A COMPARATIVE ANALYSIS OF MULTI-DIMENSIONAL INDEXING STRUCTURES FOR "EIGENIMAGES"

By

Andishe Samandari Sedighian

B. Sc. EE, University of Colorado at Colorado Springs, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

December, 1995

Computer Science
The University of British Columbia
2366 Main Mall
Vancouver, Canada
V6T 1Z4

Date: December 21, 1995

Abstract

Content-based retrieval in image management systems requires indexing of image feature vectors. Most feature vectors have a high number of dimensions (>20). This makes indexing difficult since most existing multi-dimensional indexing structures grow exponentially in size as dimensions increase. We approach this problem in three stages: i) reduce the dimensionality of the feature space, ii) evaluate existing multi-dimensional indexing structures to determine which can best organize the new feature space, and iii) customize one of the selected structures to improve search performance. To reduce the dimensionality of the feature space without losing much information we apply a statistical technique called Principal Component Analysis (PCA), using Turk and Pentland's *eigenfaces* approach. We then conduct a comparative analysis of a wide range of existing multi-dimensional indexing structures (quad trees, KD-trees, R-trees, gridfile, and multipaging), selecting three of them (bucket adaptive KD-tree, gridfile, R-tree) for further empirical comparisons. Tests show that the bucket adaptive KD-tree uses the least storage and peforms the best during search. Finally, we customize the bucket adaptive KD-tree by implementing techniques that take advantage of the characteristics of the transformed space -- namely ranked dimensions by decreasing variance and known dynamic ranges. This prunes the search space and results in very efficient searches. The number of page accesses are reduced significantly, some times leading to savings as high as 70%.

Table of Contents

List of Tables

List of Figures

Acknowledgement

First and foremost, I would like to express my sincere thanks to my supervisor, Dr. Raymond Ng, for his guidance, assistance and understanding over the past two years. I truly appreciate the fact that he always made the time to discuss concepts with me at length and to clarify and refine ideas. Moreover, I am particularly grateful that he gave me the time I needed to spend with my parents when my father was very ill.

Second, I would like to thank Dr. Robert Woodham who was the second reader on my thesis. Eventhough, as Head of the Department, his schedule was very full, he graciously accepted to spend the time necessary to review my work and provide me with valuable and thought-provoking comments.

I would also like to thank my fellow students, Richard Pollock, David Hsu and Michael McAllister, for the assistance they gave me at different times along the way. Although they may not have thought it much, the discussions I had with them on different aspects of my thesis helped me work through some of the harder concepts. I wish them all the best of luck in their academic endeavors and future careers.

A special thank-you goes to Dr. Christos Faloutsos of the Department of Computer Science at the University of Maryland, College Park, for allowing us to use code that he and his students had developed for two of the multi-dimensional indexing structures we use in this work: the gridfile (by C. Faloutsos and S. Hong) and the deferred-split R-tree (by M. Bayraktar, C. Faloutsos, Y. Tan, J. Tang and W. Wang).

I would like to especially thank my dear husband, Kamran, for all the love, support, and encouragement he gave me through thick and thin. In particular, I would like to thank him for his invaluable comments on this document.

Finally, I thank the Lord for giving me the strength and ability to accomplish this milestone, and for continuously granting me His abundant grace and bounties.

Chapter 1
INTRODUCTION

1.1 The Need for Image Database Management

Conventional database management systems (DBMS's) are designed with alphanumeric data in mind. Over the past thirty years much research has gone into designing these systems so as to give users a seamless and transparent view into the data domain being managed [GM92a]. As a result, DBMS's have been very successful and have proliferated in all areas of business and industry. In the last decade, however, there has been a significant increase in the generation of non-alphanumeric data such as graphics, maps, images, video and audio. From business to academia, from medicine to land resource management, from education to entertainment, more and more people are interacting with large image databases. Although some early attempts were made to adapt traditional DBMS's to support non-alphanumeric data [KWT74], it quickly became clear that traditional database management techniques could not effectively handle image or other non-alphanumeric data. New approaches have to be used for designing image information management systems.

Until very recently, most image database systems have generally fallen into one of two categories: 1) databases with no image understanding capabilities, or 2) vision systems which store images in a basic image repository [BPJ93, GM92a]. The first approach requires recording textual annotations which describe each image. These annotations are then entered into a traditional database and searches are performed based on keywords stored in this database. The images themselves are not really part of the database but are stored separately; they are only referenced by text strings and pointers. There are a number of serious limitations to this approach. It is quite clear that such a method is labor-intensive, involving the manual cataloging of thousands, if not hundreds of thousands, of images. In addition, the complexity of the information embedded in the images cannot be sufficiently described in a few keywords so as to distinguish a particular image from other images. Finally, there is the difficulty of anticipating every user's needs when assigning keywords to images; a user may not interpret an image in the same way the database system designer may have interpreted it upon initial insertion.

The second approach originated in the image processing community. These systems have the ability to accurately interpret complex image data. However, they are intended strictly for vision applications and research, and therefore simply maintain the images in basic image repositories. Support for database processes such as insertion, indexing, querying, and so on is very limited, and only small numbers of images (i.e., tens or hundreds) are used as testbeds. Researchers in the emerging field of Visual Information Management Systems (VIMS) believe that the creation of mere image repositories is of little value. Methods for fast retrieval of images based on their content must be devised for data sets of realistic sizes, i.e., tens or hundreds of thousands of images [Jai93]. The realization of such image management systems requires the development of new techniques in the fields of databases, computer vision, and knowledge-based systems.

Since the beginning of the 1990's, researchers in the fields of image processing and understanding, knowledge representation and knowledge based systems, and databases have begun working together to face the challenges of designing new VIMS[1]. These groups have approached the problem of content-based image retrieval in a variety of ways. As a result, a number of different prototype image database systems have emerged. Below is a sampling of these systems. A more extensive review of these systems is presented in Chapter 2:

1) **QBIC** (Query By Image Content): developed at the IBM Almaden Research Center, this system focuses on retrieving images from stocks of photo clip art based on three types of image content -- color, texture and shape [F+94, N+93].
2) **ΣNIGMA**: developed at the University of Amsterdam, this system works with MRI images of the chest. Small image patches that frequently occur in the set of MRI images are extracted and used as key features by which to search the database [GS92].
3) **Xenomania**: developed at the University of Michigan, this is a visual information system used for interactive face retrieval. Face features such as eyes, nose, and mouth are used to locate a specific face in the database [BPJ93].
4) **Face Photobook**: developed at the Media Lab at M.I.T., this system is used for face recognition. The *eigenfaces* approach of Turk and Pentland [TP91] is used to represent and retrieve images from the database [PPS94].

[1] The IRIS/IC-5 Project based in the Department of Computer Science at the University of British Columbia is one such group of collaborators from various disciplines.

1.2 Multi-dimensionality of Image Features

The majority of current image database system prototypes represent and retrieve images based on a variety of image content descriptors. The most common of these are color, texture and shape. These content descriptors are commonly termed *image features*. An image feature can be defined as a numerical value or a vector that is computed based on the low-level properties of an image (such as pixel intensity values). Various current image analysis techniques may be used to extract one or more image features from each image. For example, color histograms are computed, edge properties are extracted, texture features are calculated that represent coarseness, contrast or directionality, and shape features such as area, circularity, and eccentricity are measured.

The important information in an image can also be represented using another approach. This involves first somehow capturing the variation in the whole database of images, *independent* of any *explicit* features, and then using this information to encode and compare the individual images in the set [TP89, TP91]. The details of such a method are explained in a later chapter. However, the important difference to remember between this approach and the previous one is that in the former technique a specific feature vector (such as a color histogram or a set of texture values) is extracted from each image and used to represent it. In the latter technique each image in a database, with N rows and N columns of pixels, is first converted into a single N^2-element vector by concatenating the rows of pixel intensity values. Then, using a statistical tool, the variation in the complete set of image vectors is calculated. In mathematical terms, the principal components of the distribution of images are found. Finally, each individual image is represented as a linear combination of these components.

In both approaches, due to the complexity of the information, the images are represented as points in a multi-dimensional space. Each *point* in this space is a unique *k*-dimensional location represented by a vector $\mathbf{x} = [x_1, x_2, \ldots, x_k]$. As an example, color is usually a 3-valued tuple treated as a 3-dimensional point in a color space such as R, G, B. It is common to have image vectors with 20 or more dimensions. In this thesis, we will consider a point with over 20 dimensions as one that has *high dimensionality*, a point with 6 to 20 dimensions as one that has *medium dimensionality*, and a point with less than 6 dimensions as one that has *low dimensionality*.

The high dimensionality of image vectors can pose a real problem particularly when one considers that usually real-life image databases store tens to hundreds of thousands of images. This problem is many-faceted: 1) a large number of dimensions per image means that a lot of space must be used just to store the image vectors; 2) if the dimensions are not all independent then redundant information is stored; and 3) when calculating the similarity between an image and a query, large numbers of dimensions means increased computation time. Therefore, in order to retain reasonable storage requirements and low computation time, techniques should either be developed or adopted that can efficiently analyze and compress image vectors.

1.3 Multi-dimensional Indexing Structures

In traditional DBMS's various indexing techniques have been developed to facilitate searches on alphanumeric data (e.g., B-trees and hashing). However, these conventional methods are not suited for indexing multi-dimensional data points. Research into techniques for accessing spatial data has generated several indexing structures designed to handle multi-dimensional points and spatial objects (such as rectangles) [Güt94]. The most common spatial or multi-dimensional indexing structures include the quadtree and its family [FB74, Sam90], the KD-tree and its variants [Ben75, Ben79, Sam90], the gridfile [NHS84, Sam90], and the R-tree and its variants [Gut84, B+90, SRF87]. Since images can be represented as points in a multi-dimensional feature space, these structures are suited for organizing image feature data. Consequently, these structures have been adopted by some researchers for this very purpose. In QBIC and [A+95], for instance, an R*-tree [B+90] (a variant of the R-tree) is used to index shape and texture feature vectors respectively.

Although multi-dimensional point access methods are suited for accessing image feature data, the fact that most image features have a high number of dimensions restricts the efficacy of these methods. This is because most existing multi-dimensional indexing structures cannot deal with high-dimensional data points. They often grow exponentially with the number of dimensions -- a phenomenon known as the "dimensionality curse" [F+94, A+95]. The R-tree methods, however, seem to be the most robust with experiments indicating that the R*-tree can handle up to 20 dimensions [F+94, A+95]. The inability of existing multi-dimensional indexing structures to effectively handle the organization of high-dimensional image feature vectors is a further reason why it is necessary to find ways of compressing image feature spaces. The VIMS research community has recognized this shortcoming of multi-dimensional indexing structures and has made the following recommendation at the 1992 National Science Foundation workshop on VIMS:

> Efficient indexing methods for high dimensions need to be developed, balancing efficient use of index memory with efficient retrieval (i.e. $O(\log n)$) methods. Except in the field of information retrieval, current 'multi-dimensional' indexing methods are oriented toward relatively few dimensions, say 10 to 20. [Jai93, p.63]

1.4 Similarity Matching and Search Operations

Besides efficient multi-dimensional indexing, one of the main functionalities we need from an image database is *similarity matching*. Here, an important distinction needs to be made between searches performed on image databases and those performed on traditional alphanumeric databases. In traditional DBMS's, the retrieval of data is based on *exact matches*, for example, find all students in the database with the last name "Smith". Exact match, however, is not possible with image databases. There are primarily two reasons why a measure of similarity must be developed: 1) Users' queries are ordinarily rough estimations of the image they are looking for; for example, find an image that has a "reddish" ball, "about" this large, "somewhere" in this corner. In such a query, neither the exact color of the object, nor its size, nor its precise location are provided by the user. The user often just has a notion of what (s)he is looking for and cannot remember the exact details of an image. 2) The introduction of noise and distortion is inherent in image processing [GM92].

How can multi-dimensional indexing structures help with similarity matching? There are two important characteristics of a multi-dimensional indexing structure that assist in performing a similarity match. The first characteristic is the way in which the structure organizes the data space so that during a search only parts of that space, and hence a subset of the points in it, need to be considered to answer a query. This characteristic is important for reducing the number of images for which the similarity metric needs to be computed. The ratio of the reduced set of images to the full set of images is termed by Alexandrov, *et al.* [A+95] as the *discriminating power* of the indexing structure. The smaller the ratio, the greater the discriminating power. The second characteristic is the efficiency of the indexing structure -- i.e., how fast it can produce the reduced set of images for a fixed value of the discriminating power. The efficiency is, among other things, dependent on the search operation used. The appropriate indexing structure in conjunction with the right kind of search operation can result in efficient processing of similarity match queries and a high discriminating power.

There are several types of search operations that are useful for similarity matching. These include *range search*, *partial range search*, *nearest-neighbor search*, and *fixed-radius near neighbor search*. Following is a brief definition of each kind of search:

1) *Range Search*: The process of retrieving the appropriate records when given an "orthogonal range query," i.e., find all records whose k key values fall within specified ranges [BF79, BM80]. *Partial Range Search* is very similar except that only n of the k key values ($n < k$) have specified ranges.
2) *Nearest-Neighbor Search*: This is also known as the "best-match" or "closest point" search. It is the process of retrieving the record or m records that are most similar to a query according to some (dis)similarity measure. This (dis)similarity measure is often a distance function [SW90].
3) *Fixed-Radius Near Neighbor Search*: The process of retrieving all records that fall within some fixed radius r of a query [BM79].

These search algorithms can play two different roles during query processing in image databases. They can be used as filters to reduce the number of images on which similarity functions are implemented (e.g., first perform the proper range search and then compute a detailed similarity metric on the returned images), or they can be used to perform the complete search itself (e.g., perform nearest-neighbor search to get the m best matches).

Most often, search algorithms are used for the first purpose -- to reduce the search space on which the thorough matching is done.

In this thesis, we concentrate on finding all images that are within a fixed distance from a query. There are two main reasons for this choice: 1) often, users are not interested in the "best match" to their query but prefer to retrieve a number of similar images so that they can further modify their query using these images [A+95]. The "fixed-distance-from-query" notion of similarity fits this requirement quite well; and 2) this form of similarity matching is often used in image database systems [PK93, F+94]. The distance metric that we choose to work with is the Euclidean distance[2]. Our reasons are because Euclidean distance is a natural and common notion of distance, and is used frequently for similarity match in current image database prototypes [PK93, F+94].

The fixed-radius near neighbor problem in *k*-dimensional space is not a simple one. The distance function between two neighboring data points for this type of search could be any vector space L_p-norm, such as the L_1-norm or the L_2-norm [FBF77]. Although fixed-radius search is independent of the type of distance metric used, the metric must be a summation of some difference between each dimension of the two points under consideration. Much work has been done in this area [Ben75a, Yuv75, BSW77], however, most multi-dimensional indexing structures still only approximate fixed-radius search. Some simply use range search instead of fixed-radius search, setting the range in each dimension equal to twice the length of the specified radius [Sam90]. Others take it one step

[2] The Euclidean distance between two *k*-dimensional points, **x** and **y**, is defined as:

$$\text{dist}^2(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{k} (x_i - y_i)^2$$

further and actually compute the distance between the query and the points retrieved by the range search, further refining the answer given to the user [F+94].

Range search differs greatly from fixed-radius search in that it considers each dimension in a data point independently of the rest of the dimensions. The value of each dimension is simply compared against the appropriate query range. For this reason, unless a search requires that the dimensions be treated independently, using range search to find points that are within a fixed distance from a query is a poor approximation of fixed-radius search. Such an approximation means that the hyper-sphere of the radius search is approximated by the hyper-rectangle of the range search. This results in the retrieval of many *false hits*, i.e., images that are retrieved which in fact do not satisfy the query constraints. For uniformly distributed data, this approximation gets poorer as the number of dimensions of the feature vectors increases. Therefore, if a structure is used to find all images whose distance from a query is below a fixed threshold, as in [PK93] and [F+94], it would improve the discriminating power and the efficiency of the indexing structure if fixed-radius near neighbor search is used rather than range search.

1.5 Problem Definition and Thesis Contributions

As we have seen, images in a database can be represented as vectors in a multi-dimensional space. Frequently this image space has a high number of dimensions. This presents a few challenges to designers of image management systems: 1) high dimensional feature vectors can be costly in terms of storage and computation time; 2) most existing multi-dimensional indexing structures cannot handle points with high numbers of dimensions, so they must be chosen judiciously; and 3) more efficient search techniques need to be employed to reduce

CPU and I/O time (i.e., number of page accesses). This thesis addresses these issues by investigating the following questions:

1) How can we compress or reduce the dimensionality of the image vector space?

2) What existing multi-dimensional indexing structures are suitable for organizing the image space once it has been compressed?

3) How effective, in terms of computation time and page accesses, is range search in approximating fixed-radius near neighbor search when looking for data points within a fixed distance from a query?

In answering the first question, we chose and implemented a dimension-reducing, distance-preserving technique called Principal Component Analysis[3] (PCA). This technique transforms the data space by removing dependent dimensions and converging most of the information into the first few dimensions. Our implementation of PCA follows that of Turk and Pentland in which full image vectors are used in the calculations [TP91]. Our experiments confirm their results, demonstrating that PCA can effectively reduce high-dimensioned image vectors (or high-dimensioned specific feature vectors) while retaining most of the variance in the data set. Our findings further show that this technique works well on relatively large data sets (400 images in our test bed versus 16 images in Turk and Pentland's).

In answering the second question, we conducted a comparative analysis of existing multi-dimensional indexing structures to determine which ones could best support image

[3] PCA is also known in pattern recognition as the Karhunen-Loeve transform.

vectors that have been transformed using PCA. In this analysis, the issues of primary concern are: i) how the storage cost of the structures scales up with increasing dimensions; and ii) what kinds of searches the structures are suited for. The results of this analysis indicate that 3 of the structures are the most reasonable for our purposes. These are the bucket adaptive KD-tree, the family of R-trees, and multipaging. In order to get a concrete understanding of the suitability of these structures we decided to implement all three structures and carry out a thorough experimental evaluation of each. We implemented the bucket adaptive KD-tree using Samet's text [Sam90] as reference. However, due to limited time, we did not implement R*-tree and multipaging. Instead, we acquired an R-tree and a gridfile structure from Christos Faloutsos at the University of Maryland. Although the R-tree and gridfile implementations are not the structures we originally planned to evaluate, they are still useful for our comparison because of their similarity to R*-tree and multipaging. The experiments we conducted used data points that were extrapolated from a set of 400 gray-scale face images. The dynamic ranges of the principal components of the image vectors were initially extracted, then larger data sets, from 500 to 50,000 data points, were generated based on this original data. To compare the performance[4] of these structures over a number of dimensions, we generated data points that varied from 2 to 10 dimensions for each data set size. Using this testbed, we compared storage costs and performance of range search for the three indexing structures. The results indicate that the bucket adaptive KD-tree is the most suitable structure for organizing PCA-transformed data.

[4] We measure performance by looking at overall search time and number of page accesses.

In answering the third question, we modified the bucket adaptive KD-tree[5] implementation so that it could perform fixed-radius near neighbor search. Furthermore, we implemented a number of the ideas outlined by Bentley [Ben75a] to improve the performance of fixed-radius search. The two key techniques we implemented are termed by us as the *Early Fail Test* and the *Early Success Test.* The Early Fail Test checks to see if the sum of the squares of the *minimum* differences between a query point and the hyper-rectangle of the range of a sub-tree is greater than a fixed threshold r^2, where r is the distance (or radius) from the query point. If this sum exceeds r^2, it indicates that the entire sub-tree can be skipped. To reduce the number of calculations and speed up this test, we compute the sum of the squares of the minimum differences only until they exceed r^2. The Early Success Test is used to determine whether or not the hyper-rectangle of the range of a sub-tree falls entirely within a fixed radius from the query point. If the sum of the squares of the *maximum* differences between the query point and the hyper-rectangle is less than or equal to r^2, this indicates that all data points under that subtree satisfy the query. Therefore, further testing down that sub-tree is unnecessary. Again, to reduce the number of calculations and speed up the Early Success Test, we compute the sum of the squares of the maximum differences only until they exceed r^2. To implement both tests, we had to store the minimum and maximum values of the dimensions of the hyper-rectangle of the sub-tree at each node. Experiments comparing the performance of the optimized bucket-adaptive KD-tree and the original bucket adaptive KD-tree indicate that, for a medium number of dimensions, the optimized BA_KD-tree leads to a significant reduction in the number of page accesses and overall search time[6], sometimes as much as 70%.

[5] In this thesis the term "original bucket adaptive KD-tree" will refer to the initial implementation of this structure with only range search capability, and the term "optimized bucket adaptive KD-tree" will refer to the modified version of this structure with optimized fixed-radius near neighbor search capability.

[6] Overall search time for the optimized tree includes the extra computations necessary for performing the Early Fail and Early Success Tests.

In summary, the work of this thesis: 1) confirms that PCA is a useful statistical technique for reducing the dimensionality of image vector spaces; 2) determines that the bucket adaptive KD-tree performs well for indexing PCA-transformed data; 3) verifies that range search is a poor substitute for fixed-radius near neighbor search; and 4) demonstrates that the bucket adaptive KD-tree can be modified to handle an optimized fixed-radius search operation that greatly improves the structure's performance in finding all images within a fixed distance from a query.

