RIVAL PENALIZED COMPETITIVE LEARNING FOR CONTENT-BASED INDEXING

By

LAU Tak Kan

A Dissertation

submitted in partial fulfillment of the requirements

for the Degree of Master of Philosophy

Division of Computer Science and Engineering

The Chinese University of Hong Hong

June 1998

Abstract

Efficient and accurate information retrieval is one of the main issues in multimedia databases. In content-based multimedia retrieval databases, contents or features of the database objects are used for retrieval. To retrieve similar database objects, we often perform nearest-neighbor search. A nearest-neighbor search is used to retrieve similar database objects with features nearest to the query under the feature vector space with a given distance function (similarity measurement). The result of the nearest-neighbor search is often a natural cluster of data. Currently, many of the indexing methods do not utilize this data cluster information in the construction of the indexing structure which leads to performance degradation. To improve the retrieval performance, we (1) use *Rival Penalized Competitive Learning (RPCL)*, a stochastic clustering algorithm, to locate good approximate cluster centers efficiently and (2) use the result of RPCL clustering to construct a good indexing structure for effective nearest-neighbor search. Moreover, we present two approaches to cluster features for indexing: (1) *Non-hierarchical approach* and (2) *Hierarchical approach.* The non-hierarchical approach considers the whole feature space each time when clustering the data for indexing and retrieval. On the other hand, the hierarchical approach transforms the feature space into a sequence of nested clusters and builds a hierarchical binary indexing tree (RPCL-b-tree) for retrieval. Our experimental results show that: (1) RPCL is faster than other tested clustering methods to locate natural clusters for indexing, (2) the non-hierarchical RPCL indexing method has high performance for

producing good approximate retrieval result quickly, and (3) RPCL-b-tree is efficient to produce 100% nearest-neighbor search results and it is faster than VP-tree in general. Moreover, based on the experimental results, we work out a formula for RPCL-b-tree to describe the relationship between the searching parameters and the searching efficiency. We can also use it to compare the searching efficiency with other indexing methods for a given set of parameters.

Acknowledgement

I would like to express my eternal gratitude to my supervisor, Professor Irwin King for his academic guidance, emotional support, encouragement, and patience on the work of this thesis. I would also like to sincerely thank my examining committee, Professor Wai Chee Fu, Professor Lai Wan Chan, and Professor Helen Shen for their comments and useful suggestions on this thesis.

Moreover, many thanks go to all the team members of the Montage project, especially Tom Hung and Catherine Chan for giving me lots of images and data I used in my experiments. My thanks also go to my friends Alan Tung and John Sum for teaching me how to do research.

Finally, I am deeply grateful to my parents, grandfather, brothers, and sister for their love, support, and patience during the past two years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In this information age, efficient and accurate multimedia information management is a very important issue. People often need to manipulate many different kinds of multimedia information such as images, sound, and videos for different tasks. In the past, people may use traditional databases to manage these multimedia information, but this is usually ineffective and imprecise. These databases use keywords or text descriptors for retrieval, but it poses difficulties for the end users especially for those without special training. The main difficulties are:

1. **Lack of Standards:** Different users may use different words to describe a multimedia data object for retrieval.

2. **Lack of Descriptive Power:** Even standardize vocabulary is used, it is still hard to depict the object clearly and precisely.

We use an image database, which is a special kind of multimedia database for image management, as an example. If we want to retrieve an image of a sunset photo with the sun at the upper left corner in an image database, we may use a keyword "sunset". Not surprisingly, some sunset images having the suns at

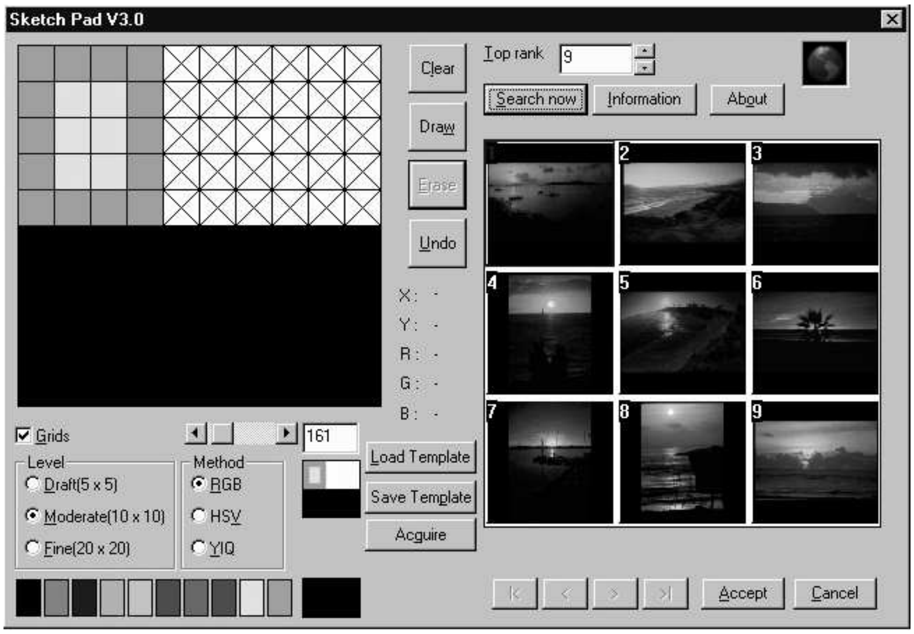


Figure 1.1: An example of image retrieval by image content.

the bottom right corner are retrieved which are opposite to what we want. This shows that it is hard to use keywords for precise image retrieval. To improve the efficiency and accuracy, we need a new kind of database which is specially designed for multimedia data organization and retrieval.

In order to manage a large amount of multimedia data efficiently and easily, multimedia databases are emerged. These databases support content-based retrieval which lets users to specify queries by features (or contents) such as *color*, *texture*, *sketch* and *shape* to retrieve database objects with features similar to the queries. For example, we may retrieve a sunset image having the sun at the upper left corner by simply using a color sketch with orange and red colors at the upper left corner and dark brown color at the bottom. The retrieved images are most likely what we want (see Figure 1.1). This shows that it is more effective and easier to use features for retrieval.

Many content-based retrieval multimedia database systems have been developed in the past few years. For example, Query by Image Content (QBIC) [54, 19, 4, 22] allows queries on databases based on color, texture, and shape of database objects. Photobook [59] makes use of semantics-preserving image compression to support search based on three image content descriptions: appearance, 2-D shape, and textural properties. VisualSEEk [64] is a content-based image and video retrieval system for World Wide Web. It uses color contents and the spatial layout of color regions of images for retrieval. Other multimedia database systems which support content-based query include Chabot [57], MMIS [56, 25, 26, 30], VIMSYS [32], ART MUSEUM [40, 35], KMeD [15, 14], and CORE [69]. Although the above databases use different approaches for management, most of them have shown that they are efficient for retrieval.

At the Chinese University of Hong Kong, we have developed an image database system called, Montage [41, 42, 45, 46] for managing and retrieving visual information efficiently and effectively. I used to be a member of the developing team of system. Montage is an image database supporting content-based retrieval by *color histogram*, *sketch*, *texture*, and *shape*. One important feature of Montage is the *Open Architecture* design. There are two aspects of this open architecture design: (1) Open DataBase Connectivity (ODBC) and (2) plug-in framework. They make the system *extensible*, *customizable*, and *flexible*.

In a typical multimedia database, all the database objects have to be pre-analyzed and then organized in a special way for retrieval. The main steps are:

1. The corresponding features from each of the database objects are first extracted. These features are usually stored in the form of real-valued multi-dimensional vectors.

2. The database may then organize the extracted features by using an indexing structure for retrieval.

3. Content-based retrieval can be performed on the indexing structure efficiently and effectively.

In summary, Figure 1.2 shows the flow of the whole process.

