Spring 5-31-2005

High-dimensional indexing methods utilizing clustering and dimensionality reduction

Lijuan Zhang
New Jersey Institute of Technology

ABSTRACT

HIGH-DIMENSIONAL INDEXING METHODS UTILIZING CLUSTERING AND DIMENSIONALITY REDUCTION

by
Lijuan Zhang

The emergence of novel database applications has resulted in the prevalence of a new paradigm for similarity search. These applications include multimedia databases, medical imaging databases, time series databases, DNA and protein sequence databases, and many others. Features of data objects are extracted and transformed into high-dimensional data points. Searching for objects becomes a search on points in the high-dimensional feature space. The dissimilarity between two objects is determined by the distance between two feature vectors. Similarity search is usually implemented as nearest neighbor search in feature vector spaces. The cost of processing k-nearest neighbor (k-NN) queries via a sequential scan increases as the number of objects and the number of features increase. A variety of multi-dimensional index structures have been proposed to improve the efficiency of k-NN query processing, which work well in low-dimensional space but lose their efficiency in high-dimensional space due to the curse of dimensionality. This inefficiency is dealt in this study by *Clustering and Singular Value Decomposition - CSVD* with indexing, *Persistent Main Memory - PMM* index, and *Stepwise Dimensionality Increasing - SDI-tree* index.

CSVD is an approximate nearest neighbor search method. The performance of CSVD with indexing is studied and the approximation to the distance in original space is investigated. For a given *Normalized Mean Square Error - NMSE*, the higher the degree of clustering, the higher the recall. However, more clusters require more disk page accesses. Certain number of clusters can be obtained to achieve a higher recall while maintaining a relatively lower query processing cost.

Clustering and Indexing using Persistent Main Memory - *CIPMM* framework is motivated by the following consideration: (a) a significant fraction of index pages are accessed randomly, incurring a high positioning time for each access; (b) disk transfer rate is improving 40% annually, while the improvement in positioning time is only 8%; (c) query processing incurs less CPU time for main memory resident than disk resident indices. CIPMM aims at reducing the elapsed time for query processing by utilizing sequential, rather than random disk accesses. A specific instance of the CIPMM framework *CIPOP*, indexing using Persistent Ordered Partition - OP-tree, is elaborated and compared with clustering and indexing using the SR-tree, CISR. The results show that CIPOP outperforms CISR, and the higher the dimensionality, the higher the performance gains.

The SDI-tree index is motivated by fanouts decrease with dimensionality increasing and shorter vectors reduce cache misses. The index is built by using feature vectors transformed via principal component analysis, resulting in a structure with fewer dimensions at higher levels and increasing the number of dimensions from one level to the other. Dimensions are retained in nonincreasing order of their variance according to a parameter p, which specifies the incremental fraction of variance at each level of the index. Experiments on three datasets have shown that SDI-trees with carefully tuned parameters access fewer disk accesses than SR-trees and VAMSR-trees and incur less CPU time than VA-Files in addition.

HIGH-DIMENSIONAL INDEXING METHODS UTILIZING CLUSTERING AND DIMENSIONALITY REDUCTION

by
Lijuan Zhang

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

May 2005

APPROVAL PAGE

HIGH-DIMENSIONAL INDEXING METHODS UTILIZING CLUSTERING AND DIMENSIONALITY REDUCTION

Lijuan Zhang

[illegible]

Dr. Alexander Thomasian, Dissertation Advisor — Date
Professor of Computer Science, NJIT

[illegible]

Dr. Narain Gehani, Committee Member — Date
Professor and Chairman of Computer Science, NJIT

[illegible]

Dr. Vincent Oria, Committee Member — Date
Assistant Professor of Computer Science, NJIT

[illegible]

Dr. Dimitrios Theodoratos, Committee Member — Date
Associate Professor of Computer Science, NJIT

[illegible]

Dr. Jian Yang, Committee Member — Date
Assistant Professor of Industrial and Manufacturing Engineering, NJIT

BIOGRAPHICAL SKETCH

Author:	Lijuan Zhang
Degree:	Doctor of Philosophy
Date:	May 2005

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2005
- Master of Science in Computer Science,
  Northeastern University, Shenyang, Liaoning, P.R. China, 1999
- Bachelor of Science in Computer Science,
  Northeastern University, Shenyang, Liaoning, P.R. China, 1996

Major: Computer Science

Presentations and Publications:

L. Zhang and A. Thomasian, "The stepwise dimensionality increasing - SDI index for high-dimensional data," *The Computer Journal*, submitted, 2005.

L. Zhang and A. Thomasian, "The persistent clustered main memory index for accelerating k-NN queries on high dimensional datasets," *Second International Workshop on Computer Vision meets Databases (CVDB)* , Baltimore, MD, June, 2005.

A. Thomasian, Y. Li and L. Zhang, "Exact k-NN queries on clustered SVD datasets," *Information Processing Letters*, to appear, 2005.

Y. Li, A. Thomasian and L. Zhang, "Optimal subspace dimensionality for k-NN search on clustered datasets," *Proc. 15th Int'l Conf. on Database and Expert Systems Applications (DEXA)*, 201-211, Zaragoza, Spain, Aug./Sep. 2004.

To my beloved husband Dongpeng,
for his endless love, care, and support.

ACKNOWLEDGMENT

First of all, I would like to express my deepest appreciation to my advisor, Dr. Alexander Thomasian, for introducing me to and guiding me through the wonderful world of high-dimensional indexing. Furthermore, I would like to thank my Committee members, Dr. Narain Gehani, Dr. Vincent Oria, Dr. Dimitrios Theodoratos, and Dr. Jian Yang, for their insightful comments on my proposal. A special thank goes to Dr. Artur Czumaj for his understanding, encouragement and support. I would like to thank my colleague, Yue Li, for her generous discussion on many problems during my research. Many thanks to my labmates Gang Fu, Chang Liu, Junilda Spirollari, Jun Xu, among others, for the good research environment they provided.

Last, but not least, I would like to thank my husband, Dongpeng, for his love, steadfast support, encouragement and sharing my happiness and sadness. Without him, I would not have the opportunity to write this acknowledgement. And finally, I thank my parents, sister, brother, parents-in-law, and sister-in-law for giving me such a big happy family.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF TABLES
(Continued)

LIST OF FIGURES

LIST OF FIGURES
(Continued)

LIST OF FIGURES
(Continued)

LIST OF FIGURES (Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

A growing number of new applications require novel database management systems to support new types of data and associated queries. Some examples are multimedia databases, medical imaging databases, DNA and protein sequence databases, time series databases, and databases for molecular biology. In these applications exact match queries no longer play a major role, instead, searching for similar patterns is essential, since it helps in prediction, decision making, and medical diagnosis.

Similarity queries can be classified into two categories: *whole match* and *sub-pattern match* [2]. The query only specifies part of the object in *sub-pattern match*, while the query and objects in the database are the same length for *whole match*. To further classify, the *nearest neighbor query*, which belongs to *whole match* category, is the focus of this study. *GEneric Multimedia object INdexIng* - GEMINI is a generic approach to indexing multimedia objects for fast similarity searching [2].

The steps for GEMINI are: determine the distance function between two objects, find one or more numerical feature-extraction functions, prove that the distance in feature space *lower-bounds* the distance in object space, and store and retrieve the feature vectors using a multi-dimensional indexing method. The distance function is usually provided by domain experts. Features of data objects are extracted and represented as multi-dimensional points. Searching for an object becomes a search on points in the multi-dimensional feature space. The dissimilarity between two objects is the distance between two feature vectors. Similarity search is transformed into nearest neighbor search in multi-dimensional feature vector space.

With the rapid deployment of different types of applications, not only is the volume of data expanding everyday, but also the dimensionality is growing higher and higher.

The efficiency of multi-dimensional index structures for nearest neighbor search deteriorates rapidly as the number of dimensions increases due to the "curse of dimensionality" [3]. Developing new approaches for searching and indexing high-dimensional data is a challenging area and has attracted the attention of many researchers.

1.1 High-Dimensional Applications

The need to efficiently access large scale multi-dimensional data drives the design of the new generation of database systems. Specific applications include the following:

- Multimedia databases, where images, audios, and videos are stored. Similarity queries would retrieve similar images, music scores, or video clips. Features of images can be color, texture and shape [4]. Color histograms and texture features based on Gabor filters are usually used. A similarity query can be *"Find k images which are most similar to the query image in terms of colors"*.

- Medical databases, where gray scale medical images like 2-D images (e.g. X-rays) and 3-D images (e.g. MRI brain scans) are stored. Quickly retrieving past cases with similar symptoms would help to diagnose new cases, as well as medical education and research. Typical queries would be *"find a patient who has a similar MRI brain scan with the current patient"*.

- Time series databases, which stores financial, marketing and production time series. Queries like *"find companies with the similar stock-price movement to this company last year"* can aid forecasting the stock-price movement of the company. Euclidean distance is usually used as the distance function between two sequences. Coefficients of Discrete Fourier Transform (DFT) can be used as the features [2].

- DNA and protein databases, which contain large collections of strings composed of letters representing nucleotides or amino acids. In the newly emerging field of bioinformatics, genome databases are being used for drug design, medical care, phylogenetic analysis, evolutionary analysis, personalized medicine, and many other applications. Searching for similar sequences can determine whether a gene responsible for some disease also appears in other species. Sequences are very long, and searching is very expensive. The bovine pancreatic trypsin inhibitor gene at EMBL (European Bioinformatics Institute, UK) data library has 3998 nucleotides [5]. The distance function is the editing distance, which is the smallest number of insertions, deletions, or substitutions required to transform one sequence to another.

The architecture of a content-based retrieval system for high-dimensional applications is illustrated in Figure 1.1. The features of images[1] or time series[2] are extracted and transformed into high-dimensional points (feature vectors) first. Then a multi-dimensional index is built based on the feature vectors. The features of a query image or time series is also extracted and transformed. Similarity search is transformed into a search of points which are close to a query point in high-dimensional feature space. The actual search is performed mainly on the index structure. Search results are returned to the user by extracting the original data based on the matched feature vectors. The performance of the whole system highly depends on the index structure.

Many tools for content-based retrieval system have been developed. Prominent examples for photographic images include IBM's QBIC (Query by Image Content) [6], MIT's Photobook system [7], VisualSeek from the Columbia University, and the multimedia Datablade from Informix /Mirage. QBIC is an early prototyping and later commercial system. Photobook describes the image content using colors and textures. Blobworld [8] is an image retrieval system using regions. It automatically segments each image into regions which roughly correspond to objects or parts of objects. Users can query the database based on the object they selected. For image and video retrieval, Virage[3] is worthy of mention.

[1]Images in Figure 1.1 are obtained from `http://amazon.ece.utexas.edu/~qasim/samples/sample_landscapes4.html`

[2]The time series sequences in Figure 1.1 are obtained from `http://www-personal.buseco.monash.edu.au/~hyndman/TSDL/`

[3]http://www.virage.com

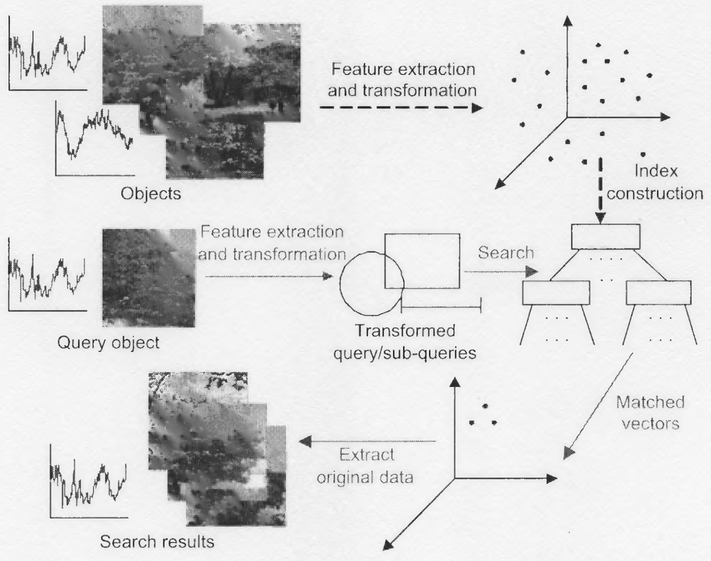

Objects
Feature extraction and transformation
Index construction
Query object
Feature extraction and transformation
Search
Transformed query/sub-queries
Matched vectors
Extract original data
Search results

Figure 1.1 Architecture of a retrieval system for high-dimensional applications.

These techniques can also be applied to medical images, artwork, and video clips. Queries based on spatial relationships of the salient objects in the images are not as well studied as feature-based queries. Spatial relationships, such as relative positioning, adjacency, overlap, and containment, enable users to ask queries of the type "show all the images where a car is to the left of a building". Systems that couple spatial and feature-based querying enable sophisticated queries to be posed such as "show all the images where a red car is in front of a building".

Visual features are classified in [9] into three levels of abstraction: *primitive* features such as color, texture and shape, *logical* features such as the identity of objects shown, and *abstract* attributes such as the significance of the scenes depicted. Color has been the most popular feature in photography used by artists. All currently available systems only use *primitive* features unless manual annotation is coupled with the visual features. Texture of images can be captured by wavelets or Gabor filters. Segments of images can be described by shape features.

This dissertation concentrates on image databases. Content-based image retrieval (CBIR) has been an active research area over the last 20 years [10]. Images are produced in an ever-increasing quantities. The need to query the visual or audio content in multimedia repositories is immediate, especially with the expanding Internet.

1.2 Motivations and Contributions

A variety of multi-dimensional indexing methods have been proposed [11]. With new applications having higher dimensionality requirements emerging, and due to the curse of dimensionality, most traditional index structures have lost their effectiveness. This dissertation addresses three aspects of indexing high-dimensional data to speed up k-nearest neighbor search.

1.2.1 Approximate Nearest Neighbor Search

The nearest neighbors problem is of major importance to high-dimensional applications. Approximate nearest neighbor search has gained increasing interest. Since the selection of features and distance metrics are rather heuristic and merely an attempt to make mathematically precise, it seems like an overkill to insist on exact nearest neighbors [12]. Resorting to an ϵ-approximate nearest neighbor for a small ϵ should suffice for most applications.

The number of features of the objects of interest ranges anywhere from tens to thousands. Dimensionality reduction techniques, such as Latent Semantic Indexing (LSI) [2], Principal Component Analysis (PCA) [13], Singular Value Decomposition (SVD) [2], and Karhunen-Loève Transform (KLT) [2], are promising methods to solve the curse of dimensionality problem and yield a dimensionality with minimum loss of distance information. There are two categories of dimensionality reduction methods: Global Dimensionality Reduction (GDR) and Local Dimensionality Reduction (LDR) [14, 3]. GDR works well for globally correlated datasets. However, datasets are usually locally correlated, which means reducing the data dimensionality using GDR causes significant loss of distance information. In this case, LDR performs dimensionality reduction on locally correlated clusters of the dataset.

Clustering and Singular Value Decomposition (CSVD) [3] is an approximate similarity search method, which clusters the dataset first before applying dimensionality reduction. A multi-dimensional index is built for each cluster in the dimensionality reduced subspace. The challenge here is to achieve dimensionality reduction with a limited loss of distance information and in particular with little effect on information retrieval performance.

In this dissertation, three multi-dimensional indices are compared, the best one is selected as the within cluster index. Then two approximate distances in the dimensionality reduced subspace are presented and one is proved to be closer to the distance in the original space. Experiments evaluate the performance of the CSVD method, which includes the precision, recall, number of pages visited, CPU time, and effects of the degree of clustering.

1.2.2 Persistent Main Memory Index

Multi-dimensional indices can be classified as disk resident indices and memory resident indices. The former aims at minimizing the number of disk pages accessed, while the latter focuses on reducing the CPU time for query processing. Disk pages can be accessed more efficiently sequentially, rather than randomly. Sequential access time is determined by the disk transfer rate, which has been increasing by 40% per year. Random access time to small index pages is mainly positioning time, which has been decreasing at a rate of less than 10% per year. This trend makes sequential disk accesses increasingly desirable compared to random accesses [15]. A method optimizing the processing time rather than the number of page accesses is developed in this dissertation.

A general framework, Clustering and Indexing with Persistent Main Memory Index - CIPMM, is proposed. In CIPMM, main memory indices are serialized into contiguous memory areas, so that indices can be saved on disk and are loadable via sequential disk accesses, for each of which positioning time is incurred only once. CIPMM utilizes the dual filtering of clustering and indexing, and the increasing disk transfer rate. A specific instance, CIPOP, partitions the dataset into small clusters first and then build a main memory *Ordered Partion - OP-tree* index [16] for each cluster. The index is then serialized and written on disk so that it can be restored as fast as possible on demand. Two serialization methods for the OP-tree are proposed. The two-phase dynamic memory allocation method is static, while the one-phase method allows a semi-dynamic allocation for dealing with the insertions of new points. Experiments show that the CIPOP outperforms Clustering and Indexing using SR-tree [17] - CISR and Clustering and Indexing using VA-File [18]-CIVAFile.

1.2.3 Stepwise Dimensionality Increasing Index

Multi-dimensional index structures can be used to improve the efficiency of k-NN query processing, but lose their effectiveness as the dimensionality increases. The curse of dimensionality manifests itself in the form of increased overlap among the nodes of the index, so that a high fraction of index pages are touched in processing k-NN queries. The increased dimensionality results in a reduced fanout and an increased index height. Fanout can be varied within an index structure. Using fewer dimensions at upper levels and more dimensions at lower levels, more branches can be checked at upper levels.

In this dissertation, a Stepwise Dimensionality Increasing - SDI-tree index is proposed, which aims at reducing the number of disk accesses and CPU processing cost. It combines dimensionality reduction with hierarchical structure with larger fanouts at top levels and smaller fanouts at lower levels. The index is built using feature vectors transformed via principal component analysis. Dimensions are retained in nonincreasing order of their variance according to a parameter p, which specifies the incremental fraction of variance at each level of the index. The optimal value for p is determined experimentally. Experiments on three datasets have shown that SDI-trees access fewer disk pages and incur less CPU time than SR-trees [17], VAMSR-trees, and Vector Approximation - VA-Files [18].

1.3 Outline

The outline of this dissertation is as follows. In Chapter 2, frequently used techniques in high-dimensional indexing are presented. In Chapter 3, several index structures are compared, the best one is selected for indexing each cluster of CSVD and the performance of CSVD with indexing is evaluated. In Chapter 4, a general framework CIPMM is proposed and a specific instance - CIPOP is elaborated. Two serialized method for the main memory index, OP-tree, are described and studied. In Chapter 5, the SDI-tree is proposed and compared with SR-trees and VAMSR-trees. Conclusions and future work are given in Chapter 6. In addition, some useful information and results are described in Appendixes. The query types in high-dimensional applications are defined in Appendix A. The characteristics of high-dimensional space are described in Appendix B. The OP-tree and the OMNI-family are compared in Appendix C. The performance of local dimensionality methods is reported in Appendix D. Finally, the routines for VAMSplit R-tree creation are given in Appendix E.

CHAPTER 2

INDEXING TECHNIQUES

In high-dimensional applications, a big challenge is to find the k nearest neighbors of a query point efficiently. Data is usually stored on secondary storage. Disk access in response to a query results in accesses to a large number of randomly placed data blocks, which is quite slow. Sequentially scanning the whole dataset is too expensive. Reducing the search space is crucial for efficient searches. Many indexing techniques have been developed, such as clustering, indexing, clustering plus indexing, and approximate methods for applications that can tolerate some error. By partitioning a large dataset into clusters, only a subset of the clusters closest to the query point need to be visited. By using indexing, the search space is expected to be reduced greatly. However, traditional index methods like B+-trees and hashing [19] are not suitable for multi-dimensional data as they can handle only one-dimensional data. Indices on lower dimensional data have been studied extensively [20, 21]. In fact, most multi-dimensional indices work well in low to medium dimensional spaces, but do not scale with dimensionality. Clustering the dataset before building the index has the advantage of introducing dual filters, which is used in this study. The cost of k-NN queries can be lowered by reducing the number of dimensions. Several approximate methods for k-NN queries have been proposed. However, their applications are limited.

This chapter is organized as follows. In Section 2.1, several similarity measures for the feature vectors of application objects are introduced. In Section 2.2, clustering methods are surveyed and compared. In Section 2.3, dimensionality reduction techniques are discussed. In Section 2.4, high-dimensional index structures are surveyed and classified, and related cost models are described. In Section 2.5, different nearest neighbor search algorithms are described.

2.1 Similarity Measures

The similarity measures are closely related to specific applications and domain experts are usually needed to provide the appropriate distance (dissimilarity) function.

Euclidean Distance The Euclidean distance is the distance of choice in time series, financial and forecasting applications [2]. One of its valuable properties is that it is preserved under orthonormal transforms [22]. The Euclidean distance is solely considered in this study.

Definition 2.1 *Given two N-dimensional vectors $\vec{x}$ and $\vec{y}$, the Euclidean distance between the two is*

$$E(\vec{x},\vec{y}) = \sqrt{\sum_{i=1}^{N} (x_i - y_i)^2}.$$

Minkowski Metrics Minkowski metrics are a family of distance functions which are generalizations of Euclidean distance formula. The formal definition is given as follows.

Definition 2.2 *Given two N-dimensional vectors $\vec{x}$ and $\vec{y}$, the Minkowski distance between them is*

$$L_p(\vec{x},\vec{y}) = (\sum_{i=1}^{N} |x_i - y_j|^p)^{1/p} .$$

When $p = 2$, it becomes Euclidean distance. For $p = 1$, it yields the Manhattan distance or city block distance which is useful as a measure of the distance between two points if walking on a grid of city streets (no real diagonals). L_p metrics assume all the dimensions are independent and of equal importance. For dimensions that are interdependent and vary in importance, the following Mahalanobis distance is introduced.

Mahalanobis distance Mahalanobis distance is the distance between two N-dimensional points scaled by the statistical variation in each dimension of the point.

Definition 2.3 *Given two N-dimensional vectors $\vec{x}$ and $\vec{y}$, the Mahalanobis distance between them is*

$$M(\vec{x},\vec{y}) = (\vec{x}-\vec{y})^T C^{-1}(\vec{x}-\vec{y}),$$

where C is the covariance matrix of the distribution where the points come from. If the dimensions are independent, C becomes the identity matrix and the distance degrades to the Euclidean distance.

Based on the above property, elliptical clusters can be found if the Mahalanobis distance is used [23]. As in Figure 2.1, the point Q is closer to the centeroid of cluster $C2$ based on Euclidean distance, however, it is closer to the centroid of cluster $C1$ if Mahalanobis distance is utilized.

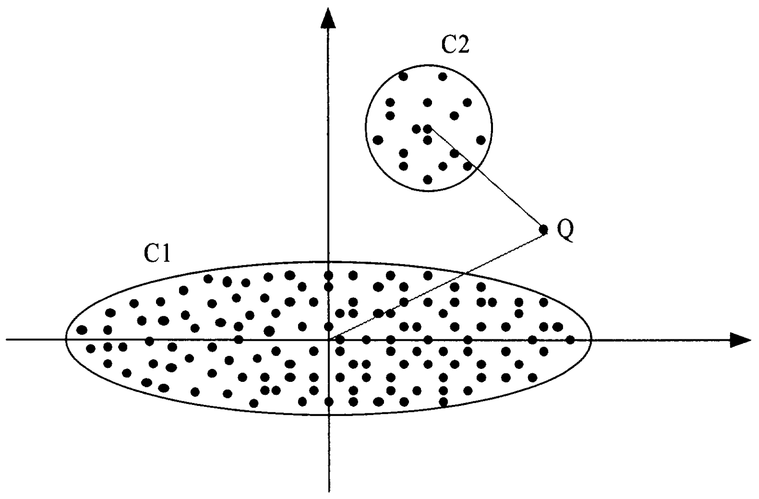


Figure 2.1 Mahalanobis distance.

2.2 Clustering

Clustering partitions a set of data into groups such that data within a group is similar to each other and data that belongs to different groups is dissimilar. Clustering is used to speed up the search for finding k nearest neighbors by reducing the number of distance computations in [24]. Clustering the dataset before building any index for each cluster is therefore desirable. A large number of clustering algorithms have been developed to address the varying requirements of different applications. Some can only discover specified number of spherical-shaped groups (e.g. k-means [25]), some attempt to discover natural-shaped groups (e.g. CURE [26]), while others can automatically determine the number of clusters (e.g. DBSCAN [27], CLARANS [28]).

K-means K-means [25] is one of the simplest unsupervised learning algorithms that solve the well known clustering problem. It tries to find a specified number of clusters (k) represented by their centroids. The algorithm first chooses k initial centroids, which can be picked using the bootstrap method in [29]. Each point is then assigned to its closest centroid. The centroid of each cluster is then recalculated based on the points currently in that cluster. The assignment is repeated until no point changes its cluster membership.

Clustering works well when clusters are compact and well separated. The shape of clusters generated by k-means is spherical. The main task is to minimize the sum of squared error (SSE) which is given below. When cluster sizes are highly variable, k-means splits large clusters to minimize the SSE.

$$SSE = \sum_{h=1}^{H} \sum_{i \in C_h} \|\vec{x}_i - \vec{\mu}_h\|^2 \tag{2.1}$$

where H: number of clusters, C_h: cluster h, μ_h: mean of cluster h, and $||.||$: Euclidean norm of a vector.

CURE CURE (Clustering Using Representatives) [26] is an agglomerative hierarchical clustering algorithm which identifies clusters having non-spherical shapes and unequal sizes. Each cluster has multiple well scattered representative points, which help CURE to capture well the geometry of non-spherical shapes. The representatives are formed as follows. The first one is the point farthest from the centroid of the cluster, while others are chosen farthest from all the previously chosen points. Then, they are shrunk toward the centroid by a factor α to moderate the effect of outliers.

The closest clusters are merged step by step until specified number of clusters k is achieved. The distance between two clusters is the minimum distance between any two representative points with each from separate clusters, which can be formally described as follows for given cluster u and v.

$$d(u, v) = min_{p \in u.rep, q \in v.rep} d(p, q)$$

A heap [30] is used to keep track of all the clusters arranged by the increasing order of distances to their closest cluster, while a k-d tree [31] stores all the representatives and is used to find the closest cluster. Since the worst case time complexity of the CURE algorithm is $O(M^2 logM)$ for M points, it can not be applied directly to large datasets, in which case, sampling is used before the whole dataset is partitioned.

DBSCAN DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [27] is a density-based clustering algorithm, which creates clusters with minimum size and density. Density for a particular point is defined as the number of points within a specified radius around the point. The desired number of clusters, k, is not an input parameter, but rather it is determined by the algorithm itself. The algorithm is based on the concept of *core point*, *border point*, and *noise point*. A point is a *core point* if the number of points within a user specified parameter, *Eps*, from the point exceeds a certain threshold, $MinPts$. A *border point* falls within the neighborhood of a core point. A *noise point* is any point that is neither a core point, nor a border point. Any two core points within a distance of *Eps* are put in the same cluster. Any border point that is close enough to a core point belongs to same cluster as the core point. All the noise points are discarded. Therefore not all the points are assigned to clusters. The worst case time complexity of DBSCAN is $O(M^2)$.

With the adoption of efficient index structures that retrieve all points within a given distance of a specified point, the complexity can be as low as $O(MlogM)$.

2.3 Dimensionality Reduction

As existing multi-dimensional indexing methods do not scale well to higher dimensions, reducing the dimensionality is an obvious and important possibility for diminishing the dimensionality problem and should be performed whenever possible [32].

Several signal processing and statistical techniques can be used to reduce the dimensionality. When data is known in advance, Principal Component Analysis - PCA [13], Singular Value Decomposition - SVD [33, 2, 34], and Karhunen-Loève Transform - KLT [2] are related methods which are optimal in dimensionality reduction. While for dynamic data, Discrete Fourier Transform - DFT, Discrete Cosine Transform - DCT [35] and Discrete Wavelet Transform - DWT [2, 34] are well known methods. They can perform as well as the static methods if the data follows specific statistical models [36]. The coefficients of DFT are used as the features for time series databases. DCT performs well when the dimensions are highly correlated.

Another set of techniques are space-filling curves. By following the visiting order of the curve, data in multi-dimensional space can be mapped into one-dimensional space, where efficient indexing methods are available.

DCT and DWT The JPEG standard divides images into pixel blocks, which are compressed using quantized DCT coefficients. Although good compression is achieved, blocking effects also appear. The JPEG2000 [37] adopts DWT using the Daubechies(9,7) biorthogonal wavelet to compress images, which achieves much higher compression and/or much lower image degradation. The basic idea of wavelets is hierarchical decomposition of a function into a set of basis and wavelet functions. By using DWT, the following flexibilities are gained: multi-resolution, fast image preview, and progressive image downloading, which is important for internet applications. The compression performance improves at the same compression ratio as the number of levels increase.

SVD and PCA The eigenvalues and eigenvectors are defined for square matrices. A closely related concept for rectangular matrix is the Singular Value Decomposition - SVD. The formal definition is given as follows:

Definition 2.4 (SVD) *Given an $M \times N$ matrix X, it can be expressed as*

$$X = USV^T,$$

where U is a column-orthonormal $M \times R$ matrix, $R \leq N$ is the rank of the matrix X, S is a diagonal $R \times R$ matrix, which contains singular values of X, and V is a column-orthonormal $N \times R$ matrix.

PCA decomposes the covariance matrix C of X as

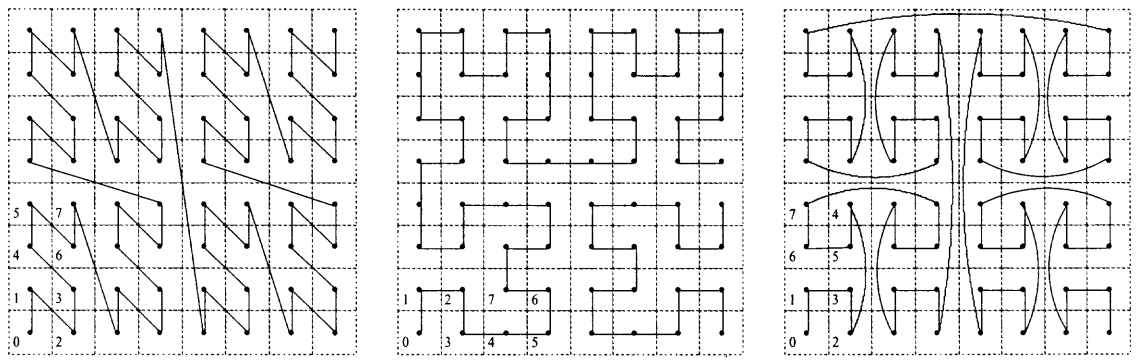
$$C = \frac{1}{M} X^T X = V \Lambda V^T,$$

where V contains eigenvalues of X, and V is the same as in SVD.

The relationship between singular values and eigenvalues is $\lambda_i = s_i^2/M, 1 \leq i \leq N$, which can be shown as follows:

$$C = \frac{1}{M} X^T X = \frac{1}{M} (USV^T)^T (USV^T) = \frac{1}{M} VS^T U^T USV^T = \frac{1}{M} VS^2 V^T = V \Lambda V^T.$$

Space-filling Curves Space-filling curves [11] provide the means to find a total order that preserves spatial proximity to some extent. This technique can be used to transform a multi-dimensional indexing problem to a classical single-attribute indexing problem. Several space-filling curves have been proposed: z-ordering [38], the Hilbert curve [39] and the Gray code [40, 41]. Starting from the lower-left, the order in which cells are visited by the curves defines a total ordering on the cells. This is shown in Figure 2.2. Experiments in [39, 42] show that the Hilbert curve is most promising.



(a)z-ordering (b)Hilbert curve (c)Gray code

Figure 2.2 Space-filling curves.

2.4 High-Dimensional Index Structures

In this section, a survey on most active index structures is presented first. The index structures can be classified into indexing on order spaces, indexing on feature vector spaces, and indexing on metric spaces, each of them are further addressed. Finally cost models for nearest neighbor query processing are described.

2.4.1 A Brief Survey of Multi-Dimensional Indices

Multi-dimensional indexing has been an active research area in recent years [11]. One categorization is disk-resident versus memory-resident indices. For the disk-resident indices, the number of disk pages accessed and the elapsed time are the major performance measures. Since the elapsed time is hard to measure accurately, the number of disk pages accessed usually determines the performance of an index. For the main-memory resident indexing, the CPU time is the key performance measure. Unfortunately main memory indices can not be scaled to larger datasets, since the index may not fit in main memory.

Using multiple B+-trees (one per dimension) or mapping multi-dimensional keys to one dimensional key (using a space filling curve like the z-ordering [11]) followed by the building of a B+-tree index are inefficient in higher dimensions. In order to achieve high performance in query processing, multi-dimensional index structures are designed to index data based on multiple dimensions simultaneously. Unfortunately, multi-dimensional index structures deteriorate in performance when the dimension of the data space increases due to a number of effects in high-dimensional space. For example, the first multi-dimensional index structures (R-trees, K-D-B-trees and grid files) work well at low dimensional spaces, they are not suitable for high-dimensional data.

Traditional DBMSs manage records which can be indexed based on their primary keys. B+-trees are the widely used index structure for unique keys (one dimensional data). Quad trees [43] proposed by Finkel and Bentley in 1974 and k-d trees [44] proposed by Bentley in 1975 are primary storage structures for composite keys (low-dimensional data). K-D-B-trees [45] proposed by Robinson in 1981 are secondary storage structures combining properties of k-d trees and B-trees for point data. With the requirement of the emerging spatial databases, e.g. Computer Aided Design (CAD) and Geographic Information System (GIS), the R-tree [46] was proposed by Guttman in 1984. It is the first index structure which can handle spatial data and is designed for secondary storage. The grid file proposed by Nievergelt et al. in the same year is a typical access method based on hashing. Due to the overlap of the minimum bounding boxes (MBRs), a query may take several paths in R-trees. R+-trees [47] proposed by Sellis et al. in 1987 and R*-trees [48] proposed by Beckmann in 1990 are two improved version of R-trees. R+-trees avoid overlap by inserting an object into multiple MBRs if necessary. R*-trees, which is the most successful variant of R-trees, incorporate a combined optimization of area, margin and overlap and use forced reinserts to reduce overlaps. Ranges are stored on each dimension, the index requires much more space and time to process queries when the dimensionality is high.

The aforementioned methods are efficient in low dimensions (2-3 dimension). Many new applications has data in the order of 10 or 100 dimensions. TV-trees [36] proposed by Lin et al. in 1994 use telescopic vectors to extend or contract the dimensions for representing the bounding boxes. SS-trees [49] proposed by White and Jain in 1996 use hypershperes to partition the space, they reduce the space storage of the index greatly without causing performance degradation. Since hyperspheres tend to have large overlaps, SR-trees [17] proposed by Katayama and Satoh in 1997 use the intersection of hyperspheres and hyperrectangles to represent regions. SR-trees outperform both R*-trees and SS-trees. Also, in 1996, X-trees [32] proposed by Berchtold et al. are partially linear and partially hierarchical index structures using supernodes to avoid overlaps. The M-tree [50] proposed by Ciaccia in 1997 is an index structure based on metric spaces.

Year 1998 is a milestone in high-dimensional indexing. Gaede and Günther publish a survey on the multi-dimensional index structures in [11]. Weber et al. [18] give a quantitative analysis of existing partitioning and clustering techniques for similarity search in high-dimensional vector spaces and conclude that existing methods are outperformed by a sequential scan when the number of dimensions exceeds ten. They also propose the Vector Approximation - VA-File and report experimental results showing that it outperforms the R*-tree and X-tree for nearest neighbor search when the number of dimensions is larger than around six. In the same year, the Pyramid technique [51] is proposed. It is the only index structure known so far that is not affected by the curse of dimensionality [52]. For uniform data and range queries, its performance improves with increasing dimensionality. CSVD [53, 33] proposed by Thomasian et al. reduces the number of dimensions by singular value decomposition to tackle the curse of dimensionality.

In 1999, Beyer et al. [54] explore the effect of dimensionality on nearest neighbor problems. They point out that as the dimensionality increases, all the points are equidistance to query point under a broad set of conditions. Even for the datasets for which this effect does not occur, a linear scan outperforms most existing high-dimensional indices in high (10-15) dimensionality.

In 2000, Ooi et al. proposed the iMinMax(θ) [55] method which maps points in high-dimensional space to single-dimensional space. Experiments show that iMinMax(θ) can outperform Pyramid for range queries. One year later, C. Yu et al. proposed the iDistance [56] method which is for nearest neighbor search in high-dimensional spaces. It partitions the data and selects a reference for each partition. Data in each cluster are transformed into single-dimensional data which are then indexed by B+-trees. k-NN searches are performed by using range queries on the B+-trees. The A-tree [57] proposed by Sakurai et al. in 2000 introduce the virtual bounding rectangles which contain and approximate MBRs and objects.

In addition to CSVD, LDR [14] and MMDR [23] methods are proposed by Chakrabarti and Mehrotra in 2000 and Jin et al. in 2003, respectively. LDR finds local correlations and performs dimensionality reduction on the locally correlated data. MMDR uses an adaptive Multi-level Mahalanobis-based Dimensionality Reduction technique to reduce the dimensionality of the original dataset before constructing the index. The OMNI-family proposed by Filho et al. in 2001 uses a set of predefined foci to filter the search space [58]. For NN searches it has to employ an estimation method for the nearest neighbor sphere radius, e.g. fractal dimensions. Also in 2003, the Δ-tree proposed by Cui et al., which is a main memory index structure, represents each level with a different number of dimensions. The number of dimensions increases towards the leaf level, which contains full dimensions of the data.

In summary, the state-of-the-art techniques can be classified as dividing the space into different shapes (e.g. hyperspheres, hyperrectangles), transforming high-dimensional space into one-dimensional space, and transforming high-dimensional space into lower-dimensional space.

2.4.2 Indexing on Order Spaces

Indexing methods in this category transform data in high-dimensional space onto one-dimensional space using space-filling curves, such as z-ordering, the Hilbert curve, etc. A single-dimensional indexing methods, like B+-tree is built for the order space. Queries are transformed into the single dimensional space first, then search on the B+-tree. In [59], N distinct dimensions are grouped into H disjoint clusters and each cluster is mapped into

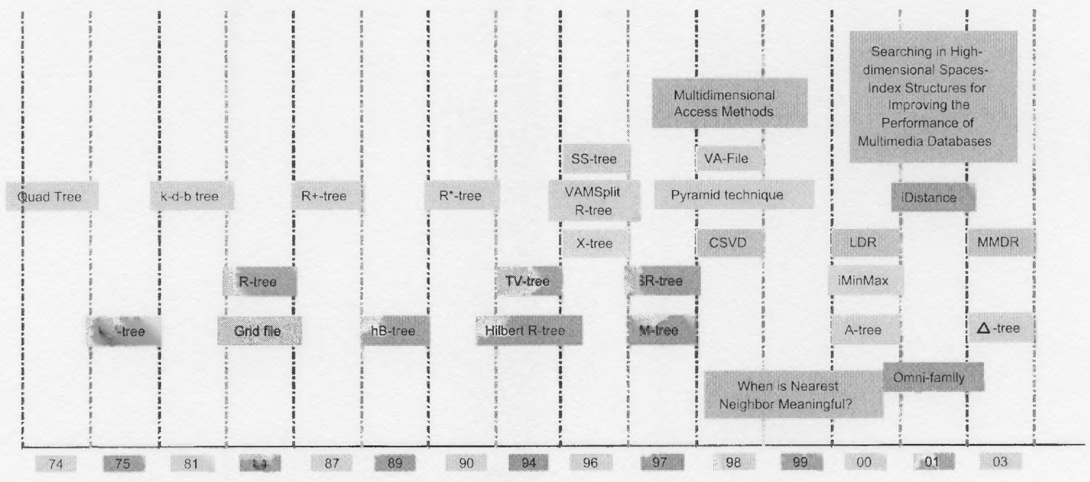


Figure 2.3 The evolution of high-dimensional indexing structures.

a one-dimensional space by using the Hilbert curve. The resulting H-dimensional space (which is much lower than N) is then indexed using efficient low-dimensional index structure, such as R-trees.