1.6 Outline of Thesis

The remaining chapters contain the following: Chapter 2 describes related works. It reviews some prototypes of image databases developed by various researchers in the image processing and database communities. Chapter 3 focuses on the PCA technique. It first explains the mathematical theory behind PCA, and then describes the *eigenfaces* approach of Turk and Pentland which we adopt for analyzing the images in our database. Chapter 4 presents in detail the comparative analysis of a wide range of existing multi-dimensional indexing structures. It explains the criteria on which our analysis is based, and gives the reasons for our choice of structures. Chapter 5 describes the testbed used for this thesis, covers the key aspects of the implementation of the three structures we chose in Chapter 4, and presents the results of our experimental evaluation of each. Chapter 6 explains the modifications that we made to the bucket adaptive KD-tree and the details of the implementation of fixed-radius near neighbor search. It also presents the results of the comparison of the performance of the optimized bucket adaptive KD-tree and the original bucket adaptive KD-tree. Finally, Chapter 7 presents our conclusions and suggestions for future work.

Chapter 2
RELATED WORK

In recent years there has been much research in the field of Visual Information Management Systems. This has resulted in the development of many new prototypes. In this chapter we examine a number of these prototypes (ΣNIGMA [GS92], AMSTERDAM [SSG92], ART MUSEUM [HK92], TRADEMARK [Kat92], QBIC [N+93], Xenomania [BPJ93], Miyabi [H+93], FINDIT [Swa93], Photobook [PPS94] and others) and see how they address the three central issues of this thesis -- namely, reduction of the number of dimensions of the feature space, organization of the feature space, and application of an efficient search technique.

2.1 Image Analysis and Reduction of Feature Dimensions

There are several kinds of image features that are used to describe the contents of images in a database. In general, the systems we investigate in this thesis look at one or more of the following features: color, texture, and shape. Below, we give a summary of the image features used in several of these systems and examine how, or whether, they attempt to reduce the dimensionality of the features that are extracted.

2.1.1 Color

One of the features most commonly extracted from an image is its color content. Swain and Ballard state: "The color spectrum of multicolored objects provides a robust, efficient cue for indexing into a large database of [images]" (p. 390) [SB90]. In [SB90], [SB91], [BGS92], and [Swa93], color is used as the only indexing feature. In QBIC [N+93] and [SSU94], color is used along with a few other features for indexing images. Images are represented by color histograms, and a metric on the histogram space is used to determine the similarity between two images. Histograms are usually compared sequentially across all the images in a database. However, since this type of recognition scheme is linearly dependent on the database size, computation time dramatically increases for large image databases.

To reduce this time Swain and Ballard [SB91, Swa93] use a technique they call *incremental intersection* which takes advantage of the fact that in a typical histogram a small number of bins usually capture the majority of pixel counts. In this scheme only the largest bins of the query and database image color histograms are compared and a partial histogram intersection value is computed for the similarity match.

The QBIC project approaches this problem in a slightly different way. Instead of reducing the number of bins used in the histogram comparison, QBIC reduces the number of images on which a full histogram comparison is performed. To better understand their approach, let us first review the QBIC definition of a full historgram comparison. The distance between two color histograms **x** and **y** is defined as:

$$d_{hist}^2(\mathbf{x}, \mathbf{y}) = (\mathbf{x} - \mathbf{y})^T \mathbf{A} (\mathbf{x} - \mathbf{y}) \qquad (2.1)$$

where **x** and **y** are (K x 1) vectors (K is the number of histogram bins), T denotes transpose, and **A** is a K x K matrix which has entries a_{ij} that describe the similarity between color *i* and color *j*. This distance takes into account the "cross-talk" between two colors such as orange and red thereby correctly computing that orange images are similar to red images or that half-red/half-blue images are quite different from all-purple ones [F+94]. To reduce the computation necessary to compare a query image to the images in the database, Faloutsos, *et al.* introduce the following filtering step. They first compute the Euclidean distance $d_{avg}(\mathbf{x},\mathbf{y})$ between the average color of the query image, **x**, and the average color of an image in the database, **y**. The average color $\mathbf{x}_{avg}$ of an image is defined as the average R, G, and B values of the pixels in that image:

$$\mathbf{x}_{avg} = (R_{avg} , G_{avg} , B_{avg})^T \qquad (2.2)$$

If $d_{avg}(\mathbf{x},\mathbf{y})$ is greater than some threshold value ε, then that image is dropped from the search. But those images whose $d_{avg}(\mathbf{x},\mathbf{y})$ is less than or equal to ε go on to have the full histogram distance computed. This technique acts as a filter on the database ensuring that there are no false dismissals, but accepting some false hits. These false hits are later discarded by performing full-histogram distancing on the set of images captured in the filter step. In this way computation time is saved as it is much cheaper to perform the full histogram comparison on only a subset of the images rather than on the whole image set.

2.1.2 Texture

Texture information is one of the basic cues on which patterns can be retrieved. Four of the systems we investigate extract texture features with which to represent the images in their database. QBIC's texture features include coarseness, contrast, and directionality [N+93]. Sakamoto, *et al.* define the texture of a region as its coarseness [SSU94]. In Texture

Photobook [PPS94] texture is represented by measurements of repetitiveness, directionality and complexity. Alexandrov, *et al.* use a series of Gabor filters to capture 120 different texture features [A+95]. The dimensionality of the texture features in the first three systems is very low therefore they do not use any feature compression techniques. However, since Alexandrov, *et al.* have a very high dimensioned feature measure, they develop a scheme for determining the relative importance of the Gabor filters. The filter ordering is pattern dependent and is based on the average spectral information of all the images in the database [A+95]. They are able to reduce the dimensionality of the texture feature down to a maximum of two dimensions.

2.1.3 Shape

Shape recognition has always been one of the major challenges in the field of computer vision. It has also become one of the most challenging aspects of content-based image retrieval. Several of the prototype systems extract shape features and use them for indexing the images in their data set. Following is a brief listing of the types of shape features that are extracted by the different systems we study: 1) Gary and Mehrotra [GM92b] use a set of polygonal approximations of the actual object boundary to define an object; 2) Grosky and Mehrotra [GM90] employ data-driven model-based shape recognition; 3) Grosky and Jiang [GJ92] use vertex angles and the lengths of their adjacent edges as shape features; 4) Hou, *et al.* [H+92] extract a sequence of feature vectors derived from the center-of-mass of the individual objects in an image; 5) Pentland, *et al.* [PPS94] use Finite Element Method models [SP93] of objects to align, compare, and describe objects; 6) Samadani, *et al.* [SHK93] extract 15 different shape features from a specified sector of an image (global features, such as total area and total intensity, and radial features, i.e., features along a radial line, such as width and variation of width); and 7) in QBIC [N+93, F+93] they extract a total of 20 shape features that include area, circularity, eccentricity, major axis orientation and a set of algebraic moment invariants. Other systems such as ΣNIGMA [GS92], Miyabi [H+93], and ART MUSEUM [HK92] use image segmentation to extract the outline of objects in an image.

As seen above, shape feature vectors frequently have a high number of dimensions. This high dimensionality makes computing the similarity between images expensive, especially when dealing with large data set sizes. Despite this high cost, most of the systems cited, with the exception of QBIC, do not try to alleviate the problem. QBIC uses a method known in pattern recognition as the Karhunen-Loeve (KL) transform, to decorrelate and compress the texture dimensions. The KL transform, known in statistics as principal component analysis, is a data-dependent orthonormal transform. It requires using a sample of the data set to compute the transformation matrix. The columns of the transformation matrix correspond to the eigenvectors with the largest eigenvalues of the covariance matrix of the feature vectors. To achieve dimensionality reduction, the first few eigenvectors are selected since they contain most of the variance of the whole data set [F+94]. The reduced-dimension shape feature vectors are then used to perform the necessary distance calculations thus reducing the computation time.

2.1.4 Summary of Image Analysis and Feature Reduction

From the above review, we can see that most of the prototypes we examined do not concern themselves with the high dimensionality of the feature vectors they extract. There seems to be two reasons for this. First, the focus of the prototypes is primarily on developing techniques for extracting relevant feature information from images in a database and using this extracted data later for similarity calculations. Second, the image testbeds used are, for the most part, small in size (i.e., from scores to a few thousand images). Consequently, the problems that a large number of feature dimensions create, namely high storage cost and expensive computations, are not an issue in these systems. This, however, does not diminish the importance of dealing with the "dimensionality curse." If systems are to handle realistic-sized databases such as digital libraries of images with tens to hundreds of thousands of images, techniques for reducing the dimensionality of image features, without suffering great loss of information, will have to be adopted or devised.

In our study, we recognize that the high dimensionality of most image representations are costly for the system. Hence, we adopt and implement an image analysis and dimension-reducing technique, dubbed *eigenfaces*, that was developed by Turk and Pentland at M.I.T. [TP91, TP89]. It is founded on the same principles as the technique used in QBIC for reducing the dimensionality of its shape feature vectors. However, instead of working with *specific* feature vectors, the *eigenfaces* technique is applied to *whole* image vectors. It performs PCA on the pixel intensity values of a sub-set of the images in a database, extracting a small number of vectors that carry most of the information in the images. These vectors become the "basis vectors" by which the images in the database are defined. Further details of this technique are given in Chapter 3.

2.2 Feature Space Organization

With a multi-dimensional vector representing each image in a database, the whole database maps to a collection of *k*-dimensional points in a *k*-dimensional space[7]. This space may then be organized by an indexing structure to assist in performing searches. However, most of the prototype systems we examine do not use any sort of indexing structure for organizing the image features they extract. They simply have the feature vectors stored in flat files or arrays and use them to sequentially compare images in the database to a given query [GJ92, HK92, GS92, SSU94, PPS94, Kat92, H+93, BPJ93, Swa93, SB91]. A few systems use one of the existing multi-dimensional indexing structures for feature organization: 1) QBIC [F+94] and Alexandrov, *et al.* [A+95] use the R*-tree for indexing their shape and texture features respectively; 2) Gary and Mehrotra [GM92b] state that any multi-dimensional point access method can be used to form the index on the shape features they extract; and 3) Binaghi, *et al.* [BGS92] use the R-tree for organizing the color information they extract from images. Two of the prototype systems [GM90, H+92] build binary search trees based on the similarity distance between features but not based on the image feature vectors themselves. Finally, one of the prototypes [SSG92] organizes the features it extracts into a relational table.

As was mentioned above, most of the systems we investigate do not use an indexing structure to organize the image feature space they generate. The reasons for this seem to be, once again, twofold: indexing is not a focus of those studies and, more importantly, small sized data sets do not prompt the need for sophisticated feature space organization. We feel that an image management system that manages tens to hundreds of thousands of

[7] Often, there is more than one multi-dimensional space that represents the images in a database since several feature vectors may be extracted from each image.

images needs to utilize a suitable indexing structure, particularly one that can handle multi-dimensional points, if it is to answer queries efficiently. The selection of an indexing structure must, however, be done carefully as not all multi-dimensional indexing structures are well suited for any multi-dimensional data. In this thesis, before choosing an indexing structure to organize our multi-dimensional image vectors, we conduct a thorough analysis of existing multi-dimensional indexing structures to determine which one(s) would be most appropriate for our use. This examination includes both a comparative analysis and an experimental evaluation of a wide range of structures. Chapter 4 provides the details of this study.

2.3 Search Techniques

Organizing the image feature space by using an indexing structure is important. This is because indexing helps to reduce search time compared to sequential searching (search time includes CPU time and I/O time). However, since most of the systems we investigate store their feature vectors in non-indexed flat files, they simply perform a sequential search through the entire database of images computing the similarity between the query and each image. This sequential process involves the following steps:

1) given a query image, a feature vector is extracted;
2) the distance between the feature vector of the query and the feature vector of each image in the database is computed;
3) images whose computed distance is below a predefined threshold are retrieved.

Step 3 in this process indicates that a fixed-radius query is the notion of similarity match that is commonly used. Additionally, the L_2-norm, or Euclidean distance, is oftentimes the distance metric of choice. For instance, in QBIC [N+93] and Texture Photobook [PK93], shape and texture features, respectively, are matched using the weighted Euclidean distance between two feature vectors. Swain and Ballard [SB91], however, measure the distance between two color histogram features using the L_1-norm[8], or "city-block metric" [Str94]; and, as we saw in section 2.1.1, QBIC measures the distance between two color histograms, **x** and **y**, using equation 2.1. Although this calculation differs from Euclidean distance, d_{hist} becomes the Euclidean distance between **x** and **y** in the special case where the matrix **A** is the identity matrix **I** [F+94]. For the relatively small databases of these prototypes, using sequential search with these distance metrics is acceptable. But, for real-life scientific and commercial applications more sophisticated search techniques must be employed.

The L_1- and L_2-norms used for similarity measurement by the above mentioned prototypes are ideal for *fixed-radius near neighbor search.* Nonetheless, the systems that utilize an indexing structure perform *range search* to find the approximate number of images that lie within the desired radius from a query [N+93, BGS92, GM92b]. Here, it is important to recognize the difference between fixed-radius near neighbor search and range search. As was explained in section 1.4, fixed-radius search uses an L_p-norm as the distance function. This type of metric is a summation of some difference between the dimensions of the two feature vectors being compared. The resultant sum is checked to see if it falls below a predefined threshold (i.e., within a specified radius). Range search, however, is designed to work for "orthogonal" queries where each dimension is considered independently of the rest. The dimensions no longer cumulatively satisfy the query

[8] The L_1-norm is defined as the sum of the absolute differences between 2 k-dimensional vectors, **x** and **y**:

$$\text{dist}_{L1}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{k} |x_i - y_i|$$

constraints. Therefore, we conjecture that using range search, out of convenience, to approximate fixed-radius near neighbor search is a mistake. It could result in poor performance, both in terms of number of false hits and overall search time. To test this hypothesis we modify an existing indexing structure and enhance it with the ability to effectively execute fixed-radius near neighbor search. We then compare the performance of the modified structure using fixed-radius search to the original structure using a range search approximation. The details of this comparison can be found in Chapter 6.

Chapter 3

PRINCIPAL COMPONENT ANALYSIS

In this thesis we choose the *eigenfaces* method of image analysis and feature compression. It was initially developed by Sirovich and Kirby [SK87, KS90] and later expanded by Turk and Pentland [TP89, TP91]. This approach is based on a statistical technique called Principal Component Analysis. In this chapter we first review the mathematical theory behind PCA, then we discuss how the theory is implemented to create *eigenimages* of a data set, and finally we share the important aspects of our implementation of this technique.

3.1 Mathematical Foundations of PCA

PCA is probably one of the oldest and best known techniques of multivariate analysis. I. T. Jolliffe provides a clear and concise description of PCA in his text [Jol86, p.1]:

> The central idea of principal component analysis (PCA) is to reduce the dimensionality of a data set which consists of a large number of interrelated variables, while retaining as much as possible of the variation present in the data set. This is achieved by transforming to a new set of variables, the principal components (PCs), which are uncorrelated, and which are ordered so that the first *few* retain most of the variation present in *all* of the original variables.

The principal components of a data set are essentially a rotation of the original orthogonal dimensions, plus a ranking of the dimensions by decreasing variance. This ranking typically captures most of the variation in the data set in the first few dimensions. This means that one can define the original set of data points using only the higher ranked principal components and still retain most of the information in the set. In addition, the Euclidean distance between data points in the original feature space remains the same as in the transformed feature space.

3.1.1 Definition of PCA

Let us first give a formal definition of principal components. Given that **x** is a vector of p random variables, the first PC is a linear function $\alpha_1{}^T\mathbf{x}$ of the elements of **x** which has a maximum variance, where α_1 is a vector of p constants:

$$\alpha_{11}, \alpha_{12}, \alpha_{13}, \dots, \alpha_{1p}\,.$$

So, the linear function becomes:

$$\alpha_1{}^T\mathbf{x} = \alpha_{11}x_1 + \alpha_{12}x_2 + \alpha_{13}x_3 + \dots + \alpha_{1p}x_p = \sum_{j=1}^{p} \alpha_{1j}\, x_j \tag{3.1}$$

The second PC is another linear function $\alpha_2{}^T\mathbf{x}$, uncorrelated with $\alpha_1{}^T\mathbf{x}$, which has maximum variance. This holds for the rest of the PCs so that the kth PC is a linear function $\alpha_k{}^T\mathbf{x}$ which has maximum variance subject to being uncorrelated with $\alpha_1{}^T\mathbf{x}$, $\alpha_2{}^T\mathbf{x}$, ... , $\alpha_{k-1}{}^T\mathbf{x}$. Up to p PCs can be found, but in general, most of the variation in **x** will be accounted for by m PCs, where $m \ll p$ [Jol86, p.2]. The following example provides a graphical depiction of the concept. Figure 3.1 is a plot of 50 observations on two highly correlated variables x_1 and x_2 . Transforming these variables to the PCs z_1 and z_2 gives us the plot in Figure 3.2. One can see that the axes went through a rotation such that there is now greater variation in the direction of z_1 than in either of the original variables, but very little variation in the direction of z_2.

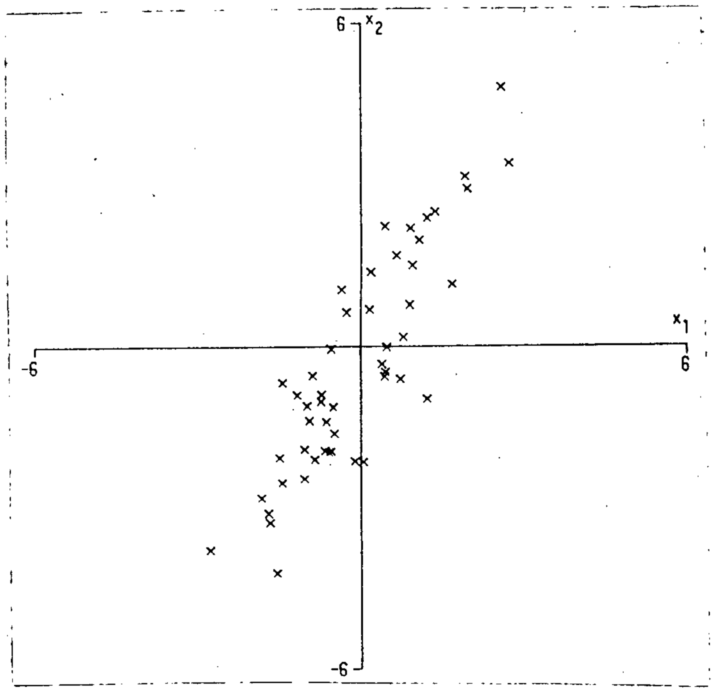


Figure 3.1 Plot of 50 observations on two variables x_1, x_2.

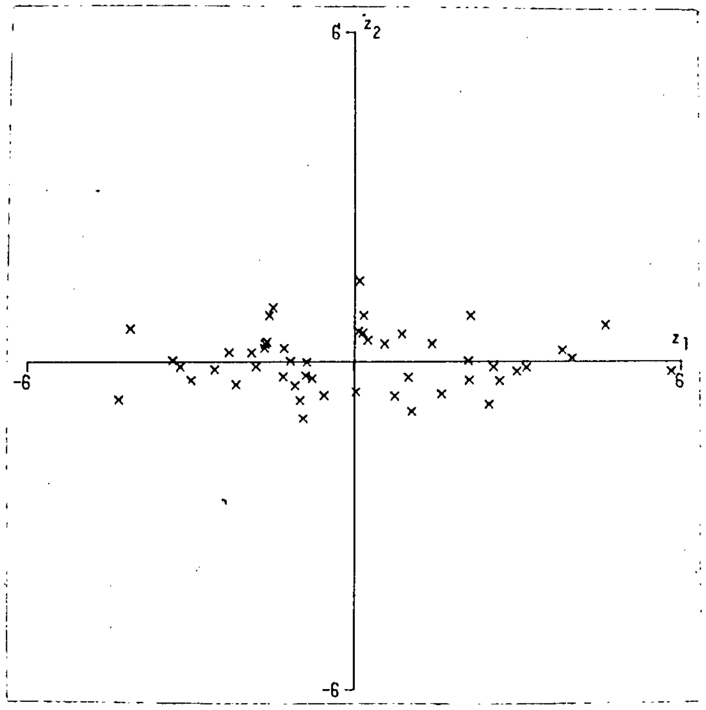


Figure 3.2 Plot of the 50 observations from Fig. 3.1 with respect to their PCs z_1, z_2.

If the data set of n points are not initially centered around the origin of the axis, as is the case in Figure 3.3 below, then the coordinate frame of the data set is first *translated* to a new origin, centered on the average point of the data set $\mathbf{x}_{avg}$ where

$$\mathbf{x}_{avg} = 1/n \sum_{i=1}^{n} \mathbf{x}_i \,, \tag{3.2}$$

and then it is *rotated* to fit the PCs. This is also demonstrated in Figure 3.3.