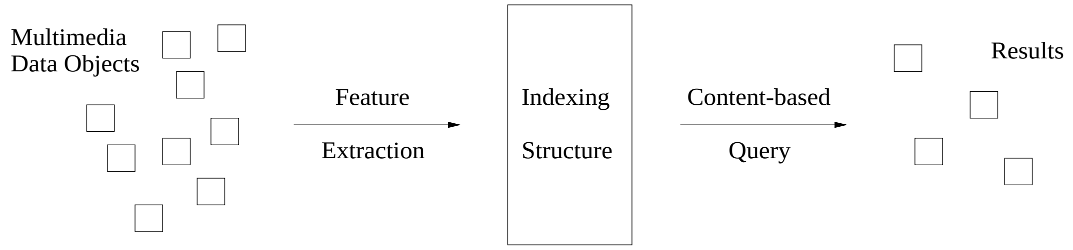


Figure 1.2: The flow of indexing and retrieval in a content-based retrieval multimedia database.

By using feature vectors, the content-based retrieval multimedia databases support similar searching. By applying a suitable distance function (see Definition 2.2 for details) to the feature vectors as the similarity measurement, the database objects can then be ranked according to a query. The top ranked objects are then retrieved as the result for similar retrieval. Nearest-neighbor search is a typical kind of similar searching. In the feature vector space or the real space, a nearest-neighbor query is simply a multi-dimensional point (or vector). The result of the query is the objects with features which are the neighbors of the query point. Using nearest-neighbor search, we can retrieve similar data objects easily.

For the multimedia databases with nearest-neighbor retrieval, a good indexing method is a key component for efficient and accurate retrieval. Nowadays, alphanumeric data indexing techniques are already well developed such as [5, 16]. However, these databases make use of features for retrieval. The alphanumeric indexing methods are not particularly suitable for indexing features because they are designed for one-dimensional vectors, but not multi-dimensional vectors like the ones used in databases. Therefore, people have begun to develop new indexing methods for content-based retrieval in databases such as R-tree [33], R+-tree [63], R*-tree [6], SR-tree [39], Quad-tree [21], k-d tree [7], VP-tree [71], MVP-tree [9], and some other methods [8, 68].

1.2 Problem Defined

Generally, multimedia databases contain database objects with features approximately in Gaussian distributions and there usually exist some natural data clusters in the feature vector space (see Section 2.4 for details). For nearest-neighbor search, a group of features will often be retrieved together as the result of a query. Therefore, if we can first calculate the natural clusters from the feature space and then build an indexing structure based on the clusters, nearest-neighbor search will become more efficient and effective.

The existing indexing methods usually generate partitions for the feature vector space which lead to indexing structures for efficient retrieval in many cases, but most of them seem to fail to retrieve similar database objects when a nearest-neighbor query lies on the partition boundary. One of the reasons is that these methods do not look at the distribution of the features to find natural clusters so that features in the same natural cluster may be partitioned into several different nodes. As a result, the performance of nearest-neighbor searches for these methods will be degraded.

In short, the problems we are facing are:

1. to find an efficient clustering method to locate natural clusters from the input feature vector set, and

2. to build a good indexing structure based on the clusters for efficient and effective retrieval.

1.3 Contributions

The main contributions of our work for solving the problems defined in the last section are shown as follows.

1. We use a clustering method *Rival Penalized Competitive Learning (RPCL)* [70] to calculate natural clusters from the feature vector set. RPCL is an unsupervised neural network heuristic algorithm for clustering. It provides a good approximate of the centers to the clusters and it is computational efficient. Therefore, we make use of RPCL to calculate natural clusters for indexing and retrieval.

2. We build indexing structures based on the natural clusters in two different approaches: (1) *Non-hierarchical approach* and (2) *Hierarchical approach*. The non-hierarchical approach considers the whole feature space to locate different numbers of natural clusters each time. The resultant clusters are indexed non-hierarchically by using an inverted file structure for retrieval [44, 43, 47]. On the other hand, the hierarchical approach transforms a feature space into a sequence of nested clusters and builds a hierarchical binary indexing tree (RPCL-b-tree) based on the clusters. We then apply a branch-and-bound technique [38] on the indexing structure for efficient retrieval (see Section 5.5 for details). In short, these two approaches make use of the information of natural clusters for efficient and effective indexing and retrieval.

Our experimental results show that:

1. RPCL is faster than k-means, competitive learning, and general hierarchical clustering methods to locate natural clusters for indexing.

2. The non-hierarchical RPCL indexing method has high *recall* and *precision* performance for producing good approximate retrieval result quickly.

3. RPCL-b-tree is faster than VP-tree to produce 100% nearest-neighbor search results.

According to the experimental results for RPCL-b-tree, we work out a formula to describe the relationship between the searching parameters and the searching efficiency. We can then make use of this formula to find out the estimated efficiency value for a given set of parameters. Besides, we can generalize the formula to other indexing methods for comparing their efficiency with a given set of parameters.

1.4 Thesis Organization

The rest of the thesis is organized as follows. We will first present some of the technical details of multimedia databases in Chapter 2. We will also introduce the problem of most of the indexing methods in that chapter. We will then present our proposed solution of the indexing problem in Chapter 3. We will use RPCL to produce clusters and describe how to build good indexing structures from the clusters with two different approaches. Chapter 4 and Chapter 5 will show the *Non-hierarchical approach* and the *Hierarchical approach* respectively. Several experiments and discussions will be presented in these chapters. We will then show how to work out the relationship formula from the experimental results. Finally, we will have a brief summary of our proposed methods together with some future works in Chapter 6.

Chapter 2

Content-based Retrieval Multimedia Database Background and Indexing Problem

In this chapter, we first give some technical backgrounds of the content-based retrieval multimedia databases: *Feature Extraction*, *Nearest-neighbor Search*, and *Content-based Indexing*. We then present some problems found in the existing content-based indexing methods.

2.1 Feature Extraction

Feature extraction is one of the main aspects in content-based retrieval multimedia databases. In a content-based retrieval multimedia database, users may want to retrieve database objects similar to a query in terms of some kinds of features. Therefore, when a multimedia data object is inserted into the database, the useful features of the object will be extracted and transformed into feature vectors. The database will then organize the feature vectors for content-based retrieval.

The definition of feature extraction is:

Definition 2.1 (Feature Extraction) *Let $DB = \{I_i\}_{i=1}^{n}$ be a set of database objects. With a set of feature parameters $\theta = \{\theta_i\}_{i=1}^{m}$, a feature extraction function f is defined as:*

$$f : I \times \theta \to \mathcal{R}^d \ ,$$

which extracts a real-valued d-dimensional vector.

We use a simple example here to explain the above definition. Let $DB = \{I_1, \ldots, I_{10}\}$ be a set of 10 images and $\theta = \{\theta_1\}$ be the image feature parameter set where θ_1 indicates the number of top colors considered for extraction. $f(I_5, 2)$ will return a real-value vector based on the top two colors in the image I_5.

Many features can be used for feature extraction, such as, *color*, *sketch*, *texture*, and *shape*. Here are some examples for images.

1. **For color**, the overall color of an image is analyzed and a color histogram is built and transformed to a feature vector [27].

2. **For sketch**, an image query may be a hand-drawn sketch of the target image. The regionalized color information of an image is extracted to form feature vector so that the query can be compared to each of the images region by region for retrieval [36].

3. **For texture**, some statistical methods are usually used to analyze the texture information of an image [65]. Some researchers use Gabor filter for image scaling and orientation in texture analysis [48, 50].

4. **For shape**, it is still a hot research topic because it is difficult to extract shape information from an image precisely. In fact, different shape features

may be used for extraction such as outline based features, region based features, and combined features [37, 51, 52, 60, 66, 67].

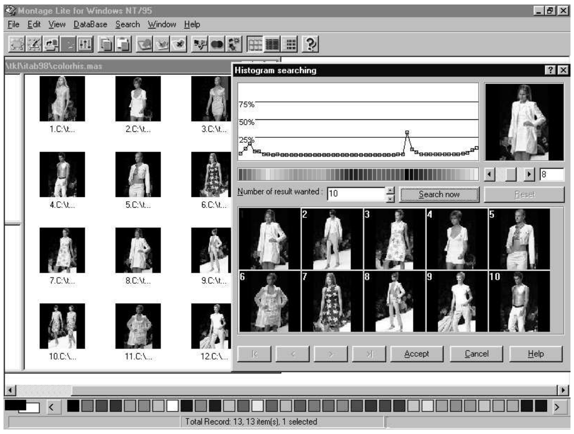
Figure 2.1 shows the image retrievals using the above four features: color histogram, color sketch, texture, and shape respectively. Other features for content-based retrieval include volume, spatial constraints, objective attributes, subjective attributes, etc [31].

