point binary, requiring exactly 4 bytes with no delimiters. Both Single R-Tree and R-Forest require less storage because of the use of binary floating point versus text, with remaining storage used by the internal nodes which defines the tree structure. In the tables one can see R-Forest is typically 7-10% larger then Single R-Tree, there are two reasons, mostly the additional storage of MBR-Center points in the internal nodes, which reduces the fan out of the tree and the extra trees in the forest (requiring additional root pages).

Finally from table 28 LSB-Forest requires significantly more storage then both Single R-Tree and R-Forest, this is because of duplication of points in each of the 20

indices. One can surmise that the current LSB-Forest implementation will require at least 20 times the storage of either single R-Tree or R-Forest, but according to table 28 LSB-Forest actually consumes over 30 times the storage required by R-Forest.

The following five tables 29 to 33 give details on, Single R-Tree, R-Forest Single Depth traversal, R-Forest with restricted access R = 90, access limited to 1.79% and LSB-Forest for uniform data. From these tables I am mainly interested in the effect on average distance ratio and maximum distance ratio as the number of data points increases.

Table 29. Single R-Tree Uniform data (restricted access to 90 pages)

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 90 | 90 | 1 | 1.2357921 | 1.5722369 | 11 | N/A |
| 1,000,000 | 90 | 90 | 1 | 1.2749728 | 1.6467464 | 3 | N/A |
| 1,500,000 | 90 | 90 | 1 | 1.2796416 | 1.8361105 | 1 | N/A |
| 2,000,000 | 90 | 90 | 1 | 1.2987328 | 1.6874301 | 4 | N/A |
| 2,500,000 | 90 | 90 | 1 | 1.3080638 | 1.6823499 | 2 | N/A |

Table 30. Single Depth traversal of R-Forest Uniform data

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 4 | 4 | 1 | 1.4047821 | 1.9991306 | 2 | N/A |
| 1,000,000 | 5 | 5 | 1 | 1.4472568 | 1.9645486 | 3 | N/A |
| 1,500,000 | 5 | 5 | 1 | 1.4682898 | 2.1272115 | 1 | N/A |
| 2,000,000 | 5 | 5 | 1.089074 | 1.4651433 | 2.0519709 | 0 | N/A |
| 2,500,000 | 5 | 5 | 1.081338 | 1.5045992 | 2.2149800 | 0 | N/A |

Table 31. R = 90 applied to R-Forest Uniform data

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 90 | 90 | 1 | 1.1449490 | 1.7055257 | 59 | 2.3399 - 3.9984 |
| 1,000,000 | 90 | 90 | 1 | 1.1792920 | 1.5132410 | 28 | 2.4616 – 3.9944 |
| 1,500,000 | 90 | 90 | 1 | 1.2030123 | 1.5642070 | 27 | 2.472 – 3.9998 |
| 2,000,000 | 90 | 90 | 1 | 1.2116060 | 1.5294560 | 17 | 2.2693 – 3.9943 |
| 2,500,000 | 90 | 90 | 1 | 1.2249650 | 1.6962460 | 22 | 2.3281 – 3.9864 |

Table 32. Access 1.79% applied to R-Forest Uniform data

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 485 | 485 | 1 | 1.0752559 | 1.4326893 | 216 | 2.7778 – 4.0000 |
| 1,000,000 | 981 | 983 | 1 | 1.0657700 | 1.3459810 | 237 | 2.9581 – 3.9999 |
| 1,500,000 | 1,475 | 1,477 | 1 | 1.0695048 | 1.3982370 | 239 | 2.8605 – 3.9999 |
| 2,000,000 | 1,968 | 1,971 | 1 | 1.0639729 | 1.3240748 | 253 | 3.0169 – 3.9999 |
| 2,500,000 | 2,463 | 2,468 | 1 | 1.0615019 | 1.5754540 | 281 | 2.3589 – 3.9999 |

Table 33. LSB-Forest with 20 trees Uniform data

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 82 | 90 | 1.000003 | 1.3270760 | 2.1124630 | 0 | N/A |
| 1,000,000 | 102 | 108 | 1 | 1.3585970 | 1.8127100 | 5 | N/A |
| 1,500,000 | 102 | 108 | 1 | 1.3767020 | 1.8649410 | 3 | N/A |
| 2,000,000 | 102 | 107 | 1 | 1.4044370 | 2.0268500 | 2 | N/A |
| 2,500,000 | 102 | 107 | 1 | 1.4052360 | 2.1282670 | 3 | N/A |

In figure 94 I have plotted distance ratio on the y-axis against the uniform data sets for all five tests as listed in tables 29 through 33.

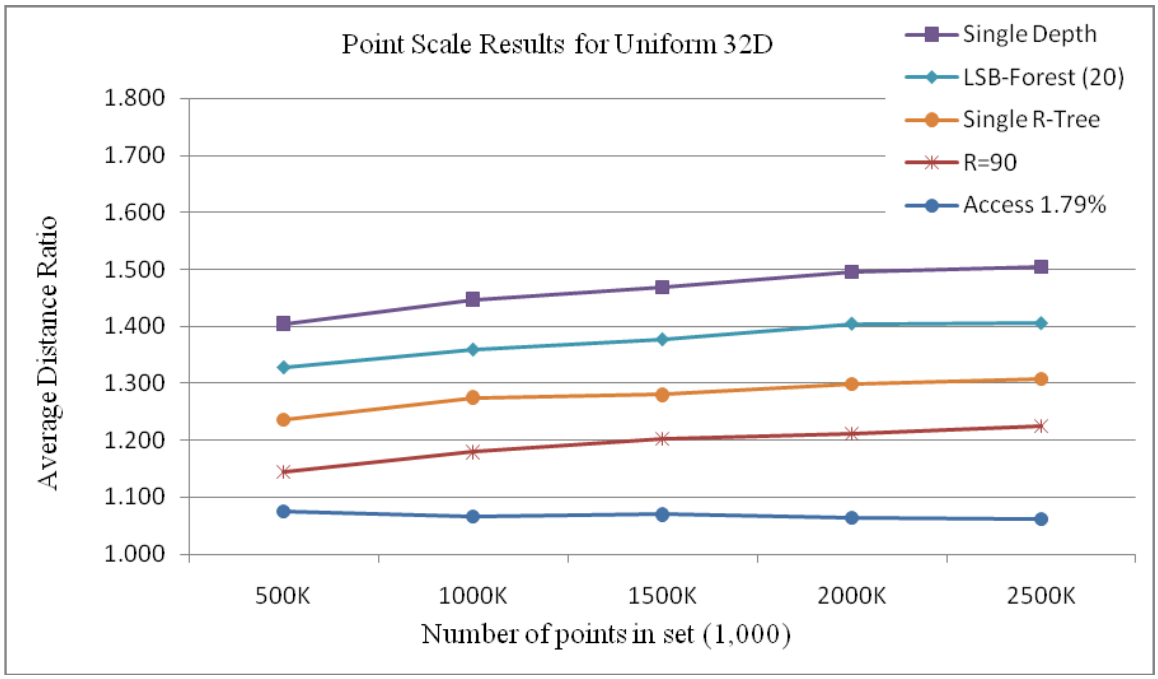



Figure 94, Results of all five experiments for uniform data with increasing number of points

In Figure 94 point scaling test for uniform data observe that both R-Forest (labed as R = 90) and LSB-Forest (labed as LSB-Forest (20) performed very well as the number of points increased. There was a change in the average distance ratio as the number of points increased but this slight increase appears to grow propotional to the increase in points. Also included in this test was an experiment with Single Depth R-Forest and Single R-Tree with restricted access at 90 pages, in the case of Single Depth R-Forest one would expect it to do the worst as it is allowed to acces only a single leaf node. What is a little suprising is that Single R-Tree with restricted access at 90 pages acually does better the LSB-Forest. Finally in the case of R-Forest with access limited to 1.79% of the

forest, we see the best results, but this would be expected because R-Forest was allowed to do more searching. There is one interesting observation concerning access limieted to 1.79% that needs to be pointed out, looking back at table 20 column "Average Page Count" which as the name suggest is the average page count of all the tests, we see that the average is less then the maximum allowed. This means that from some of the test eithor K-Factor pruning or *Mindist (Q, MBR)* < *CB* actually terminate the search before the access restriction was reached.