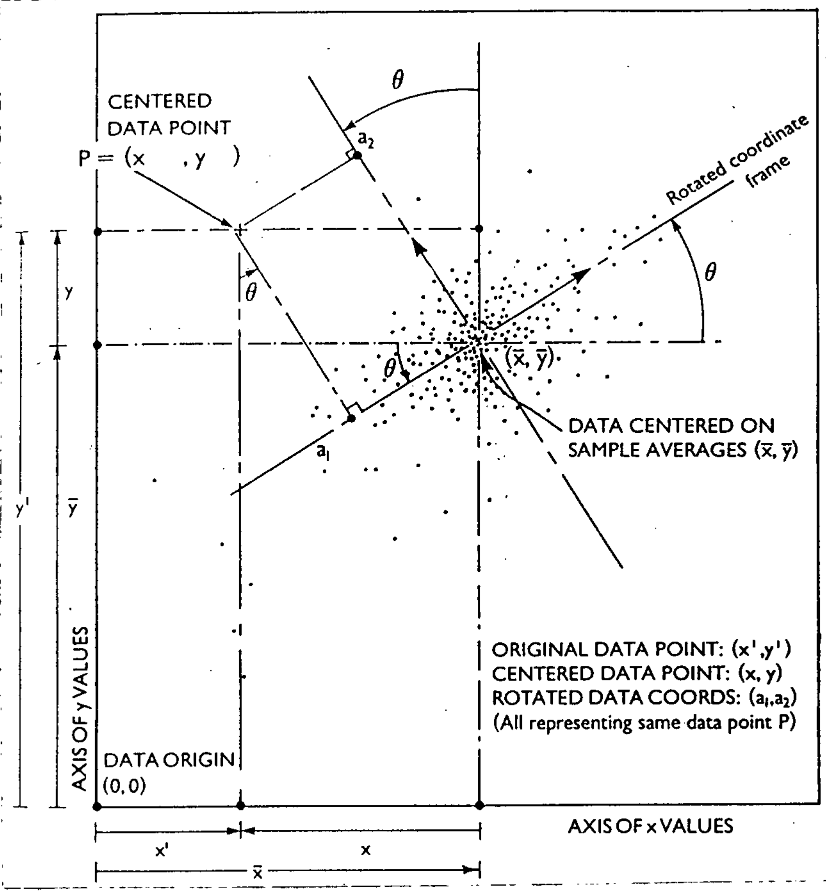

CENTERED
DATA POINT
P = (x , y)
θ
a₂
Rotated coordinate frame
θ
y
θ
θ
(x̄, ȳ)
a₁
DATA CENTERED ON SAMPLE AVERAGES (x̄, ȳ)
y'
ȳ
AXIS OF y VALUES
ORIGINAL DATA POINT: (x',y')
CENTERED DATA POINT: (x, y)
ROTATED DATA COORDS: (a₁,a₂)
(All representing same data point P)
DATA ORIGIN
(0,0)
AXIS OF x VALUES
x'
x
x̄

Figure 3.3 Centering and rotating a data set to fit its principal components.

The coordinates of the data points can now be defined in terms of the PC axes by projecting their centered values on to the principal components:

Original Data Point: $\mathbf{x}$

Centered Data Point: $\phi = \mathbf{x} - \mathbf{x}_{avg}$

Rotated Data Coords: $\omega = A\ (\phi)$ where $A^T = [\alpha_1{}^T, \alpha_2{}^T, \ldots, \alpha_p{}^T]$

(i.e., each term in ω is the projection of ϕ onto each of the PCs)

So the original vector **x** can now be represented as a linear combination of the PCs:

$$\mathbf{x} = \omega_1\alpha_1 + \omega_2\alpha_2 + \ldots + \omega_p\alpha_p \tag{3.3}$$

This representation shows that $\alpha_1, \ldots, \alpha_p$ form a basis of the transformed data space [Pre88].

3.1.2 How to compute PCs

Now that PCs have been defined, let us look at how they are computed. Consider again the vector of random variables **x**. Calculating the covariance matrix of this vector gives us a matrix Σ whose (i, j)-th element is the covariance between the ith and jth elements of **x** for $i \neq j$, and the variance of the jth element of **x** for $i = j$. It turns out that the kth PC, for $k = 1, 2, \ldots, p$, is given by $z_k = \alpha_k{}^T\mathbf{x}$ where α_k is an eigenvector of Σ corresponding to its kth largest eigenvalue λ_k. The derivation of this is given in any textbook on multivariate analysis [Jol86, Pre88, Flu88]. It is important to note that $\mathrm{var}(z_k) = \lambda_k$; in other words, the variance of the kth PC is equal to the kth largest eigenvalue of Σ. Therefore, the sum of the eigenvalues of the covariance matrix gives the total variance of the data set:

$$\textit{total variance} = \sum_{k=1}^{p} \lambda_k \tag{3.4}$$

This detail plays an important role in reducing the number of dimensions of the data. A common heuristic for dimension reduction is to select the eigenvectors that contain between 60 to 80 percent of the variance in the data set [Dun89]. If the eigenvalues are sorted in decreasing order (i.e., $\lambda_k \geq \lambda_{k+1}$) then the first q of the p PCs can be chosen such that

$$0.60 \leq \sum_{k=1}^{q} \lambda_k \,/\, (\textit{total variance}) \leq 0.80\,. \tag{3.5}$$

3.2 Calculating Eigenimages

The *eigenfaces* approach to image analysis and dimension reduction can be briefly described in the following steps: 1) take a sample M from a set of *n* images in a database[9]; 2) find the M principal components of the distribution of the sample images; 3) select the first M' (<< M) principal components (eigenfaces) as features by which to describe the total face population; 4) represent each face in the database as a linear combination of these M' eigenfaces. This method has similarities to the technique that was used in QBIC for reducing the dimensionality of the shape feature vectors. However, the fundamental difference lies in that here the full image is used in the PCA calculations, whereas with QBIC only a specific feature vector that is initially extracted from each image is used in the PCA calculation.

Following are the details of the *eigenfaces* approach. Let the images in a database of size *n* be represented as vectors of intensity values, with dimension N^2:

$$\mathbf{x}_i \in \mathbf{R}^{N^2 \times 1} \quad \text{for } i = 1, \ldots, n \tag{3.6}$$

The vector $\mathbf{x}_i$ is formed by raster-scan ordering the rows of the image into one long vector (i.e., each pixel becomes an element or attribute of the image vector). A training set of M images is selected from this set, where M is less than *n* yet is representative of the total set of images. First, the average image of the training set is calculated:

$$\mathbf{a} = 1/\mathrm{M} \sum_{j=1}^{\mathrm{M}} \mathbf{x}_j \tag{3.7}$$

Next, the covariance matrix of the set is computed:

$$\mathbf{C} = 1/\mathrm{M} \sum_{j=1}^{\mathrm{M}} (\mathbf{x}_j - \mathbf{a})(\mathbf{x}_j - \mathbf{a})^{\mathrm{T}} \tag{3.8}$$

where each term of the sum signifies a dyadic product [SK87].

[9] Turk and Pentland use faces [TP91].

If we set $\phi_j = (\mathbf{x}_j - \mathbf{a})$ then we can write $\mathbf{C}$ as $\mathbf{XX}^T$ where $\mathbf{X}$ is the $N^2 \times M$ matrix $[\phi_1 \; \phi_2 \; \ldots \; \phi_M]$. $\mathbf{C}$ is therefore an $N^2 \times N^2$ symmetric positive matrix. By the laws of linear algebra, a real symmetric matrix can be factored into $\mathbf{C} = \mathbf{Q} \, \Lambda \, \mathbf{Q}^T$ with the orthonormal eigenvectors of $\mathbf{C}$ in $\mathbf{Q}$ and the eigenvalues in the diagonal matrix Λ [Str88, p.296]. This equation is equivalent to $\mathbf{CQ} = \mathbf{Q}\Lambda$ so we can rewrite it as:

$$\mathbf{C}\,\mathbf{q}_j = \mathbf{XX}^T\mathbf{q}_j = \lambda_j\mathbf{q}_j \quad \text{for } j = 1, \ldots, M \tag{3.9}$$

Therefore, each $\mathbf{q}_j$ is an eigenvector of the covariance matrix $\mathbf{C}$, having an associated eigenvalue of λ_j. Since each row of $\mathbf{C}$ is a linear combination of ϕ_j, this means that C has rank M-1, with only M-1 (rather than N^2) non-zero eigenvectors [TP89]. This means that we can save computation time by first solving for the eigenvectors $\mathbf{u}_j$ of the following smaller dimension problem:

$$\mathbf{X}^T\mathbf{X}\mathbf{u}_j = \lambda_j\mathbf{u}_j \quad \text{for } j = 1, \ldots, M \tag{3.10}$$

Having obtained $\mathbf{u}_j$, we can calculate the eigenvectors for $\mathbf{C}$ by pre-multiplying $\mathbf{X}$ on both sides of the above equation to get:

$$\mathbf{XX}^T(\mathbf{Xu}_j) = \lambda_j(\mathbf{Xu}_j) \quad \text{for } j = 1, \ldots, M \tag{3.11}$$

From equations 3.9 and 3.11 we see that the eigenvectors of $\mathbf{C}$ are $\mathbf{q}_j = \mathbf{Xu}_j$. The associated eigenvalues are used to order the eigenvectors, beginning with the eigenvector with the largest eigenvalue. These eigenvectors give the coefficients of the PCs of the image space, and the ordering imposed by the eigenvalues maximizes the variance of the corresponding PC. So, the PCs are linear combinations of the M training images giving rise to the eigenimages. Figures 3.4 - 3.6 provide pictorial examples of the process of eigenimage calculation. Figure 3.4 shows 20 sample images from our data set. Figure 3.5 shows the average of these 20 images, and Figure 3.6 shows the first 4 eigenimages (eigenfaces) which carry 63.7% of the variance in the 20 images of Figure 3.4.

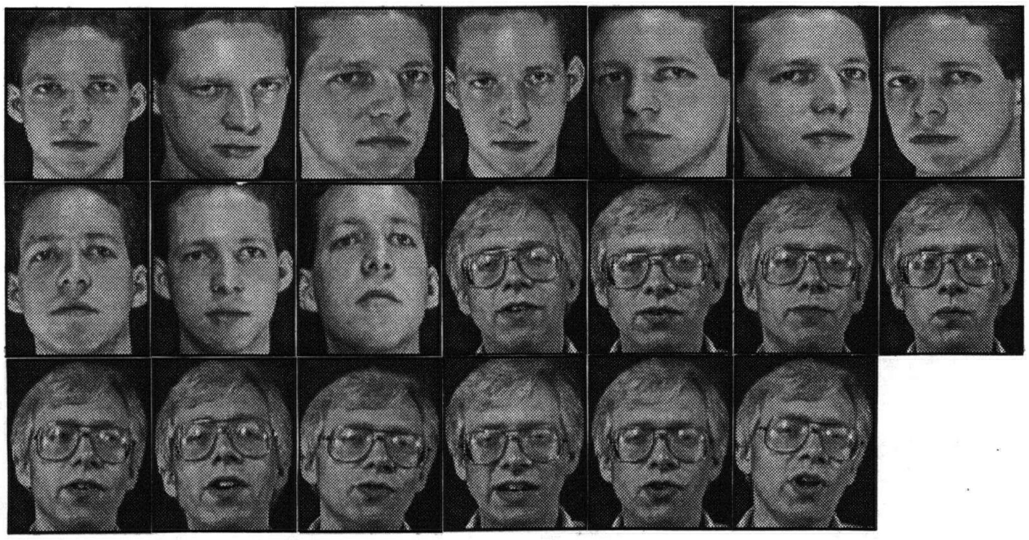


Figure 3.4 Twenty sample face images from our data set.

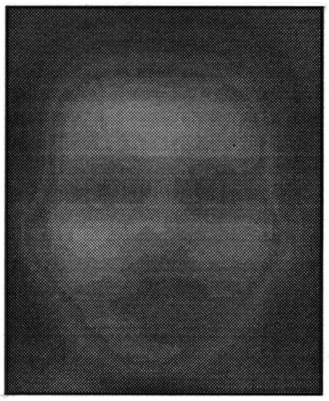


Figure 3.5 Average of the twenty face images in Figure 3.4.

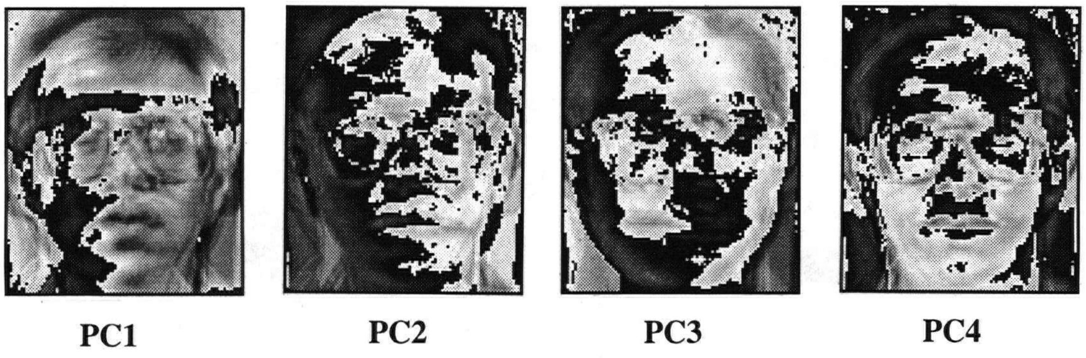


Figure 3.6 The 4 eigenfaces of the images in Figure 3.4.

Using the heuristic for dimension reduction given by equantion 3.5, M' of the M PCs are chosen (where M' is much less than M). All the images in the database can now be transformed into their eigenimage components by projecting them into the new eigenspace. This gives the following set of weights for each image :

$$\omega_k = \mathbf{u}_k^{\mathrm{T}} \phi_k \quad \text{for } k = 1, \ldots, \mathrm{M}'. \tag{3.12}$$

Each image becomes a point in an M' dimensional space $\omega = (\omega_1, \omega_2, \ldots, \omega_{\mathrm{M}'})$. When a query image is provided, it is also projected into this subspace. The appropriate similarity-metric between the weight vector of the query image and the weight vector of a database image is then calculated.

3.3 Implementation and Results

To implement the *eigenfaces* approach, we used the Vista software environment [PKL94] and the MATLAB math package. Vista is designed to support "the modular implementation and execution of computer vision algorithms" [PL94]. The image testbed consisted of 400 gray-scale face images which were acquired from an ftp-site of the Olivetti Research Laboratory in the United Kingdom. The face images consisted of 10 different pictures taken from each of 40 distinct individuals. For some of the subjects the lighting is varied, the facial expressions are varied (i.e., open/closed eyes, smiling/not smiling, etc.), and/or the facial details are varied (i.e., with/without glasses). All images are taken against a dark homogeneous background with the subjects in an upright frontal position. Each image is 112×92 pixels, with 256 possible gray levels per pixel. Before performing PCA, we converted the face images from PGM to the Vista format. The dimensions of the original image vectors were set at 10,304 (112×92). We ran experiments on 2 different sizes of training sets and collected the following data:

1. Using *all* 400 images as the training set (i.e., M = 400) we initially extract 399 (M - 1) PCs. For different percentages of variance, we get the following reduced number of PCs:

% Variance	# of PCs
61.4%	11
70.0%	20
75.3%	30
80.1%	44

Table 3.1 Reduced PCs for different % of Variance using M = 400.

2. Using *one-third* of the 400 images as the training set (i.e., M = 134) we initially extract 133 (M - 1) PCs. For different percentages of variance, we get the following reduced number of PCs:

% Variance	# of PCs
60.4%	9
70.3%	16
75.4%	22
80.1%	30

Table 3.2 Reduced PCs for different % of Variance using M = 134.

These results are consistent with the work of Turk and Pentland [TP91]: for a test case of M = 16 images they extract M' = 7 principal components (they do not indicate what percent variance they use). Furthermore, our test results show that this technique works well for a much larger set of data. We also compare our results to QBIC's implementation and find that in QBIC [F+94] PCA is applied to a specific set of 20 shape measurements that are initially extracted from each image. Their results show that 75% of the variance in their data set is captured in the first 2 PCs, thus reducing the dimensionality of the shape feature space. Our experiments have demonstrated good results for spaces with much higher dimensionality. This indicates that if specific image features are extracted with dimensions much greater than 20, PCA can be very effective in reducing their dimensionality without losing much of the information in the data.

We acknowledge the fact that in our implementation we use a data set that is similar to that of Turk and Pentland (both data sets consist of gray-scale face images). Face images are fairly uniform in their content with limited scene variation. Our intent was to further test this approach on a set of images with greater variety in their content, (e.g., pictures of animals or an art gallery). However, due to time constraints we were unable to prepare and test a second set of image data.

Chapter 4

EVALUATING MULTI-DIMENSIONAL INDEXING STRUCTURES

Once an image space is transformed and compressed using PCA, it should be organized so that similarity match queries can be efficiently performed. A multi-dimensional indexing structure should be wisely chosen so that it can take advantage of the main characteristics of PCA-transformed data. These characteristics are: i) the components are ranked by decreasing variance, ii) the dynamic range of the dimensions of the space are known, and iii) the number of dimensions is still fairly high. In order to find an appropriate structure, we conducted a comprehensive evaluation of a wide range of existing multi-dimensional indexing structures. In this chapter, we first review the structures we looked at in our evaluation. Then we discuss the criteria by which our comparative analysis was conducted. Finally, we present the structures we chose and the rationale behind our choices.

4.1 Review of Multi-dimensional Indexing Structures

There are numerous data structuring techniques in use for representing multi-dimensional point data. They can be divided into two major categories: hierarchical and non-hierarchical data structures. Within these categories, some data structures organize the data while others organize the *embedding space* from which the data are drawn [Sam90]. Below, we begin with a review of the structures we feel are the most relevant to our needs.

4.1.1 Non-Hierarchical Data Structures

Non-hierarchical multi-dimensional data structures decompose a data space in a flat manner, that is, the data points are typically stored either in a sorted list or in the form of a k-dimensional array. Non-hierarchical structures organize the embedding space of the data into regions that contain records. Both non-hierarchical data structures we consider below are bucket methods.

4.1.1.1 Gridfile

The gridfile [NHS84, Sam90] expands on the idea of the fixed grid (or cell) method. The fixed grid method, which divides the record space into equal-sized cells, is essentially a directory in the form of a k-dimensional array with one record per cell [Sam90]. Unlike the grid method, however, the cell block sizes in the gridfile adapt to the distribution of the records and the data holding capacity of the cell blocks. The size of the cell blocks are therefore not uniform. The gridfile aims at providing symmetric access to every key field, and tries to meet the following 2 principles: 1) retrieve records with at most two disk accesses, and 2) handle range queries efficiently. It does this using a *grid directory* that consists of 2 parts. The first is a dynamic k-dimensional array that contains one entry for each cell or *grid block*. The second part is a set of k 1-dimensional arrays called *linear scales*. These scales define the partitioning of each dimension. Each combination of scale partitionings acts as an address to a cell or grid block in the *k*-dimensional array, and each cell contains a pointer to a data bucket that holds the values of the records that fall in the corresponding grid block. As records are added or removed, the following split and merge policy is used: all records in a grid block must be stored in the same data bucket and, several grid blocks can share a single data bucket as long as this union of grid blocks forms a *k*-dimensional hyper-rectangle in the record space [NHS84]. Therefore, either the data buckets split and merge or the *k*-dimensional array of the grid directory splits and merges.

4.1.1.2 Multipaging

Multidimensional paging, better known as multipaging [Mer78, Mer84], is somewhat similar to the gridfile except that in multipaging the data is known *a priori* and no updates are made to the record space. As in the gridfile, multipaging uses linear scales, called *axial arrays*, that define the partitioning of each dimension. These axial arrays are partitioned based on the distribution of the data points along each axis and the application of certain constraints. The constraints used are: page capacity, load factor (total number of data points/total capacity of all pages), tuple probe factor (number of tuple overflows/total number of data points), and page probe factor (number of page overflows/total number of pages). The goal is to have pages which, on the whole, have neither too many nor too few data points. Instead of using a *k*-dimensional array of pointers to the grid blocks, multipaging accesses a data page using an address that is computed directly from the linear scales. This saves space over the gridfile at the cost of requiring bucket overflow areas for densely populated areas of the record space. Multipaging, therefore, cannot guarantee record retrieval with only two disk accesses, as in gridfile. This version of multipaging is known as static multipaging [Mer78]. A second version, called dynamic multipaging [MO82], also exists which provides the ability to dynamically update the record space.

4.1.2 Hierarchical Data Structures

Hierarchical multi-dimensional data structures are ones which recursively partition a set of k-dimensional data points using a tree-like structure. The root of the tree contains the entire hyper-surface of the data space. As one traverses down the structure, each node of the tree contains a successively smaller hyper-surface of the data space.

4.1.2.1 Point-Based Structures

The point quad-tree [FB74] is a generalization of the binary search tree for data with k dimensions. It recursively divides the data space into 2^k partitions (e.g. 4 for 2-D data, 8 for 3-D data, etc.) using a single record as each partition point or node of the tree. This, however, makes deletion a complicated process. The pseudo quad-tree [OvL82] was developed to alleviate this problem. Instead of data points it uses arbitrary points, not in the data set, as the nodes of the tree. These points are chosen such that the remaining set is split in the most balanced manner. This recursive partitioning continues until each partition contains, at most, a single data point of the original set.

The point KD-tree is a k-dimensional binary search tree that is also designed as a generalization of the standard one-dimensional binary search tree [Ben75b, Ben79, BF79]. At each node a single record is used as the partition point and the tree is divided into 2 sub-trees. Only one of the k keys is used for making branching decisions; this key is called the *discriminator key*. Each dimension is used as the discriminator key in a cyclic manner as one progresses down the tree. In two dimensions this means using the x coordinate for making branching decisions at the root and even levels of the tree and the y coordinate for making branching decisions at odd levels. The pseudo KD-tree, like the pseudo quad-tree, uses arbitrary points that are not in the data set to split the data space [OvL82]. The data points themselves appear at the leaf nodes.