2.4.3 Indexing on Feature Vector Spaces

This category can be further divided into Space Partitioning (SP)-based and Data Partitioning (DP)-based index structures [60]. A DP-based index structure uses Bounding Regions (BRs) to divide the space. The BRs tend to be heavily overlapped at high dimensions. The index will have a lower fanouts with the dimensionality increasing. The BRs can be bounding boxes (e.g., R-tree [46], R+-tree [47], R*-tree [48], X-tree [32]) or bounding spheres (e.g. SS-tree [49]) or intersection of both (e.g. SR-tree [17]). An SP-based index structure partitions the space into mutually disjoint subspaces recursively. Some of the examples are the k-d-tree [31], the K-D-B-tree [45], and the hB-tree [61]. SP-based techniques have fanout independent of dimensionality. The hybrid tree [60] combines positive aspects of DP-based and SP-based techniques to achieve improved search performance in high dimensions. The DP-based index structure have much impact on this study, they will be addressed in separate chapters.

The VA-File differs from any partitioning scheme and clustering technique, in that it is a flat sequentially accessed file. Experiments in [18] show that its performance improves as dimensionality increases. Thus, it is worth to describe here.

VA-File The Vector Approximation File (VA-File) [18] represents each data object using the cell into which it falls. Due to the sparsity of high-dimensional space, it is very unlikely that several points can share a cell. Let b_i be the number of bits to represent the partition along dimension i, the total number of bits to represent a cell in N-dimensional space is $b = \sum_{i=1}^{N} b_i$ and the total number of cells is 2^b. The probability of a point falling into a cell is 2^{-b} and the probability of at least two points fall into one cell is approximately $N/2^b$. For a $M \times N$ dataset with $M = 10^6 \approx 2^{20}$ and $N = 50$, with $b_i = 2, i = 1, \cdots, N$, the probability for sharing a cell is 2^{-80}. This discussion is true when the data points are uniformly distributed in space.

Nearest neighbor search sequentially scans the VA-File to determine the upper bound and lower bound distance from the query to each cell. During the filter step, if the lower bound of an object approximation is greater than the current upper bound, it is out of consideration. Otherwise, it is a candidate. During the refine step, all the candidates are sorted according to their lower bound distances. The actual objects are retrieved and the distance to the query is computed. The nearest ones are returned. The drawback is that the performance is highly dependent on the number of bits per dimension b_i. Table 2.1 shows that $b_i = 4$ gives best performance for k-nearest neighbors with $k = 20$ for dataset TXT55. The floating point operations (FO) include scanning the VA-File plus postprocessing.

2.4.4 Indexing on Metric Spaces

A broad class of index structures, metric trees [62], transform the feature vector space into metric space, and then index the metric space. A metric space is a pair, $\mathcal{M} = (\mathcal{F}, d)$, where $\mathcal{F}$ is a domain of feature values, and d is a distance function with the following properties:

Table 2.1 Average 20-NN Search Results over 1000 Randomly Selected Queries on Dataset TXT55; PV - Average Number of Points Visited, FO - Average Number of Floating Point Operations, CPU - Average CPU Time in Seconds

Dim	b_i=3			b_i=4			b_i=5			b_i=6		
	PV	FO	CPU	PV	*FO*	CPU	PV	FO	CPU	PV	FO	CPU
55	157.52	2867390	0.1	74.51	*15813.8*	0.038	46.7	680349	0.041	34.05	512842	0.036
50	158.17	2630820	0.092	74.67	*14399.8*	0.035	46.77	655570	0.038	34.09	497484	0.033
40	154.32	198983	0.073	73.58	*11389.4*	0.031	46.03	567998	0.034	34.04	450326	0.031
30	155.12	1466710	0.056	74.38	*8614.11*	0.026	46.44	476371	0.028	34.35	387146	0.026
20	167.37	892583	0.038	79.41	*6044.72*	0.02	48.59	348464	0.023	35.49	294135	0.021
10	239.78	426341	0.022	99.6	*3627.91*	0.012	56.12	210860	0.015	38.47	182599	0.013
5	444.57	224213	0.013	161.77	*2746.58*	0.009	78.69	137875	0.01	47.55	125564	0.011

- **Symmetry** $d(F_x, F_y) = d(F_y, F_x)$
- **Non-negativity** $\begin{cases} d(F_x, F_y) > 0 & F_x \neq F_y \\ d(F_x, F_y) = 0 & otherwise \end{cases}$
- **Triangle inequality** $d(F_x, F_y) \leq d(F_x, F_z) + d(F_z, F_y)$

A metric tree organizes and partitions the search space based on relative distances of objects, rather than their absolute positions in a multi-dimensional space. It requires that the function used to measure the distance (dissimilarity) between objects is a metric, so that the triangle inequality property applies and can be used to prune the search space.

The vp-tree The Vantage Point - vp-tree [1] partitions a dataset according to distances between the objects and a reference (vantage) point. The corner point is chosen as the vantage point and the median value of the distances is chosen as separating radius to partition dataset into two balanced subsets. The same procedure is applied recursively on each subset, which is shown in Figure 2.4. The mvp-tree [63] uses multiple vantage points and exploits pre-computed distances in the leaf nodes to provide further filtering during search operations. Both of the trees are built in a top-down manner, balance can not be guaranteed during insertion and deletion. Costly reorganization are required to prevent performance degradation.

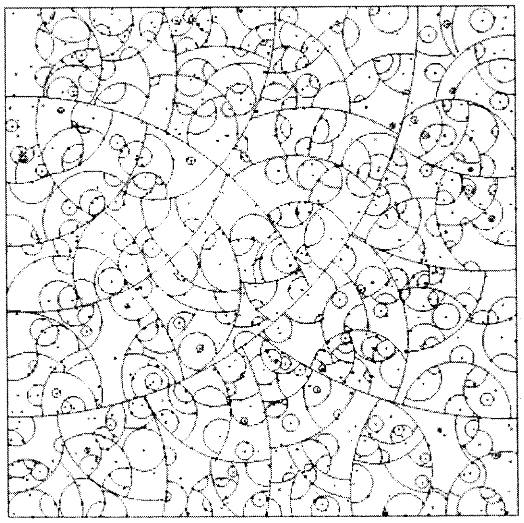


Figure 2.4 The vp-tree [1].

The M-tree The M-tree [50] is a paged metric-tree index. It is balanced and able to deal with dynamic data. Leaf nodes of an M-tree store the feature vectors of the indexed objects O_j and distances to their parents, whereas internal nodes store routing objects O_r, distances to their parents O_p, covering radii $r(O_r)$ and corresponding covering tree pointers. The M-tree reduces the number of distance computations by storing distances. The following lemmas [50] are used to prune search space for a given query Q and search radius $r(Q)$, which is illustrated as in Figure 2.5.

Lemma 2.1 *If $d(Q, O_p) > r(Q) + r(O_p)$, then $d(Q, O_j) > r(Q)$ for each object O_j in the tree $T(O_p)$ rooted at O_p. Thus, $T(O_p)$ can be safely pruned from the search.*

Lemma 2.2 *if $|d(Q, O_p) - d(O_r, O_p)| > r(Q) + r(O_r)$, then $d(O_r, Q) > r(Q) + r(O_r)$. Thus, $T(O_r)$ can be safely pruned from the search.*

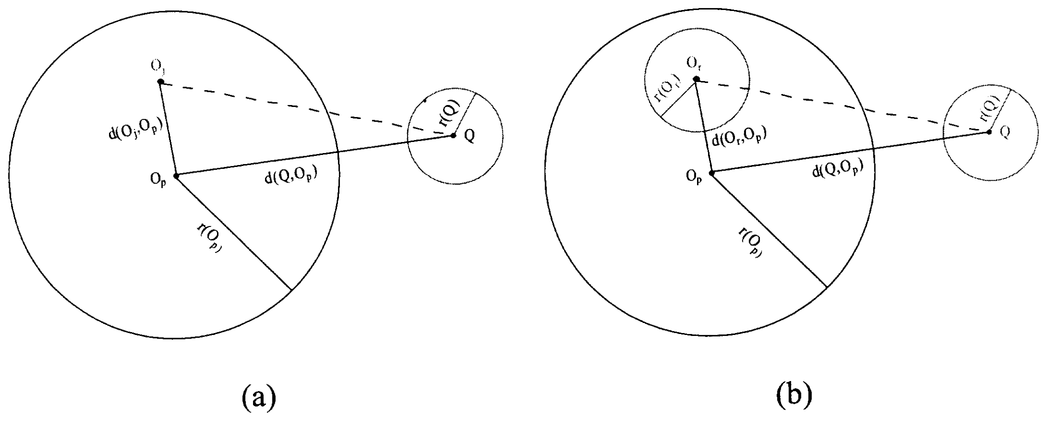


Figure 2.5 Pruning principles for (a) Lemma 2.1, (b) Lemma 2.2.

The OMNI-Family The OMNI-Family [58] is a set of indexing methods based on the same underlying theory that all the points S_i located between l and u are candidate results for a spherical query with radius r and given point Q for a specific focus F_i, where $l = d(Q, F_i) - r$, $u = d(Q, F_i) + r$. For multiple foci, the candidates are the intersections of S_i. Figure 2.6 shows the search candidates for a range query centered at Q with search radius r.

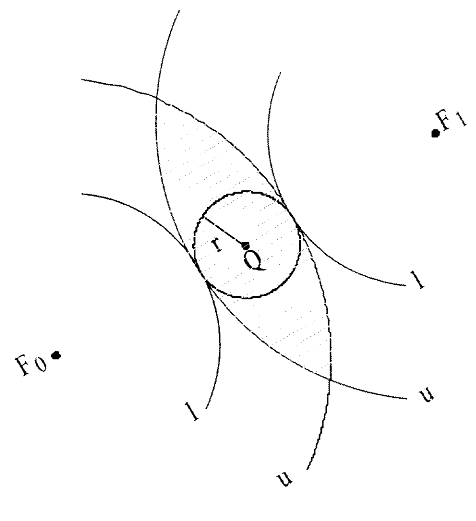


Figure 2.6 Search candidates (blind area) for a range query (Q, r) in the case of two foci.

Given a dataset, a set of foci was found. For each point in the dataset, calculate and store the distance to each of the foci. The search process can be applied to sequential scan, B+-trees and R-trees. For B+-trees, the distances for each focus F_i are indexed, a range query is performed on each index, finally the intersection is obtained. For R-trees, the distances for all the foci, which forms a lower dimensional data, are indexed, and single range query are performed.

The iDistance The iDistance [56] is proposed for efficient k-NN search in a high dimensional space. Data is partitioned into several clusters and each partition has a reference point. The data in each cluster are transformed into a single dimensional space according to the similarity with respect to a reference point. The one-dimensional value of different clusters are disjoint. A B+-tree can be used to index the one-dimensional space and k-NN search are implemented using range searches. The search starts with a small radius and the radius is increased step by step to form a bigger query sphere. The iDistance is lossy since multiple data points in the high-dimensional space may be mapped to the same value in the single dimensional space.

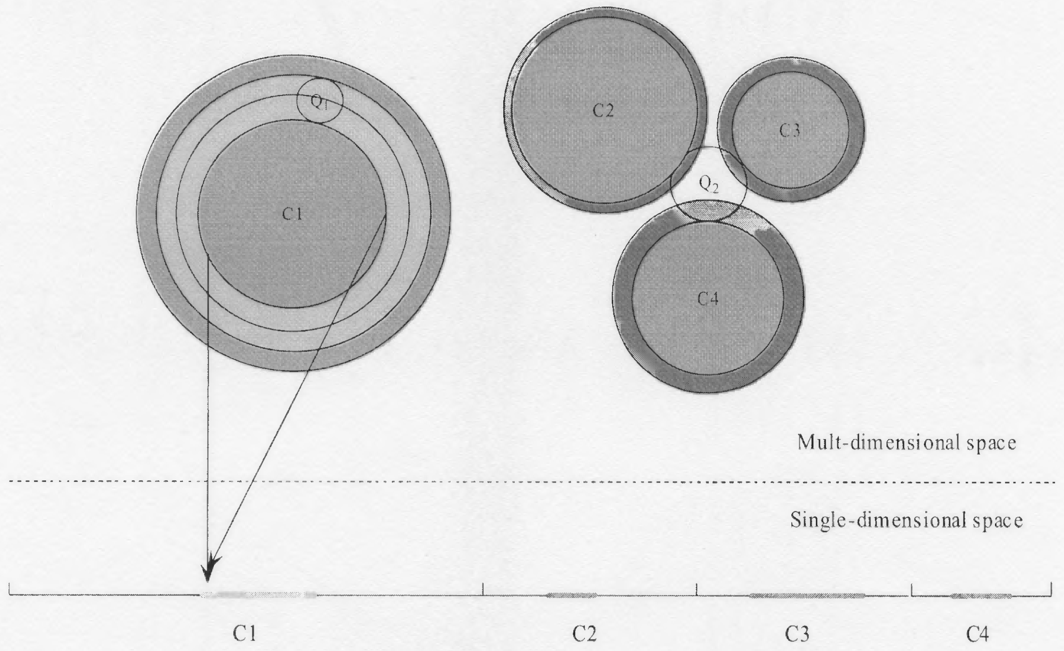


Figure 2.7 Searching space for range queries using the iDistance.

2.4.5 Cost Models for Nearest Neighbor Query Processing

Due to the high practical relevance of nearest neighbor queries, cost models for estimating the number of necessary page accesses have been proposed [64], such as the traditional NN-model [65], exact NN-model [66] and analytical NN-model [67].

The traditional NN-model [65] assumes that the number of data objects converges to infinity and boundary effects are not considered, which is unrealistic.

The exact NN-model [66] determines the number of data pages which has to be accessed on the average taking into account boundary effects. Experiments show that the traditional NN-model overestimate the cost by orders of magnitude in high dimensions, while the exact NN-model is accurate up to a moderate relative error. It has been used for constructing the X-tree index [32] and DABS index [68].

The analytical NN-model [67] provides a closed formula for the processing costs of nearest neighbor queries depending on the dimensionality, the block size and the dataset size. Experiments show that the analytical cost model provides an accurate prediction of R*-tree performance over a wide range of dimensions.

2.5 Nearest Neighbor Search Algorithms

Efficient support of nearest neighbor search is important in modern database applications. Researches have focused on two aspects: developing algorithms applied to existing index structures and developing specialized index structures suitable for nearest neighbor search. Examples of specialized index structures are NN-cell approach [69], SR-tree [17], SS-tree [49], OP-tree [16], PAT-tree [70], and iDistance [56]. In this section, the focus is on the first aspect.

Two classes of algorithms have been proposed for k-NN search. One class utilizes branch and bound algorithms, and the other class utilizes range queries. Two popular algorithms in the first category are the HS [71] and RKV [72] algorithm. They can be applied to any hierarchical index structures, such as R*-trees [48], SS-trees [49], and X-trees [32]. In the second case, iterative range queries are utilized to evaluate nearest neighbor queries.

A number of incremental algorithms for similarity ranking have also been proposed that can efficiently compute the (k+1)th nearest neighbor, after the k nearest neighbors are returned [73, 74]. A global priority queue of the objects to be visited is used.

2.5.1 Sequential Scan

For a given query point, the distance to each data objects is calculated and stored in a minimum priority queue with the key as the distance. The queue can be implemented with fixed length k. The top k objects are the k nearest neighbors. A simple way is to keep k candidates, each time when a new candidate appears, remove the farthest one and insert the new one.

2.5.2 The RKV Algorithm

Since R-tree [46] is designed for window queries which is defined in Appendix A, Roussopoulos et al. proposed a branch-and-bound R-tree traversal algorithm to find nearest neighbors in [72], which is referred to as the RKV algorithm [52] in this dissertation. An R-tree is built by first presorting the data files using a Hilbert [39] number generating function, and then applying a modified version of [75] R-tree packing technique according to the suggestion of [76]. Traversal of the tree is ordered and pruned based on a number of heuristics. In fact, the algorithm is not limited to the R-tree. Cheung and Fu simplified this algorithm without reducing its efficiency in [77].

Two important metrics, minimum distance - MINDIST and minimax distance - MINMAXDIST are introduced. MINDIST is the nearest possible distance between a point and a Minimum Bounding Rectangle - MBR [52], which means no points in the region has distance to the given point closer than the MINDIST. The MINMAXDIST guarantees that there is an object within the page region at a distance less than or equal to MINMAXDIST [52]. Figure 2.8 shows the two metrics in two-dimensional space.

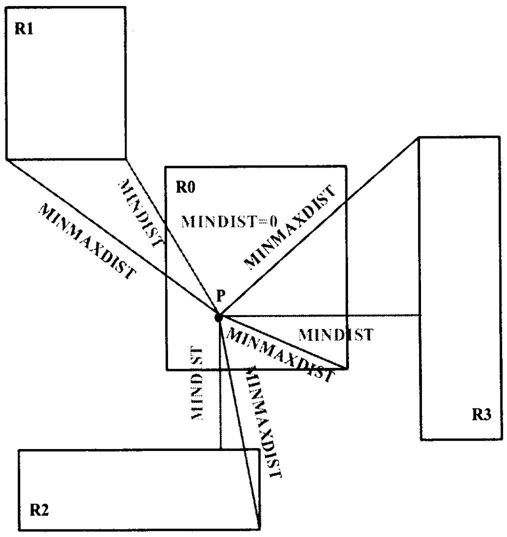


Figure 2.8 MINDIST and MINMAXDIST in two-dimensional space.

To give the formal definition, let a N-dimensional rectangle R be represented by $(l_1, u_1, \cdots, l_N, u_N)$, where l_i and u_i are the lower and upper boundaries along dimension i.

Definition 2.5 (MINDIST) *The MINDIST between a point P and a rectangle R is defined as:*

$$MINDIST(P, R) = \sum_{i=1}^{N} |p_i - r_i|^2$$

where

$$r_i = \begin{cases} l_i & p_i < l_i \\ u_i & p_i > u_i \\ p_i & otherwise \end{cases}$$

Definition 2.6 (MINMAXDIST) *The MINMAXDIST between a point P and a rectangle R is defined as:*

$$MINMAXDIST(P, R) = min_{1 \leq k \leq N}(|p_k - rm_k|^2 + \sum_{i \neq k, 1 \leq i \leq n} |p_i - rM_i|^2)$$

where

$$rm_k = \begin{cases} l_k & p_k \le \frac{l_k+u_k}{2} \\ u_k & otherwise \end{cases} \quad and \quad rM_i = \begin{cases} l_i & p_i \ge \frac{l_i+u_i}{2} \\ u_i & otherwise \end{cases}$$

The RKV algorithm accesses pages in a depth-first order. During the search, any page region whose MINDIST is larger than the current farthest distance d_k will be pruned, any MBR whose MINMAXDIST is smaller than d_k will be visited. Although the MINDIST metric produces most optimistic orderings, it is not always the best choice [72]. The pseudocode is given as Algorithm 1.

Algorithm 1 RKV_KNN(Node∗ n, Query∗ q, int k)

```
1:  $d_k = \infty$;                                  //initiate distance to $k$th NN found so far
2:  if $n$ is not a leaf node then
3:      compute the metrics to each entry;         //metric: MINDIST or MINMAXDIST
4:      sort entries according to the metrics;
5:      for each entry $e$ do
6:          if MINDIST($e, q$) $\le d_k$ then
7:              RKV_KNN($e.node$, $q$, $k$);       //recursively search subtree $e.node$
8:              if MINMAXDIST($e, q$) $< d_k$ then
9:                  $d_k$ = MINMAXDIST($e, q$);
10: else
11:     for each object $o$ do
12:         compute the distance $d$ to $q$;
13:         if $d < d_k$ then
14:             insert $(d, o)$ into the result set;
15:             refine $d_k$;
```

2.5.3 The HS Algorithm

Hjaltason and Samet propose an incremental nearest neighbor finding algorithm in [71]. The algorithm can be adapted to any tree-like hierarchical index structures. A minimum priority queue is used to store addresses to the internal nodes (pages), leaf nodes (pages) and data objects and their distances to the query with the distance as the priority. Index pages are accessed in the order of increasing distance to the query point, which means the accessed pages can jump between different levels and branches of the hierarchical index structure [52]. Let's call a page *active* if its parent has been processed but not the page itself. APL denotes Active Page List which is implemented as the priority queue. Figure 2.9 illustrate the priority queue in the incremental algorithm.

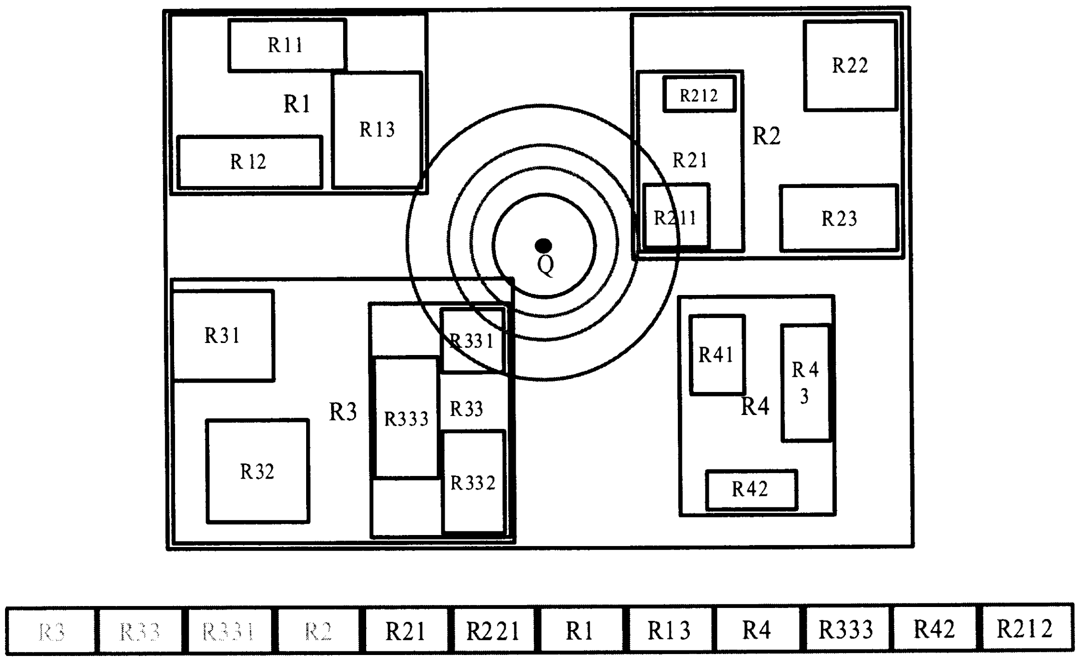


Figure 2.9 The priority queue used in the incremental algoirthm for finding nearest neighbors.

The HS algorithm referred here is the extended incremental algorithm for k-nearest neighbor processing. Two priority queues are usually used, one is a min priority queue *pque_index* for the indexed pages, the other is a max priority queue *pque_knn* with fixed length k for the results. From the algorithm described below, *pque_index* does not store information related to data objects which will be determined whether they should be inserted into the result queue or not once they are encountered. This can reduce the burden of operations on the queue since too many data points may be encountered. In experiments, this can also reduce the running time greatly. The HS algorithm has been shown to be optimal in terms of the number of pages accesses, which means it accesses as few pages as possible for a given index. The proof for the optimality can be found in [52]. The HS algorithm can be summarized as Algorithm 2.

Algorithm 2 HS_KNN(Node$*$ r, Query$*$ q, int k)

```
1: $d_k = \infty$;                    //initiate distance to $k$th NN found so far
2: push $(0, r)$ into pque_index;
3: while pque_index is not empty do
4:     pop up the top element $t$;
5:     if $t.d \geq d_k$ then
6:         break;
7:     if $t.node$ is not a leaf node then
8:         for each entry $e$ do
9:             calculate the MINDIST $d$ to $q$;
10:            if $(d < d_k)$ then
11:                insert $(d, e.node)$ into pque_index;
12:    else
13:        for each object $o$ do
14:            calculate the distance $d$ to $q$;
15:            if $(d < d_k)$ then
16:                insert $(d, o)$ into pque_knn;
17:                update $d_k$;
```

2.5.4 Range Search Based Algorithm

k-NN queries are performed using iterative range queries. To retrieve the complete answer set, the distance between query Q and the kth nearest neighbor is required. Unfortunately, this radius is hard to predetermine. Usually, the approach begins with a relatively small radius. The correlation fractal dimension can be used to estimate this radius for a given k [78]. The data within the radius are checked and a set of candidate nearest neighbors is found out. Then a larger radius is searched iteratively until no more new candidate is added. This is very time consuming, since it is difficult to determine how large the radius should be increased at each iteration. If the increase is too small, many iterations will be needed. Otherwise, too much data will be examined.

2.5.5 Multi-Step Nearest Neighbor Search

There is a context where indices are built based on dimensionality reduced data and want to find the nearest neighbors in the original data. Korn et al. proposes a multi-step algorithm in [79, 80] by finding k-nearest neighbors first based on the dimensionality reduced indices, then obtain the distance d_k for the kth neighbors in the original space, and finally run a range query with radius d_k. Thomasian et al. extend this algorithm to multiple clusters. Seidl and Kriegel propose an optimal multi-step algorithm in [81] by incorporating the original distance finding step into the nearest neighbor search step on the dimensionality reduced indices.

2.5.6 Approximate Nearest Neighbor Search

The above mentioned nearest neighbor search focuses on getting exact results for queries, where exactness is defined in terms of the feature vectors and a distance function between them [82]. However, exact results are very difficult to obtain. Besides, the meaning of exact is highly subjective and depends on the way the feature vectors are created and the distance function defined between the feature vectors. The data itself is an approximate representation of real world entities, so close approximations may be good enough for human perception. The quality of the result set is measured by a combination of *recall* and *precision* [82]. *Recall* is a measure of completeness of retrieval and *precision* is a measure of purity of retrieval. The irrelevant objects in the result set are called *false hits* and the relevant objects that are not in the result set are *false dismissals*.

A variety of approximate nearest neighbor search algorithms are developed to improve the query processing. Current approaches either reduce the dataset that needs to be examined, or reduce the representation size of each data object [82]. Global Dimensionality Reduction (GDR), Local Dimensionality Reduction (LDR) [14], Clustering and Singular Value Decomposition (CSVD) [53, 33, 3], and Multi-level Mahalanobis-based Dimensionality Reduction (MMDR) [23] are efficient approximate nearest neighbor search algorithms. The performance study of CSVD with indexing is studied in Chapter 3.

Year	Index	Query Type	Compared with	Dataset	Metric	Comments
75	k-d tree	Exact Match Partial Match Range NN	-	-	-	-
81	k-d-b tree	Partial Match Range	-	-	-	-
84	R-tree	Range	-	-	-	-
84	Grid file	-	-	-	-	-
87	R+-tree	Range	R-tree	-	-	-
90	R*-tree	Partial Match Range	R-tree	2-D	-	Both spatial and point objects
94	TV-tree	Exact Match Range Spatial Join NN	R*-tree	27-D	L_1	Experimented exact match and range query
96	SS-tree	NN	R*-tree	11-D uniform 11-D Gaussian 100-D Eigenface	-	-
96	X-tree	Point NN	R*-tree TV-tree	16-D fourier 16-D CAD data 32-D uniform	-	R*-tree better than TV-tree when D<16 for point query
97	SR-tree	NN	SS-tree VAMSplit R-tree	16-D uniform 16-D histogram	-	-
98	Pyramid	Range	X-tree Hilbert R-tree Seqscan	100-D uniform 16-D text (www) 13-D warehouse	L_{max}	The warehouse data includes 2 categorical, 2 int, and 2 float
98	VA file	NN	R*-tree X-tree Seqscan	45-D features	-	All approaches to NN in HDVSs ultimately become linear at high dimensionality. VA-file outperforms all other methods known to the authors when D >= 6, The tree methods degenerate to a scan through all the leaf nodes for NN
00	iMinMax	Range	Pyramid	8-50-D uniform 30-D normal 30-D exponential	-	Uniform and skewed data sets
01	iDistance	NN	A-tree Seqscan iMinMax	30-D uniform 30-D clustered	-	-
01	Omni-family	Range NN	Seqscan SlimTree	EnglishWords 16-D Eigenfaces 30-D Sinthetic	L_{edit} L_2 L_2	For NN using OmniR-tree and OmniB-tree need to use fractal dimension to estimate the range for NN sphere
03	Delta-tree	NN	iDistance M-tree TV-tree CR-tree Seqscan VA-file	64-D uniform 8-64 LDRgen 64-D histogram	-	64-D color histogram is obtained from the Corel Database

Figure 2.10 Outline of high-dimensional indexing structures.

CHAPTER 3

PERFORMANCE STUDY OF CSVD WITH INDEXING

3.1 Introduction

The nearest neighbors problem is of major importance to a variety of applications, where similarity search is usually employed. Typically, the application objects are represented using the extracted features, which are in fact high-dimensional data points, and a distance metric provided by domain experts is used to measure the (dis)similarity of objects. A great attention has been paid to find the exact nearest neighbors in terms of the feature vectors and the provided distance function. However, the selection of features and distance metrics is based on heuristics, so the meaning of exact is highly subjective and depends on an approximation of real world entities. Close approximations may be good enough for human perception and it seems an overkill to insist on the exact nearest neighbor.

Clustering and Singular Value Decomposition (CSVD) [33, 3] is an approximate similarity search method in high-dimensional spaces. CSVD groups homogeneous data into clusters, and reduces the dimensionality by using SVD. Cluster selection relies on a branch-and-bound algorithm, and within-cluster searches can be performed with sequential scan or indexing methods.

The within-cluster index can be any multi-dimensional index structure, either main memory or disk resident. A main memory index structure (ordered partition index [16]) is used in [3]. There are two drawbacks. One is that the memory should be large enough to hold all the indices, and the other is that a sufficient number of queries need to be executed to trade off the cost of building the index, since it is volatile. Therefore, this study focus on using disk resident index structures. First three multi-dimensional indices are compared, the best one is selected as the within-cluster index. Then two approximate distance in the dimensionality reduced subspace is presented and one is proved to be much closer to the distance in the original space. Experiments evaluate the performance of the CSVD method, which includes the precision, recall, number of pages visited, CPU time, and effects of the degree of clustering.

The outline of this chapter is as follows. After introducing the CSVD method in Section 3.2, three disk-resident index structures, R*-trees, SR-trees and hybrid-trees, are described in Section 3.3. Section 3.4 describes the approximate nearest neighbor search algorithm for CSVD. In Section 3.5, the performance of two nearest neighbor search algorithms applied on the same structure are studied and the performance of different index structures using the same algorithm are compared. The SR-tree index shows the best performance and is selected as the within-cluster index for studying CSVD performance.

3.2 Clustering and Singular Value Decomposition

Given an $M \times N$ dataset X, let μ_j be the mean of column j, μ be a vector composed of μ_j and $X' = X - 1_M \mu^T$. SVD decomposes X' as $X' = USV^T$, where U is an M $\times$ N matrix, V contains the eigenvectors and S contains singular values of X'. PCA decomposes the covariance matrix $C = \frac{1}{M} X'^T X'$ as $C = V \Lambda V^T$, where $s_j^2/M = \lambda_j$. Let $Y = X'V$, the number of dimensions n can be obtained for a given error tolerance and dimensionality reduction is achieved by keeping the first n dimensions of Y.

NMSE The *Normalized Mean Squared Error - NMSE* quantify the loss of distance information caused by dimensionality reduction [3, 83]. Equation 3.1 and 3.2 define the NMSE for one cluster and H clusters, respectively.

$$NMSE = \frac{\sum_{i=1}^{M} \sum_{j=n+1}^{N} y_{i,j}^2}{\sum_{i=1}^{M} \sum_{j=1}^{N} y_{i,j}^2} = \frac{\sum_{i=1}^{M} \sum_{j=n+1}^{N} y_{i,j}^2}{\sum_{i=1}^{M} \sum_{j=1}^{N} (x_{i,j} - \mu_j)^2} = \frac{\sum_{j=n+1}^{N} \lambda_j}{\sum_{j=1}^{N} \lambda_j}. \tag{3.1}$$

$$NMSE = \frac{\sum_{h=1}^{H} m_h \sum_{j=n_h+1}^{N} \lambda_{h,j}}{\sum_{h=1}^{H} m_h \sum_{j=1}^{N} \lambda_{h,j}}. \tag{3.2}$$

Recall* and *Precision are useful measures for approximate methods. *Recall* is the percentage of relevant elements which are retrieved, while *precision* is the percentage of retrieved elements which are relevant. Let R_v denote the subset containing the k nearest neighbors of query Q. To account for the approximation, more than k elements are requested. Let R_t be the set of points retrieved and $| \,.\, |$ denote the cardinality. *Recall* $\mathcal{R}$ and *precision* $\mathcal{P}$ are given as:

$$\mathcal{R} = E[|R_v \cap R_t|/|R_v|], \qquad \mathcal{P} = E[|R_v \cap R_t|/|R_t|].$$

$\mathcal{R}$ and $\mathcal{P}$ are inversely related. One can be increased at the expense of another. Let k^* denote the number of results that must be retrieved for a k-NN query to yield a *precision* equal to $\mathcal{P}$ and *recall* equal to $\mathcal{R}$, then $k^* = k \cdot \mathcal{R}/\mathcal{P}$.

Before constructing the index, data is preprocessed. Numerical values on different features can be appropriately scaled to equalize their relative importance when the metric of choice is the Euclidean distance. Studentization can be applied to each feature by subtracting the mean and dividing the result by the standard deviation. The CSVD proceeds as the following five steps:

Step 1. Specifying a target NMSE to tolerate.

Step 2. Partitioning the dataset.

The dataset is partitioned into H clusters, each containing data that are close to each other in terms of Euclidean distance. All classical clustering methods such as k-means, LBG [84], and TSVQ [85] are applicable, but in fact the k-means method is used in this study. Each cluster has a radius which is defined as the distance between its centroid and the point farthest from the centroid. The k-means method is usually run several times and the partition with the smallest SSE (Equation 2.1) is kept.

To reduce the number of clusters visited during queries, outliers can be found from the whole dataset, and kept in a separate list, then the remaining dataset (excluding the outliers) is clustered. The radius of each cluster is expected to be smaller, and thus less clusters are visited. However, in the experiments during this study for high-dimensional data, this is not the case.

Step 3. Rotating each partition into an uncorrelated frame of reference.

The principal components of each cluster are found by applying SVD to clusters individually. The data is rotated into the reference frame composed of the principal components.

Step 4. Reducing the dimensionality of the partitions.

Dimensionality reduction is a global procedure which is applied to all the clusters simultaneously. An $H \cdot N$ array $\mathcal{L}$ is constructed, the jth element of which is a triple $(\kappa_j, \partial_j, \lambda_{\partial_j}^{(\kappa_j)})$; where $\kappa_j \in \{1, \cdots, H\}$ denotes a cluster, $\partial_j \in \{1, \cdots, N\}$ is the label of a dimension, and $\lambda_{\partial_j}^{(\kappa_j)} = \lambda_j$ is the eigenvalue associated with dimension ∂_j of cluster κ_j. The elements of $\mathcal{L}$ are sorted in *increasing* order of eigenvalues, so that $\lambda_j \leq \lambda_{j+1}$ for each j. The ∂_j dimension of κ_jth cluster is removed from the beginning of the array $\mathcal{L}$ until the target *NMSE* specified in step 1 is reached.

Step 5. Constructing the within-cluster index.

Due to the fact that nearest neighbor search based on sequential scan of large datasets is computationally expensive, a multi-dimensional index is constructed for each cluster. Any of the known indexing techniques can be relied on if the intrinsic dimension of the cluster is low. Otherwise, an index which can handle relatively higher dimensions should be selected.

3.3 Performance Comparison of Index Structures

The selection of the within-cluster index plays an important role on the performance of nearest neighbor search of CSVD. When the properties of a dataset are known, a broad class of indices can be used. For example, well-known efficient spatial indices in the R-tree family for small intrinsic dimensions, specialized static indices for static high-dimensional datasets, etc. Otherwise, an index with good overall performance is preferred.

Indexing method can be categorized into data partitioning and space partitioning method. Data partitioning methods are based on hyperrectangles or hyperspheres. To determine which method works better with CSVD, three typical index structures, R*-tree [48], hybrid tree[60] and SR-tree[17], are studied. The R*-tree is based on hyperrectangles, the SR-tree is based on hyperrectangles and hyperspheres, and both are data partitioning indexing methods, while the hybrid tree is based on both data partitioning and space partitioning.

Analytical modelling of the performance for index structures is a difficult task. Moreover, comparisons based on theoretical upper bounds for worst case performance do not reflect the performance of real world applications [86]. Furthermore, there is no well-established benchmark. Therefore, empirical comparisons, which rely on the size of the index, the search time of the query, the pages visited, etc. are used instead. A fair comparison has to take into account the data type, search algorithm, and running platform. Even the implementation plays an important role. A good implementation for a bad algorithm can outperform a bad implementation for a good algorithm. The program codes used in this study are obtained from the original author, thus the possibility of a bad implementation is reduced. The R*-tree and hybrid tree codes are migrated from UNIX to Windows.

Due to the different nearest neighbor search algorithm they use, the HS algorithm [71] and RKV algorithm [72] based on the same index structure are first compared, and the conclusion is that the HS algorithm is always better. Then the HS algorithm is implemented on each of the structure, and the nearest neighbor search performance is compared. The above idea is especially useful to identify the best method to partition the data space, since in the area of high-dimensional indexing an essential problem is how to partition the data space. Various indices are compared based on different data space partitioning, each with its own algorithm. It is hard to tell whether the partitioning method improves the performance, or the k-NN search algorithm improves the performance. This idea can also be used to develop new index structures which combines the best partitioning with the best algorithm.

The R*-tree The R*-tree [48] is the most successful variant of the R-tree, which is a multi-dimensional generalization of the B-tree. The R*-tree uses hyperrectangles to partition the search space. The hyperrectangle associated with a particular node covers all the hyperrectangles of its children. The tree is constructed by inserting the feature vectors one at a time. Different orders of the same data can result in well or poorly constructed trees, thus affecting the search performance. Node splitting and merging are required for insertion and deletion of objects. The commonly used nearest neighbor search algorithm is the RKV algorithm, which is proposed based on the R-tree [46].