#### 7.4.2.2.2 Clustered Results

With clustered data distribution table 34 compares, the storage requirements in megabytes (MB) for the raw input data files, to a single R-Tree index (without any MBR-Center point storage), R-Forest with 81 indices with MBR-Center points, and LSB-Forest with 20 indices.

Table 34. Storage requirements for Clustered data

| No. Data Points | Original file Size (MB) | Single R-Tree (MB) | R-Forest 81 R-Tree Indices (MB) | LSB-Forest with 20 trees (MB) |
|---|---|---|---|---|
| 500,000 | 138.00 | 91.40 | 97.70 | 3,164.16 |
| 1,000,000 | 277.00 | 182.00 | 194.00 | 6,338.56 |
| 1,500,000 | 414.00 | 274.00 | 294.00 | 9,553.92 |
| 2,000,000 | 553.00 | 365.00 | 388.00 | 13,414.40 |
| 2,500,000 | 690.00 | 457.00 | 485.00 | 16,793.60 |

From table 34 one can see that single R-Tree and R-Forest require less storage then the original cluster input data file. Just as I saw in the uniform scale test in the previous section the storage requirements are about the same.

The following five tables 35 to 39 give details on Single R-Tree with restricted access, R-Forest Single Depth traversal, R-Forest with restricted access R = 90, access up to 1.79% and LSB-Forest all using clustered data

Table 35. Single R-Tree Clustered data (restricted access to 90 pages)

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 90 | 90 | 1 | 1.1092306 | 1.5521631 | 93 | N/A |
| 1,000,000 | 90 | 90 | 1 | 1.1436697 | 1.6029472 | 48 | N/A |
| 1,500,000 | 90 | 90 | 1 | 1.1574753 | 1.5694672 | 40 | N/A |
| 2,000,000 | 90 | 90 | 1 | 1.1662114 | 1.6084118 | 25 | N/A |
| 2,500,000 | 90 | 90 | 1 | 1.1688950 | 1.5523123 | 26 | N/A |

Table 36. Single Depth traversal of R-Forest Clustered data

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 5 | 5 | 1 | 1.3285248 | 1.9750211 | 3 | N/A |
| 1,000,000 | 5 | 5 | 1 | 1.3570832 | 1.8579776 | 5 | N/A |
| 1,500,000 | 5 | 5 | 1 | 1.3982780 | 1.8855572 | 2 | N/A |
| 2,000,000 | 5 | 5 | 1.043976 | 1.4116585 | 1.9274184 | 0 | N/A |
| 2,500,000 | 5 | 5 | 1.011823 | 1.4122558 | 1.8363388 | 0 | N/A |

Table 37. R = 90 applied to R-Forest Clustered data

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 90 | 90 | 1 | 1.0965226 | 1.5063106 | 151 | 2.4889 – 3.9998 |
| 1,000,000 | 90 | 90 | 1 | 1.1283862 | 1.4486867 | 86 | 2.6025 – 3.9994 |
| 1,500,000 | 90 | 90 | 1 | 1.1553972 | 1.5983523 | 62 | 2.4720 – 3.9998 |
| 2,000,000 | 90 | 90 | 1 | 1.1636150 | 1.5709365 | 29 | 2.439 – 4.0000 |
| 2,500,000 | 90 | 90 | 1 | 1.1787359 | 1.5098280 | 25 | 2.4533 – 3.9991 |

Table 38. Access 1.79% applied to R-Forest Clustered data

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 497 | 501 | 1 | 1.0189910 | 1.4266880 | 649 | 2.7955 – 4.0000 |
| 1,000,000 | 970 | 1,003 | 1 | 1.0167810 | 1.2681640 | 665 | 3.1505 – 4.0000 |
| 1,500,000 | 1,438 | 1,509 | 1 | 1.0218775 | 1.3485930 | 628 | 2.9617 – 4.0000 |
| 2,000,000 | 1,900 | 2,002 | 1 | 1.0176131 | 1.4096250 | 662 | 2.8369 – 4.0000 |
| 2,500,000 | 2,445 | 2,515 | 1 | 1.0193330 | 1.3051853 | 663 | 3.0403 – 4.0000 |

Table 39. LSB-Forest with 20 trees Clustered data

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 82 | 87 | 1 | 1.3686350 | 1.8822250 | 5 | N/A |
| 1,000,000 | 102 | 106 | 1 | 1.4093490 | 2.0534170 | 1 | N/A |
| 1,500,000 | 102 | 107 | 1 | 1.4202570 | 1.9923040 | 3 | N/A |
| 2,000,000 | 102 | 107 | 1 | 1.4275810 | 2.0136310 | 5 | N/A |
| 2,500,000 | 102 | 106 | 1 | 1.4410270 | 2.0657070 | 1 | N/A |

In figure 94 I have plotted distance ratio on the y-axis against the clustered data sets for all five tests as listed in tables 35 through 39.

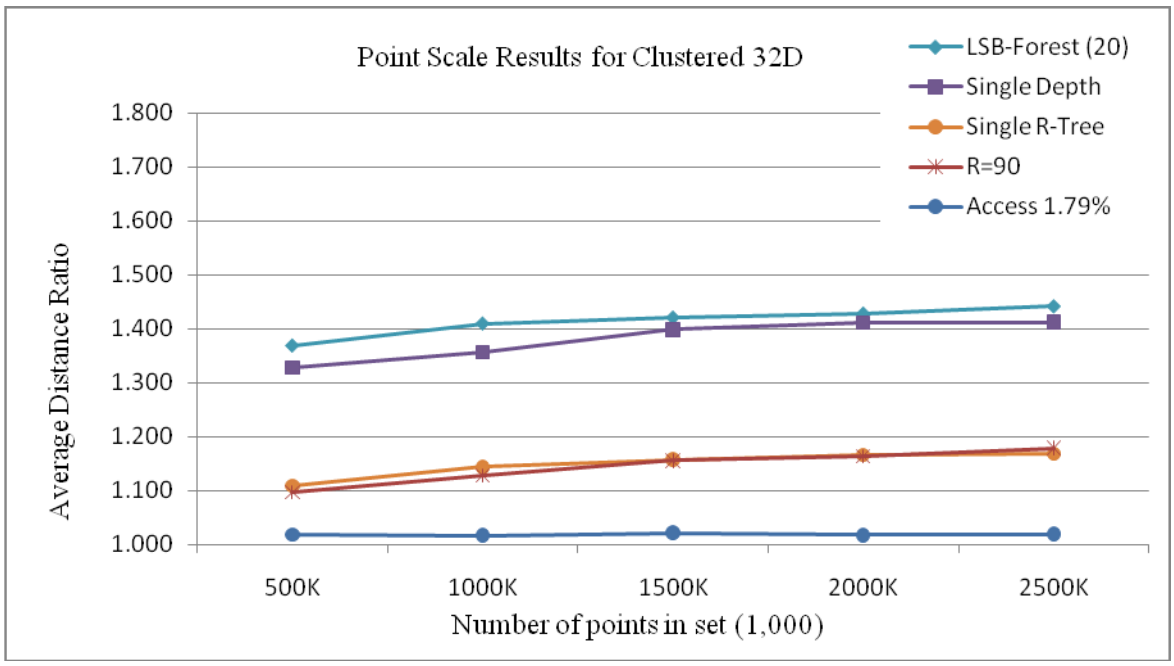



Figure 95. Results of all five experiments for clustered data with increasing number of points

In figure 95 above there is one interesting annomally that requires further explanations that is results for Single R-Tree and R-Forest with 90 page restricted access. Here both performed about the same. Which seems to indicate that R-Forest is not much of an improvement. One explanation for this has to do with the clustered data, it turns out that for this data set the points are not equally distributed across the R-Tree indices in the forest, in fact several trees end up with 6-9% of the points, this is due to the clustered data. Most likely both Single R-Tree and R-Forest are finding the clusters, thereby the points are being organized into simulare R-Tree. In turn these are always selected first during a query and access restriction cuts the search to about the same as single R-Tree, hence simular results. What is interesting about these scaling tests for clusterd data are the LSB-Forest and Single Depth R-Forest results, unlike the same tests for uniformed data, LSB-Forest actually did worst then Single Depth R-Forest, a review of tables 36 and

39 show that even the "Max. Dist Ratio" for Single Depth R-Forest was better than LSB-Forest.