The adaptive KD-tree [BF79] is a static data structure meaning that all points must be known *a priori* to build the tree. It is designed in the spirit of the pseudo KD-tree so that the data points are stored only at the leaf nodes. However, each interior node holds the median (along a discriminator key) of the set of points that fall under that node; the discriminator is chosen to be the key for which the spread[10] of values is at a maximum. Therefore, the choice of discriminators is no longer cyclical, as in the KD-tree, and the discriminators are not necessarily the same across nodes on the same level.

A range tree [BM80] is a data structure specifically designed for fast range searching in k dimensions at the expense of high preprocessing and storage costs. It is asymptotically faster than the point quad-tree and KD-tree, but has significantly higher storage requirements [Sam90].

[10]The spread can be measured using any statistical measurement, such as the variance or the distance from the minimum to the maximum value.

4.1.2.2 Region-Based Structures

These structures partition the embedding space of the data. The point-region (PR) quad-tree recursively partitions a k-dimensional data space into 2^k equal-sized regions until there is no more than one data point per sub-division. All data points are contained in the leaf nodes of the tree [Sam90]. The PR KD-tree is similar to the PR quad-tree except that it recursively partitions the space into 2 equal-sized regions along a single dimension [Sam90]. As in PR quad-trees, a sub-division contains no more than one data point and all data points are found in the leaf nodes.

4.1.2.3 Bucket Methods

Bucket methods are designed to ensure efficient access to data stored on disk. They collect data points into sets that correspond to storage units (i.e., pages) of the disk. The bucket PR quad-tree is analogous to the PR quad-tree described above except that leaf nodes are not restricted to holding one record. The bucket capacity can be set ahead of time to c ($c > 1$). Similarly, in a bucket PR KD-tree regions are split in half until no leaf node has more than c data points [Sam90].

The bucket adaptive KD-tree splits the data space along the dimension with the greatest spread, just as does the adaptive KD-tree. However, the leaf nodes are now buckets with capacity c ($c > 1$) [Sam90, MHN84].

R-trees [Gut84, Gre89] are designed as index structures for k-dimensional rectangles or objects. They are height-balanced multiway trees, like the B-tree, and store a set of rectangles in each node. At the leaf nodes these rectangles are the actual objects in the data set. In the non-leaf nodes the rectangular regions stored are the bounding rectangles that enclose all rectangles in descendant nodes. The bounding boxes of non-leaf nodes on a given level can overlap. This poses a problem for R-trees because during search the more sibling nodes there are that overlap, the more paths must be followed for a given spatial query.

R^+-trees [SRF87, Gre89] avoid this problem by clipping rectangles that intersect at the same intermediary level. This creates a search space that is divided into disjoint sub-regions and ensures that no sibling bounding boxes overlap. However, the leaves of an R^+-tree contain duplicate entries so more space is required and there can be more levels in the search path.

R*-trees [B+90], which support both point and region data, were developed to try to overcome some of the problems associated with R-trees and R^+-trees. Whereas R-trees are bound by the order in which rectangles are inserted, R*-trees can forcibly reinsert as many entries as needed to dynamically re-organize the structure during insertion. This re-distribution is done with the aim of reducing the area, perimeter and overlap of rectangles in internal nodes. Experiments have shown that R*-trees perform better than R-trees in accessing region data, and can even outperform the gridfile in accessing point data [B+90].

4.2 Comparative Analysis

The following sections explain the criteria we used to select the best three candidates from the above cited multi-dimensional indexing structures.

4.2.1 Criteria of the Analysis

Our observations of the qualities of PCA-transformed data led us to conclude that there are a number of characteristics a multi-dimensional indexing structure should have which would make it suitable for organizing this type of data. These characteristics, and why we feel they are important for our analysis, are enumerated below in a question-and-answer format:

1) *How does the size of the structure scale up with the number of dimensions in an image space?*

   As we saw in Chapter 3, the dimensions of the image vectors we initially use are extremely high. Therefore, even after transforming the feature space and reducing the dimensionality with PCA, the new feature space still has a large number of dimensions. Therefore, we look for structures that can handle large numbers of dimensions without exploding in size.

2) *Are the attributes or dimensions prioritized during construction of the index? (i.e., is the structure symmetric or non-symmetric?)*

   For PCA-transformed data the components of the vectors are ranked by decreasing order of variance. This means the components are, in a sense, prioritized since each successive component adds less to the overall information being stored in the data. Therefore, we look for structures that can capture this prioritization.

3) *What are the different kinds of searches this structure can handle?*

   As was mentioned in the introduction, we are concerned with performing fixed-radius near neighbor search. Therefore, we look for structures that can support either fixed-radius search or range search (with which to approximate fixed-radius).

4) *Does the structure adapt to the distribution of the data in the record space?*
5) *In particular, how does the structure perform with non-uniformly distributed data?*

   PCA-transformed data is typically not uniformly distributed. Therefore we look for structures that do not become seriously unbalanced with non-uniformly distributed data and whose performance does not severely degrade.

6) *Is the structure suitable for a static or a dynamic set of data?*

   For this thesis, we focus on the ability of the structure to perform efficient searches rather than on its ability to handle updates. Therefore, we choose to use a static data set for our experiments and do not concern ourselves with the ease with which data points can be inserted or deleted from the indexing structure.

4.2.2 Answers to the Criteria Questions

We can immediately eliminate the point-based hierarchical structures (point quad- and KD-trees, pseudo quad- and KD-trees, and adaptive KD-tree) from further analysis as they explode in size for large numbers of multi-dimensional data points. The main reason for this is that each node in these structures stores a single data point. The non-bucket region-based hierarchical structures can also be eliminated right away because they divide the embedding space of the data into fixed-size regions that hold no more than one data point. This again leads to an explosion in index size for large data sets. Therefore, we focus our analysis on those hierarchical and non-hierarchical indexing structures that utilize *buckets*. Of the R-tree variants, the one that has been shown to be most suitable for the access of point data is the R*-tree [B+90]. Hence, we proceed our analysis with the following structures:

- Bucket PR quad-tree (B_PR_Q-Tree)
- Bucket PR KD-tree (B_PR_KD-tree)
- Bucket adaptive KD-tree (BA_KD-tree)
- Gridfile
- Multipaging
- R*-tree

Following are the answers of each of these structures to the questions we asked in the previous section:

1) How does the size of the structure scale up with the number of dimensions in an image space?

B_PR_Q-Tree:	Each internal node splits into 2^k sons, therefore the tree branches explode out very quickly.
B_PR _KD-tree:	Each internal node splits into 2 sons. Since the number of levels of the tree increases linearly as a multiple of the number of dimensions, the number of nodes increases exponentially.
BA_KD-tree:	The number of levels of the tree increase as the dimensions increase but not linearly, as in B_PR_KD-tree. This is because nodes are not required to divide in a cyclic manner along every dimension. This also means that the increase in number of nodes is not as sharp.
Gridfile:	The designers claim that it works for up to 10 dimensions. The grid directory array becomes too large and unwieldy with increasing numbers of dimensions.
Multipaging:	Since there is no grid directory array the structure does not explode in size with increasing numbers of dimensions. It will, however, take increased computation time to find appropriate linear scale partitions.
R-Tree*:	If fan-out of the nodes remains > 2, it seems to be quite robust for higher dimensions. Experiments show that it will work well for approximately 20 dimensions [F+94, A+95].

The BA_KD-tree, Gridfile, Multipaging and R*-tree are better than the top two at handling a large number of dimensions. Gridfile, however, seems to be the weakest of the four.

2) Are the attributes or dimensions prioritized during construction of the index?

B_PR_Q-Tree:	No. All dimensions are treated equally.
B_PR _KD-tree:	Yes. The dimensions are accessed at each level in a cyclical predetermined order (usually dimension 0, 1, ..., *k*).
BA_KD-tree:	Yes. The dimensions are accessed at each level based on which dimension has the greatest spread for the subset of data points being examined.
Gridfile:	No. It is designed for symmetric access where every dimension is treated as a primary key.
Multipaging:	No. Also considered to have symmetric access.
R-Tree*:	No. All dimensions are treated equally.

The KD-trees are the only structures that have the ability to prioritize dimensions. The BA_KD-tree is especially interesting because it prioritizes dimensions based on the spread of data points along each dimension.

3) What are the different kinds of searches this structure can handle?

B_PR_Q-Tree:	Point, range, and fixed-radius near neighbor search.
B_PR _KD-tree:	Point, range, fixed-radius near neighbor, and *n*-nearest neighbor search.
BA_KD-tree:	Point, range, fixed-radius near neighbor, and *n*-nearest neighbor search.
Gridfile:	Point and range search.
Multipaging:	Point and range search.
R-Tree*:	Point, range, and *n*-nearest neighbor search.

The top three structures can handle fixed-radius near neighbor search, while the bottom three can only approximate it by supporting range search.

4) Does the structure adapt to the distribution of the data in the record space?

B_PR_Q-Tree:	No. It performs a regular decomposition of the embedding space splitting it up along predefined lines.
B_PR _KD-tree:	No. It performs a regular decomposition of the embedding space splitting it up along predefined lines.
BA_KD-tree:	Yes. The record space is split up based on the distribution of the data and not along predefined lines.
Gridfile:	Yes. The record space is split up based on the distribution of the data and not along predefined lines.
Multipaging:	Yes. The record space is split up based on the distribution of the data, not along predefined lines.
R-Tree*:	Yes. The record space is split up based on the distribution of the data, not along predefined lines.

The last four structures are sensitive to the distribution of the data points and are less likely to be unbalanced, i.e., they will have fewer overflowed or underused buckets.

5) In particular, how does the structure perform with non-uniformly distributed data?

B_PR_Q-Tree:	Performs poorly with non-uniformly distributed data (i.e., data that has clusters). The tree will have many empty nodes

	and will become unbalanced therefore decreasing the efficiency of data retrieval.
B_PR _KD-tree:	Performs poorly with non-uniformly distributed data. Has the same problems as B_PR_Q-Tree.
BA_KD-tree:	No significant effect on storage or retrieval efficiency with non-uniformly distributed data since the structure is sensitive to the distribution of the data.
Gridfile:	No significant effect on storage or retrieval efficiency with non-uniformly distributed data. Only exception is that it does not perform particularly well on range queries.
Multipaging:	Performs poorly with non-uniformly distributed data. The pages become unevenly occupied, i.e., either overflowed or empty pages.
R-Tree*:	Robust against non-uniformly distributed data.

Of the six structures, BA_KD-tree, Gridfile and R*-Tree deteriorate the least in performance with non-uniformly distributed data.

6. Is the structure suitable for a static or a dynamic set of data?

B_PR_Q-Tree:	Suitable for both, although updating data is somewhat complex.
B_PR _KD-tree:	Suitable for both, although updating data is somewhat complex.
BA_KD-tree:	Suitable for static data only. All data points must be known a priori.
Gridfile:	Suitable for both, although it is typically used with dynamic data.
Multipaging:	There are 2 types of multipaging: one for static data and the other for dynamic data.
R-Tree*:	Suitable for both, although it is typically used with dynamic data.

Any of the 6 structures can be used for organizing a static data set. BA_KD-tree, however, is specifically designed for it.

4.2.3 Results of the Analysis

Out of the six structures examined above, the three multi-dimensional indexing structures we chose for further experimental evaluation were:

- The Bucket Adaptive KD-tree
- The R*-Tree
- Multipaging

The reasons for our choices are: 1) these structures do not seem to grow as rapidly as the rest when the number of dimensions increases; 2) all of these structures either approximate fixed-radius near neighbor search using range search or they support the operation itself; 3) all three structures are sensitive to the distribution of data and are not divided along predefmined lines; and 4) the performance of these structures does not degenerate seriously with non-uniformly distributed data. The added bonus that comes with the bucket adaptive KD-tree is that it can prioritize access to dimensions according to the spread of data along each dimension. Finally, we feel that it would be interesting to compare the performance of 3 structures that come from 3 different families of data structures.

Although we chose these structures for further experimental evaluation, due to time constraints it became clear that we would only be able to implement the bucket adaptive KD-tree ourselves. We attempted to acquire the code for the two other structures from elsewhere but were unable to. In their stead we made the following subsitutions: in place of the R*-tree we used a deferred-split R-tree structure and in place of multipaging we used a gridfile structure. Both were acquired from the University of Maryland. We feel that the deferred-split (D-S) R-tree is an acceptable substitution for the R*-tree because the D-S R-tree has a similar data structure as the R*-tree and so has comparable storage requirements, and the performance of the R*-tree deteriorates for large numbers of dimensions and data set sizes [A+95] making it little better than the D-S R-tree. We feel that the gridfile is an acceptable substitution for multipaging because both structures are non-hierarchical data structures that provide symmetric, direct access to the records. At high dimensions and large data set sizes multipaging may save space by not requiring a *k*-dimensional array, but this is at the expense of requiring many overflow buckets [Sam90]. Therefore, the search performance of gridfile and multipaging should also be comparable. The implementation of each structure and the results of the experimental evaluations we conducted are presented in the next chapter.

Chapter 5

IMPLEMENTATION DETAILS & EXPERIMENTAL EVALUATION OF THE KD-TREE, R-TREE & GRIDFILE

In this chapter we describe the implementation of the three multi-dimensional indexing structures that were chosen as a result of the comparative analysis presented in Chapter 4. We also present the results of the experimental evaluation of these structures and show why the bucket adaptive KD-tree is the structure best suited for PCA-transformed data.

As was mentioned previously, the R-Tree and gridfile structures were acquired, by permission of Christos Faloutsos, from the Department of Computer Science at the University of Marlyand, College Park. We implemented the bucket adaptive KD-tree structure ourselves using Samet's description in *The Design and Analysis of Spatial Data Structures* [Sam90]. The structures are all implemented in C source code and run on Sun SPARC workstations. The operating system is SunOS Release 4.1.3.

5.1 Indexing Structure Implementations

5.1.1 Bucket Adaptive KD-tree

5.1.1.1 Data Structures

Our implementation of the bucket adaptive KD-tree creates two files: a data file (`.dat` file) that stores the data buckets or pages, and an index file (`.dir` file) that stores the internal nodes of the tree. This implementation uses two page buffers: one is for writing and reading data pages to and from the `.dat` file, and the other is for writing and reading index pages to and from the `.dir` file. The page size we use is 1024 bytes. This can be changed to test the performance of the structure using larger (2048 or 4096 bytes) or smaller page sizes (512 bytes). The number of dimensions of the data points can also be varied by changing an option on the command line.

The data points are stored as integers in data buckets that are referenced by the leaf nodes of the tree. The data points can not be deleted since we treat the data set as a static one. Each data bucket is the size of a single page. A data bucket holds all *k* dimensions of each of the data points in it along with a record identifier or data ID, so the greater the number of dimensions, the smaller is bucket capacity. In a bucket adaptive KD-tree the data buckets will not all appear on the same level.

The layout on disk of a page from the data file for 2-dimensional data points looks as follows:

```
          |<------ DATA_PT[0] ----->|........|<-DATA_PT[buck_size-1]-->|
┌---------┬--------┬--------┬--------┬---\\---┬--------┬--------┬--------┬--------┐
|no_points|dim0_val|dim1_val|data_id|........|dim0_val|dim1_val|data_id|        |
└---------┴--------┴--------┴--------┴---\\---┴--------┴--------┴--------┴--------┘
|<-HEADER>|
          |<-------DATA_SIZE------->|
|<--------------------------------- PAGE_SIZE --------------------------------->|
```

The total number of points stored in a data page or bucket is saved at the head of the page. The maximum number of points that can be stored in a page is calculated as:

```
buck_size = (PAGE_SIZE - HEADER)/DATA_SIZE;
```

The internal nodes of the tree hold a discriminating coordinate and the median of that coordinate, along with pointers to the left and right sons of the node. The data structure that is used in the bucket adaptive KD-tree for both the internal and leaf nodes has the following components:

```
typedef struct kd_intnode{
  int node_num;
  int median;
  int disc_coord;
  int lt_buk_blkno;
  int rt_buk_blkno;
  struct kd_intnode *lt_son;
  struct kd_intnode *rt_son;
} KD_INTNODE;
```

`node_num:`	is the number of the node, which translates into the offset of the node in the `.dir` file.
`disc_coord:`	is the discriminating coordinate of the node.
`median:`	is the median value of the discriminating coordinate.
`lt_son:`	is a pointer to the left son of the node; it is NIL if there is no left son or if the left son is a data bucket.

`rt_son:`	is a pointer to the right son of the node; it is NIL if there is no right son or if the right son is a data bucket.
`lt_buk_blkno:`	is the offset of the page in the `.dat` file that contains the data points of the left data bucket.
`rt_buk_blkno:`	is the offset of the page in the `.dat` file that contains the data points of the right data bucket.

The pages in the index (`.dir`) file contain information about the nodes in the tree. The information that is stored per node in the `.dir` file consists of the following six integer values:

1) the median value of the discriminating coordinate,
2) the discriminating coordinate,
3) a code that identifies the left son as either an internal node or a leaf node,
4) the node number of the left son (if the left son is an internal node) or the page offset into the `.dat` file (if the left son is a data bucket),
5) a code that identifies the right son as either an internal node or a leaf node,
6) the node number of the right son (if the right son is an internal node) or the page offset into the `.dat` file (if the right son is a data bucket).

The layout on disk of a page from the index file looks as follows:

```
         |<--------------------- NODE[0] -------------------->|    |<NODE[max-1]>|
┌--------┬------┬----------┬---------┬----------┬--------┬---------┬-\\-┬------\\-----┬--┐
|no_nodes|median|disc_coord|lft_code|lft_offset|rt_code|rt_offset|....|             |  |
└--------┴------┴----------┴---------┴----------┴--------┴---------┴-\\-┴------\\-----┴--┘
|<HEADER>|
         |<--------------------- NODE_SIZE ------------------>|
|<----------------------------------- PAGE_SIZE ---------------------------------->|
```

The header of the first page of the index file holds the total number of pages in the file and the number of nodes stored in the first page. The header of each remaining page only holds the number of nodes stored in that page. The maximum number of nodes that can be stored in an index page is calculated as:

```
    nodes_per_page = (PAGE_SIZE - HEADER)/NODE_SIZE;
```

Since data points are not stored at the nodes of the tree, the number of nodes that can fit into a page of the index file does not change with an increase in the number of dimensions.

5.1.1.2 Insertion

To create the bucket adaptive KD-tree all the data points must be known *a priori*. This means that instead of inserting the data points one-at-a-time into the tree structure, the program is given a file with all of the *k*-dimensional data points. The following steps are then taken recursively until all the data points in that file are placed into the KD-tree:

1. Calculate the variance along each dimension and find the dimension with the greatest variance or spread (we use the difference between the minimum and maximum values of the points as the spread of each dimension). The dimension with the greatest variance is called the *discriminating coordinate*.

2. Find the median value of the discriminating coordinate.

3. Allocate space for a new node. Store both the discriminating coordinate and its median in the current node.

4. Compare the value of the discriminating coordinate for each data point to the median value. Place the data points with a value greater than or equal to the median under the right son of the node and the data points with a value less than the median under the left son of the node.

5. If the number of data points under the left son is greater than the bucket capacity then repeat steps 1 - 5 for those data points. But, if the number of data points under the left son is less than or equal to the bucket capacity then a data bucket is created, the data points are stored in it, and it is inserted into the data file. The same process is then applied to the data points under the right son.

5.1.1.3 Search

There are two kinds of search operations available with the KD-tree structure: point search and range search. We will concentrate on the range search implementation since point search is not a concern of ours in this thesis. We perform range search at this stage of the experimentation to approximate fixed-radius near neighbor search. Therefore, the queries consist of a query point and a radius that define the hyper-sphere of the query. This hyper-sphere is approximated by a hyper-rectangle which is generated by using the given radius to calculate the range of the query along each dimension.

Range search is performed by descending down the tree structure from its root, comparing the query range to the median value of the discriminating coordinate of a node. If the lower bound of the query range for the discriminating coordinate is greater than or equal to the median, then the search continues down the right son. If the upper bound of the query range for the discriminating coordinate is less than the median, then the search moves down the left son. If the query range for the discriminating coordinate overlaps the median, then the search continues down both the left and the right sons, starting with the left son. When a data bucket is reached, a sequential search is performed on all the data points stored in the bucket checking all dimensions, to determine which points fall within the query range. Those that fit within the bounds of the query range are returned as answers to the query.

In our implementation, when performing a search operation, the complete tree structure is placed into main memory. To get a proper count of the number of pages accessed during a search, and to correctly compute the search time, accessing an index page is simulated. This means that every time a node is reached that resides on a page other than the page that is currently in the index buffer, the new page is read from disk into the buffer.

In this way a page access occurs even though the contents of that page are not actually used in the course of the search.

5.1.2 Deferred-Split R-Tree

This implementation generates two files: the `.rtree` file that holds all the nodes of the tree, and the `.info` file that keeps track of the space utilization of the tree for the purpose of facilitating insertions and deletions. The data records are stored in the leaf nodes. Since our records are points and not spatial objects we treat the points as "degenerated rectangles" [B+90], that is, each point becomes a *k*-dimensional rectangle where, for example in the 2-d case, the lower left corner and the upper right corner of the rectangle are both set to the x and y values of the data point itself. Non-leaf nodes contain a *k*-dimensional rectangle which is the minimum bounding rectangle (MBR) of all rectangles in the lower node's entries.

