EFFICIENT SIMILARITY SEARCH BASED ON DATA DISTRIBUTION PROPERTIES IN HIGH DIMENSIONS

By

Jinhua Li

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

2000
ABSTRACT

EFFICIENT SIMILARITY SEARCH BASED ON DATA DISTRIBUTION PROPERTIES IN HIGH DIMENSIONS

By

Jinhua Li

The characteristics of data distribution in high dimensional Euclidean space play a fundamental role in many areas of computer science, including database systems, pattern recognition, and multimedia. We have studied and formalized a few data distribution properties in high dimensional Euclidean space, such as the distance properties and the angle property. We have exploited these basic properties in developing efficient applications, such as the nearest neighbor queries in high dimensions.

Nearest Neighbor search is a fundamental task in many applications. At present state of the art approaches to nearest neighbor search are not efficient in high dimensions. In this dissertation we present an efficient Angle based Balanced index structure called the AB-tree, which uses heuristics to decide whether or not to access a node in the index tree based on the estimated angle and the weight of the node. Extensive experiments on synthetic data as well as real data demonstrate that the AB-tree outperforms other index structures such as the SS-tree by orders of magnitude while maintaining 90% true nearest neighbors.
To Hui and Draco.
ACKNOWLEDGEMENTS

The author wishes to think Dr. Sakti Pramanik for his great advisorship and management. I am grateful for many discussions and invaluable comments he provides.

I would like to thank my guidance committee members: Dr George C. Stockman, Dr. John J. Forsyth, and Dr. Ramanathapuram V Ramamoorthi for their help and guidance.
# TABLE OF CONTENTS

List of Tables viii

List of Figures ix

Introduction 1

Chapter 1: Related Work 6
  1.1 One-dimensional Access Methods 7
  1.2 Main Memory Access Structures 8
  1.3 Multimedia Indexing Methods 10
     1.3.1 Point Access Methods 11
     1.3.2 Spatial Access Methods 15
  1.4 Approximate Nearest Neighbor Search 23

Chapter 2: High Dimensional Data Distribution Properties 26
  2.1 Distance Properties 28
  2.2 Radius Properties 30
     2.2.1 Radius of the Minimum Bounding Hypersphere 31
     2.2.2 Nonuniform Distribution of Data Points inside the MBH 32
  2.3 Angle Property 32
  2.4 Interpretation of the Distribution of High Dimensional Data in Bound-
     ing Hyperspheres Based on the Properties 35
  2.5 Experimental Results with Cluster Data 36
  2.6 Related Research on Data Distribution Properties 38

Chapter 3: Angle based Balanced index tree: AB-tree 41
  3.1 Computing Better Bounding Hyperspheres 41
  3.2 Criteria for Node Access 42
     3.2.1 Impact of Angle Property 43
     3.2.2 Impact of Distance Property and Weight of Nodes 44
     3.2.3 Criteria Based on Multiple Threshold Angles 45
     3.2.4 Criteria Based on Distance 47
  3.3 Search Algorithm for AB-tree 48
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>Clustered AB-tree and Cached AB-tree</td>
<td>52</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Clustering Approach</td>
<td>52</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Implementation of Clustered AB-tree</td>
<td>54</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Cached AB-tree</td>
<td>57</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Implementation of Cached AB-tree</td>
<td>58</td>
</tr>
</tbody>
</table>

**Chapter 4: Parameter Estimation**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>61</td>
</tr>
<tr>
<td>4.2</td>
<td>Parameter Autotuning</td>
<td>63</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Complexity</td>
<td>65</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Algorithm for Parameter Autotuning</td>
<td>67</td>
</tr>
<tr>
<td>4.3</td>
<td>Tradeoff Between Performance and Accuracy</td>
<td>70</td>
</tr>
<tr>
<td>4.4</td>
<td>Stability of Parameters</td>
<td>72</td>
</tr>
<tr>
<td>4.5</td>
<td>Maintenance Cost</td>
<td>73</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Effect of Parameter Reuse on Maintenance Cost</td>
<td>75</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Maintenance Cost for Clustered AB-Tree and Cached AB-Tree</td>
<td>76</td>
</tr>
</tbody>
</table>

**Chapter 5: Experimental Results**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Synthetic Data</td>
<td>80</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Uniform Data</td>
<td>80</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Clustered Data</td>
<td>82</td>
</tr>
<tr>
<td>5.2</td>
<td>Performance Issues</td>
<td>83</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Effect of Database Size</td>
<td>83</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Effect of Dimension</td>
<td>84</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Effect of Number of Nearest Neighbors</td>
<td>85</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Effect of Block Size</td>
<td>86</td>
</tr>
<tr>
<td>5.3</td>
<td>Performance Gain of Clustered AB-tree</td>
<td>86</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Performance Sensitivity and the Number of Clusters</td>
<td>87</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Performance Comparison of Clustered AB-tree with Standard AB-tree, SS-tree and VA-file</td>
<td>88</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance Gain of Cached AB-tree</td>
<td>89</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Effect of Cache Size</td>
<td>89</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Performance Gain of Cached AB-tree Over AB-tree</td>
<td>91</td>
</tr>
<tr>
<td>5.5</td>
<td>Performance Comparison Using Real Data</td>
<td>92</td>
</tr>
<tr>
<td>5.6</td>
<td>Discussions on Approximate Nearest Neighbor Search</td>
<td>93</td>
</tr>
</tbody>
</table>

**Chapter 6: PicFinder: A Prototype Image Database System**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>97</td>
</tr>
<tr>
<td>6.2</td>
<td>The Design of the PicFinder System</td>
<td>99</td>
</tr>
<tr>
<td>6.3</td>
<td>The PicFinder Interface</td>
<td>102</td>
</tr>
</tbody>
</table>

**Chapter 7: Conclusions and Future Work**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>107</td>
</tr>
</tbody>
</table>
# List of Tables

4.1 Performance of AB-tree Using Different Number of Threshold Angles

4.2 Performance for Varying Database Sizes

4.3 Calibration Cost (Database Size: 800,000)

4.4 Effect of Initial Values of Threshold Angles

4.5 Detailed Calibration Cost Without Parameter Reuse

4.6 Detailed Calibration Cost With Parameter Reuse

5.1 Number of Node Accesses for Different Cache Level

5.2 Relative Error Bound in Distance for 90% Accuracy
## List of Figures

2.1 Effects of Dimension and Number of Points on $d_{max}/d_{min}$ .......................... 29
2.2 Probability Distribution Function of Distance for Dimension=100 ................ 30
2.3 Effects of Dimension and Number of Points on Radius of MBH .......................... 31
2.4 The Angle $\theta$ of Point P ........................................................................... 33
2.5 Distribution of d-Dimensional Data With Respect to the Angle $\Theta$ ................ 33
2.6 Probability of d-dimensional Data Points With Angle Less Than $\theta$ ............ 34
2.7 Distribution of High Dimensional Data in Bounding Hyperspheres ............... 35
2.8 Properties in High Dimensions ....................................................................... 37

3.1 Illustration of Angle Property .......................................................................... 43
3.2 Special Case: Undefined Intersecting Angle ..................................................... 47

4.1 Accuracy vs. Number of Node Accesses ............................................................ 71
4.2 Parameters For Varying Database Sizes (16 dimensions) ............................... 76

5.1 Uniform Data ..................................................................................................... 81
5.2 Clustered Data (database size: 800,000) .......................................................... 82
5.3 Effect of Database Size With Various Accuracies ............................................. 83
5.4 Clustered Data: Effect of Database Size (Dimension: 16) ............................... 84
5.5 Effect of Dimension .................................................. 84
5.6 Varying the Number of NN’s ........................................ 85
5.7 Effect of block size .................................................... 86
5.8 Performance of Clustered AB-tree for Varying Number of Subtrees .. 87
5.9 Performance Comparison for Different Number of Clusters .......... 88
5.10 Performance Comparison for Varying Dimensions ................. 89
5.11 Performance of Cached AB-tree for 800,000 32-dimensional Database 90
5.12 Performance of Cached AB-tree for 800,000 64-dimensional Database 91
5.13 Performance Gain of Cached AB-tree over AB-tree .............. 92
5.14 Real Data: 44,300 32-dimensional color histogram feature vectors .. 93
5.15 Result of Query Using Exact NN-search .......................... 95
5.16 Result of Query Using Approximate NN-search ................... 96

6.1 A Diagram of PicFinder .............................................. 100
6.2 The PicFinder Interface ............................................. 102
6.3 The QUERY window ................................................. 103
6.4 The HELP window .................................................. 104
6.5 The FEATURES and INDEXES window ............................ 105
6.6 The RESULTS window .............................................. 106
**Introduction**

Understanding the distribution of data points in high dimensional Euclidean space has been a focus of computer science research for years. Much of this research deals with the development of efficient algorithms that exploit the spatial relationships of data points in high dimensions. For example, in building index structures for databases with high dimensional data, bounding hyper-rectangles and bounding hyperspheres are often used. Various nearest neighbor search algorithms have been proposed in the past that try to minimize the overlap between these hyperspheres and/or the hyper-rectangles. However, the relationships between the amount of overlapping space and the number of data points inside the overlapping space needs to be understood for developing efficient algorithms for applications involving high dimensional data.

The focus of this research has been to study the properties of data distribution in high dimensional Euclidean space and exploit these properties for efficient application development. One of the targeted applications for this research is the nearest neighbor queries in high dimensional Euclidean space. Nearest neighbor queries are used for databases with non-traditional data such as image and video. With the rapidly increasing wealth of computational power available to developers, applications have begun to shift away from the storage and manipulation of only simple data types.
such as numbers and strings. Business, industry, and computer science research have moved toward applications that require very large databases that store and manipulate non-traditional data. One of the most common types of queries on these types of data is a similarity, or nearest neighbor (NN), query. The goal of this type of query is to retrieve an object that is most similar to a given object. One variation of a similarity query is a $K$ nearest neighbor ($K$NN) query. In this type of query, the system is asked to return the $K$ most similar objects to a given object.

In order to determine similarity between non-traditional data objects, like images and video, the objects must be represented in a manner that conveys some sort of semantics. Typically, this is done with high-dimensional feature vectors where the features are defined by a domain expert. For example, a feature vector for an image may represent the color contained in the image, the texture inherent in the image, or even objects present in the image.

Problem of implementing $NN$ search queries is that the notion of $NN$ in high dimensions become less meaningful for certain classes of data and the search complexity increases rapidly with dimensions. Beyer, et. al. [12] have identified a broad class of workload (in terms of data and query distributions) for which the difference in distance between the nearest neighbors and other points in the data set becomes negligible in high dimensions. For these classes of workload the notion of $NN$ is not very meaningful. However, there are classes of workload, such as data with a set of distinct clusters, where $NN$ may be meaningful in high dimensions.

There are two broad classes of indexing schemes, namely, those using space partitioning methods, such as KDB-tree [66], hB-tree [52], and lsdh-tree [35], and those
using data partitioning methods, such as R-trees [34, 5], SS-tree [79], SR-tree [42], M-tree [18], X-tree [11], and TV-tree [50]. All these indexing schemes do not perform better than linear scans [77] for exact NN search queries in high dimensional databases. A hybrid approach [16] has been proposed for indexing in high dimensional databases. It performs significantly better than previously proposed indexing schemes [52, 42] for feature based queries but performs worse than linear scan for NN search queries. Weber, et. al. [77] have shown that under the assumption of uniformity and independence, and when the number of dimensions is sufficiently high, a sequential scan may outperform indexing schemes for exact NN search queries. Consequently, they proposed VA-file based on approximations to make the sequential scan faster.

Another approach that has recently been used to avoid the curse of dimensionality is approximate nearest neighbor search [80]. An exact nearest neighbor search is not necessary and may be an overkill, since the features and the similarity measure themselves are chosen based on heuristic. In fact, in multimedia database systems, such as IBM’s QBIC [29] or MIT’s Photobook [60], the mapping of attributes of objects to coordinates of vectors is heuristic, and so is the choice of metric. Recently, Zezula, et. al. [80] have proposed an indexing scheme for approximate NN search queries which performs well in high dimensions.

This dissertation presents an indexing scheme called the AB-tree [61] that very efficiently finds approximate nearest neighbors in high dimensional databases. AB-tree is based on a few interesting data distribution properties [62], such as the distance properties and the angle property, which were observed in our experiments for large
databases with high dimensional data. We have applied this indexing technique to content-based image-retrieval system [64]. Here, we aim to index images based on descriptions derived automatically from the image. Such descriptions are adapted to the image and are usually more effective than external descriptors, in characterizing the visual information illustrated in the image. The similarity between two images can then be determined by comparing their descriptions. The system is designed to administer a heterogeneous collection of natural images. For administering heterogeneous collections, where all kinds of images are likely, we are restricted to low level image-descriptions, which do not involve interpretation. Extensive experiments on synthetic and real data demonstrate that AB-tree outperforms SS-tree by a factor of 86 in 64 dimensions while maintaining 90% accuracy [63]. We have also shown that the AB-tree performs better than VA-file, the best known sequential access method, by a factor of 8. Furthermore, We have explored other heuristics such as clustering and caching to enhance the performance of AB-tree and have developed two variants of AB-tree for very large high dimensional databases: clustered AB-tree and cached AB-tree, which outperform standard AB-tree by a factor of 1.5 to 4.

This dissertation is organized as follows: Chapter 1 discusses related work in multimedia indexing schemes and approximate nearest neighbor search; Chapter 2 describes some distribution properties of high dimensional data; Chapter 3 introduces AB-tree and associated search algorithms; Chapter 4 discusses parameter estimation for AB-tree algorithms; Chapter 5 presents experimental results; Chapter 6 presents PicFinder, a prototype image database system that uses AB-tree indexing scheme described in this dissertation; and Chapter 7 provides a summary of this thesis and
presents a list of contributions as well as future work.
Chapter 1

Related Work

Indexing mechanisms must be incorporated into databases in order to perform queries efficiently. This is true for conventional databases as well as for multimedia databases, where multimedia data objects can be represented as a set of high-dimensional vectors (or points) in feature space. Numerous proposals have been given in recent years for designing index structures to accelerate queries in high dimensions. This chapter gives an overview of multimedia indexing methods. Multimedia databases are spatial databases. Thus multimedia indexing methods often refer to spatial index structures or multidimensional access methods.

There exist many multimedia indexing structures. As more new indexing schemes come out, it becomes more and more difficult to recognize their merits and faults, since every new method seems to claim superiority to at least one access method that has been published previously. However, there are many different criteria to define optimality and many parameters that determine performance. Both time and space efficiency of an access method strongly depend on the data to be processed and the queries to be asked. At present no access method has proven itself to be superior to all its competitors. This chapter does not try to resolve this problem but rather to
give an overview of the pros and cons of a variety of access structures. Performance of exact NN search queries is not good in high dimensions using the existing indexing methods. This chapter also discusses the need for approximate NN search in high dimensions.

1.1 One-dimensional Access Methods

Classical one-dimensional access methods are an important foundation for almost all multidimensional access methods. Although the survey on hashing functions by Knott [44] is somewhat dated, it represents a good coverage of the different approaches. In practice, the most common one-dimensional structures include linear hashing [49, 51] extendible hashing [25], and the B-tree [3].

Other than hashing schemes, the B-tree and its variant [20] organize the data in a hierarchical manner. B-trees are balanced trees that correspond to a nesting of intervals. Each node \( n \) corresponds to a disk page \( D(n) \) and an interval \( I(n) \). If \( n \) is an interior node then the intervals \( I(n) \) corresponding to the immediate descendants of \( n \) are mutually disjoint subsets of \( I(n) \). Leaf nodes contain pointers to data items; depending on the type of B-tree, interior nodes may do so as well. B-trees have an upper and lower bound for the number of descendants of a node. The lower bound prevents the degeneration of trees and leads to efficient storage utilization. Nodes whose number of descendants drops below the lower bound are deleted and its content is distributed among the adjacent nodes at the same tree level. The upper bound follows from the fact that each tree node corresponds to exactly one disk page. If during an insertion a node reaches its capacity, it is split into two. Splits may
propagate up the tree. As the size of the intervals depends on the given data (and the insertion sequence), the B-tree is an adaptive data structure.