Here is an example to illustrate the detail of feature extraction using color histogram in an image database (see Figure 2.2). Given an image, its overall color is begin analyzed in order to get a feature vector. All the colors in the image are quantized into n representative colors. By calculating the frequency of each representative color, a n-bucket color histogram is formed. For fair comparison to the other color histograms, the sum of the frequencies is normalized to 1. After normalization, the histogram is transformed into a n-dimensional feature vector for indexing and retrieval.

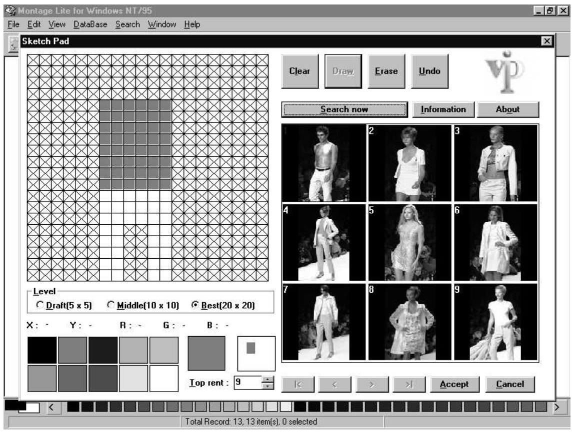
2.2 Nearest-neighbor Search

By using the extracted feature vectors, content-based retrieval multimedia databases allow users to perform similar searching. For similar searching, the database objects with features similar to the query will be retrieved. Nearest-neighbor (NN) search is one of the common similar searching techniques used in the multimedia databases for content-based retrieval.

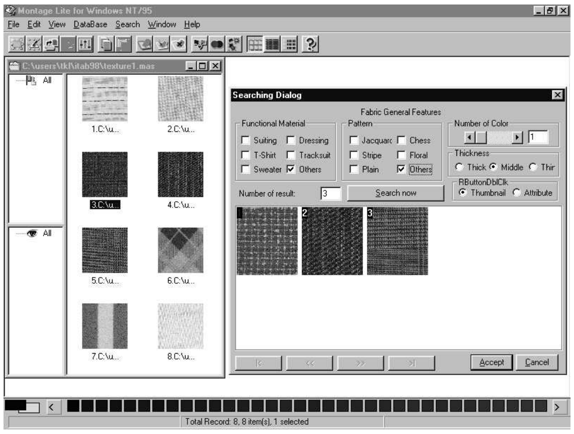
Nearest-neighbor search usually makes use of a distance function for similarity measurement. In order to determine how close or how similar two features are in the feature vector space, a distance function is defined to measure their similarity. With the two features as the input parameters, it usually outputs a real value such that the smaller the value, the more the similar between each the two input features.



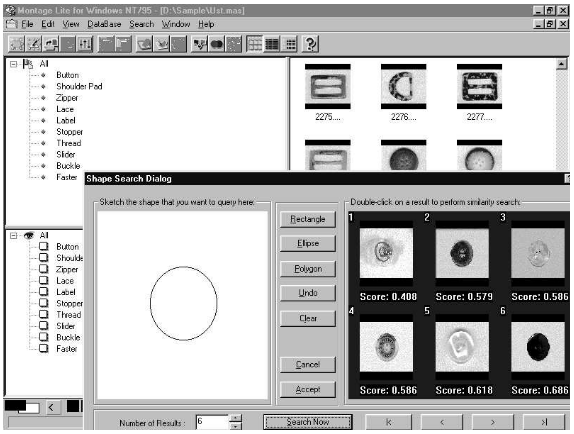
(a)



(b)



(c)



(d)

Figure 2.1: (a) Query by color histogram. (b) Query by sketch. (c) Query by texture. (d) Query by shape.

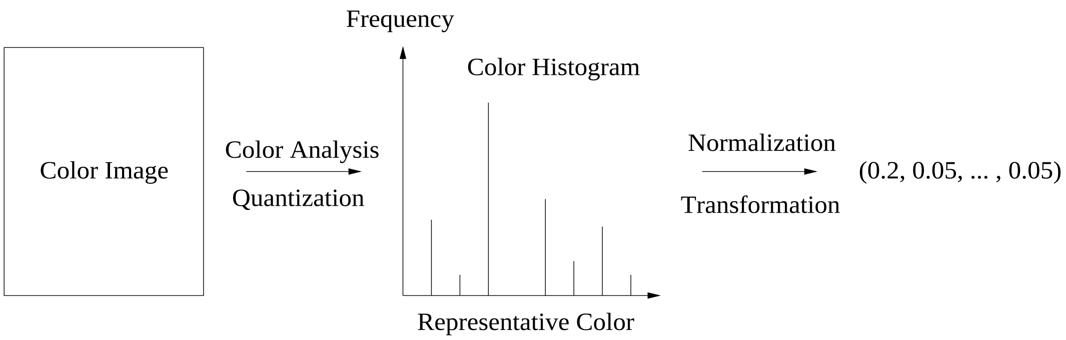


Figure 2.2: Feature extraction of a color image using color histogram.

Definition 2.2 (Distance Function) *A typical distance function D is defined as follows.*

$$D : F \times F \to \mathcal{R}$$

satisfying:

1. $D(x, y) \geq 0$,
2. $D(x, y) = D(y, x)$,
3. $D(x, y) = 0$ *iff* $x = y$, *and*
4. $D(x, y) + D(y, z) \geq D(x, z)$

where x, y, and $z \in F$ and F is a feature vector set.

L_2-norm (Euclidean distance) is one of the common distance functions and it is defined as: D as: $D(x, y) = \|x - y\| = \sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$.

A nearest-neighbor search in a content-based retrieval multimedia database is a retrieval of database objects with features nearest to a query under the feature space with a given distance function. There are two main kinds of nearest-neighbor searches: *range nearest-neighbor search* and k *nearest-neighbor search.*

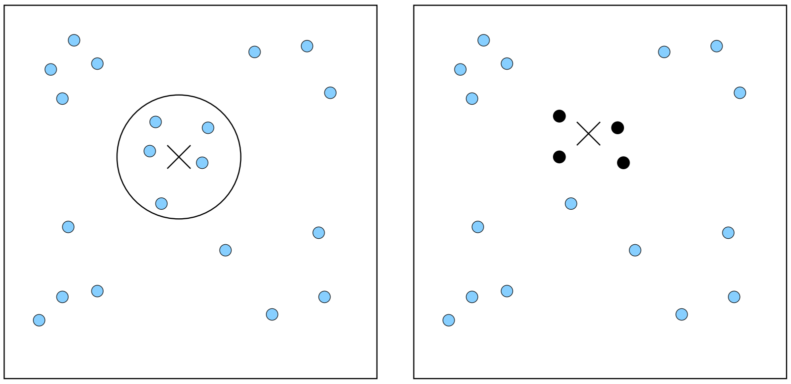
Definition 2.3 (Range Nearest-neighbor Search) *Given a set of N features $X = \{x_i\}_{i=1}^{N}$, a range nearest-neighbor query $\hat{x}$ returns the set P of features:*

$$P = \{x | x \in X \text{ and } 0 \leq D(x, \hat{x}) \leq \epsilon\}, \tag{2.1}$$

where ϵ is a pre-defined positive real number and D is a distance function.

Definition 2.4 (k Nearest-neighbor Search) *Given a set of N features $X = \{x_i\}_{i=1}^{N}$, a k nearest-neighbor query $\hat{x}$ returns the set $P \subseteq X$ satisfying:*

1. $|P| = k$ *for* $1 \leq k \leq N$, *and*



(a) (b)

Figure 2.3: (a) Range nearest-neighbor search in 2D. (b) k nearest-neighbor search in 2D ($k = 4$).