Each node has [e, E] entries. Good performance is obtained when $e = 0.4 * E$ as recommended in [48]. The size of an entry S_e is $sizeof(childptr) + sizeof(double) \times 2 \times M$ and the fanout is $\lfloor (pagesize - hdrsize - sizeof(bitmap))/S_e \rfloor$. The format of a R*-tree node is:

Table 3.1 Node Structure of the R*-tree

header(8 bytes)	entry1	entry2	...	entry$k$	bitmap
	childptr, cover rectangle				

The SR-tree (Sphere/Rectangle-tree) R*-trees [48] use hyperrectangles and SS-trees [49] use hyperspheres to partition the data space. Experiments show that bounding hyperspheres occupy much larger volume than bounding rectangles, and bounding hyperrectangles have much longer diameter than bounding spheres [17]. This affect the search efficiency of R*-trees and SS-trees.

SR-trees [17] combine the advantages of R*-trees and SS-trees. The region of each node is determined by the intersection of a bounding sphere and a bounding rectangle, which results in a significant reduction in the overlap between two sibling nodes of the SR-tree, especially for high dimensions. Figure 3.1 illustrates a SR-tree with 2-D representation and hierarchical representation. The SR-tree reduces both the volume and the diameter of regions compared with the R*-tree and the SS-tree and is more suitable for nearest neighbor queries. The storage required for the SR-tree is higher than the R*-tree and the SS-tree, and furthermore the creation cost of the SR-tree is higher than that of the SS-tree. On the other hand, the SR-tree provides a good performance for high-dimensional nearest neighbor queries. Figure 3.2 gives the internal and leaf node structure.

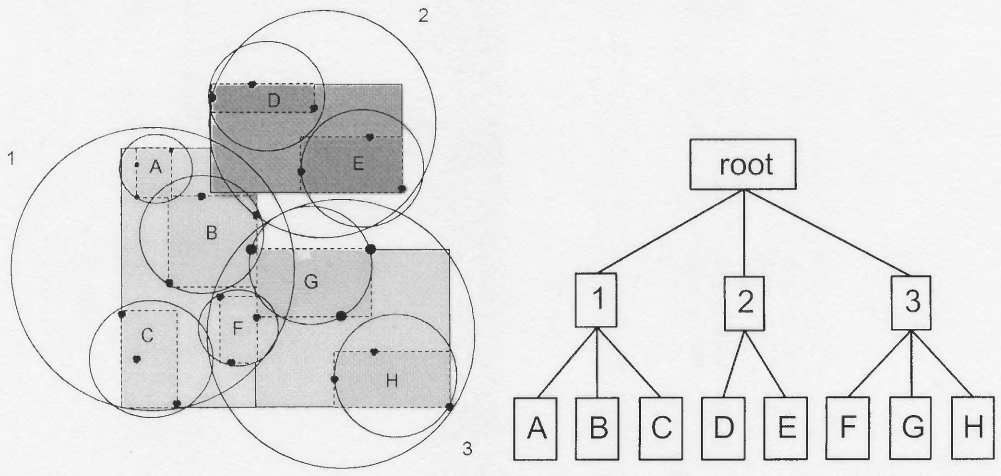


Figure 3.1 Two representations of the SR-tree.

The Hybrid Tree The hybrid tree [60] is neither a pure data partitioning (DP) index structure, nor a pure space partitioning (SP) index structure. A DP-based index consists of bounding regions (BRs) arranged in a containment hierarchy, like R-tree family, SS-trees [48], and SR-trees [17], while a SP-based index consists of recursively partitioned disjoint subspaces, like K-D-B trees [45] and hB-trees [61]. The hybrid tree combines positive aspects of DP-based indices and SP-based indices to achieve better scalability.

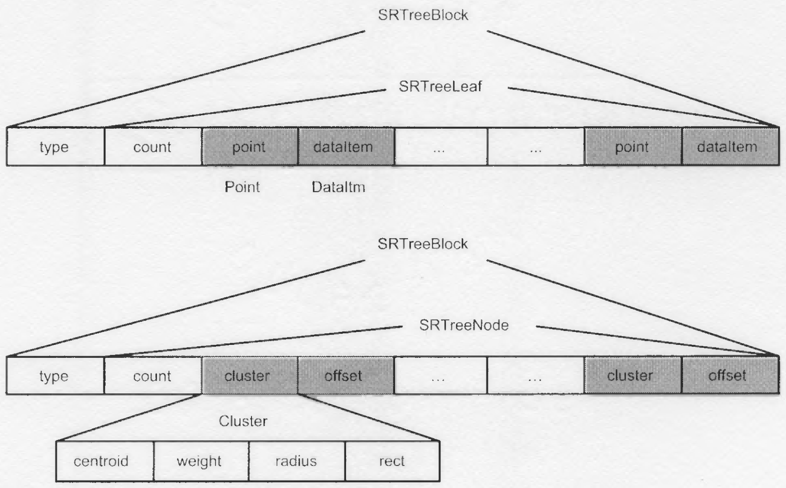


Figure 3.2 The internal (top) and leaf (bottom) node structure of the SR-tree.

The hybrid tree uses space partitioning strategies when a node splits. The split subspaces can be overlapped when trying to achieve an overlap-free split would cause downward cascading splits. The partitioning inside each index node is organized as a k-d tree [11] capable of representing possibly overlapping splits. This enables faster intranode search compared to array-based organization. The k-d tree stores both the split dimension and two split positions. The hybrid tree uses a single dimension to split the space, which makes its fanout independent of the dimensionality, thus has larger fanouts and smaller sizes.

Since operations on SP-based structures assume disjoint splits, the hybrid-tree treats the indexed subspaces as bounding regions in a DP-based data structure. A *logical* mapping is defined to map the kd-tree based representation to an "array of BRs" representation. Thus the algorithms used in DP-based data structures can be applied directly to the hybrid tree. The BRs are not computed during the tree traversal, rather computed only when necessary.

3.4 Approximate Nearest Neighbor Search

The approximate k-NN algorithm used by CSVD is as follows [3]:

Preprocessing The query point is studentized to yield Q. In order to remove the effect of different scales of each feature, the dataset is studentized during the index construction step, which means the N columns of the dataset X are studentized separately to obtain zero mean and unit variance. For each column j, the empirical mean u_j is subtracted and the result is divided by the estimated standard deviation $\hat{\sigma}_j$. The element of studentized dataset S is obtained by $s_{ij} = (x_{ij} - u_j)/\hat{\sigma}_j, 1 \leq i \leq M, 1 \leq j \leq N$.

Primary Cluster Identification The primary cluster to which Q belongs is identified. For k-means clustering, it is the cluster with the closest centroid to q. This conforms to the nature of the spherical property originally generated by k-means.

Computation of Distances from Clusters The distance between Q and a cluster c is defined as $max\left\{0, D(q, u^{(c)}) - R^{(c)}\right\}$. $D(q, u^{(c)})$ is the distance between Q and $u^{(c)}$, which is the centroid of cluster c. The clusters are stored in increasing distance order and ties are broken using $D(Q, u^{(c)})$.

Searching the Primary Cluster A k^* candidate results are produced to achieve a desired recall for a k-NN query. The results are kept in a maximum priority queue *pque_knn* with length k^*, the priority is based on the distance to the query. Let d_k be the distance of the top element of *pque_knn*.

Searching Candidate Clusters The next cluster is searched if its distance from Q does not exceed d_k; otherwise, the search terminates. If there exist points closer to the query than d_k, the points are inserted to *pque_knn* and d_k is updated.

Postprocessing The distances between Q and the k^* returned results are computed in the original space and the closest k results are returned.

During the within-cluster search, an approximate distance to the distance in the original space is required. In paper [3], an approximate distance $D'^2(P, Q) = D^2(P', Q)$ between projected point $P' = \{p_j\}, j \in \{1, \cdots, n\}$ of P and a query $Q = \{q_j\}, j \in$

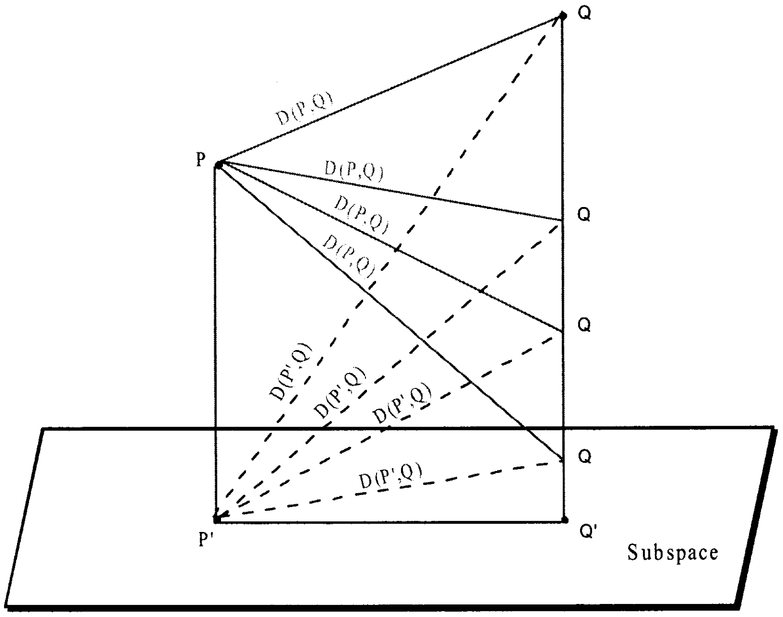


Figure 3.3 Effect of distance approximation by $D'(P, Q)$.

$\{1, \cdots, N\}$ to the original space is used with $D^2(P', Q) = D^2(P', Q') + \sum_{j=n+1}^{N} q_j^2$. This distance does not lower-bound the original distance $D^2(p, q)$, which is shown in Figure 3.3. With the position of the query point changing from farther to closer to the subspace, the approximate distance varies from being larger to smaller than the original distance.

An more approximate distance $\tilde{D}^2(p, q)$ to the squared Euclidean distance $D^2(q, p)$ between the query point Q and a data point P is observed, which is stated in the following lemma.

Lemma 3.1 *Given point $P = \{p_j\}$ and $Q = \{q_j\}$, $j \in \{1, \cdots, N\}$, and projected point $P' = \{p_j\}$ and $Q' = \{q_j\}$, $j \in \{1, \cdots, n\}$, $n \leq N$, the Euclidean distance between P and Q is:*

$$D^2(P, Q) = \sum_{j=1}^{N} (p_j - q_j)^2. \tag{3.3}$$

An common approximate distance to $D(P, Q)$ is:

$$D''^2(P, Q) = D^2(P', Q') = \sum_{j=1}^{n} (p_j - q_j)^2. \tag{3.4}$$

An more accurate approximate distance is:

$$\tilde{D}^2(P,Q) = D''^2(P,Q) + \left(\sqrt{\sum_{j=n+1}^{N} p_j^2} - \sqrt{\sum_{j=n+1}^{N} q_j^2} \right)^2 . \tag{3.5}$$

Proof of Lemma 3.1

That $D''^2 \leq \tilde{D}^2 \leq D^2$ is proven at this point. Since $D''^2 \leq \tilde{D}^2$ is trivial, the following is to show that $\tilde{D}^2 \leq D^2$ holds.

$$\begin{aligned}
D^2(P,Q) &= D''^2(P,Q) + \sum_{j=n+1}^{N} (p_j - q_j)^2 \\
&= D''^2(P,Q) + \sum_{j=n+1}^{N} p_j^2 + \sum_{j=n+1}^{N} q_j^2 - 2 \cdot \sum_{j=n+1}^{N} p_j \cdot q_j \\
\tilde{D}^2(P,Q) &= D''^2(P,Q) + \sum_{j=n+1}^{N} p_j^2 + \sum_{j=n+1}^{N} q_j^2 - 2 \cdot \sqrt{\sum_{j=n+1}^{N} p_j^2} \cdot \sqrt{\sum_{j=n+1}^{N} q_j^2}
\end{aligned}$$

The first three items are the same, so only the last item needs to be compared. Let

$$E = p_j \cdot q_k \qquad F = p_k \cdot q_j$$

$$\begin{aligned}
A &= (\sum_{j=n+1}^{N} p_j \cdot q_j)^2 \\
&= \sum_{j=n+1}^{N} (p_j \cdot q_j)^2 + \sum_{j=n+1}^{N} \sum_{k=j+1}^{N} (2 \cdot p_j \cdot q_k \cdot p_k \cdot q_j) \\
&= \sum_{j=n+1}^{N} (p_j \cdot q_j)^2 + \sum_{j=n+1}^{N} \sum_{k=j+1}^{N} (2 \cdot E \cdot F)
\end{aligned}$$

$$\begin{aligned}
B &= \sum_{j=n+1}^{N} p_j^2 \cdot \sum_{j=n+1}^{N} q_j^2 \\
&= \sum_{j=n+1}^{N} (p_j \cdot q_j)^2 + \sum_{j=n+1}^{N} \sum_{k=j+1}^{N} (p_j^2 \cdot q_k^2 + p_k^2 \cdot q_j^2) \\
&= \sum_{j=n+1}^{N} (p_j \cdot q_j)^2 + \sum_{j=n+1}^{N} \sum_{k=j+1}^{N} (E^2 + F^2)
\end{aligned}$$

Given that $(E - F)^2 > 0$, it follows $A < B$ and $\tilde{D}^2 \leq D^2$.

End of Proof.

3.5 Performance Study

3.5.1 Experiment Setup

Three real-life datasets are utilized, they are: texture dataset with 55 dimensions (TXT55), color histogram with 64 dimensions (COLH64) Gabor dataset with 60 dimensions (GABOR60), and one synthetic dataset with 64 dimensions (SYN64). More details for each dataset are specified in Table 3.2. For preprocessing, the raw dataset of TXT55 and GABOR60 are studentized since the value of different dimensions vary considerably, while SYN64 and COLH64 are not studentized since the value of different dimensions are close to each other.

The experiments are run on a laptop with Intel Pentium M CPU 1.1GHz and 768MB RAM under Windows XP Professional.

3.5.2 Experimental Comparison of Index Structures

Three index structures are used in this study. They are hybrid trees[1], R*-trees[2] and SR-trees[3]. The hybrid tree and R*-tree are migrated from UNIX to Windows 2000.

Pages of the index are 8192 bytes. 1000 queries are randomly chosen without replacement from the original datasets. k-NN queries with $k = 20$ are issued to evaluate the performance for different indices. The performance metrics are index sizes, number of pages accessed, and elapsed time.

[1]Code at `http://www.ics.uci.edu/~kaushik/research/htree.html`
[2]Code received from C. Faloutsos and D. Chakrabarti at CMU
[3]Code at `http://research.nii.ac.jp/~katayama/homepage/research/srtree/`

Table 3.2 Characteristics of the Four Datasets used in Experiments; N: Number of Features, M: Number of Points

Name	N	M	Description/Source
SYN64	64	99,972	Synthetic dataset generated by the source code used in [14] with the same parameters.
COLH64	64	68,041	8×8 color histograms extracted from 68,041 color images obtained from `http://kdd.ics.uci.edu/databases/CorelFeatures`.
GABOR60	60	56,644	Gabor features extracted from Landsat MMS images from different parts of the country, obtained from V. Castelli.
TXT55	55	79,814	Gabor, spatial, and wavelet features from 400 photos, which also utilized in [3].

Comparison of the HS and RKV Algorithm Given an index structure, the search algorithm plays an important role in its performance. Both the HS and RKV algorithm are implemented on the SR-tree and R*-tree. Experiments on different datasets and varied dimensionality over a dataset show that the HS algorithm always visits fewer pages than the RKV algorithm. Table 3.3, 3.4 and 3.5 report the results. The different dimensionalities are obtained by keeping the first n dimensions, which is determined by using SVD and a given NMSE.

Index Size Index sizes are compared for the four datasets and nine additional datasets with variable number of dimensions that are generated from SYN64 by using the principal component analysis and keeping the most significant dimensions. Figure 3.4 shows that

Table 3.3 Comparison of the HS and RKV Algorithm Applied on R*-trees and SR-trees over Different Datasets

Page Accesses	SR-tree		R*-tree	
Dataset	*HS*	RKV	*HS*	RKV
SYN64	*363*	406	*7372*	7993
COLH64	*790*	1200	*5883*	7099
TXT55	*274*	746	*1429*	2533
GABOR60	*29*	58	*198*	783

Table 3.4 Comparison of the HS and RKV Algorithm Applied on R*-trees and SR-trees with Varying Dimensionality for SYN64

Page Accesses	Dim	1	2	4	5	7	14	33	45	60	64
SR-tree	HS	*4*	*6*	*14*	*21*	*30*	*54*	*143*	*213*	*338*	*363*
	RKV	3	5	14	21	30	53	151	220	354	406
R*-tree	HS	*2*	*4*	*11*	*19*	*47*	*197*	*2134*	*4680*	*6663*	*7372*
	RKV	2	4	11	30	54	217	2379	5108	7190	7993

R*-trees are much larger than the other two, while SR-trees and hybrid trees always have almost the same size. This is because R*-trees store duplicate coordinates in the leaf nodes.

Number of Pages Accessed The search performance of an index structure is more important than the size of the index. The average number of page accesses for a 20-NN query is evaluated in Table 3.6. It shows that the R*-tree accesses significantly more pages than the hybrid tree and the SR-tree, while the hybrid tree accesses a few more pages than the SR-tree. For SYN64, R*-trees access 32.7% of the index pages, while hybrid-trees and SR-trees access only 3.1% and 4.0%, respectively.

Table 3.5 Comparison of the HS and RKV Algorithm Applied on SR-trees with Varying Dimensionality for TXT55

Page Accesses	Dimensionality	5	10	20	30	40	50	55
SR-tree	HS	*19*	*46*	*84*	*145*	*198*	*271*	*274*
	RKV	24	94	150	364	550	772	746

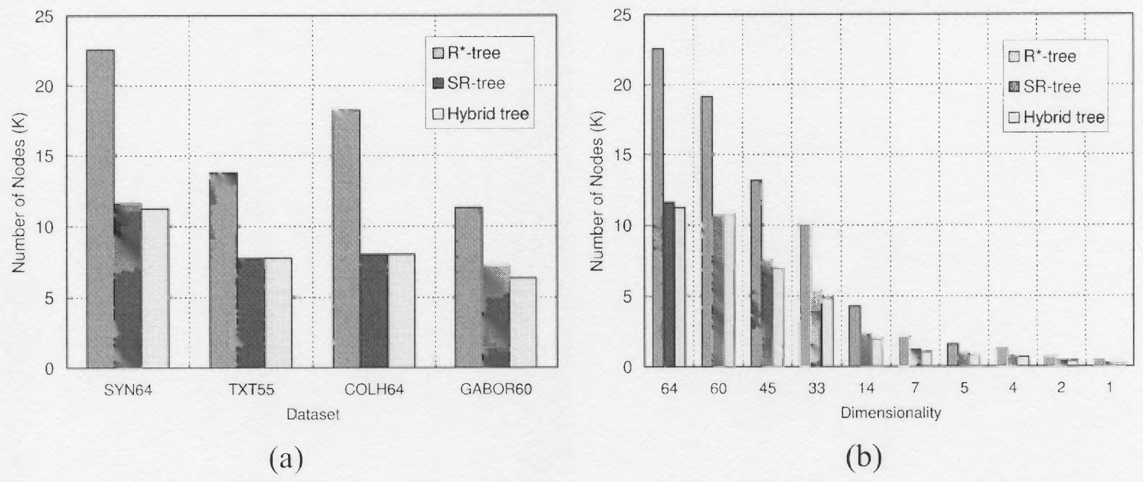


Figure 3.4 Index size comparison over (a) different datasets, (b) different dimensionality of SYN64.

For the scalability on dimensionality, Figure 3.5 shows that the R*-tree works well under low-to-medium dimensions (≤ 14 dimension) and has very poor performance as the dimensionality increases. The number of pages visited is in the thousands when dimensions ≥ 22, which can be seen from Figure 3.5(a). The performance of the R*-tree is always worse than the SR-tree and hybrid tree. The hybrid tree has a smaller index size than the SR-tree. It visits more pages than the SR-tree and the gap widens with the dimensionality increasing.

CPU Time In terms of the CPU time, experiments are performed both on different datasets as in Table 3.7 and different dimensions for one dataset as in Figure 3.6. In both cases, the

Table 3.6 Number of Page Accesses for Processing k-NN Queries

	SR-tree			Hybrid tree			R*-tree		
	Page Accesses	Total	Ratio	Page Accesses	Total	Ratio	Page Accesses	Total	Ratio
SYN64	363	11570	3.1%	448	11220	4.0%	7372	22542	32.7%
COLH64	790	8005	9.9%	793	7773	10.2%	5883	18232	32.3%
TXT55	274	7728	3.5%	533	8018	6.6%	1429	13761	10.4%
GABOR60	29	7223	0.4%	42	6363	0.7%	198	11288	1.8%

R*-tree runs slower than both the SR-tree and hybrid tree, while the SR-tree is consistently faster than the hybrid tree.

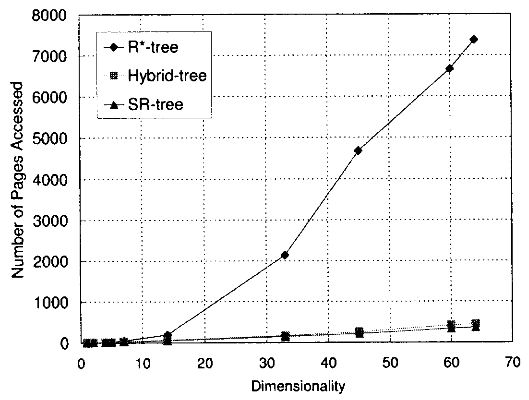
Table 3.7 Comparison of CPU Time of SR-tree, Hybrid tree and R*-tree on Four Datasets

CPU	*SR-tree*	Hybrid tree	R*-tree
SYN64	*0.0226*	0.106	0.7778
COLH64	*0.0518*	0.1716	1.91
TXT55	*0.0199*	0.1218	1.0516
GABOR60	*0.0018*	0.0384	0.336

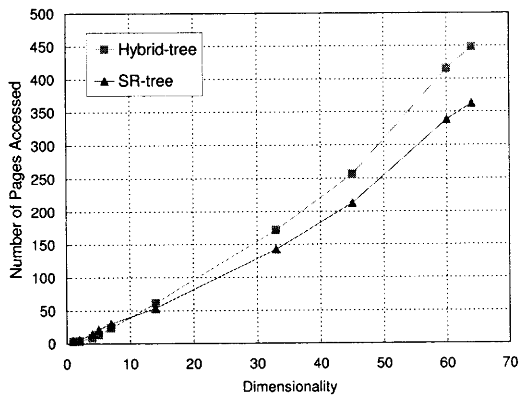
In conclusion, the SR-tree gives the best overall performance. For the following performance evaluation of CSVD, it will be used as the within-cluster index.

3.5.3 Performance Study of CSVD with Indexing

Experiments in [83, 87] show that the GM1 (Global Method 1) outperforms the GM2 (Global Method 2) and LM (Local Method) in terms of precision and CPU time. Thus CSVD-GM1 is used in this study.



(a)



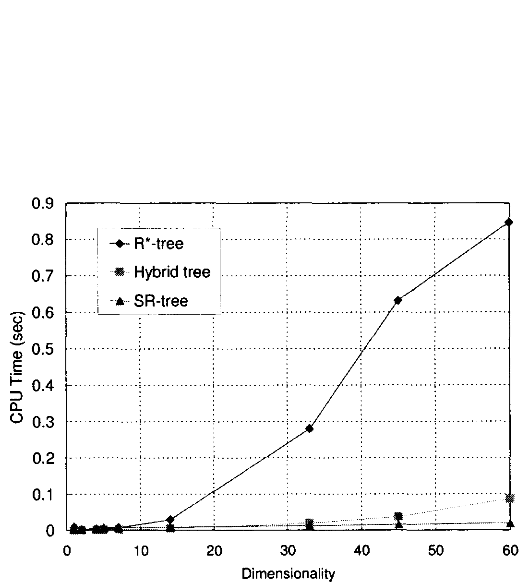
(b)

Figure 3.5 Comparison of R*-trees, SR-trees and hybrid trees on number of pages accessed using the HS algorithm over variable dimensionality on SYN64.

Number of Dimensions Retained Given an NMSE, the more clusters the dataset is partitioned into, the fewer the number of dimensions retained, which results in a higher data compression ratio. The larger the NMSE, the more the dimensionality reduction. Table 3.8 shows the average number of dimensions retained over varying NMSE and number of clusters on TXT55.

Table 3.8 Average Number of Dimensions per Point for TXT55

Dimensions	NMSE								
Clusters	0	0.01	0.05	0.1	0.15	0.2	0.3	0.4	0.5
1	55	42	28	22	18	16	12	9	6
2	55	38	24	18	15	13	9	7	5
4	55	36	22	16	12	10	6	4	3
8	55	36	22	15	12	10	6	4	3
16	55	34	21	15	11	9	6	4	3
32	55	32	20	14	11	8	6	4	2
64	55	30	18	13	10	8	5	3	2
128	55	29	17	12	10	8	5	4	2



(a)

Figure 3.6 Comparison of CPU time for R*-trees, SR-trees and hybrid trees on different dimensionality of SYN64.

Recall The recall drops with the NMSE increasing (the number of dimensions retained decreasing) as shown in Figure 3.7 and 3.8. Figure 3.7(b) shows that given the same NMSE, SVD (1 cluster) always produce lower recalls than CSVD, while the more clusters the dataset is partitioned, the higher recalls CSVD obtains. In terms of indexing, 1 and 5 clusters visit almost the same number of pages, while 16, 32, and 64 clusters visit more pages. The same applies to CPU time. Figure 3.7 shows that 16 is a better choice for the number of clusters when CSVD is applied to SYN64, since it visits less number of pages, resulting in less CPU time, while keeping a higher recall.

Figure 3.8 shows the results on TXT55. 1 cluster is omitted, since the compression ratio is worse compared to clustered cases. 16 or 32 is a good choice for balancing the processing cost and a higher recall.

Precision When the NMSE is large, the recall is very low. To achieve certain recall, more nearest neighbors (k^*) need to be asked. k^* can be calculated offline, and Table 3.9 gives the value of k^* to attain recall = 0.8 for 20-NN queries for TXT55. For a larger NMSE, k^* also becomes larger, and this may cause a slow response time. There is a tradeoff between the dimensionality reduction and the query cost for exact queries. A detailed study to find

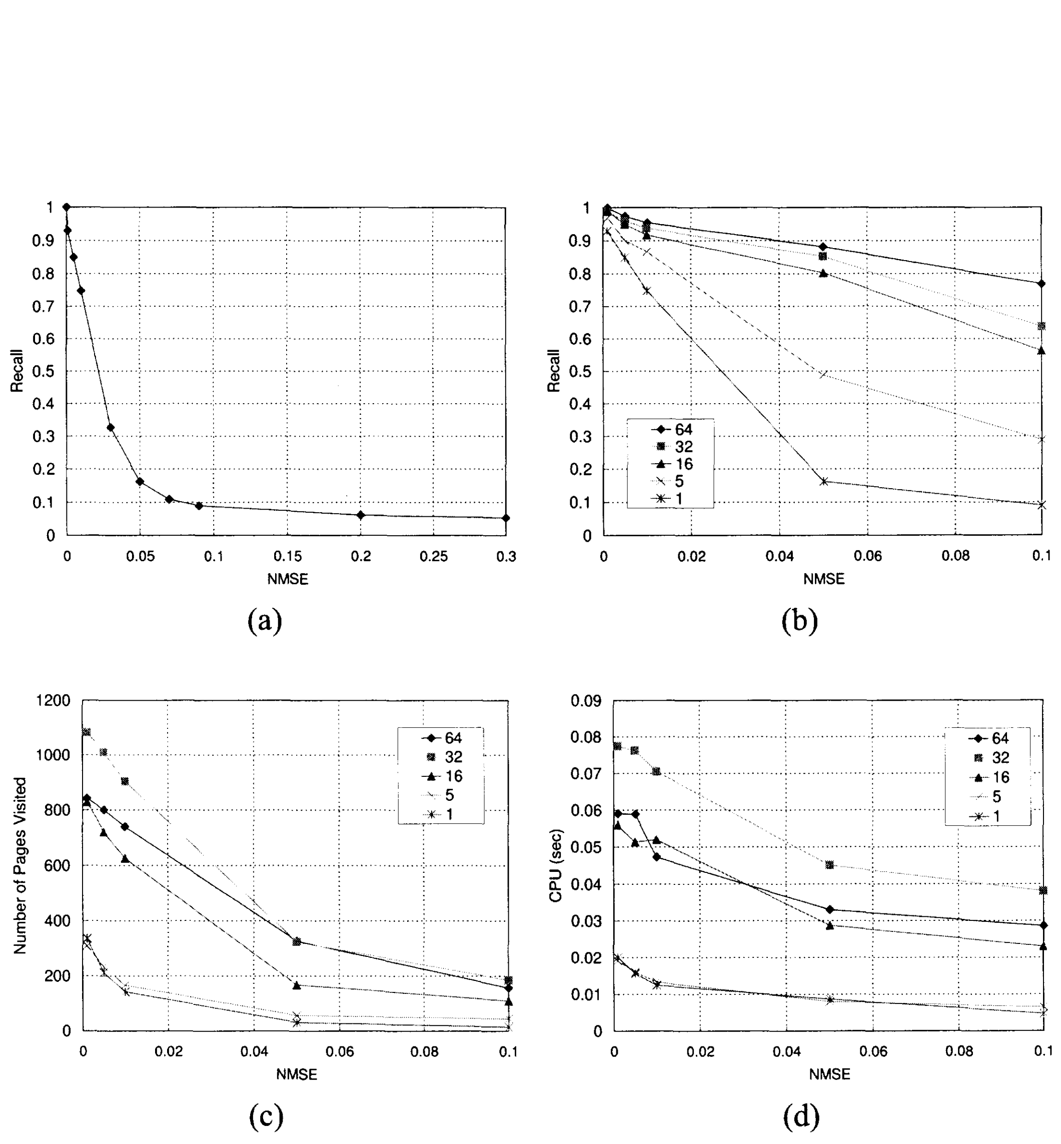


Figure 3.7 Performance comparison of different number of clusters versus NMSE for approximate 20-NNs on SYN64 using SR-trees. (a) Recall for 1 cluster. (b) Recall for 1, 5, 16, 32 and 64 clusters. (c) Number of pages visited. (d) CPU time.

the optimal value for NMSE is performed in [88]. In Figure 3.8, the precision goes higher with more clusters and has the highest precision for 128 clusters.

Number of Clusters Visited Figure 3.9 shows the effect of clustering on reducing the search space for nearest neighbor queries. Less than 3 out of 16 clusters and 6 out of 64 clusters need to be visited to execute approximate 20-NN queries for SYN64. This speeds up the query processing significantly, while a very high recall is achieved when the dataset

Table 3.9 The Value of k^* to Attain Recall = 0.8 for 20-NN Queries

k^* for recall=0.8	NMSE							
Number of Clusters	0.01	0.05	0.1	0.15	0.2	0.3	0.4	0.5
1	16	17	19	20	22	24	29	74
2	16	17	18	20	22	34	72	168
4	16	17	18	19	21	30	72	400
8	16	17	18	19	20	28	65	158
16	16	16	17	18	19	22	33	60
32	16	16	17	18	18	22	32	57
64	16	16	17	18	19	23	34	80
128	16	16	17	17	18	21	26	40

is partitioned to 64 clusters (Figure 3.7). For TXT55, the search space is reduced to half when the dataset is partitioned into 128 clusters.

Evaluation of Two Approximate Distances In this experiment, the effect of two approximate distances, $\tilde{D}(p, q)$ and $D(p', q)$, is quantified. In the context of exact nearest neighbor search [89] developed based on the CSVD, Table 3.10 gives the average number of points retrieved per query using both distances. The result shows that fewer points need to checked when using $\tilde{D}(p, q)$, and the difference is magnified as NMSE increases.

3.6 Conclusions

The fanout of an index structure is directly affected by the page size. With the dimensionality increasing, for a given page size, the fanout decreases since the size of a node entry is a monotonically increasing function of dimensionality. The reduction of the fanout may require more nodes to be accessed on queries and causes the increase of the query cost.

For the evaluated index structures, the hybrid tree has a smaller index size than the R*-tree, which is smaller than the SR-tree. The idea is to compare different space/data

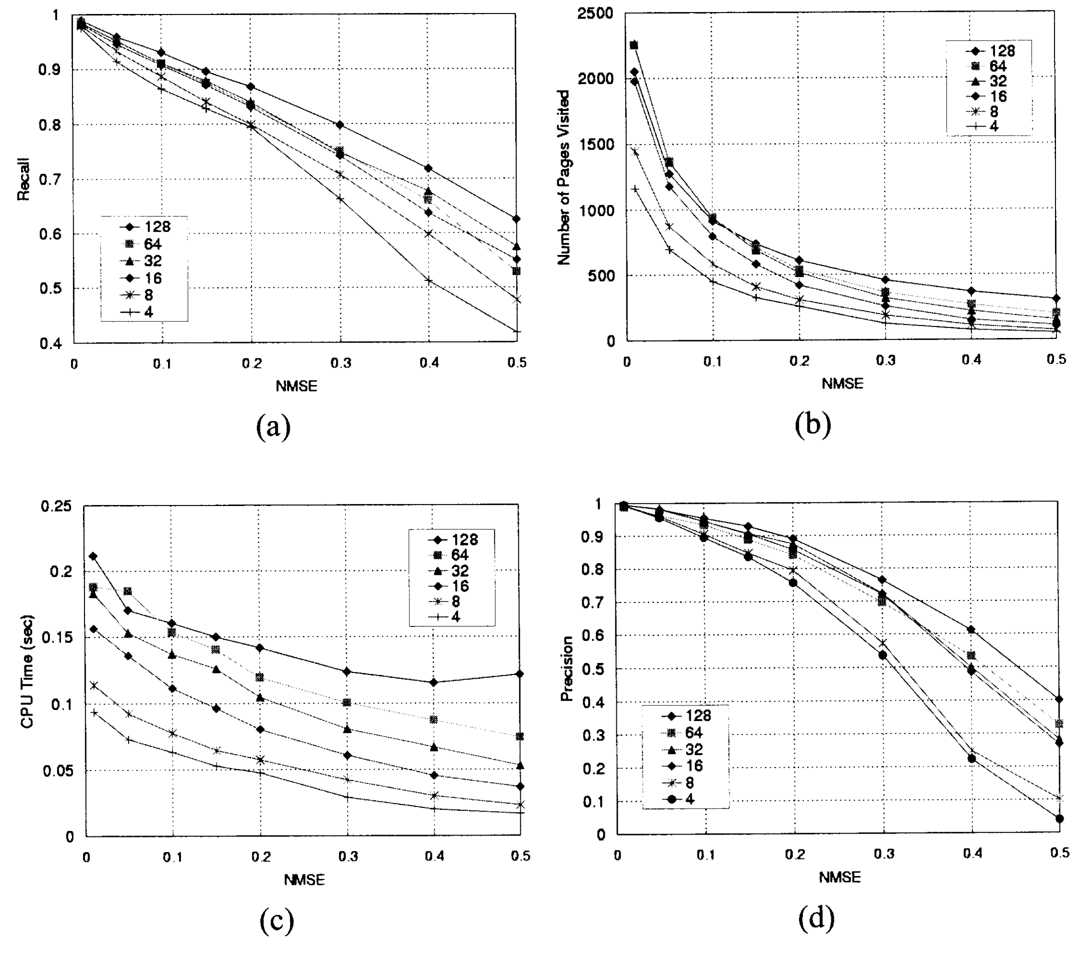


Figure 3.8 (a, b, c) Recall, number of pages visited, and CPU time for different number of clusters versus NMSE for approximate 20-NN search on TXT55 using SR-trees. (d) Precision to achieve recall = 0.8.

partitioning methods in high-dimensional space using the same search algorithm can be applied to other index structures.

The performance of CSVD is affected by the number of clusters visited and *Normalized Mean Square Error - NMSE.* CSVD is better than SVD in terms of recall and precision. The larger the NMSE, the fewer number of dimensions retained. The higher the degree of clustering for partitioning the dataset, the higher the recall and precision. However, more clusters require more disk accesses, resulting in more CPU time. Certain number of clusters exist to achieve a higher recall, while requiring a relatively lower query processing

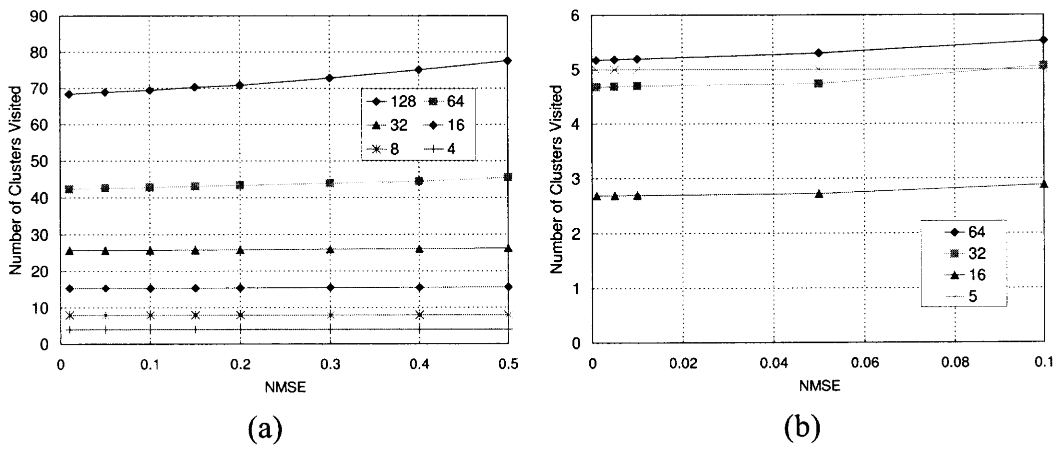


Figure 3.9 Number of clusters visited versus NMSE for approximate 20-NN search. (a) For TXT55. (b) For SYN64.

cost. In this study, the k-means clustering algorithm is used to partition the dataset, since spherical clusters are desirable for the nearest neighbor search algorithm based on the Euclidean distance. Very few clusters are checked for SYN64, while many more clusters are checked for TXT55, since SYN64 is constructed with spherical clusters, while TXT55 is a real dataset with the generated clusters highly overlapped. Clustering high-dimensional data is challenging problem.

Table 3.10 Average Number of Points Retrieved per Query for Gabor60

	NMSE											
NumClsts	0.01		0.03		0.1		0.15		0.2		0.3	
	$\tilde{D}$	D'	$\tilde{D}$	D'	$\tilde{D}$	D'	$\tilde{D}$	D'	$\tilde{D}$	D'	$\tilde{D}$	D'
1	*48*	48	*78*	81	*122*	137	*633*	1155	*2053*	4708	*8399*	16891
4	*171*	175	*328*	341	*613*	643	*1045*	1101	*1768*	2001	*5759*	9102
5	*117*	124	*201*	212	*308*	324	*493*	520	*794*	885	*2469*	3722

CHAPTER 4

PERSISTENT MAIN MEMORY INDEX

Similarity search implemented via k-Nearest-Neighbor (k-NN) queries is an extremely useful paradigm in *content based image retrieval* (CBIR), which is costly on multi-dimensional indices due to the curse of dimensionality. Most of multi-dimensional indices are inefficient in processing k-NN queries on high-dimensional data, since a significant fraction of index pages are accessed randomly, incurring a high positioning (seek plus rotational latency) time for each access. Moreover, the transfer rate is improving at a 40% annually, while the improvement in positioning time is only 8%. k-NN query processing can be improved by utilizing the double filtering effect of clustering and indexing on a persistent version of the main memory index. In this chapter, a specific instance, CIPOP, is presented. The Ordered-Partition tree (OP-tree) [16], a highly efficient index in processing k-NN queries, is used as the main memory index. The OP-tree is made persistent by writing it onto disk after serialization, i.e. arranging its nodes into contiguous memory locations, so that the high transfer rate of modern disk drives is exploited.