Hierarchical access methods such as the B-tree are scalable and behave well in the case of skewed input; they are nearly independent of the distribution of the input data. This is not necessarily true for hashing techniques, whose performance may degenerate depending on the given input data and hash function. This problem is aggravated by the use of order-preserving hash functions [55, 30] that try to preserve neighborhood relationships between data items, in order to support range queries. As a result, highly skewed data keeps accumulating at a few selected locations in image space. For uniformly distributed data, however, extendible as well as linear hashing outperform the B-tree on the average for exact match queries, insertions and deletions.

1.2 Main Memory Access Structures

Early multidimensional access methods did not account for paged secondary memory and are therefore less suited for large spatial databases. In this section, we review two fundamental data structures: KD-tree [7, 6], and quadtree [68], which are adapted and incorporated in several access methods for disk-based data.

One of the most prominent d-dimensional data structures is the KD-tree [7]. The KD-tree is a binary search tree that represents a recursive subdivision of the universe into subspaces by means of (d-1) dimensional hyperplanes. The hyperplanes are iso-oriented, and their direction alternates between the d possibilities. Each splitting
hyperplane has to contain at least one data point, which is used for its representation in the tree. Interior nodes have one or two descendants each and function as a discriminator to guide the search. Searching and insertion of new points are straightforward operations. Deletion is somewhat more complicated and may cause a reorganization of the subtree below the data point to be deleted. One disadvantage of the KD-tree is that the structure is sensitive to the order in which the points are inserted. The adaptive KD-tree [6] mitigates these problems by choosing a split such that one finds about the same number of elements on both sides. While the splitting hyperplanes are still parallel to the axes, they do not have to contain a data point and their directions do not have to be strictly alternating anymore. As a result, the split points are not part of the input data; all data points are stored in the leaves. Interior nodes contain the dimension (e.g. x or y) and the coordinate of the corresponding split. Splitting is continued recursively until each subspace contains only a certain number of points. The adaptive KD-tree is a rather static structure; it is obviously difficult to keep the tree balanced in the presence of frequent insertions and deletions. The structure works best if all the data is known a priori and if updates are rare. Another variant of the KD-tree is the bintree [75]. This structure partitions the universe recursively into d-dimensional boxes of equal size until each one contains only a certain number of points. Even though this kind of partitioning is less adaptive, it has several advantages, such as the implicit knowledge of the partitioning hyperplanes. A disadvantage common to all KD-trees is that for certain distributions no hyperplane can be found which splits the data points evenly [53].

The quadtree with its many variants is a close relative of the KD-tree. For an
extensive discussion of this structure, see [68, 69, 70]. While the term quadtree usually refers to the two-dimensional variant, the basic idea applies to arbitrary $d$. Like the KD-tree, the quadtree decomposes the universe by means of iso-oriented hyperplanes. An important difference, however, is the fact that quadtrees are not binary trees anymore. In $d$ dimensions, the interior nodes of a quadtree have $2^d$ descendants, each corresponding to an interval-shaped partition of the given subspace. These partitions do not have to be of equal size, although that is often the case. For $d = 2$, for example, each interior node has four descendants, each corresponding to a rectangle. These rectangles are typically referred to as the NW, NE, SW, and SE quadrants. The decomposition into subspaces is usually continued until the number of objects in each partition is below a given threshold. Quadtrees are therefore not necessarily balanced; subtrees corresponding to densely populated regions may be deeper than others may. Searching in a quadtree is similar to searching in an ordinary binary search tree. At each level, one has to decide which of the four subtrees need to be included in the future search. In the case of a point query, typically only one subtree qualifies, whereas for range queries there are often several. We repeat this search step recursively until we reach the leaves of the tree.

1.3 Multimedia Indexing Methods

The multidimensional data structures presented in the previous section do not explicitly take secondary storage management into account. They have originally been designed for main memory applications where all the data is available without accessing the disk. Despite growing main memories, this is of course not always the
case. For large multimedia databases, it is necessary for an index structure to be disk-based as opposed to being memory-based. The access methods presented in this section have been designed with secondary storage management in mind. Their operations are closely coordinated with the operating system to ensure that overall performance is optimized. From the spatial database viewpoint, we can distinguish between two major classes of access methods [72].

- Point Access Methods (PAMs), which are used to organize multidimensional point objects (e.g.: cities, where each city is represented by three co-ordinates: longitude, latitude and altitude).

- Spatial Access Methods (SAMs), which are used to organize point objects as well as arbitrary shaped objects (e.g.: street segments, land plots).

In this section, first we will present a selection of point access methods. Then, we will present a selection of spatial access methods as well as some new promising indexing methods in high dimensions.

1.3.1 Point Access Methods

Usually, the points in the database are organized in a number of buckets, each of which corresponds to a disk page and to some subspace of the universe. The subspaces (often referred to as data regions, bucket regions or simply regions, even though their dimension may be greater than two) need not be rectilinear, although they often are. The buckets are accessed by means of a search tree or some d-dimensional hash function.

The grid file [54], for example, uses a directory and a grid-like partition of the universe to answer an exact match query with exactly two disk accesses. Furthermore, there are multidimensional hashing schemes [74, 46, 47], multilevel grid files [78, 40],
and hash trees [59, 58], which organize the directory as a tree structure. Tree-based access methods are usually a generalization of the B-tree to higher dimensions, such as the KDB-tree [66] or the hB-tree [53].

Although there is no total order for objects in two- and higher-dimensional space that completely preserves spatial proximity, there have been numerous attempts to construct hashing functions that preserve proximity at least to some extent. The goal of all these heuristics is that objects that are located close to each other in original space are likely to be stored close together on the disk. This could contribute substantially to minimizing the number of disk accesses per range query.

The Grid File  As a typical representative for an access method based on hashing, we will first discuss the grid file and some of its variants  [36, 59, 78, 24, 13]. The grid file superimposes a d-dimensional orthogonal grid on the universe. Because the grid is not necessarily regular, the resulting cells may be of different shapes and sizes. A grid directory associates one or more of these cells with data buckets, which are stored on one disk page each. Each cell is associated with one bucket, but a bucket may contain several adjacent cells. Since the directory may grow large, it is usually kept on secondary storage. To guarantee that data items are always found with no more than two disk accesses for exact match queries, the grid itself is kept in main memory, represented by d one-dimensional arrays called scales. However, the grid file suffers from two disadvantages:

- It does not work well if the attribute values are correlated.
- It might need a large directory if the dimensionality of the address space is high.
For a theoretical analysis of the grid file and some of its variants see [65, 4].

**Multidimensional Linear Hashing**  Unlike multidimensional extendible hashing, multidimensional linear hashing uses no or only a very small directory. It therefore occupies relatively little storage compared to extendible hashing, and it is usually possible to keep all relevant information in main memory. Several different strategies have been proposed to perform the required address computation. Kriegel and Seeger [46] proposed a variant of linear hashing called multidimensional order-preserving linear hashing with partial expansions (MOLHPE). Another variant that has better order-preserving properties than MOLHPE has been reported by Huth, Six, and Widmayer [23]. Their dynamic z-hashing uses a space-filling technique called z-ordering [57] to guarantee that points that are located close to each other are also stored close together on the disk. One disadvantage of z-hashing is that a number of useless data blocks will be generated, similar as in the interpolation-based grid file [59]. Widmayer (1991) later noted, however, that both z-hashing and MOLHPE are of limited use in practice, due to their inability to adapt to different data distributions.

**The KDB-tree**  The KDB-tree combines some of the properties of the adaptive KD-tree and the B-tree to handle multidimensional points. It partitions the universe in the manner of an adaptive KD-tree and associates the resulting subspaces with tree nodes. Each interior node corresponds to an interval-shaped region. Regions corresponding to nodes at the same tree level are mutually disjoint; their union is the complete universe. The leaf nodes store the data points that are located in the
corresponding partition. Like the B-tree, the KDB-tree is a perfectly balanced tree that adapts well to the distribution of the data. Other than for B-trees, however, no minimum space utilization can be guaranteed.

**The hB-Tree** The hB-tree (holey brick tree) [53, 52] is related to the KDB-tree in that it utilizes KD-trees to organize the space represented by its interior nodes. One of the most noteworthy differences is that node splitting is based on multiple attributes. As a result, nodes no longer correspond to d-dimensional intervals, but to intervals from which smaller intervals have been excised. Similar to the BANG file, the result is a somewhat fractal structure (a holey brick) with an external enclosing region and several cavities called extracted regions. As we will see later, this technique avoids the cascading of splits that is typical for many other structures.

**Space-Filling Curves for Point Data** We already mentioned the main reason why the design of multidimensional access methods is so difficult compared to the one-dimensional case: There is no total order that preserves spatial proximity. One way out of this dilemma is to find heuristic solutions, i.e., to look for total orders that preserve spatial proximity at least to some extent. The idea is that if two objects are located close together in original space, there should at least be a high probability that they are close together in the total order, i.e., in the one-dimensional image space. For the organization of this total order one could then use a one-dimensional access method (such as a B+-tree), which may provide good performance at least for some spatial queries.
One thing all proposals have in common is that they first partition the universe with a grid. Each of the grid cells is labeled with a unique number that defines its position in the total order (the space-filling curve). The points in the given data set are then sorted and indexed according to the grid cell they are contained in. Note that while the labeling is independent of the given data, it is obviously critical for the preservation of proximity in one-dimensional address space. That is, the way we label the cells determines how clustered adjacent cells are stored on secondary memory.

Based on several experiments, Abel and Mark [1] conclude that z-ordering [57] and the Hilbert curve [27] are most suitable as a multidimensional access method. Jagadish [39] and Faloutsos and Rong [26] all prefer the Hilbert curve among those two. Z-ordering is one of the few access methods that has found its way into commercial database products. In particular, Oracle has adapted and integrated the technique into its database system [37].

An important advantage of all space-filling curves is that they are practically insensitive to the number of dimensions if the one-dimensional keys can be arbitrarily large. Everything is mapped into one-dimensional space, and one’s favorite one-dimensional access method can be applied to manage the data. An obvious disadvantage of space-filling curves is that incompatible index partitions cannot be joined without recomputing the codes of at least one of the two indexes.

1.3.2 Spatial Access Methods

All multidimensional access methods presented in Section 1.3.1 have been designed to handle sets of data points and to support spatial searches on them. None of those
methods is directly applicable to databases containing objects with a spatial extension. Typical examples include geographic databases, containing mostly polygons, or mechanical CAD data, consisting of three-dimensional polyhedra. In order to handle such extended objects, the proposed methods in the literature form the following two classes [72]:

1. Methods that use space-filling curves

2. Methods that use treelike structures

In the following, we will first discuss transformation to higher-dimensional space, and then describe space-filling curves for extended object, and final present several promising treelike indexing structures and some new indexing methods in high dimensions.

**Mapping to Higher-Dimensional Space** Simple geometric shapes can be represented as points in higher-dimensional space. The original and final spaces are called native space and parameter space, respectively [56]. A range query in native space can also be translated to a range query in parameter space [28].

The strong point of this idea is that we can turn any PAM into a SAM with very little effort. This approach has been used or suggested in several settings, for example, with grid files, B-trees, and hB-trees, as the underlying PAM.

The weak points are the following: First, the parameter space has high dimensionality, inviting "dimensionality curse" problems earlier on. Second, except for range queries, there are no published algorithms for nearest neighbor and spatial join queries. Third, If the original database contains more complex objects, they have to
be approximated - e.g. by a rectangle or a sphere - before transformation. In this case, the point access method can only lead to a partial solution.

**Space-Filling Curves for Extended Objects** Space-filling curves are a very different type of transformation approach that seems to suffer less from some of the drawbacks listed in the previous subsection. Space-filling curves can be used to represent extended objects by a list of grid cells or, equivalently, a list of one-dimensional intervals that define the position of the grid cells concerned. A In other words, a complex spatial object is approximated not by only one simpler object, but by the union of several such objects.

The z-ordering approach by Orenstein and Merrett [57] is one of the more popular approaches of this kind. A region typically breaks into one or more pieces (blocks), each of which can be described by a z-value. A z-ordering is based on an underlying grid, the resulting set of regions is usually only an approximation of the original object. The termination criterion depends on the accuracy or granularity (maximum number of bits) desired. More regions obviously yield more accuracy, but they also increase the overhead, which affects the overall performance of the resulting data structure. As pointed out by Orenstein [55] there are two possibly conflicting objectives: First, the number of regions to approximate the object should be small, since this results in less index entries. Second, the accuracy of the approximation should be high, since this reduces the expected number of false drops (i.e., objects that are paged in from secondary memory, only to find out that they do not satisfy the search predicate).
**R-Trees** The R-tree proposed by Guttman [34] can be thought of an extension of the B-tree for multidimensional objects. A spatial object is represented by its minimum-bounding rectangle (MBR). Nonleaf nodes contain entries of the form (ptr, R), where ptr is a pointer to a child node in the R-tree; R is the MBR that covers all rectangles in the child node. Leaf nodes contain entries of the form (obj-id, R) where obj-id is a pointer to the object description, and R is the MBR of the object. The main innovation in the R-tree is that parent nodes are allowed to overlap. This way, the R-tree can guarantee good space utilization and remain balanced. Search algorithms can be applied almost unchanged. The only differences are due to the fact that the overlap may increase the number of search paths we have to follow. Even a point query may require the investigation of multiple search paths because there may be several subspaces at any index level that include the search point. For range and region queries, the average number of search paths increases as well.

The R-tree inspired much subsequent work, whose main focus was to improve the search time. When the database is static, other optimizations can be incorporated into the index structures in order to enhance search performance. A packing technique is proposed in [67] to minimize the overlap between different nodes in the R-tree for static data. The idea was to order the data in, say, ascending x-low value, and scan the list, filling each leaf node to capacity. An improved packing technique based on the Hilbert curve is proposed in [41]: the idea is to sort the data rectangles on the Hilbert value of their centers.

One of the most successful idea in R-tree is the idea of deferred splitting: Beckmann et al. proposed the R*-tree [5], which was reported to outperform Guttman's
R-tree by about 30%. The main idea is the concept of forced reinsert, which tries to defer the splits to achieve better utilization: When a node overflows, some of its children are carefully chosen, and they are deleted and reinserted, usually resulting a better-structure R-tree.

A class of variations considers more general minimum bounding shapes, trying to minimize the dead space that an MBR may cover. Gunther proposed the cell trees [15], which introduce diagonal cuts in arbitrary orientation. Jagadish proposed the polygon trees (P-trees) [38], where the minimum bounding shapes are polygons with slopes of sides 0, 45, 90, and 135 degrees. Minimum bounding shapes that are concave or even have holes have been suggested (e.g., in the hB-tree [53]).

However, as dimension rises, the increasing overlap among the bounding hyper-rectangles deteriorates search efficiency. In addition, the bounding hyper-rectangles become less efficient in dividing the space into neighborhoods as dimension increases. A neighborhood exists when many of the nearest neighbors to a point reside within the same bounding region. Because the R-trees do not create neighborhoods, nearest neighbor searching is inefficient with R-trees in high dimensions.

**X-tree** The X-tree [11] is a modified R-tree that attempts to decrease the overlap of bounding hyper-rectangles by using two techniques: First, the X-tree introduces an overlap-free split algorithm that is based on the split history of the tree. Second, if the overlap-free split algorithm would lead to an unbalanced directory, the X-tree omits the split and the according directory node becomes a supernode. These supernodes span more than one disk block and keep bounding regions disjoint. In this way, the
X-tree outperforms R*-tree by a factor of up to 400 for point queries.

**SS-tree**  On the other hand, the SS-tree [79] partitions the search space by utilizing bounding hyperspheres instead of bounding hyperrectangles. This decreases the amount of storage space needed to represent a bounding region inside of an index node (the space needed to store a bounding hypersphere is nearly half the space needed to store a bounding hyperrectangle). As a result, the fanout of index nodes increases. The SS-tree improves the performance of NN searching in high dimensions over the R*-tree. However, the bounding hyperspheres cover much more volume in the search space than bounding hyper-rectangles. This causes a great deal of overlap among bounding hyperspheres in the tree. The result is that the performance of NN searching degrades significantly as dimension increases.

**SR-tree**  In order to partition the space into neighborhoods while minimizing the overlap among bounding regions, the SR-tree [42] has been proposed. The SR-tree utilizes the intersection of a hypersphere and a hyperrectangle as its bounding region. This decreases overlap among bounding regions and improves NN search performance. However, the SR-tree must store a bounding hyperrectangle and a bounding hypersphere for each node in the tree. This decreases the fanout of the index nodes and increases the total number of nodes in the tree.
**M-tree**  M-tree [18] is another higher dimensional feature vector indexing method. The main idea behind the M-tree is to combine the advantages of balanced and dynamic spatial access methods with the capabilities of static metric trees to index objects using features and distance functions. Leaf nodes of an M-tree contain indexed objects, whereas non-leaf nodes store so-called routing objects. A routing object is a database object to which a routing role is assigned by a specific routing algorithm.