2. $D(\hat{x}, x) \leq D(\hat{x}, y)$ *for* $y \in X - P$.

where D *is a distance function.*

Range NN search gives the database objects with features located inside the query circle which has the query point as the center and a small positive real number ϵ as the radius (see Figure 2.3(a)). k-NN search gives the objects with features which are the top k nearest neighbors to the query (see Figure 2.3(b)).

Many different algorithms for nearest-neighbor search have been proposed. Table 2.1 shows some of the algorithms for finding the most nearest neighbor. We discuss their efficiency and then try to find a suitable one for our proposed indexing method. We make use of the total number of distance computations between the sample data and the query needed to measure the efficiency of these algorithms. Basically, more distance computations will lessen the efficiency. In our work, we are using real-valued multidimensional feature vectors for indexing and retrieval and using Euclidean distance as the similarity measurement. Therefore, by considering the methods using the Euclidean distance as the metric, Kamgar's

Algorithm	Data	Metric	Result
Burkhard and Keller (1973) [12]: Some approaches to best-match file searching	1000 randomly generated registers of a file using 30-bits keys	Hamming distance	$\sim$ 700 average distance computations ($\sim$ 70 %)
Fukunaga and Narendra (1975) [23]: A branch-and-bound algorithm for computing K-nearest neighbors based on a hierarchical indexing structure	1000 2D uniform samples data	Euclidean distance	$\sim$ 580 average distance computations ($\sim$ 58 %)
Feustel and Shapiro (1982) [20]: The nearest-neighbor problem in an abstract metric space	29 randomly generated 5-vertices directed graphs	Graph-isomorphism-based discrete-valued distance	$\sim$ 3 average distance computations ($\sim$ 10 %)
Kamgar and Kanal (1985) [38]: An improved branch-and-bound algorithm for computing k-nearest neighbors based on a hierarchical indexing structure	1000 2D samples uniform sample data	Euclidean distance	$\sim$ 165 average distance computations ($\sim$ 16.5 %)
Roussopoulos et al. (1995) [61]: Nearest neighbor queries for R-tree	1K, 4K, 16K, 64K, and 256K synthetic uniformly distributed data sets	MINDIST and MINMAXDIST distances	The no. of nearest neighbors increased the no. of pages accessed grew in a linear ratio
Nene and Nayer (1997) [53]: A simple algorithm for nearest-neighbor search in high dimensions	30000 and 100000 high dimensional uniform and normal distribution samples	Euclidean distance	$\sim$ 20 % of search time used than exhaustive search for 30000 10D data and $\sim$ 40 % of search time used than exhaustive search for 30000 25D data

Table 2.1: Searching performance of some nearest-neighbor search algorithms.

improved branch-and-bound method [38] is the most suitable nearest-neighbor search algorithm for our proposed indexing method.

2.3 Content-based Indexing Methods

In the past two decades, people have developed many indexing methods for content-based retrieval in multimedia databases. In this section, we concentrate on two main kinds of content-based indexing methods: *rectangle-based indexing* and *partition-based indexing*.

Rectangle-based Indexing

The rectangle-based indexing methods make use of rectangles to organize the features into groups for indexing. Examples are *R-tree*, *R+-tree*, *R*-tree*, and *SR-tree*.

R-tree

R-tree [33] is a generalization version of the B-tree [5, 16] for multi-dimensional data indexing. It uses rectangles to partition the data into groups. The partition process will proceed hierarchically and an indexing tree will then be produced.

- **Properties:** R-tree is a height-balanced tree and it has two kinds of nodes: *Leaf Node* and *Non-leaf Node*. Let M be the maximum number of entries that a node can contain and $m \leq M/2$ be the minimum number. Every leaf node except the root contains between m and M records which are pointing to the database objects. Every non-leaf node except the root has between m and M children. The root node has at least two children unless it is a leaf node.

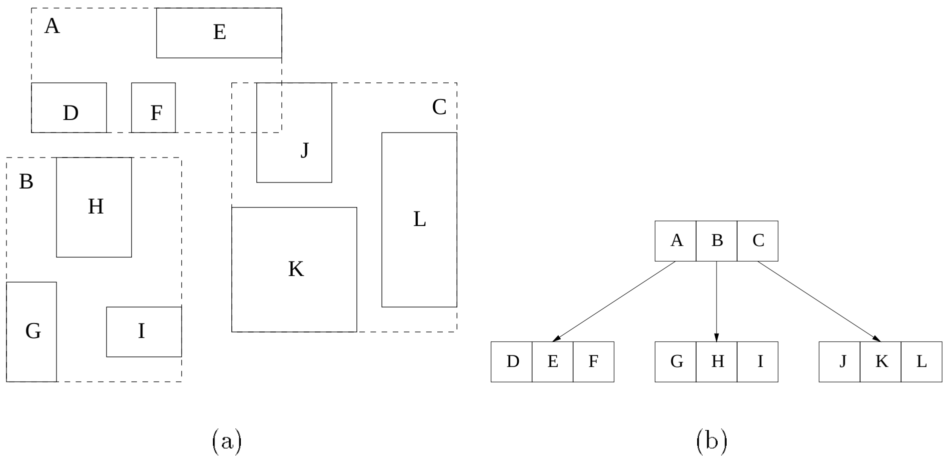


Figure 2.4: (a) An input data set partitioned by using minimum bounding rectangles. (b) The corresponding R-tree structure.

- **Insertion:** R-tree is built by inserting the data objects one by one. Figure 2.4 shows an example. Starting from the root node with a minimum bounding rectangle (MBR) which is the smallest rectangle containing all the data objects for the node, data objects are inserted until the node contains more than M objects. When the node is overflow, a splitting algorithm is applied to split the corresponding rectangle into several small rectangles for child nodes. It tries to optimize the area of the MBRs to each child node. From now on, a target node is selected first for each to-be inserted data object and splitting is performed when the inserted node becomes overflow.

- **Deletion:** Apart from insertion, deletion is also a main operation of R-tree. After deleting a data object from a node, a merging algorithm will be applied if the deleted node contains less than m objects.

- **Searching:** Given a R-tree and a query rectangle, all the nodes with MBRs overlapping the query rectangle will be examined in order to find the results of the query.

R-tree works fine for many cases, but it is not efficient when a query lies on the overlapping area of two or more minimum bounding rectangles. All the involved rectangles have to be examined to find out the results of the query which lessen the efficiency of the retrieval. Therefore, it is better to decrease the overlapping area as much as possible so as to make the retrieval faster.

R+-tree

R+-tree [63] is a variation of R-tree. Unlike R-tree, its searching and updating algorithms are modified in order to avoid the overlapping rectangles in the intermediate nodes of the indexing tree. According to the experimental results in [63], R+-tree has a better searching performance than R-tree. Also, it is more efficient for indexing point data and point queries than R-tree.

R*-tree

R*-tree [6] is another variation to R-tree. The authors of R*-tree show in [6] that overlapping-region-technique does not imply bad average searching performance. Below are the essential parameters of the retrieval performance.

1. The area covered by a directory rectangle should be minimized.

2. The overlap between directory rectangles should be minimized.

3. The margin of a directory rectangle should be minimized.

4. Storage utilization should be optimized.

Therefore, the authors modify the splitting algorithms using in R-tree so as to improve the retrieval performance by reducing the area, margin, and overlap of the rectangles. Moreover, the storage utilization is higher than R-tree. In short, from the experimental results in [6], R*-tree outperforms the other R-tree variants.

SR-tree

The *SR-tree* [39] stands for the Sphere/Rectangle-tree and it is an extension of the R*-tree [6] and the SS-tree [68]. The distinctive feature of the SR-tree is that it makes use of both rectangles and spheres for indexing. This improves the performance on nearest-neighbor queries by reducing both the volume and the diameter of regions compared with the R*-tree and the SS-tree. According to the performance experiments in [39], SR-tree outperforms R*-tree especially for high dimensional data.

Partition-based Indexing

The partition-based indexing methods make use of lines or curves to produce partitions to the input feature vector space for indexing. Examples are *Quad-tree*, *k-d tree*, *VP-tree*, and *MVP-tree*.