Every page in the `.rtree` file is a node of the R-tree, whether internal or leaf node. The page size is set at 1024 bytes, and the number of in-memory page buffers is set at 50. The swapping of index pages from memory to disk is based on a Least-Recently-Used page scheme. To work with different dimensions one must change the value of the dimension constant in the header file and recompile the program. The branching factor (i.e., the maximum number of entries per node) changes based on the page size and the number of dimensions of the data points. The minimum requirement of entries per non-root node is set at 50% of the branching factor. In the R-tree, all leaf nodes appear on the same level.

5.1.2.1 Insertion

The basic operations of the R-tree implementation which we are interested in are Insert and Range Search. Data points are required to be inserted one-at-a-time as rectangles. In the 2-D case this means inputting the following values: low-x, low-y, high-x, high-y. A data ID is automatically generated for the inserted rectangle.

Insertion in an R-tree is similar to insertion in a B-tree. New index records are added to the leaf nodes; nodes that overflow are split, and splits propagate up the tree. A split works in the following manner: when a new entry needs to be added to a full node containing M entries, it is necessary to divide the collection of M+1 entries between two nodes. The division should be done in a way that makes it as unlikely as possible that both new nodes will need to be examined on subsequent searches. Since the decision to visit a node depends on whether its MBR overlaps the search area, the total area of the two MBR's after a split should be minimized. This implementation provides several options for techniques of deferred splitting. However, since we use *area enlargement*, which is the default splitting heuristic, we will not discuss the other options. This heuristic finds a sibling node that has the minimum joint area with the overfull node. This R-tree implementation is functional only up to 11 dimensions, when using positive and negative integers.

5.1.2.2 Search

To perform range search, this implementation requires the user to input a *k*-dimensional search rectangle and the program returns all records that overlap with this rectangle. However, since we want to approximate a fixed-radius near neighbor search we modified the code so that we could input a query point and a radius and the program itself would compute the appropriate search rectangle for the query. The search algorithm descends the tree from the root in a manner similar to a B-tree, following the nodes whose MBR overlap the search rectangle. More than one subtree under a node may need to be searched. Our modifications included adding code for counting the number of pages accessed and recording the time required to perform a search.

5.1.3 Gridfile

In the gridfile the *grid directory* consists of 2 parts: i) a dynamic *k*-dimensional array that contains one entry for each *grid block* (each entry is a pointer to the corresponding data bucket), and ii) a set of *k* 1-dimensional arrays or *scales* that define the partitioning of each dimension. There are 4 files that are generated by this program: the index (`.dir`) file that stores the *k*-dimensional array, the data (`.dat`) file that holds all the data buckets, the `.sca` file that stores the cutpoints of each linear scale, and the `.info` file that holds general information about the gridfile structure. The page size is set at 1024 bytes and there are 2 page buffers that are utilized -- one for writing to and reading from the data file, and the other for writing to and reading from the index file.

The product of the size of the *k* scales determines the size of the *k*-dimensional array, and the positional value of the scale elements are the coordinates of the position (or offset) of a grid block in the index file. The grid block (cell) stores an integer value that is the page offset into the data file where the data bucket that holds the data points for that grid block exists. (Each data bucket corresponds to a page in the data file.) There may be several grid cells that point to the same page in the data file. Such a group of grid blocks is termed a *data block*. The index file can hold a maximum of 512 data bucket references per page.

5.1.3.1 Insertion

Again, we only look at the Insert and Range Search operations of the gridfile. When a new gridfile is created, the user is requested to first input the upper and lower bounds of each dimension. Then, the data points must be inserted one-at-a-time. The program checks that the points do not fall outside of the specified bounds. As the number of points increases and a data block overflows the data block must be split. To perform a split, every dimension is first tested to see which allows a better distribution of the points in that data block. The dimension with the best distribution is chosen as the one along which the split is to occur. The midpoint of that dimension for that data block is then computed. The scale of that dimension is checked to see if the computed midpoint already exists as one of the cutpoints in the scale. If this is the case, the data points are re-distributed into 2 data buckets, and the index and data files are updated. If the midpoint is not a cutpoint of the scale then a new cutpoint is added to the scale, the whole index file is restructured to include the new resulting grid blocks, the data points are re-distributed into new data buckets and the data file is updated.

5.1.3.2 Search

To perform a range search the user is required to input the upper and lower range for each dimension of the search rectangle. Since we approximate fixed-radius near neighbor search, we modified this process so that we could input a query point and a radius. The search rectangle is then automatically generated. The search algorithm looks for all data blocks whose bounds overlap with the bounds of the search rectangle.

5.2 Data Testbed

The testbed of images we used for this thesis was explained in detail in Chapter 3. We selected one-third of the 400 images for the training set by choosing every third image. After performing PCA on this training set we found that 70% of the variance of this set was captured in the first 16 PCs. To get the weight vectors that would represent the whole image set, we projected each of the 400 images onto this set of 16 PCs. The resulting 16-dimensional data points were converted into integers and stored in a data file.

We used this initial set of 400 16-D data points to generate larger sets of data. This was done by first extracting the dynamic range of each dimension, and then randomly generating data points that were constrained by the bounds of the extracted dynamic ranges. Each component of a data point was an integer value that was either positive or negative. Eight different data set sizes were generated in this way (400, 500, 1,000, 2,000, 4,000, 8,000, 16,000, and 50,000) for each of 5 different dimensions (2, 4, 6, 8 and 10.) The reason we only used up to 10 of the 16 dimensions is because, as we mentioned before, the R-tree implementation could only manage up to 11 dimensions of positive and negative integers before it crashed. We decided that rather than spend time trying to fix someone else's code, we would limit our tests to a maximum of 10 dimensions and concentrate on our research questions.

5.3 Results of Experimental Evaluation

These structures were compared both in terms of their space efficiency and their performance at range search. Unfortunately, the gridfile implementation had several bugs when it was first acquired. Even after necessary modifications were made, it would crash when we tried to index more than 2,000 data points that had greater than 2 dimensions, and when we tried indexing more than 4,000 data points that had 2 dimensions. As the dimensions and the data set size increased, the gridfile structure would take longer and longer to insert data points, until it would eventually hang indefinitely. This was undoubtedly because the *k*-dimensional array would become prohibitively large with high dimensions and large data set sizes. We ran some range searches on those gridfile indexes we were able to build but found that their performance was woefully bad. For 2 dimensional data the gridfile accessed up to 5 times more pages than the R-tree and up to 8 times more pages than the bucket adaptive KD-tree. As a result of these problems, we decided to concentrate our experimental evaluation only on the two tree structures. The following sections present the results of tests run on the bucket adaptive KD-tree and the deferred-split R-tree.

5.3.1 Storage Cost

The storage cost of the KD-tree and the R-tree, in terms of total number of pages required, is displayed in Figure 5.1 on the next page. The page size is set at 1024 bytes.

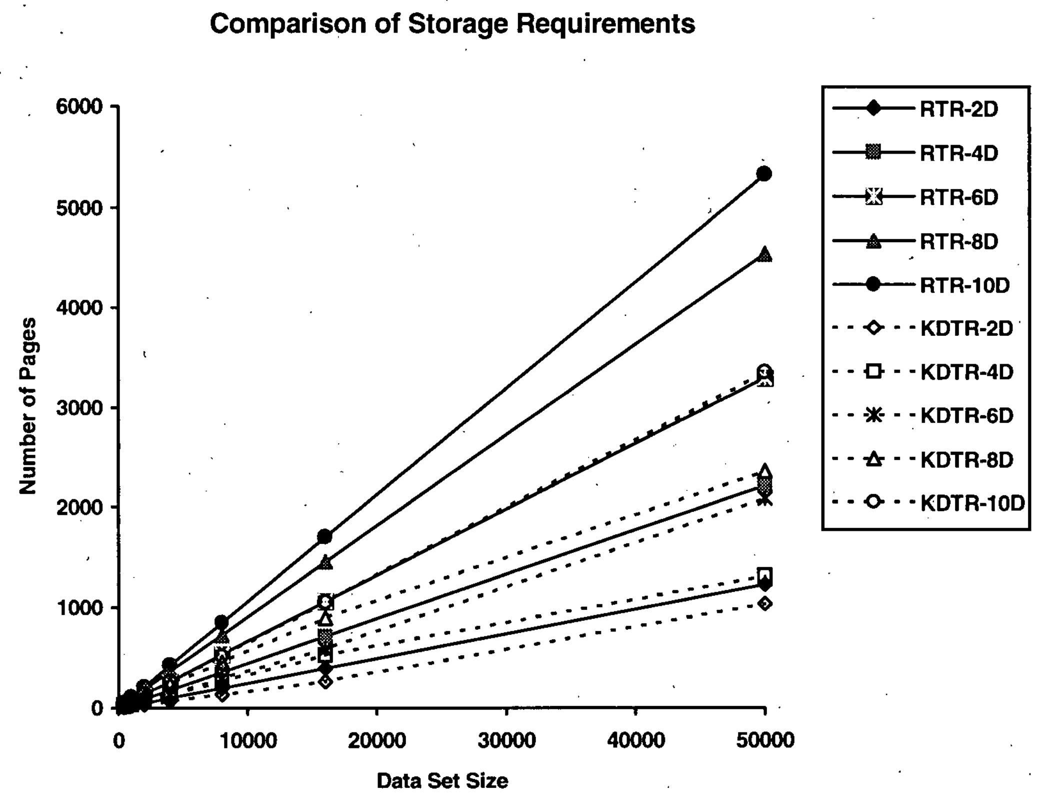


Figure 5.1 Comparison of page usage of R-tree and KD-tree structures over increasing data set sizes and increasing dimensions

For both structures, the storage requirements rise linearly with an increase in the data set size, but our implementation of the KD-tree uses less space than the R-tree. The KD-tree structure requires, on average, 1.57 times less storage than the R-tree structure. In comparing the two implementations we find that for the R-tree the greatest cost in storage comes from the number of nodes in the tree, whereas for the KD-tree the greatest cost in storage comes from the number of data buckets and not the internal nodes. As the number of dimensions increase, the number of entries per node of the R-tree decreases thus increasing the number of nodes in the tree and hence the total number of storage pages required. In the KD-tree, however, an increase in the number of dimensions creates a decrease in the capacity of the data buckets, thus increasing the number of data buckets (i.e., storage pages) required. But, as we see in Figure 5.2 below, the capacity of the KD-tree data buckets is always approximately twice that of the R-tree nodes. Therefore, as dimensions increase, the increase in storage requirements for the KD-tree is not as great as it is for the R-tree. The page capacities seen here are dependent on the particular implementations we used and may compare differently if other implementations were used.

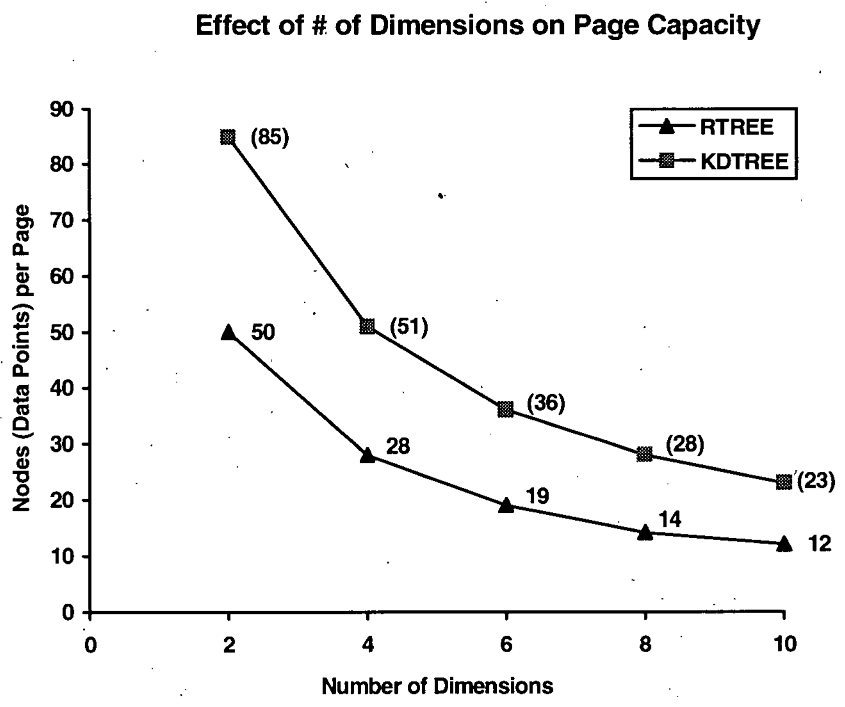


Figure 5.2 Effect of increase in dimensions on capacity of R-tree nodes and KD-tree data buckets.

5.3.2 Search Performance

To test the range search performance of the KD-tree and R-tree, 10 queries were used. The query points were generated the same way as were the data points for the testbed. Numbers were generated for each dimension of the query such that they would fall within the dynamic ranges of the respective dimensions. We created the query points in this way to ensure that they would not be far from the data space. A radius was also specified for each data set. The radii were chosen such that for each data set sizes there would be approximately 50 data points that would actually fall within the hyper-sphere of the query. The average overall search time and the average number of page accesses required by each structure was recorded and compared.

The discriminating power (i.e., the ratio of the number of retrieved data points to the total number of points in the set) of both the structures turned out to be the same because both structures compared the data points at the leaf nodes to the query rectangle, and retrieved only those data points that fell within the query range. The graph in Figure 5.3 shows the change in discriminating power as the number of dimensions increases.

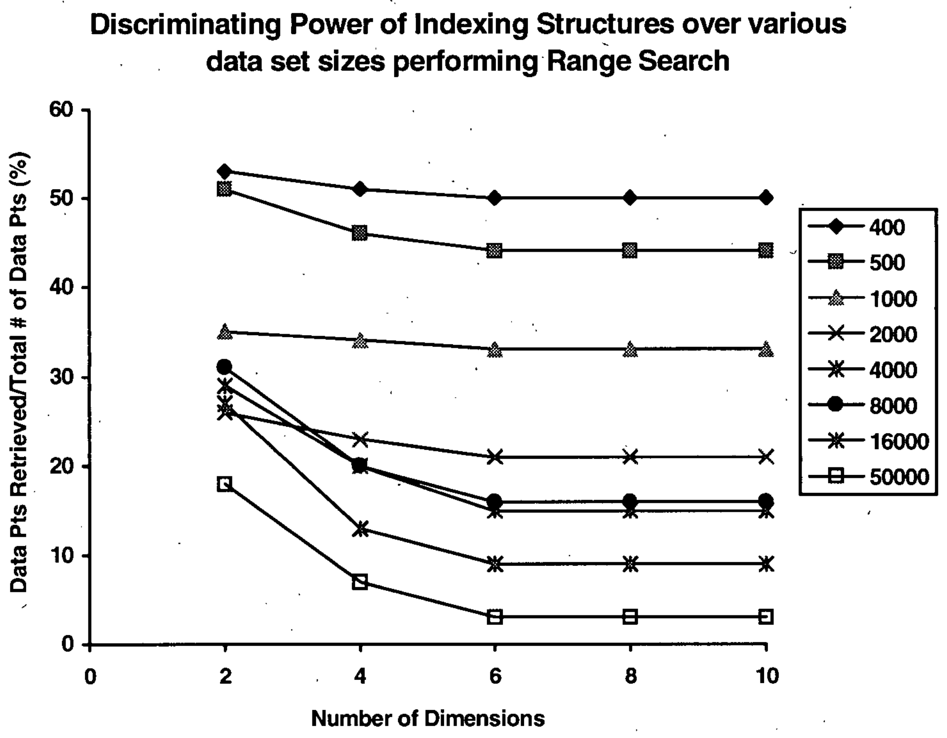


Figure 5.3 Discriminating Power of Indexing Structures

The interesting aspect of the graph in Figure 5.3 is that the ratio of points retrieved in response to the query to the total number of points in the data set decreases steadily from 2 to 6 dimensions (i.e., the discriminating power improves). However, it remains the same from 6 to 10 dimensions. This indicates that, when performing range search, the addition of dimensions does not improve the discriminating power of the structures. We conjecture that there are two factors for this phenomenon: 1) the data points are PCA-transformed so the later dimensions have significantly smaller dynamic ranges than the first few dimensions and add less information about the data points; 2) we are using range search therefore each dimension is treated independently of the other. These two factors in combination mean that: a) the ranges of the higher dimensions fall completely within the range of the query rectangle so the higher dimension values will always satisfy the query, and b) since each dimension is tested independently of the rest, only the first few dimensions that overlap the query ranges actually provide discriminating power to the structure -- the later dimensions are unable to further refine the discriminating power. We will come back to this point in the next chapter when we look at the design of the optimized BA_KD-tree.

The more dramatic difference between the performance of the two structures lies in a comparison of the number of pages accessed and the overall search time during the range search. The number of pages accessed is a count of the total number of both index and data pages that are entered during the search. It is not just a count of the number of disk reads. In this way, the type of buffering scheme used by each structure does not affect the number of pages accessed. We needed to do this since the buffering scheme for the R-tree and the KD-tree were different. Figures 5.4 and 5.5 on the next page graphically demonstrate the average number of page accesses made by the KD-tree and the R-tree over the 10 queries.

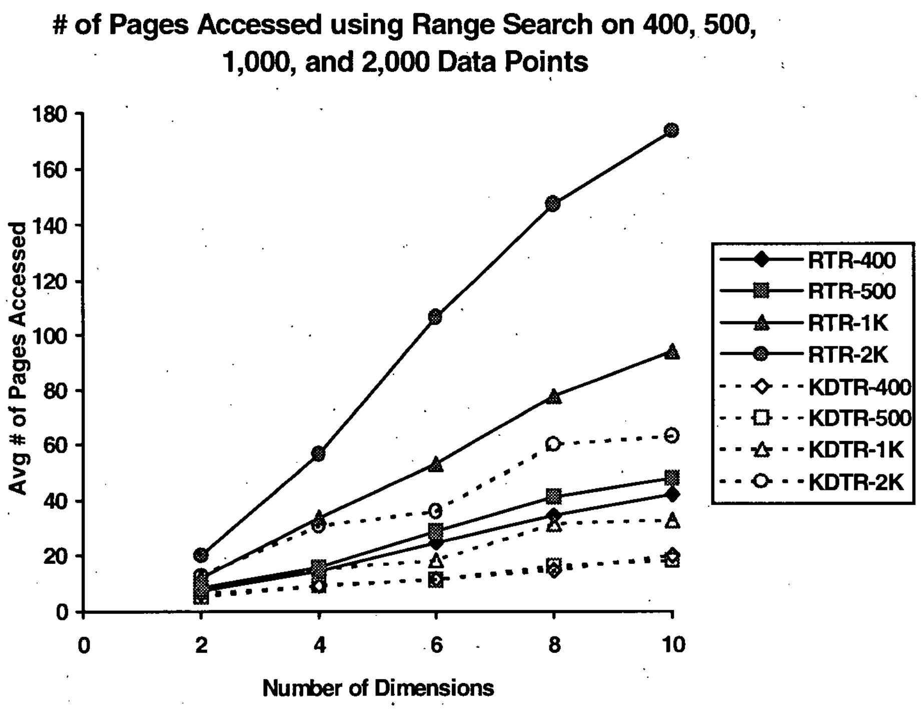


Figure 5.4 Comparison of # of page accesses for 400, 500, 1K, and 2K data points.

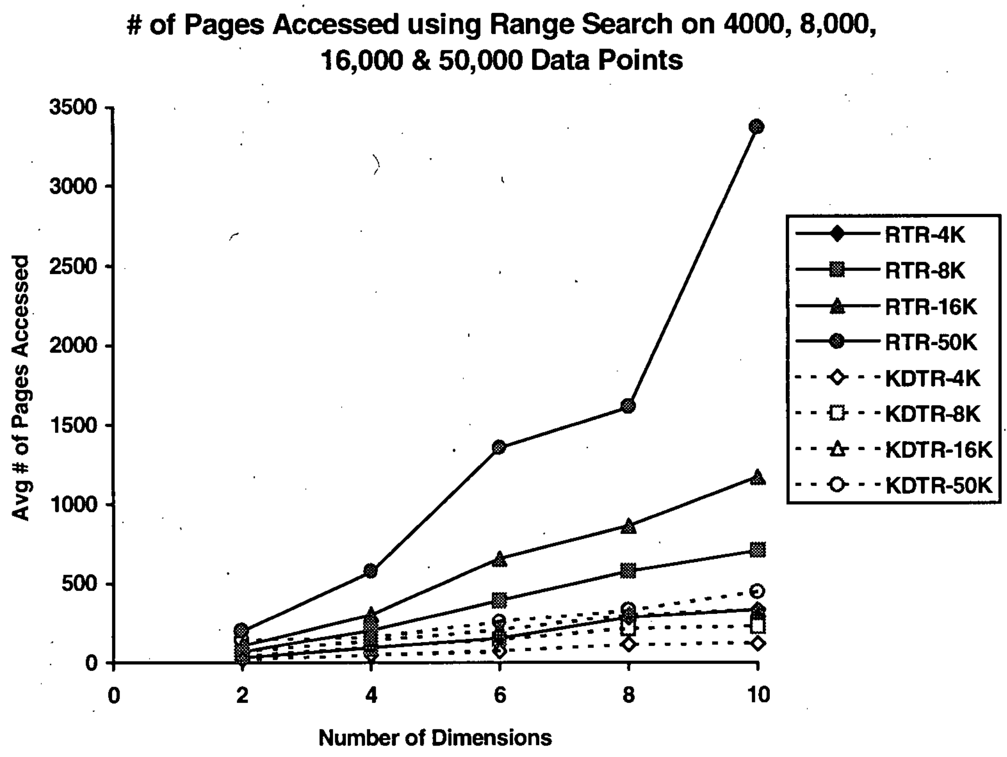


Figure 5.5 Comparison of # of page accesses for 4K, 8K, 16K and 50K data points.