#### 7.4.2.2.3 Gaussian Results

With Gaussian data distribution table 40 compares the storage requirements in megabytes (MB) for the raw input data files, to a single R-Tree index (without any MBR-Center point storage), R-Forest with 81 indices with MBR-Center points, and LSB-Forest with 20 indices.

Table 40. Storage requirements for Gaussian data

| No. Data Points | Original file Size (MB) | Single R-Tree (MB) | R-Forest 81 R-Tree Indices (MB) | LSB-Forest with 20 trees (MB) |
|---|---|---|---|---|
| 500,000 | 140.00 | 88.00 | 93.90 | 3,164.16 |
| 1,000,000 | 281.00 | 176.00 | 186.00 | 6,338.56 |
| 1,500,000 | 422.00 | 264.00 | 280.00 | 9,553.92 |
| 2,000,000 | 562.00 | 352.00 | 372.00 | 13,414.40 |
| 2,500,000 | 703.00 | 440.00 | 464.00 | 16,793.60 |

From table 40 one can see that single R-Tree and R-Forest require less storage then the original Gaussian input data file. Just as we saw in the uniform scale test in the previous section the storage requirements are about the same.

The following five tables 41 though 45 give details on Single R-Tree with restricted access, R-Forest Single Depth traversal, R-Forest with restricted access R = 90, Access up to 1.79% and LSB-Forest for Gaussian data. From these tables I am mainly interested in the effect on average distance ratio and maximum distance ratio as the number of data points increases.

Table 41. Single R-Tree Gaussian data (restricted access to 90 pages)

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 90 | 90 | 1 | 1.2128924 | 1.6002194 | 8 | N/A |
| 1,000,000 | 90 | 90 | 1 | 1.2483582 | 1.7312400 | 6 | N/A |
| 1,500,000 | 90 | 90 | 1 | 1.2658091 | 1.5902518 | 3 | N/A |
| 2,000,000 | 90 | 90 | 1 | 1.2787363 | 1.6211771 | 1 | N/A |
| 2,500,000 | 90 | 90 | 1 | 1.2863919 | 1.5891909 | 2 | N/A |

Table 42. Single Depth traversal of R-Forest Gaussian data

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 4 | 4 | 1 | 1.3865899 | 1.7886287 | 2 | N/A |
| 1,000,000 | 4 | 4 | 1.004998 | 1.4302894 | 1.9532247 | 0 | N/A |
| 1,500,000 | 5 | 5 | 1 | 1.4401352 | 2.1044584 | 1 | N/A |
| 2,000,000 | 5 | 5 | 1.051041 | 1.4638637 | 1.9889443 | 0 | N/A |
| 2,500,000 | 5 | 5 | 1 | 1.4681421 | 1.9538078 | 1 | N/A |

Table 43. R = 90 applied to R-Forest Gaussian data

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 90 | 90 | 1 | 1.1371187 | 1.4757850 | 54 | 2.6921 – 3.9982 |
| 1,000,000 | 90 | 90 | 1 | 1.1654481 | 1.7502172 | 31 | 2.1401 – 3.9978 |
| 1,500,000 | 90 | 90 | 1 | 1.1832530 | 1.5361650 | 28 | 2.1470 – 3.9996 |
| 2,000,000 | 90 | 90 | 1 | 1.1998810 | 1.5577043 | 12 | 2.3430 – 3.9888 |
| 2,500,000 | 90 | 90 | 1 | 1.2074802 | 1.5579109 | 13 | 1.9551 – 3.9958 |

Table 44. Access 1.79% applied to R-Forest Gaussian data

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 473 | 478 | 1 | 1.0721158 | 1.3767850 | 191 | 2.7728 – 3.9999 |
| 1,000,000 | 945 | 972 | 1 | 1.0694699 | 1.5583096 | 194 | 2.5511 – 3.9999 |
| 1,500,000 | 1,420 | 1,463 | 1 | 1.0636808 | 1.3288630 | 218 | 3.0089 – 3.9999 |
| 2,000,000 | 1,897 | 1,952 | 1 | 1.0672356 | 1.3288760 | 218 | 2.9999 – 3.9999 |
| 2,500,000 | 2,371 | 2,443 | 1 | 1.0670222 | 1.3065228 | 195 | 3.0585 – 4.0000 |

Table 45. LSB-Forest with 20 trees Gaussian data

| No. Data Points | Average Page Count | Max Page Count | Min. Dist. Ratio | Average Dist. Ratio | Max. Dist. Ratio | Correct NN found | Low Bound Ratio |
|---|---|---|---|---|---|---|---|
| 500,000 | 82 | 87 | 1 | 1.4327680 | 2.0359090 | 1 | N/A |
| 1,000,000 | 102 | 106 | 1.00001 | 1.4630430 | 2.0486570 | 0 | N/A |
| 1,500,000 | 102 | 106 | 1 | 1.4784300 | 2.2600310 | 1 | N/A |
| 2,000,000 | 102 | 107 | 1 | 1.4044370 | 2.0268500 | 2 | N/A |
| 2,500,000 | 102 | 107 | 1 | 1.4650130 | 2.0429830 | 2 | N/A |

In figure 96 I have plotted distance ratio on the y-axis against the Gaussian data points sets for five different tests as listed in tables 41 through 45.

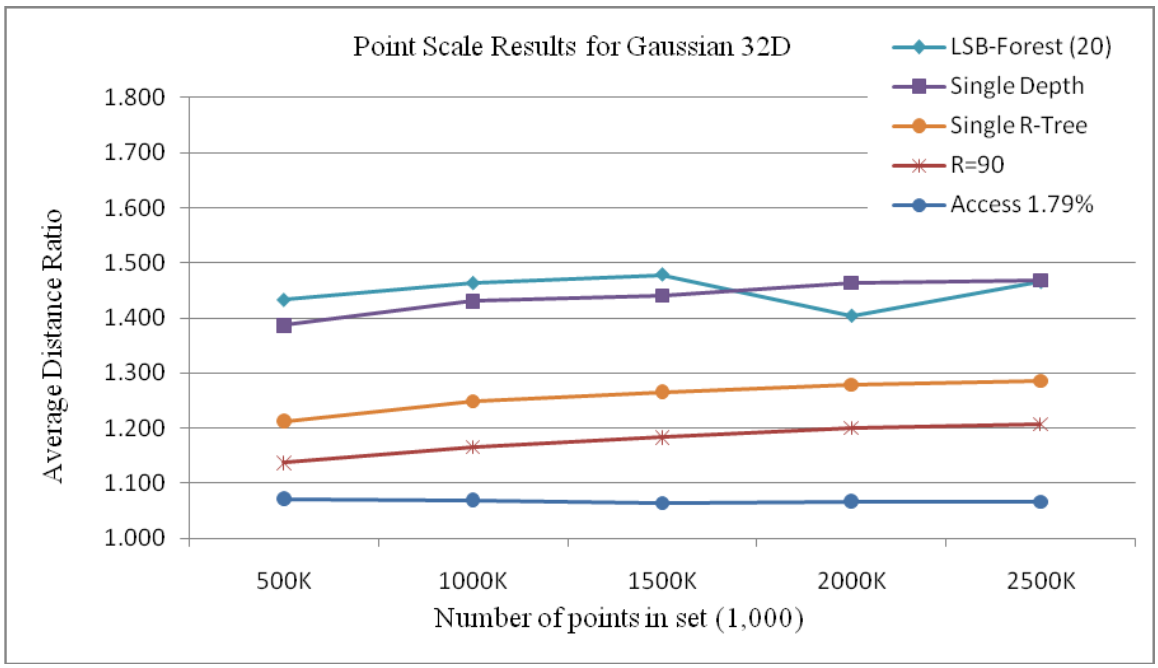



Figure 96. Results of all five experiments for Gaussian data with increasing number of points

For Gausian we observer simular results as seen in the clusterd results, Single Depth R-Forest performs better in 3 out of 5 data sets, with the exceptions being 2,000,000 and 2,500,000 points sets where it did worst. As for Single R-Tree and R-Forest, I did consistantly better then LSB-Forest.