Experimental results to optimize OP-tree parameters are first reported. OP-trees and sequential scans with options for the Karhunen-Loève transform and Euclidean distance calculation are next compared. Comparisons against OMNI-based sequential scan are also reported. A clustered and persistent version of the OP-tree against a clustered version of the SR-tree and the VA-File method are then compared. It is observed that the OP-tree index outperforms the other two methods and that the improvement increases with the number of dimensions. Since the OP-tree is static, a semi-dynamic version is finally studied.

4.1 Introduction

Similarity search is a popular paradigm in *content based image retrieval - CBIR*, where images are represented by high-dimensional feature vectors based on color, texture, and shape [90]. k-nearest-neighbors - k-NN queries with the Euclidean distance function are commonly used for CBIR, although more general distance functions have been considered [2].

The k nearest neighbors can be determined by scanning the feature vectors of images sequentially, while updating a max-priority-queue with k elements. This may be costly if the number of images M and the number of dimensions of feature vectors (N) is high. The Vector Approximation - VA-File method [18], which involves a scan of quantized feature vectors is quite competitive in CPU cost, however, and is therefore used in the comparison.

A variety of methods have been proposed to speed up k-NN queries: clustering, indexing, clustering plus indexing, and dimensionality reduction, e.g., via Karhunen-Loève (K-L) transform. By partitioning a large dataset into clusters, only a few clusters closest to the query point need to be visited. Each cluster can be represented by a sequential or indexed structure. A side benefit of clustering is that the smaller dataset or index can be held in main memory.

The processing cost of k-NN queries is expected to be lowered considerably if the feature vectors are indexed by a multi-dimensional index, such as the R-tree [2, 11]. The indices can be categorized into *main memory resident* and *disk resident* indices [11]. The former aims at reducing the CPU time, but are restricted by the size of the main memory and the fact that such indices are volatile. Memory size is not a problem for paged disk resident indices. CPU time is the main performance metric for main memory indices, while for disk resident indices it is the number of disk I/Os.

Indexing has a similar effect to clustering, i.e. only the minimum bounding rectangles (MBRs) in R-trees [46] and hyperspheres in SS-trees [49], which overlap the search hypersphere for k-NN queries, are visited. The hypersphere is defined as the query point

Q as the centroid and the distance to the k^{th} nearest neighbor from Q as the radius. A comparison of the performance of R-trees, hybrid trees [14], and SR-trees (spherical and rectangular trees) [17] appears in Chapter 3, where it is shown that SR-trees outperform the other two index structures from the viewpoint of the number of page accesses in processing k-NN queries. Clustering plus indexing schemes take advantage of the dual filtering effect of both clustering and indexing to reduce the search space even further.

k-NN queries can be accelerated by applying the query to a dimensionality reduced dataset. This can yield approximate results for applications which can tolerate some degree of error or exact results by using postprocessing steps [89]. Several approximate methods have been proposed, such as CSVD [3], LDR [14] and MMDR [23].

Multi-dimensional indices are designed with the efficiency of disk access in mind. Each index node corresponds to a disk page, the index is usually height balanced, and a high fanout is used to minimize the depth of the tree. In processing a k-NN query, pages on the search path are loaded into main memory one by one via random disk accesses. Such disk accesses are slow and are improving at an intangible rate due to the mechanical nature of the disk. Most multi-dimensional index structures work well in low to medium dimensional spaces, but the fraction of index pages touched by k-NN queries grows quickly with the number of dimensions due to the dimensionality curse [68]. Too many random disk accesses degrades the performance of disk resident index dramatically.

Many studies compare disk resident index structures by counting the number of (random) disk I/Os (a small fraction of index pages are cached in main memory). Such a view made sense in the mid-1980s when a typical 1 MIPS machine would have 64-128KB of RAM, but not anymore [91]. As DRAM is becoming cheaper, DRAM-based main memories are becoming larger, so that a seemingly promising way is to keep the index in main memory. However, the sizes of the datasets for similarity search applications are potentially increasing at a faster rate than DRAM, so that the main memory may not be large enough to hold the whole index. Secondly the memory is volatile, i.e., the index has to be rebuilt each time it is used. Main memory indices can be made persistent, e.g., e.g., K-D-B versus k-d trees [11]. The solution chosen here is to partition the dataset into several clusters such that the index of each cluster can be held in main memory.

Data can be accessed more efficiently via sequential accesses of large disk files, rather than random accesses of disk pages. Sequential access time is determined by the disk transfer rate, which has been increasing by 40% per year. Random access time to small index pages is mainly positioning time, which has been decreasing at a rate less than 10% per year. This trend makes sequential disk accesses increasingly desirable compared to random accesses [92, 15], which are associated with popular disk resident indexing structures.

The Maxtor DiamondMax Plus 9 160G hard disk drives has 7200 RPM (rotations per minute) or $T_{rot} = 8.33$ ms per rotation and average seek time $T_{seek} = 9.3$ ms. There are $S_I = 610$ (resp. $S_O = 1102$) 512 byte sectors on inner (resp. outer) tracks and the mean number of sectors is $S_{mean} \approx 2S_O/3 + S_I/3 \approx 938$. The mean transfer rate is $(938 \times 512 \times 10^{-6})/(60/7200) \approx 57.6$ Megabytes/second, so that $T_{page\_xfer} = 8.192/57.6 = 0.142$ ms. The number of pages that can be accessed sequentially during a single random disk access is given by: $n_{seq'l} \approx (T_{seek} + T_{latency} + T_{page\_xfer})/T_{page\_xfer}$. Note that T_{seek} is incurred only once for sequential accesses, while it is incurred for every random access. For the disk under consideration: $\bar{n}_{seq'l} \approx 96$ pages, which means instead of accessing $\bar{n}_{random}$ 8 KB pages randomly, a $768\bar{n}_{random}$ KB dataset can be loaded from disk sequentially.

To reduce the random disk accesses and take advantage of the efficient processing of main memory indices, a general framework, Clustering and Indexing using Persistent Main Memory indices - CIPMM, is proposed to accelerate k-NN processing of high- dimensional datasets with a large number of points. The framework first partitions the dataset into clusters of manageable size and then builds a main memory index for each cluster, for which positioning time is incurred only once. The index is written out to disk as a nonpaged BLOB (binary large object). The index can be reloaded quickly on demand. Clustering is used to reduce the size of the BLOBs, so that they fit in main memory. Since OP-trees and SR-trees are efficient in processing nearest neighbor search, the proposed method is evaluated by indexing using the OP-tree (CIPOP) and compared with indexing using the SR-tree (CISR). The results show that the proposed CIPOP outperforms CISR, and the higher the dimensionality, the better the performance gains. Since the OP-tree is static, a semi-dynamic version is also studied.

Two serialization methods for the OP-tree are proposed: a two-phase method and a one-phase method. The two-phase serialization builds the index using dynamic storage allocation as an ordinary linked tree structure, arranges the nodes into contiguous memory locations, and then writes it to disk. The drawback is that the index has to be reserialized and rewritten when new data is inserted. The one-phase method is flexible in that insertions of new points requires only the rewriting of the modified parts of the index file.

The chapter is organized as follows. Section 4.2 surveys related work in this area. Section 4.3 describes the structure and k-NN search algorithm of the OP-tree. The performance of the OP-tree is studied by varying the parameters, the split factor and the leaf node capacity, and compared with the OMNI sequential scan [58] and sequential scans with the option of KL transformation and shortcut method to compute the Euclidean distance. The results show that the OP-tree outperforms all the considered sequential scan methods. Section 4.4 describes the CIPOP, the two-phase serialization method, and its k-NN processing steps. Experiments show that CIPOP outperforms CISR. Section 4.5 describes the insertion methods and the one-phase serialization method. Properties of the OP-tree on inserting variable fraction of points using partial or full KL-Transform are studied. One synthetic and three real-life datasets are used in this study. Conclusions and future work are given in Section 4.6.

4.2 Related Work

Indexing structures, which have been proposed to cope with the shortcomings of indices in the R-tree family, are first discussed. Dimensionality reduction via the Karhunen-Loève (K-L) transform is then addressed.

4.2.1 Indexing Structures

There are many studies of the efficiency of k-NN queries which belong to the R-tree family. According to [2] R-trees function successfully for 20-30 dimensions, after which the dimensionality curse results in accesses to a large fraction of index pages, which is tantamount to a sequential search of the dataset. Some researchers have realized that index structures will benefit from accessing large chunks of data via sequential disk accesses. Examples are the X-tree [32], the DABS-tree[68], and the Clindex [93].

The hierarchical organization is an efficient organization for low dimensionality, since there is little overlap between directory rectangles [32]. However, the linear organization is more efficient for very high dimensionality, since most of the directory has to be searched due to the high overlap. In this case a linearly organized directory needs less space and can be read from disk much faster than a page-by-page reading of the directory. The X-tree is partially hierarchical and partially linear. Data producing high overlap is organized linearly, while data producing low overlap is organized hierarchically. The linear organization reduces the number of random disk accesses.

A cost model is used to identify the reason why sequential scan outperforms most index structures [68]. The conclusion is that indices access data in small portions. The DABS-tree, a linear single-level directory, adjusts the block size dynamically. Each directory entry consists of the minimum bounding rectangle of the page region, the number of entries currently stored in the page and the reference to the page. The size of a page grows or shrinks on demand, which is optimally determined during update operations according to a query cost model. Pages have 100% utilization. A k-d tree is used to guarantee overlap-free page regions. Given a query, the distances between the query and each entry in the index are sorted in nonincreasing order, after which qualifying pages are loaded and processed in that order.

The Clindex[93] combines clustering and indexing for approximate similarity search. A large dataset is partitioned into small clusters first, and then a mapping table is built for indexing the clusters. Each cluster is stored on disk sequentially as a separate file, so that it can be retrieved with one access. The distances from each object in that cluster to the query object are calculated, and the most similar objects are returned. Once the page or the cluster are loaded in one disk IO, both the DABS-Tree[68] and Clindex[93] sequentially scan all the data points in that page or cluster, and find the nearest neighbors.

4.2.2 Karhunen-Loève Transform

Singular value decomposition - SVD and *Principal Component Analysis - PCA* are two methods that lead to a *Karhunen-Loève (K-L)* transform of the original dataset, after which the dataset is amenable to optimal dimensionality reduction. Given an $M \times N$ matrix X for M images with N features, PCA computes the covariance matrix $C = X^t X/M$, which is then decomposed as $C = V \Lambda V^t$. The eigenvectors of matrix V define the principal components. Λ is a diagonal matrix of eigenvalues: $(\lambda_1, \lambda_2, \ldots, \lambda_N)$, which without loss of generality are assumed to be in nonincreasing order. The KL transformation yields $Y = XV$, where the coordinates of Y are aligned with the principal components of X. The *normalized mean square error - NMSE* with n retained dimensions: $NMSE = \sum_{i=n+1}^{N} \lambda_i / \sum_{i=1}^{N} \lambda_i$ is minimized for a given n, which is optimal in terms of dimensionality reduction. SVD decomposes X as $X = USV^T$, where V is the eigenmatrix and S is a diagonal matrix of singular values with $\lambda_n = s_n^2/M,\ 1 \leq n \leq N$ [3].

Clustering combined with SVD or PCA first clusters the feature vectors constituting X and then applies SVD or PCA to individual clusters. The intuition behind this methods is that datasets tend to be composed of heterogeneous points and that PCA yields better results when applied to homogeneous data. It has been shown experimentally that given a target NMSE, a global method results in more dimensionality reduction than a local method [87]. The global method is used in this study to obtain different number of dimensions for a dataset.

4.3 The OP-tree

The OP-tree [16] is a k-d tree like balanced hierarchical index structure, which recursively partitions the points of a dataset into a fixed number of regions according to a prespecified split factor along consecutive dimensions until the leaf capacity c is not exceed. The authors of [16] proved that the ordered partition algorithm can find k nearest neighbors in a constant expected time. Simulations show that it is distribution free and only 4.6 distance calculations, on the average, were required to find a nearest neighbor among 10000 samples drawn from a bivariate normal distribution.

The OP-tree treats the features asymmetrically, using the features with the highest variance in partitioning the data first. A good ordering of the features will result in a more efficient search, since the features with the highest variance offer the largest contribution to the expected squared Euclidean distance [3]. The dimensions of the dataset can be ordered using PCA and the dataset can be transformed into uncorrelated coordinates with corresponding eigenvalues in nonincreasing order.

4.3.1 Number of Nodes

Each leaf node holds only one point and the number of level in the tree (l) is equal to the number of dimensions in [16]. For an $M \times N$ dataset the per-dimension splits are given as: $\vec{S} = (s_1, s_2, \ldots, s_{N-1})$. Splitting over the first $N-1$ dimensions yields a tree with T nodes (leaf and non-leaf nodes):

$$T = \sum_{i=1}^{N-1} \prod_{j=1}^{i} s_j + M + 1,$$

where each term corresponds to the number of internal nodes (except the root), leaf nodes and the root, respectively.

Since high-dimensional datasets (say N) are dealt, even a split factor of two results in 2^N leaf nodes, which is usually much larger than the number of points in the dataset (M). So that even if the leaf-node capacity is rather small, the partitioning need to be carried out is limited.

For small N and large M, the OP-tree can be extended to have a parameter: the leaf capacity c. The dimensions are reused in a round-robin manner. Two parameters are used for constructing the OP-tree: c and $\vec{S}$ When all splits are s-ways, i.e., $s_1 = s_2 = \ldots = s_{N-1} = s_N = s$, the number of level of the OP-tree (l) is the smallest integer that satisfies $c \times s^{l-1} < M \leq c \times s^l$, which leads to:

$$l = \lceil log_s(M/c) \rceil. \tag{4.1}$$

The total number of nodes is given by:

$$s^{l+1} - 1 = s^{\lceil log_s(M/c) \rceil + 1} - 1 \tag{4.2}$$

The selection of $\vec{S}$ and c affects the performance of the index structure, as shown in Section 4.3.4.

The partition of the OP-tree for a hypothetical dataset is shown in Figure 4.1 with $s_1 = 4, s_2 = 3$. The data is partitioned 4-ways along the first dimension and 3-ways along the second. Each leaf node can hold at most 3 points. Assigning larger s_i for dimension i with larger variance seems to have its advantages. But in practice, it is hard to find the number of splits along each dimension.

4.3.2 Data Structure

The OP-tree can be implemented either by a linked list (Figure 4.2) or by an array representation to keep track of the splits along each dimension. Using a linked list representation, each internal node is a 5-tuple: {*lower*, *upper*, *child*, *left*, *right*} and each leaf node is a

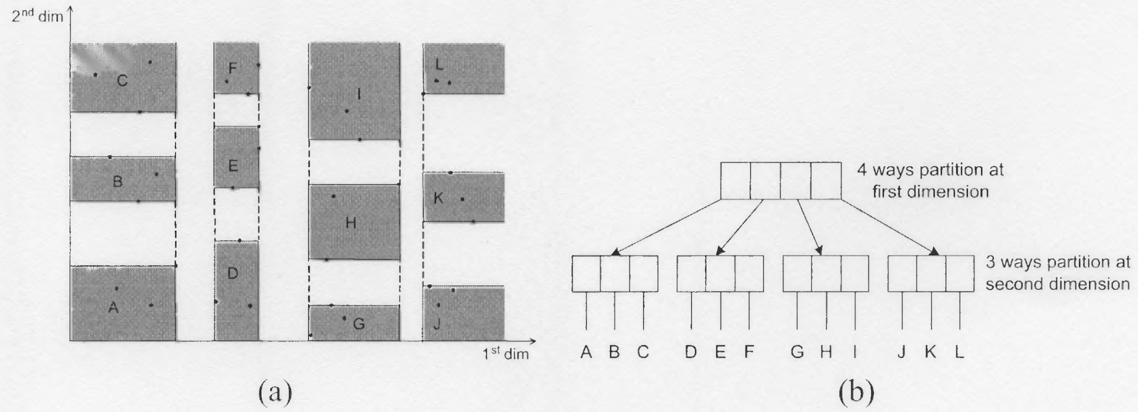

2nd dim
1st dim
C
B
A
F
E
D
I
H
G
L
K
J
4 ways partition at first dimension
3 ways partition at second dimension
A B C D E F G H I J K L
(a)
(b)

Figure 4.1 (a) The partition for a two-dimensional OP-tree. (b) The corresponding hierarchical structure.

6-tuple: {*lower*, *upper*, *left*, *right*, *ids*, *points*}. *Lower* and *upper* are the lower and upper values for bounding the region, *left* and *right* are pointers pointing to the left and right siblings, *child* is a pointer pointing to the leftmost child for the internal node, *ids* is a pointer to the list of ids corresponding to the feature vectors and *points* is a pointer to the list of feature vectors for the leaf node. While using an array representation, each node (including internal and leaf node) is a 3-tuple: {*child, lower, upper*}.

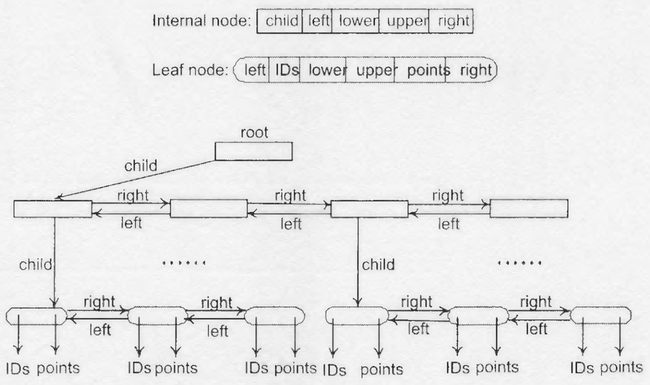

Internal node: child left lower upper right
Leaf node: left IDs lower upper points right
root
child
right
left
child
......
IDs points

Figure 4.2 Index structure for the OP-tree. Child: pointer to the left most child. Left (right): pointer to left (right) sibling. IDs: pointer to a list of point IDs. Points: pointer to data points. Lower (upper): lower (upper) bound values.

Both representations are implemented. The linked list has two versions: one with the system's memory management, the other with a self-implemented page manager, which allocates space in a contiguous manner. Experimentation shows that the array representation runs slightly faster than the linked list representation, especially when the number of splits is high (say more than 20). This is because the array is more cache friendly by being amenable to cache prefetching. The linked list representation utilizes more space than the array representation, but it provides the flexibility of varying split factors. The linked list representation is used in the following study.

4.3.3 k-NN Search

The pseudo-code described in Algorithm 3 is an improved version of the algorithm given in [16]. It starts with the region to which Q belongs or Q has the shortest MINDIST. Branches are pruned by only calculating partial distance up to that dimension. Line 1-2 search the leaf nodes. The actual points are searched and the current k^{th} nearest neighbor distance from the query is updated. Line 3-13 search the internal nodes. Line 4-6 handle the case where the number of levels of the tree is greater than the number of dimensions. Line 9 calculates the distance from the query point to the current node which actually is a hyperrectangle. With the Euclidean distance, the distance calculation for each node only requires at most two floating point operations (one multiplication and one addition) and two tests (see Equation 4.3). Line 12-13 recursively search each candidate node.

$$\begin{aligned} D_l^2 &= D_{l-1}^2 + D^2(q_l, a_l, b_l). \\ D^2(q_l, a_l, b_l) &= \begin{cases} (a_l - q_l)^2 & q_l < a_l \\ 0 & a_l \le q_l \le b_l \\ (q_l - b_l)^2 & b_l \le q_l \end{cases} . \end{aligned} \tag{4.3}$$

A max-priority-queue with size k is used to implement the k-NN search. The key value is the distance between the query point to the visited points. Each time when a point closer to the query point is found, the top element is removed, and the new element constructed from the better point is inserted.

Algorithm 3 knnSearch(Q, r, d_p, l)

```
1: if ($r$ is a leaf node) then
2:     leafSearch($r$);
3: else                                                  //$r$ is an internal node
4:     if ($l$ == N) then
5:         $l$ = 0;
6:     if ($l$ == 0) then                     //set the distance to current node $r$
7:         $d$ = 0;
8:     else                      //calculate $d$ using the distance to parent $d_p$
9:         $d$ = nodeDistance($Q$, $r$, $d_p$, $l$);
10:    find the closest child $c$ of $r$ to $Q$;
11:    while ($c$ exists && isCandidate($Q$, $c$, $d$, $l$+1)) do
12:        knnSearch($Q$, $c$, $d$, $l$+1);
13:        find the next closest child $c$ of $r$ to $Q$;
```

4.3.4 Experimental Evaluation

The experiments are carried out for k-NN queries with $k = 20$. One thousand queries are selected by simple random sampling without replacement (SRSWOR) from each dataset. All the experiments are conducted on a Dell Precision 330 with Intel Pentium 4, 1700 MHz CPU and 512 MB RAM, running Windows 2000 Professional.

The OP-tree is studied extensively in this section. The array and linked list representations of the OP-tree are first compared. Rules-of-thumb are developed for selecting the split factor and the leaf node capacity ensure robust performance. The OP-tree is next compared with sequential scans with the option of KL-transform or shortcut Euclidean distance calculation method and it shows that the OP-tree outperforms the sequential scan methods. The performance of the OP-tree against the SR-tree is then compared from the viewpoint of CPU time and the OP-tree is a winner in this case too. At the end, the OP-tree is compared with the OMNI-family.

Eigenvalues Figure 4.3 illustrates eigenvalues for the four datasets in Table 3.2. For SYN64 and GABOR60, most eigenvalues are very small and the first keeps most of the variance, while for TXT55 and COLH64, the eigenvalues decrease smoothly.

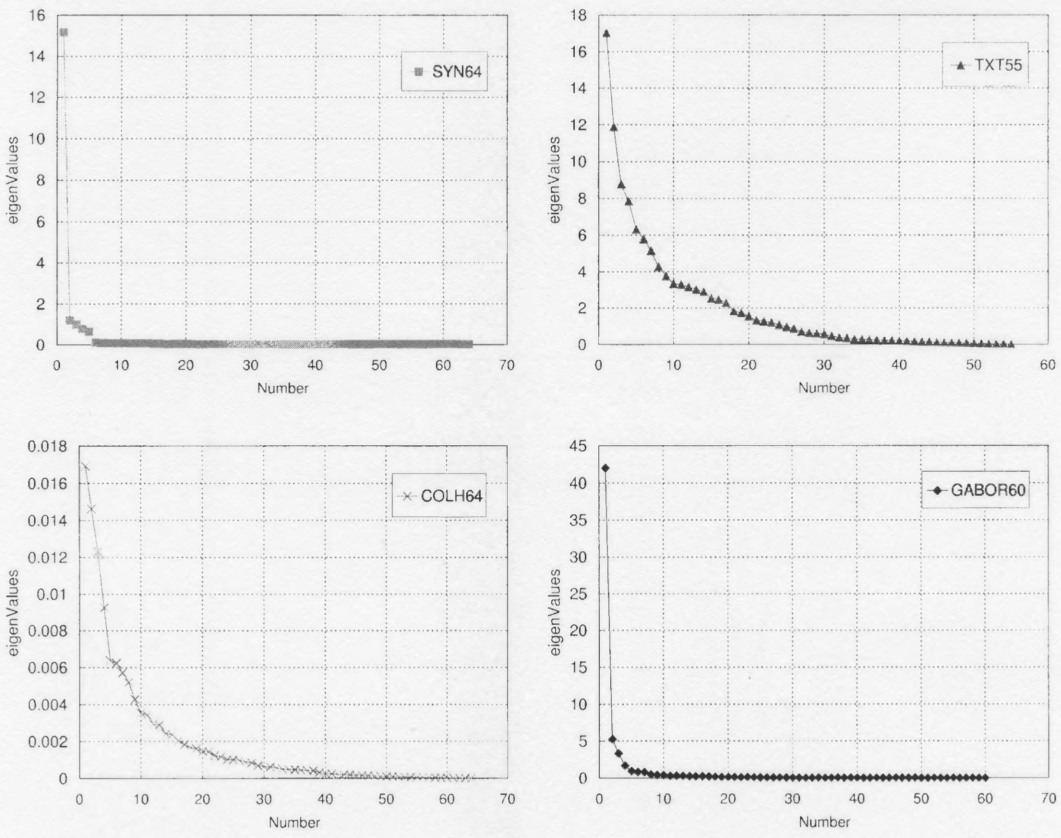


Figure 4.3 Eigenvalues for the four datasets.

Array versus linked list Both the array and linked list representations are implemented. The linked list has two versions: one with the system's memory management, one with the self-implemented page manager (PM). Figure 4.4 gives the experimental results on four datasets for k-NN query with $k = 20$. The results show that the array implementation takes less CPU time than the linked list, and the linked list managed by the page manager is slower than the other two methods. This is reasonable, since pointers have to be traced to locate different nodes by using linked list, while each pointer, which is the page base address and offset, has to be resolved.

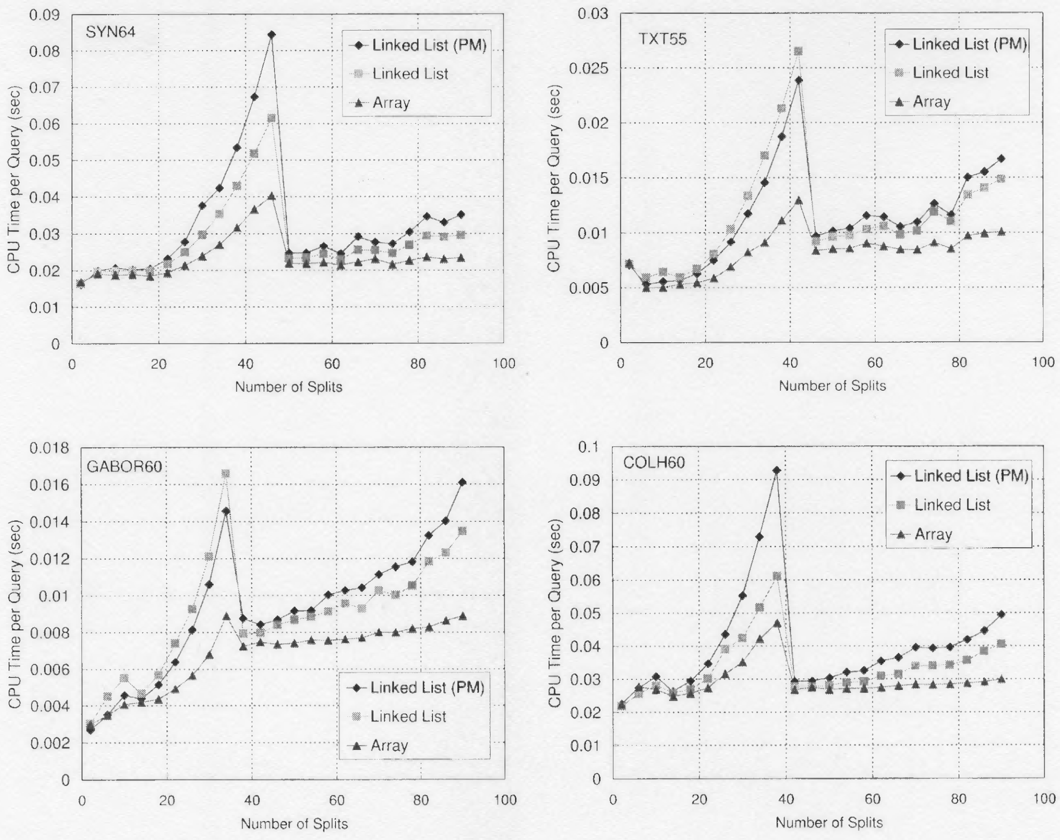


Figure 4.4 Comparison of linked list and array implementation with $c = 40$ for all cases.

Selecting Parameters of the OP-Tree The performance of the OP-tree in processing k-NN queries is affected by the split factor (s) and the node capacity (c). To select appropriate parameters for the OP-tree, the average CPU time (in milliseconds) over one thousand 20-NN queries on each dataset are measured as plotted in Figure 4.5 and Figure 4.6.

In Figure 4.5, the number of tree levels, which is 12 when the split factor is two, decreases as the number of splits increases. The CPU time increases initially, but there is a sudden drop in CPU time, when the number of tree levels drops to two. This can be explained as follows: when the tree level is high, branches can be pruned based on more

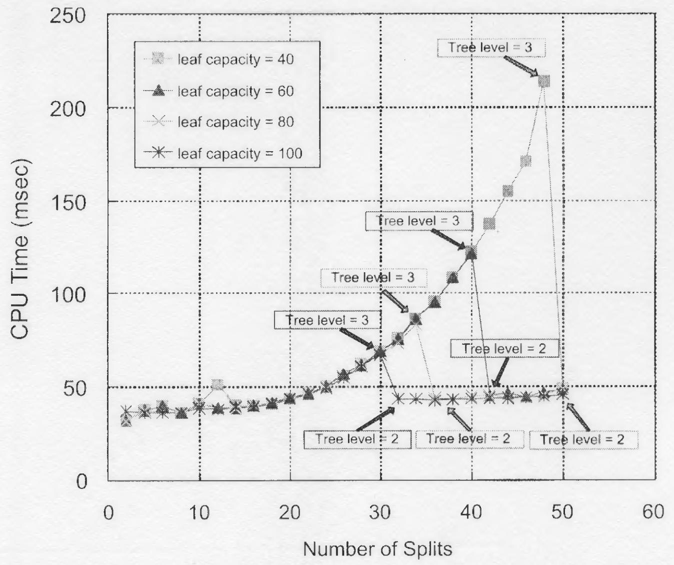


Figure 4.5 Average CPU time for processing 1000 20-NN queries on SYN64 versus number of splits as leaf node capacity is varied.

dimensions, while when the tree level is low, i.e., two, the OP-tree benefits from sequential scan. In conclusion, a low fanout seems to be a sure bet regardless of the capacity of the leaf node. A five-way split is used in [3]. The parameters optimized the CPU time are selected as in Table 4.2, which will be used in the following experiments.

Some datasets have a significant fraction of the variance in first few dimensions. In the case of SYN64 the first three eigenvalues are 15.1533, 1.1858, 0.978623. Assigning a higher split factor to the first dimension is intuitively appealing. Table 4.1 presents experimental results via varying the split factor for the first dimension, while maintaining all other split factors at two. There is a small reduction in the number of points accessed for $s_1 = 4$ over $s_1 = 2$, while this number increases beyond $s_1 > 4$. A split factor of two is desirable for all dimensions, unless λ_1 is very large.

Effect of leaf capacity on CPU time of k-NN search Figure 4.7 shows that the CPU time is slightly higher for larger k and has little sensitivity on the leaf capacity as long as it is not too small. The former is due to the fact that the search radius and consequently the hypersphere for k-NN queries will be larger during the search process and more leaf

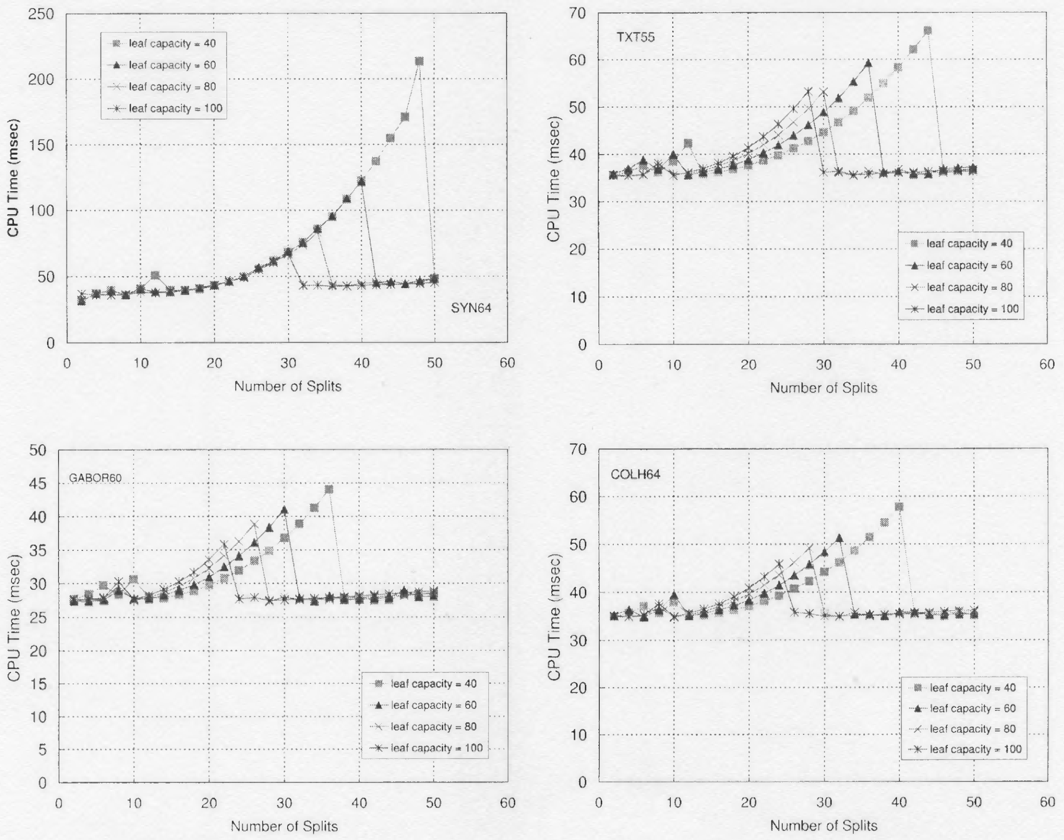


Figure 4.6 CPU time for processing 1000 20-NN queries versus number of splits with respect to different leaf capacities.

Table 4.1 Varying the Split Factor for the First Few Dimensions While All Others are Set to Two

s_1	$s_2 \ldots s_4$	$s_5 \ldots s_{64}$	$avg\ \#\ of\ points\ visited$
20	2	2	39165
20	10	2	35523
16	2	2	36879
10	4	2	39381
8	2	2	33757
4	2	2	30403
4	4	2	30403
2	2	2	32184

Table 4.2 Parameters for the OP-tree for Different Datasets

Dataset	SYN64	COLH64	GABOR60	TXT55
Number of splits (fanout)	2	2	2	8
Leaf node capacity	40	40	60	40

nodes will be visited and more points will be processed. Since a max-priority-queue is used to hold the results of k-NN queries, the maintenance of k-NN points has little effect on performance.

In the latter case, the leaf capacity c determines l as given by Equation 4.1 is shown in Table 4.3. When l is high, such as 16 for SYN64 while c is 1, the CPU time is high due to large number of internal nodes need to be examined. For higher values of c, the value of l ranges from 9 - 13 and the CPU time does not vary significantly. The number of internal nodes need to be checked is reduced dramatically (about three times from $l = 17$ to 15).

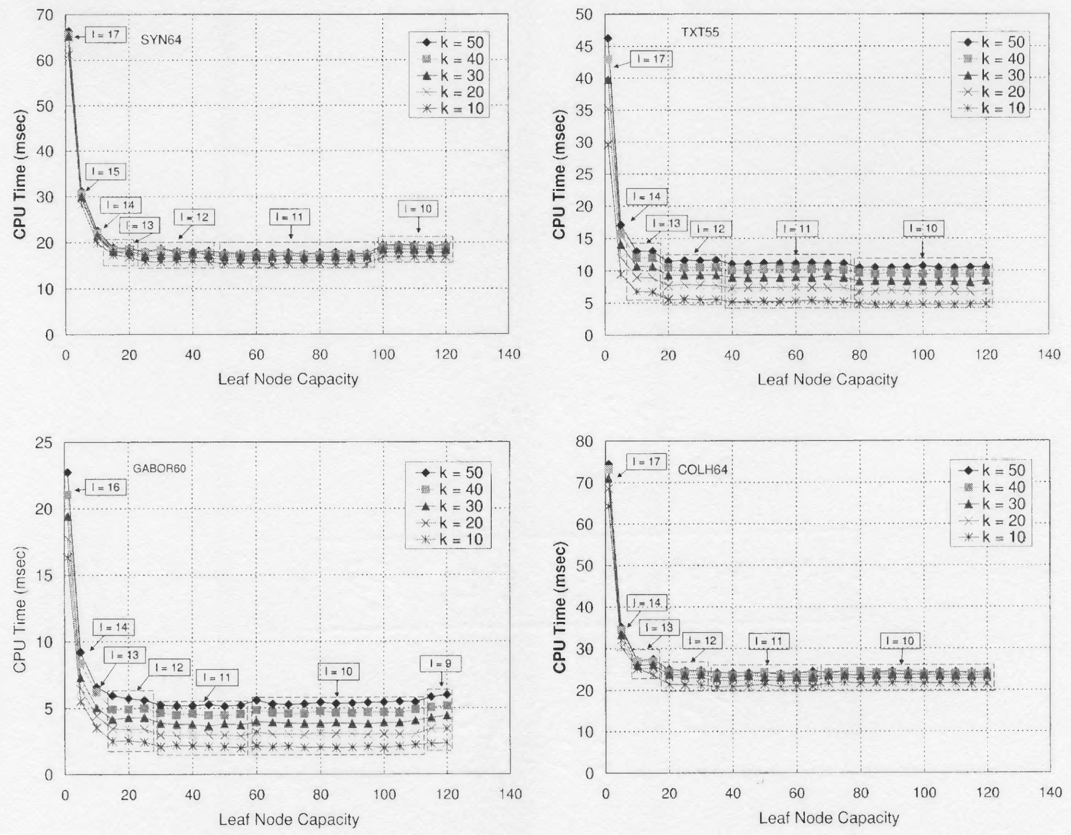


Figure 4.7 Effect of leaf node capacity on CPU time of k-NN search with split factor of 2 and $k = 20$.

OP-Trees versus Sequential Scan Methods Before the performance of the OP-tree with other indexing structures is compared, a question remaining to be answered is whether the OP-trees indeed outperform a sequential scan of the original dataset (X) or the dataset transformed into its principal components $Y = XV$ (see Section 4.2.2). All datasets are assumed to be main memory resident, since they will incur about the same loading time if they were originally disk resident. The effect of a shortcut method for Euclidean distance calculation is also investigated. In summary, there are three issues under consideration: OP-trees versus sequential scan, X versus Y matrices, and a standard versus a shortcut method for Euclidean distance calculation.

The CPU time versus the number of nearest neighbors to be found (k) is plotted in Figure 4.8 (a). k-NN processing is carried out via a sequential scan of the X and Y

Table 4.3 Tree Levels With 2 Splits on Each Dimension.