The x-axis in these figures is the number of dimensions and the y-axis is the average number of pages accessed. Each line in the graph represents a different data set size: from 400 to 2,000 data points in Figure 5.4, and from 4,000 to 50,000 data points in Figure 5.5.

From these graphs we can see that the bucket adaptive KD-tree performs range search with much fewer page accesses than does the R-tree structure. This distinction is particularly noticeable as the size of the data set increases and the number of dimensions increase. At the highest level, with 50,000 data points and 10 dimensions, the number of page accesses made by the KD-tree is approximately 7.6 times less than that of the R-tree. The average overall search time for the 10 queries are shown in Figures 5.6 and 5.7 below. Overall search time includes CPU time and I/O time.

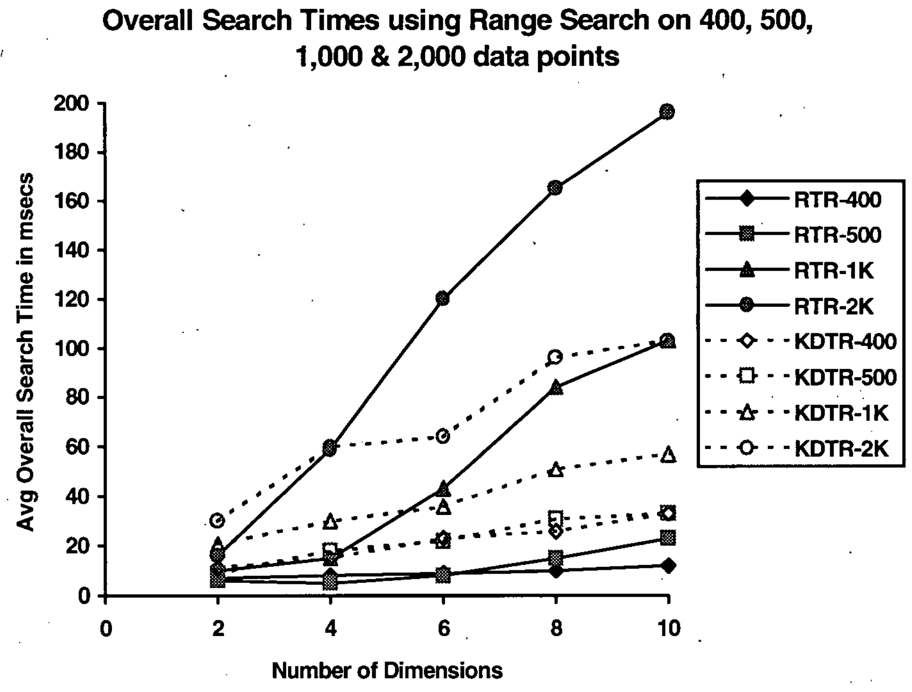


Figure 5.6 Comparison of overall search times for 400, 500, 1K and 2K data points.

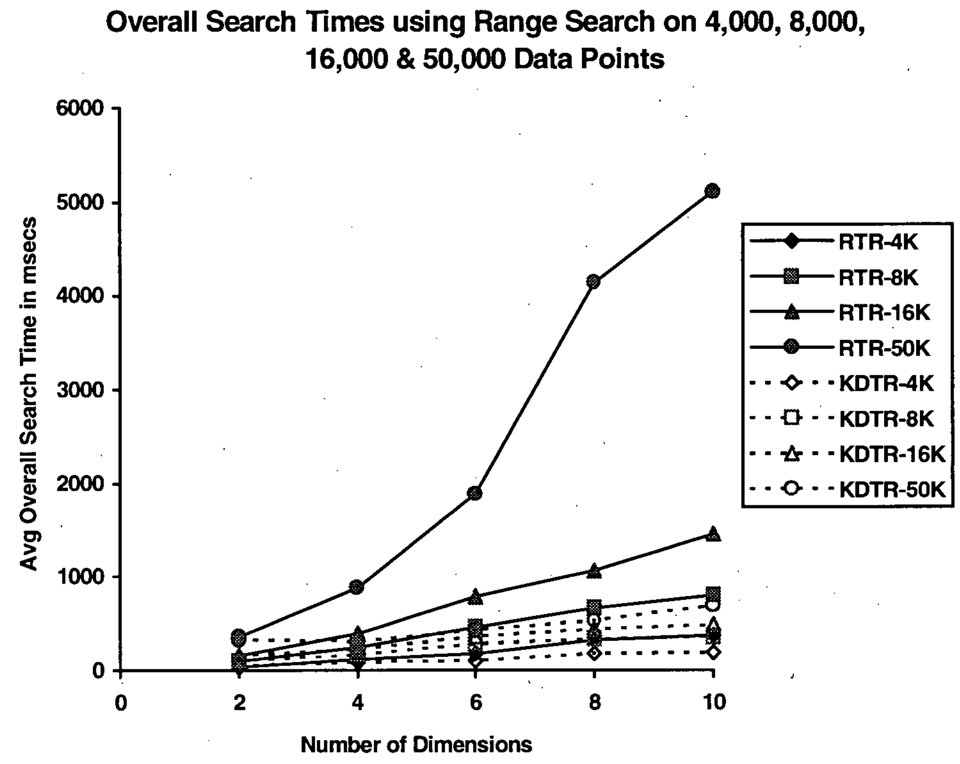


Figure 5.7 Comparison of overall search times for 4K, 8K, 16K and 50K data points.

The x-axis in Figures 5.6 and 5.7 is the number of dimensions and the y-axis is the average overall search time in milliseconds. Each line in the graph represents a different data set size: from 400 to 2,000 data points in Figure 5.6, and from 4,000 to 50,000 data points in Figure 5.7.

In Figures 5.6 and 5.7, just as in the previous two figures, one can see that the performance of the R-tree deteriorates more rapidly than that of the KD-tree as the data size and the dimensions increase. For 50,000 data points with 10 dimensions the KD-tree is 7.4 times faster than the R-tree. However, in Figure 5.6 we see that for 400 and 500 data points, although the R-tree accesses more pages than the KD-tree, its overall search time is less. This is due to the fact that the R-tree uses 50 page buffers, in comparison to the KD-tree's 2 page buffers, resulting in much fewer page faults. It seems that the rate of increase in the overall search time for both structures follows the rate of increase of the number of page accesses. This indicates that, particularly for high dimensions and large data sets, the search time is dominated by I/O time (i.e., page accesses).

5.4 Conclusion

The results of the comparative analysis and the experimental evaluation indicate that the bucket adaptive KD-tree is the best choice of the three structures for handling large numbers of multi-dimensional PCA-transformed data. From the experimental evaluation we find that the BA_KD-tree clearly outperforms the other structures in range search, in terms of both number of pages accessed and overall search time, and that the storage cost of our implementation of the BA_KD-tree is much less than the gridfile and almost two-thirds that of the R-tree. From the comparative analysis in Chapter 4 we find that the KD-tree adapts to the distribution of points in the data space and is able to handle non-uniformly distributed data without becoming very unbalanced. More importantly, in the construction of the KD-tree the dimensions are prioritized based on the one with the greatest spread. This means that in the highest levels of the tree, the discriminating coordinates of the nodes are the first few PCs. This, in turn, indicates that in a search, the initial pruning decisions are based on the dimensions which carry most of the information about the data set. This feature can be used to help us optimize the tree and quickly reduce search space. Finally, the BA_KD-tree is designed for use with a static data set which is what we need for the work of this thesis.

Chapter 6

IMPLEMENTATION & EVALUATION OF OPTIMIZED BUCKET ADAPTIVE KD-TREE

In this chapter we discuss the implementation of the optimized bucket adaptive KD-tree structure and we describe the fixed-radius near neighbor search algorithm. Then we compare the performance of the optimized structure to its original version.

In Chapter 5 we observed that the discriminating power of the indexing structures we evaluated increased only over the first 6 dimensions (see Figure 5.3). Adding more dimensions to the data points did not improve the discriminating power of these structures. We probed further into this phenomenon and found that our conjecture in Chapter 5 was right. Due to the fact that the data points are PCA-transformed, the dimensions with less variance have significantly smaller dynamic ranges than those with greater variance. This means that the ranges of the former dimensions are a subset of the corresponding ranges of the query rectangle. Below is a sample of the dynamic ranges for the first 10 dimensions of the data set with 8000 points:

Dimension Number	**Upper Bound**	**Lower Bound**	**Spread**
1	709	-634	1343
2	620	-596	1216
3	292	-275	567
4	291	-285	576
5	257	-300	557
6	228	-167	395
7	157	-126	283
8	114	-109	223
9	111	-88	199
10	85	-115	200

Table 6.1 Dynamic Range and Spread of first 10 PCs for 8000 data points.

Since we use range search for the evaluation, we define it in the following manner:

$$\bigwedge_{i=1}^{k} (\text{dist}_i(\mathbf{x}, \mathbf{q}) \leq r) \tag{6.1}$$

where $\mathbf{x}$ is a data point, $\mathbf{q}$ is the query point, k is the number of dimensions, r is the distance from the query point, and $\text{dist}_i(\ )$ is the difference between the i-th elements of $\mathbf{x}$ and $\mathbf{q}$. Each dimension is compared against the query ranges independent of the other dimensions. Therefore, given our type of data, the values of the higher dimensions will always satisfy the query constraints and hence will not contribute to further pruning the search space.

Fixed-radius near neighbor search would be much more effective in pruning the space because the values of the dimensions are not treated independently but are used in a summation. This means that each dimension adds to the total distance between a data point and the query. As the number of dimensions increase, the resulting distance value is more and more refined. This can be represented in the following equation:

$$\sum_{i=1}^{k} (\text{dist}(\mathbf{x}_i , \mathbf{q}_i))^2 \leq r^2 \tag{6.2}$$

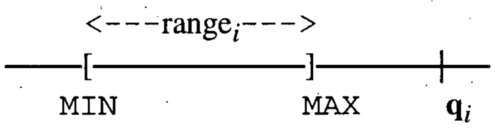
where all variables are defined the same as for equation 6.1. We optimize the original bucket adaptive KD-tree and implement fixed-radius near neighbor search to see if we can improve over the performance of the original tree. Furthermore, we make the search more efficient by incorporating a few of the ideas that were outlined by Bentley in 1975 [Ben75a]. The two key techniques we implement are what we call the *Early Fail Test* and the *Early Success Test*. These are explained in detail below.

6.1 The Early Fail Test

The Early Fail Test (EFT) checks to see if the minimum distance between a query point and the hyper-rectangle of the range of a sub-tree is greater than a fixed threshold r^2, where r is the distance (or radius) from the query point. If the minimum distance exceeds r^2 it indicates that the entire sub-tree is outside of the hyper-sphere of the query and hence that sub-tree can be eliminated from the search space.

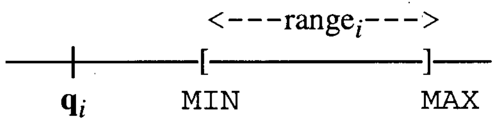
To calculate the minimum distance between a query point and a hyper-rectangle, the sum of the squares of the *minimum* differences between the elements of the query point and the corresponding ranges of the hyper-rectangle is computed. To find the minimum difference between the i-th element of a query point and the i-th range of a hyper-rectangle, there are 3 conditions that need to be checked:

a) if the i-th element of the query point is larger than the maximum value of the i-th range of the hyper-rectangle, as in the diagram,



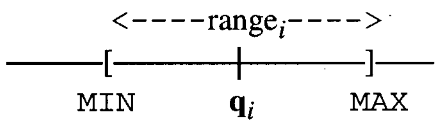
then the minimum difference is: $\mathbf{q}_i - \mathtt{MAX}(\text{range}_i)$

b) if the i-th element of the query point is smaller than the minimum value of the i-th range of the hyper-rectangle, as in the diagram,



then the minimum difference is: $\mathtt{MIN}(\text{range}_i) - \mathbf{q}_i$

c) if the i-th element of the query point intersects with the i-th range of the hyper-rectangle, as in the diagram,



then the minimum difference is *zero*.

To reduce the number of calculations and speed up this test, we note that the we only need to know if

$$\sum_{i=1}^{j} (\text{dist}(\mathbf{x}_i , \mathbf{q}_i))^2 > r^2 \quad \text{where } j < k. \tag{6.3}$$

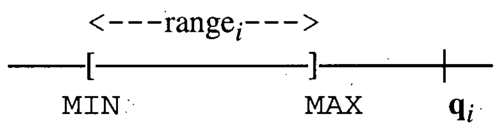
As we sum the squares of the minimum differences of the elements, the summation may exceed the threshold before every element is examined. Therefore, we can save some computation time by summing the squares of the minimum differences only until they exceed r^2. This works particularly well with PCA-transformed data since most of the information of the data is stored in the first few dimensions.

6.2 The Early Success Test

The Early Success Test (EST) is used to determine whether or not the hyper-rectangle of the range of a sub-tree falls entirely within a fixed radius from the query point. If this is true, then it indicates that all of the data points in the leaf nodes under that sub-tree satisfy the query, hence further testing down that sub-tree is not needed. To find out whether the hyper-rectangle is within a fixed radius from the query point one must compute the maximum distance between the elements of the query point and the corresponding ranges of the hyper-rectangle. If this distance is less than or equal to r^2 (as defined previously), it indicates that the hyper-rectangle is indeed within the hyper-sphere of the query.

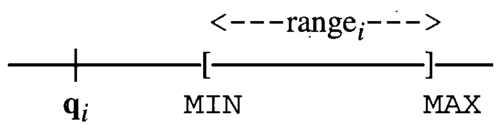
The maximum distance is computed by summing the squares of the *maximum* differences between the elements of the query point and the corresponding ranges of the hyper-rectangle. To find the maximum difference between the i-th element of a query point and the i-th range of a hyper-rectangle, there are again 3 conditions that need to be checked:

a) if the i-th element of the query is larger than the maximum value of the i-th range of the hyper-rectangle, as in the diagram,



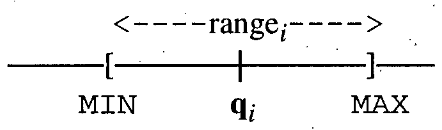
then the maximum difference is: $\mathbf{q}_i$ – MIN(range$_i$).

b) if the i-th element of the query is smaller than the minimum value of the i-th range of the hyper-rectangle, as in the diagram,



then the maximum difference is: MAX(range$_i$) – $\mathbf{q}_i$.

c) if the i-th element of the query intersects with the i-th range of the hyper-rectangle, as in the diagram,



then the maximum difference is: MAX[MAX(range$_i$) – $\mathbf{q}_i$, $\mathbf{q}_i$ – MIN(range$_i$)].

In the EST, to save computation time, we also sum the squares of the maximum differences only until they exceed r^2.

6.3 Changes to Tree Structure

The EFT and EST can only work if certain modifications are made to the tree structure. Added information must be stored in the internal and leaf nodes. We describe the features of the new structure below. The page size is set to 1024 bytes and we use two page buffers, just as in the original BA_KD-tree structure.

The primary change in the node structure is that the minimum and maximum values of the range of the hyper-rectangle of a sub-tree are stored at each node along with the page offset values of the data buckets that can be found under that sub-tree. There are two data structures that are used to define the internal and leaf nodes:

1) NXT_INFO structure:

```
typedef struct nxt_info
{
  int **bkt_rng;
  char *bkt_nums;
} NXT_INFO;
```

`bkt_rng:`	is a pointer to a two-dimensional array that holds the minimum and maximum values of the range of a hyper-rectangle. `bkt_rng[0]` holds the *k* minimum values and `bkt_rng[1]` holds the *k* maximum values.
`bkt_nums:`	is a pointer to a string that holds the page offsets of all the data buckets that fall under a sub-tree.

2) KD_INTNODE structure:

```
typedef struct kd_intnode
{
     int node_num;
     int median;
     int disc_coord;
     int lt_buk_blkno;
     int rt_buk_blkno;
     struct kd_intnode *lt_son;
     struct kd_intnode *rt_son;
     NXT_INFO *lt_nxt_info;
     NXT_INFO *rt_nxt_info;
     struct kd_intnode *parent;
} KD_INTNODE;
```

`node_num, median, disc_coord, lt_buk_blkno, rt_buk_blkno, lt_son, rt_son:`

These elements of the structure are identical to those in the original bucket adaptive KD-tree and so will not be re-defined here.

`lt_nxt_info:`	is a pointer to a `NXT_INFO` structure that holds pertinent information on the left son.
`rt_nxt_info:`	is a pointer to a `NXT_INFO` structure that holds pertinent information on the right son.
`parent:`	pointer to the parent of the node.

A leaf node is distinguished from an internal node because a leaf node will have the `lt_buk_blk_no` and `rt_buk_blkno` filled with a data bucket page offset value and the `lt_son`, `rt_son`, `lt_nxt_info` and `rt_nxt_info` all set to NIL.

With this scheme, the header of a data bucket page in the `.dat` file is changed to include the upper and lower ranges of the data points in that page. As we build the tree, when a data bucket is reached, the upper and lower ranges of the data points are extracted from the header of the bucket and passed up to its parent leaf node. These ranges are merged and recursively passed up to the parent nodes all the way to the root. Similarly, the data bucket page offsets (i.e., the page numbers of the data buckets in the `.dat` file) are passed up and stored at each node.

During execution of a search we store the entire tree in main memory. Therefore, the contents of the nodes stored in the index (`.dir`) file remain the same as they were for the original KD-tree: discriminating coordinate of node, median value of discriminating coordinate, left son code identifier, left son offset value, right son code identifier, and right son offset value. However, it would be more natural to keep the index on disk and simply bring into memory those index pages that are required. If this is done, it means that what is stored on disk for each node would have to include the following additional data: i) the upper and lower ranges of the sub-trees of the left and right sons, and ii) the first and the last data bucket identifiers (i.e., page offsets) from the list of data buckets that fall under the sub-tree rooted at that node.

6.4 Search Algorithm

The new fixed-radius near neighbor search is performed in the following manner:

Step 1: Starting at the root of the tree, the left son undergoes the EFT. To do this we use the information that is stored at the root about the son. If the left son does not fail, i.e., $\Sigma\ \text{dist}_i^2 \leq r^2$, then, *Step 1.1*: the left son undergoes the EST. To do this, once again we use the information that is stored at the root about the son. If the left son passes this test, i.e., the whole sub-tree rooted at the son satisfies the query, then

Step 1.1.1: the search down that path ends and the data bucket page offsets that are also stored at the root are used to sequentially read the data buckets into memory and retrieve the data points in them as answers to the query.

Step 1.2: If the left son does not pass the EST then the search continues down the left son repeating Steps 1 - 1.2. If the left son is a data bucket and not an internal node then that data bucket is read into memory and its contents are checked for matching data points.

Step 2: If the left son does fail in the EFT, i.e. $\Sigma \text{ dist}_i^2 > r^2$, then the search down that path ends and we begin testing the right son, starting with Step 1.

6.5 Results of Experimental Evaluation

To test the performance of the optimized bucket-adaptive KD-tree we ran the same 10 queries that were used in the previous experimental evaluation. We compared the results of these runs to those of the original bucket adaptive KD-tree. In the best case, when the entire tree is stored in main-memory and the number of nodes per page of the index file is the same as that of the original BA_KD-tree, we find that the optimized BA_KD-tree leads to a significant reduction in the number of page accesses and overall search time. For 50,000 data points and 10 dimensions we observe a reduction of 70% for both the number of page accesses and the overall search time. The overall search time for the optimized BA_KD-tree includes the extra computations necessary for performing the EFT and EST. The graphs in Figures 6.1 and 6.2 illustrate these savings.