#### 7.4.2.2.4 Discussion of Scale Results

There are several conclusions I can draw from the point scale results, for Single R-Tree, R-Forest and LSB-Forest.

- All scale well and do not degrade in terms of distance ratio as the number of points increases and with different data distributions.
- LSB-Forest performs worst in clustered and Gaussian point scale tests, and in most cases worst then Single Depth R-Forest (which is the minimum amount of

work possible). Only in the case of uniform data did Single Depth perform worst then LSB-Forest.

- Consider the point scale experiments R-Forest with access restriction R = 90 and 1.79% access restriction (tables 31, 32, 37, 38, 43, 44), all perform within the distance ratio as described by LSB-Forest paper [111]. Of course I cannot claim a guarantee on the bound, but looking at the tables of 30,000 tests all were within the bounds offered by paper.
- Comparing R-Forest to LSB-Forest we can see that for the same amount of work in terms of page access, even as the number of points increase, R-Forest continues to outperform LSB-Forest.
- In almost all cases of R-Forest Access restricted to 1.79% the average page count is less than the maximum page count, This means there were several cases where K-Factor or *Mindist (Q, MBR)* < *CB* terminated the search by pruning the remaining R-Trees in the forest.

## 7.5 Unrestricted Access

On last set of experiments I will present involves removal of any restriction on the amount of IO, allowing R-Forest to run with only K-Factor and Mindist pruning. This set of experiments was run to show the affect use of K-Factor and disjoint forest on large data set with increasing number of points. Using the large uniform, clustered and Gaussian sets (500,000 to 2,500,000 points), I ran all 15 tests with unrestricted access, pages size 4KB and K-Factor equal 4.0. These results can be seen in tables 46 to 48 and figures 97 to 99, it should be noted no comparison to LSB-Forest is given as search

termination is not controllable feature, without modifying their source code. In tables 46 to 48 I listed the maximum and average distance ratio, number of correct NN returned, average number of pages accessed, maximum number of pages accessed, finally average, median and maximum percentage of the forest accessed. In figures 97 to 99 I plotted the maximum, average and median percentage of the forest accessed.

Table 46. Results for uniform data unlimited access with increasing number of points

| **Uniformed** | **500K** | **1,000K** | **1,500K** | **2,000K** | **2,500K** |
|---|---|---|---|---|---|
| **Max. Dist. Ratio** | 1.278936 | 1.200821 | 1.24518 | 1.19897 | 1.2686 |
| **Average Dist. Ratio** | 1.015265 | 1.014384 | 1.018435 | 1.014587 | 1.015364 |
| **Min. Dist. Ratio** | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| **No. Correct** | 651 | 652 | 650 | 679 | 671 |
| **Aver. Page Access** | 2,165 | 3,923 | 5,668 | 7,039 | 8,396 |
| **Max Page access** | 5,948 | 9,824 | 14,964 | 17,377 | 20,793 |
| **Max. % Accessed** | 23.9% | 19.7% | 20.1% | 17.5% | 16.7% |
| **Aver. % Accessed** | 8.7% | 7.88% | 7.6% | 7.07% | 6.73% |
| **Median % Accessed** | 8.0% | 7.85% | 7.1% | 6.5% | 6.1% |

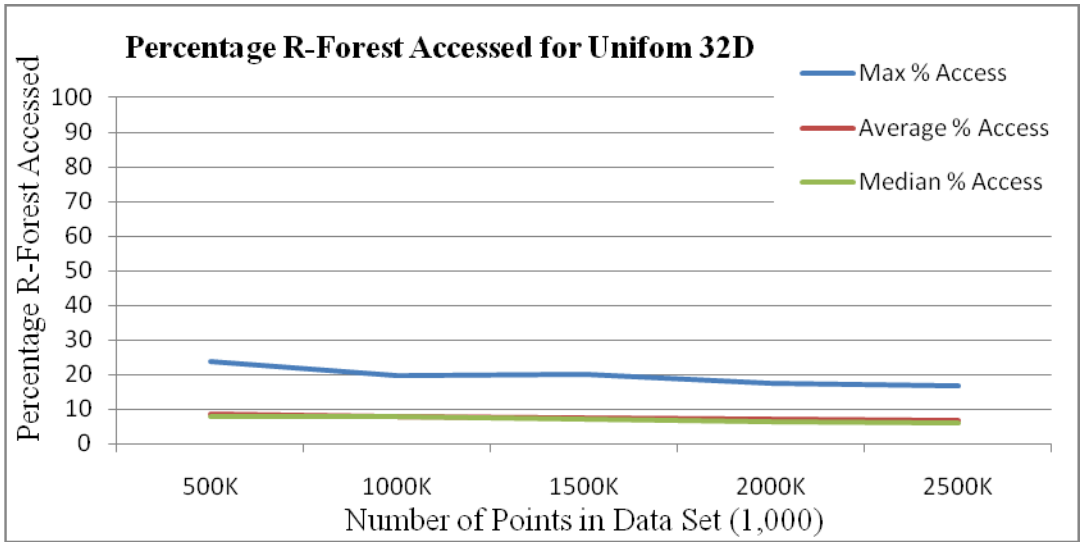



Figure 97. Plot of maximum, average and median percent of R-Forest pages accessed for uniform data

In table 46 the first item to point out are the number of test where R-Forest returned the correct NN, here 651 to 679 test points (65%- 67%). Of the remaining points the maximum and average distance ratio, were better than any of the previous test run on these data. These results are to be expected as there was no hard termination on the number of pages accessed allowing more IO to occur. Also included in the table are the results for maximum, average and median percentage of R-Forest accessed. It is here that we can see the usefulness of disjoint forest and K-Factor. First the maximum amount of the forest accessed was below 24% and actually declined as the number of points increased. Second the average and median was between 8.7% and 6.1% which also declined as slightly as the number of points increased (this can be seen in figure 97). Most likely the slight decline in percentage of access was caused by the increased in the number of points there by increasing the density of the data space.

In table 47 and figure 98 I have presented the results for the same test using clustered data sets. Here as in the previous test the results are very good with 88% to 91% of the test returning a correct NN and the maximum amount of forest accessed between 4.2% and 3.9% with the average and median ranging between, 3.1% to 2.58%.

Table 47. Results for clustered data unlimited access with increasing number of points

| **Clustered** | **500K** | **1,000K** | **1,500K** | **2,000K** | **2,500K** |
|---|---|---|---|---|---|
| **Max. Dist. Ratio** | 1.135895 | 1.98878 | 1.258626 | 1.183506 | 1.229128 |
| **Average Dist. Ratio** | 1.003653 | 1.00326 | 1.005055 | 1.003856 | 1.004323 |
| **Min. Dist. Ratio** | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| **No. Correct** | 898 | 900 | 887 | 898 | 910 |
| **Aver. Page Access** | 748 | 1,347 | 2,026 | 2,570 | 3,321 |
| **Max Page access** | 1,051 | 2,026 | 3,007 | 3,969 | 4,832 |
| **Max. % Accessed** | 4.2% | 4.1% | 4.0% | 4.0% | 3.9% |
| **Aver. % Accessed** | 2.99% | 2.71% | 2.71% | 2.58% | 2.67% |
| **Median % Accessed** | 3.1% | 2.8% | 2.9% | 2.7% | 2.8% |