Quad-tree

Quad-tree [21] is an early developed indexing method for multi-dimensional data objects and it is the generalization of the binary search tree.

- **Properties:** Quad-tree divides the vector space into subspaces for different directions. In two-dimensional space, for example, each non-leaf node has four child nodes representing its four directions NW, NE, SW, and SE (see Figure 2.5).

- **Insertion:** Data objects are inserted one by one. Starting from the root node, the direction of a to-be inserted data object to the root node is determined and the corresponding child node will be selected for further processing until the leaf node level is reached.

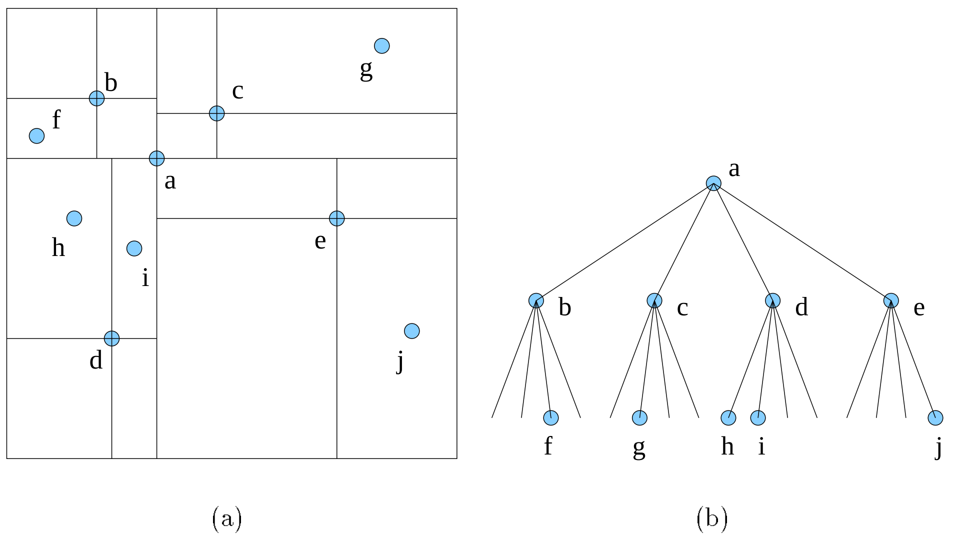


Figure 2.5: (a) An input 2-D data set for quad-tree. (b) The corresponding quad-tree structure.

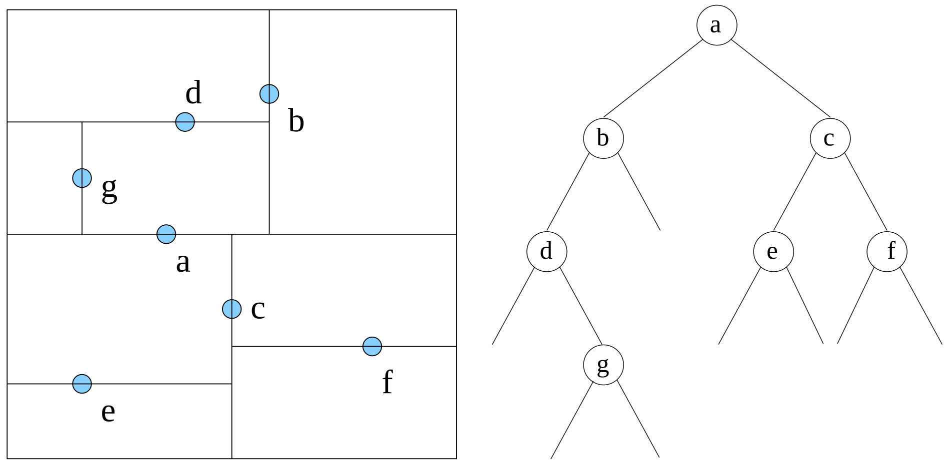
- **Searching:** Searching in a quad-tree is performed according to the direction of the query to each node. In k-dimensional case, for example, it is required to consider all k coordinates of a given query at each node to determine its direction. The query is first compared to the root node in order to determine which child node is examined next. The above process will then repeat until the target leaf node is found.

For a given 2-D data set, the insertion algorithm yields $n \log n$ performance. The quad-tree seems to be an efficient for two-dimensional space.

k-d tree

k-d tree [7] is a multi-dimensional binary search tree where k denotes the dimensionality of the search space.

- **Properties/Searching:** Unlike quad-tree that all k coordinates have to be tested at each node, only a different attribute value is tested at each level of



(a) (b)

Figure 2.6: (a) An input data set for k-d tree. (b) The corresponding k-d tree structure.

the tree for indexing and searching. For example, in two-dimensional space, we are comparing the x coordinates at even levels whereas y coordinates at odd levels.

- **Insertion:** For insertion, a to-be inserted data object is first compared to the root node using a suitable attribute value. A child node is then selected for further processing. The process repeats until a leaf node is found. Finally, the data object is inserted and it partitions the space associated with the leaf node into two sub-spaces for two child nodes according to a suitable attribute value. Figure 2.6 shows an example of k-d tree.

In short, by considering only an appropriate coordinate at each node, k-d tree is relatively more efficient than quad-tree for indexing and retrieval.

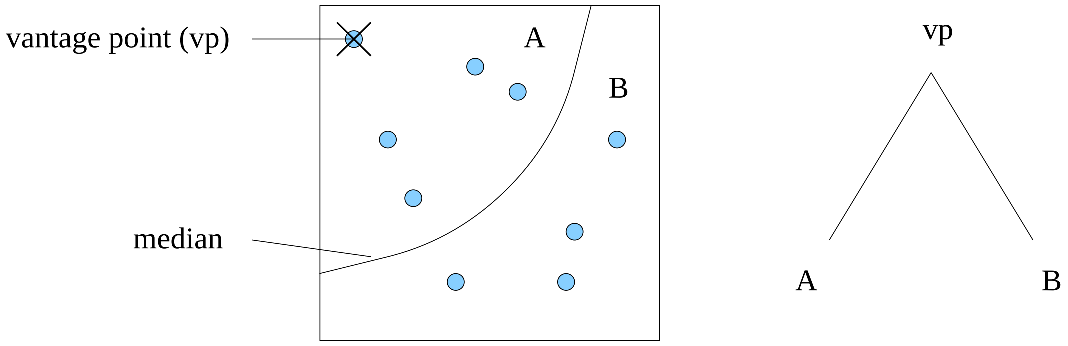


Figure 2.7: A simple VP-tree for the data set on the left.

VP-tree

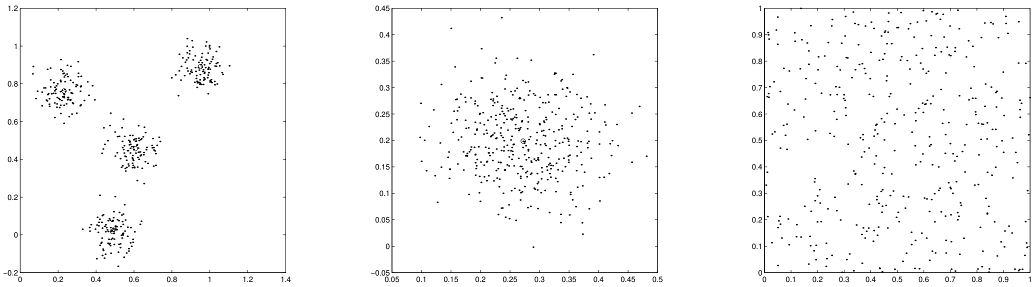
Vantage point tree (VP-tree) [13, 71] is an indexing method for multi-dimensional nearest-neighbor search.

- **Properties:** Like k-d tree, each VP-tree node cuts the space. Unlike k-d tree, VP-tree partitions the feature vector space based on the distances between the feature vectors and a calculated vantage point.

- **Building Indexing Tree:** According to the median of these distances, the whole feature space is divided into two sets: close vector set and far vector set. The process will continue in both sets individually. Finally, an indexing tree structure will be built based on the resultant vector sets (see Figure 2.7).