**Pyramid-Technique**  The Pyramid Technique [9] is a new indexing method for high-dimensional data spaces. The Pyramid-Technique is highly adapted to range query processing using the maximum metric $L_{\text{max}}$. In contrast to all other index structures the performance of the Pyramid-Technique does not deteriorate when processing range queries on data of higher dimensionality. The Pyramid-Technique is based on a special partitioning strategy that is optimized for high-dimensional data. The basic idea is to divide the data space first into 2d pyramids sharing the center points of the space as a top. In a second step, the pyramids are cut into slices parallel to the basic of the pyramid. These slices form the data pages. Furthermore, this partition provides a mapping from the given d-dimensional space to a 1-dimensional space. Therefore, a B+-tree can be used to manage the transformed data. Pyramid-technique clearly outperforms other index structures for range queries. However, for queries having a bad selectivity, i.e. a high number of answers, or extremely skewed queries, a linear scan of the database is faster than the Pyramid-Technique.
**Voronoi Cells** In [10], a technique is proposed based on the precomputation of the solution space of any arbitrary nearest-neighbor search. This corresponds to the computation of the voronoi cells of data points. Since voronoi cells may become rather complex when going to higher dimensions, the authors presented a new algorithm for the approximation of high-dimensional voronoi cells using a set of minimum bounding hyperrectangles. The voronoi cells are stored in an index structure efficient for high-dimensional data spaces. As a result, nearest neighbor search corresponds to a simple point query on the index structure. Although the technique is based on a precomputation of the solution space, it supports insertions of new data points. However, this precomputation technique is only suitable for searching one nearest neighbor of a query point.

**Multi-Step k-Nearest Neighbor Search** While algorithms that directly based on index work well for simple medium-dimensional similarity distance functions, they do not perform efficiently in high dimensions. A multi-step query processing strategy can be used to improve the performance. In a multi-step query processing environment, one or more filter steps produce sets of candidates that are exactly evaluated in one or more subsequent refinement steps. The crucial correctness requirement is to prevent the system from producing false drops. This means that no actual result may be dismissed from the set of candidates. The number of candidates is a fundamental efficiency parameter. A multi-step algorithm for k-nearest neighbor search has already been developed and successfully been applied to similarity search in 3-D medical image databases [45]. Seidl and Kriegel [71] present a novel multi-step algorithm
that is guaranteed to produce the minimum number of candidates. Experimental evaluations demonstrate the significant performance gain over the previous solution.

**VA-file** Weber, et. al. [77] have shown that under the assumption of uniformity and independence, and when the number of dimensions is sufficiently high, sequential scan may outperform indexing schemes for *exact NN* search queries. Consequently, they proposed the vector approximation file (VA-file) based on approximations to make the sequential scan as fast as possible. The VA-file divides the data space into $2^b$ rectangular cells where $b$ denotes a user specified number of bits (e.g. some number of bits per dimension). Instead of hierarchically organizing these cells like in grid-files or R-trees, the VA-file allocates a unique bit-string of length $b$ for each cell, and approximates data points that fall into a cell by that bit-string. A VA-file is an array of these compact, geometric approximations. By scanning the entire approximation file first (filtering step), *NN* queries need only visit a fraction of the vectors. The VA-file outperforms R-trees and X-tree if dimensionality becomes large. However, Performance of VA-file method deteriorates linearly with the size of databases due to their sequential nature.

### 1.4 Approximate Nearest Neighbor Search

The number of features involved in real world applications may approach thousands. Dimension reduction techniques have been proposed to reduce the number of features. The TV-tree [50] attempts to use only a subset of the dimensions for indexing. The most significant features are used at higher levels of the tree, while more features get
utilized as the tree is descended. The TV-tree gets impressive results; however, it can only be used in certain types of applications [79]. Despite the use of dimension reduction techniques such as principal component analysis, vector spaces of several hundred dimensions are typical. Unfortunately, this relaxation of goals has not removed the curse of dimensionality.

Another approach that has recently been used to avoid the curse of dimensionality is approximate nearest neighbor. An exact nearest neighbor search is not necessary and may be an overkill, since the features and the similarity measure themselves are chosen based on heuristics, and are not mathematically precise parameters. In fact, in multimedia database systems, such as IBM’s QBIC [29] or MIT’s Photobook [60], the mapping of attributes of objects to coordinates of vectors is heuristic, and so is the choice of metric.

Recent work in computational geometry has produced substantial results on nearest neighbor search in main memory. Dobkin and Lipton [22] were the first to provide an algorithm for nearest neighbors in d dimensional Euclidean space, with query time $O(2^d \times \log(n))$ and preprocessing cost $O(n^{2(d+1)})$. Clarkson [19] showed that query time complexity could be reduced to $O((1/\epsilon)^{(d/2) \log(n)})$ with $O((1/\epsilon)^{(d/2)} \times \log(p) \times n)$ space, where $\epsilon$ is the relative error in distance and $p$ is the ratio between the furthest-pair and closest-pair interpoint distances. Later Chan [17] showed that the factor of $\log(p)$ could be removed from the space complexity. Kleinberg [43] showed that it is possible to eliminate exponential dependencies on dimension in query time, but with $O(n \times \log(d))^{(2d)}$ space. Recently, Indyk and Motwani [38] and independently Kushilevitz, Ostrovsky and Rabani [48], have announced algorithms that eliminate
all exponential dependencies in dimension, yielding a query time $O(d \times (\log^{O(1)}(dn)))$ and space $O((dn)^{(O(1)})$. The $O$-notation hides constant factors that depend exponentially on $\epsilon$, but not on dimension. Arya, Mount, Netanyahu and Silverman [2] gave an algorithm with query time $O(c \times \log(n))$ and space $O(dn)$, where $c < d \times [1 + 6 \times d/\epsilon]^d$. The exponential factors in query time do imply that the algorithm is not practical for large values of $d$.

Most of the methods mentioned above used data structures that are stored in main memory. Much of the query time includes the computation cost in main memory. Our focus is on nearest neighbor search in databases. Thus I/O cost is the main source of query time. We [61] propose a fast approximate nearest neighbor search in high dimensional databases. The fast approximate search index structure called the AB-tree, which is based on the SS-tree. AB-tree uses heuristic to decide whether or not to access a node in the index tree based on the intersecting angle and the weight of the node. The heuristic is motivated by an interesting observation: high dimensional data points inside a bounding hypersphere usually fall within a small interval of angle around 90 degree, which is justified by sound theoretical as well as experimental arguments. In contrast to the algorithms that have been proposed in earlier works, the performance of the AB-tree algorithm does not deteriorate when processing nearest neighbor queries as dimension increases.
Chapter 2

High Dimensional Data Distribution Properties

Understanding the distribution of data points in high dimensional Euclidean space has been a focus of computer science research for years. Much of this research deals with the development of efficient algorithms that exploit the spatial relationships of data points in high dimensions. For example, in building index structures for databases with high dimensional data, bounding hyper-rectangles and bounding hyperspheres are often used. Various nearest neighbor search algorithms have been proposed in the past that try to minimize the overlap between these hyperspheres and/or the hyper-rectangles. However, the relationships between the amount of overlapping space and the number of data points inside the overlapping space needs to be understood for developing efficient algorithms for applications involving high dimensional data.

The focus of this research is to study the properties of data distribution in high dimensional Euclidean space and exploit these properties for efficient application development. The basic model for characterizing the properties of data distribution in high dimensional Euclidean space is based on the positions of the points in the
hyperspace relative to various high dimensional geometrical shapes such as hyperspheres, hypercubes, hyperplanes and hyperangles. For example, if the maximum and the minimum pairwise distances of a set of random points are denoted by $d_{\text{max}}$ and $d_{\text{min}}$, then an important property is that the ratio of $d_{\text{max}}$ and $d_{\text{min}}$ for a given set of points approaches the value 1 with increasing dimensionality. This property implies that the data points are distributed close to the surface of a hypersphere. We are representing this relative distance property using the maximum and the minimum pairwise distances of a set of points. There are other ways to capture this relative distance property which will be discussed later.

We investigated several such properties that govern the behavior of data distribution in high dimensional Euclidean space. Some of these properties that we investigated in this research are described in the following sections.

- **Distance Properties**
  - Distance Property 1 (DP1):
    For a given dimension, the ratio $d_{\text{max}}/d_{\text{min}}$ increases slowly as the number of points increases.
  - Distance Property 2 (DP2):
    For a fixed number of points, $d_{\text{max}}/d_{\text{min}}$ decreases and becomes close to 1 as the dimension increases.

- **Radius Properties**
  - Radius Property 1 (RP1):
    The radius of a hypersphere that bounds a set of points increases very slowly with the number of points.
  - Radius Property 2 (RP2):
    The radius of a hypersphere that bounds a set of points increases rapidly with an increase in dimensionality.
  - Radius Property 3 (RP3):
    Even for a small number of uniformly distributed points (e.g. 5 points) within a hypercube, much of the bounding hypersphere for this set of points lies outside the hypercube.
• Angle Property

  – Angle Property 1 (AP1):
  Given a set of points within a bounding hypersphere, and some reference point P, we define an angle $\theta$ between the reference point P and each point Q in the hypersphere, with respect to the center C of the hypersphere. The distribution of points with respect to the angles $\theta$ is proportional to $\sin^d \theta$ where $d$ is the dimension of the points. As dimensionality gets larger, for any given reference point, the value of the angle $\theta$ falls inside a decreasing interval of angles around $\pi/2$ (90 degrees).

2.1 Distance Properties

In this section we discuss the distance properties (DP1 and DP2). These properties describe how close the points are from being equidistant to each other. For example, when the ratio $d_{\text{max}}/d_{\text{min}}$ is 1, all data points are equidistant from each other, and the points lie on the surface of a hypersphere of radius

$$r_D = \sqrt{\frac{D}{2 \times (D + 1)}} \times d$$

(2.1.1)

where $D$ is the dimension and $d$ is the distance between a pair of points.

When the ratio is not 1 but close to 1, the data points are close to equidistant from each other. Some of the experimental results are shown in Figure 2.1. All experimental results have been averaged over 1000 random trials. For a fixed number of points in a fixed dimension, the distribution of the ratio $d_{\text{max}}/d_{\text{min}}$, is similar to a sharp Gaussian curve with a standard deviation less than 5 percent of the average. As the dimension and/or the number of points increase, the standard deviation becomes smaller. Figure 2.1(a) shows that for a 100-dimension dataset, as the number of points increases from 5 to 50000, $d_{\text{max}}/d_{\text{min}}$ increases slowly from 1.19 to 1.76. It demonstrates that in high dimensions, a dataset of 50000 points has nearly the same
characteristics as that of 5 points. In Figure 2.1(b), we show the effect of dimension on $d_{\text{max}}/d_{\text{min}}$ for a dataset with 500 data points. It is clear that $d_{\text{max}}/d_{\text{min}}$ decreases as the dimension increases and becomes close to 1. This means that as the dimension increases, the data points lie within a narrow band of the surface of a hypersphere.

![Figure 2.1](image)

(a) $d_{\text{max}}/d_{\text{min}}$ vs. Number Of Points

(b) $d_{\text{max}}/d_{\text{min}}$ vs. Dimension

Figure 2.1: Effects of Dimension and Number of Points on $d_{\text{max}}/d_{\text{min}}$

This property of data points in high dimensions could also be described by probabilistic models. In the following we give a brief description of a probabilistic model describing the property that the points lie within a narrow band of a hypersphere.

**Points Lie Near the Surface of a Hypersphere: A Probabilistic Approach**

Assume that each coordinate of the d dimensions has uniform distribution. The probability density function of distance from a center with coordinates, $c_i$, $i=1$ to $d$, to a random point is given by the following generating function:

$$G(s) = \frac{1}{(c_m + 1)^d} \left[ \prod_{j=1}^{d} \sum_{i=0}^{c_m} s^{(i-c_j)^2} \right]$$

(2.1.2)

The coefficient of $s^{n^2}$ in the above generating function will give the probability density of points at a distance $n$.  

29
The plot of this probability distribution function for 100 dimensions is shown in Figure 2.2. The figure shows that the points are distributed in a narrow band. Superimposed on this figure are the results of experiments for 100,000 points. The experimental results follow more closely the above theoretical distribution if the number of points are increased.

Figure 2.2: Probability Distribution Function of Distance for Dimension=100

The figure shows that the distances between a given center and the random points fall within a narrow band. As the number of dimensions increases the relative narrowness of the band with respect to the distance continues to decrease.

2.2 Radius Properties

Radius properties play a significant role in describing the relative positions of groups of data points inside a hypersphere. For example, a bounding hypersphere for a set of points encloses other bounding hyperspheres containing subsets of these points. By property RP1 these enclosed hyperspheres are almost as big as their bounding
hypersphere in high dimensions. This indicates that all these hyperspheres are almost totally overlapping with each other in high dimensions.

2.2.1 Radius of the Minimum Bounding Hypersphere

In this subsection, we study the effects of dimension and the number of points on the radius of Minimum Bounding Hyperspheres (MBH). The results are shown in Figure 2.3. First, we have tested the effect of the number of points on the radius of MBH. For a 100-dimension dataset, as the number of points increases from 5 to 50000, the radius of MBH increases slowly from 4.54 to 5.32. This is shown in Figure 2.3(a). It is clear that the radius of MBH is not sensitive to the number of points in high dimension. The radius of MBH of 50 points is very close to that of 50000 points in high dimension. On the other hand, for a given set of 500 points the radius of MBH increases significantly as the dimension increases. This is shown in Figure 2.3(b).

![Graphs](a) Radius vs. Number of points (b) Radius vs. Dimension

Figure 2.3: Effects of Dimension and Number of Points on Radius of MBH
2.2.2 Nonuniform Distribution of Data Points inside the MBH

In this subsection, we give a justification that a set of uniformly distributed data inside a hypercube lie near the surface of the hypersphere symmetrically around the center and nonuniformly in high dimension. Radius property RP3 shows that even for a small number of points much of the MBH lies outside the hypercube. This is because the maximum radius of the hypersphere enclosed inside a unit hypercube is 0.5, and the radius of the MBH is greater than 4 (Figure 2.3(a)) even for a data set of 5 points. Parts of the hypersphere outside the hypercube are always empty. This results in the nonuniform distribution of data points inside the hypersphere. Because the radius of the MBH increases significantly as the number of dimensions increases, more and more parts of hypersphere are outside the hypercube as the dimension increases, which means the data points inside the hypersphere become more and more nonuniform (e.g., clustered) as the dimension increases.

2.3 Angle Property

In this section we present an angle-based representation of high-dimensional Euclidean space, and compare its property to those of the distance-based properties discussed above. We found that the angle-based property has characteristics similar to the distance-based properties. This is because they both use only one parameter to classify the high-dimensional Euclidean space.

For a given reference point R, we define the angle \( \theta \) of a data point P as the angle between P and R with respect to the center C of the bounding hyper-sphere, as shown
in Figure 2.4. We can map the whole sphere into an angle range \([0, \pi]\). The volume of a d-dimensional hypersphere with a radius \(r\) is \(Vol^d(r)\) as shown below.

\[
Vol^d(r) = \int_0^\pi \frac{\pi^{(d-1)/2}}{\Gamma(d/2 + 1)} r^d \sin^d \theta \, d\theta
\]

(2.3.1)

where \(\Gamma(x + 1) = x \cdot \Gamma(x)\), \(\Gamma(1) = 1\) and \(\Gamma(\frac{1}{2}) = \sqrt{\pi}\).

Thus, the distribution of the data with respect to the angle \(\theta\) is proportional to \(\sin^d\theta\), where \(d\) is the dimension of the data set. The distributions of data points with respect to the angle \(\theta\) in different dimensions shown in Figure 2.5 are mapped into the small range around \(\pi/2\).