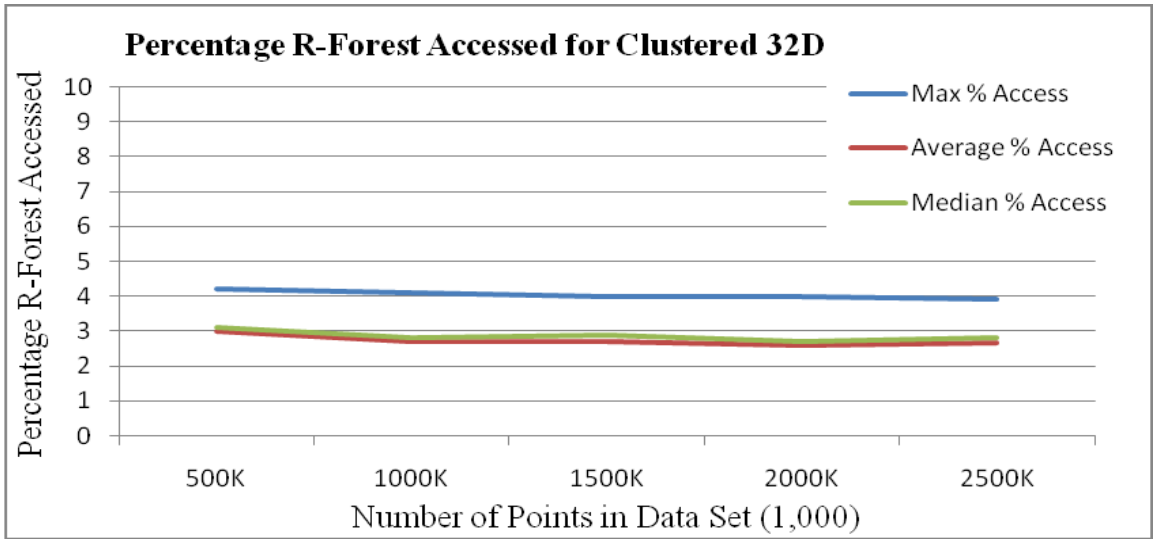



Figure 98. Plot of maximum, average median percent of R-Forest pages accessed for clustered data (note scale change to 0 to 10%)

For the clustered data set again I have a slight decline in all three maximum, average and median percentage of access, as the number of points increases, as seen in table 47 and figure 98 (note the scale on the y-axis is 0% to 10%).

Finally, for my Gaussian data sets I have presented my results in table 48 and figure 99. First we see that for Gaussian data only about 50% of the test points returned the exact NN, yet the average distance ratio of the ANN returned was lower than any other test run on these data sets. Again, this is expected as I am allowing the R-Forest to run without any termination condition on the number of pages accessed. With this set of tests, I did have higher level of access up to 44.6% with the maximum holding between 44.6% to 32.3% and the average between 9.34% and 7.38%. Because of the higher maximum access I decided to include the median percentage of forest accessed to show that higher percentages were not as bad as one would think.

Table. 48 Results for Gaussian data unlimited access with increasing number of points

| Gaussian | 500K | 1,000K | 1,500K | 2,000K | 2,500K |
|---|---|---|---|---|---|
| **Max. Dist. Ratio** | 1.374841 | 1.234803 | 1.277911 | 1.320243 | 1.23646 |
| **Average Dist. Ratio** | 1.02502 | 1.024198 | 1.025251 | 1.02579 | 1.02599 |
| **Min. Dist. Ratio** | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| **No. Correct** | 518 | 536 | 500 | 512 | 507 |
| **Aver. Page Access** | 2,238 | 3,987 | 5,633 | 7,033 | 8,811 |
| **Max Page access** | 10,684 | 17,466 | 24,758 | 40,049 | 38,465 |
| **Max. % Accessed** | 44.6% | 36.7% | 34.6% | 42.1% | 32.3% |
| **Aver. % Accessed** | 9.34% | 8.37% | 7.86% | 7.38% | 7.40% |
| **Median % Accessed** | 7.85% | 7.1% | 6.5% | 5.8% | 6.1% |

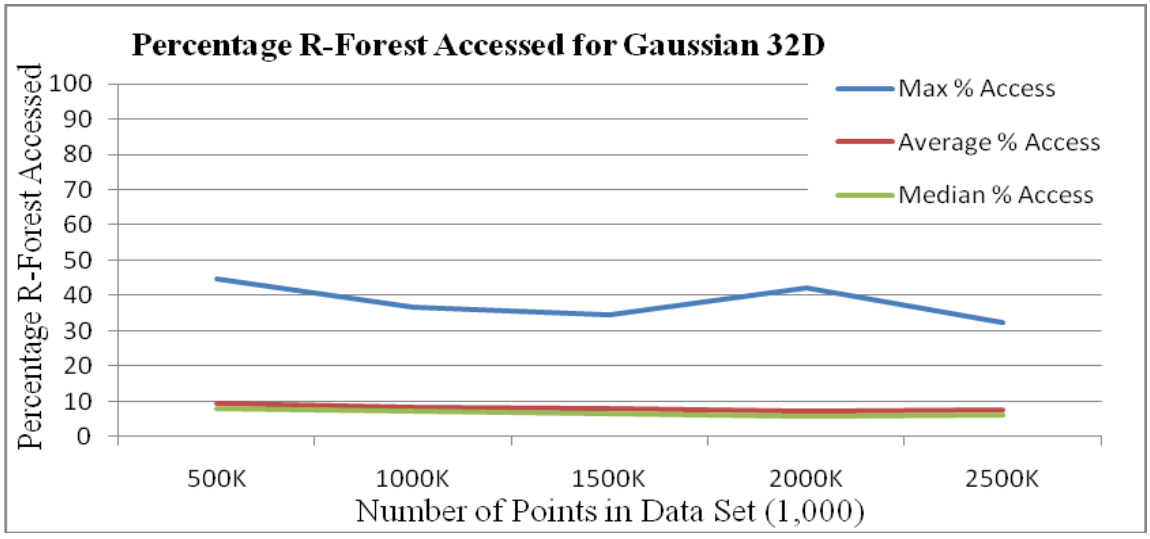



Figure 99. Plot of maximum, average median percent of R-Forest pages accessed for Gaussian data

Looking at the average percentage of access in table 48 and figure 99 one can see that my average is well below the maximum for the same data set. Also the median percent of access is 1% to 2% less than the average. Because of these results I looked at the details of each set of test results and found that in every set I had 40 – 50 points that exceed 23% of total forest accessed with the maximum at 44.6%. What these results

show is that at first the Gaussian results do not look as good as the other data distribution results, but in reality 95% of the test accessed 23% or less of the forest, meaning that I only had around 5% that were not good as compared to my previous results.

The one main conclusion that can be drawn from these three sets of experiments is that the use of disjoint sets and K-Factor was able to limit IO of the Forest, without degrading the quality of the results. Granted R-Forest still accessed up to 23% of the pages in 95% of all of the tests (5% in Gaussian where the exception), but when compared to other pruning methods for single R-Tree (presented in chapter 5), I got much better results in terms of distance ratio.

## 7.6 Conclusion

In conclusion my use of R-Forest ANN performed very well. Using a set of disjointed R-Trees over a given data set with some additional parameters I was able to achieve results better than current methods (LSB-Forest). Testing shows that subspace partitioning on a few dimensions helps to elevate some of the problems encountered by a single R-Tree index. Not only that, but my method allowed for the implementation of a new pruning parameter (K-Factor) that improved my results, even when access is restricted equal to that of other methods (LSB-Forest). Results presented show that I returned better or equal (in terms of distance ratio) ANN with the same amount of work (IO) as that of LSB-Forest.

I have run experiments over different data distributions, increasing dimensionality, and number of points. These experiments show that there is no dependence on any domain knowledge, meaning that parameters do not require any

adjusting to achieve better results with different data distributions, number of dimensions or number of points. As the number of dimensions increases, I do not see any degradation of distance ratio, even as I perform the same amount of IO (work). Finally, as the size of the data sets increase I am able to achieve reasonable results again without much lose in the quality of the results returned. From these results one can see that in practice the R-Forest is a better choice than the LSB-Forest, and one can possibly argue the "lost" of guarantee bound is really not a significant issue. All of the results presented in the chapter support the use of R-Forest as solution for approximate nearest neighbor queries in high dimensional space.