- **Searching:** For searching in VP-tree, a query is first compared to the vantage point associated to the root node and then determined which child node is going to be examined. The process repeats until the target leaf node is found. The data objects associated with the leaf node satisfying the searching criteria of the query will be retrieved as the result.

The experimental results in [71] show that VP-tree outperforms k-d tree in many cases.



(a) (b) (c)

Figure 2.8: Different data distributions. (a) Mixture Gaussian. (b) Super Gaussian. (c) Uniform.

MVP-tree

Multi-vantage point tree (MVP-tree) [9] is a distance based indexing method for similarity queries on high-dimensional metric spaces. Like VP-tree, it uses vantage point for indexing. Unlike VP-tree, it uses more than one vantage point to partition the feature vector space. Experiments in [9] show that MVP-tree outperforms the VP-tree 20% to 80% for varying query ranges.

2.4 Indexing Problem

The distributions of the features in multimedia databases can usually be approximated by one of the two main distributions: *Gaussian* and *Uniform*. For example, a typical image database often contains many different kinds of images such as sunset pictures, mountain photos, etc. In terms of the color histogram feature, the sunset images will have similar color histograms and form a data cluster in the feature vector space. With the same reason, the mountain photos form another cluster. We can easily use a mixture of Gaussian distributions (see Figure 2.8(a) and section 4.2.1 for details) to approximate this kind of distribution. Here is

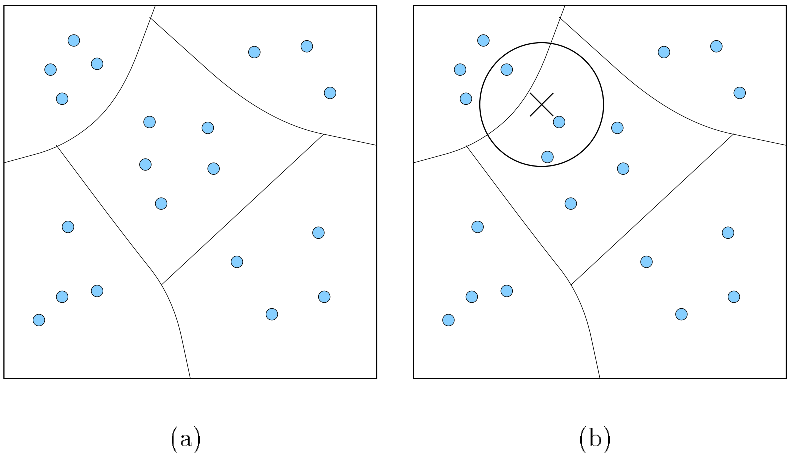


Figure 2.9: (a) A data set with natural clusters. (b) A boundary nearest-neighbor query.

another example, for a face image database, all the face images have some common characteristics. However, each face image is still different from the others. Therefore, the distribution of the image features is most likely a super Gaussian (see Figure 2.8(b)). If an image database is used to manage general images which are in different types from each other, the image features will be probably uniformly distributed (see Figure 2.8(c)). In general, we can assume multimedia databases often have database objects with features in Gaussian distributions and there usually exist natural clusters in their image feature vector spaces (see Figure 2.9(a)).

For nearest-neighbor search, a group of features will often be retrieved together as the result of the query. Therefore, if we can first calculate the natural clusters from the feature vector space and then build an indexing structure based on the clusters, nearest-neighbor search on the structure will become more efficient and effective.

People have developed many indexing methods for content-based retrieval in multimedia databases and they seem to work fine for many cases in general, but most of them seem to fail to retrieve similar database objects when a nearest-neighbor query lies on the partition boundary. One of the main reasons is that these methods do not look at the distribution of the features to find natural clusters so that features in the same natural cluster may be partitioned wrongly into several different nodes. We call this as the boundary problem (see Figure 2.9(b)). For example, the rectangle-based indexing methods such as R-tree, R+-tree, and R*-tree are built based on the input sequence of the data objects so that they cannot pay attention to the distribution of the input data and calculate natural clusters. The partition-based indexing method such as VP-tree partitions the data object space according to the median distances from the data objects to the vantage points, but it still cannot exactly find out the natural clusters for retrieval. As a result, the performance of nearest-neighbor retrievals for these methods is reduced by the boundary problem.

Therefore, we are going to work out a new indexing method for the above problem. We need to find an efficient clustering method to calculate natural clusters from the feature vector space and then build a good indexing structure based on the natural clusters. We want the new indexing method can lessen the above problem and make the content-based retrieval more efficient and effective.

Chapter 3

Data Clustering Methods for Indexing

3.1 Proposed Solution to Indexing Problem

We propose to use an efficient clustering algorithm for content-based indexing in order to lessen the indexing problems mentioned in the last chapter. Under the assumption that there usually exist natural clusters in the feature vector space, we make use of an efficient clustering method to locate those natural clusters form the features. We will discuss which clustering algorithm is good for us in this chapter. In the following chapters, we will describe how to build indexing structures based on the natural clusters for nearest-neighbor retrieval. We will also present several performance experiments to show that our proposed method is accurate and efficient.

3.2 Brief Description of Several Clustering Methods

In this section, we give a brief description of several clustering methods: *k-means*, *Competitive Learning*, *Rival Penalized Competitive Learning*, and *general hierarchical clustering methods* to produce cluster partitions form a given data set.

3.2.1 K-means

The k-means [3, 49] method groups the feature vectors into k cluster partitions. Given n feature vector, the algorithm is shown as follows.

Algorithm 3.1 K-means(D, k)

▷ *Input: the input data set D and the expected number of clusters k*
▷ *Output: the k clusters of D*
$\{s_i\}_{i=1}^{k} \leftarrow k$ randomly selected feature vectors from D
▷ *$\{s_i\}_{i=1}^{k}$ is a temporary data set*
for $i = 1$ **to** k **do**
    $C_i \leftarrow \{s_i\}$ ▷ *C_i is a cluster*
    $c_i \leftarrow s_i$ ▷ *c_i is the cluster center of C_i*
end for
$P = \{\}$
while true **do** ▷ *forever loop*
    for each data object j of remaining $n - k$ data objects from D **do**
        $C_i \leftarrow C_i \cup \{j\}$ if the center c_i of C_i is the nearest center to j
        recompute the center c_i of C_i
    end for
    if $P = \bigcup_{i=1}^{k}\{c_i\}$ **then do**
        return $\{C_i\}_{i=1}^{k}$
    else
        $P = \{\}$
        for $i = 1$ **to** k **do**
            $C_i \leftarrow \{c_i\}$
            $P \leftarrow P \cup \{c_i\}$
        end for
    end if
end while

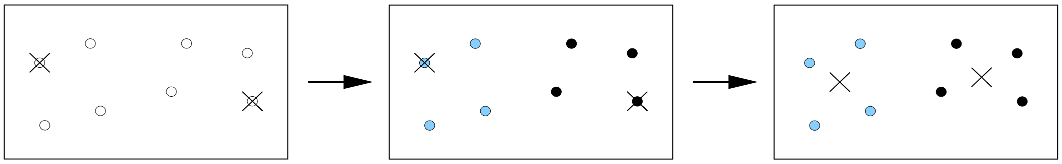


Figure 3.1: k-means clustering.

Figure 3.1 shows an example of using k-means for clustering. In the figure, the dots represent the input data objects and the crosses indicate the centers of the clusters. By applying the k-means clustering algorithm, the resultant clusters are obtained.

3.2.2 Competitive Learning (CL)

Competitive Learning (CL) [62, 34] is an unsupervised neural network learning algorithm to produce cluster partitions. In this section, we present the technique of using competitive learning for clustering. There are some basic conditions of the competitive learning rule:

- Start with a set of neurons that are all the same except for some randomly distributed synaptic weights which make each of them respond differently to a set of input patterns.
- Limit the "strength" of each neuron.
- Allow the neurons to compete for the right to respond to a given subset of inputs.