![Figure 2.5: Distribution of d-Dimensional Data With Respect to the Angle Θ](image_url)
Therefore, the probability of d-dimensional data points with the angle less than $\theta$ ($0 \leq \theta \leq 180^0$) is determined by the following:

$$
Prob(\theta, d) = \frac{\Gamma \left( \frac{d}{2} + 1 \right)}{\Gamma \left( \frac{d-1}{2} + 1 \right) \times \sqrt{\pi}} \int_0^\theta \sin^d \theta d\theta
$$

(2.3.2)

We will use this probability for our heuristic given in later sections. The probabilities of data points with the angle less than $\theta$ in different dimensions are shown in Figure 2.6. It is obvious that the probability of a data point with angle less than 90 degree decreases dramatically as dimension increases. For a given dimension, the probability of a data point with the angle less than $\theta$ decreases exponentially as $\theta$ decreases, and becomes small even for quite a large angle. For example, the probability of a 64-dimensional data point with angle less than 65 degree is only $1.8 \times 10^{-4}$.

![Figure 2.6: Probability of d-dimensional Data Points With Angle Less Than $\theta$](image)

Figure 2.6: Probability of d-dimensional Data Points With Angle Less Than $\theta$
2.4 Interpretation of the Distribution of High Dimensional Data in Bounding Hyperspheres Based on the Properties

We have introduced properties relating to the distance between points in high dimensions, the radius of bounding hyperspheres in high dimensions, and the angular distribution of a set of points in a bounding hypersphere in high dimensions. By combining these properties together, we get a better idea of how points are distributed in bounding hyperspheres in high dimensions. Figure 2.7 illustrates the distribution according to the properties we have introduced.

![Diagram]

Figure 2.7: Distribution of High Dimensional Data in Bounding Hyperspheres

In the figure, there are two bounding hyperspheres enclosed by a larger hypersphere. By DP2, all the points bounded by a hypersphere fall close to the surface of the hypersphere. By RP2, the radii of the enclosed hyperspheres get almost as large as the radius of their bounding hypersphere. Lastly, by AP1, the points fall in a small angle interval around 90 degrees with respect to the bounding hypersphere center, C, and the reference point, Q.
2.5 Experimental Results with Cluster Data

Figure 2.8 contains graphs that illustrate three of the properties described in the beginning of this chapter. The experiments were performed on clustered data sets that were created for each dimension depicted. Each data set contains 100000 points. There were 100 clusters in each data set with 1000 points in each cluster. The points for the data sets were generated in the range $[0,1)$ for each dimension. In order to create a cluster, a random point is generated in the space. This point serves as the cluster center. Next, a random radius is produced for the cluster. Points are then generated uniformly on the surface of the cluster hypersphere and are subsequently shifted randomly along the radius. The data space was partitioned hierarchically with bounding hyperspheres. The lowest level of bounding hyperspheres contained the actual data points. At this level, the hyperspheres are called leaves.

Figure 2.8(a) shows the ratio of the average of distances from the center of a bounding hypersphere to each point enclosed within the hypersphere and the radius of the bounding hypersphere. For this experiment, the ratio was computed for leaf. Inside each bounding hypersphere, the distance to each point in the leaf from the center of the hypersphere was computed. The average distance over all the points in the leaf was found and divided by the radius of the bounding hypersphere. The average ratio was taken over all leaves. The minimum and maximum ratios were also recorded over all leaves (the maximum is always 100% since the radius is the distance to the farthest point from the center). It can be seen that the average and minimum ratios increase as dimension increases. This implies that all the points inside each
Figure 2.8: Properties in High Dimensions

leaf are falling closer to the surface of the bounding hypersphere as dimensionality rises.

In Figure 2.8(b), a comparison is done between the radii of the leaves versus the radii of their parent-hyperspheres (the hyperspheres that each bound a set of leaves). The ratio displayed is the radius of a leaf divided by the radius of its parent. The average ratio is taken over all leaves in the tree. The graph shows that the average and maximum ratios increase as dimension increases. This implies that the size of the leaf
bounding hyperspheres approaches the size of the leaf parent bounding hyperspheres as dimensionality gets larger.

Figure 2.8(c) shows the distribution of the angles as described above. For these experiments, each leaf for a given dimensionality was examined. For a given query point, the angle was computed between the query point and each point in the leaf with respect to the center of the leaf bounding hypersphere. Thus, the graph shows the distribution of 100000 angles (one angle for each point in the data set) for each dimensionality. In the experiments, each distribution was averaged over 100 query points. As dimensionality increases, the angles begin to cluster around 90 degrees.

In Section 2.3, we showed that the distribution of the data with respect to the angle $\theta$ is proportional to $\sin^D\theta$ where $D$ is the dimensionality of the data set. The experimental results of Figure 2.8(c) are consistent with this theoretical result.

## 2.6 Related Research on Data Distribution Properties

In recent years several researchers have studied data distribution properties for high dimensional vectors in Euclidean space. Bozkaya, et.al. [14] have presented distribution of pairwise distances for 50,000 random vectors on a unit hypercube. This distribution is for only 20 dimensions. They show that the distribution of pairwise distances is similar to a sharp Gaussian curve. Thus, the pairwise distances between any two points fall mostly within a small interval. This is similar to our distance property $DP2$. While the above paper investigates distribution of pairwise distances,
DP2 considers $d_{\text{max}}/d_{\text{min}}$ for a given set of points. Katayama et.al. [42] have investigated the distribution of $d_{\text{min}}/d_{\text{max}}$ for increasing dimensionality for up to 64 dimensions using a set of 100,000 random points, created on a unit hypercube. They have shown experimentally that the ratio of $d_{\text{min}}/d_{\text{max}}$ increases with dimension and the value of $d_{\text{min}}$ grows drastically as dimensionality increases. This result is similar to the DP2 property of ours which was derived independently. DP2 also describes an asymptotic behavior of the ratio $d_{\text{max}}/d_{\text{min}}$. From the experimental results the authors [42] conclude that the variation in pairwise distances reduces as dimensionality increases and that each point, contained in the uniform data set, has similar distances to the others in high dimensionality. Berchtold et. al. [8] presented a new cost model for nearest neighbor search in high-dimensional data space. In this paper the authors [8] consider the intersection volume of a hyperrectangle and a query hypersphere. Though this work does not discuss the properties described here, the intersection volume has some relationships to our angle property.

Properties similar to DP2 and AP1 have also been investigated by researchers in mathematics probability theory and statistics. Freedman et. al. [21] have studied the dimensional behavior of the lengths of random vectors in high dimension. The authors [21] show that most vectors have lengths near $\sigma^2 D$ where $\sigma^2$ is positive and finite and D is the dimensionality, as the number of points and dimension approach infinity. However, this paper does not consider $d_{\text{max}}/d_{\text{min}}$ for a given set of points. Further, a measure of $d_{\text{max}}/d_{\text{min}}$ for various dimensionality is helpful in characterizing data distribution for various dimensions. This paper also shows that most vectors are nearly orthogonal to each other as dimension approaches infinity. Our angle property
$A_{P1}$ provides a quantitative relationship between the angle and dimension.
Chapter 3

Angle based Balanced index tree: AB-tree

The heuristic we have developed to decrease unnecessary node accesses in the search for K-NN’s is based on the estimated angle between the bounding hypersphere and the candidate set hypersphere (the candidate set hypersphere is the hypersphere with the query point as the center and the distance from the query point to the current $K^{th}$ NN as the radius) as well as the weight of the node; the weight of a node is the number of data points inside the node. In the following sections we will first discuss an algorithm for computing better bounding hyperspheres than the one computed by the traditional method. We will then discuss the criteria for node accesses used in the AB-tree followed by a description of the search algorithm for the AB-tree. Finally, we present two improved variants of AB-tree: the clustered AB-tree and the cached AB-tree.

3.1 Computing Better Bounding Hyperspheres

In the traditional approach to computing a bounding hypersphere for a given set of points the center of the bounding hypersphere is computed first by averaging the
coordinates of the points. The radius of the bounding hypersphere is then computed by finding the farthest point from the center. Algorithm 3.1.1 computes a more efficient bounding hypersphere than the above approach. Experimental results show a reduction in the size of radius by about 15%, on the average. This reduces overlaps between the bounding hyperspheres.

Algorithm 3.1.1 Compute Better Bounding Hyperspheres

1. Get the initial center \( O \) of bounding hypersphere using the average of the coordinates of the data points;
2. Repeat {
   2.1 Find the data point \( V \) farthest away from \( O \). \( d(V,O) \) is the radius \( r \);
   2.2 Find the data point \( U \) which is the second farthest away from \( O \);
   2.3 Move \( O \) toward \( V \) along the direction of \( OV \) by \( (r - d(U,O))/2 \);
} Until \( (r - d(U,O)) \leq \text{tolerance} \) (we used tolerance of \( 1e-8 \));

/* The loop normally terminates in 4 to 8 iterations */

3.2 Criteria for Node Access

In this section, we discuss the criteria for node access used in the AB-tree. The basic idea is first estimate the possible number of points inside the intersection between candidate set hypersphere and the bounding hypersphere of the node. If the possible number of points inside the intersection is smaller than some threshold, then it is worth discarding the node to improve performance while sacrificing small accuracy. We first present the impact of the angle property in Section 3.2.1, then we discuss the impact of distance property and weight of node in Section 3.2.2, and finally we present the criteria for node access based multiple angles and distance in Section 3.2.3 and Section 3.2.4, respectively.
3.2.1 Impact of Angle Property

By angle property, high dimensional data points inside a bounding hypersphere fall within a small interval of angles around 90 degrees. Figure 3.1(a) illustrates this. Since this is true for any given query point, this property can be added into existing NN searching algorithms that will reduce the number of nodes in the index tree that need to be accessed.

![Diagram](image)

(a) Empty Space in Bounding Hypersphere
(b) Overlap in High Dimensions

Figure 3.1: Illustration of Angle Property

In an indexing mechanism that utilizes bounding hyperspheres to partition the data space (e.g. the SR-tree and the SS-tree), the bounding hyperspheres are defined by a center point and a radius. The hyperspheres may enclose a set of data points or a set of smaller hyperspheres. Each bounding hypersphere represents a node in the index tree.

In the existing NN search algorithms, if the candidate set hypersphere intersects with a bounding hypersphere, then that node must be accessed. However, we know
that if the overlap between these two hyperspheres is small, then it is likely that no points fall inside the overlap. Hence, there is no need to access the node.

In the heuristic, an angle must first be defined. The search algorithm will not access a node if the candidate set hypersphere does not intersect with the bounding hypersphere for the node at a given angle. For example, in Figure 3.1(b), the defined angle for the heuristic is $\theta_1$. The candidate set hypersphere only intersects with the bounding hypersphere at angle $\theta_3$, which is less than the tangent angle $\theta_2$ in this case. Because $\theta_3$ is less than the angle $\theta_1$, the search algorithm will not access the node (i.e., discard the node).

Tangent angle $\theta_2$ can be computed by finding the angle of the tangent vector to the candidate set hypersphere that originates from C since the distance from C to Q and the radius of the candidate set hypersphere are known. Intersecting angle $\theta_3$ can also be computed based on radii of the candidate set hypersphere and candidate set hypersphere as well as the distance from C to Q. In our heuristic, we are using the estimated angle, which is either the tangent angle $\theta_2$ or intersecting angle $\theta_3$ based on the weight of a node.

### 3.2.2 Impact of Distance Property and Weight of Nodes

As mentioned in the beginning of Section 3.2, first we need to determine the upper bound of the possible number of points inside the overlap area between the candidate set hypersphere and the bounding hypersphere of a node. If this upper bound is less than a predefined threshold $T$, the search algorithm will not access the node (i.e., discard the node). The possible number of points inside the overlap area is
not only dependent on the estimated angle, but also dependent on the dimension as well as the weight of a node. By distance property 2, we know the points are distributed around the surface of a hypersphere. So we can use the possible number of points falling within the angle \([0, \theta]\) as an appropriate upper bound of the possible number of points inside the overlap area, where \(\theta\) is the estimated angle between the bounding hypersphere of the node and the candidate set hypersphere. Thus the upper bound of the possible number of points inside the overlap area can be estimated as 
\[ N \times \text{prob}(\theta, d) \]
where \(N\) is the weight of the node, \(d\) is the dimension of the data set, and \(\text{prob}(\theta,d)\) is the probability of \(d\)-dimensional data points with the angle less than \(\theta\) \((0 \leq \theta \leq 180^\circ)\) as described in Section 2.3. Therefore, the criteria for node access is

\[ N \times \text{prob}(\theta, d) \geq T \]  \hspace{1cm} (3.2.1)

where \(T\) is the predefined threshold. It should be noted that with this heuristic incorporated into the search algorithm, the algorithm is no longer guaranteed to find the entire true \(KNN\) for a given query point. Changing the threshold \(T\) can control the accuracy of the results. As the threshold \(T\) decreases, the accuracy increases; however, the efficiency decreases. So the best threshold \(T\) for a particular application may be a tradeoff between efficiency and accuracy.

\subsection*{3.2.3 Criteria Based on Multiple Threshold Angles}

One threshold \(T\) can not capture the different effects of index nodes and leaf nodes, even if \(T\) is dependent on the weights of nodes, because discarding an index node will discard the subtree with the index node as root. It is worth using smaller threshold \(T_i\)
for index nodes to increase the accuracy. To keep the same accuracy, the threshold $T_i$ for leaf nodes can be increased. This results in smaller total number of node accesses since the number of index nodes is much smaller than the number of leaf nodes. In summary, one threshold per level of a tree is needed.

In order to use the criteria for node access presented in previous section, $\text{prob}(\theta,d)$ must be known. The formula for $\text{prob}(\theta,d)$ given in Section 2.3 is for uniform data. In real world, most data are clustered. It’s expensive to get accurate $\text{prob}(\theta,d)$ of clustered data. Also as the clustered data evolves, $\text{prob}(\theta,d)$ might change and it’s expensive to recompute $\text{prob}(\theta,d)$. Therefore, we need to simplify the criteria for node access. It can be simplified to one threshold angle per level of a tree, because the weights of nodes in same level of a tree are similar.

In AB-tree, we redefine several threshold angles for nodes with different range of data points, respectively, i.e. we redefine $n$ threshold angles $(\theta_1, \theta_2, \ldots, \theta_n)$ with corresponding weight range $(W_1, W_2, \ldots, W_{n+1})$. Then the criteria for node access based on multiple angle is

$$W_i < N < W_{i+1} \quad \text{and} \quad \theta > \theta_i,$$  \hspace{1cm} (3.2.2)  

where $n \geq i \geq 1$, $N$ is the weight of node, $\theta$ is the estimated angle. For a node with weight $N$ and $W_i < N < W_{i+1}$, where $n \geq i \geq 1$, if estimated angle $\theta > \theta_i$, then the search algorithm accesses the node, otherwise discards the node. The different ranges of data points correspond to different levels of tree, i.e. $(W_1, W_2)$ is for leaf nodes, $(W_2, W_3)$ is for direct parent nodes of leaf nodes, etc.. The predefined threshold angles are dependent on the desired accuracy of results. For a given required accuracy, we
calibrate the threshold angles for each individual database.

3.2.4 Criteria Based on Distance

In some case, the possible number of points inside the overlap area can not be correctly estimated based on the angles. One example of such a case is when the candidate set hypersphere falls inside of a bounding hypersphere. In this case, the intersecting angle between the candidate set hypersphere and the bounding hypersphere is not defined, but it is desirable to examine the hypersphere that overlap with the candidate set hypersphere. If the hypersphere is not examined, the quality of NN’s may suffer. This case is illustrated in Figure 3.2. Additional criteria based on distance is introduced to ensure that nodes with a high probability of containing NN’s to a given query point are not discarded by the heuristic. The criteria based on distance is that if the distance from query point Q to the center of bounding hypersphere is less than the distance from Q to the current $K^{th}$ NN, then the node is examined.

Figure 3.2: Special Case: Undefined Intersecting Angle
3.3 Search Algorithm for AB-tree

In the exact NN search algorithms, a node is always accessed if it is overlapping with the candidate set hypersphere, the hypersphere with the query point Q as its center and the distance from Q to the current $K^{th}$ NN as its radius. In our approximate NN searching algorithm, if the candidate set hypersphere overlaps with a node, the node is accessed only if it satisfies one of two conditions: (1) the radius of the candidate set hypersphere is greater than the distance from Q to the node parent’s hypersphere center, C, (C falls within the candidate set hypersphere), as described in Section 3.2.4, or (2) the estimated angle and the weight of the node satisfy the criteria for node access, as described in Section 3.2.3.