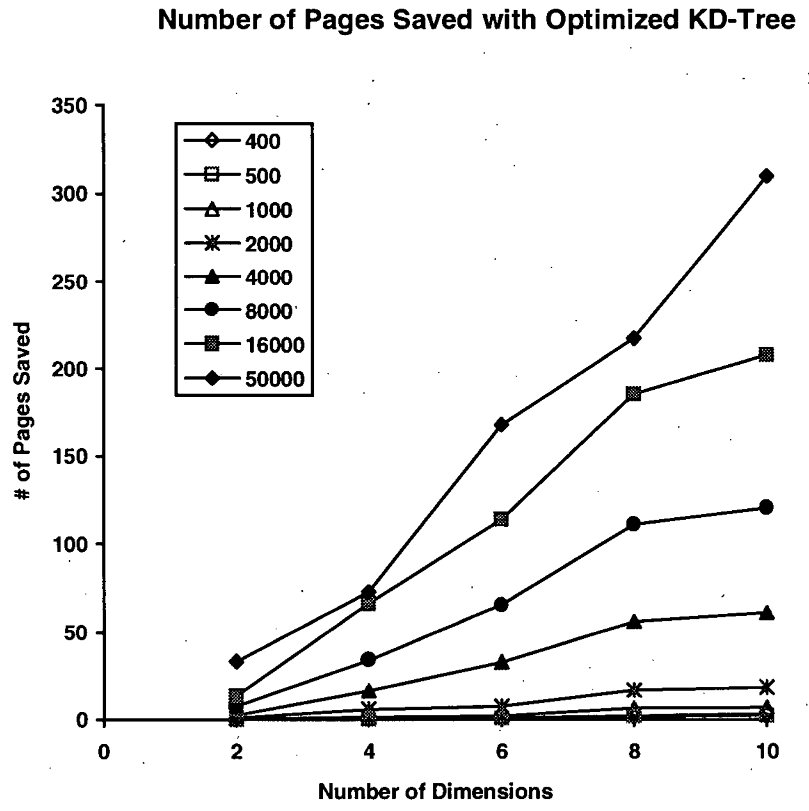


Figure 6.1 Number of Page Accesses Saved using the Optimized BA_KD-tree and fixed-radius search, for 8 different data set sizes and 5 different dimensions

In Figure 6.1 the x-axis is the number of dimensions of the data points and the y-axis is the average number of page accesses *saved* by the optimized bucket adaptive KD-tree over the 10 queries. Each line in the graph represents the savings for a particular data set size, from 400 data points at the bottom, to 50,000 data points at the top. One can see that at low dimensions and small data set sizes, the savings in number of pages accessed is fairly small. But, as the dimensions increase and the data set sizes get large the optimized BA_KD-tree saves an increasing number of page accesses. At 10 dimensions, for the largest data set (50,000), the optimized tree saves over 300 page accesses. Figure 6.2 below shows the ratio of the number of pages accessed by the original BA_KD-tree over the number of pages accessed by the optimized BA_KD-tree for all data set sizes and all dimensions. We find that there is an asymptotic increase in the magnitude of this ratio as the data set sizes increase, and that this magnitude is larger for greater numbers of dimensions.

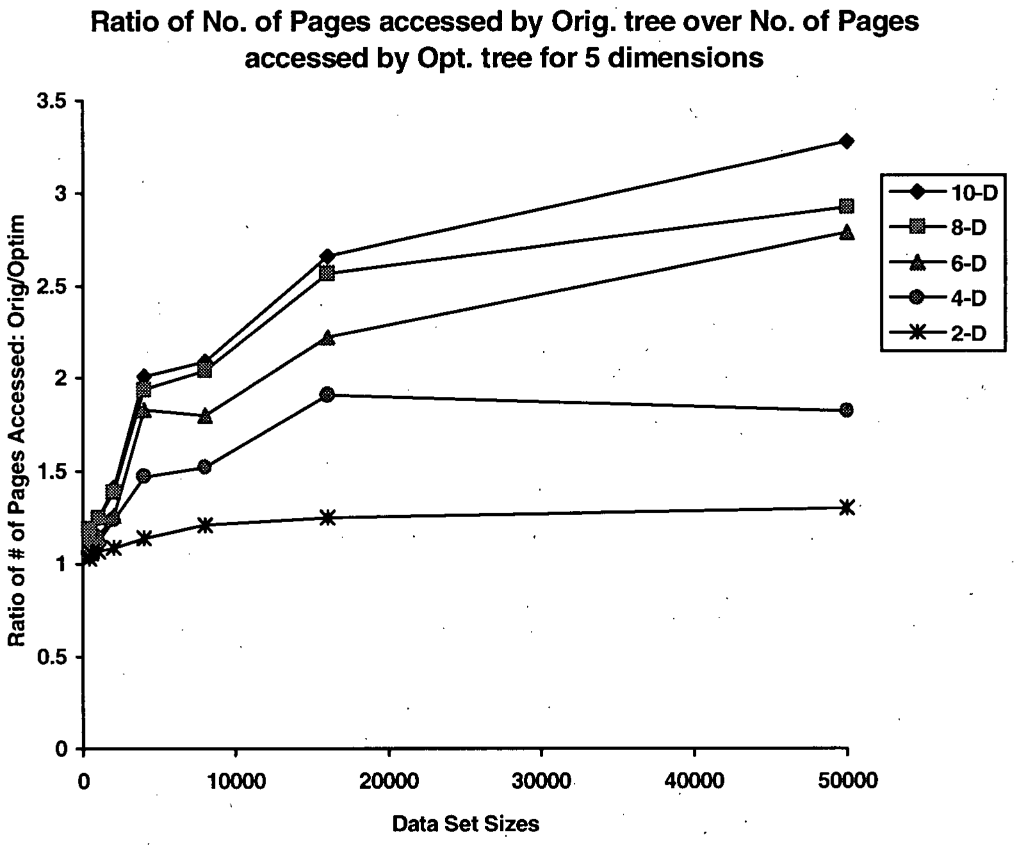


Figure 6.2 Ratio of no. of pages accessed by Original BA_KD-tree over no. of pages accessed by Optimized BA_KD-tree for all dimensions.

At the high end of the savings, we find that for a data set of 400 10-dimensional data points the optimized BA_KD-tree accesses 1.19 times fewer pages than the original BA_KD-tree, while for a data set of 50,000 10-dimensional data points the optimized BA_KD-tree accesses 3.28 times fewer pages.

The *percentage* of savings in pages accessed over the original BA_KD-tree are shown in Figure 6.3 for the larger data set sizes.

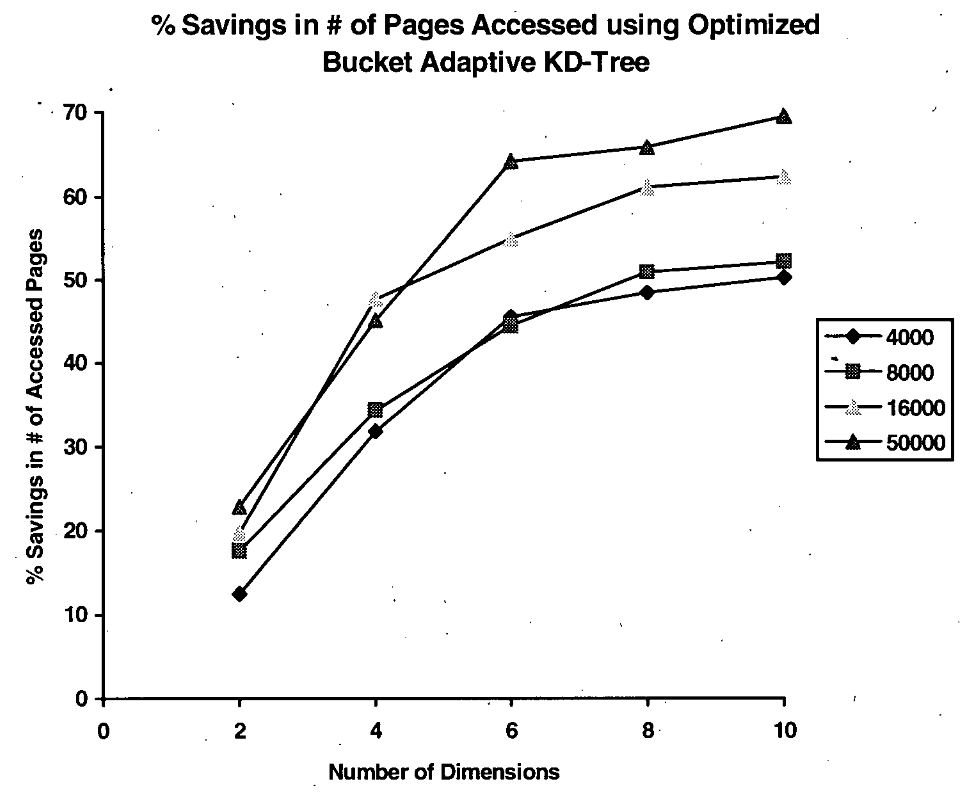


Figure 6.3 Percentage of Savings in # of Pages Accessed by Optimized BA_KD-tree for large data sets.

The x-axis represents the number of dimensions of the data points and the y-axis shows the percentage of savings in the number of pages accessed by the optimized KD-tree. Each line in the graph represents the savings for a particular data set size, from 4,000 data points at the bottom, to 50,000 data points at the top.

The savings vary from 12.5% on the low end (i.e., for 4,000 data points and 2 dimensions) to 69.5% on the high end (i.e., for 50,000 data points and 10 dimensions). For all of these data set sizes, the savings rise dramatically from 2 to 6 dimensions then begin to level off at higher dimensions. Nevertheless, this leveling happens at over 40% in savings. The large percentage of savings at the high end of the spectrum is very encouraging as this is where the performance of all existing multi-dimensional indexing structures typically begins to rapidly deteriorate.

The overall search time saved by using the optimized KD-tree is shown in Figure 6.4.

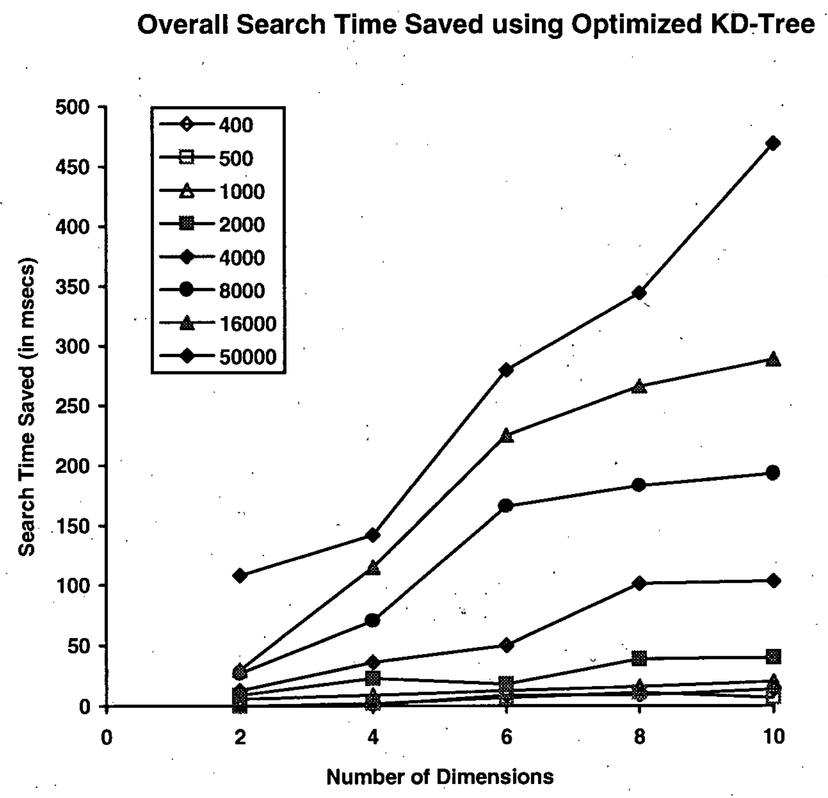


Figure 6.4 Overall Search Time Saved using the Optimized BA_KD-tree and fixed-radius search, for 8 different data set sizes and 5 different dimensions.

The x-axis represents the number of dimensions of the data points and the y-axis shows the average overall search time, in milliseconds, saved by the optimized bucket adaptive KD-tree over the 10 queries. Each line in the graph represents the savings for a particular data set size, from 400 data points at the bottom, to 50,000 data points at the top. As we saw in Chapter 5, the major cost to the overall search time comes from page accesses. Therefore, it comes as no surprise that the savings in overall search time, as dimensions and data set sizes increase, follows the same trend as the savings in number of page accesses. At low dimensions and small data set sizes, the savings in overall search time is small. But, as the dimensions and the data set sizes increase, the savings in search time increases. At 10 dimensions, for the largest data set (50,000), the optimized BA_KD-tree saves approximately 0.5 seconds over the original BA_KD-tree. Figure 6.5 below shows the ratio of the overall search time of the original BA_KD-tree over the overall search time of the optimized BA_KD-tree for all data set sizes and all dimensions. We find that there is again an asymptotic increase in the magnitude of this ratio as the data set sizes increase, and that this magnitude gets larger for greater numbers of dimensions.

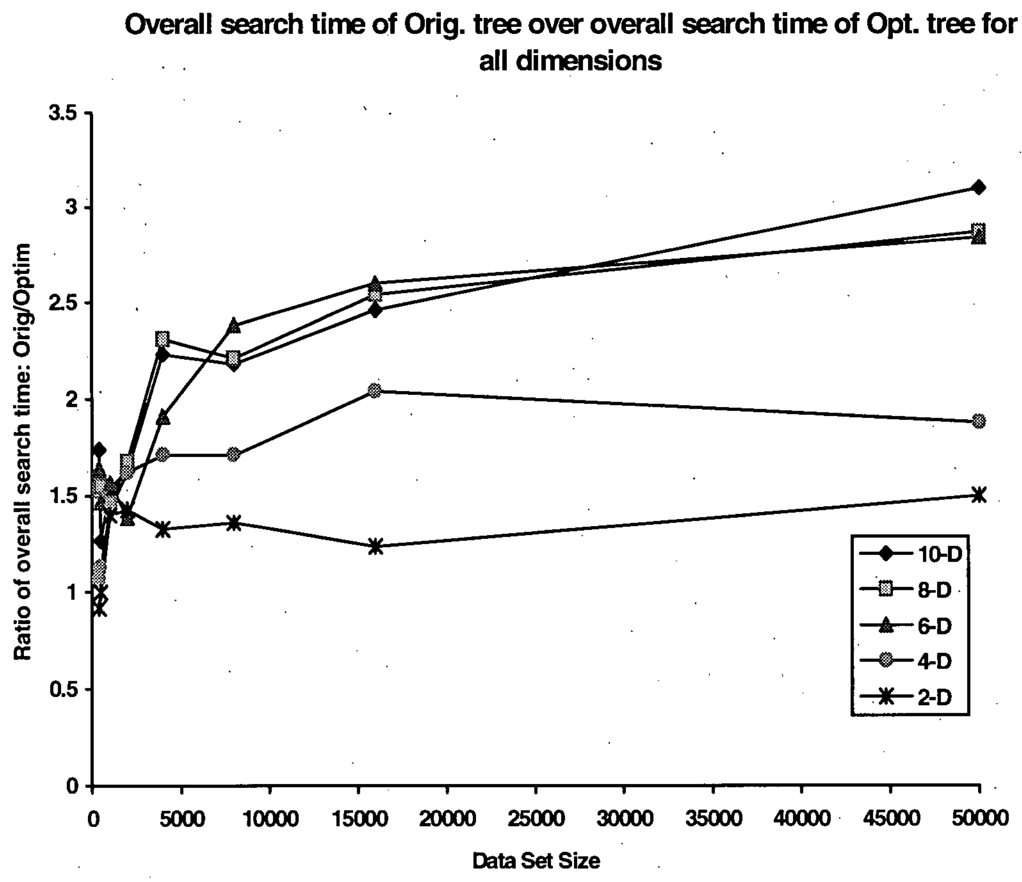


Figure 6.5 Ratio of overall search time of Original BA_KD-tree over overall search time of Optimized BA_KD-tree for 10-D data.

At the high end of the savings, we find that for a data set of 400 10-dimensional data points the optimized BA_KD-tree is 1.74 times faster than the original BA_KD-tree in performing a search, and for a data set of 50,000 10-dimensional data points it is 3.10 times faster than the original BA_KD-tree.

The *percentage* of savings in overall search time over the original BA_KD-tree are shown in Figure 6.6 for the larger data set sizes.

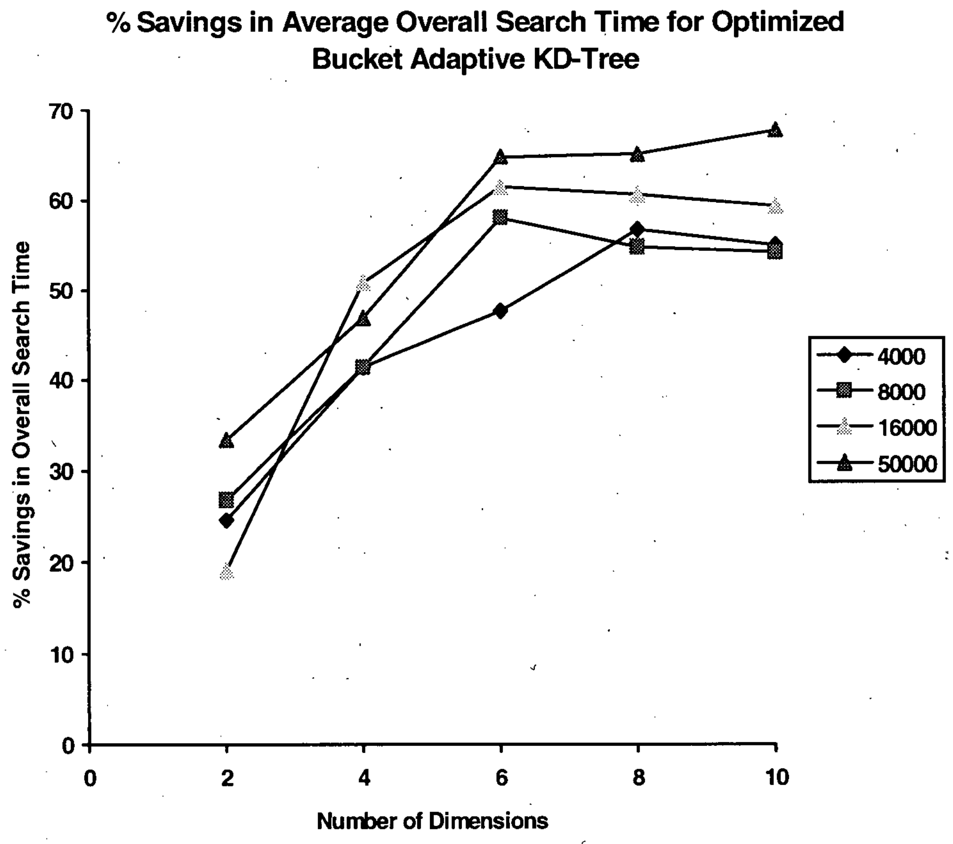


Figure 6.6 Percentage of savings in overall search time using Optimized BA_KD-tree, for large data sets.

The x-axis represents the number of dimensions of the data points and the y-axis shows the percentage of savings in the overall search time, in milliseconds, of the optimized BA_KD-tree over the original BA_KD-tree. Each line in the graph represents the savings for a particular data set size, from 4,000 data points at the bottom, to 50,000 data points at the top.

The savings in overall search time vary from 24.5% on the low end (i.e., for 4,000 data points and 2 dimensions) to 67.8% on the high end (i.e., for 50,000 data points and 10 dimensions). For all of these data set sizes, the savings rise dramatically from 2 to 6 dimensions then begin to level off at higher dimensions. Nevertheless, this leveling happens at over 50% in savings. Again, this large percentage of savings at the high end of the spectrum is very encouraging.

Our investigations into where most of the savings in page accesses occur, led us to discover that many of the data buckets that are accessed during the range search performed on the original BA_KD-tree are never retrieved during the fixed-radius search performed on the optimized BA_KD-tree. Moreover, the majority of these data buckets have zero or very few data points in them that even satisfy the range query, and all of these data points are, in fact, false hits. This means that a lot of I/O and CPU time is wasted by the range search as it accesses and examines data pages that only hold false hits. The fixed-radius near neighbor search on the optimized BA_KD-tree saves this time by eliminating the need to even access these pages.

Table 6.2 on the next page shows us the percentage of the data buckets retrieved by range search that are no longer retrieved by fixed-radius search. These are average values over the 10 queries.

Data Set Size	# of False hits found in Bucket	2-D (%Bkts)	4-D (%Bkts)	6-D (%Bkts)	8-D (%Bkts)	10-D (%Bkts)
400	0	0.0	0.0	2.17	4.87	6.72
	1	0.0	3.61	4.22	3.83	3.67
	2	1.25	0.0	.63	1.01	1.38
	3	1.67	1.25	1.00	1.67	0.56
	4-8	0.0	0.0	2.25	2.71	2.48
500	0	2.5	1.43	1.11	2.29	1.98
	1	0.0	0.0	0.0	2.54	2.28
	2	1.67	1.11	1.58	2.40	2.56
	3	1.25	.83	0.67	1.74	2.00
	4-9	0.0	1.67	3.43	2.90	4.38
1000	0	1.11	1.53	1.98	4.06	3.91
	1	0.83	2.08	2.15	4.51	4.41
	2	0.0	1.39	1.62	2.21	2.12
	3	0.0	1.39	3.00	3.28	3.56
	4-13	3.34	3.34	2.79	2.44	2.39
2000	0	0.0	5.00	5.86	8.00	8.51
	1	1.22	2.08	2.28	3.85	4.06
	2	1.11	1.18	2.19	3.18	3.16
	3	0.0	2.29	1.45	2.71	2.60
	4-14	6.28	4.81	4.48	2.83	2.86
4000	0	1.42	4.97	15.5	20.39	21.13
	1	0.78	4.42	9.58	11.16	11.46
	2	0.32	4.36	4.81	5.52	5.75
	3	0.0	2.00	5.55	3.69	3.76
	4-30	7.71	10.31	4.02	2.87	3.03
8000	0	0.37	2.14	6.86	10.35	10.96
	1	0.19	2.71	7.91	10.92	11.24
	2	0.19	2.64	8.02	8.44	7.89
	3	0.47	3.00	6.31	6.34	6.41
	4	0.29	2.68	3.75	3.62	3.67
	5-53	9.52	14.36	7.74	5.39	4.90
16000	0	0.14	10.56	19.89	25.59	26.75
	1	0.68	6.08	11.27	10.16	13.23
	2	0.41	4.28	6.17	6.65	7.76
	3	0.42	4.01	3.61	4.63	4.48
	4	0.14	2.57	2.69	2.67	2.76
	5-66	10.13	10.24	4.83	2.85	3.43

Table 6.2 Percent of Buckets saved from being accessed in the Optimized BA_KD-tree.