For a specific input pattern, the neurons compete among themselves and only one of them will win the competition which is called a *winner-takes-all* neuron. The rule will then move the synaptic weight vector of the winning neuron toward the input pattern. In multimedia databases, the feature vectors are the input patterns. By training the neurons with the feature vectors under the competitive

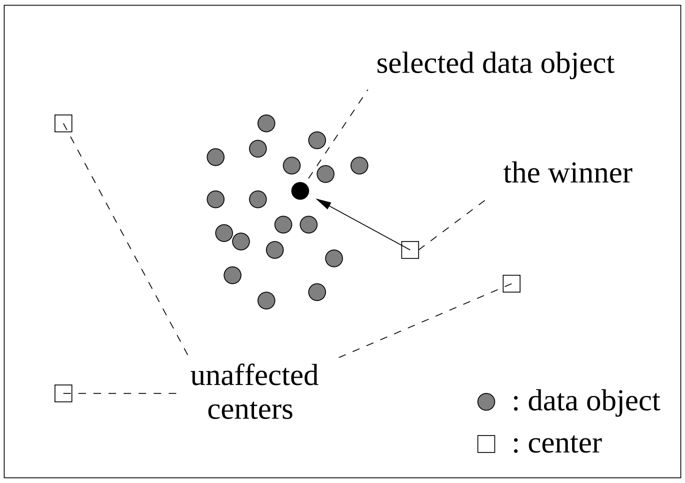


Figure 3.2: Competitive learning clustering.

learning rule, the weight vectors of the neurons will become the cluster centers of the feature vectors.

Let k be the number of clusters, or the number of neurons, and c_i, $i = 1, 2, \ldots, k$, be the cluster center points. The algorithm of competitive learning clustering is outlined as follows.

(Step 0) Initialization: Randomly pick k points as the k initial cluster centers.

(Step 1) Competition: Randomly take a feature vector x from the feature sample set X, the winner-takes-all neuron w is that whose cluster center (weight vector) c_w is the closet to x in the sense of L_2-norm distance (Euclidean distance), i.e.,

$$\|x - c_w\|^2 = \min_i \|x - c_i\|^2 \ . \tag{3.1}$$

(Step 2) Updating Cluster Centers: Update the cluster center c_i by

$$\Delta c_i = \begin{cases} \alpha_w (x - c_i), & \text{if } i = w, \\ 0, & \text{otherwise.} \end{cases} \tag{3.2}$$

where $0 \leq \alpha_w \leq 1$ is the learning rate for the winner-takes-all neuron.

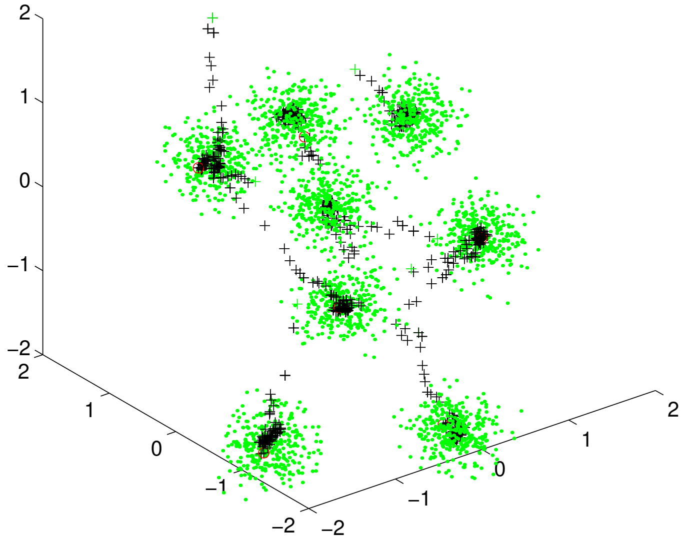


Figure 3.3: Competitive learning clustering for an eight Gaussian mixture distribution of 2560 3-dimensional synthetic feature vectors. The path demonstrates how each unit travels from the initial location to the final approximate cluster center.

Steps 1 and 2 are iterated until the iteration converges or the number of iterations reaches a pre-specified value. The final cluster centers are the results of the competitive learning clustering (see Figure 3.2). Figure 3.3 demonstrates eight cluster centers generated by competitive learning algorithm for an eight Gaussian mixture distribution of feature vectors.

3.2.3 Rival Penalized Competitive Learning (RPCL)

Rival Penalized Competitive Learning (RPCL) [70] is a variant of competitive learning (CL). Instead of moving only the winning neuron, RPCL moves also the first runner-up neuron away from the randomly selected feature vector in each iteration.

Assuming k cluster centers, the basic idea behind RPCL is that in each iteration, the cluster center for the winner's unit is accentuated where as the weight

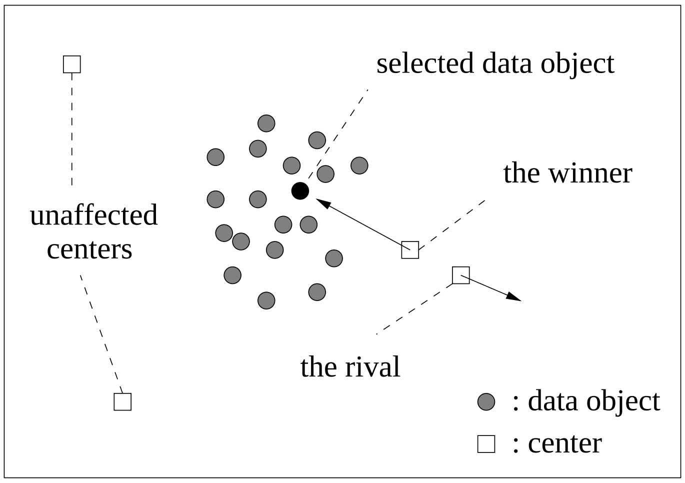


Figure 3.4: RPCL clustering.

for the second winner, or the rival, is attenuated. The remaining $k-2$ centers are unaffected. The winner is defined as the cluster center that is the closest to the randomly selected data object. In our work, we use the special version of the RPCL clustering algorithm when $k=2$. In other words, we only have a winner and a rival (second winner) (see Figure 3.4).

Let k, c_w, c_r to denote the number of clusters, cluster center points for winner and rival clusters respectively. The algorithm of RPCL is outlined as follows.

(Step 0) Initialization: Randomly pick c_1 and c_2 as the initial cluster centers.
(Step 1) Winner-Take-All Rule: Randomly take a feature vector x from the feature sample set X, and for $i = 1, 2, \ldots, k$ let

$$u_i = \begin{cases} 1, & \text{if } i = w \text{ such that } \gamma_w \|x - c_w\|^2 = \min_j \gamma_j \|x - c_j\|^2, \\ -1, & \text{if } i = r \text{ such that } \gamma_r \|x - c_r\|^2 = \min_j \gamma_j \|x - c_j\|^2, \\ 0, & \text{otherwise.} \end{cases} \tag{3.3}$$

where $\gamma_j = n_j / \sum_{i=1}^{k} n_i$ and n_i is the cumulative number of the occurrences of $u_i = 1$. This term is added to ensure that every cluster center will eventually become the winner somehow. It is called the Frequency Sensitive Competitive Learning (FSCL) [2] as an algorithm that reduces the winning rate of the frequent winners.

(Step 2) Updating Cluster Centers: Update the cluster center vector $c_i, i = 1, 2, \ldots, k$ by

$$\Delta c_i = \begin{cases} \alpha_w(x - c_i), & \text{if } u_i = 1, \\ -\alpha_r(x - c_i), & \text{if } u_i = -1, \\ 0, & \text{otherwise.} \end{cases} \tag{3.4}$$

where $0 \leq \alpha_w, \alpha_r \leq 1$ are the learning rates for the winner and rival unit, respectively.

Steps 1 and 2 are iterated until one of the following criteria is satisfied: (1) the iteration converges, or (2) the number of iterations reaches a pre-specified value. Actually pre-specified value is hard to find. We conducted many experiments and obtained the value empirically for our work. In fact, more research on this topic is needed in order to find out a better stopping procedure.

Unlike FSCL, RPCL usually gives one candidate cluster center to one cluster and all the extra candidate centers will go to infinite eventually. With this property, we may make use of RPCL to determine the actual number of natural clusters from an input data set.