The following algorithm 3.3.1 utilizes the criteria for node access in the search for NN’s to a query point Q:

Algorithm 3.3.1 K Nearest Neighbor Search for AB-tree

```
{ 
  //This algorithm begins by invoking ExamineNode 
  //on the root node of the index tree. 
  ExamineNode(rootNode);
}
```

AB-tree is derived from SS-tree [79]. Besides the angle criteria, AB-tree uses a different ordering strategy than SS-tree as explained below. In SS-tree, the children of a node is sorted in ascending order by distance from the query point to the surfaces of bounding hyperspheres of the children in the first step. In the second step, for each child in the sorted list, it is accessed if the candidate set hypersphere overlaps with its
Algorithm 3.3.2 ExamineNode(Node)

```java
{
    if (Node is an index node)
    {
        Invoke ExamineIndexNode(Node);
    }
    else
    {
        Invoke ExamineLeafNode(Node);
    }
}
```

bounding hypersphere. The second step repeats until reaching a child such that its bounding hypersphere does not overlap with the candidate set hypersphere. Because the rest of the children remaining in the sorted list will not overlap with the candidate set hypersphere, these two steps guarantee that a child of this node is always accessed if it is overlapping with the candidate set hypersphere. These two steps are applied to all the nodes accessed during search. Therefore, SS-tree guarantees that a node is always accessed if it is overlapping with the candidate set hypersphere.

Instead of using distance to the surface, AB-tree uses the distance from the query point to the center of bounding hypersphere as shown in algorithm 3.3.3, because ordering by center gives better order with respect to the possible number of nodes inside the overlap area. When algorithm 3.3.3 terminates due to reaching a child such that its bounding hypersphere does not overlap with the candidate set hypersphere, it is possible some children in the remaining sorted list may overlap with the candidate set hypersphere. Thus, ordering by center does not guarantee that a child of this node is always accessed if it is overlapping with the candidate set hypersphere. For
Algorithm 3.3.3 ExamineIndexNode(Node)

{ 
1. Sort the children of Node in ascending order by distance from 
   the query point Q to the centers of the children; 
2. For each child in the sorted list (set CurrentChild to the 
current child) 
{ 
   if(K candidate neighbors have not yet been found) 
   { 
      Invoke ExamineNode(CurrentChild); 
   } 
   else if(the candidate set hypersphere (the hypersphere 
   centered at Q with the distance from Q to the 
current $k^{th}$ NN as the radius) overlaps with the 
bounding hypersphere of CurrentChild) 
   { 
      if( (the distance from Q to the center of CurrentChild 
is less than the distance from Q to the current 
$k^{th}$ NN) 
or 
      (The estimated angle $\theta$ between the candidate 
set hypersphere and the bounding hypersphere 
of CurrentChild and the weight $N$ of CurrentChild 
satisfy the criteria for node access: 
$W_i \leq N < W_{i+1}$ and $\theta > \theta_i$, where $1 \leq i \leq n$ ) 
   } 
   { 
      Invoke ExamineNode(CurrentChild); 
   } 
} 
else 
{ 
   break; 
} 
}
Algorithm 3.3.4 ExamineLeafNode(Node)
{
    For each point in the leaf(set CurrentPoint to the current point)
    {
        Compute the distance D from the query point Q to CurrentPoint;
        if(K candidate neighbors have not yet been found)
        {
            Add CurrentPoint to the candidate set of neighbors;
            Sort the candidate set;
        }
        else if(D is less than the distance from Q to the current \(k^{th}\) NN)
        {
            Remove the current \(k^{th}\) NN;
            Insert CurrentPoint into the candidate set of neighbors;
            Sort the candidate set;
        }
    }
}

exact nearest neighbor search, only ordering by surface can be used. However, even if the discarded children in the remaining sorted list overlap with the candidate set hypersphere in the ordering by center case, the possibility of nodes inside the overlap area is very small. Hence, there is no need to access these child nodes in approximate nearest neighbor search. For approximate nearest neighbor search, this ordering change alone can provide about 3 times speedup for searching a database of 800,000 64 dimension data points with 100 clusters while getting 100% accuracy. This 100% accuracy is not guaranteed. However, we always got near 100% accuracy when we only used ordering by center in our experiments.
3.4 Clustered AB-tree and Cached AB-tree

The AB-tree can handle very large databases. In certain range as the size of database increases, the performance gain of AB-tree over other existing methods increases. We discuss the effect of the size of database on performance in Section 5.2.1. However, for very huge databases, it will yield better performance if we can partition the dataset into many smaller subdatasets and only access a few subdatasets during search.

There are three ways to use AB-tree in this situation.

- Semantic Clustering: Partition dataset into many clusters using special clustering methods. Each cluster has special semantic meaning. Build an AB-tree for each cluster. During search only one AB-tree is accessed since each cluster has distinguished semantic meaning.

- Generic Clustering(Clustered AB-tree): Partition dataset into many clusters using generic clustering methods [32]. Each cluster does not have special semantic meaning. Build an AB-tree for each cluster. Multiple AB-trees are accessed during search.

- Cached AB-tree: Build one AB-tree for all. Treat subtrees at certain level of the AB-tree as clusters in Generic Clustering approach.

Semantic Clustering is similar to the previous standalone AB-tree case, and is not our focus here. In the following subsections we will discuss Clustered AB-tree and Cached AB-tree.

3.4.1 Clustering Approach

In order to create index structures for high dimensional databases, one approach is to partition the data set into groups using bounding hyper-polygons, such as hyperspheres and hyper-rectangles. The search algorithms for these indexing schemes generally utilize the partition information to avoid accessing those nodes that do not
overlap with the bounding hyper-polygons of the query (candidate set). Therefore, good partitioning is critical to the efficient operation of searching the index trees. In most high dimensional index trees, the partitioning is achieved mainly by splitting an overflowing node. There are several index schemes that use reinsert to refine the partitioning. However, they are greatly limited by several factors as described below.

Since data is inserted one by one, the partitioning is not globally optimized. Reinserting may help alleviate the problem but it cannot eliminate it altogether. This is because reinserts are done only for data of a full node. All data points in other nodes remain untouched. We have compared the performance of the proposed index tree with other index trees which use reinserts.

Another problem is that data partitioning by bounding hyper-polygons does not work well for higher level nodes. At leaf nodes, data partitioning works directly on the data points. However, in higher level nodes the partitioning is dependent on the partitioning of the nodes below it. As the level of a node gets higher and higher, the partitioning becomes less and less precise. However, it is the good partitioning of the higher level nodes that impacts the search performance of tree-based indexing because going to the wrong branches at higher levels will lead to accessing wrong nodes at the lower levels.

We propose a new approach to get better partitioning of the data set. This is done by exploiting a clustering technique. Here, we first cluster the whole data set into several groups. Then an index tree is created for each group. Conceptually, the index tree for the whole data set can be created by combining all the root nodes of these trees. This approach of creating multiple trees along with the proposed algorithm to
filter root nodes based on the angle property described before, provide an efficient search for high dimensional data.

We use the random sampling technique from CURE [32] for the basic clustering algorithm. In CURE, in order to capture the semantics of the data set, the size of a sample has to be large enough to guarantee some probability that at least one data point is drawn from each cluster. After the sampling phase, a hierarchical clustering algorithm called CURE is run over the sample. Because our focus is on getting better searching performance and not capturing the semantics of the data set, we do not need to draw a large sample as CURE does. In fact, our experimental results show that performance will remain almost the same after some threshold on the number of subtrees. We also do not need to use the hierarchical clustering technique used in CURE. Instead, we create AB-tree for each cluster.

3.4.2 Implementation of Clustered AB-tree

We use the random sampling technique described in [32] to cluster the data sets. In our method, a recursive phase is used to refine the result. In the first step, a random sample is drawn from the data set which serves as the initial centers of the groups. The second step is used to put the data points into the closest group based on their distance to the centers. In the third step, new group centers are computed by averaging the data points within each group. In the third step, the group with insufficient data points are removed to achieve higher tree utilization. These data points are inserted into other groups in the next recursive step. We repeat the second step and the third step until the group centers become stable. The detailed algorithm
is given in algorithm 3.4.1.

Algorithm 3.4.1 Clustering( D: A set of data points; N: number of subtrees )

{  
Randomly pick N data points, W₀, W₁, ..., Wₙ₋₁, from data set D;
Repeat{  
  Cᵢ = Wᵢ, for 0≤i≤N;
  reset cluster i, for 0≤i≤N;
  for each data point P in D {
    find Cᵢ, such that Cᵢ is the closest to P;
    Put P into cluster i;
  }
  calculate the new centroids Wᵢ for each cluster i;
  remove centroids Wᵢ if cluster i has insufficient members;
} Until group centers don't change (Wᵢ equals to Cᵢ, for 0≤i≤N);
}

In clustered AB-tree, we create a subtree for each partition of the cluster. We can consider the subtrees as direct children of a conceptual root node of the clustered AB-tree. Thus, same searching technique can be applied to each subtree as that of AB-tree in Section 3.3. The detailed k nearest neighbor search algorithm for clustered AB-tree is given in algorithm 3.4.2

Algorithm ExamineNode (Node) in algorithm 3.4.2 is same as algorithm 3.3.2 given in Section 3.3.
Algorithm 3.4.2 K Nearest Neighbor Search for Clustered AB-tree

{ 
1. Sort the subtrees in ascending order by distance from 
the query point Q to the roots of the subtrees;
2. For each root in the sorted list(set CurrentNode to the 
current root)
   {
      if(K candidate neighbors have not yet been found)
         {
            Invoke ExamineNode(CurrentNode);
         }
      else if(the candidate set hypersphere overlaps with the 
bounding hypersphere of CurrentNode)
         {
            if( (the distance from Q to the center of CurrentNode 
is less than the distance from Q to the current 
k_{th} NN) 
or
               (The estimated angle $\theta$ between the candidate 
set hypersphere and the bounding hypersphere 
of CurrentNode and the weight N of CurrentNode 
satisfy the criteria for node access: 
$W_i \leq N < W_{i+1}$ and $\theta > \theta_i$, where $1 \leq i \leq n$ )
            )
            {
               Invoke ExamineNode(CurrentNode);
            }
      }
      else 
      { 
         break;
      }
   }
}

3.4.3 Cached AB-tree

As shown later in Section 5.3, the clustered AB-tree outperforms AB-tree by up to 4 times. However, the major drawback of clustered AB-tree is that the whole dataset is required in advance in order to partition them into many well grouped clusters using the clustering algorithm. As databases evolve, the clusters may deteriorate and become unbalanced due to deletions and insertions.

Thus one central balanced AB-tree is more suitable for dynamic databases. For very huge databases, we can cache one whole level of nodes in a AB-tree in memory to reduce the number of node accesses. As mentioned in Section 1.2, main memory access structure is not suitable for huge databases. However, we can utilize the speed of main memory by caching since main memory is decreasing in cost. The hierarchical structure of AB-tree make it easy to cache the nodes of different levels due to different available cache sizes. As the cache size increases, the number of node accesses decreases. We will address the effect of cache size in Section 5.4.

Because a whole level of nodes is in memory, the cached nodes can be sorted in ascending order by distance from the query point Q to the center of the node without any node accesses. This ordering at the cached nodes of AB-tree provides better detailed ordering information than that in the same level of original AB-tree. Because this ordering is corresponding to each query point at the cached level, while ordering through travelling tree is only related to the local ordering in current subtree. This ordering makes the radius of candidate set decrease quicker than AB-tree does during search, thus increase the performance. This ordering along with the proposed
algorithm to filter nodes based on the angle property, provide more efficient search for huge high dimensional databases. We will discuss the search algorithm for Cached AB-tree in the following Section 3.4.4, and present the performance gain of cached AB-tree over AB-tree in Section 5.4

### 3.4.4 Implementation of Cached AB-tree

In cached AB-tree, we cache all the nodes at a particular level if the AB-tree by only storing the coordinate of center, radius and weight of each node. Which level to cache is dependent on the trade off between cache size and performance. The effect of cache size is addressed in Section 5.4. The subtree with a cached node as root is accessed only if the angle filtering condition is satisfied. The angle filtering condition is given in Algorithm 3.

The search algorithm for cached AB-tree has two steps. We first sort the cached nodes in ascending order by distance from the query point Q to the centers of the nodes. Then same searching technique as that of AB-tree given in Section 3.3 is applied. Algorithm 3.4.3 is the detailed approximate nearest neighbor search algorithm for cached AB-tree.

Algorithm ExamineNode (Node) in algorithm 3.4.3 is same as algorithm 3.3.2 given in Section 3.3.
Algorithm 3.4.3 K Nearest Neighbor Search for Cached AB-tree

{
    1. Sort the cached Nodes in ascending order by distance from
       the query point Q to the centers of the nodes;
    2. For each node in the sorted list(set CurrentNode to the
       current node)
       {
           if(K candidate neighbors have not yet been found)
           {
               Invoke ExamineNode(CurrentNode);
           }
           else if(the candidate set hypersphere overlaps with the
                   bounding hypersphere of CurrentNode)
           {
               if( (the distance from Q to the center of CurrentNode
                   is less than the distance from Q to the current
                   \( k^{th} \) NN)
                   or
                   (The estimated angle \( \theta \) between the candidate
                   set hypersphere and the bounding hypersphere
                   of CurrentNode and the weight \( N \) of CurrentNode
                   satisfy the criteria for node access:
                   \( W_i \leq N < W_{i+1} \) and \( \theta > \theta_i \), where \( 1 \leq i \leq n \) )
               }
               
               Invoke ExamineNode(CurrentNode);
           }
           else
           {
               break;
           }
       }
}
Chapter 4

Parameter Estimation

The basic idea of AB-tree is to reduce the number of node accesses during search using the conditions of angle filtering based on the Angle Property presented in Section 2.3. The parameter set (threshold angles, etc.) of the angle filtering conditions controls the accuracy of the results. As the threshold angles decrease, the accuracy increases; however, the efficiency decreases. So the best threshold angles for a particular application may be a tradeoff between efficiency and accuracy.

One important requirement in AB-tree is to derive the right parameter set for a desired accuracy. This chapter discusses how to generate a set of threshold angles automatically that maximizes the performance of AB-tree for a given database with some desired accuracy. First we present the need for parameter autotuning in Section 4.1, and the implementation of parameter autotuning in Section 4.2. Then we discuss the tradeoff between performance and accuracy in Section 4.3. Finally, we discuss the stability of the derived parameters with respect to database updates in Section 4.4 and maintenance cost in Section 4.5.
4.1 Introduction

As discussed in Section 3.2, we can use one threshold angle in AB-tree. The threshold angle was initialized to some value based on the desired accuracy and the probability of d-dimensional data points with the angle less than \( \theta \) \((0 \leq \theta \leq 180^\circ)\) discussed in Section 2.3. Then we used binary search to find the right threshold angle for the desired accuracy. Because the final threshold angle is found by the binary search, we initialize the threshold angle to 60\(^\circ\) degree based on the experience. We will discuss the effect of initial values of threshold angles on maintenance cost in Section 4.5. The binary search for one threshold angle is detailed as in algorithm 4.1.1.

We first tuned the threshold angle manually based on algorithm 4.1.1 **Binary-Search**(desiredAccuracy). With experience, we mixed different approaches such as linear interpolation and binary search to find the threshold angle quickly. We found that the AB-tree using one threshold angle for 90% accuracy outperforms the AB-tree with one threshold angle for 99% accuracy by several times. However, we can increase the performance of AB-tree further, by using multiple threshold angles as discussed in Section 3.2.3. This is because one threshold angle can not capture the different effects of the nodes in higher levels and those in lower levels in AB-tree. Table 4.1 shows the performance of AB-tree using different number of threshold angles for 800,000 16-dimensional database with 90% accuracy, except that the accuracy is 100% for 0 threshold angle (exact search). The results in Table 4.1 are the averages of the numbers of node accesses for 1000 random 20 nearest neighbor search queries. However, as the number of threshold angles increases, it is no longer practical to
Algorithm 4.1.1 BinarySearch(desiredAccuracy)

{
    Initialize{
        threshold angle: \( \theta_T = 60 \);
        maximum angle: \( \theta_{\text{max}} = 90 \);
        minimum angle: \( \theta_{\text{min}} = 0 \);
    }
    Repeat{
        Do 100 trials of k-nearest neighbor searches
        based on the threshold angle;
        Compute the accuracy of the results;
        If(computedAccuracy < desiredAccuracy)
        {
            \( \theta_T = (\theta_T + \theta_{\text{min}})/2 \);
            \( \theta_{\text{max}} = \theta_T \);
        }
        else
        {
            \( \theta_T = (\theta_T + \theta_{\text{max}})/2 \);
            \( \theta_{\text{min}} = \theta_T \);
        }
        } Until \( [(\theta_{\text{max}} - \theta_{\text{min}}) \leq 1] \).
    Return \( \theta_T \);
}
<table>
<thead>
<tr>
<th>Number of Threshold Angles</th>
<th>Number of Node Accesses</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (exact search)</td>
<td>1392</td>
<td>100%</td>
</tr>
<tr>
<td>1 (Granularity Level 1)</td>
<td>377</td>
<td>90%</td>
</tr>
<tr>
<td>3 (Granularity Level 2)</td>
<td>41</td>
<td>90%</td>
</tr>
<tr>
<td>one per node (Granularity Level 3)</td>
<td>136</td>
<td>90%</td>
</tr>
</tbody>
</table>

Table 4.1: Performance of AB-tree Using Different Number of Threshold Angles

derive all threshold angles manually. Thus, it’s necessary to be able to generate the
threshold angles automatically.