## 8. Future Work

The work presented in my dissertation has led me to believe that there are additional avenues for improvement that need to be explored as well as future work in the lower bound data that can be returned after a ANN query using R-Forest.

1. The addition of the MBR-Center point has opened up the possibility of some additional pruning heuristics, such as storing the MBR-Center point's minimum and maximum distance to all points in a leaf node and average distance between points in the leaf. It might be possible to use these to define additional pruning heuristics or to attempt to predict which node has a better chance of improving the current best ANN found.
2. Revisit the R-Tree insertion algorithms now that I'm using disjoint sets and have the MBR-Center point for internal nodes could it be used to determine an insertion path for new points. Rather than using the standard R-Tree algorithm

which attempts to minimize MBR growth why not attempt to insert closest to the MBR-Center point. In my implementation the MBR-Center point is an actual point, so my thoughts are that insertion closer to the MBR-Center might improve the R-Trees in R-Forest in terms of query efficiency.

3. I have done some work toward using K-Factor as a terminating condition, as it might be possible for it to provide a pseudo guarantee on the bounds of the results returned. The work I have done was not fully tested and needs more research.
4. The feedback that is now possible because of the disjointed sets needs to be researched further. At this time I'm only providing the minimum distance to MBR pruned by K-Factor which is an estimate on the lower bound of the space not searched. It is also possible to provide feedback on entire R-Trees that were not searched, the question here is how accurate are these and how useful. This area needs further research.

## References


[1] Adler, David.W. IBMDB2 Spatial Extender - Spatial data within the RDBMS. In: Proceedings of VLDB Conference, 2001, pp. 687–690.

[2] Altschul S. F., Gish W., Miller W., Myers E. W., Lipman D. J., A Basic Local Alignment Search Tool, Journal of Molecular Biology, Vol. 215, no. 3, 1990, pp. 403-410.

[3] A. Andoni and P. Indyk. New LSH-based algorithm for approximate nearest neighbor. Technical Report MIT-CSAIL-TR-2005-073, MIT, Cambridge, MA, December 2005.

[4] A. Andoni, P. Indyk, Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions, FOCS, 2006, pp.117-122.

[5] Aoki, P.M. Generalizing Search in Generalized Search Trees. in The 14th IEEE International Conference on Data Engineering. Orlando, FL. 1998, pp 380-389.

[6] Arge, L., et al. Efficient bulk operations on dynamic R-trees. in 1st Workshop on Algorithm Engineering and Experimentation. Baltimore, MD. 1999, pp 328-348.

[7] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, 1994, pp. 573–582,

[8] S. Arya and D. M. Mount. Approximate Range Searching. In 11th Annual Symposium on Computation Geometry, ACM Press, June 1995, pp 172–181.

[9] Beckmann, n., Begel H. P., Schneider R., Seeger B., Informatlk P.,The R*-tree: An Efficient and Robust Access Method for Points and Rectangles+. ACM-SIGMOD Conference on Management of Data. Atlantic City, NJ, 1990, pp 322-331.

[10] Norbert Beckmann, Bernhard Seeger: A revised R*-Tree in comparison with related index structures. SIGMOD Conference, 2009. pp 799-812.

[11] Bently Jon Louis, Multidimensional Binary Search Trees Used for Associative Searching. Communications of the ACM, Vol. 18 no 9, 1975, pp 509-519.

[12] Benetis, R., et al. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. in 7th International Database Engineering and Application Symposium. Edmonton, Canada. 2002, pp 229-249.

[13] M. Bern. Approximate closest-point queries in high dimensions. Inform. Process. Letters, 1993, pp 45:95.

[14] S. Berchtold, C. B¨ohm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space, In Proceedings of the 16th ACM SIGACTSIGMOD-SIGART Symposium on Principles of Database Systems (PODS'97), Tucson, AZ, May 1997, pp 78–86.

[15] Bohm, C., S. Berchtold, and D.A. Keim, Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Computing Surveys, Vol 33 no 3, 2001, pp 322-373.

[16] D. Borthakur. The Hadoop Distributed File System: Architecture and Design. http://hadoop.apache.org/core/docs/current/hdfs_design.pdf.

[17] Brechtold, S., D.A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. in The 22nd International Conference of Very Large Databases. Mumbai, India, 1996, pp 28-39.

[18] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. Communications of the ACM, Vol 16 no 4, 1973, pp 230–236.

[19] Michael A. Casey, Malcolm Slaney: Song Intersection by Approximate Nearest Neighbor Search. ISMIR, 2006, pp. 144-149.

[20] Guang-Ho Cha, Xiaoming Zhu, P. Petkovic, Chin-Wan Chung, An efficient indexing method for nearest neighbor searches in high-dirnensional image databases. IEEE Transactions on Multimedia Vol 4 no 1, 2002, pp 76-87.

[21] Yong-Sheng Chen, Yi-Ping Hung, and Chiou-Shann Fuh Fast algorithm for nearest neighbor search based on a lower bound tree, Proceedings of the 8th International Conference on Computer Vision, Vancouver, Canada, vol. 1, July 2001, pp. 446-453.

[22] Choubey, R., L. Chen, and E.A. Rundensteiner. GBI: A Generalized R-Tree Bulk-Insertion Strategy. in 6th International Symposium on Large Spatial Databases. Hong Kong, 1999, pp 91-108.

[23] Tat-Seng Chua , Jinhui Tang , Richang Hong , Haojie Li , Zhiping Luo , Yantao Zheng, NUS-WIDE: a real-world web image database from

National University of Singapore, Proceedings of the ACM International Conference on Image and Video Retrieval, Greece, 2009, article no 48.

[24] Chun J., Okada Y., Tokuyama T.: Distance Trisector of a segment and a point. Interdiscip. Inf. Sci. vol 16 no 1, 2010, pp 119–125.

[25] P. Ciaccia, and M. Patella. Bulk loading the M-tree. in 9th Australasian Database Conference. Perth, Australia, 1998, pp 15-26.

[26] P. Ciaccia, M. Patella: PAC Nearest Neighbor Queries: Using the Distance Distribution for Searching in High-Dimensional Metric Spaces. Proc. 7th SEBD, 1999, pp 259-273.

[27] E. Cohen. M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman. C. Yang. Finding Interesting Associations without Support Pruning. Proceedings of the 16th International Conference on Data Engineering, 2000, p 489.

[28] Corral, Antonio, Joaquin Cañadas, and Michael Vassilakopoulos. Improvements of Approximate Algorithms for Distance-Based Queries. HDMS, 2002, pp 83-102.

[29] Cui, B., Shen, H.T., Shen, J., Tan, K.L., Exploring Bit-Difference for Approximate KNN Search in High-dimensional Databases. Proc. 16th Australian Database Conference, 2005, pp 165-174.

[30] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. Proceedings of the ACM Symposium on Computational Geometry, 2004, pp 253-262.

[31] Christian Digout, Mario A. Nascimento, High-Dimensional Similarity Searches Using A Metric Pseudo-Grid. ICDE Workshops 2005, p 1174.

[32] Matthijs Douze, Herve Jegou, Cordelia Schmid, An Image-Based Approach to Video Copy Detection With Spatio-Temporal Post-Filtering, Multimedia, IEEE Transactions On, Vol 12, no 4, 2010, pp 257 – 266.

[33] Eduardo Valle, Matthieu Cord, and Sylvie Philipp-Foliguet. High-dimensional descriptor indexing for large multimedia databases. Proceeding of the 17th ACM conference on Information and knowledge management, 2008, pp 739-748.

[34] Faloutsos C., Barber R., Flickner M., Hafner J., et al.: Efficient and Effective Querying by Image Content, Journal of Intelligent Information Systems, Vol. 3, 1994, pp. 231-262.

[35] Hakan Ferhatosmanoglu, Ertem Tuncel, Divyakant Agrawal, Amr El Abbadi. Vector Approximation based indexing for non-uniform high dimensional data sets. In Proceedings of the 9th ACM International Conference on Information and Knowledge Management, McLen, VA, USA, 2000, pp 202-209.