TreeLevels	Leaf Capacity																								
Dataset	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100	105	110	115	120
SYN64	16	14	13	12	12	11	11	11	11	11	10	10	10	10	10	10	10	10	10	10	9	9	9	9	9
TXT55	16	13	12	12	11	11	11	11	10	10	10	10	10	10	10	10	9	9	9	9	9	9	9	9	9
GABOR60	15	13	12	11	11	11	10	10	10	10	10	10	9	9	9	9	9	9	9	9	9	9	9	8	8
COLH64	16	13	12	12	11	11	11	10	10	10	10	10	10	10	9	9	9	9	9	9	9	9	9	9	9

datasets for SYN64. The standard and shortcut methods to compute Euclidean distances are considered. The following conclusions can be drawn:

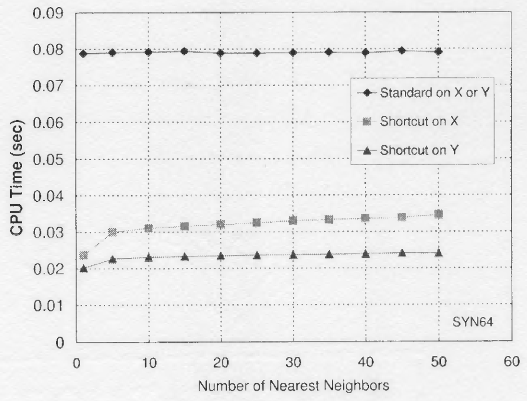
1. The shortcut method is 2 to 3 times faster than the standard method.

2. The shortcut method when applied to the X matrix requires 50% more CPU time than the Y matrix, which is due to the fact that the columns of the Y matrix are ordered according to their variance. This allows the decision to exclude a point to be made after considering a few dimensions.

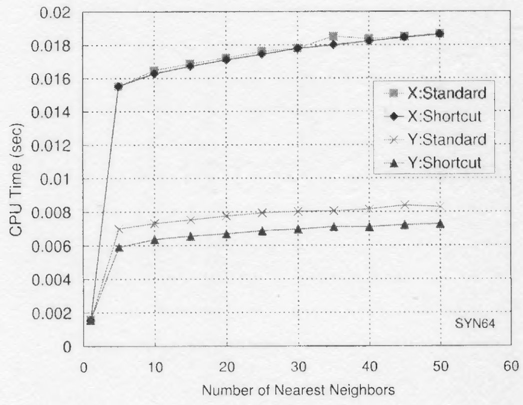
3. CPU time increases very slowly with k (except for $k = 1$, which is due to the efficiency of max-priority queue.

Figure 4.8 (b) shows the effect of indexing the Y versus X dataset via the OP-tree. The shortcut method does not improve performance with the X matrix and the improvement is small for the Y matrix, as explained below. There is a significant improvement in performance when the OP-tree is built with the Y rather than X matrix.

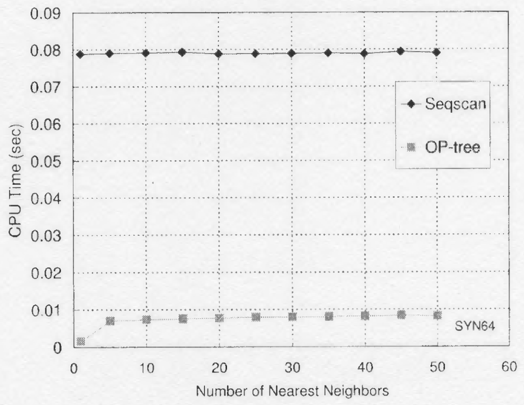
Figure 4.8 (c) gives the improvement in CPU time of the OP-tree versus sequential scan method using the standard method on Y, since the shortcut method does not provide much improvement with the OP-tree. The OP-tree index improves CPU time eight times for $k > 1$. The improvement is more than 50 times for $k = 1$ with the standard and shortcut method.



(a)



(b)



(c)

Figure 4.8 The effect of shortcut Euclidean distance calculation on CPU time in processing k-NN queries on SYN64 versus k. (a) Sequential scan. (b) OP-tree. (c) The performance gains of OP-tree over sequential scan.

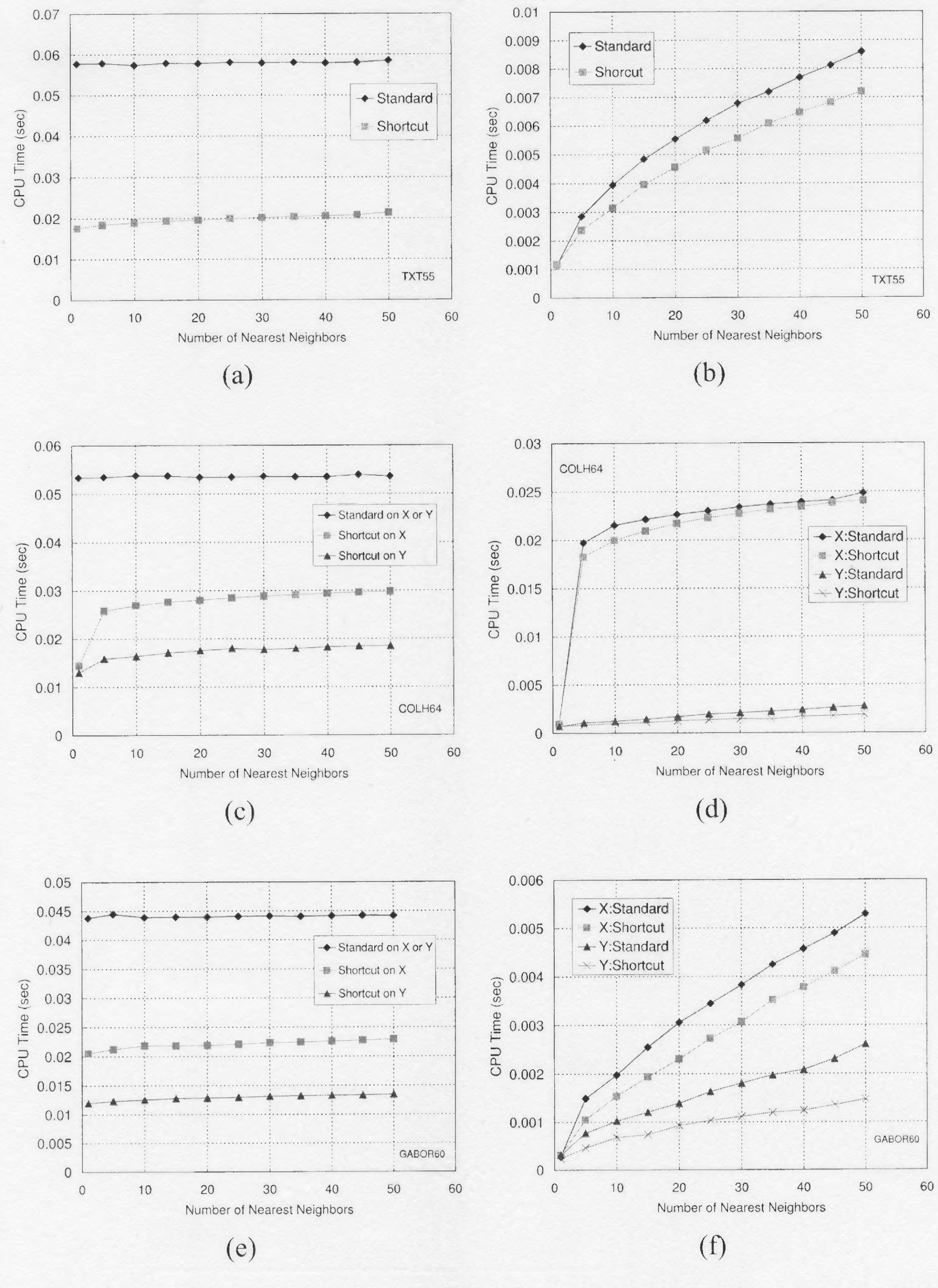


Figure 4.9 The effect of shortcut Euclidean distance calculation on CPU time in processing k-NN queries on TXT55 versus k. (a, c, e) Sequential Scan. (b, d, f) OP-tree. (a, b) TXT55. (c, d) COLH64. (e, f) GABOR60.

Figure 4.9 shows the experimental results for the other three datasets.

To explain the difference in cost of processing k-NN queries the following loop without and with a test for early exit is considered. $p[N]$ and $q[N]$ denote the feature vectors corresponding to a sample point P and the query Q. The vectors may be in the original or the transformed domain. The squared Euclidean distance between the two points ($D^2(P, Q)$) can be calculated as follows:

```
Standard Euclidean Distance Calculation:
    $for(i = 0; i < N; i++)\{$
        $temp = p[i] - q[i];$
        $dist += temp * temp;$
    $\}$
```

```
Shortcut Euclidean Distance Calculation:
    $for(i = 0; i < N; i++)\{$
        $temp = p[i] - q[i];$
        $dist += temp * temp;$
        $if(dist \leq farthest) \quad break;$
    $\}$
```

Let t_{ma} denote the cost of the multiplication and additions and t_{ex} the cost of the break statement. The shortcut method improves performance if the early exit for a point Q is taken at dimension n such that

$$n \times (t_{ma} + t_{ex}) < N \times t_{ma} ,$$

where t_{ma} is the cost for one multiplication plus one addition and one substraction and t_{ex} the cost for one test. The total cost for the standard method is $N \times t_{ma}$, while the total cost for the shortcut method is $n \times t_{ma} + n \times t_{ex}$.

The maximum value of n for which the shortcut method is still preferable is given as:

$$n_{max} = N \frac{t_{ma}}{t_{ma} + t_{ex}}.$$

There are reasons to believe that t_{ex} is much larger than t_{ma}, so that n_{max}/N is rather small.

When dealing with a large number of points, it is the average number of dimensions to the exit point which matters ($\bar{n}$). Values of $\bar{n}$ in experiments are reported in Table 4.4, 4.5, 4.6, and 4.7. It is observed that $\bar{n}$ is quite small for sequential scan, because there are a large number of points in the dataset and the majority of which are at a great distance from the query point Q, so that they can be excluded easily.

The OP-tree is an excellent index in that it limits the number of points in the dataset that need to be considered for k-NN processing. All the points being considered are in neighborhoods close to Q, so that more dimensions need to be considered to find the nearest neighbors.

As far as the improvement in $\bar{n}$ going from the X to the Y dataset is concerned, this can be attributed to the fact that the dimensions of Y are ordered according to their variances. The improvements is a factor of two for sequential scan and a factor of 1.5 for the OP-tree.

Table 4.4 Average Number of Dimensions Early Terminated by the Shortcut Method for SYN64

$\bar{n}$	1	5	10	15	20	25	30	35	40	45	50
seqscan(X)	3.09	5.67	6.07	6.33	6.54	6.72	6.88	7.03	7.16	7.29	7.40
seqscan(Y)	1.77	2.71	2.85	2.94	3.01	3.07	3.13	3.18	3.22	3.26	3.31
Op-tree(X)	23.06	13.28	13.56	13.73	14.02	14.11	14.18	14.25	14.32	14.38	15.66
Op-tree(Y)	19.24	8.1	9.20	9.44	9.62	9.77	9.90	10.02	10.12	10.22	10.31

There are other ways to make the evaluation of k-NN queries more efficient. For example, precompute the norm for all points, e.g., $||p|| = \sum_{i=0}^{N-1} p[i]^2$, so that $D^2(\vec{p}, \vec{q}) = ||\vec{p}|| + ||\vec{q}|| - 2\vec{p} \cdot \vec{q}$. An inner product instruction is provided by some processors. Fur-

Table 4.5 Average Number of Dimensions Early Terminated by the Shortcut Method for TXT55

$\bar{n}$	1	5	10	15	20	25	30	35	40	45	50
seqscan(Y)	2.65	3.22	3.51	3.73	3.91	4.07	4.22	4.35	4.47	4.59	4.70
Op-tree(Y)	14.67	4.69	5.34	5.75	6.05	6.28	6.48	6.65	6.80	6.93	7.05

Table 4.6 Average Number of Dimensions Early Terminated by the Shortcut Method for COLH64

$\bar{n}$	1	5	10	15	20	25	30	35	40	45	50
seqscan(X)	2.23	9.10	9.91	10.42	10.81	11.13	11.40	11.64	11.86	12.06	12.24
seqscan(Y)	1.32	2.86	3.21	3.45	3.64	3.80	3.95	4.08	4.20	4.31	4.42
Op-tree(X)	16.63	11.17	11.71	12.08	12.37	12.61	12.82	13.01	13.18	13.35	13.49
Op-tree(Y)	16.12	3.37	3.47	3.50	3.65	3.71	3.80	3.82	3.88	3.91	3.94

thermore, some numerical packages, such as IBM's ESSL generate very efficient code for vector computations.

OP-trees versus SR-trees From Figures 4.10, it is observed that SR-trees spent much more time on internal node searching than leaf node searching, while OP-trees spent less time, especially when the dimensionality is high. SR-trees are created with the minimum utilization 0.4 and the reinsert factor 0.3 as suggested in [17] throughout this chapter. Since leaf node searching mainly focus on the distance calculation of the query to each point in that leaf, the OP-tree performance can be improved further by using the optimized distance calculation.

OP-trees versus the OMNI-Family The OP-tree performs consistently better than the naive-seqscan and OMNI-seqscan for k-NN queries and performs better only when the

Table 4.7 Average Number of Dimensions Early Terminated by the Shortcut Method for GABOR60

$\bar{n}$	1	5	10	15	20	25	30	35	40	45	50
seqscan(X)	11.76	12.34	12.64	12.89	13.09	13.27	13.42	13.57	13.73	13.91	14.10
seqscan(Y)	2.29	2.58	2.77	2.93	3.06	3.16	3.25	3.34	3.42	3.50	3.57
Op-tree(X)	22.18	4.97	5.87	6.35	6.77	7.09	7.45	7.76	7.97	8.18	8.41
Op-tree(Y)	12.41	3.18	3.27	3.30	3.40	3.45	3.48	3.52	3.58	3.63	3.67

selectivity is relatively low for range queries. The experimental results are reported as in Appendix C,

4.4 The CIPOP Indexing Method

The viability of persistent versions of main memory indices is first discussed. The serialization method is next introduced, which transforms the linked list index structure built based on dynamic storage allocation into contiguous main memory locations. The k-NN processing is then described and finally the experimental results are reported.

4.4.1 Motivation

Clustering before indexing has a double filtering effect, since only clusters or index nodes intersecting the hypersphere of the current k nearest-neighbors of the query point need to be visited [18]. The index can be main memory or disk resident. For disk resident indices the time for processing queries is referred to as elapsed time $T_{elapsed}$, which is the sum of T_{io} for loading the pages on the search path and T_{cpu} for querying the pages.

$$T_{elapsed} = T_{io} + T_{cpu}. \tag{4.4}$$

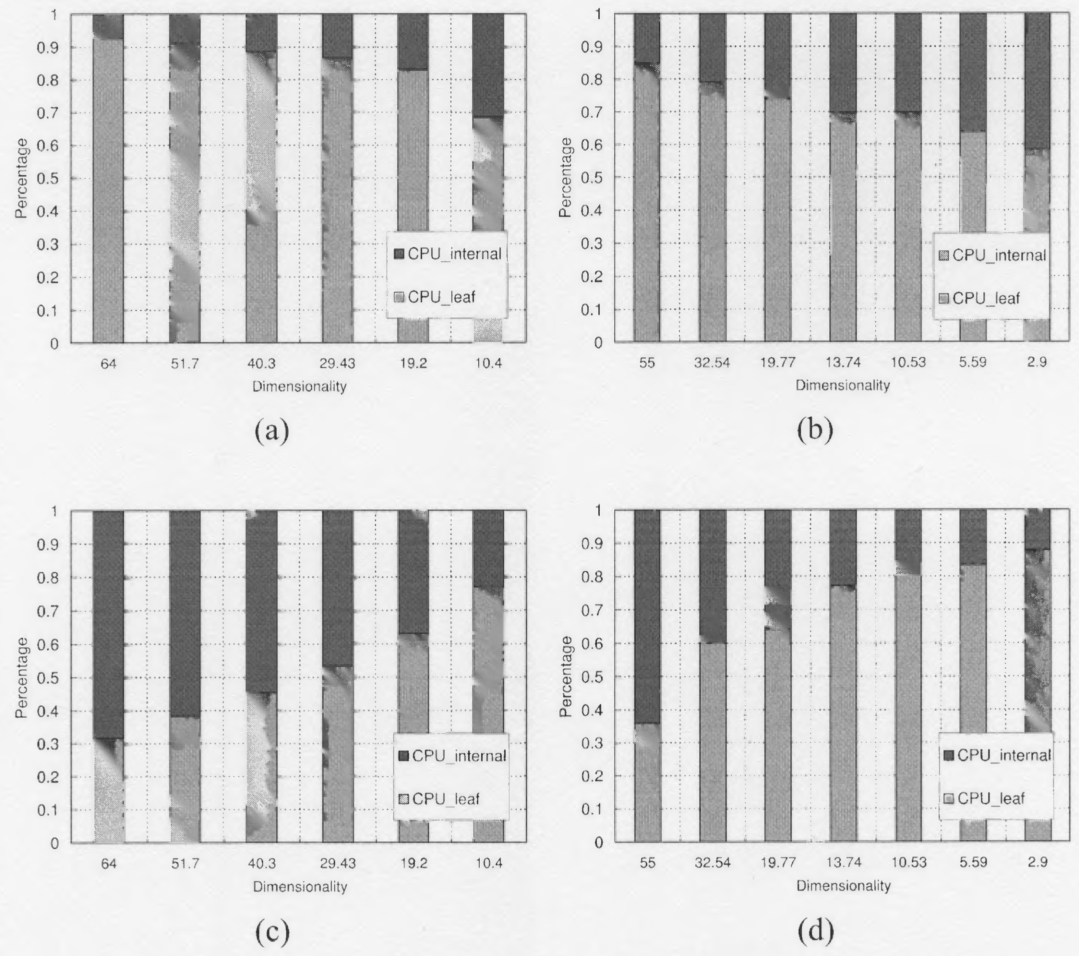


Figure 4.10 Percentage of time spending on leaf searching and internal node search. (a, b) For OP-tree. (c, d) For SR-tree. (a, c) For 16 clusters of SYN64. (b, d) For 32 clusters of TXT55.

For memory-resident indices, since the index is not persistent, the time for processing k-NN queries is the sum of the time to build the index (T_{build}) and querying it (T_{cpu}).

$$T_{elapsed} = T_{build} + T_{cpu}. \tag{4.5}$$

The query processing cost should be prorated over a sufficiently large number of queries to make this method viable [1]. Moreover, the main memory should be large enough to hold the index.

Main memory indices are designed to reduce the CPU time, while disk resident indices are designed to minimize the number of disk accesses by only retrieving the nodes on the search path. CPU time for k-NN processing with main memory indices is expected to be less than that for disk resident indices, which is due to the fact that data in main memory indices is densely packed, ensuing a lower cache miss rate. This is shown by the experimental study in Section 4.4.4.

The alternative solution used in this study is to make the main memory index persistent by writing it onto disk. Serialization is required to make the index more compact, before it is written to disk. CIPMM, defined below, takes advantage of the reduced CPU time afforded by main memory indices and the high disk transfer rate to load persistent versions of main memory indices into main memory.

Definition 4.1 (CIPMM) *A general framework for Clustering and Indexing using Persistent Main Memory indices.*

The CIPMM can be applied to any main memory index structures with the difference that the indices are stored on disk as a single file, so that they can be loaded into main memory efficiently. The steps for constructing a CIPMM are:

- Partition the dataset into clusters.
- Build a main memory index for each cluster.
- Serialize the indices.
- Make indices persistent.

[1]A similar situation is encountered when conversion cost from a (relational) ROLAP table into a (multi-dimensional) MOLAP array is incurred for efficient analytical processing [94].

In this study the standard k-means clustering algorithm [95] is used with the initial set of centroids selected far apart from each other [29]. The number of clusters is chosen such that the index representing each cluster can be held in main memory. The clustering algorithm is run multiple times, and the partition with the minimal sum of squared error is selected.

The performance of persistent main memory OP-trees [16] is compared with SR-trees [17] and VA-Files [18] as the within cluster index, which are referred to CIPOP, CISR and CIVAFile, respectively.

Definition 4.2 (CIPOP) *A specific instance of CIPMM using Persistent OP-trees.*

Definition 4.3 (CISR) *Clustering and Indexing using SR-trees.*

Definition 4.4 (CIVAFile) *Clustering and Indexing using VA-Files.*

The CIPOP will be used to illustrate how a main memory index can be serialized, stored, and restored. In this case the KL transform is applied, which benefits the OP-tree as shown before.

4.4.2 Two-Phase Serialization

The need for serialization is justified as follows. When the index is built in main memory using system's dynamic memory allocator, the allocated nodes are scattered in main memory space. Writing these separately allocated nodes individually and reading them back could be very expensive, since there are a large number of small nodes and each of which requires the overhead of allocating space before it is loaded into main memory. This problem can be alleviated by following the first phase of building the index with a second phase, which serializes the index into a *Contiguous Memory Area - CMA* without losing any structural information. The CMA can be written to or read from the disk as one file access.

The OP-tree is serialized into a CMA as shown in Figure 4.11. The CMA consists of four areas as follows.

Area A: The internal nodes and the leaf nodes. The pointers in the nodes are the offset to the base address of the CMA. The size of area A = number of leaf nodes × sizeof(leaf node) + number of internal nodes × sizeof(internal node).

Area B: A list of point IDs, the order of the IDs depends on the order of the leaf nodes in area A. The size of area B = number of points × sizeof(point ID).

Area C: A list of pointers to the points. The size of area C = number of points × sizeof (pointer).

Area D: The actual data in row order. The size of area D = number of points × dimension × sizeof(data type for a point).

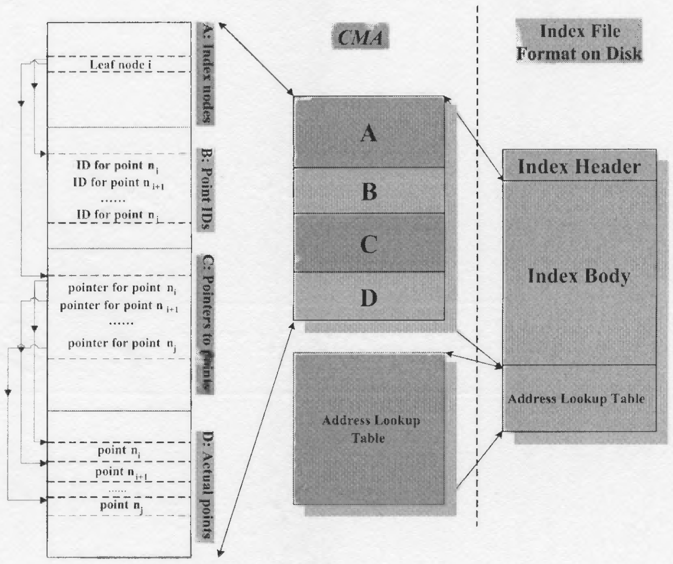


Figure 4.11 The structure of the contiguous memory area (CMA) and the index file format on disk.

The size of the CMA is the sum of the sizes of the areas, which can be computed by traversing the index. This is required to determine the size of the buffer, which has to be preallocated for CMA. To build the CMA the dynamically allocated index tree is traversed in depth-first order. For each node visited, its content is copied into the next available space in the CMA and pointers are changed to be the relative offsets to the base address. At the same time, addresses of words containing pointers are recorded into an *address lookup table*.

The CMA is written to disk with the following additional information: the number of points in the cluster, the number of dataset dimensions, split factor for each dimension, the size of the index, the size of the address lookup table, the index, and the address lookup table.

4.4.3 k-NN Processing

The algorithm for processing a k-NN query is described in Algorithm 4. The primary cluster [3] is first identified. Since the k-means method is used for clustering, this is the cluster with the closest centroid to the query Q. The distance between Q and a cluster c is defined as $max\left\{0, D(Q, u^{(c)}) - R^{(c)}\right\}$. $D(Q, u^{(c)})$ means the distance between Q and $u^{(c)}$. The clusters are stored in increasing distance order and ties are broken using $D(Q, u^{(c)})$ [3].

The OP-tree is loaded into main memory by using sequential disk accesses, i.e., by first reading the index header, then reading in the index body and address lookup table into a CMA according to the information in the header. Finally, the relative offsets are adjusted to absolute addresses in memory.The entries in the address lookup table are used to replace the offsets in the index with the actual address based on the new base address.

After the primary cluster is loaded from disk, it is searched using Algorithm 3. A set of candidate results and the radius, $search\_radius$, for the current search sphere are determined. The next cluster is searched if its distance from Q does not exceed the $search\_radius$. Otherwise, the search terminates. If points closer to Q are found, they are inserted to the current search results and $search\_radius$ is updated.

For example, in Figure 4.12, after searching cluster C4 which is the primary cluster for query Q, cluster C2 will be searched as the next candidate cluster. The search sphere will possibly shrink during each visit of a cluster. This process continues until the search sphere does not intersect with any of the clusters. In this example, c4, c2, and c3 are visited in order.

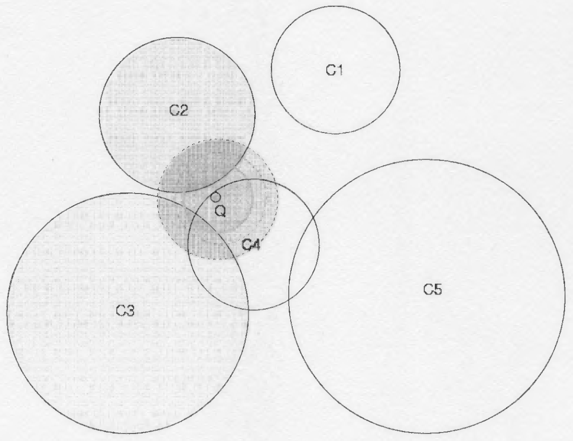


Figure 4.12 Cluster identification.

Algorithm 4 CIPOPKnnSearch(Q, r, d_p, l)

```
1: calculate the distance from $Q$ to the edge of each cluster;
2: sort the distances, ties are broken by considering the distance to the centroid of the cluster
3: find the primary cluster $C$;
4: load the index of $C$ to $I$;
5: call knnSearch($Q$, $I$, 0, 0);
6: while (the next candidate cluster $C$ exists) do
7:     load the index of $C$ to $I$;
8:     call knnSearch($Q$, $I$, 0, 0);
```

There are three factors contributing to CPU time: (a) the maintenance of the k-NN priority queue, (b) searching the index nodes, (c) scanning the points in the leaf nodes.

4.4.4 Experimental Evaluation

In this section, the performance of CIPOP is studied and compared with CISR and CIVAFile.

Index Size The size of the CISR is almost twice as large as the original dataset, while the increase in size is less than 1% for CIPOP as reported in Table 4.8. The performance of CIPOP benefits from its smaller size.

Table 4.8 Sizes for the CIPOP, CISR, and Original Dataset

Dataset	SYN64	COLH64	GABOR60	TXT55
CIPOP (KB)	51,686	35,083	27,348	35,414
CISR (KB)	92,328	64,176	57,968	61,832
Original (KB)	51,186	34,823	27,189	35,118

CPU and Elapsed Time The performance of three disk-resident indexing structures are compared with each other in Chapter 3. The disk-resident indices considered are R-trees [46], SR-trees [17], hybrid trees [60]. Since the SR-tree incurs fewer page accesses, it is chosen for comparison in this study. For SR-trees the page size is set to 8 KB and the recommended parameters in [17] are used with the minimum utilization 0.4 and the reinsert factor 0.3. The VA-File is also selected as a reference since it is efficient in processing nearest neighbor queries for high-dimensional data.

The CPU time of CIPOP, CISR, and CIVAFile is first compared, then the elapsed time is compared. Due to caching and aggressive prefetching used by modern operating systems, it is difficult to compare the two methods by measuring the elapsed time when disk I/O is involved. The operating system prefetches anticipatory blocks as soon as one block is touched [96]. The CPU and I/O time are overlapped in an unpredictable manner. In the experiments with SR-tree under windows 2000 with 512MB memory [96], after running hundreds of queries, the measured elapsed time is quite close and slightly higher than the CPU time. This means that prefetching of data from disk is heavily overlapped with CPU processing. Instead, the computed elapsed time [60, 93, 15] is utilized, rather than the measured elapsed time.

The average elapsed time $\bar{t}$ over 1000 queries on H clusters is calculated as follows: $\bar{t}_e = \bar{t}_{cpu} + \bar{t}_{io}$. In the case of the SR-tree, $\bar{t}_{io} = \bar{p} \times \bar{t}_p$, where $\bar{p}$ is the average number of pages accessed and $\bar{t}_p$ is the average time to access an 8KB page. In the case of the persistent OP-tree, $\bar{t}_{io} = \bar{h} \times (t_{positioning} + \bar{s}/xfer\_rate)$, where $t_{positioning} = t_{seek} + t_{latency}$, $\bar{s}$ is the average file size loaded for each cluster given as

$$\bar{s} = \sum_{i=1}^{H} f[i] \times s[i],$$

$\bar{h}$ is the mean number of clusters visited by each query, and $f[i]$ is the frequency of accesses to cluster i and

$$f[i] = \frac{Access[i]}{\sum_{i=1}^{H} Access[i]}.$$

For one cluster, the average CPU time and elapsed time over one thousand k-NN queries with $k = 20$ for the four datasets are reported in Table 4.9. It is observed that the OP-tree outperforms the SR-tree more than ten-fold in one case and less than 2-fold in another. This is not unexpected. the SR-tree is an efficient index from the viewpoint of reducing the number of page accesses for k-NN queries, while the OP-tree minimizes the CPU time for tree traversal for k-NN queries.

Table 4.9 Comparison of CPU time and Elapsed Time in Seconds for CIPOP and CISR with Respect to Different Datasets (Single Cluster)

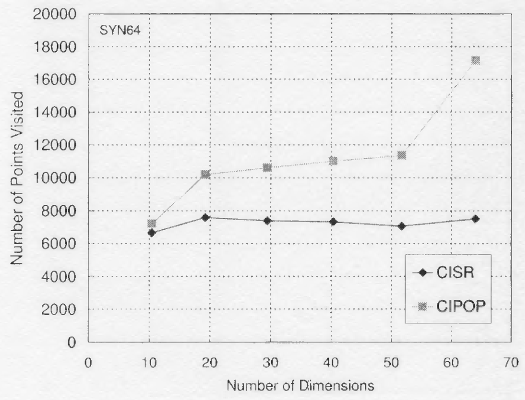
CPU Time	SYN64	COLH64	TXT55	GABOR60
CISR	0.037	0.093	0.07	0.005
CIPOP	0.016	0.023	0.005	0.003

Elapsed Time	SYN64	COLH64	TXT55	GABOR60
CISR	5.36	13.95	10.22	0.75
CIPOP	0.93	0.64	0.65	0.50

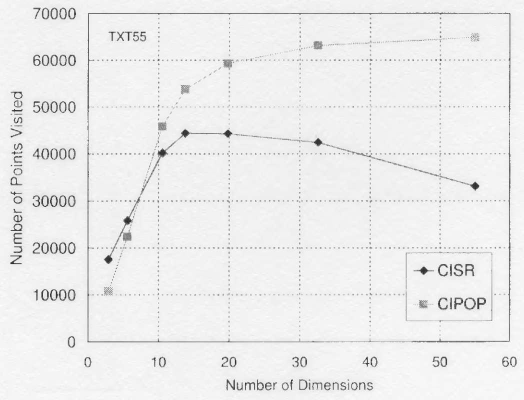
For multiple clusters, two clustered datasets are considered: SYN64 partitioned into 16 clusters and TXT55 partitioned into 32 clusters. To study the effect of variable number of dimensions, the global dimensionality reduction method in [3] is utilized for a given target NMSE, and the average number of dimensions is used as the dimension of the dataset. Figures 4.13 reports the number of points visited, CPU time, and elapsed time versus the average number of retained dimensions for both clustered datasets. The target NMSEs for SYN64 are given by $\{0, 0.01, 0.02, 0.03, 0.04, 0.1\}$ and for TXT55 by $\{0, 0.01, 0.05, 0.1, 0.15, 0.3, 0.45\}$. The CIPOP outperforms the CISR and CIVAFile up to a factor of ten in terms of CPU time, although CIPOP visits more points. In the case of elapsed time, the CIPOP is so much faster than the CISR that they are plotted in a logarithmic scale in base 10 for 1000 queries as shown in Figure 4.13(c,f). In both cases the standard method for distance calculation is used. Moreover, the higher the number of dimensions, the larger the difference.

To understand why CIPOP runs much faster than CISR, the time spent on searching internal nodes and leaf nodes is broken down for the two clustered datasets with CIPOP and CISR indices in Figure 4.10. We observe that CISR spends more time than CIPOP on internal nodes and that this is especially so when the dimensionality is high. The reason is that searching an internal node on an OP-tree only incurs one multiplication and two additions, while searching an internal nodes on SR-tree incurs the cost for the distance calculation and sorting. The higher the dimension, the higher the distance calculation cost.

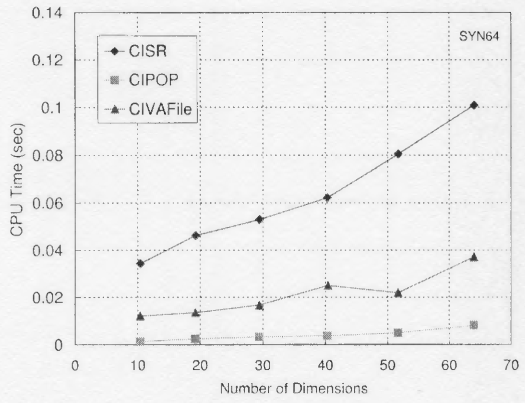
Clustering Experiments have shown that the elapsed time is insensitive to the number of clusters (H) (Figure 4.14). The elapsed time is measured by flushing the buffer for each run. When H is small, few clusters are visited, so that less time is spent on positioning time (sum of seek and rotational latency), but more time is spent on transfer time, and the CPU time for processing k-NN queries. When H is large, more clusters are visited, so that positioning time is higher, but the transfer time per cluster and CPU processing time is lower. Experiments on clustered dimensionality reduced datasets are carried out, varying the number of clusters from 2 to 64 in multiples of 2 (Table 4.10 for SYN64). The elapsed time does not exhibit an optimum H. Same conclusion is drawn for the other datasets as



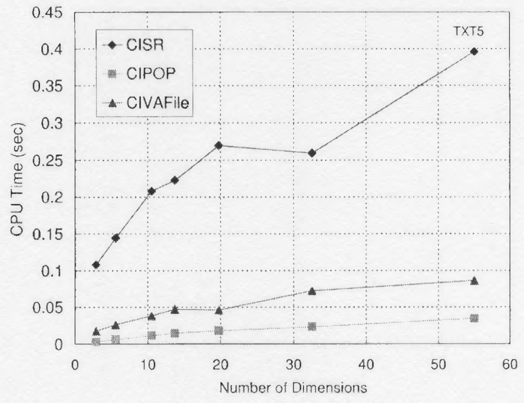
(a)



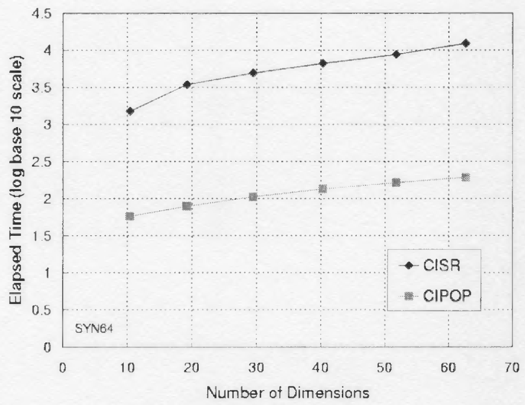
(d)



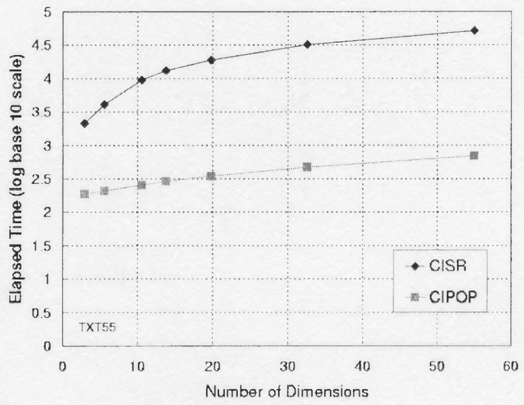
(b)



(e)



(c)



(f)

Figure 4.13 Performance comparison in processing k-NN queries for SYN64 with 16 clusters (left column) and TXT55 with 32 clusters (right column). (a, d) Number of points visited. (b, e) CPU time. (c, f) Computed elapsed time versus number of retained dimensions. Elapsed time is for 1000 queries, while others are averaged.

in Figure 4.14. This is partially due to the aforementioned compensating effects, but also caching, since the time measured to process a 1000 k-NN queries, i.e., the large file cache was primed with all clusters after processing a few queries.

Table 4.10 Average Number of Clusters Visited ($\bar{n}_c$)

$\bar{n}_c$	Number of Clusters						
	2	4	8	12	16	32	64
8dim	1.262	2.286	3.922	5.514	7.27	9.423	16.526
16dim	1.311	3.196	3.936	7.841	9.788	15.282	25.49
32dim	1.362	3.361	5.969	8.095	10.969	17.158	27.608
64dim	1.987	3.987	6.286	7.132	12.119	20.902	39.164

4.5 Persistent Semi-Dynamic OP-tree

The OP-tree is a static structure. In this section, it is extended to be semi-dynamic. Several strategies for handling the insertion are presented: (a) adding levels, (b) varying fanouts, (c) chaining overflow data, and (d) forced reinsertions. The index is then serialized using a one-phase serialization method so that it can be saved to and loaded from disk with a sequential disk access and only the modified pages needed to be rewritten. The proposed methods are evaluated by experimentation.

4.5.1 Semi-Dynamic OP-tree

Two parameters are used for constructing an OP-tree: the number of splits at level l, denoted by s_l, and the leaf capacity c, which is the maximum number of points that can be held in a leaf node. When all splits are s-ways, the number of levels l is the smallest integer that satisfies $N \leq c \times s^l$, which leads to $l = \lceil log_s(N/c) \rceil$. The selection of the two parameters affects the performance of the index structure.