4.2 Parameter Autotuning

First issue in parameter autotuning of AB-tree is to determine the number of threshold
angles. We will discuss the following three levels of granularities for motivating our
approach to autotuning.

- Granularity Level 1: One threshold angle for all nodes
- Granularity Level 2: One threshold angle each index level of AB-tree
- Granularity Level 3: One threshold angle per node

Granularity Level 1 is too coarse. One threshold angle can not capture the dif-
ferent effects of the nodes in higher levels and those in lower levels in AB-tree. The
performance gain achieved is limited when Granularity Level 1 is used, as shown in
Table 4.1.

Granularity Level 3 is too fine grained. The total number of nodes in the AB-tree
for a huge database is very large. It is not feasible to maintain so many parameters
in a data structure separate from AB-tree, and therefore we maintain each threshold
angle with the corresponding node of AB-tree. This built-in threshold angle is de-
termined based on the local information of the corresponding node, and is updated
dynamically as the node is updated resulting from database updates. The advantage of this approach is that there is no need to retune the built-in angles separately as database changes. One problem of this built-in angle approach is to come up with a generic algorithm to compute the built-in threshold angles. We have used several heuristics to estimate the built-in angles, such as using the minimal angle of the angles between any two data points inside the node with respect to the center of the node. The performance of Granularity Level 3 is much better than that of Granularity Level 1, as shown in Table 4.1. However, one big disadvantage of Granularity Level 3 is that the accuracy and performance vary as database changes, and there is no way to retune the accuracy to meet different accuracy requirements since the accuracy for a given database is fixed once the algorithm for computing the built-in threshold angles is fixed. For example, two AB-trees are needed based on the different algorithms for computing the built-in threshold angles, if there are two different desired accuracy requirements for a same database. This can be easily achieved in Granularity Level 2 by using different sets of several threshold angles to meet different accuracy requirements with one AB-tree.

Granularity Level 2 has the advantage of Granularity Level 1 such that the accuracy is tunable. Granularity Level 2 also has the advantage of Granularity Level 3 such that nodes in different levels in AB-tree are using different threshold angles. Thus, we choose Granularity Level 2 due to its robustness and high performance.

We will discuss the complexity of parameter autotuning in Section 4.2.1, then present algorithm for parameter autotuning in Section 4.2.2.
4.2.1 Complexity

For a given AB-tree, we know the dimension of the feature vector, the data type of the feature vector, size of record stored in the leaves, the height, block size and average block utility of AB-tree. The maximum number of points stored in leaf nodes, \( Weight[0] \), is determined by equation 4.2.1.

\[
weight[0] = \frac{blockSize}{dimension \times typeSize + recordSize} \tag{4.2.1}
\]

Where \( typeSize \) is the size of one component of feature vector, and \( recordSize \) is the size of data record.

The \textit{fanout} of AB-tree is determined by equation 4.2.2.

\[
fanout = \frac{blockSize}{dimension \times typeSize + pointerSize} \tag{4.2.2}
\]

Taking into account the block utility of AB-tree, the maximum number of points stored in a node \( i \) levels above the leaf nodes, \( weight[i] \), is determined by equation 4.2.3,

\[
weight[i] = weight[i - 1] \times fanout \times blockUtility \tag{4.2.3}
\]

where \( weight[i - 1] \) is the maximum number of points stored in a node \( (i - 1) \) levels above the leave level.

Noting that the root of AB-tree is always accessed, the number of threshold angles is determined by equation 4.2.4.

\[
NumberOfThresholdAngles = heightOfTree - 1 \tag{4.2.4}
\]
where $heightOfTree$ is the height of AB-tree, and is roughly estimated by equation 4.2.5.

\[
heightOfTree \approx \log_{\text{fanout}} \text{numberOfPoints}
\]  

(4.2.5)

where $\text{numberOfPoints}$ is the total number of data points in the AB-tree.

The possible value of a threshold angle is in the range of $[0, 90]$. For simplicity, assume a threshold angle is an integer. Then it needs $(90^\text{NumberOfThresholdAngles})$ number of iterations to find the global optimal threshold angles such that the performance is best for the requested accuracy. Each iteration requires 100 trials of approximate k-nearest neighbor search. Based on the complexity, it's infeasible to find the globally optimal angles. Our approach is to find the best threshold angle for the leaf nodes first using algorithm 4.1.1 BinarySearch(desiredAccuracy) as discussed in Section 4.1, then find the best threshold angle for the parent nodes of leaf nodes, and so on until all the threshold angles for the required accuracy are found. Therefore, it needs only $(\text{NumberOfThresholdAngles} \times \log_2 90)$ number of iterations. Our approach does not guarantee finding globally optimal threshold angles. It does find good threshold angles which yield high performance for a desired accuracy.

For clustered AB-tree and cached AB-tree, the parameter tuning is almost the same as that of AB-tree, except that the $\text{NumberOfThresholdAngles}$ is smaller. For clustered AB-tree, the height of any subtree is smaller than that of AB-tree, because the sizes of subtrees are much smaller than that of AB-tree. For cached AB-tree, $\text{NumberOfThresholdAngles}$ is equal to the height of AB-tree minus the cache level. Thus the complexity and cost of finding threshold angles for clustered AB-tree
and cached AB-tree is smaller than that of AB-tree. We will discuss the maintenance cost for AB-tree in Section 4.5

### 4.2.2 Algorithm for Parameter Autotuning

To tune the parameters, an angle parameter file is initialized based on the information of AB-tree in the first step. In the second step, k nearest neighbor search is done 100 times based on the angle parameter file. In the third step, analyze the results and update the angle parameter file based on the resulting accuracy and the required accuracy. We repeat the second step and the third step until all the threshold angles are found. The detailed algorithm is given in algorithm 4.2.1.

**Algorithm 4.2.1 ParameterAutotuning(requiredAccuracy)**

```
{ 
    InitializeAngleParameterFile();
    GenerateAngleParameterFile(requiredAccuracy);
}
```

We can reduce the calibration cost by reusing the threshold angles for higher accuracy as the initial values of threshold angles for lower accuracy, as shown in algorithm 4.2.2 InitializeAngleParameterFile(). Experiments indicates that accuracy decreases monotonically with the increase of threshold angles. Even if the monotonicity between accuracy and threshold angles could not hold for some very rare cases, algorithm 4.2.2 will generate threshold angles at similar cost due to the nature of binary search, except that the threshold angles may not be good. However,
Algorithm 4.2.2 InitializeAngleParameterFile()

{  
    if (the angle parameter file for accuracy higher than the required accuracy exits)
    {
        // reuse the angle parameter file for higher accuracy
        // do nothing here
    }
    else
    {
        // Initialize the angle parameter file
        weight[0] = blockSize/(dimension*typeSize+recordSize);
        fanout = blockSize/(dimension*typeSize+pointerSize);
        $\theta_T^0 = 60$;
        for (i=1; i<height-1; i++)
        {
            weight[i] = weight[i-1]*fanout*blockUtility;
            $\theta_T^i = 0$;
        }
        write threshold angles and weights to the angle parameter file;
    }
}
Algorithm 4.2.3 GenerateAngleParameterFile(requiredAccuracy)

{ 
   Initialize{ 
      numberOfThresholdAngles = height - 1; 
      maximum angle: $\theta_{max} = 90$; 
      minimum angle: $\theta_{min} = 0$; 
      read threshold angles from the angle parameter file; 
   } 
   for(J=0; J< numberOfThresholdAngles; J++) 
   { 
      //Use binary search to find the $J^{th}$ threshold angle -- $\theta_T^J$ 
      //with the desired accuracy 
      desiredAccuracy=(requiredAccuracy*numberOfThresholdAngles-J); 
      Repeat{ 
         Do 100 trials of k-nearest neighbor searches 
         based on the threshold angle; 
         Compute the accuracy of the results; 
         If( computedAccuracy < desiredAccuracy ) 
         { 
            $\theta_T^J = (\theta_T^J + \theta_{min})/2$; 
            $\theta_{max} = \theta_T^J$; 
         } 
         else 
         { 
            $\theta_T^J = (\theta_T^J + \theta_{max})/2$; 
            $\theta_{min} = \theta_T^J$; 
         } 
      } Until $[(\theta_{max} - \theta_{min}) \leq 1]$.
      $\theta_{max} = \theta_T^J$; 
      $\theta_{min} = 0$; 
   }
}

69
this never happens in all our experiments. Figure 4.2 shows the monotonical relationship between accuracy and the threshold angle for leaf nodes for 16 dimensions. We will discuss the effect of parameter reuse on maintenance cost in Section 4.5.1.

The threshold angles in higher levels of AB-tree should be less than those in the lower levels. Because the number of points inside a node in higher levels is larger than that in lower levels, the possible number of points inside the overlap area between the candidate set hypershare and the bounding hypershare of a node in higher levels is higher than that in lower levels with a same threshold angle as discussed in Section 3.2.2. According to the same criteria for node access shown in equation 3.2.1, the larger the number of points inside a node, the smaller the threshold angle. Therefore, the initial maximum angle for level J is set to the computed threshold angle for level (J-1), as shown in algorithm 4.2.3 GenerateAngleParameterFile(requiredAccuracy) on page 69. This reduces the number of iterations needed to computed the threshold angle for level J. This reduction in the number of iterations is more significant as the number of levels increases.

4.3 Tradeoff Between Performance and Accuracy

In this section we demonstrate the effect of parameter tuning on performance and accuracy. Tests were performed on clustered data sets that were created for 8, 16, 32, 64, and 128 dimensions. Each data set consisted of 100,000 data points that were partitioned into 100 clusters. Thus, each cluster contained 1000 points. The points for the clustered data sets were generated in the range [0,1) for each dimension. In order to create a cluster, a random point is generated in the space. This point serves
as the cluster center. Next, a random range for each dimension is produced for the cluster. Points are then generated uniformly inside the hyperrectangle.

![Graph showing accuracy vs. number of node accesses](image)

**Figure 4.1: Accuracy vs. Number of Node Accesses**

For each data set, the algorithms search for the 20 nearest neighbors for each of 1000 query points. The query points were taken from within the data set. The number of disk accesses was aggregated for 1000 queries, and the results were averaged over them. All experimental results are based on 20 nearest neighbor search.

Figure 4.1 shows the numbers of node access required during 20 nearest neighbor search for different dimensions with various accuracies. For all dimensions the number of node accesses decreases dramatically as the accuracy decreases. When accuracy decreases from 100% to 70%, the number of node accesses for 20 nearest neighbor search decreases from 86 to 10 for 8 dimension, from 332 to 14 for 16 dimensions, and from 698 to 33 for 32 dimension. For higher dimension, the minimum achievable accuracy is larger. This is due to the first distance condition in the search algorithm for AB-tree as presented in Section 3.3. Figure 4.1 demonstrates that it is worthwhile
to trade accuracy for performance of AB-tree. In chapter 5, we will discuss the performance of AB-tree under different conditions such as block size, database size, caching, and other, with 90% accuracy. It should be noted that the performance of AB-tree will be much better if we use smaller accuracy.

4.4 Stability of Parameters

As mentioned before, we need to calibrate the threshold angles for each individual database based on the desired accuracy. Thus, the important issue here is that the threshold angles be stable enough to handle a reasonable change of the database. We test the stability of the threshold angles on 32-dimensional clustered database. The threshold angles are calibrated on a database of size 100,000. The algorithms search for the 20 nearest neighbors for each of 1000 query points. The query points were randomly taken from within the data set. The number of disk accesses was aggregated for 1000 queries, and the results were averaged over them. We then add more data to the database, and repeat the same search experiment using the same calibrated threshold angles. As shown in Table 4.2, the accuracy varies less than 2% when database size increases by 15%.

<table>
<thead>
<tr>
<th>Data Points</th>
<th>32-dimensional clustered database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy (%)</td>
<td>90 90 90 89</td>
</tr>
<tr>
<td>ε</td>
<td>0.014 0.014 0.014 0.016</td>
</tr>
<tr>
<td>Speedup</td>
<td>21 20 22 22</td>
</tr>
</tbody>
</table>

Table 4.2: Performance for Varying Database Sizes
4.5 Maintenance Cost

After databases evolve, the threshold angles may not meet the requirement for performance and accuracy. In order to maintain the right threshold angles, we need to check the performance and accuracy of AB-tree periodically. The checking period depends on the changing frequency of the database. If the resulting accuracy is smaller than the required accuracy or the performance decreases a lot compared to the original performance of AB-tree, the angle parameter file is not effective any more. In this situation, we need to recalibrate the threshold angles for desired performance and accuracy. The cost of calibration is determined by equation 4.5.1 as discussed in Section 4.2.1.

\[
Cost = (NumberOfThresholdAngles \times \log_2 90)(iterations) \quad (4.5.1)
\]

Each iteration requires 100 trials of approximate k-nearest neighbor search. Because the threshold angles in higher level of AB-tree should be less than those in the lower level, the actual number of iterations is always smaller than that in equation 4.5.1.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Iterations</td>
<td>17</td>
<td>17</td>
<td>22</td>
<td>25</td>
</tr>
<tr>
<td>Time(S)</td>
<td>512</td>
<td>845</td>
<td>2078</td>
<td>8250</td>
</tr>
</tbody>
</table>

Table 4.3: Calibration Cost (Database Size: 800,000)

Table 4.3 shows the calibration cost for four clustered data sets with 90% accuracy requirement. Four clustered data sets were created for 8, 16, 32 and 64 dimensions. Each data set consisted of 800,000 data points that were partitioned into 100 clusters. The block size is 8k. The 8k block size is not good for dimensions higher than 64.
Thus the calibration cost for 64 dimension is significantly higher, but still reasonable.

If we increase the block size for 64 dimension, the cost decreases. We will discuss the effect of block size on the performance of AB-tree in Section 5.2.