[36] R. S. Filho, A. Traina, C. T. Jr., and C. Faloutsos. Similarity search without tears: The OMNI family of all-purpose access methods. In Proc. of the 17th IEEE Intl. Conf. on Data Engineering, 2001, pp 623–630.

[37] Junhao Gan, Jianlin Feng, Qiong Fang, Wilferd Ng. 2012. Locality-Sensitive Hashing Scheme Based on Dynamic Collision Counting. SIGMOD12, May 2012, pp. 541-552.

[38] Michael Greenspan, Mike Yurick, Approximate K-D tree search for efficient ICP, Proc. 4th Int. Conf. 3-D Digital Imaging Model, 2003, pp.442 - 448.

[39] S. Har-Peled. A replacement for voronoi diagrams of near linear size. Annual Symposium on Foundations of Computer Science, 2001, pp 94-103.

[40] Hellerstein, J., J. Naughton, and A. Pfeifer. Generalized search trees for database systems. in The 21st International Conference on Very Large Databases. Zurich, 1995, pp 562-573.

[41] A. Hinneburg, C.C. Aggarwal, and D.A. Keim, What Is the Nearest Neighbor in High Dimensional Spaces? Proc. Very Large Databases Conference, 2000, pp. 506-515.

[42] Hjaltason, G. and H. Samet, Distance Browsing in Spatial Databases. ACM Transactions on Database Systems, Vol 24 no 2, 1999, pp 265-318.

[43] Gionis, A., P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. in 25th International Conference on Very Large Data Bases. San Francisco, CA, USA, 1999, pp 518-529.

[44] Guttman, A. R-trees: A dynamic index structure for spatial searching. ACM-SIGMOD Conference on Management of Data. 1984, Vol 14, no 2, 1984, pp 47 – 57.

[45] IBM Informix R-tree Index, User's Guide, 2005. IBM Informix Dynamic Server v10 Information Center, Feb. 2009.

[46] K. Imai, A. Kawamura, J. Matoušek, Y. Muramatsu and T. Tokuyama, Distance ksectors and zone diagrams, Extended abstract in EuroCG, 2009, pp. 191–194.

[47] P. Indyk, G. Iyengar, N. Shivakumar. Finding pirated video sequences on the Internet. Technical Report, Computer Science Department, Stanford University. 1999.

[48] P. Indyk and R. Motwani. Approximate nearest neighbor: towards removing the curse of dimensionality. Proceedings of the Symposium on Theory of Computing, 1998, pp 604-613.

[49] Venkata Narasimha Inukollu, Sailaja Arsi and Srinivasa Rao Ravuri, High Level View of Cloud Security: Issues and Solutions, International Journal of Computer Science & Information Technolo, Vol. 6 Issue 2, 2014, pp 51-61.

[50] Kalos M. H., Whitlock P. A, Monte Carlo Methods, Wiley, New York, 1986.

[51] K. Kanth, S. Ravada, J. Sharma, J. Banerjee: Indexing Medium-dimensionality Data in Oracle. SIGMOD Conference, 1999, pp 521-522.

[52] Kothuri Venkata Ravi Kanth, Siva Ravada, Daniel Abugov: Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. SIGMOD Conference, 2002, pp 546-557.

[53] Katayama, N. and S. Satoh. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. in The 1997 ACM SIGMOD International Conference on Management of Data, 1997, pp 369-380.

[54] Katayama, Norio, and Shin'ichi Satoh. Distinctiveness-sensitive nearest-neighbor search for efficient similarity retrieval of multimedia information. Data Engineering, 2001. Proceedings. 17th International Conference on. IEEE, 2001, pp 493–502.

[55] Knowledge Discovery and Data KDD Cup 2004 biology dataset (Bio_train.dat). URL - http://osmot.cs.cornell.edu/kddcup/datasets.html Last Accessed 3-29-2014.

[56] J. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, 1997, pp 599-608

[57] D.E. Knuth, The Art of Computer Programming Vol. 3: Sorting and Searching, 1968, pp 471-480.

[58] N. Koudas, B. Ooi, K. Tan, and R. Zhang, Approximate NN Queries on Streams with Guaranteed Error/Performance Bounds, Proc. Int'l Conf. Very Large Data Bases, VLDB, 2004, pp 804-815.

[59] Korn, F. and S. Muthurksirhnan. Influence sets based on reverse nearest neighbor queries. in ACM SIGMOD International Conference on Management of Data, 2000, pp 201-212.

[60] M. Krátký, V. Snášel, J. Pokorný, P. Zezula, and T. Skopal,: Efficient Processing of Narrow Range Queries in the R-Tree. Technical Report ARG-TR-01-2004, Department of Computer Science, VŠB-Technical University of Ostrava, Czech Republic, 2004,

[61] R. Krauthgamer and J. R. Lee. Navigating nets: Simple algorithms for proximity search. Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, 2004, pp 798-807.

[62] R. Kurniawati, J. S. Jin, J. A. Shepard, SS+-tree: An improved index structure for similarity searches in a high-dimensional feature space, Proc. Conf. Storage and Retrieval for Image and Video Databases V, vol. 3022, 1997, pp.110–120.

[63] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. Proceedings of the Thirtieth ACM Symposium on Theory of Computing, 1998, pp 614–623.

[64] F. Laviolette, M. Marchand, and M. Shah. A PAC-Bayes approach to the set covering machine, Advances in Neural Information Processing Systems (NIPS), Vol 18, 2005, pp 481-488.

[65] King-Ip Lin, Mike Nolen, Congjun Yang, Applying Bulk Insertion Techniques for Dynamic Reverse Nearest Neighbor Problems, In Proceedings of the 8th International Database Engineering and Applications Symposium, July 2003, pp 290-297.

[66] King-Ip Lin, Mike Nolen, Koteswarea Kommeneni, Utilizing indexes for approximate and nearest neighbor queries, The 10th International Database Engineering and Applications Symposium, 2005, pp 83–88.

[67] Q. Lv, W. Josephson, Z. Wang, M. Charikar, K. Li, Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search, VLDB, 2007. pp 950-961.

[68] Ting Liu, Andrew W. Moore, Alexander G. Gray, Ke Yang: An Investigation of Practical Approximate Nearest Neighbor Algorithms, Neural Information Processing Systems, 2004, pp 825-832.

[69] Mehrotra R., Gary J.: Feature-Based Retrieval of Similar Shapes, Proceedings. 9th International Conference on Data Engineering, 1993, pp 108-115.

[70] Mehdi Sharifzadeh and Cyrus Shahabi. 2010. VoR-tree: R-trees with Voronoi diagrams for efficient processing of spatial nearest neighbor queries. Proceedings of VLDB Endow, September 2010, pp 1231-1242.

[71] Matthew L. Miller, Manuel Acevedo Rodriguez, Ingemar J. Cox, Audio fingerprinting: nearest neighbor search in high dimensional binary spaces. IEEE Workshop on Multimedia Signal Processing, IEEE Signal Processing Society, 2002, pp 182-185.

[72] Hiroshi Murase , Shree K. Nayar, Visual learning and recognition of 3-D objects from appearance, International Journal of Computer Vision, Vol 14 no 1, 1995, pp 5-24.

[73] MySQL 5.0 Reference Manual, http://downloads.mysql.com/docs/refman-5.0-en.a4.pdf, 2008.

[74] D. Nist´er and H. Stew´enius. Scalable recognition with a vocabulary tree. In Proc. CVPR, 2006, pp 2161-2168.

[75] Michael Nolen, King-Ip Lin Approximate high-dimensional nearest neighbor queries using R-Forest, Proceedings of the 17th International Database Engineering & Applications Symposium, October 2013, pp. 48-57.

[76] NUS_WIDE Research data sets URL - http://lms.comp.nus.edu.sg/research/NUS-WIDE.htm, Last Access 03-29-2014.

[77] A. Papadopoulos and Y. Manolopoulos, Performance of Nearest Neighbor Queries in R-Trees, Proceedings of the 6th International Conference Database Theory, 1997. pp 394-408.