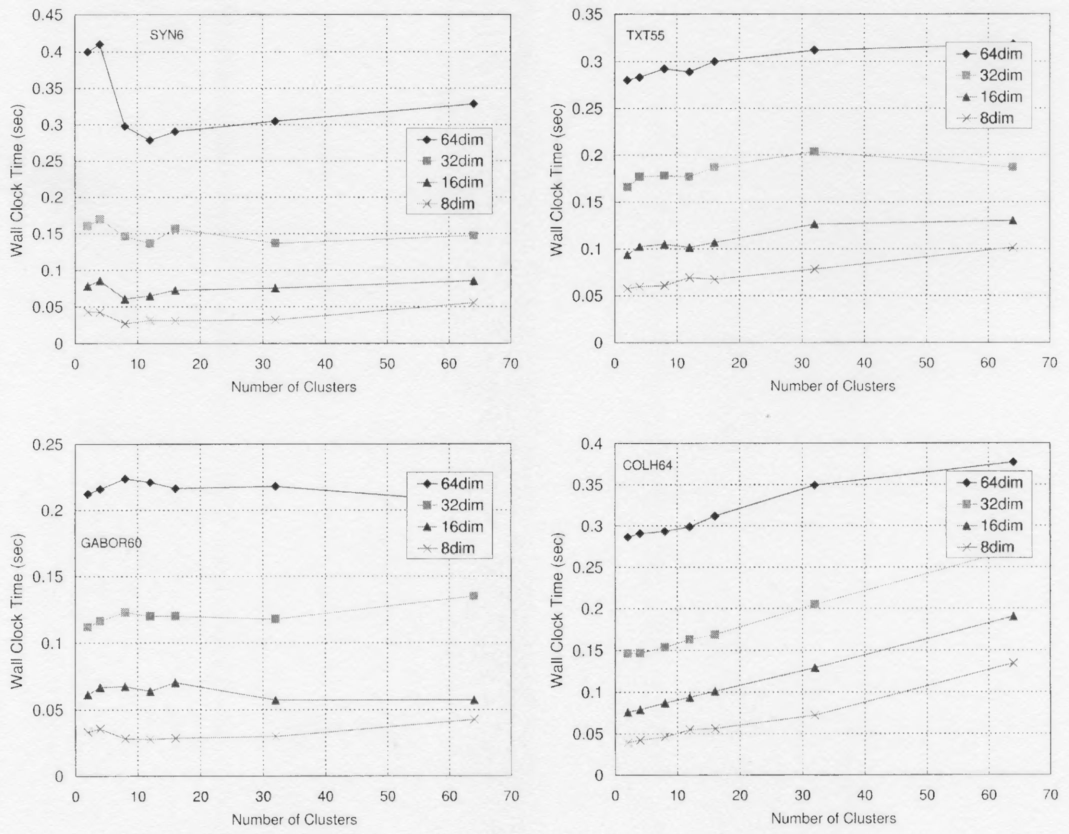


Figure 4.14 Try to find optimal number of clusters for 20-NN search.

The k-NN search algorithm for OP-trees is given in Algorithm 3. Noting that if the first dimensions have most contribution to the Euclidean distance, the pruning ability will be enhanced. Before the index is built, the Karhunen-Loève (K-L) transform is applied to the dataset, so that dimensions with the higher variance are assigned to top levels of the index. In this way, the OP-tree recursively divides space one dimension at a time, starting with dimensions with higher variances [3].

Methods for Dynamic Insertions The OP-tree is extended to handle dynamic insertions in the following way. For a new point, the bucket to which the point belongs is first determined. If the bucket is full, the following four methods are considered to deal with bucket overflows:

(a)Adding levels: The original leaf node is replaced with a new internal node. s_{l+1} new leaf nodes are created and the pointers between the internal node and the leaf nodes are adjusted appropriately (Figure 4.17(a)). This maintains the property of the OP-tree. In this way, 80% of the CPU time was spent on searching leaf nodes regardless how many new points are inserted, which is shown in Figure 4.15. Each deletion just remove the data from the index. When the bucket is empty, no more action is needed.

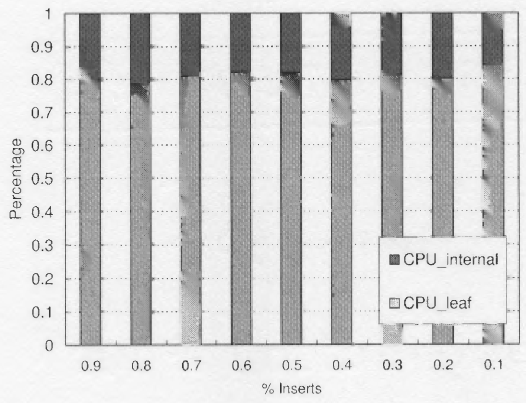


Figure 4.15 Percentage of time spending on leaf searching and internal node searching for SYN64.

(b)Varying fanouts: The leaf node is replaced by two newly created leaf nodes with the data points equally split among the two. When more and more nodes are split, the linked list of leaf nodes will become longer and take more time to traverse. Although the OP-tree will remain height balanced, it will have unequal fanouts in different neighborhoods (Figure 4.17(b)). The fanout can be reduced when a bucket is empty, or two neighboring buckets are less than 50% full, or more generally the sum of the number of points belonging to the same set of sibling leaf nodes can fit into fewer than the current number of buckets.

(c) Chaining overflow data: When a bucket overflow, space for another bucket is allocated and pointers associated with each bucket is updated to point to the next bucket

(Figure 4.17(c)). During deletion, if two successive buckets are less than 50% full they can be merged.

(d) Forced reinsertions: A certain percentage of the full nodes p are reinserted. The steps are: 1. sort the distances between each point with the centroid of the $c + 1$ points in decreasing order. 2. remove the first p points. Adjust the lower and upper bounds of the nodes on the path to the leaf node. 3. insert the p points. In step 2, it is easy to operate at the leaf node. But to modify the lower and upper bounds of nodes at higher levels, all the descendants need to be known in advance. This is very time consuming and inefficient.

Figure 4.16 illustrates how to remove one point. After removing the point marked with star (suppose it occurs at the 2nd dimension), lower bound and upper bound for all the ancestors (the first dimension) have to be adjusted. It is easy to change u2 to u2', but for changing u1 to u1', we have to visit I and G. In general, for each ancestor, all the descendants need to be visited. This is very time consuming and inefficient. this strategy is not implemented.

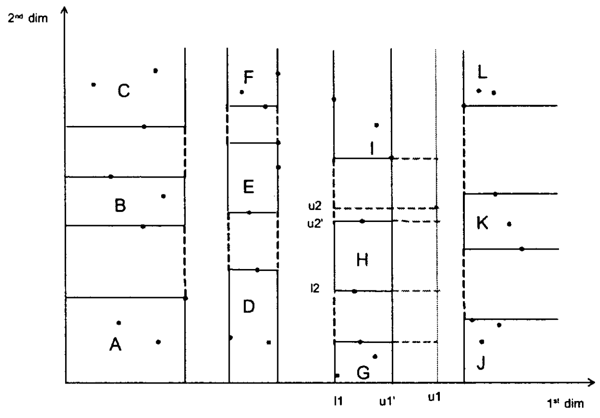


Figure 4.16 Point removal.

k-NN queries based on different percent of insertions for different strategies are compared in terms of the CPU time in Section 4.5.4. The conclusion is that the adding levels method outperforms the others.

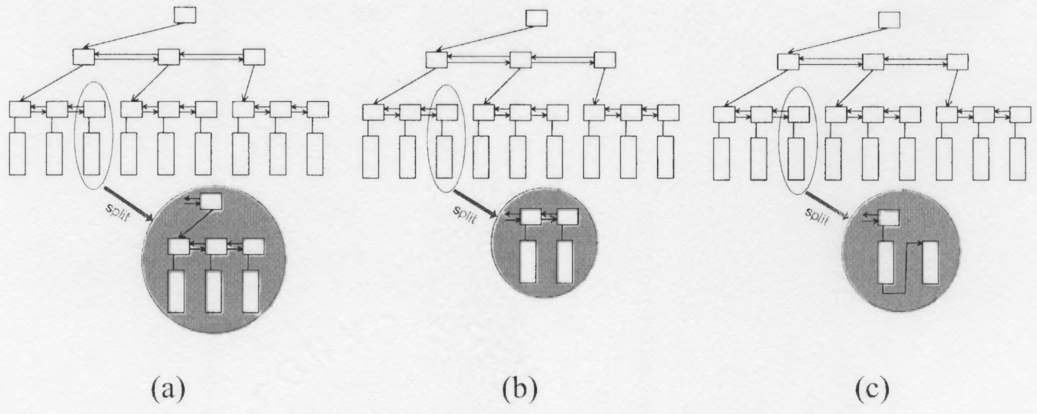


Figure 4.17 (a) Adding levels. (b) Varying fanouts. (c) Chaining overflow data.

4.5.2 One-Phase Serialization

A index can be serialized by first creating it and then compact it into a contiguous space with full space utilization [97]. But whenever new data is inserted, it has to be reserialized and the whole index file need to be rewritten. The index is serialized during the process of creating it. Some space in the structure is reserved to support updates. Although the utilization is not full, the index is more expandable. The insertion of new points requires only the rewriting of the modified parts of the index

A page manager is responsible for memory allocation. Each page can hold only one type of data, but there can be multiple pages for each data type. There are four data types: *Node* (including internal nodes and leaf nodes), *OrderedPointSet*, *OrderedPoint*, and *Point*. Each node is a 5-tuple {*lower*, *upper*, *child*, *left*, *right*}. *Lower* and *upper* are the lower and upper values for bounding the region, *left* and *right* are pointers pointing to the left and right siblings, *child* is a pointer pointing to the leftmost child for the internal node, the list of feature vectors for the leaf node (*OrderedPointSet*). *OrderedPointSet* is a set of *OrderedPoint*, with the size specifying the cardinality of the set and a pointer pointing to it. *OrderedPoint* is a *Point* with its ID. *Point* contains the actual feature vector or coordinates.

Here, all pointers are all logical pointers consisting of a page number and offset, each of which is an unsigned 16-byte integer. Each page has a maximum of 64 KB. The actual memory address is calculated as: actual_address = page_base_address + offset.

The page manager has two direct hash tables: *all_pages* and *active_pages*. *all_pages* keeps a list of base addresses for all the allocated pages. The hash key is the page number. *active_pages* keeps a list of base addresses for the active pages, which are the pages with free space for further allocation. The hash key is the type number. Each data type has one active page. To allocate more space for a data type, say *Node*, the active page for that type is first checked to see whether there is enough space available. If so, the page number and offset are returned, otherwise, a new page from the memory pool is allocated. At the same time, an entry is added in *all_pages* and the active pages for type *Node* in *active_pages* is modified, then the newly allocated page number and offset (0) are returned. This is illustrated in Figure 4.18. The pointer marked with 'X' points to the old active page for *Node*. Since the page does not have enough space to satisfy the request, a new page pointed by the current active pointer is allocated.

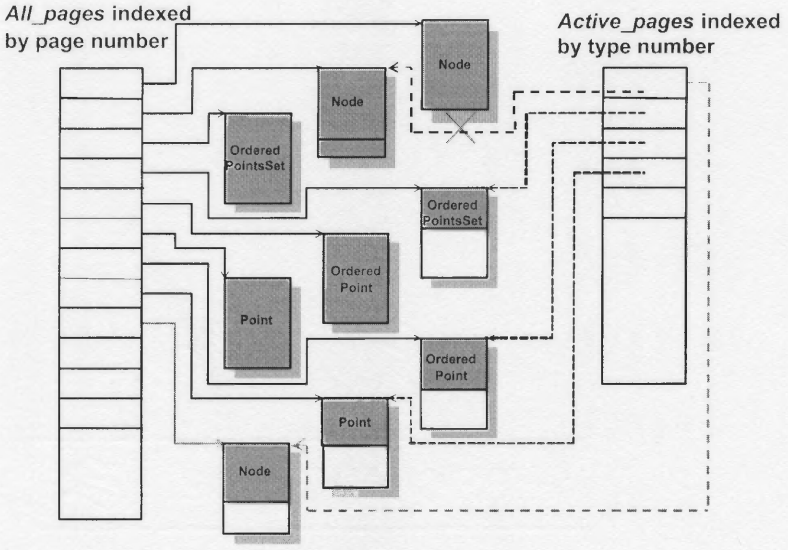


Figure 4.18 Page manager with two hash tables: *all_pages* and *active_pages*. *all_pages* is a list of page pointers pointing to all the allocated pages. *active_pages* is a list of page pointers pointing to the current active pages for each data type.

Once the index structure is serialized, it is made persistent by writing the pages out to the disk one by one in addition to the aforementioned metadata. Figure 4.19 shows the index file format and page layout.

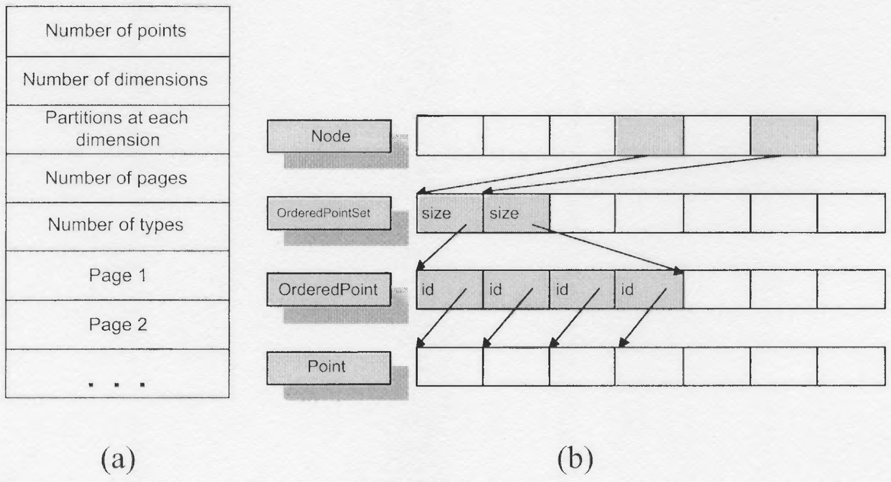


Figure 4.19 (a)Index file format. (b)Page layout.

Loading the Persistent OP-tree The OP-tree can be loaded by a sequential disk access. The metadata and all the actual pages are read in sequentially. Two hash tables *all_pages* and *active_pages* are built after the loading process. Once finished reading, the index structure is fully restored as before it is written out.

4.5.3 Scalability

The OP-tree can be build in main memory for a relatively large dataset due to rapid growth in main memory sizes [92]. The main memory is three orders of magnitude smaller than (aggregate) disk capacity, so that not all indices can be held in main memory.

When the number of feature vectors (or data points) to be indexed is very large, a clustering step is introduced to partition the dataset into clusters before building the index, For each cluster, an OP-tree is built and serialized.

The standard k-means clustering algorithm [95] is used to partition the dataset. The number of clusters is chosen, such that the index of each cluster can be held in main memory. The clustering algorithm is run multiple times, and the partition with the minimal sum of squared error, which satisfies the main memory constraint is selected.

When adding new data to a clustered dataset, the cluster to which it belongs is first identified and loaded. The insertion process marks all newly allocated and modified pages as *dirty* so that they are saved on disk.

For k-NN queries, a "high-level" main memory resident index is used to determine the clusters that need to be loaded to process a k-NN query. The OP-trees of relevant clusters are loaded into main memory via sequential disk accesses. The filter and refine paradigm by clustering and indexing will obviously improve query processing performance. The detailed description can be found in [97].

4.5.4 Experimental Evaluation

File Size For one cluster, the SR-tree takes almost twice as much disk space as the one-phase persistent OP-tree while the one-phase persistent OP-tree is slightly larger than the original dataset as in Table 4.11.

Table 4.11 Parameters for the OP-tree for Different Datasets and Resulting File Sizes, Size of the SR-tree, the Original Size, and Index Building Time

Dataset	SYN64	COLH64	GABOR60	TXT55
Number of splits (fanout)	2	2	2	8
Leaf node capacity	40	40	60	40
File size (Two-phase) (KB)	51,686	35,083	27,348	35,414
File size (One-phase) ((KB)	51,921	35,019	27,401	36,236
SR-tree (KB)	92,328	64,176	57,968	61,832
Original dataset (KB)	51,186	34,823	27,189	35,118
Tree building (Two-phase) (sec)	19.7	10.0	9.9	12.1
Tree building (One-phase) (sec)	17.6	9.5	9.6	11.7

Figure 4.20 shows that the file sizes of TXT55, COLH64 and GABOR60 do not vary much, regardless how many new data points are inserted using the adding level method. For SYN64 there is a major increase in file size if more than 10% of the points are to be inserted. The reason is that there are more cases when a newly inserted point incurs a split of its own. For the average wall clock time, the trend is almost the same as the filesize, since the time is determined by the file loading time which depends on the filesize. It is reassuring that the increase in file size and CPU processing in SYN64 occurs at the same time.

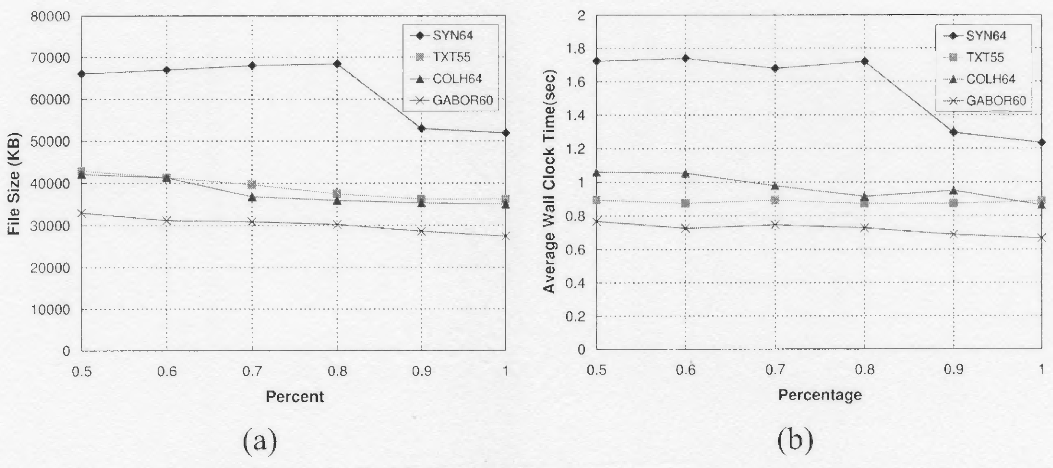


Figure 4.20 Performance of one-phase serialization. (a) Index size. (b) Wall clock time.

Different Strategies for Splitting The first three splitting methods during insertions described in Section 4.5.1 are implemented. Indices are first built based on 90%, 80%, ..., and 10% of the original data, then 10%, 20%, ..., and 90% of the original data obtained by simple random sampling without replacement are inserted respectively. 1000 20-NN queries are run ten times. The average number of internal nodes and leaf nodes, and the average number of points visited and CPU time for 1000 20-NN queries are reported in Table 4.12. The average CPU time and number of points visited for 1000 20-NNs based on different strategies are also plotted in Figure 4.21. The results show that method (a) is the best since it is insensitive to the number of points inserted and visits less points than method (b) and

(c) when most of the points are inserted. In the following experiment, the adding levels method is used.

Table 4.12 Comparison of Three Split Policies for 20-NN Queries with 2 Splits

		Add Level (a)				Expand Level (b)				Chained (c)			
build	insert	internals	leaves	points	cpu	internals	leaves	points	cpu	internals	leaves	points	cpu
0.1	0.9	1522.32	1208.48	31639.2	0.0184	148.34	1733.25	47978.5	0.0239	148.34	130.86	51228.1	0.0252
0.2	0.8	1540.37	1215.38	31836.8	0.0185	148.34	1733.25	47978.5	0.0207	279.18	235.85	46198.3	0.0234
0.3	0.7	1555.06	1219.39	31770.7	0.0182	515.68	1353.16	36360.6	0.0184	515.68	414.68	40638	0.0212
0.4	0.6	1555.71	1222.47	31805.3	0.0183	516.1	1372.86	36355.2	0.0185	516.1	415.37	40651.3	0.0213
0.5	0.5	1583.14	1252.76	31941.7	0.0179	930.8	1217.44	31360.9	0.0163	930.8	702.81	34297.7	0.0187
0.6	0.4	1604.55	1269.68	31839.2	0.0179	930.55	1240.41	31232.1	0.0163	930.55	702.19	34274	0.0188
0.7	0.3	1629.2	1291.19	31789.7	0.0181	930.94	1266.79	31127.3	0.0164	930.94	702.66	34281.3	0.0188
0.8	0.2	1633.65	1296.13	31815.6	0.0181	930.81	1275.36	31115.6	0.0164	930.81	702.59	34300.9	0.0188
0.9	0.1	1632.84	1318.71	32181.4	0.0171	1632.84	1318.71	32181.4	0.0171	1632.84	1318.71	32181.4	0.0188

Partial versus Full K-L Transform In this section, the effects of transforming the data onto a new frame of reference applied on the OP-tree are studied, which is based on the uncorrelated eigenvectors obtained from an increasing fraction of the dataset. Without K-L transform (KLT), the percentages of number of points visited for SYN64, COLH64 and GABOR60 are about 31%, 67%, and 10% respectively. With K-L transform, the percentages are 13%, 28%, and 4% respectively. By incrementally inserting points, the average tree levels are almost the same as the original index. Figure 4.22 and 4.23 give the average of ten experiments to build the tree by randomly selecting a fraction of points to insert and in each case 1000 k-NN queries are run. The OP-trees built with partial KLT (eigenvectors obtained based on fraction of the data) have almost the same number of tree levels and stan-

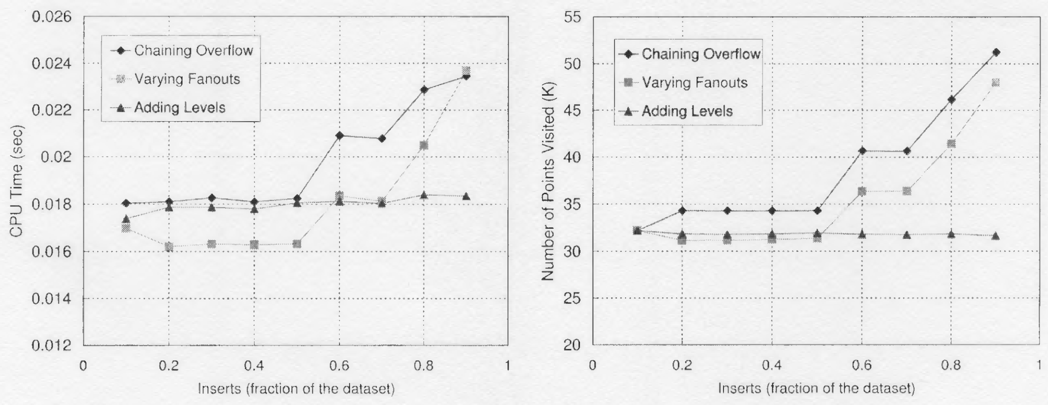


Figure 4.21 Average CPU time and number of points visited for 1000 20-NN queries based on different strategies for SYN64.

dard deviations as those with full KLT (eigenvectors are obtained based on the whole data). In either case, the more number of points inserted, the more standard deviations obtained. The OP-trees with partial KLT visit more points, and needs a little bit more CPU time for k-NN queries. This shows that the OP-tree is better built for data transformed using KLT, and is suitable for dynamic insertion.

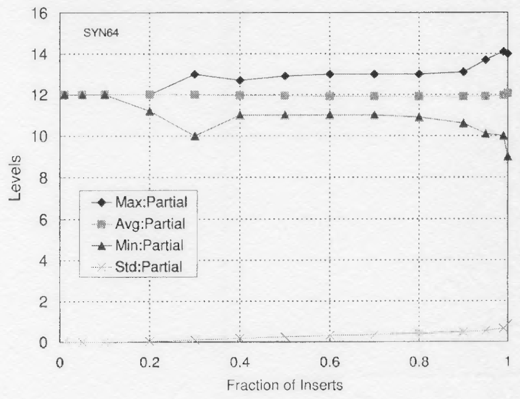
4.6 Conclusions

In this chapter a static main memory resident ordered partition index (OP-tree), which is highly efficient in processing k-NN queries, is introduced. The performance of OP-trees versus sequential scans is compared and a significant improvement is observed, which is higher when the index is built on the K-L transformed dataset Y, rather than the original X dataset. The effect of a shortcut Euclidean distance calculation method is also considered, which is more effective when used with sequential scans over the Y matrix. The selection of parameters of the OP-tree is experimentally carried out and results show that a split factor of two provides the best performance, although a higher split factor for dimensions with higher eigenvalues (for their covariance matrix) reduces the number points touched slightly. The results also show that in conducting k-NN queries the leaf node size should be larger than k. A sensitivity to the cache line size is also expected in this case.

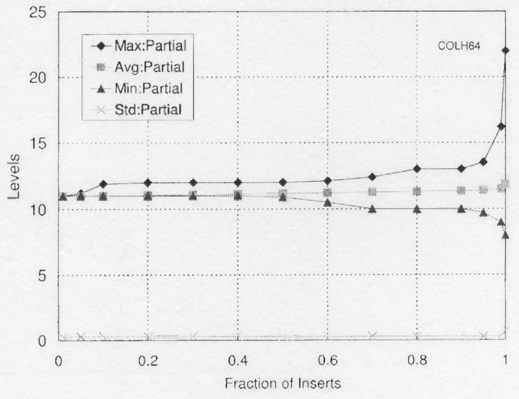
To demonstrate that a sequentially loadable index can outperform traditional multi-dimensional indexing structures, which are accessed one page at a time, the CIPMM framework is proposed, which is a general framework for clustering and indexing using persistent main memory indices. The OP-tree is selected as the main memory index to illustrate the proposed method, which is referred to as CIPOP. A two-phase serialization method is described, which is built via dynamic memory allocation, then serialized by writing it onto a contiguous memory area, and saved and reloaded into main memory incurring a single sequential access. The CIPOP method is compared against CISR in processing k-NN queries is compared and results show that CIPOP outperforms CISR as far as CPU time and calculated elapsed time is concerned.

Since the OP-tree is static, it is extended to be semi-dynamic. There are numerous alternatives in implementing the dynamic index, and three are proposed and evaluated. Point deletion can be implemented by utilizing a bit to indicate whether an entry is deleted or not. The reorganization of index pages can be initiated as a low priority process when the effective utilization of a page drops below a certain threshold. The differential file paradigm [98] can be used to defer the insertion of the new points, so that they can be inserted more efficiently as a batch. Because the two-phase serialization method have to be rebuilt and reserialized each time when new data is inserted, a one-phase serialization method is proposed. In this case only modified pages need to be written to disk. This implementation is quite flexible in dealing with insertions of new points, but many implementation alternatives remain to be investigated.

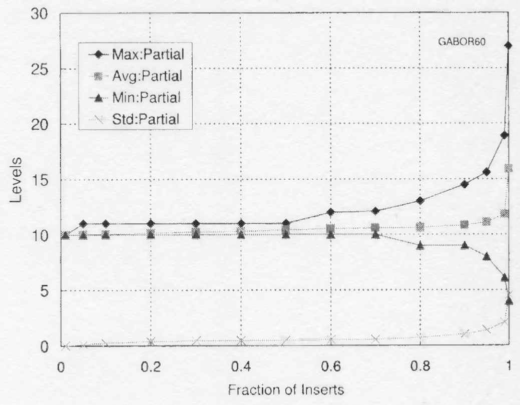
Modern operating systems prefetch the pages of the index, so that CPU and disk access time are heavily overlapped. The calculated elapsed time, which has been a common practice in performance studies of indexing structures, is used.



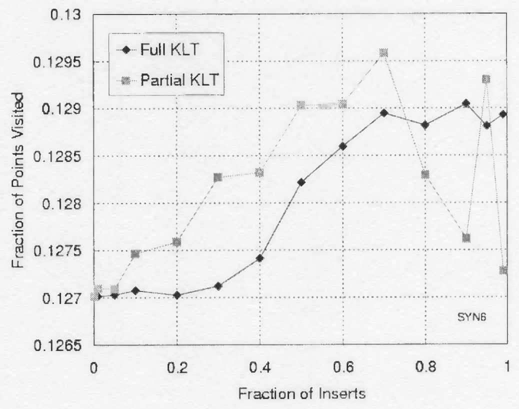
(a)



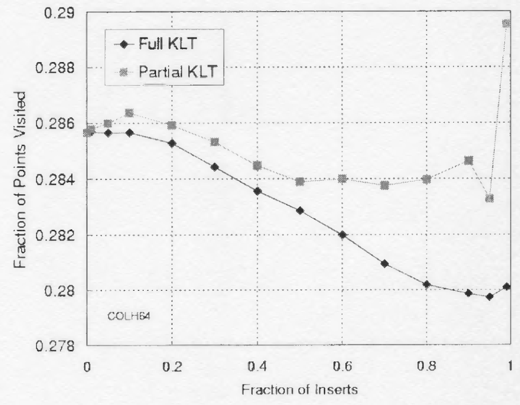
(b)



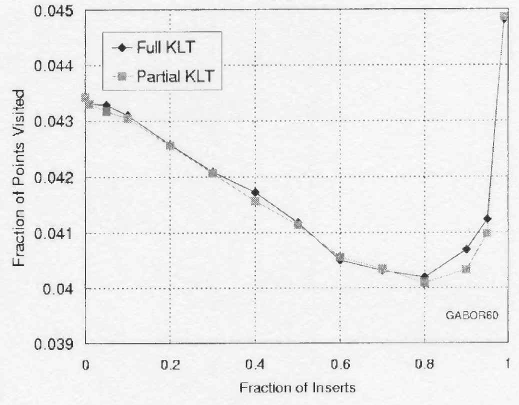
(c)



(d)



(e)



(f)

Figure 4.22 Comparison for different measurements on three datasets for applying partial K-L transform and full K-L transform. (a, b, c) are levels of the trees and their standard deviations. (d, e, f) are percentages of points visited.

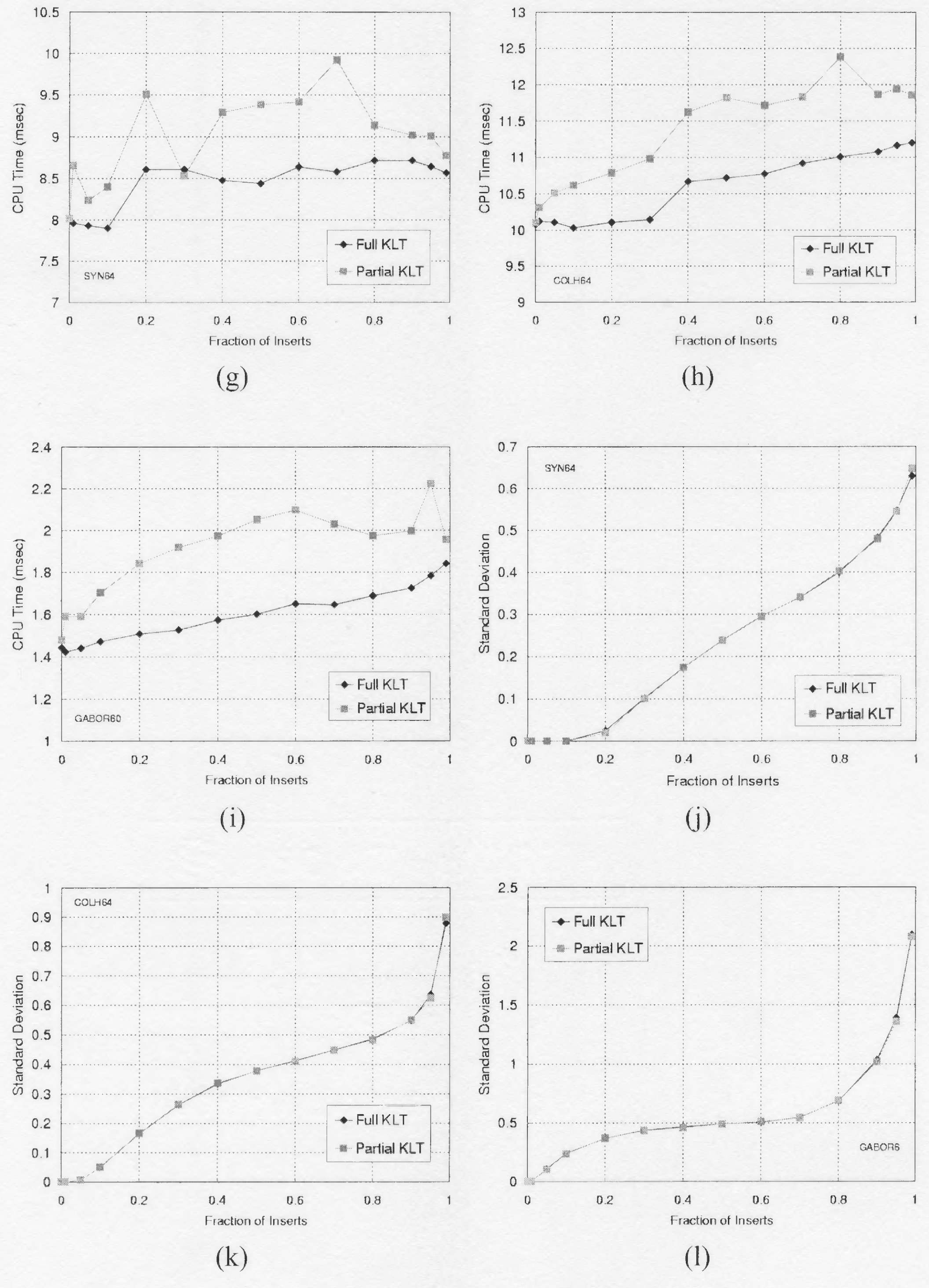


Figure 4.23 Comparison for different measurements on three datasets for applying partial K-L transform and full K-L transform. (g, h, i) are average CPU time for running 1000 20-NN queries. (j, k, l) are standard deviations for the levels of the tree.

CHAPTER 5

THE STEPWISE DIMENSIONALITY INCREASING - SDI INDEX FOR HIGH-DIMENSIONAL DATA

Similarity search is a powerful paradigm for image and multimedia databases, time series databases, and DNA and protein sequence databases. Features of data objects are extracted and transformed into high-dimensional vectors. Object similarity is determined by the distance of the endpoints of these feature vectors and is usually implemented as k-NN queries. The cost of processing k-NN queries via a sequential scan increases with the number of objects and the number of features. Multi-dimensional index structures can be used to proposed to improve the efficiency of k-NN query processing. but lose their effectiveness as the dimensionality increases. The curse of dimensionality manifests itself in the form of increased overlap among the nodes of the index, so that a high fraction of index pages are touched in processing k-NN queries. The increased dimensionality results in a reduced fanout and an increased index height. In this chapter, a *Stepwise Dimensionality Increasing* - SDI-tree index is proposed, which aims at reducing the number of disk accesses and CPU processing cost. The index is built by using feature vectors transformed via principal component analysis, resulting in a structure with fewer dimensions at higher levels and increasing the number of dimensions from one level to the other. Dimensions are retained in nonincreasing order of their variance according to a parameter p, which specifies the incremental fraction of variance at each level of the index. The optimal value for p is determined experimentally. Experiments on three datasets have shown that SDI-trees access fewer disk pages than SR-trees and VAMSR-trees and incur less CPU time than Vector Approximation - VA-Files in addition.

5.1 Introduction

The proliferation of novel database applications has necessitated new algorithms and paradigms. Such applications include content-based image retrieval in image and multimedia databases, sequence similarity search in DNA and protein databases, and similarity matching in time series databases. Objects are represented by N-dimensional feature vectors, where N can be quite high (in the hundreds) and similarity is defined by Euclidean distance or more complex distance functions [91]. In the case of image databases, for example, features are based on color, texture, and shape.

The processing of k-nearest neighbor - k-NN queries on the feature vector space is a popular similarity search paradigm, whose performance is investigated in this study. These queries can be carried out by scanning the dataset of M objects with the N-dimensional feature vectors, but the CPU time and hence the elapsed time for this operation might be unacceptably high for the online processing of k-NN queries. Building a multi-dimensional index on the dataset is a popular method to reduce the cost, which ensures that only points in appropriate neighborhoods are inspected [91].

With the increasing dimensionality of feature vectors, most multi-dimensional indices lose their effectiveness. The so-called dimensionality curse [2] is due to an increased overlap among the nodes of the index and a low fanout, which results in increased index height. For a typical feature vector based hierarchical index structure, each node corresponds to a page. Given a fixed page size S (minus space dedicated to bookkeeping), number of dimensions N, and $s = sizeof(dataType)$, the fanout for different index structures is as follows.

Hyperrectangles R*-trees [48] consist of a hierarchy of hyperrectangles, with higher levels hyperrectangles embedding those at the lower levels. Each hyperrectangle is specified uniquely by the lower left and upper right coordinates of two extreme points positioned diagonally in N-dimensional space or alternatively, by the centroid and its distance from the N "sides" of the hyperrectangle. The cost is the same in both cases, so that the fanout is $F \approx S/(2 * N * s)$.[1]

Hyperspheres Similarity Search - SS-trees consist of a tree of hyperspheres [49], with each node embedding the nodes at the lower level. SS-trees have been shown to outperform R-trees. Each hypersphere is represented by its centroid in N-dimensional space and its radius, i.e., the fanout is $F \approx S/((N+1) * s)$.

Hyperspheres and hyperrectangles Spherical-Rectangular - SR-trees [17] combine SS-trees with R-trees, which encapsulate all of the points in the index. The region of the index is the intersection of the bounding hyperrectangle and the bounding hypersphere. This results in a significant reduction in the size of the region, since the radius of the hypersphere is determined by the distance of the farthest point from its centroid. The space requirement per node is the sum of the space requirements in SS-trees and R* trees, so that the fanout is $F \approx S/((3 * N + 1) * s)$.

The fanout is only five ($F = 5$) for SR-trees with page size $S = 8KB$, $dataType$ which is $double$ ($s = 8$ bytes), and $N = 64$ dimensions. A direct consequence is that the number of pages retrieved grows with the height of the tree: $L = \lceil log_F(B) \rceil$, where B is the number of leaf nodes. A higher L contributes to access cost, although the highest levels of the tree are usually cached in main memory. The *Stepwise Dimensionality Increasing - SDI-tree* indexing structure is the product of dimensionality reduction and hierarchical organization. It uses a reduced number of dimensions at the higher levels of the tree to increase the fanout, so that the tree height is reduced. The number of dimensions increases level by level, until full dimensionality is attained at the lower levels.

The intuition comes from real life situations, where objects are categorized into a few broad classes based on a few features first, but as the classification is further refined, more and more features are added [36]. Each level has the characteristics of the level above it,

[1]An R*-tree might be implemented as a Spatial Access Method - SAM, rather than a Point Access Method - PAM [11], so that points in the leaf nodes are represented as hyperrectangles.

plus some distinguishing features. The differences become negligible at the lower levels of the tree.

Several dimensionality reduction methods have been applied recently in database applications [2]. Principal Component Analysis - PCA [13], Singular Value Decomposition - SVD [33], and Karhunen-Loève Transform - KLT [2] are different ways to achieve the same goal. When all of the objects are known in advance, these methods introduce the least *normalized mean squared error - NMSE*, in transforming the data from N to $n < N$ dimensions [33, 3]. A brief description of these methods is given as part of related work in Section 2.

At the top levels of the index a few principal components with the highest variance are used, but more and more dimensions are included at the lower levels of the tree. All the dimensions are stored at the leaf nodes. In this manner fanouts at the upper levels are large, and more branches can be hopefully pruned by retrieving just one disk page. The inefficiency associated with the curse of dimensionality thus can be lowered. In fact, experiments show that SDI-trees, especially those with carefully tuned parameters, incur fewer disk accesses than SR-trees and VAMSR-trees.

A reduced number of dimensions results in a lower cost for processing k-NN queries, but the search may no longer be exact, i.e., yield a recall [2] below 100% [3]. A postprocessing step to achieve exact k-NN processing given in [80] and extended in [89] is not required for SDI-trees, since the lowest level of the index has all of the dimensions. CPU time is improved due to the fact that shorter vectors can be cached more efficiently [99].

The rest of this chapter is organized as follows. Section 5.2 is a review of related work in this area, followed by a brief review of the SVD and PCA methods. Section 5.3 defines the SDI-tree and its nearest neighbor search algorithm. Section 5.4 evaluates the performance of the SDI-tree and compares it with other index structures. Conclusion and future plans are discussed in Section 5.5.