<table>
<thead>
<tr>
<th>$\theta^0_T$</th>
<th>$\theta^1_T$</th>
<th>$\theta^2_T$</th>
<th>Iterations</th>
<th>Time(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>0</td>
<td>0</td>
<td>17</td>
<td>885</td>
</tr>
<tr>
<td>45</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>823</td>
</tr>
<tr>
<td>60</td>
<td>0</td>
<td>0</td>
<td>17</td>
<td>845</td>
</tr>
<tr>
<td>40</td>
<td>24</td>
<td>17</td>
<td>15</td>
<td>314</td>
</tr>
</tbody>
</table>

Table 4.4: Effect of Initial Values of Threshold Angles

Table 4.4 shows the effect of initial values of threshold angles on calibration cost for 800,000 16-dimensional database with 90% accuracy requirement. The calibration cost for the initial values ($\theta^0_T = 30, \theta^1_T = 0, \theta^2_T = 0$), is similar to those for ($\theta^0_T = 45, \theta^1_T = 0, \theta^2_T = 0$) and ($\theta^0_T = 60, \theta^1_T = 0, \theta^2_T = 0$), where $\theta^0_T$ is the threshold angle for leaf level, $\theta^1_T$ for parent of leaf, and $\theta^2_T$ for next level above parent of leaf. For 16-dimension with 90% accuracy requirement, the final value of $\theta^0_T$ is between (30, 60). Thus, the cost for the initial values ($\theta^0_T = 45, \theta^1_T = 0, \theta^2_T = 0$) is slightly smaller than those for ($\theta^0_T = 30, \theta^1_T = 0, \theta^2_T = 0$) and ($\theta^0_T = 60, \theta^1_T = 0, \theta^2_T = 0$) as shown in Table 4.4. If we could know the final value of $\theta^0_T$, we could set the initial value of $\theta^0_T$ to $\theta^0_{initial}$ such that the final value of $\theta^0_T$ is in one of $[0, \theta^0_{initial}]$ and $[\theta^0_{initial}, 90]$ whichever has shorter range, and then the number of iterations and cost for calibration would be smaller. Based on experiments we have found that the final value of $\theta^0_T$ is usually larger than 60 as dimension increases and/or the required accuracy decreases. Therefore, we choose 60 as the initial value of $\theta^0_T$ in our approach because it might need slightly smaller calibration cost in some cases.
4.5.1 Effect of Parameter Reuse on Maintenance Cost

We can further reduce the calibration cost dramatically with parameter reuse. Table 4.5 shows the detailed calibration cost for 800,000 16-dimensional database with 90% accuracy requirement using regular initial values ($\theta_0^0 = 60$, $\theta_1^0 = 0$, $\theta_2^0 = 0$). Table 4.6 shows the detailed calibration cost for the same 16 dimensional database with parameter reuse. As shown in Table 4.6, we use the threshold angles ($\theta_0^0 = 40$, $\theta_1^1 = 24$, $\theta_2^1 = 17$) for 95% accuracy as the initial values of threshold angles for 90% accuracy. The number of iterations is reduced from 17 to 15 by reusing the threshold angles for higher accuracy. Furthermore, the number of node accesses of first several iterations for calibration without parameter reuse as shown in Table 4.5 is about 10 times larger than those for calibration with parameter reuse as shown in Table 4.6, because the proper initial values of threshold angles for high levels filter out a lot of unnecessary node accesses. By reusing the threshold angles for 95% accuracy as the initial values of threshold angles for 90% accuracy, the total run time of calibration is reduced from 845 seconds to 314 seconds for the 16 dimensional database, as shown in Table 4.4. In general, we can reduce the calibration cost dramatically by reusing the threshold angles for higher accuracy as the initial values of threshold angles for lower accuracy.

As shown in Table 4.2, the accuracy varies less than 2% for up to 15% change of the original database. Figure 4.2 further demonstrates that the threshold angle $\theta_T^0$ for some accuracy for 1,000,000 16-dimensional database is almost the same as that for the same accuracy for 800,000 16-dimensional database. Therefore, for databases with
small changes, we may use the threshold angles for higher accuracy for the original database as the initial values of threshold angles for lower accuracy for the evolved database to reduce the calibration cost significantly.

![Diagram showing the relationship between accuracy and threshold angle for varying database sizes.](image)

Figure 4.2: Parameters For Varying Database Sizes (16 dimensions)

### 4.5.2 Maintenance Cost for Clustered AB-Tree and Cached AB-Tree

The maintenance cost for clustered AB-tree and cached AB-tree includes two parts: maintaining clusters or cache and maintaining threshold angles. If database changes little, then only threshold angles need to be updated. The maintenance cost of threshold angles for clustered AB-tree and cached AB-tree is smaller than that of AB-tree as discussed in Section 4.2.1. However, for clustered AB-tree the clusters may become unbalanced after database evolve heavily, i.e. some subtrees may contain too few data points while other subtrees contain too many data points. Therefore, subtrees with too few data points are deleted and the data points are reinserted, and each subtree with too many data points is split into two new subtrees. For cached AB-tree, the
\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Iteration & $\theta_0^T$ & $\theta_1^T$ & $\theta_2^T$ & Accuracy(\%) & Number of Node Accesses \\
\hline
1 & 60 & 0 & 0 & 75.7 & 373 \\
2 & 30 & 0 & 0 & 98.8 & 515 \\
3 & 45 & 0 & 0 & 93.5 & 392 \\
4 & 52.5 & 0 & 0 & 85.4 & 377 \\
5 & 48.8 & 0 & 0 & 90.5 & 382 \\
6 & 46.9 & 0 & 0 & 92.1 & 386 \\
7 & 46.9 & 23.4 & 0 & 91.1 & 54.2 \\
8 & 46.9 & 35.2 & 0 & 86.1 & 36.8 \\
9 & 46.9 & 29.3 & 0 & 88.2 & 40.4 \\
10 & 46.9 & 26.4 & 0 & 90 & 44.6 \\
11 & 46.9 & 24.9 & 0 & 90.6 & 48.5 \\
12 & 46.9 & 24.9 & 12.5 & 90.6 & 47.9 \\
13 & 46.9 & 24.9 & 18.7 & 90.2 & 42.6 \\
14 & 46.9 & 24.9 & 21.8 & 85.7 & 39.1 \\
15 & 46.9 & 24.9 & 20.2 & 88.4 & 41 \\
16 & 46.9 & 24.9 & 19.5 & 89.1 & 41.7 \\
17 & 46.9 & 24.9 & 19.1 & 89.4 & 42.2 \\
\hline
\end{tabular}
\caption{Detailed Calibration Cost Without Parameter Reuse}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Iteration & $\theta_0^T$ & $\theta_1^T$ & $\theta_2^T$ & Accuracy(\%) & Number of Node Accesses \\
\hline
1 & 40 & 24 & 17 & 95.3 & 71.8 \\
2 & 65 & 24 & 17 & 69.5 & 34.4 \\
3 & 52.5 & 24 & 17 & 84.2 & 38.4 \\
4 & 46.3 & 24 & 17 & 91.1 & 48.67 \\
5 & 43.1 & 24 & 17 & 93.6 & 57.9 \\
6 & 44.7 & 24 & 17 & 92.3 & 53.1 \\
7 & 44.7 & 34.3 & 17 & 88 & 38.4 \\
8 & 44.7 & 29.2 & 17 & 89.5 & 42.1 \\
9 & 44.7 & 26.6 & 17 & 91.2 & 45.7 \\
10 & 44.7 & 27.9 & 17 & 90.3 & 43.6 \\
11 & 44.7 & 27.2 & 17 & 90.8 & 44.4 \\
12 & 44.7 & 27.2 & 22.1 & 86.1 & 39.2 \\
13 & 44.7 & 27.2 & 19.6 & 89.7 & 41.8 \\
14 & 44.7 & 27.2 & 20.8 & 88.3 & 40.7 \\
15 & 44.7 & 27.2 & 20.2 & 89.6 & 41.6 \\
\hline
\end{tabular}
\caption{Detailed Calibration Cost With Parameter Reuse}
\end{table}
cache is updated in memory by cached AB-tree as database updates.
Chapter 5
Experimental Results

The performance of AB-tree has been compared with that of SS-tree [79] and VA-File [77]. All tests were done on Sun Sparc workstations running Solaris 5.6. All index nodes and leaf nodes in the trees were set to 8K to conform to the disk block size of the operating system, except that in Section 5.2.4 we use different block sizes to study the effect of block size on the performance of AB-tree. Also the minimum utilization factor of the index trees that were created was set to 40% while the reinsert fraction parameter was set to 30%

Tests were performed on three separate types of data: uniform data sets, clustered data sets, and a real data set. For each data set, the algorithms search for the 20 nearest neighbors for each of 1000 query points. The query points were taken from within the data set. The number of disk accesses was aggregated for 1000 queries, and the results were averaged over them. The NN accuracy or percentage of the real NN’s was set to above 90%. The performance of AB-tree will be much better if we use smaller accuracy as discussed in Section 4.3.

This chapter is organized as follows: Section 5.1 presents the performance of AB-tree on two different types of synthetic data: uniform data and clustered data;
Section 5.2 discusses the performance of AB-tree under different parameters such as database size, dimension, number of nearest neighbors, and block size; Section 5.3 and Section 5.4 present the performance of two improved variants of AB-tree: clustered AB-tree and cached AB-tree; Section 5.5 describes the experimental results on real data; and Section 5.6 discusses approximate nearest neighbor search.

5.1 Synthetic Data

In this section, we discuss the performance of AB-tree on two different types of synthetic data: uniform data and clustered data. We have used normalized I/O cost [16] to compare the performance. The normalized I/O cost is the ratio of the average number of disk accesses required to execute a query using the indexing technique to the number of disk accesses to execute a sequential scan. The latter is computed by \( \text{NumberOfDataPoints} \times \text{Dimensionality} \times \text{sizeof(DataType)} / \text{PageSize} \). The normalized I/O cost of sequential scan is 0.1 instead of 1.0 based on the assumption that sequential disk accesses are about 10 times faster than random accesses. Same assumption is used to calculate the number of disk accesses in the filtering step of VA-File. For any index mechanism, a normalized I/O cost of more than 0.1 indicates worse I/O performance compared to sequential scan.

5.1.1 Uniform Data

Uniform data sets were created for 8, 16, and 32 dimensions. Each data set consisted of 100,000 data points that were randomly generated in the range \([0,1]\) for each dimension.
Figure 5.1: Uniform Data

Figure 5.1 shows the normalized I/O cost for 20 NN searches. As shown in Figure 5.1, the normalized I/O cost increases dramatically as the dimension increases, and reaches 4.3 in 32 dimension for the SS-tree. This means that SS-tree indexing is not very effective and failed to appropriately group points into neighborhoods. On the other hand, the normalized I/O cost is less than 0.1 in 8 dimension for the AB-tree while maintaining 90% accuracy. The AB-tree outperforms the SS-tree by a factor of up to 10. However, the performance of AB-tree is worse than that of VA-File, the best known sequential access method.

The authors of almost all of the papers mentioned in page 2 discuss experiments with uniform data sets. It has been shown in these papers that NN searching algorithms perform very poorly for uniform data as dimensionality increases. It has been shown that an algorithm must search almost the entire database in order to find NNs to a given query point. In such cases, it may be better to do a sequential search instead of using an indexing mechanism. Weber, et. al. [77] have shown that under
the assumption of uniformity and independence, and when the number of dimensions is sufficiently high, sequential scan may outperform indexing schemes for exact NN search queries.

5.1.2 Clustered Data

Clustered data sets were created for 16, 32 and 64 dimensions. Each data set consisted of 800,000 data points that were partitioned into 100 clusters. Thus, each cluster contained 8000 points. The points for the clustered data sets were generated in the range [0,1) for each dimension. In order to create a cluster, a random point is generated in the space. This point serves as the cluster center. Next, a random range for each dimension is produced for the cluster. Points are then generated uniformly inside the hyperrectangle.

![Clustered Data Graph](image)

Figure 5.2: Clustered Data (database size: 800,000)

The normalized I/O cost for 20 NN searches is shown in Figure 5.2. Figure 5.2 shows that the normalized I/O cost increases dramatically for SS-tree as dimension
increases, while it grows very slowly for the AB-tree. AB-tree outperforms SS-tree by a factor of 86 in 64 dimension and VA-File by up to 7.8 times.

5.2 Performance Issues

In this section, we investigate some factors, such as size of database, dimension, number of nearest neighbors, and block size that have important impacts on the performance of AB-tree using clustered data sets.

5.2.1 Effect of Database Size

![Graph showing speedup against accuracy for different database sizes](image)

Figure 5.3: Effect of Database Size With Various Accuracies

The performance gain of AB-tree is dependent on the size of the database. The larger the database size, the better the performance. As shown in Figure 5.3, for 64 dimensional database, as the number of data points increases from 10,000 to 500,000, the speedup of AB-tree over SS-tree increases from 34 to 98 times for 90% accuracy.

The Figure 5.4 shows the performance of AB-tree and VA-file under various database sizes. As the size of database increases from 50,000 to 200000, the number
of disk accesses for VA-file increases almost linearly due to its sequential scan of the approximation file in the filtering step, while that for AB-tree remains almost the same.

5.2.2 Effect of Dimension

As shown in Figure 5.5, the speedup of AB-tree over SS-tree increases dramatically
as the accuracy decreases. Also the performance gain of AB-tree over SS-tree increases as dimension increases. For 90% accuracy, the speedup of AB-tree over SS-tree is 10 for 8 dimension, 24 for 16 dimension, 21 for 32 dimension, 55 for 64 dimension, and 72 for 128 dimension.

5.2.3 Effect of Number of Nearest Neighbors

We also performed tests with varying number of NN’s (5, 10, 20, and 50 NN’s) using the 100,000 64-dimensional data set. 1000 queries were performed for each of 5, 10, 20, and 50 NN’s. The results were shown in Figure 5.6. The percentage of nodes accessed increases as the number of nearest neighbor increases for the SS-tree, while it remains almost constant for the AB-tree. The search time increases dramatically as the number of nearest neighbor increases for the SS-tree, while it grows very slowly for the AB-tree. Both the decrease in the percentage of nodes accessed and the speed-up in the search time improve as the number of nearest neighbor increases.

(a) node accesses vs. KNN  
(b) Time vs. KNN

Figure 5.6: Varying the Number of NN’s
5.2.4 Effect of Block Size

In this experiment, we collect the performance data for 64 dimension with 8k, 16k, and 32k block size. The fanout is 31 for 8k block size, 62 for 16k and 124 for 32k. We assume that the actual time of fetch a large disk block is determined by the seeking time and the transferring time is very small.

![Graph showing the relationship between Fanout and Number of Node Accesses.](image)

Figure 5.7: Effect of block size

As shown in Figure 5.7, the number of node accesses drops dramatically when block size goes from 8k to 16k. After 16k, it remains roughly the same.

5.3 Performance Gain of Clustered AB-tree

For clustered AB-tree, we investigate the effect of number of clusters on the performance of clustered AB-tree, and also compare its performance with that of AB-tree.
5.3.1 Performance Sensitivity and the Number of Clusters

As described earlier, clustered AB-tree partitions the data set into certain number of clusters and then creates a tree for each cluster. Because of better clustering, angle property can be effectively utilized to eliminate most of the subtrees to be searched. In order to use clustered AB-tree, the number of subtrees needs to be determined first. In real world, the actual number of clusters in a data set may not be known. Therefore, we designed experiments to study the sensitivity of the performance of clustered AB-tree on the number of subtrees. In the experiment, we created three separate data sets with 100, 200 and 400 actual clusters. Since the clustered data set is created in a controlled way, we are able to compare the performance for data sets with varying number of subtrees with that of the perfect subtrees (perfect subtrees mean one subtree for each actual cluster). For each data set, we measure the search performance of the clustered AB-trees with perfect, 250, 500 and 1000 subtrees.

The Figure 5.8 shows that the number of subtrees has very small effect on the
performance. In the experiment with 400 actual clusters, the performance of 250 subtrees, which is less than the number of actual clusters, is as good as that of 400 subtrees. Since the performance is stable for varying number of subtrees, we have used 250 subtrees for the following experiments.

5.3.2 Performance Comparison of Clustered AB-tree with Standard AB-tree, SS-tree and VA-file

![Performance Comparison for Different Number of Clusters](image)

Figure 5.9: Performance Comparison for Different Number of Clusters

In this experiment, we compared the performance of clustered AB-tree with those of standard AB-tree, SS-tree and VA-file. Two types of performance comparisons have been done. Figure 5.9 gives the performance for different number of clusters while Figure 5.10 gives the performance for varying number of dimensions. Both Figure 5.9 and 5.10 show that clustered AB-tree outperforms standard AB-tree, SS-tree and VA file by an order of magnitude. For example, in 64 dimensions and 400 clusters the clustered AB-tree outperforms standard AB-tree by a factor of up to 4, SS-tree by up to 400 times and VA-file by up to 14 times.
Figure 5.10: Performance Comparison for Varying Dimensions

5.4 Performance Gain of Cached AB-tree

For cached AB-tree, our experiment is focused on the effect of cache size and the performance gain of cached AB-tree over AB-tree.

Tests were performed on two clustered data sets, one 32 dimensional and the other 64 dimensional. Each data set consisted of 800,000 data points that were partitioned into 100 clusters.

5.4.1 Effect of Cache Size

As shown in Figure 5.11, the number of node accesses decreases with the increasing number of cache level. The height of the 32-dimensional AB-tree with 800,000 data points is 5. Thus the maximum cache level is 4. The cache level 1 is corresponding to caching only the root, which is original AB-tree without caching. As the cache level increases from 1 to 4, the number of node accesses decreases from 113 to 55, and the speedup over SS-tree increases from 66 to 135 times.
As shown in Figure 5.12, the maximum cache level is 5, since the height of the 64-dimensional AB-tree with 800,000 data points is 6. As the cache level increases from 1 to 5, the number of node accesses decrease from 301 to 129, the speedup over SS-tree increases from 58 to 128.