[78] Rina Panigrahy, Entropy based nearest neighbor search in high dimensions. SODA, 2006, pp 1186-1195.

[79] Pentland, A., Picard, R.W., Scalroff, S. Photobook: Tools for Content Based Manipulation of Image Databases. In SPIE, Vol. 2185 1994, pp 34-47.

[80] James Philbin, Ondˇrej Chum, Michael Isard, Josef Sivic and Andrew Zisserman, Object Retrieval with Large Vocabularies and Fast Spatial Matching, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2007, pp 1–8.

[81] The PostgreSQL Comprehensive Manual, http://www.postgresql.org/docs/manuals/).

[82] Andrew Rabinovich, Andrea Vedaldi, Carolina Galleguillos, Eric Wiewiora and Serge Belongie Objects in Context. International Conference on Computer Vision, Rio de Janeiro, Brazil. 2007.

[83] C. P. Robert and G. Casella. Monte Carlo Statistical Methods. Springer-Verlag New York, Inc., 2005.

[84] Roussopoulos, N., S. Kelley, and F. Vincent. Nearest Neighbor Queries. ACM SIGMOD International Conference on Management of Data, 1995, Vol. 24, no. 2, pp 71-79.

[85] Sakurai, Y., Yoshikawa, M., Uemura, S., & Kojima, H. (2000, September). The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation. In VLDBl, 2000, pp. 5-16.

[86] Venu Satuluri, Srinivasan Parthasarathy. Bayesian Locality Sensitive Hashing for Fast Similarity Search, Proc VLDB Endow, Vol 5 no. 5, 2012, pp. 430-441.

[87] Sellis, T., Roussopoulos, N., Faloutsos, C. The R+-tree: A Dynamic Index for Multidimensional Objects. Proceedings 13rd Int'l Conference on Very Large Data Bases. Brighton, England, 1987, pp 507–518.

[88] Shoichet B. K., Bodian D. L., Kuntz I. D. Molecular Docking Using Shape Descriptors, Journal of Computational Chemistry, Vol. 13, no. 3, 1992, pp. 380-397.

[89] A. Singh, H. Ferhatosmanoglu, and A.S. Tosun, High Dimensional Reverse Nearest Neighbor Queries, Proc. Conf. Information and Knowledge Management (CIKM), 2003, pp. 91-98.

[90] Josef Sivic, Andrew Zisserman. Video Google: A text retrieval approach to object matching in videos. In Proc. International Conference on Computer Vision, Oct 2003, pp 1470-1477.

[91] Josef Sivic, Andrew Zisserman, Efficient Visual Search for Objects in Videos, Proceedings of the IEEE, Volume: 96, Issue: 4, April 2008, pp 548–566.

[92] Stanoi, I., D. Agrawal, and A. El Abbadi. Reverse Nearest Neighbor Queries for Dynamic Databases. in ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, 2000, pp 44-53.

[93] Stanoi, I., et al. Discovery of Influence Sets in Frequently Updated Databases. in Proceedings of the International Conference on Very Large Databases, 2001, pp 99-108.

[94] Yufei Tao , Dimitris Papadias , Xiang Lian, Reverse kNN search in arbitrary dimensionality, Proceedings of the Thirtieth international conference on Very large data bases, 2004, pp.744-755.

[95] Yufei Tao , Man Lung Yiu , Nikos Mamoulis, Reverse Nearest Neighbor Search in Metric Spaces, IEEE Transactions on Knowledge and Data Engineering, v.18 n.9, September 2006, pp 1239-1252.

[96] E. Tuncle, H. Ferhatosmanoglu, and K. Rose, VQ-Index: An Index Structure for Similarity Searching in Multimedia Databases, ACM Multimedia, 2002, pp. 543-553.

[97] The UCI KDD Archive Information and Computer Science University of California, Irvine, URL - http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.html Last Access 07-18-2005

[98] US Government geologic web site. URL - http://mapping.usgs.gov/www/gnis/ Last access 10-2-2002

[99] J. K. Uhlmann.,Satisfying General Proximity/ Similarity Queries with Metric Trees, Information Processing Letters, 1991, Vol 40, pp 175-179.

[100] Urruty, T., Belkouch, F., and Djeraba, C. Efficient Indexing for High Dimensional Data: Applications to a Video Search Tool, ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 2006.

[101] Kaushik Velusamy, Deepthi Venkitaramanan, Nivetha Vijayaraju, Greeshma Suresh and Divya Madhu, Inverted Indexing In Big Data Using Hadoop Multiple Node Cluster, International Journal of Advanced Computer Science and Applications, Vol. 4 No. 11, 2013, pp 156-161.

[102] Deepthi Venkitaramanan, Nivetha Vijayaraju Kaushik Velusamy, Greeshhma Suresh and Divya M, Comparison of Hadoop Multiple node Cluster Performance over Physical and Virtual Nodes Using Inverted Index Data Structure for Search over Wikipedia Data Set, International Journal of Applied Engineering Research, Vol 9, No 16, 2014 pp 3515-3531.

[103] Stephan Volmer, Fast Approximate Nearest-Neighbor Queries in Metric Feature Spaces by Buoy Indexing, Proceedings of the 5th International Conference on Visual Information Systems, 2002, pp 36-49.

[104] R. Webber, J. Schek, and S. Blott, A quantitative analysis and performance study for similarity-search methods in high-dimensional space, in Proceedings of the International Conference on Very Large Databases, August 1998, pp 194–205.

[105] J. Weinberger: SpatialWare Extends Microsoft SQL Server Capabilities. MapInfo Magazine, vol 6 no 2, 2001.

[106] White, D.A. and R. Jain. Similarity Indexing with the SStree. in The 12th International Conference on Data Engineering, New Orleans, 1996, pp 516-523.

[107] White, D.A. and R. Jain. Similarity Indexing: Algorithms and Performance. in Proceedings SPIE, Vol. 2670, San Diego, 1996, pp 62-73.

[108] W. Wu , X. Cheng , M. Ding , K. Xing , F. Liu and P. Deng, Localized Outlying and Boundary Data Detection in Sensor Networks, IEEE Trans. Knowl. Data Eng., vol. 19, no. 8, 2007, pp.1145 -1157.

[109] Yang, C. and K.I. Lin. An index structure for efficient reverse nearest neighbor queries. in IEEE International Conference on Data Engineering. Heidelberg, Germany, 2001, pp 485-492.

[110] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In Proceedings of the 5th Symposium on Combinatorial Pattern Matching, 1994, pp 198–212.

[111] Yufei Tao, Ke Yi, Cheng Sheng, Panos Kalnis: Quality and efficiency in high dimensional nearest neighbor search. SIGMOD Conference, 2009, pp 563-576.

[112] Yufei Tao, Ke YI, Cheng Sheng, Panos Kalnis, Efficient and Accurate Nearest Neighbor and Closest Pair Search in High Dimensional Space. ACM Transactions of Database Systems (TODS 2010), vol. 35, No. 3, Article 20, 2010, pp 1-46.

[113] Department of Computer Science and Engineering, The Chinese University of Hong Kong URL - http://www.cse.cuhk.edu.hk/~taoyf/paper/tods10-lsb.html, Last access 06-23-2011.

[114] Ziaiang. Yand, Wei Tsang. Ooi, Qibin. Sun, Hierarchical, Non-Uniform Locality Sensitive Hashing and Its Application to Video Identification, Proceedings International Conference. Multimedia and Expo, 2004. pp 743-746.

[115] Hao Zhang, Alexander C. Berg, Michael Maire, Jitendra Malik: SVM-KNN: Discriminative Nearest Neighbor Classification for Visual Category Recognition. CVPR 2006, pp 2126-2136.

[116] X.Zhou, G.Wang, J.X.Yu and G.Yu, M+-tree : A New Dynamical Multidimensional Index for Metric Spaces, Proceedings of 14th Australasian Database Conference. ADC 2003, pp 161-168.

[117] X. Zhou, G. Wang, X. Zhou and G. Yu, BM+-tree: A Hyperplane-based Index Method for High-Dimensional Metric Spaces, Proceedings of DASFAA 2005, pp 398-409.