5.2 Related Work

There has been enormous activity in developing and evaluating multi-dimensional indices in the last two decades [11, 91, 100]. Here two novel index structures are first described, which bear the most similarity to SDI-trees. The mathematics behind the SVD and PCA dimensionality reduction methods are then specified.

5.2.1 Background on Index Structures

The Δ-tree is a memory-resident index structure [99], which addresses the problem of minimizing misses in L2 caches (with 32-128 byte line sizes) as the dimensionality of feature vectors increases. This results in a reduced CPU time. Each level of the index represents the data space starting with a few dimensions and expanding to full dimensions, while keeping the fanout fixed. The nodes of the index increase in size from the highest level to the lower level and the tree may not be height-balanced. This is not a problem since the index is main memory resident. Experiments show that the index reduces the cost of distance calculation exploiting small cache line sizes.

The *Telescopic Vector - TV-tree* is a disk resident index with nodes corresponding to disk pages [36]. TV-trees partition the data space using Telescopic Minimum Bounding Regions - TMBRs, which have telescopic vectors as their centers. These vectors can be contracted and extended dynamically by telescopic functions defined in [36], only if they have the same number of active dimensions (α). Features are ordered using the Karhunen-Loève-transform applied to the whole dataset, so that the first few dimensions provide the most discrimination. The discriminatory power of the index is heavily affected by the value of the parameter α, which is difficult to determine. In case the number of levels is large, the tree will still suffer from the curse of dimensionality. The top levels of TV-trees have higher fanouts, thus reducing the disk I/O cost for disk accesses. Experimental results on a dataset consisting of dictionary words are reported in the paper.

The SDI-tree differs from the Δ-tree in that it is a disk resident index structure with fixed node size, while the Δ-tree is a main memory resident index with variable node sizes and fixed fanouts. The SDI-tree differs from the TV-tree in that it uses a single parameter, specifying the fraction of variance to be added to each level, without the risk of having a large number of active dimensions.

In the experiments, the SDI-tree is compared with the VAMSR-tree and the Vector Approximation - VA-File [18]. The VAMSR-tree uses the same split algorithm as VAM-Split R-tree [101], but it is based on an SR-tree structure, which is statically built in a bottom-up manner. The dataset is recursively split top-down using dimension with the maximum variance and choosing a pivot, which is approximately the median.

The VA-File method represents each data object with the cell to which it belongs. Cells are defined by a multi-dimensional grid, where dimension i is partitioned 2^{b_i} ways. Nearest neighbor queries sequentially scan the VA-File to filter the search space. This is followed by a refinement step, which retrieves the actual objects and returns the nearest neighbors.

5.2.2 Background on Dimensionality Reduction

SVD and PCA are different computational methods to achieve the same goal, i.e., to rotate a dataset onto its principal components, so that optimal dimensionality reduction can be attained by eliminating principal components with the smallest variance. Both methods can be used to transform the feature vectors of the original dataset (say X) into an uncorrelated frame of reference (say Y). The coordinates of Y are in fact the principal components, which without loss of generality are in nonincreasing order of corresponding eigenvalues [2].

Given an $M \times N$ dataset X with M objects each represented by N features, let μ_j denote the mean for column j:

$$\mu_j = \frac{1}{M}\sum_{i=1}^{M} x_{i,j}, \qquad 1 \le j \le N.$$

Let 1_M denote a column vector of length M with all elements equal to 1. SVD decomposes $X - 1_M\mu^T$, whose columns have zero means, as follows:

$$X - 1_M\mu^T = USV^T,$$

U is an $M \times N$ column-orthonormal matrix, S is a $N \times N$ diagonal matrix of singular values, and V is an $N \times N$ unitary matrix of the eigenvectors.

Given that C denotes the covariance matrix of dataset X, PCA decomposes C as follows:

$$C = X^TX/M - \mu\mu^T = V\Lambda V^T,$$

where V is the matrix of eigenvalues and V^T is its transpose. The diagonal matrix Λ contains the eigenvalues of C in nonincreasing order: $\lambda_1 \ge \lambda_2 \ge \cdots \ge \lambda_N$. All the eigenvalues are positive, since the covariance matrix C is positive semi-definite. It is known that $\lambda_j = s_j^2/M,\ 1 \le j \le N$.

The following transformation yields zero-mean, uncorrelated features, which are used for dimensionality reduction and indexing.

$$Y = (X - 1_M\mu^T)V.$$

5.3 Stepwise Dimensionality Increasing - SDI tree

The SDI-tree shown in Figure 5.1 is a disk-resident index with each node corresponding to a disk page, which is suited for high-dimensional indexing. Starting with a few features at the highest level, the number of retained feature vector elements is increased to include all of the dimensions.

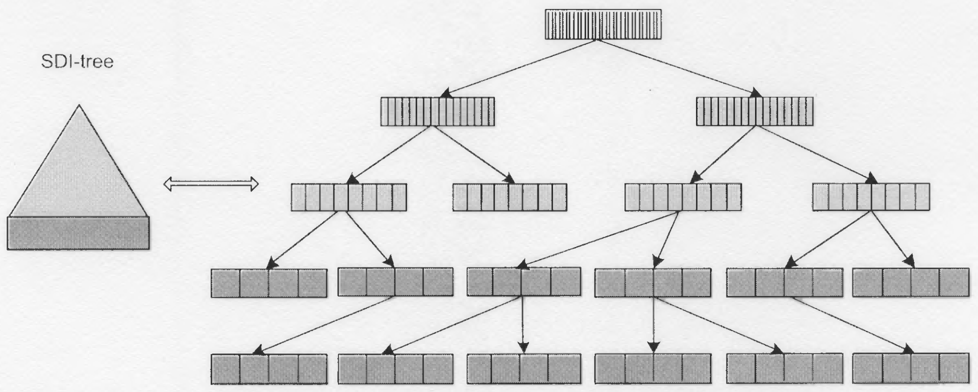


Figure 5.1 The SDI-tree representation.

In this section, the structure of the index is first described, followed by index construction and the algorithm for processing k-NN queries.

5.3.1 The Index Structure

A node of the index is an array of entries as shown in Figure 5.2. The size of each entry is denoted by $EntrySize$, which is a function of the dimensionality. Given that the number of dimensions at level l is $n_l \leq N$ and the page size S, the fanout at level l is: $F_l \approx S/EntrySize(n_l)$. The nodes of the tree are organized as hyperspheres, but unlike the SS-tree [49], both the centroid and the radius is calculated based on n_l dimensions. The number of points covered by this node is used to update the centroid when new data points are inserted.

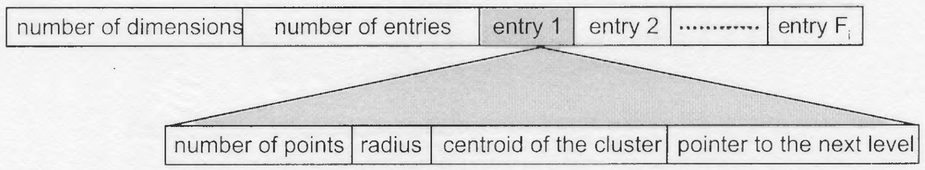


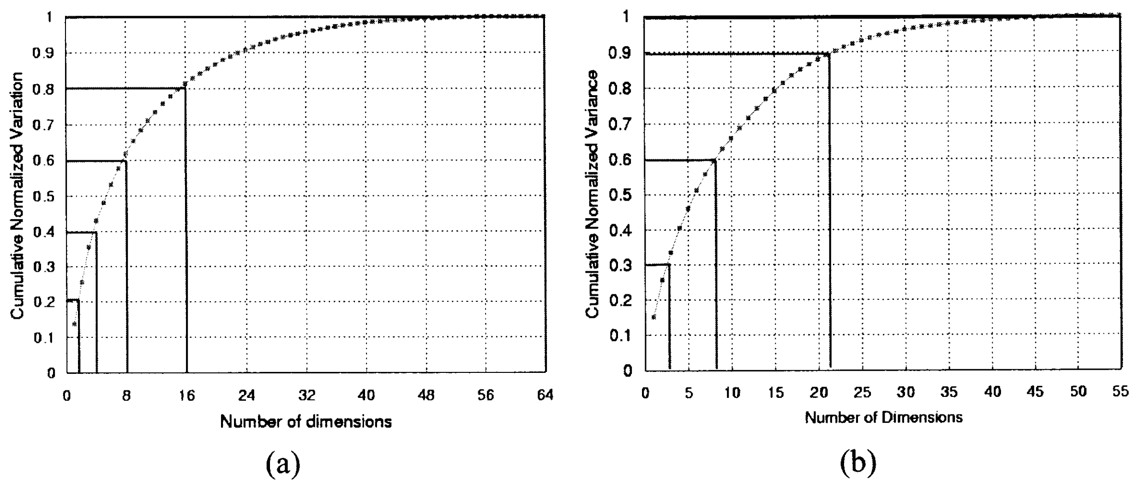
Figure 5.2 Index node structure.

To determine the number of dimensions n_l and the fanout F_l at level l, a parameter p is employed to specify the fraction of variance introduced at successive levels of the index, starting with the highest level and until 100% variance is achieved. At level l ($l \geq 1$), the number of dimensions is selected as the smallest n_l satisfying:

$$\sum_{i=1}^{n_l} \lambda_i / \sum_{i=1}^{N} \lambda_i \geq l \times p, \tag{5.1}$$

where λ_i, $1 \leq i \leq N$, correspond to the eigenvalues of the covariance matrix. The ratio is equal to one for $n_l = N$. Since $\sum_{k=n_l+1}^{N} \lambda_k / \sum_{k=1}^{N} \lambda_k$ is the *Normalized Mean Squared Error - NMSE* [3], this is the error introduced by the index at level l.

Figure 5.3(a) shows the cumulative normalized variance versus the number of dimensions for dataset COLH64. With $p = 0.20$ the number of dimensions at level one through five is given as 2, 4, 8, 16 and 64, respectively. Figure 5.3(b) shows the case for dataset TXT55 with $p = 0.30$. The number of dimensions at level one through four in this case is 2, 8, 21, and 55.



(a) (b)

Figure 5.3 Cumulative variance v.s. number of dimensions. (a) COLH64. (b) TXT55.

Given the fanout F_l at level l, which can be determined simply from the cumulative normalized variation graph for a certain value of p, it seems to be possible to compute the number of levels of the tree. This is impossible in practice, however, since given the number of clusters, clustering algorithms may produce clusters with highly unequal sizes.

Not all leaf nodes are at the same depth, and the height of the tree slightly varies. The method in [99] assigns a target value to the number of levels L of the index and then assign $1/L$ of the variance to each level.

5.3.2 Index Construction

The SDI-tree is constructed by recursively partitioning the dataset Y (in the transformed domain) into F_l clusters based on n_i dimensions at level l. Since hypersphere clusters are preferred, the k-means clustering method [95] is utilized, with the initial set of centroids selected to maximize their pairwise distances, as in [29]. For each subcluster $C_{l,j}$, $j = 1, \dots, F_l$, if it fits in a page, then a leaf node is created. Otherwise, recursively construct the subtree rooted at $C_{l,j}$. The algorithm for constructing the index is described as in Algorithm 5.

Algorithm 5 constructIndex(l, p, $data$)

```
1: if ($data$ fits in one page) then
2:     create a leaf node $\mathcal{L}$; return $\mathcal{L}$;
3: else
4:     find the minimum dimension $n_l$ up to which the calculated variance exceeds $p \times l$;
5:     calculate the fanout $F_l$;
6:     while (average number of points in each cluster < half of the leaf capacity) do
7:         $n_l$++;
8:         calculate the fanout $F_l$;
9:     partition the $data$ into $F_l$ clusters using $n_l$ dimensions;
10:    create an internal node $\mathcal{I}$;
11:    for each subcluster $C$ do
12:        constructIndex($l$+1, $p$, $C$);
13:        fill out the entry in $\mathcal{I}$;
14:    return $\mathcal{I}$;
```

The SDI-tree can be made balanced by generating equal sized clusters. The method proposed in [102] is experimented to ensure that the size of each subcluster does not exceed half of the original cluster. Let C_m denote the subcluster with maximum size, which

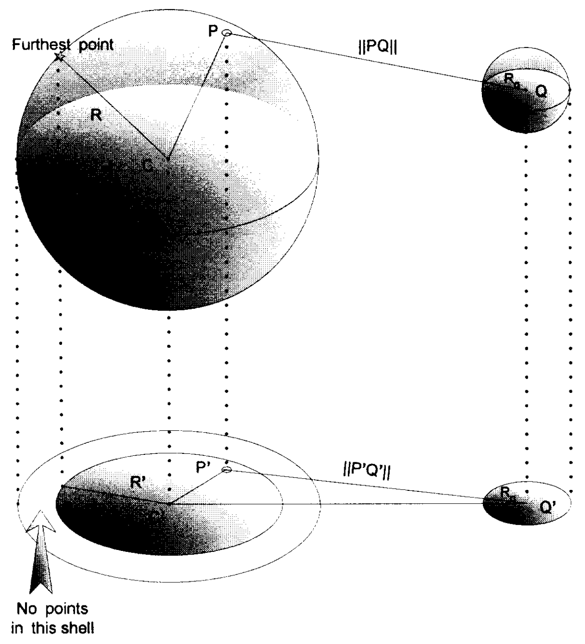


Figure 5.4 Cluster prune with projected distance.

Two priority queues are used: (i) $pque\_index$ is a minimum priority queue for those nodes whose parent has been processed, but itself has not. (ii) $pque\_knn$ is a maximum priority queue with fixed length k for the candidate nearest neighbors. Line 1 transforms the query point Q to Q' according to the eigenmatrix of the dataset. Line 7-8 terminates the search process when the distance from the query to a node (either internal or leaf) is greater than the current search radius. This follows the lower bounding property. Line 11-12 inserts the nodes which have distance less than or equal to the current search radius to $pque\_index$. Line 15-16 inserts the points with distance less than or equal to the current search radius to $pque\_index$. Line 17-18 adjusts the search radius.

5.4 A Performance Study

Three datasets are used in the experiments: COLH64, GABOR60 and TXT55. COLH64 is 68,041 $\times$ 64 color histograms extracted from 68,014 color images[2]. GABOR60 is 56,644 $\times$ 60 Gabor features extracted from MMS (Multimission Modular Spacecraft for Landsat

[2]http://kdd.ics.uci.edu/databases/CorelFeatures

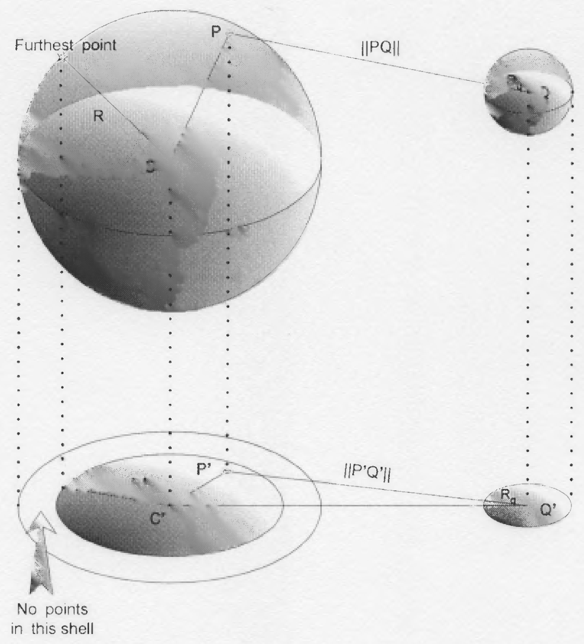


Figure 5.4 Cluster prune with projected distance.

Two priority queues are used: (i) $pque\_index$ is a minimum priority queue for those nodes whose parent has been processed, but itself has not. (ii) $pque\_knn$ is a maximum priority queue with fixed length k for the candidate nearest neighbors. Line 1 transforms the query point Q to Q' according to the eigenmatrix of the dataset. Line 7-8 terminates the search process when the distance from the query to a node (either internal or leaf) is greater than the current search radius. This follows the lower bounding property. Line 11-12 inserts the nodes which have distance less than or equal to the current search radius to $pque\_index$. Line 15-16 inserts the points with distance less than or equal to the current search radius to $pque\_index$. Line 17-18 adjusts the search radius.

5.4 A Performance Study

Three datasets are used in the experiments: COLH64, GABOR60 and TXT55. COLH64 is 68,041 $\times$ 64 color histograms extracted from 68,014 color images[2]. GABOR60 is 56,644 $\times$ 60 Gabor features extracted from MMS (Multimission Modular Spacecraft for Landsat

[2]http://kdd.ics.uci.edu/databases/CorelFeatures

Algorithm 6 knnSearch(Q, $root$, k)

```
1: transform $Q$ to $Q'$ according to the eigenmatrix;
2: PriorityQueue $pque\_index$, $pque\_knn$;
3: insert (0, $root$) into $pque\_index$;
4: $R_q = \infty$;                                          // initialize the search radius
5: while ($pque\_index$ is not empty) do
6:     $top = pque\_index$.extractMin();
7:     if ($top.dist > R_q$) then
8:         break;
9:     if ($top.node$ is an internal node) then
10:        for each child $c$ of $top.node$ do
11:            if (dist_s($Q'$, $c$)-$R' \leq R_q$) then             // distance in subspace
12:                insert (dist_s($Q'$, $c$)-$R'$, $c$) into $pque\_index$;
13:    else                                                  // $top.node$ is a leaf node
14:        for each point $P$ in the $top.node$ do
15:            if (dist($Q$, $P$) $\leq R_q$) then
16:                insert (dist($Q$, $P$), $P$) into $pque\_knn$;     // distance in origin space
17:                if ($pque\_knn$ is full) then              // $pque\_knn$ has fixed length $k$
18:                    update $R_q$;
```

4) images from different parts of the country. TXT55 is 79,814 $\times$ 55 Gabor, spatial, and wavelet features from 400 photos.

The average disk accesses of one thousand 20-nearest-neighbor queries are measured for SDI-trees, SR-trees and VAMSR-trees. The queries are randomly selected without replacement from the datasets. The page size is 8KB in all cases. The split factor and reinsert factor for the SR-tree is 40% and 30%, respectively. Given that the already introduced nearest neighbor search algorithm for SDI-tree is based on the HS algorithm and this algorithm accesses as few pages as theoretically possible [52], to make the comparison fair the HS algorithm for SR-tree is implemented, instead of the embedded RKV algorithm provided at [72]. The SR-tree is used as a reference since it outperforms SS-tree and R*-tree [17].

The variance increment p affects the number of levels. Experimental results show that the number of levels differ at most by two levels. This is also the difference between the maximum and minimum number of levels of the tree in the case of datasets used in this experiments. When $p < 0.3$, the number of disk accesses is very high, so in Figure 5.5 the number of disk accesses versus p ranging from 0.3 to 1 is plotted. The minimum number of disk accesses is achieved at $p = 0.4$, where the height of the tree ranges from 3 to 5. The graph for the Gabor dataset is not plotted,since p has little effect on the number of disk accesses, although $p = 0.8$ yields a slightly better performance.

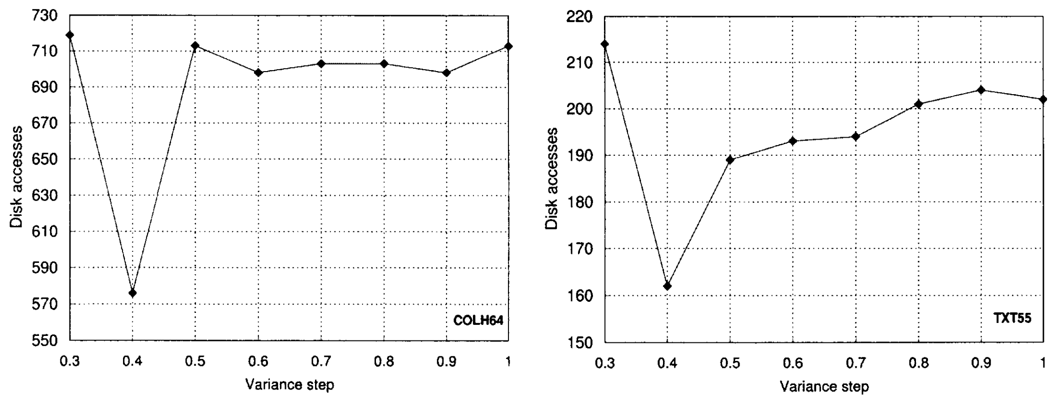


Figure 5.5 Number of disk accesses versus variance increment step (p).

Figure 5.6 compares the performance of SDI-trees, SR-trees and VAMSR-trees[3]. The SDI-tree has fewer levels, but more nodes than the other two indices. The number of nodes of the SDI-tree can be greatly reduced, to as much as one-fifth of the original size, by enforcing the full utilization of the leaf nodes. This can be achieved by sorting the data along the maximum variance dimension and assigning the points one by one to the leaf nodes. Unfortunately the overlaps increase, resulting in a significant degradation in search performance (disk accesses and number of points visited). Since application search efficiency is more important than space on modern disk drives, the full utilization implementation was not adopted.

In processing k-NN queries, the SDI-tree visits less internal nodes and leaf nodes, which leads to fewer disk accesses and fewer points checked. In terms of the total disk ac-

[3]http://research.nii.ac.jp/ katayama/homepage/research/srtree/

cesses, Figure 5.6 shows that the SDI-tree improves 14% over SR-tree, 20% over VAMSR-tree for COLH64 and 41% over SR-tree, 35% over VAMSR-tree for TXT55.

Dataset	Index	Variance step	Internal created	Leaf created	Level	Internal visited	Leaf visited	Points visited	Disk accesses
	SDI-tree	0.4	1697	27484	4	*262*	*314*	*2333*	*576*
COLH64	SR-tree		2198	5823	7	434	232	2900	666
	VAMSR-tree		1138	4537	6	343	378	5660	721
	SDI-tree	0.4	1512	26979	5	*89*	*73*	*620*	*162*
TXT55	SR-tree		1794	5934	6	210	64	943	274
	VAMSR-tree		890	4435	5	122	126	2258	248
	SDI-tree	0.8	1264	19053	4	*8*	*13*	*60*	*21*
GABOR60	SR-tree		2121	5124	7	22	7	80	29
	VAMSR-tree		889	3541	6	30	11	171	41

Figure 5.6 Performance comparison of 20-NN searches for SDI-trees, SR-trees and VAMSR-trees.

The k-NN search performance has also been studied over datasets with varying number of dimensions, where after the transformation onto the principal components, the dimensions with the highest variability are retained. Figure 5.7 depicts the detailed search performance in terms of page accesses, points visited, floating point operations and the CPU time for TXT55. For the SDI-tree the value of p providing the optimal performance is selected.

The SDI-tree requires less CPU time than the other two methods, since it always touches fewer points and hence incurs fewer floating point operations. Another reason why it requires less CPU time is that it deals with shorter vectors. Since the SDI-tree also accesses fewer pages, it outperforms the other two methods in terms of the elapsed time, which includes disk access time. The SDI-tree is suitable for high-dimensional data, since with the dimensionality increasing, the gap between SDI-tree and other methods widens.

The SDI-tree is also compared with the VA-File [18]. Although the VA-File visits fewer points in the dataset with the original precision, it has to sequentially scan the whole approximation file, which contributes to CPU time. The performance (number of points visited, floating point operations ,and search time) of the VA-File is highly dependent on the number of bits (b_i) per dimension. b_i is varied and the best one ($b_i = 4$ for all dimensions) is chosen in the experiments. The search time is plotted in Figure 5.7(d) and it is observed that the SDI-tree outperforms the VA-File.

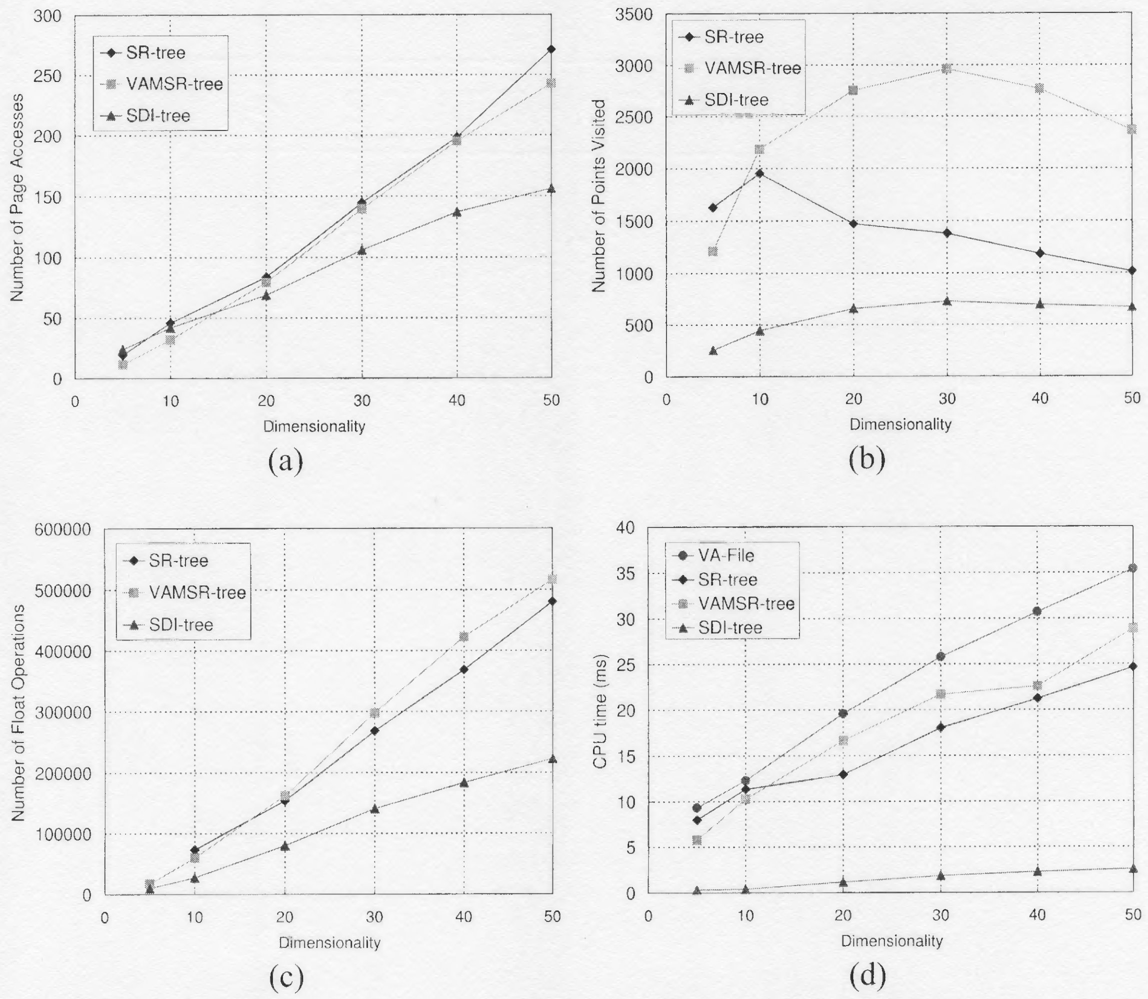


Figure 5.7 Details of k-NN search performance with $k = 20$ versus dimensionality over 1000 randomly chosen queries.

5.5 Conclusions and Future Plans

The SDI tree assigns a variable number of dimensions to the successive levels of an index, whose nodes are spherical and are obtained using the k-means clustering method. The dataset to be indexed is subjected to principal component analysis, the dimensions are ordered in decreasing order of their variance, and a fraction p of the variance of the dataset is assigned to each level.

Experimental results with three datasets show that SDI-trees with carefully tuned parameters access fewer pages from disk, visit fewer points and incur fewer floating point operations, resulting in less CPU time than the SR-tree and VAMSR-tree. The SDI-tree also outperforms the VA-File in terms of CPU time. Combining the fact that it accesses less pages, the elapsed time (including disk IOs) will also outperforms the other two. The SDI-tree is especially suitable for high-dimensional data, since with the dimensionality increasing, the gap between SDI-tree and other indices widens.

Real world data is usually correlated either globally or locally [14, 3]. Global correlation means most of the variance in the dataset can be captured by a few principal components. It may be that global correlation does not exist and there are subsets of data that are locally correlated. In the future, elliptical clustering [103] will be applied before building the index which may be called clustered SDI - CSDI. The performance gains of the CSDI method will be studied. Furthermore, since both the Δ-tree and SDI-tree are static indices, methods to make them dynamic will be investigated. Other indexing structures will also be used as references for comparison.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

High-dimensional indexing is indispensable to facilitate similarity search for modern database applications. Due to the curse of dimensionality, the efficiency of indexing methods degrades. In this dissertation, the inefficiency is dealt with from three viewpoints.

- Dimensionality Reduction

  Singular Value Decomposition - SVD can be used to reduce dimensionality for globally correlated datasets. Clustering and SVD (CSVD) can achieve higher dimensionality reduction for datasets with local correlations. Exact nearest neighbor search is difficult to obtain when the dimensionality is high. An approximate nearest neighbor search algorithm has been proposed by using CSVD in [3] with minimum loss of distance information. The performance of R*-trees, SR-trees, and hybrid trees is compared from the viewpoint of CPU time and disk accesses. The SR-tree, with the best overall performance, is selected as the within cluster index of the CSVD method. The performance of CSVD with indexing has been studied in this dissertation. The larger the *NMSE*, the few number of dimensions retained. Given an *NMSE*, the higher the degree of clustering, the higher the recall. However, more clusters require more disk page accesses. Experiments on two datasets show that certain number of clusters can be obtained to achieve a higher recall while maintaining a relatively lower query processing cost.

- Persistent Main Memory Index

  Multi-dimensional index can be classified into disk resident index and main memory resident index. Disk resident indices aim at minimizing the number of disk accesses for query processing, while main memory resident indices focus on reducing the

CPU time for query processing. Since sequentially accessing large chunks of data is tantamount to randomly accessing multiple small pieces of data and the idea that a sequentially loadable index can outperform traditional multi-dimensional indexing structures, which are accessed one page at a time, a general framework, *Clustering and Indexing using Persistent Main Memory - CIPMM*, is proposed. A specific instance *CIPOP*, indexing using the persistent OP-tree, is elaborated and evaluated. Experiments show that *CIPOP* outperforms *CISR*, clustering and indexing using the SR-tree, in terms of CPU time and calculated elapsed time for k-NN queries. Since the OP-tree is static, semi-dynamic methods have been investigated and a serialization method supporting insertions have also been proposed.

- Dimensionality Varying Index

  Multi-dimensional index usually has fixed fanout and dimensionality. The fanout decreases monotonically with the dimensionality increasing and lower fanouts increase levels of the index. Dimensionality can be varied from top levels of the index to lower levels starting with dimensions having larger variances, which can be achieved by applying *SVD* or *PCA (Principal Component Analysis)*. A *Stepwise Dimensionality Increasing - SDI-tree* is proposed. Branches can be pruned as early as possible since dimensions at top levels contribute most to the Euclidean distance. Experiments on three datasets show that the SDI-tree with carefully tuned parameters has fewer disk accesses, visits fewer points, and incurs fewer floating point operations, resulting in less CPU time than the SR-tree and VAMSR-tree. This fact implies that the SDI-tree has less elapsed time than the other two. It also outperforms the VA-File in terms of CPU time. The SDI-tree is especially suitable for high-dimensional data, since with the dimensionality increasing, the gap between SDI-tree and other indices widens.

As far as future work is concerned, the k-means clustering method needs to be compared with other methods, especially elliptical clustering [103]. More work remains to be done on the effectiveness of the point insertion methods into the OP-tree. Performance of SDI-trees is expected to further improve if clustering is introduced prior to build the index. More recently proposed high-dimensional indexing methods, such as iDistance [56], are to be investigated.

APPENDIX A

QUERY TYPES

There are six types of queries in high-dimensional applications [52]. Let $D(P, Q)$ be the distance between point P and Q, the queries can be defined as follows.

Definition A.1 (Point Query) *Given a query point Q, find if there is a point P such that $D(P, Q) = 0$.*

Definition A.2 (Window Query) *Given a rectangular region in the data space, find all points falling within the region. The specified hyperrectangle is always parallel to the axis.*

Definition A.3 (Range/Spherical Query) *Given a query point Q and a maximum search distance r, select all the points P, such that $D(P, Q) \leq r$.*

Definition A.4 (Nearest Neighbor Query) *Find a point P in the dataset which is the closest point to the query point Q.*

Definition A.5 (k-Nearest Neighbor Query) *Given a query point Q and an integer $k \geq 1$, select k points $knn(Q, k)$ in a dataset X, which have the shortest distance from Q.*

$$\begin{aligned} knn(Q,k) = \{&P_0, \cdots, P_{k-1} \in X | \\ &\neg\exists o \in X - \{P_0, \cdots, P_{k-1}\} \wedge \neg\exists i \in \{0, \cdots, k-1\} : D(P_i, Q) > D(o, Q)\}. \end{aligned}$$

Definition A.6 (Approximate Nearest Neighbor Query) *Find points which are not much farther away from the query point Q than the exact nearest neighbor.*

Above queries can be generally referred to as similarity search, which is an important operation in high-dimensional applications. Since a meaningful hyperrectangle for window queries and value for r in spherical queries is difficult to specify, all types of nearest neighbor queries are more interesting. Approximate queries are useful since similarity in real world applications is not an exact concept and approximate queries can reduce the processing cost.

APPENDIX B

CHARACTERISTICS OF HIGH-DIMENSIONAL SPACE

High-dimensional space has its distinctive characteristics which result in multi-dimensional indices to lose their efficiency when the number of dimensions is relatively high. The following are some of the characteristics.

Lack of Imagination Given d-dimensional cubic data space $[0,1]^d$ and the centroid $c = (0.5, \ldots, 0.5)$, Lemma in [52] "Every N-dimensional sphere touching (or intersecting) all the (N-1)-dimensional boundaries of the data space also contains the centroid c." seems sound. Actually it does not held when the dimensionality is high. Consider a 16-dimensional sphere centered at $p = (0.3, \ldots, 0.3)$ with radius 0.7, it can not contain the centroid c even it touches all 15-dimensional surface since the (Euclidean) distance between p and c is 0.8.

Sparsity Due to the exponential growth of the volume, data space in high dimension is sparsely populated. The probability that a point lying within a window query with side w is: $P(w) = w^N$ in the d-dimensional space $[0,1]^N$. When the dimensionality is high, even with very large w, the window query is not likely to contain a point, e.g., a window query with $w = 0.95$ only selects 0.59% of the data points in a uniform unit data space for N = 100. Similar effect can be observed for spherical queries. The probability that an arbitrary point lies inside the largest possible sphere (radius r) within a hyper-cube whose sides are $2r$ is given as:

$$P[r] = \frac{\pi^{\frac{N}{2}}}{\Gamma(\frac{N}{2}+1)} r^N,$$

where the Γ function is defined as:

$$\Gamma(\frac{N}{2}+1) = \begin{cases} (\frac{N}{2})! & if\ N\ is\ even \\ \frac{\sqrt{\pi}N!}{2^N(\frac{N-1}{2})!} & if\ N\ is\ odd \end{cases} .$$

The probability of a point lying within a window query ($w = 0.5$) and range query ($r = 0.5$) versus different dimensions is depicted in Figure B.1. It can be seen that the probability decreases sharply as the number of dimensions increases [18].

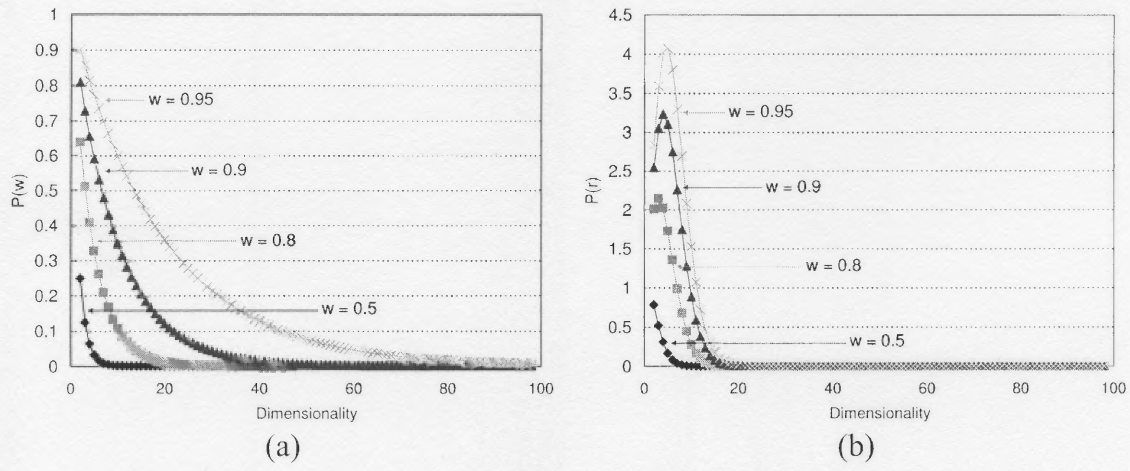


Figure B.1 (a) Window query. (b) Range query. Both are in N-dimensional space $[0, 1]^N$.

Since space organizing techniques index the whole domain space, a query window may overlap part of a page that actually contains no points at all, which is the "dead space" indexing problem [52].

The Surface is Everything The index structure usually splits the data space using (N-1)-dimensional hyperplanes [52]. It recursively selects a split dimension and chooses a split value along that dimension until the number of data items can be held in a data page. The whole process can be described as a split tree which is actually a binary tree. The number of split dimensions for a given data page is on the average:

$$N_s = \log_2 (\frac{M}{C(N)}),$$

where M is the number of data items, C(N) is the capacity of a data page. If all dimensions are equally used as split dimensions (e.g. uniformly distributed data), a data page can be split at most once or twice in each dimension. Thus, the majority of the data pages are located at the surface of the data space. In other words, the probability of a point is closer than 0.1 to an (N-1)-dimensional surface is increasing sharply with the dimensionality increasing, which is shown in Figure B.2.

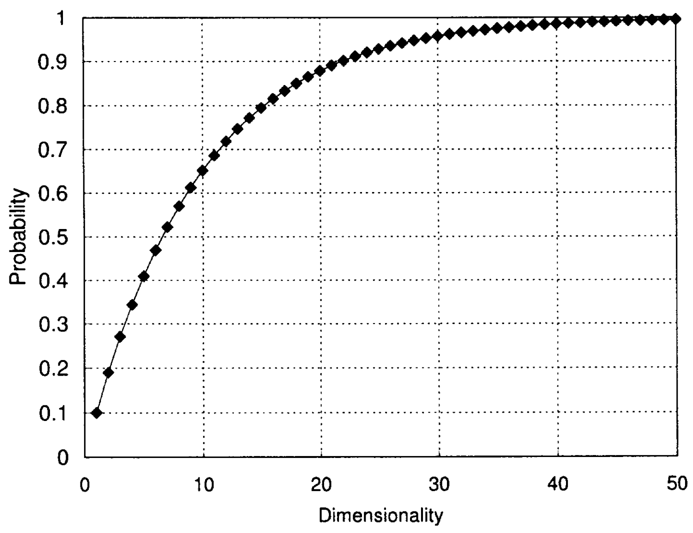


Figure B.2 The probability of a point closer than 0.1 to an (N-1)-dimensional surface.