The caching approach can also be applied to SS-tree. However, As shown in Table 5.1, the performance of SS-tree is not good even with caching. For example, even with the maximum cache level, the numbers of node accesses for both databases are around 3000. Only combining caching with angle filter condition can provide an effective search scheme.

The required cache sizes for different cache levels are shown in Table 5.1. For 32d database, the cache size is less than 205kB for up to cache level 3, and the cache size is about 4.3MB for cache level 4. For 64d database, the cache size is less than 160kB for up to cache level 3, and the cache size is about 1.6MB for cache level 4, 18MB for level 5. Thus the memory size requirement is not a problem for our caching
Figure 5.12: Performance of Cached AB-tree for 800,000 64-dimensional Database

Table 5.1: Number of Node Accesses for Different Cache Level

<table>
<thead>
<tr>
<th>Cache Level</th>
<th>32d</th>
<th>64d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AB-tree</td>
<td>SS-tree</td>
</tr>
<tr>
<td>1</td>
<td>113.47</td>
<td>7512</td>
</tr>
<tr>
<td>2</td>
<td>83.21</td>
<td>4254</td>
</tr>
<tr>
<td>3</td>
<td>73.12</td>
<td>4177</td>
</tr>
<tr>
<td>4</td>
<td>55</td>
<td>2884</td>
</tr>
<tr>
<td>5</td>
<td>N/A</td>
<td>128</td>
</tr>
</tbody>
</table>

approach. However, as the caching level increases, the total number of nodes cached in memory increases dramatically. This increases the cpu time in first sorting step dramatically. The balance between number of node accesses in step two and the cpu time in sorting phase of algorithm 3.4.3 presented in Section 3.4.4 shows for 32d dataset the best performance is achieved with cache level 3, while for 64d dataset the best performance is achieved with cache level 4.

5.4.2 Performance Gain of Cached AB-tree Over AB-tree

Figure 5.13 shows the performance gain of cached AB-tree over AB-tree. The cached AB-tree outperforms AB-tree by a factor of 2 with cache level 3 for 32-dimensional
Figure 5.13: Performance Gain of Cached AB-tree over AB-tree

database and cache level 4 for 64-dimensional database. The performance of cached AB-tree is not as good as that of clustered AB-tree shown in Section 5.3.2. This is because the dataset in clustered AB-tree is partitioned into many small clusters based on the global information. Each cluster in clustered AB-tree is well grouped than that in cached AB-tree. However, this advantage of clustered AB-tree is also its disadvantage. As database evolves, the existing clusters may not be the good clusters for the changed database, and the performance may deteriorate. Therefore, clustered AB-tree is suitable for less dynamic databases, while cached AB-tree is more suitable for dynamic databases.

5.5 Performance Comparison Using Real Data

The real data set consists of 44,300 32-dimensional vectors representing color histogram of images obtained from IRIS-USA. The resulting normalized I/O cost for 20 nearest neighbor search were shown in Figure 5.14. AB-tree outperforms SS-tree by
Figure 5.14: Real Data: 44,300 32-dimensional color histogram feature vectors

a factor of 9.3 and VA-File by up to 3.1 times.

The performance improvement in real data set is not as significant as that in clustered data due to the data size. Because the number of disk accesses for VA-file is proportional to the data size, while AB-tree is insensitive to the data size. As data size increases, the performance gain for AB-tree over the competitors also increases.

5.6 Discussions on Approximate Nearest Neighbor Search

There has been a lot of recent work done on approximate nearest neighbor searches in computational geometry [2, 17, 43, 48, 38]. A point $P$ is a $(1 + \epsilon)$-approximate $j^{th}$ nearest neighbor to a point $Q$ if its distance from $Q$ is a factor of at most $(1 + \epsilon)$ times the distance to $Q$’s true $j^{th}$ nearest neighbor. An answer to the approximate $k$-nearest neighbors query is a sequence of $k$ distinct data points $P_1$, $P_2$, ..., $P_k$, such that $P_j$ is a $(1 + \epsilon)$-approximate $j^{th}$ nearest neighbor to $\theta$, for $k \geq j \geq 1$. Most of these works
<table>
<thead>
<tr>
<th>Dimension</th>
<th>Uniform Data</th>
<th>Clusted Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>0.035</td>
<td>0.0217</td>
</tr>
</tbody>
</table>

Table 5.2: Relative Error Bound in Distance for 90% Accuracy

used data structures that are stored in main memory. Thus the query time consists of mostly the computation cost. Our focus here is on approximate nearest neighbor search in databases. Thus I/O cost is the primary source of query time. We introduce the approximate nearest neighbor search based on the accuracy the percentage of the true nearest neighbors in the answer. To be comparable to $(1 + \epsilon)$-approximate K-nearest neighbor, we measure the $\epsilon$'s of 1000 20-$NN$ queries on the previous uniform and clustered databases. As shown in Table 5.2, for 90% accuracy, $\epsilon$ is less than 0.035 in 8 dimensions, decreases as dimension increases, and is less than 0.012 in 64 dimensions for clustered data. As mentioned in the Section 1.4, the approximate nearest neighbor search with 90% accuracy and the relative error bound in distance $\epsilon < 0.035$ is just as good as the exact nearest neighbor search.

Figure 5.15 and 5.16 show sample retrieval results using exact and approximate K-$NN$ search based on color histogram, respectively. Ten images were requested (k=10). The first image is also the query image. With 90% accuracy, the first 9 images are exactly the same. So the approximate is as good as the exact.
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. image 671900</td>
<td>2. image b35859</td>
<td>3. image 145700</td>
<td>4 image b43001</td>
</tr>
<tr>
<td>5. image b11146</td>
<td>6. image 145900</td>
<td>7. image 671700</td>
<td>8. image 587900</td>
</tr>
<tr>
<td>9. image 145800</td>
<td>10. image b36578</td>
<td></td>
<td></td>
</tr>
<tr>
<td>![Image 9]</td>
<td>![Image 10]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.15: Result of Query Using Exact $NN$-search
<table>
<thead>
<tr>
<th>1. image 671900</th>
<th>2. image b35859</th>
<th>3. image 145700</th>
<th>4. image b43001</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image 1" /></td>
<td><img src="image2.png" alt="Image 2" /></td>
<td><img src="image3.png" alt="Image 3" /></td>
<td><img src="image4.png" alt="Image 4" /></td>
</tr>
<tr>
<td>5. image b11146</td>
<td>6. image 145900</td>
<td>7. image 671700</td>
<td>8. image 587900</td>
</tr>
<tr>
<td><img src="image5.png" alt="Image 5" /></td>
<td><img src="image6.png" alt="Image 6" /></td>
<td><img src="image7.png" alt="Image 7" /></td>
<td><img src="image8.png" alt="Image 8" /></td>
</tr>
<tr>
<td>9. image 145800</td>
<td>10. image 579700</td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image9.png" alt="Image 9" /></td>
<td><img src="image10.png" alt="Image 10" /></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.16: Result of Query Using Approximate $NN$-search
Chapter 6

PicFinder: A Prototype Image Database System

In order to provide a practical example for testing the AB-tree described in chapter 3, we have designed and implemented a prototype system. PicFinder, a prototype image database system, is a content based image retrieval system. It has over forty four thousand color images. It currently has distance measures based on color and texture. It has two different indexing schemes: SS-tree for exact search, AB-tree for approximate search. The main purpose of this system is to demonstrate superior performance of AB-tree over that of SS-tree while providing almost the same quality of hits.

6.1 Introduction

Besides the traditional high volume sources of digital imagery, such as remote-sensing agencies and medical imaging facilities, a variety of new and diverse domains, such as art museums, news agencies, and the travel industry, have recently adopted digital technology for archiving images. In the face of this explosive proliferation of image repositories, we find ourselves ill equipped to efficiently retrieve images from large
collections. For collections containing more than even several hundred images, manual browsing is practically impossible. Initial attempts at developing computerized image-management systems consisted of extended traditional databases to handle images. This approach has not been very successful, mainly because of the use of externally generated descriptors used to index images in database systems.

In recent years, a new field of research has emerged: content-based image retrieval. Here, we aim to index images based on descriptions, such as color, texture, etc., derived automatically from the image. Such descriptions are adapted to the image and are usually more effective than external descriptors such as keywords, in characterizing the visual information illustrated in the image. The similarity between two images can then be determined by comparing their descriptions.

Content-based image retrieval systems such as IBM’s QBIC  [29], Webseek [73], PicToSeek [31], and ImageRover [76] use the query by similar images paradigm. They differ in how they find the initial set of images. QBIC displays an initial set of images. The user selects an image, then the search engine ranks the database images by similarity to the selected image with respect to color, texture, shape, or all of these criteria. Webseek and ImageRover use text queries to narrow the initial set of images, and PicToSeek asks the user to supply an initial image. In our system: PicFinder, the user can either supply his/her own initial query image or choose one from an initial set of images provided by PicFinder.

As discussed in Section 1.4, features and similarity measures are chosen based on heuristics, and are not mathematically precise parameters in content-based image retrieval systems. Approximate nearest neighbor searches are best suited for these
systems. The major contribution of PicFinder is to provide users a fast content-based image retrieval system based on AB-tree indexing as presented in chapter 3. In PicFinder users can choose indexing scheme AB-tree for approximate search, or SS-tree for exact search, or compare the performance and the quality of nearest neighbors between the two structures.

6.2 The Design of the PicFinder System

The PicFinder System is a prototype content based image retrieval system. It uses client-server structure. The client is a Java applet written in Java version 1.1, and runs in any web browser supporting Java version 1.1.7 or higher. The server is written in C++. It consists of approximately 30,000 lines of source code. The images in the database consist of 44,300 color images taken from a commercial image gallery published by IMSI-USA. This image database is a heterogeneous collection of images, including animals, background objects, food, plants, people, military, religion, scenic, structures, transportation, and etc.. They are stored in JPEG format on a local hard drive, taking about 1 gigabytes of space. PicFinder has been tested on UNIX and MS-WINDOWS. The URL of PicFinder is” http://www.cse.msu.edu/~lijinhua/PicFinder”

Figure 6.1 shows the system overview of PicFinder, including the relationship between client and server. When a user sends an image query from a Java-enabled web browser or client program to the server, the feature extraction module extracts the selected feature from the query image, then the matcher module uses the extracted feature from query image to search the feature database using the selected index structure and sends the best-ranked hit images back to the browser. The primary
modules of PicFinder consist of

- Java client for visual query input and displaying results,
- feature extraction,
- index structures,
- matcher.

The feature extraction module is based on IBM’s QBIC. It currently can extract the color feature and texture feature from an image in the following commonly used image formats: bitmap, gif, jpeg, and tiff.

- Color feature is the color distribution for each image in a predetermined 32 color space. Image similarity is based on the similarity of the color distribution.
- Texture feature is the texture information for each image. Image similarity is based on several texture attributes such as directionality, coarseness, and contrast.
PicFinder has two different indexing schemes: SS-tree for exact search, AB-tree for approximate search. The matcher module compares the extracted feature from query image to the feature database using the selected index structure and sends the best-ranked hit images back to the browser. We will discuss the Java client, the user interface of PicFinder, in Section 6.3.

PicFinder allows the creation of composite distance measure based on multiple features. The composite distance measure is a weighted average of the distance measures of selected features. Systems such as QBIC [29] and Virage [33] offer the user the ability to take weighted combination of color, texture, shape, and position measures, combined with keyword search. In a multi-feature query, QBIC searches through different types of feature data in the database in order to find images that closely resemble the query image. All features are treated equally during the database search, and all involved features are searched at the same time. An example of a multi-feature query would be finding images in the database that have a color distribution and texture similar to a query image. For multi-feature queries, you can weight features to specify their relative importance, which provides flexibility for advanced applications where the returned results must be fine tuned. Results of PicFinder can be combined to form a new query based on the user’s feedback. Search this new composite query may result in yielding more desirable results.
6.3 The PicFinder interface

Figure 6.2: The PicFinder Interface

The PicFinder interface is shown in Figure 6.2. Based on the functionality, it is divided into four windows: the QUERY window in Figure 6.3, the HELP window in Figure 6.4, the FEATURES and INDEXES window in Figure 6.5, and the RESULTS window in Figure 6.6.

To use PicFinder, an initial query image is needed. There are several ways to create an initial query image.
1. Type the URL of the user supplied query image into the TextField right to "Input image URL" label in the QUERY window. Then click "Load image" button to load the query image into the Query Image Area left to "Load image" button. PicFinder supports http, ftp, and file protocols, and supports the following commonly used image formats: bitmap, gif, jpeg, and tiff.

2. Click "Random image" in the QUERY window to pick a randomly selected image from the database as query image and load it into the Query Image Area. Click "Pre image" or "Next image" button to select the image before or after the current query image in the database as new query image.

3. Double click any image in the RESULTS window in Figure 6.6 to load it into the Query Image Area as query image.

![Image with URL and buttons](http://www.cse.msu.edu/~lijinhua/imgdb/10063160.jpg)

**Figure 6.3: The QUERY window**

The HELP window displays the help, tool tips, status and statistics results in the TextArea. Clicks the "Clear" button to delete all text in the TextArea. Clicks the "Help" button to display the overall help. The HELP window displays tool tips whenever user presses left button of mouse over any functional part of PicFinder
interface. The search status, warning or error message, and the statistics of search results are all displayed in the HELP window.

![HELP window](image)

Figure 6.4: The HELP window

In the FEATURES and INDEXES window, user can select or deselect a feature in "feature List" by clicking it, and select search scheme and number of hits as shown in Figure 6.5(a). In this case the "Define distance function for multi-feature selections" function is disabled. If user selects multiple features, the "Define distance function for multi-feature selections" function is enabled, and user can either use the default "equally weighted average", or "weighted average" and type in weights for each selected features as shown in Figure 6.5(b).

User runs the query by clicks the "Search" button in Figure 6.2. The results of the query are displayed in the RESULTS window. Initially, the RESULTS window contains the five images and their distances to the query image judged to be the closest by the system. The user can scan back and forth from the entire set of images returned by the system using the "Prev 5 images" and "Next 5 images" buttons. The images in the RESULTS window can be clicked on to move that image to the QUERY window as a new query image. Alternatively, User can select the images he likes or
Figure 6.5: The FEATURES and INDEXES window
dislikes by entering the weight(s) into the TextFields under the selected images as shown in Figure 6.6, then click "Refined search" button to do a new search based on user’s feedback.

Figure 6.6: The RESULTS window
Chapter 7

Conclusions and Future Work

The overall goal of our research has been to study the characteristics of data distribution in high dimensional Euclidean space and develop efficient applications for high dimensional data. One of the applications that was targeted for this research is the nearest neighbor queries in high dimensions.

To enable fast searches, we present an approximate search index tree called the AB-tree that uses heuristic to decide whether or not to access a node in the tree. The heuristic is based on the angle property and distance properties of data distribution in high dimensions. Extensive experiments on synthetic data and real data demonstrate that the AB-tree outperforms other index structures such as the SS-tree by orders of magnitude while maintaining 90% true nearest neighbors.

We summarize the contributions of this research as follows:

- We have studied and formalized data distribution properties in high dimensional Euclidean space, such as the distance properties that provide justification for poor performance of existing indexing methods, and the angle property that provides the basis for designing efficient indexing schemes.
• We have developed an efficient angle based balanced index structure for high
dimensional databases: the AB-tree, which is based on the SS-tree and improves
the performance of multimedia database accesses by several orders of magnitude
over the current methods.

• We have explored other heuristics such as clustering and caching to enhance
the performance of the AB-tree and have developed two variants of the AB-
tree for huge high dimensional databases: the clustered AB-tree and the cached
AB-tree, which outperform the standard AB-tree by a factor of 1.5 to 4.

• We have designed and implemented a prototype content based image retrieval
system called PicFinder that allows users to find approximate matches to query
images quickly using combinations of pre-defined distance measures. PicFinder
has been tested on a database of 44,000 images, and is capable of handling
millions of images. PicFinder demonstrates superior performance of AB-tree
over that of SS-tree while providing almost the same quality of hits.

As part of future work, we intend to extend AB-tree to very high dimensions for
applications like document search, and investigate the impact of the missing nearest
neighbors on real applications. For angle based filtering to be effective, the threshold
angles need to be determined automatically and efficiently. We have proposed a
heuristic with logarithmic time complexity to archive this. However, this heuristic
provides suboptimal threshold angles. Better threshold angles may be found providing
even better performance gain for AB-tree than that presented in this dissertation.
Bibliography