The table shows the distribution of the percentage of these buckets according to the number of data points (starting from 0) found in them that satisfy the range query. As the number of dimensions and data set sizes increase, the percentage of these "dud" data buckets increases. For example, for the set of 400 10-D data points, approximately 7% of these data buckets are ones in which 0 data points satisfy the range query, and for the set of 16,000 10-D data points 27% of these data buckets are ones in which 0 data points satisfy the range query. Within a data set, the percent of "dud" data buckets decreases with an increase in the number of false hits returned, with this trend becoming clearer for larger numbers of dimensions. For example, for 4,000 8 dimensional data points, over 20% of the "dud" data buckets have 0 false hits and only about 3% have 4 or more false hits. So, a greater percentage of these "dud" data buckets are ones from which very few false hits are returned. On the whole, the table shows that from 3% to 60% of the data buckets retrieved in range search return only false hits.

Table 6.3 presents highlights of the data in Table 6.2. It focuses on the percent of "dud" data buckets accessed using range search on the original BA_KD-tree from which only **0** to **3** false hits are retrieved. The table shows the variation in these percentages as the data set sizes increase (down the rows), and the number of dimensions increase (across the columns).

Data Set Size	**2-D (%Bkts)**	**4-D (%Bkts)**	**6-D (%Bkts)**	**8-D (%Bkts)**	**10-D (%Bkts)**
400	2.92	4.86	8.02	11.38	12.33
500	5.42	3.37	3.36	8.97	8.82
1000	1.94	6.39	8.75	14.06	14.00
2000	2.33	10.55	11.78	17.74	18.33
4000	2.52	15.75	35.44	40.76	42.10
8000	1.22	10.49	29.10	36.05	36.50
16000	1.65	24.93	40.94	47.03	52.22

Table 6.3 Percentage of Buckets saved that contain only 0 to 3 points that satisfy the Range Search.

As the dimensions increase over a single data set size, we see that the percentage of "dud" data buckets with only 0 to 3 false hits increases -- particularly in the larger data sets. For example, for 16,000 data points, the "dud" data buckets retrieved by range search increase from less than 2% at 2-D to over 52% at 10-D. As the data set sizes increase over a single dimension we also see that the percentage of "dud" data buckets increases. Again, this is clearer for larger dimensions.

The data presented in Tables 6.2 and 6.3 highlights the fact that the optimized BA_KD-tree with fixed-radius near neighbor search eliminates the need to access many unnecessary pages and hence precipitates a marked increase in overall savings.

Chapter 7
CONCLUSIONS

7.1 Summary

There are many research problems that need to be addressed in the design of useful and usable visual information management systems. In this thesis we examine three issues which we feel are essential: 1) the reduction of image vector dimensionality; 2) the choice of a multi-dimensional indexing structure that is suitable for organizing the reduced image space; and 3) the choice of a search algorithm that is effective and efficient in finding images that are within a fixed distance from a query.

We demonstrate that the *eigenfaces* approach to image analysis is a useful technique for reducing very-high-dimensioned image vectors while retaining most of the information in the image data set. In our implementation of this technique, the process of extracting principal components and projecting the image data points onto this new set of axes takes approximately one-and-a-half hours for 400 images. We are able to reduce image vectors with over 10,000 dimensions to vectors with around 20 dimensions, while retaining 70% of the variance of the images. We feel this indicates that high-dimensioned vectors of specific image features can also be effectively reduced using principal component analysis.

A thorough comparative analysis of many existing multi-dimensional indexing structures, plus a subsequent experimental evaluation of three of these structures, demonstrate that the bucket adaptive KD-tree is quite suitable for indexing PCA-transformed data points. This structure is particularly suited for PCA-transformed data because it partitions the data space based on the dimensions with the greatest spread. Since the first few PCs have the greatest variance they also typically have the greatest spread. This means that the structure partitions the data space and performs searches primarily using the first few dimensions. The internal nodes of the tree structure hold a small amount of information which does not grow with an increase in the number of dimensions. This and the previous factor provide for a fairly well-balanced tree that is not prohibitive in size. Our implementation of the bucket adaptive KD-tree requires on average 1.57 times less storage space than the R-tree structure. More significantly, it can perform range search up to 7.4 times faster than the R-tree.

Finally, we show that range search is a poor substitute for fixed-radius near neighbor search if one wants to find all images that are within a fixed distance from a query. The optimized bucket adaptive KD-tree with fixed-radius near neighbor search greatly improves the efficiency of the search and assists in reducing the number of false hits. The combination of the optimized structure and the fixed-radius search saves as much as 70% in the number of pages accessed during search and performs the search up to 3 times faster than the original BA_KD-tree structure using range search.

In summary, to efficiently index high-dimensioned image vectors that have been transformed by principal component analysis the optimized bucket adaptive KD-tree is shown to be the best suited multi-dimensional indexing structure, and it is seen to perform very well using fixed-radius near neighbor search to find images that are within a fixed distance from a query. We note, however, that in the experimental results one can see asymptotic behavior when the data points have greater than 6 dimensions. We feel that this is linked to the particular data testbed we used (i.e., gray-scale face images). Had other kinds of data, such as color images, been used they might have allowed fuller testing of this structure with truly high-dimensioned image vectors. This issue is discussed further in the next section.

7.2 Future Work

The results of this research are very promising. However, there are many aspects that should be examined further to ensure the usefulness of the technique we have implemented. In this section we consider two general areas in which further work can be pursued. The first involves issues that are directly related to what was done in this thesis work, and the second involves issues that would be interesting for future researchers to examine.

In the first category, i.e., issues directly related to this thesis, there are a number of things that can be done to refine our findings.

First, it would be preferable if we could acquire or implement an R*-tree structure (rather than an R-tree structure) and multipaging so as to do a true experimental evaluation of the three multi-dimensional indexing structures we originally chose in the comparative analysis. This would give us a better understanding of how the optimized BA_KD-tree compares to other structures that are claimed to efficiently handle high-dimensioned point data.

Second, in order to obtain a more realistic reflection of the storage costs and performance of both the original and the optimized BA_KD-tree structures it would be preferable for us to implement them such that the entire tree is not stored in main memory. The performance of the optimized BA_KD-tree would be directly affected since the size of the nodes in the tree would increase because the upper and lower range of the hyper-rectangle of the sub-trees of each child are stored at the nodes; this reduces the capacity of the index pages and, as the dimensions of the data increase, the index page capacity decreases (i.e., fewer nodes can be stored per index page). This will increase the storage space required by the optimized BA_KD-tree and will, in turn, effect its search performance. It would be interesting to see how much the optimized BA_KD-tree performance would deteriorate.

Third, it would be interesting to further investigate the effectiveness of the Early Fail and Early Success Tests. One could determine at what levels they are successful the most and use this information to limit the number of levels at which the ranges of the hyper-rectangles of sub-trees are stored. Furthermore, it would be interesting to determine what is the savings in search space that these tests precipitate, i.e., what is the reduction in the number of nodes visited in the optimized BA_KD-tree versus the original BA_KD-tree as a result of EFT and EST.

Fourth, in order to have more statistically reliable data it would be preferable to use a much larger number of queries, for example 1,000 or 2,000, rather than 10.

In the second category, i.e., issues for future research, there are also a number of items that would be interesting to pursue.

First, to further probe the issue of image analysis and feature reduction, it would be interesting to use an image testbed that is not a set of gray-scale face images. Collections of face images are fairly homogeneous in their content. This makes them ideal for use with the *eigenfaces* approach since most of the variance in the image set can be captured in a few dimensions. However, an image database that contains images of an art gallery or of animals may not be as convenient for the *eigenfaces* approach since there would be great variation in the contents of the images. Therefore, a varied image set should be used to see how well the *eigenfaces* approach can reduce the dimensions of image vectors from such a set.

Second, one of the elements that Bentley [Ben75a] brings up in his discussion on fixed-radius near neighbor search is that one could use different indexing structures for different parts of the search problem. In the two KD-tree structures we use in this thesis, the multi-dimensional points in the data buckets are sequentially examined to determine if they satisfy the query. Rather than use this brute force technique, it would be interesting to test the effect on performance of sorting the data points in a bucket by one of the dimensions. With PCA-transformed, the first dimension may be ideal for such a scheme since it carries most of the information in the data points.

Third, it would be interesting to compare the performance of fixed-radius near neighbor search to *n*-nearest neighbor search since the latter search technique has frequently been used for similarity matching, not only in image databases but also in several other types of applications such as pattern classification, estimating multivariate density, and minimizing head movement on direct access I/O devies [SW90].

BIBLIOGRAPHY

[A+95] A. D. Alexandrov, W. Y. Ma, A. El Abbadi, and B. S. Manjunath. "Adaptive Filtering and Indexing for Image Databases." *SPIE Proceedings, Storage and Retrieval for Image and Video Databases, III*, Vol. 2420, pp. 12-23, February 1995.

[Ben75a] Jon Louis Bentley. "A Survey of Techniques for Fixed Radius Near Neighbor Searching." *Stanford Linear Accelerator Center Technical Report*, No. 186, August 1975.

[Ben75b] J. L. Bentley. "Multidimensional Binary Search Trees Used for Associative Searching." *Communications of the ACM*, Vol. 8, No. 9: 509-517, September 1975.

[Ben79] J. L. Bentley. "Multidimensional Binary Search Trees in Database Applications." *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 4: 333-340, July 1979.

[BF79] J. L. Bentley and J. H. Friedman. "Data Structures for Range Searching." *Computing Surveys*, Vol. 11, No. 4: 397-409, December 1979.

[BGS92] Elisabetta Binaghi, Isabella Gagliardi, and Raimondo Schettini. "Indexing and Fuzzy Logic-Based Retrieval of Color Images." *Visual Database Systems, II, IFIP Transactions A-7*, pp. 79-92, 1992.

[BM79] J. L. Bentley and H. A. Maurer. "A Note on Euclidean Near Neighbor Searching in the Plane." *Information Processing Letters*, Vol. 8, No. 3: 133-136, March 1979.

[BM80] J. L. Bentley and H. A. Maurer. "Efficient Worst-Case Data Structures for Range Searching." *Acta Informatica*, Vol. 13, pp. 155-168, 1980.

[BPJ93] Jeffrey R. Bach, Santanu Paul and Ramesh Jain. "A Visual Information Management System for the Interactive Retrieval of Faces." *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3, No. 4: 619-628, August 1993.

[BS75] J. L. Bentley and D. F. Stanat. "Analysis of Range Searches in Quad Trees." *Information Processing Letters*, Vol. 3, No. 6: 170-173, July 1975.

[BSW77] J. L. Bentley, D. F. Stanat, and E. H. Williams, Jr. "The Complexity of Finding Fixed-Radius Near Neighbors." *Information Processing Letters*, Vol. 6, No. 8: 209-212, December 1977.

[B+90] N. Beckman, H. P. Kriegel, R. Schneider, and B. Seeger. "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles." *Proceedings of the ACM SIGMOD Conference*, pp. 322-331, 1990.

[Dun89] George H. Dunteman. *Principal Components Analysis.* SAGE Publications, Newbury Park, California, 1989.

[FB74] R. A. Finkel and J. L. Bentley. "Quad Trees: A Data Structure for Retrieval on Composite Keys." *Acta Informatica*, Vol. 4, pp. 1-9, 1974.

[FBF77] Jerome H. Friedman, J. L. Bentley and Raphael Ari Finkel. "An Algorithm for Finding Best Matches in Logarithmic Expected Time." *ACM Transactions on Mathematical Software*, Vol. 3, No. 3: 209-226, September 1977.

[Flu88] Bernhard Flury. *Common Principal Components and Related Multivariate Models*. John Wiley & Sons, New York, NY, 1988.

[F+94] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. "Efficient and Effective Querying by Image Content." *Journal of Intelligent Information Systems*, Vol. 3, No. 3/4: 231-262, 1994.

[GJ92] William I. Grosky and Zhaowei Jiang. "A hierarchical approach to feature indexing." *SPIE Proceedings, Image Storage and Retrieval Systems*, Vol. 1662, pp. 9-20, 1992.

[GM90] William I. Grosky and Rajiv Mehrotra. "Index-Based Object Recognition in Pictorial Data Management." *Computer Vision, Graphics, and Image Processing*, Vol. 52, pp. 416-436, 1990.

[GM92a] William I. Grosky and Rajiv Mehrotra. "Image Database Management." *Advances in Computers*, Vol. 35, pp. 237-291, 1992.

[GM92b] James E. Gary and Rajiv Mehrotra. "Shape Similarity-Based Retrieval in Image Database Systems." *SPIE Proceedings, Image Storage and Retrieval Systems*, Vol. 1662, pp. 2-8, 1992.

[Gre89] Diane Greene. "An Implementation and Performance Analysis of Spatial Data Access Methods." *IEEE Proceedings of the 5th International Conference on Data Engineering*, Los Angeles, pp. 606-615, 1989.

[GS92] T. Gevers and A. W. M. Smeulders. "Indexing of Images by Pictorial Information." *Visual Database Systems, II, IFIP Transactions A-7*, pp. 93-100, 1992.

[Gut84] A. Guttman. "R-Trees: A Dynamic Index Structure for Spatial Searching." *Proceedings of the ACM SIGMOD Conference*, pp. 47-57, June 1984.

[Güt94] Ralf Hartmut Güting. "An Introduction to Spatial Database Systems." Invited contribution to a special issue on Database Systems of the *VLDB Journal*, Vol. 3, No. 4, October 1994.

[HK92] Kyoji Hirata and Toshikazu Kato. "Query by Visual Example - Content based Image Retrieval." *Advances in Database Technology EDBT '92, Third International Conference on Extending Database Technology*, pp. 56-71, March 1992.

[H+92] T.-Y. Hou, A. Hsu, P. Liu, and M.-Y. Chiu. "A content-based indexing technique using relative geometry features." *SPIE Proceedings, Image Storage and Retrieval Systems*, Vol. 1662, pp. 59-68, 1992.

[H+93] K. Hirata, Y. Hara, N. Shibata, and F. Hirbayashi. "Media-based Navigation for Hypermedia Systems." *Hypertext '93 Proceedings*, pp. 159-173, November 1993.

[Jai93] Ramesh Jain. "NSF Workshop on Visual Information Management Systems." *SIGMOD RECORD*, Vol. 22, No. 3: 57-75, September 1993.

[Jol86] I. T. Jolliffe. *Principal Component Analysis*. Springer-Verlag, New York, NY, 1986.

[Kat92] Toshikazu Kato. "Database architecture for content-based image retrieval." *SPIE Proceedings, Image Storage and Retrieval Systems*, Vol. 1662, pp. 112-123, 1992.

[KS90] M. Kirby and L. Sirovich. "Application of the Karhunen-Loeve Procedure for the Characterization of Human Faces." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 12, No. 1: 103-108, January 1990.

[KWT74] T. Kunii, S. Weyl and J. M. Tennebaum. "A Relational Database Schema for Describing Complex Pictures with Color and Texture." *Proceedings of the Second International Joint Conference on Pattern Recognition*, Lyngby-Copenhagen, Denmark, pp. 310-316, August 1974.

[Mer78] T. H. Merrett. "Multidimensional paging for efficient database querying." *Proceedings of ICMOD 78, International Conference on Data Base Management Systems*, Milano, Italy, pp. 277-290, June 1978.

[Mer84] T. H. Merrett. *Relational Information Systems.* Reston Publishing Company, Inc., Reston, Virginia, 1984.

[MHN84] T. Matsuyama, L. V. Hao and M. Nagao. "A File Organization for Geographic Information Systems Based on Spatial Proximity." *Computer Vision, Graphics, and Image Processing*, Vol. 26, pp. 303-318, 1984.

[MO82] T. H. Merrett and E. J. Otoo. "Dynamic Multipaging: A Storage Structure For Large Shared Data Banks." *Improving Database Usability and Responsiveness*, P. Scheuermann *ed.*, Jerusalem, pp. 237-256, June 1982.

[N+93] W. Niblack, *et al.* "The QBIC Project: Querying Images By Content Using Color, Texture, and Shape." *Proceedings IS&T and SPIE, Electronic Imaging '93*, Vol. 1908, pp. 173-181, 1993.

[NHS84] J. Nievergelt, H. Hinterberger and K. C. Sevcik. "The Grid File: An Adaptable, Symmetric Multikey File Structure." *ACM Transactions on Database Systems*, Vol. 9, pp. 38-71, 1984.

[OvL82] Mark H. Overmars and Jan van Leeuwen. "Dynamic Multi-Dimensional Data Structures Based on Quad- and KD-trees." *Acta Informatica*, Vol. 17, pp. 267-285, 1982.

[PK93] R. W. Picard and T. Kabir. "Finding Similar Patterns in Large Image Databases." *M.I.T. Media Laboratory Perceptual Computing Section Technical Report*, No. 205, 1993.

[PKL94] Art Pope, Daniel Ko, and David Lowe. *Introduction to Vista Programming for Vista V2.1* (on-line programmer's manual), June 1994.

[PL94] Art Pope and David Lowe. "Vista: A Software Environment for Computer Vision Research." *Proceedings 1994 IEEE Computer Society Conference on Computer Vision & Pattern Recognition*, pp. 768-772, Seattle, WA, June 1994.

[PPS94] A. Pentland, R. W. Picard and S. Sclaroff. "Photobook: Tools for Content-Based Manipulation of Image Databases." *SPIE Proceedings, Storage and Retrieval for Image and Video Databases II*, Vol. 2185, pp. 34-47, 1994.

[Pre88] Rudolph W. Preisendorfer. *Principal component analysis in meteorology and oceanography*. Elsevier Science Pub. Co., New York, NY, 1988.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, New York, NY, 1990.

[SB90] Michael J. Swain and Dana H. Ballard. "Indexing Via Color Histograms." *IEEE Proceedings, 3rd International Conference on Computer Vision*, pp. 390-393, December, 1990.

[SB91] Michael J. Swain and Dana H. Ballard. "Color Indexing." *International Journal of Computer Vision*, Vol. 7, No. 1: 11-32, 1991.

[SHK93] Ramin Samadani, Cecilia Han and Lalitesh K. Katragadda. "Content-Based Event Selection From Satellite Images of the Aurora." *Proceedings IS&T and SPIE, Electronic Imaging '93*, Vol. 1908, pp. 50-55, 1993.

[SK87] L. Sirovich and M. Kirby. "Low-dimensional procedure for the characterization of human faces." *Journal of the Optical Society of America A*, Vol. 4, No. 3, pp. 519-524, March 1987.

[SP93] S. Sclaroff and A. Pentland. "A finite-element framework for correspondence and matching." *4th International Conference on Computer Vision*, pp. 308-313, May 11-14, 1993, Berlin, Germany. (Also available as: M.I.T. Media Laboratory Perceptual Computing Technical Note No. 201.)

[SRF87] T. Sellis, N. Roussopoulos and C. Faloutsos. "The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects." *Proceedings of the 13th VLDB Conference*, pp. 507-518, Brighton 1987.

[SSG92] P. L. Stanchev, A. W. M. Smeulders and F. C. A. Groen. "An Approach to Image Indexing of Documents." *Visual Database Systems, II, IFIP Transactions A-7*, pp. 63-77, 1992.

[SSU94] Hiroaki Sakamoto, Hideharu Suzuki and Akira Uemori. "Flexible Montage Retrieval for Image Data" *SPIE Proceedings, Storage and Retrieval for Image and Video Databases II*, Vol. 2185, pp. 25-33, 1994.

[Str88] Gilbert Strang. *Linear Algebra and its Applications*. 3rd edition. Harcourt, Brace, Jovanovich Publishers, San Diego, California, 1988.

[Str94] Markus A. Stricker. "Bounds for the discrimination power of color indexing techniques." *SPIE Proceedings, Storage and Retrieval for Image and Video Databases II*, Vol. 2185, pp. 15-24, 1994.

[SW90] Dennis Shasha and Tsong-Li Wang. "New Techniques for Best-Match Retrieval." *ACM Transactions on Information Systems*, Vol. 8, No. 2: 140-158, April 1990.

[Swa93] Michael J. Swain. "Interactive Indexing into Image Databases." *Proceedings IS&T/SPIE, International Symposium on Electronic Imaging: Storage and Retrieval for Image and Video Databases*, Vol. 1908, pp. 95-103, February 1993.

[TP89] Matthew Turk and Alex Pentland. "Face Processing: Models For Recognition." *SPIE Proceedings, Intelligent Robots and Computer Vision VIII: Algorithms and Techniques*, Vol. 1192, pp. 22-32, 1989.

[TP91] Matthew Turk and Alex Pentland. "Eigenfaces for Recognition." *Journal of Cognitive Neuroscience*, Vol. 3, No. 1: 71-86, 1991.

[Yuv75] Gideon Yuval. "Finding Near Neighbours in *K*-Dimensional Space." *Information Processing Letters*, Vol. 3, No. 4: 113-114, March 1975.