Indistinctive Nearest Neighbors Since high-dimensional space has high degree of freedom, all of the points seem to be at similar distance to a given point such that no significant difference exists. For example, when points are uniformly distributed in a unit hypercube, the distance between two points is almost the same for any combination of two points. The distance to the nearest data approaches the distance to the farthest data point as dimensionality increases [54]. A new nearest neighbor search algorithm which determines the distinctiveness of the nearest neighbors during search operation is proposed in [104]. Indistinctive nearest neighbors are more likely to occur as dimensionality increases. When M N-dimensional points are distributed uniformly within the hypersphere centered at the query point with radius R, the expected distance to k-th nearest neighbor d_k is given as follows [104]:

$$E\{d_k\} \approx \frac{\Gamma(k+1/N)}{\Gamma(k)} \frac{\Gamma(M+1)}{\Gamma(M+1+1/N)} R, \tag{B.1}$$

The ratio of the $(k+1)^{th}$-NN distance to the k^{th}-NN distance can be obtained[105] as in Equation B.2, which indicates that the ratio decreases monotonically as the dimensionality increases for high-dimensional uniform distribution.

$$\frac{E\{d_{(k+1)}\}}{E\{d_k\}} \approx 1 + \frac{1}{kN}. \tag{B.2}$$

This effect of dimensionality on nearest neighbors causes that the NN search performance degrades, since many points have almost the same similarity with the nearest neighbor and NN search operation is forced to examine many points before determining the true nearest neighbor [104].

Overlap The overlap can be defined as follows[32]:

Definition B.1 (Overlap) *The overlap of an R-tree node is the percentage of space covered by more than one hyperrectangle. If the R-tree node contains n hyperrectangles the overlap is*

$$Overlap = \frac{Vol(\bigcup_{i,j \in 1 \cdots n, i \neq j} (R_i \cap R_j))}{Vol(\bigcup_{i \in 1 \cdots n} R_i)}$$

Definition B.2 (Weighted Overlap) *The weighted overlap of an R-tree node is the percentage of data objects that fall in the overlapping portion of the space.*

$$WeightedOverlap = \frac{|\{p | p \in \bigcup_{i,j \in 1 \cdots n, i \neq j} (R_i \cap R_j)\}|}{|\{p | p \in \bigcup_{i \in 1 \cdots n} R_i\}|}$$

When query points are expected to be uniformly distributed over the data space, Definition B.1 is an appropriate measure. When the distribution of queries corresponds to the distribution of the data and is nonuniform, Definition B.2 is more appropriate. Experiments in [32] show that the overlap of the bounding boxes in the R*-tree directory is rapidly increasing to about 90% when increasing the dimensionality to five. Note that overlap is not an R-tree specific problem, but a general problem in indexing high-dimensional data.

All of the above effects lead to the so-called "curse of dimensionality" i.e., the performance of indexing methods deteriorates when going to higher dimensions. Simple sequential scan can outperform the indexing methods beyond 10-15 dimensions [54].

APPENDIX C

THE OP-TREE VERSUS THE OMNI-FAMILY

C.1 The OMNI-Family

The OMNI-Family [58] reduces distance calculations according to a set of focal points. The focal points are selected as orthogonal as possible to minimize the search space. The distances between each focal point and all the data points are calculated. Given a query point q and a search radius r, the search space can be pruned to be $[d_{q,f_i} - r, d_{q,f_i} + r]$, where d_{q,f_i} is the distance between q and focal point f_i. Given F focal points, the search space is reduced to be $\cap_{i=1}^{F}[d_{q,f_i} - r, d_{q,f_i} + r]$. Since the focal points are preselected, one more extra focal point will maximize the average reduction if they are far apart and equally distant from each other. So a good number of focal points would be between $\lceil \hat{D} \rceil + 1$ and $2 * \lceil \hat{D} \rceil + 1$, where $\hat{D}$ is the intrinsic dimension. The correlation fractal dimension can be used as an approximation of the intrinsic dimension for a dataset. The linear algorithm in [78] to estimate the correlation fractal dimension is used in this experiment. Results are shown in Table C.1.

Table C.1 Fractal Dimensions

	SYN64	COLH64	GABOR60	TXT55
OrigDim	64	64	60	55
FracDim	2.3	4.2	6.0	1.6

To find the range $\cap_{i=1}^{F}[d_{q,f_i} - r, d_{q,f_i} + r]$, several alternative methods can be applied: sequential scan, B+-trees, and R-trees. In the case of B+-trees, a B+-tree is built for each focal point, the intersection of the range search results on each tree is the desired range. In the case of R-trees, the distances between each point and all the focal points form a F dimensional data, and a R-tree is built for this multi-dimensional data with pointers to the original data stored on the leaf nodes. The k nearest neighbor search can be applied directly on the R-tree. The prune distance is decreased whenever a new closer candidate is found based on the original data.

C.2 Experimental Results

The OMNI sequential scan is implemented and compared with the the standard sequential scan and the OP-tree in this section. To perform k-NN search on the OMNI sequential scan, a range search with an initial radius, which is estimated using fractal dimensions as in [78], is issued, then another range search with the distance to the current k-th nearest neighbor is performed.

Number of Focal Points Selection The CPU time for 1000 queries is measured versus variable number of focal points as in Figure C.1. The CPU time is first decreased, then increased with the number of focal points increasing. The minimum CPU time occurs in the range of [4, 7] for SYN64 which has fractal dimension 2.3 as shown in Table C.1. This range matches the $\lceil \hat{D} \rceil + 1$, $2 * \lceil \hat{D} \rceil + 1$ as mentioned above. Four is used as the number of focal points in the following experiments.

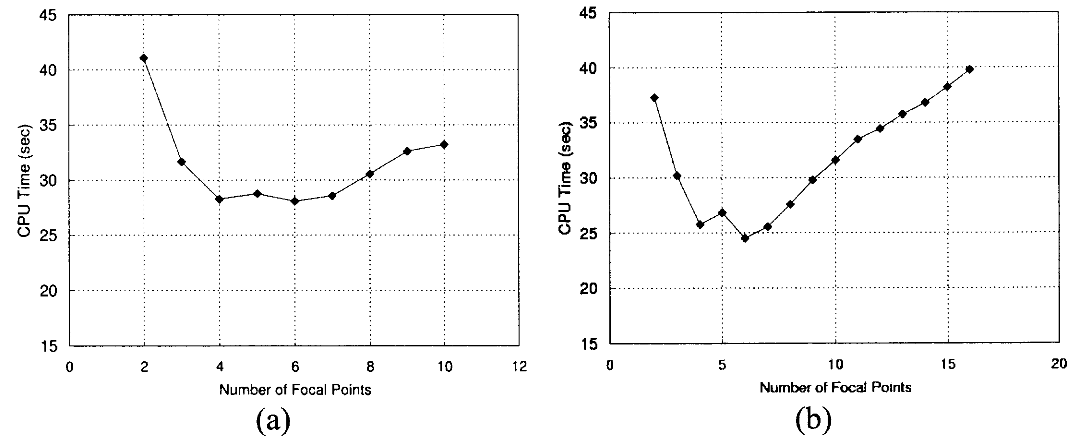
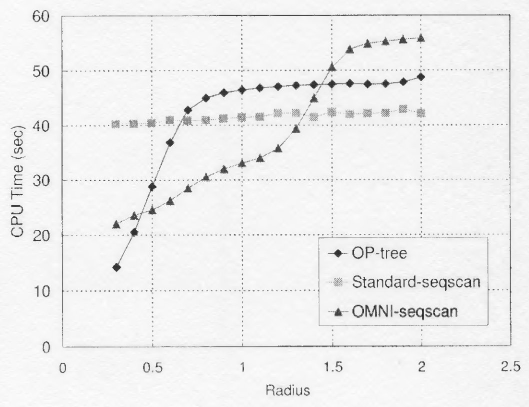
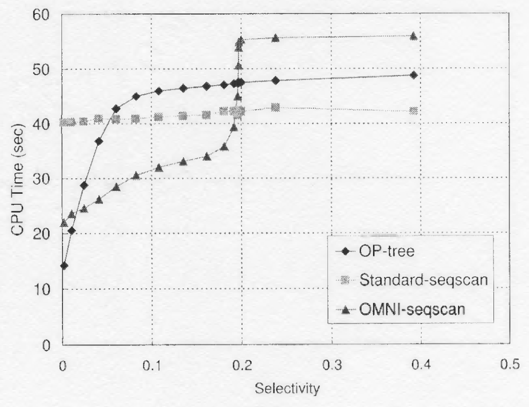


Figure C.1 CPU time versus number of focal points. (a) k-NN query with $k = 20$. (b) Range query with radius = 0.5 and selectivity 2.4%.

Range Query The OP-tree, standard and OMNI sequential scan are compared for range query as in Figure C.2. It shows that the OP-tree outperforms the other two when the selectivity is relatively low. This is due to the highly overlapped property in high-dimensional space, which leads to multiple search paths being visited. The OMNI sequential scan is better than standard sequential scan until a relatively high selectivity (e.g. 20%). The selectivity is obtained from the corresponding radius.



(a)



(b)

Figure C.2 Range query. (a) CPU time versus the radius. (b) CPU time versus the selectivity.

k-NN Search The OP-tree is also compared with the standard and OMNI sequential scan for k-NN queries. All of the three methods are insensitive to the number of nearest neighbors. The OP-tree performs consistently better than the other two methods.

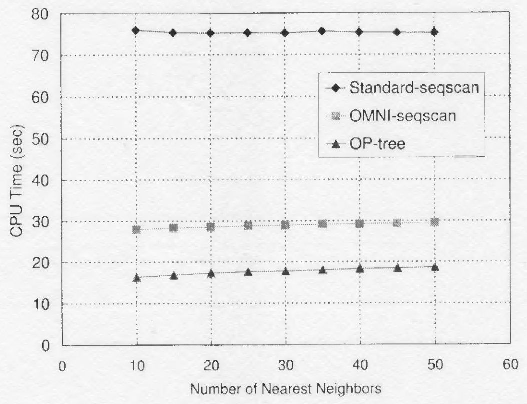


Figure C.3 The CPU time versus k for k-NN queries for three methods.

APPENDIX D

PERFORMANCE COMPARISON OF LOCAL DIMENSIONALITY REDUCTION METHODS

In this section, the performance of CSVD and LDR [14] is compared with indexing structures built for the clusters with SR-trees and hybrid trees. In both cases the page size is 8KB. The SR-tree is chosen because it is reported that the SR-tree outperforms both the SS-tree and the R*-tree in [17]. The hybrid tree is chosen because it has been used in conjunction with the LDR method [14].

Experimental setup. First, the LDR clustering method is used to generate four sets of partitions and the NMSE for each set of partitions is calculated. Then the k-means algorithm is used to cluster the whole dataset into the same number of partitions for each set. Next according to the NMSEs in the first step, the GM1 method [83] is used to reduce the dimensionality. Finally, an index is built for each partition. Both Figure D.1 (a) and Figure D.2 (a) show that index sizes for CSVD generated clusters are smaller than those for LDR generated clusters.

Results for 20-nearest-neighbor search. The SR-tree is used as the within cluster indexing structure, with split factor 0.4 and reinsert factor 0.3, and 1000 randomly generated 20-nearest-neighbor queries are performed for CSVD and LDR. The average recall, number of pages visited and elapsed system time for each query are obtained. Figure D.1 shows that CSVD has higher recall, accesses a fewer number of pages, and incurs less elapsed system time on the average.

The hybrid tree is also used as the within cluster indexing structure and carried out the same experiments as for the SR-tree. This experiment is LDR favored since hybrid tree is used by the original LDR paper [14]). Figure D.2 leads to the same observation. The figure for recall is omitted since it is independent of indexing structure, and is the same as

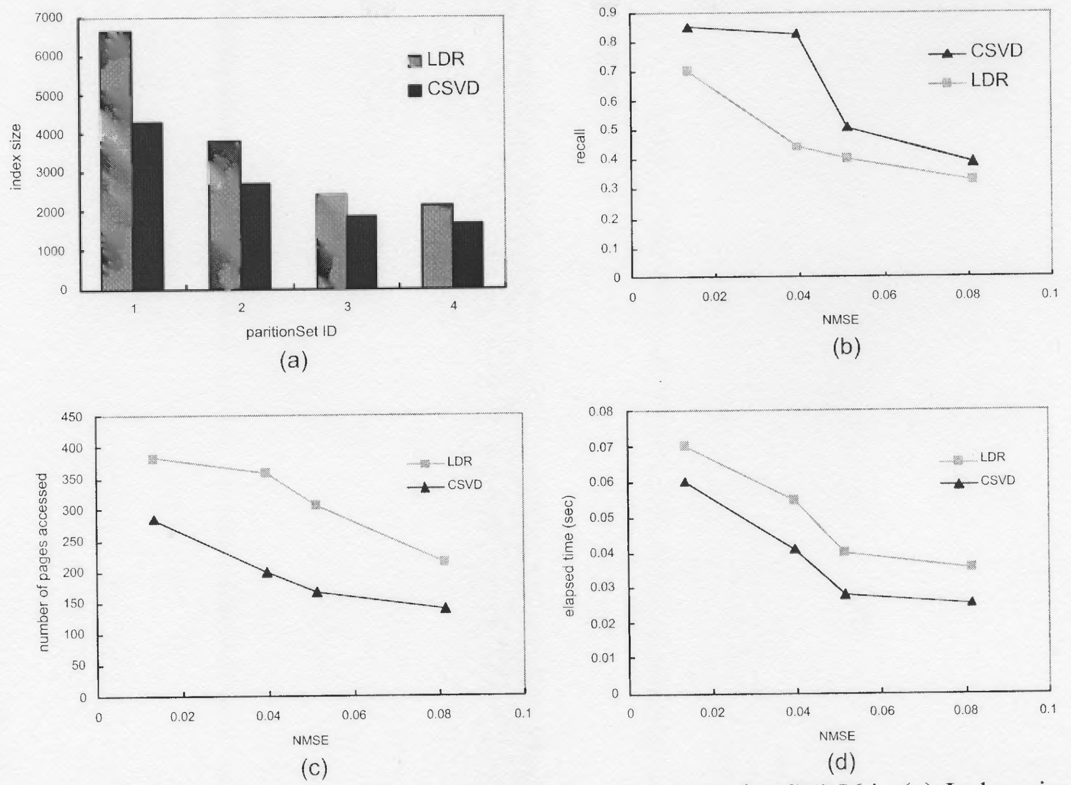


Figure D.1 Results of using SR-tree as indexing structure for SYN64. (a) Index size for four sets of partitions. (b) Recall versus NMSE. (c) Average number of pages accessed versus NMSE. (d) Average elapsed system time versus NMSE. (b, c, d) The average for 1000 20-nearest-neighbor searches.

figure D.1 (b). The elapsed system time per query is much longer (several seconds), which is because the index is implemented to require load from disk before querying on it.

In conclusion, given an NMSE, CSVD incurs less query processing cost while keeping higher recalls than LDR.

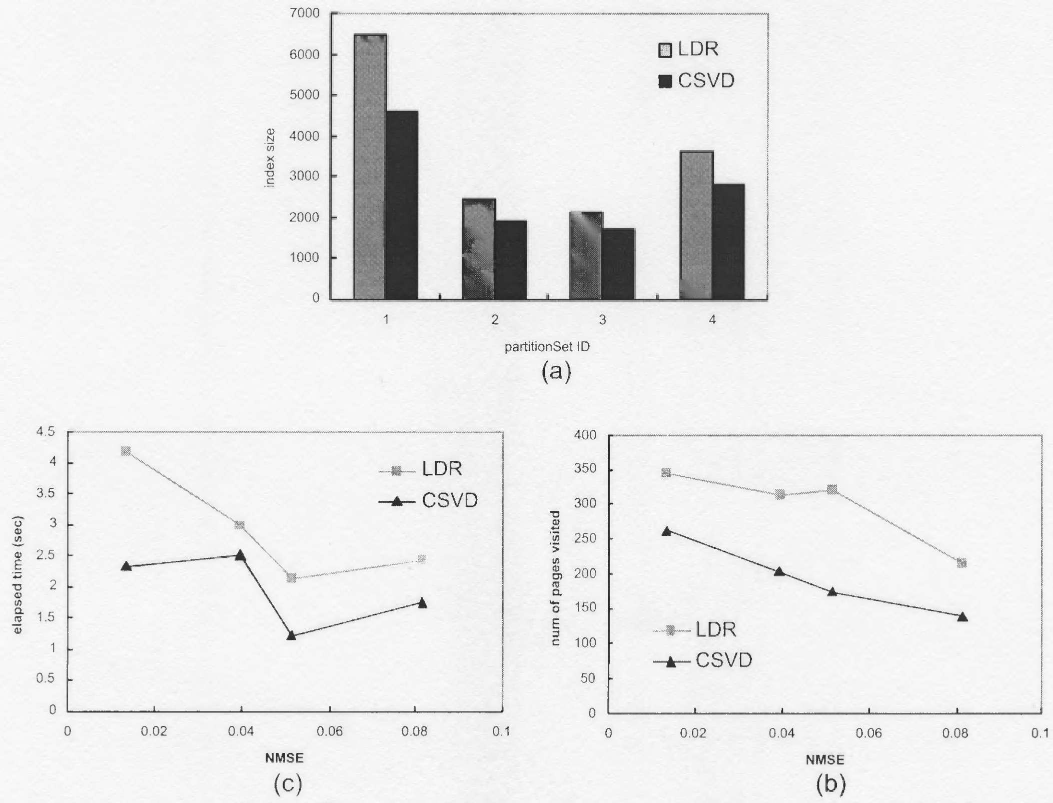


Figure D.2 Results of using the hybrid tree as indexing structure for SYN64. (a)Index size for four sets of partitions. (b) Average number of pages accessed versus NMSE. (c) Average elapsed system time versus NMSE. All the measurements are the average for 1000 20-nearest-neighbor searches.

APPENDIX E

VAMSPLIT R-TREE CREATION

```
REntry CreateVAMSRTree(FVectPtr start, FVectPtr end)
{
    REntry out_entry, *entries;

    entries = new REntry[INTERNAL_FANOUT*MAX_TREE_LEVELS];
    BuildVAMSRTree(start, end, entries, -1);              // level= = -1 means always return root of tree
    out_entry = entries[0];                                      // root is returned in first entry
    delete[] entries;
    return out entry;
}

int BuildVAMSRTree(FVectPtr start, FVectPtr end, REntry* entries, int level)
{
    int size, child_level, cscap, lo_size, lo_entries, hi_entries, out_entries;

    size = end - start; // STL convention: end points to the location after the last element
    if (size ≤ BUCKET_SIZE) {
        entries[0] = CreateRNodeBucket(start, end);
        return 1;
    }
    // Calculate b(s_p) AKA cscap, the child subtrees' capacity
    if (size ≤ 2*BUCKET_SIZE) {
        child_level = 0;
        cscap = 1;
    } else {
        // It would be faster to use a lookup table of possible cscap values...
        child_level = (int)((log(size/(2*BUCKET_SIZE)))/LOG_INTERNAL_FANOUT);
        cscap = (int)(BUCKET_SIZE*pow(INTERNAL_FANOUT, child_level));
    }
    lo_size = SplitDataset(start, end, size, cscap);
    lo_entries = BuildVAMSRTree(start, start+lo_size, entries, child_level);
    hi_entries = BuildVAMSRTree(start+lo_size, end, entries+lo_entries, child_level);
    out_entries = lo_entries + hi_entries;

    // Create a new node, if needed
    if (level == -1 || child_level < level) {
        entries[0] = CreateRNodeFromEntries(entries, out_entries);
        out_entries = 1;
    }
    return out_entries;
}

int SplitDataset(FVectPtr start, FVectPtr end, int size, int cscap)
{
    int lo_size = cscap ** (size / (2 * cscap));                 // Calc s_l using b(s_p) AKA cscap
    int split_dim = FindMaxVarianceDimension(start, end);
    SelectOnDimension(split_dim, start, end, lo_size);
    return lo_size;
}
```

C++ routines for VAMSplit R-tree creation excerpted from [101].

REFERENCES

[1] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, (Austin, TX), pp. 311–321, January 1993.

[2] C. Faloutsos, *Searching Multimedia Databases by Content*. Kluwer Academic Publishers, 1996.

[3] V. Castelli, A. Thomasian, and C.-S. Li, "CSVD: Clustering and singular value decomposition for approximate similarity search in high-dimensional spaces," *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, vol. 15, pp. 671–685, May/June 2003.

[4] J. R. Smith, *Integrated Spatial and Feature Image Systems:Retrieval, Analysis and Compression*. Doctor of philosophy, Graduate School of Arts and Sciences, Columbia University, 1997.

[5] A. M. Lesk, *Introduction to Bioinformatics*. Oxford University Press, 2002.

[6] W. Niblack, R. Barber, *et al.*, "The QBIC project: querying images by content using color texture and shape," in *Proc. Storage and Retrieval for Image and Video Databases (SPIE)*, vol. 1908, (San Jose, CA), pp. 173–187, February 1993.

[7] A. Pentland, R. W. Picard, and S. Sclaroff, "Photobook: Content-based manipulation of image databases," *Int'l Journal of Computer Vision*, vol. 18, pp. 233–254, June 1996.

[8] C. Carson, S. Belongie, H. Greenspan, and J. Malik, "Blobworld: Image segmentation using expectation-maximization and its application to image querying," *IEEE Trans. on Pattern Analysis and Machine Intelligence (PAMI)*, vol. 24, pp. 1026–1038, August 2002.

[9] J. P. Eakins and M. E. Graham, "Content-based image retrieval - A report to the JISC technology applications programme." `http://www.unn.ac.uk/iidr/CBIR/report.html`, 1999.

[10] H. Müller, N. Michoux, D. Bandon, and A. Geissbuhler, "A review of content-based image retrieval applications - clinical benefits and future directions," *Int'l Journal of Medical Informatics*, vol. 73, pp. 1–23, February 2004.

[11] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Computing Surveys*, vol. 30, pp. 170–231, June 1998.

[12] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proc. ACM Symposium on the Theory of Computing (STOC)*, (Dallas, TX), pp. 604–613, May 1998.

[13] I. Jolliffe, *Principal Component Analysis*. Springer, 2002.

[14] K. Chakrabarti and S. Mehrotra, "Local dimensionality reduction: A new approach to indexing high dimensional spaces," in *Proc. 26th Int'l Conf. on Very Large Data Bases (VLDB)*, (Cairo, Egypt), pp. 89–100, September 2000.

[15] M. Jürgens, *Index Structures for Data Warehouses*. Springer, 2002.

[16] B. S. Kim and S. B. Park, "A fast k nearest neighbor finding algorithm based on the ordered partition," *Trans. Pattern Analysis and Machine Intelligence (PAMI)*, vol. 8, pp. 761–766, November 1986.

[17] N. Katayama and S. Satoh, "The SR-tree: An index structure for high-dimensional nearest neighbor queries," in *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, (Tucson, AZ), pp. 369–380, May 1997.

[18] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *Proc. 24th Int'l Conf. on Very Large Data Bases (VLDB)*, (New York), pp. 194–205, August 1998.

[19] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. McGraw-Hill, 3rd ed., 2003.

[20] E. Bertino, O. B. Chin, *et al.*, *Indexing Techniques for Advanced Database Systems*. Kluwer Academic Publishers, 1997.

[21] Y. Manopopoulos, Y. Theodoridis, and V. J. Tsotras, *Advanced Database Indexing*. Kluwer Academic Publishers, 2000.

[22] R. Agrawal, C. Faloutsos, and A. N. Swami, "Efficient similarity search in sequence databases," in *Proc. 4th Int'l Conf. of Foundations of Data Organization and Algorithms (FODO)* (D. Lomet, ed.), (Chicago, IL), pp. 69–84, Springer Verlag, October 1993.

[23] H. Jin, B. C. Ooi, H. T. Shen, C. Yu, and A. Y. Zhou, "An adaptive and efficient dimensionality reduction algorithm for high-dimensional indexing," in *Proc. 19th Int'l Conf. on Data Engineering (ICDE)*, (Bangalore, India), pp. 87–98, March 2003.

[24] K. Fukunaga and P. M. Narendra, "A branch and bound algorithm for computing k-nearest neighbors," *IEEE Trans. on Computers*, vol. 24, pp. 750–753, July 1975.

[25] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. of 5th Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, (Berkeley, CA), pp. 281–297, University of California Press, 1967.

[26] S. Guha, R. Rastogi, and K. Shim, "CURE: an efficient clustering algorithm for large databases," in *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, (Seattle, WA), pp. 73–84, June 1998.

[27] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noises," in *Proc. 2nd Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*, (Portland, OR), pp. 226–231, August 1996.

[28] R. T. Ng and J. Han, "Efficient and effective clustering methods for spatial data mining," in *Proc. 20th Int'l Conf. on Very Large Data Bases (VLDB)*, (Santiago de Chile, Chile), pp. 144–155, September 1994.

[29] T. F. Gonzalez, "Clustering to minimize the maximum intercluster distance," *Theoretical Computer Science*, vol. 38, pp. 293–306, 1985.

[30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2nd ed., 2001.

[31] J. H. Friedman, F. Baskett, and L. J. Shustek, "An algorithm for finding nearest neighbors," *IEEE Trans. on Computers*, vol. 24, pp. 1000–1006, October 1975.

[32] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The X-tree: An index structure for high-dimensional data," in *Proc. 22nd Int'l Conf. on Very Large Data Bases (VLDB)*, (San Jose, CA), pp. 28–39, August 1996.

[33] A. Thomasian, V. Castelli, and C.-S. Li, "Clustering and singular value decomposition for approximate indexing in high dimensional spaces," in *Proc. 7th Int'l Conf. on Information and Knowledge Management (CIKM)*, (Bethesda, MD), pp. 201–207, November 1998.

[34] D. Barbará *et al.*, "The New Jersey data reduction report," *IEEE Data Eng. Bull.*, vol. 20, no. 4, pp. 3–45, 1997.

[35] G. K. Wallace, "The JPEG still picture compression standard," *Comm. of the ACM*, vol. 34, pp. 30–44, April 1991.

[36] K.-I. Lin, H. V. Jagadish, and C. Faloutsos, "The TV-tree: An index structure for high-dimensional data," *The VLDB Journal*, vol. 3, no. 4, pp. 517–542, 1994.

[37] A. N. Skodras, C. A. Christopoulos, and T. Ebrahimi, "JPEG2000: The upcoming still image compression standard," in *Proc. 11th Portuguese Conf. on Pattern Recognition*, (Porto, Portugal), pp. 359–366, May 2000.

[38] J. A. Orenstein and T. H. Merrett, "A class of data structures for associative searching," in *Proc. 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, (Waterloo, Ontario, Canada), pp. 181–190, April 1984.

[39] H. V. Jagadish, "Linear clustering of objects with multiple attributes," in *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, (Atlantic City, NJ), pp. 332–342, May 1990.

[40] C. Faloutsos, "Multiattribute hashing using gray-codes," in *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, (Washington, D.C.), pp. 227–238, May 1986.

[41] C. Faloutsos, "Gray-codes for partial match and range queries," *IEEE Trans. Softw. Eng.*, vol. 14, no. 10, pp. 1381–1393, 1988.

[42] C. Faloutsos and Y. Rong, "DOT: A spatial access method using fractals," in *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)*, (Kobe, Japan), pp. 152–159, April 1991.

[43] R. A. Finkel and J. L. Bentley, "Quad trees: A data structure for retrieval of composite keys," *Acta Informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[44] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[45] J. T. Robinson, "The K-D-B-tree: A search structure for large multidimensional dynamic indexes," in *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, (Ann Arbor, MI), pp. 10–18, April 1981.

[46] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, (Boston, MA), pp. 47–57, June 1984.

[47] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-tree: A dynamic index for multi-dimensional objects," in *Proc. 13th Int'l Conf. on Very Large Data Bases (VLDB)*, (Brighton, England), pp. 507–518, September 1987.

[48] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, (Atlantic City, NJ), pp. 322–331, May 1990.

[49] D. A. White and R. Jain, "Similarity indexing with the SS-tree," in *Proc. 12th IEEE Int'l Conf. on Data Engineering(ICDE)*, (New Orleans, LA), pp. 516–523, March 1996.

[50] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *Proc. 23rd Int'l Conf. on Very Large Data Bases (VLDB)*, (Athens, Greece), pp. 426–435, August 1997.

[51] S. Berchtold, C. Böhm, and H.-P. Kriegel, "The Pyramid-Technique: Towards breaking the curse of dimensionality," in *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, (Seattle, Washington), pp. 142–153, June 1998.

[52] C. Böhm, S. Berchtold, and D. A. Keim, "Searching in high-dimensional spaces - index structures for improving the performance of multimedia databases," *ACM Computing Surveys*, vol. 33, pp. 322–373, September 2001.

[53] A. Thomasian, V. Castelli, and C.-S. Li, "RCSVD: Recursive clustering and singular value decomposition for approximate high-dimensionality indexing," Tech. Rep. RC20704, IBM T.J. Watson Research Center, 1997.

[54] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is nearest neighbors meaningful?," in *Proc. 7th Int'l Conf. on Database Theory (ICDT)*, (Jerusalem, Israel), pp. 217–235, January 1999.

[55] B. C. Ooi, K.-L. Tan, C. Yu, and S. Bressan, "Indexing the edges - a simple and yet efficient approach to high-dimensional indexing," in *Proc. 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, (Dallas, TX), pp. 166–174, May 2000.

[56] C. Yu, B. C. Ooi, K.-L. Tan, and H. Jagadish, "Indexing the distance: An efficient method to knn processing," in *Proc. 27th Int'l Conf. on Very Large Data Bases (VLDB)*, (Roma, Italy), pp. 421–430, September 2001.

[57] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima, "The A-tree: An index structure for high-dimensional spaces using relative approximation," in *Proc. 26th Int'l Conf. on Very Large Data Bases (VLDB)*, (Cairo, Egypt), pp. 516–526, September 2000.

[58] R. F. S. Filho, A. Traina, C. T. Jr., and C. Faloutsos, "Similarity search without tears: The OMNI family of all-purpose access methods," in *Proc. 17th Int'l Conf. on Data Engineering (ICDE)*, (Heidelberg, Germany), pp. 623–630, April 2001.

[59] C. H. Goh, A. Lim, B. C. Ooi, and K.-L. Tan, "Efficient indexing of high-dimensional data through dimensionality reduction," *Data & Knowledge Engineering (DKE)*, vol. 32, pp. 115–130, February 2000.

[60] K. Chakrabarti and S. Mehrotra, "The hybrid tree: An index structure for high dimensional feature spaces," in *Proc. 15th Int'l Conf. on Data Engineering (ICDE)*, (Sydney, Australia), pp. 440–447, February 1999.

[61] D. B. Lomet and B. Salzberg, "The hB-tree: A multiattribute indexing method with good guaranteed performance," *ACM Trans. on Database Systems (TODS)*, vol. 15, no. 4, pp. 625–658, 1990.

[62] J. K. Uhlmann, "Satisfying general proximity/similarity queries with metric trees," *Information Processing Letters (IPL)*, vol. 40, no. 4, pp. 175–179, 1991.

[63] T. Bozkaya and M. Ozsoyoglu, "Distance-based indexing for high-dimensional metric spaces," in *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, (Tucson, AZ), pp. 357–368, May 1997.

[64] S. Berchtold and D. A. Keim, "High-dimensional index structures database support for next decade's applications (tutorial)," in *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, (Seattle, WA), pp. 501–501, June 1998.

[65] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. on Mathematical Software*, vol. 3, no. 3, pp. 209–226, 1977.

[66] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel, "A cost model for nearest neighbor search in high-dimensional data space," in *Proc. 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, (Tucson, AZ), pp. 78–86, June 1997.

[67] S. Berchtold, C. Böhm, D. Keim, F. Krebs, and H.-P. Kriegel, "On optimizing nearest neighbor queries in high-dimensional data spaces," in *Proc. Int'l Conf. on Database Theory (ICDT)*, (London, UK), pp. 435–449, January 2001.

[68] C. Böhm and H.-P. Kriegel, "Dynamically optimizing high-dimensional index structures," in *Proc. 7th Int'l Conf. on Extending Database Technology (EDBT)*, (Konstanz, Germany), pp. 36–50, March 2000.

[69] S. Berchtold, B. Ertl, D. A. Keim, H.-P. Kriegel, and T. Seidl, "Fast nearest neighbor search in high-dimensional space," in *Proc. 14th Int'l Conf. on Data Engineering (ICDE)*, (Orlando, FL), pp. 209–218, February 1998.

[70] J. McNames, "A fast nearest-neighbor algorithm based on a principal axis search tree," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 23, pp. 964–975, September 2001.

[71] G. R. Hjaltason and H. Samet, "Ranking in spatial databases," in *Proc. 4th Int'l Symp. on Large Spatial Databases*, (Portland, ME), pp. 83–95, August 1995.

[72] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, (San Jose, CA), pp. 71–79, May 1995.

[73] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," *ACM Trans. on Database Systems*, vol. 24, no. 2, pp. 265–318, 1999.

[74] A. Henrich, "A distance scan algorithm for spatial access structures," in *Proc. 2nd ACM Workshop on Geographic Information Systems*, (Gaithersburg, MD), pp. 136–143, December 1994.

[75] N. Roussopoulos and D. Leifker, "Direct spatial search on pictorial databases using packed R-trees," in *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, (Austin, TX), pp. 17–31, May 1985.

[76] I. Kamel and C. Faloutsos, "Hilbert R-tree: an improved R-tree using fractals," in *Proc. 20th Int'l Conf. on Very Large Data Bases (VLDB)*, (Santiago de Chile, Chile), pp. 500–509, September 1994.

[77] K. L. Cheung and A. W. Fu, "Enhanced nearest neighbor search on the R-tree," *ACM SIGMOD Record*, vol. 27, no. 3, pp. 16–21, 1998.

[78] C. T. Jr., A. Traina, L. Wu, and C. Faloutsos, "Fast feature selection using the fractal dimension," in *Proc. XV Brazilian Symposium on Databases (SBBD)*, (Paraiba, Brazil), October 2000.

[79] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas, "Fast nearest neighbor search in medical image databases," in *Proc. 22th Int'l Conf. on Very Large Data Bases (VLDB)*, (Bombay, India), pp. 215–226, September 1996.

[80] P. F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas, "Fast and effective retrieval of medical tumor shapes," *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, vol. 10, pp. 889–904, November 1998.

[81] T. Seidl and H.-P. Kriegel, "Optimal multi-step k-nearest neighbor search," in *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, (Seattle, WA), pp. 54–165, June 1998.

[82] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. E. Abbadi, "Approximate nearest neighbor searching in multimedia databases," in *Proc. 17th Int'l Conf. on Data Engineering (ICDE)*, (Heidelberg, Germany), pp. 503–511, April 2001.

[83] Y. Li, *Efficient Similarity Search in High-Dimensional Data Spaces.* PhD thesis, New Jersey Institute of Technology, Newark, NJ, May 2004.

[84] Y. Linde, A. Buzo, and R. Gray, "An algorithm for vector quantizer design," *IEEE Trans. Comm.*, vol. 28, pp. 84–95, January 1980.

[85] A. Nobel, "Recursive partitioning to reduce distortion," *IEEE Trans. Information Theory*, vol. 43, pp. 1122–1133, July 1997.

[86] R. Widhopf, "Affic: A foundation for index comparisons," in *Proc. 9th Int'l Conf. on Extending Database Technology (EDBT)*, (Heraklion, Crete, Greece), pp. 868–871, March 2004.

[87] A. Thomasian, Y. Li, and L. Zhang, "Performance comparison of local dimensionality reduction methods," Tech. Rep. ISL-03-01, Integrated Systems Laboratory, Computer Science Department, New Jersey Institute of Technology, Newark, NJ, June 2003.

[88] Y. Li, A. Thomasian, and L. Zhang, "Optimal subspace dimensionality for k-nn search on clustered datasets," in *Proc. 15th Int'l Conf. on Database and Expert Systems Applications (DEXA)*, (Zaragoza, Spain), pp. 201–211, August 2004.

[89] A. Thomasian, Y. Li, and L. Zhang, "Exact k-NN queries on clustered SVD datasets," *Information Processing Letters (IPL), To appear*, 2005.

[90] V. Castelli and L. D. Bergman, eds., *Image Databases: Search and Retrieval of Digital Imagery.* John Wiley and Sons, 2002.

[91] V. Castelli, "Multidimensional indexing structures for content-based retrieval," in *Image Databases: Search and Retrieval of Digital Imagery*, pp. 373–434, John Wiley and Sons, 2002.

[92] J. Gray and P. Shenoy, "Rules of thumb in data engineering," in *Proc. Int'l Conf. on Data Engineering (ICDE)*, (San Diego, CA), pp. 3–12, April 2000.

[93] C. Li, E. Y. Chang, H. Garcia-Molina, and G. Wiederhold, "Clustering for approximate similarity search in high-dimensional spaces," *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, vol. 14, pp. 792–808, July-August 2002.

[94] Y. Zhao, P. Deshpande, and J. F. Naughton, "An array-based algorithm for simultaneous multidimensional aggregates," in *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, (Tucson, AZ), pp. 159–170, May 1997.

[95] F. Farnstrom, J. Lewis, and C. Elkan, "Scalability for clustering algorithms revisited," *ACM SIGKDD Explorations*, vol. 2, pp. 51–57, August 2000.

[96] M. Friedman and O. Pentakalos, *Windows 2000 Performance Guide*. O'Reilly Publishers, 2000.

[97] L. Zhang and A. Thomasian, "Persistent clustered main memory index for accelerating k-NN queries on high dimensional datasets," Tech. Rep. ISL-04-05, Integrated Systems Laboratory, Computer Science Department, New Jersey Institute of Technology, Newark, NJ, October 2004.

[98] H. Aghili and D. G. Severance, "A practical guide to the design of differential files for recovery of on-line databases," *ACM Trans. Database Systems (TODS)*, vol. 7, no. 4, pp. 540–565, 1982.

[99] B. Cui, B. C. Ooi, J. Su, and K.-L. Tan, "Indexing high-dimensional data for efficient in-memory similarity search," *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, vol. 17, pp. 339–353, March 2005.

[100] C. Yu, *High-Dimensional Indexing: Transformational Approaches to High-Dimensional Range and Similarity Searches*, vol. 2431 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[101] D. A. White and R. Jain, "Similarity indexing: Algorithms and performance," in *Storage and Retrieval for Image and Video Databases (SPIE)*, vol. 2670, (San Jose, CA), pp. 62–73, 1996.

[102] D. Yu and A. Zhang, "ClusterTree: Integration of cluster representation and nearest-neighbor search for large data sets with high dimensions," *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, vol. 15, pp. 1316–1337, September/October 2003.

[103] K. Sung and T.Poggio, "Example-based learning for view-based human face detection," *Trans. Pattern Analysis and Machine Intelligence (PAMI)*, vol. 20, pp. 39–51, January 1998.

[104] N. Katayama and S. Satoh, "Distinctiveness-sensitive nearest neighbor search for efficient similarity retrieval of multimedia information," in *Proc. 17th Int'l Conf. on Data Engineering (ICDE)*, (Heidelberg, Germany), pp. 493–502, April 2001.

[105] K. Fukunaga, *Introduction to Statistical Pattern Recognition (2nd ed.)*. Academic Press, 1990.