3.2.4 General Hierarchical Clustering Methods

Apart from the above three clustering algorithms, we are going to describe some general hierarchical clustering methods here.

A hierarchical clustering algorithm usually produces a series of partitions, from a single whole data cluster to n one-element clusters [18]. There are two main approaches: *agglomerative* and *divisive*. The agglomerative approach proceeds by a series of successive fusions of the n individuals into groups whereas the divisive approach separates the n individuals successively into finer groups. Some general hierarchical clustering methods can be found in [1, 11, 28, 29, 55, 58].

3.3 Why RPCL?

From the clustering methods mentioned in the last section, we try to find out a suitable one for our indexing method. We find that k-means and general hierarchical clustering methods calculate very good data clusters from a given data set, but they are usually computationally intensive. Competitive Learning (CL) and RPCL are heuristic algorithms which produce good approximate cluster centers to a given data set and they are much faster than k-means and general hierarchical clustering methods.

Eventually, we choose RPCL as the clustering algorithm used in our indexing method. The main reasons are:

1. RPCL is a very fast clustering method and faster than CL in general.

2. RPCL gives a good approximate of the cluster centers and we may then calculate the actual data cluster easily form the centers.

3. RPCL can usually find the actual number of clusters from an input data set.

4. Only a low storage is enough for RPCL clustering because only the information of the cluster centers is needed to keep.

After clustering, we can make use of the cluster partitions generated by RPCL to build indexing structures for content-based indexing. There are two approaches to perform top-down clustering and build indexing structures based on the generated cluster partitions: (1) *hierarchical approach* and (2) *non-hierarchical approach*. We are going to present their details in Chapters 4 and 5 respectively.

Chapter 4

Non-hierarchical RPCL Indexing

4.1 The Non-hierarchical Approach

The non-hierarchical approach of our method considers the whole feature vector space each time for clustering by RPCL. We use an example here to explain its basic idea. Given a set of feature vectors, our method clusters the set into 2 clusters at the first time (see Figure 4.1). If four partitions are required at the next time, our method will consider the whole space again and clusters the set into 4 clusters (see Figure 4.2). We find that the latter clusters may not be necessary nested into the former clusters, but this method ensures to obtain the correct natural clusters.

In non-hierarchical RPCL indexing, we usually construct the indexing structure for an input feature vector set in a batch mode. RPCL is firstly used to locate the natural clusters from the feature vector set. Based on the generated cluster partitions, we make use of the inverted file structure for indexing of the feature vectors (see Figure 4.1 and 4.2). For example, given the feature vector space having only two partitions C_1 and C_2 with centers c_1 and c_2 respectively, a feature vector v will belong to C_1 if $D(v, c_1) < D(v, c_2)$ and it will be indexed as "c_1". D is a distance function and we use L_2-norm in our work. Basically, 2^i

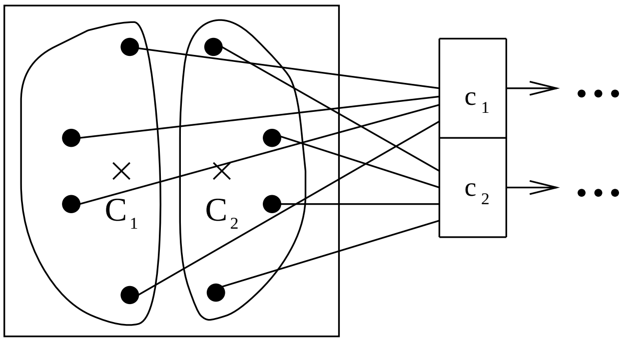


Figure 4.1: Two cluster partitions generated by the non-hierarchical approach. The dots are the database objects whereas the crosses are the centers. An inverted file (the right one) is used for indexing.

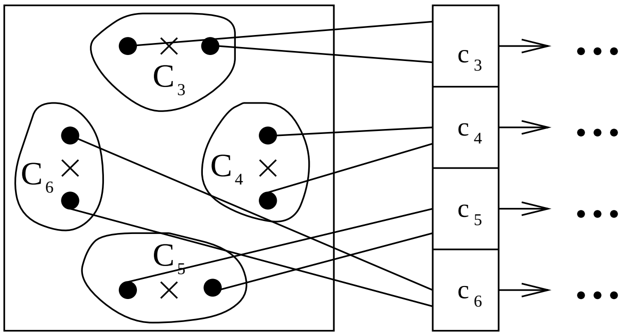


Figure 4.2: Four cluster partitions generated by the non-hierarchical approach. The dots are the database objects whereas the crosses are the centers. An inverted file (the right one) is used for indexing.

clusters are calculated for the i-th level. The whole process starts from the top level with 2 clusters and stops when all the natural clusters are located at the bottom level.

For searching on the indexing structure, a nearest-neighbor query q is compared to all the cluster centers at the user-specified level. All the vectors belonged to the cluster partition whose center is the closest to q will be retrieved.

4.2 Performance Experiments

We conducted several experiments to evaluate the performance of our non-hierarchical approach using RPCL clustering for indexing and retrieval based on its accuracy and efficiency. We also tested others methods: k-means, CL, and a traditional hierarchical partitioning indexing method VP-tree for reference in these experiments.

4.2.1 Experimental Setup

We conducted four different sets of experiments for the four methods: RPCL, k-means, CL, and VP-tree to test their accuracy and efficiency for indexing and retrieval. All of the experiments were conducted on an Ultra Sparc 1 machine running Matlab V4.2c. From Chapters 1 and 2, we know that a cluster of feature vectors is often retrieved as the result of a query for nearest-neighbor search. An indexing method which can locate natural clusters from the input feature vector set accurately and quickly will make nearest-neighbor search more effective and efficient. Therefore, in these experiments, we restrict to retrieve the first visited feature vector cluster or leaf node as the result of a nearest-neighbor query so that, based on the result, we can show that how accurate and efficient the tested methods are to locate natural clusters for indexing and retrieval.

We used two performance measurements: *Recall* and *Precision* in the experiments to measure the accuracy of the tested methods (see Figure 4.3). Given a set of user-specified target database objects, Recall and Precision are defined as:

$$\text{Recall} = \frac{\text{Number of target database objects retrieved}}{\text{Number of target database objects}}, \tag{4.1}$$

$$\text{Precision} = \frac{\text{Number of target database objects retrieved}}{\text{Number of database objects retrieved}}, \tag{4.2}$$

where $0 \leq \text{Recall}, \text{Precision} \leq 1$. Recall shows the ratio of target database objects are actually retrieved out of all the expected target database objects whereas Precision indicates the ratio of target database objects in the retrieved set. For

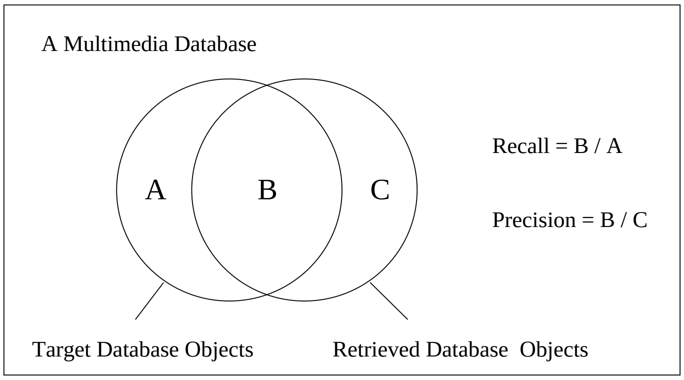


Figure 4.3: Recall and Precision.

example, there are 10 database objects and 4 of them are pre-specified as target database objects. For a query, 5 database objects are retrieved and 3 of them are target database objects. In this case, Recall is 0.75 and Precision is 0.6. Basically, the higher the Recall and Precision, the more accurate the method for retrieval. By using Recall and Precision, we can calculate the accuracy for each of the generated clusters based on the information of its corresponding natural cluster. If we do not use them for accuracy, we can only evaluate the accuracy by using only a small set of queries. Therefore, we use Recall and Precision to evaluate the accuracy of these methods in the experiments.

We used the following three different kinds of feature vector sets in the experiments:

1. **Synthetic Data in Gaussian Distribution:**

   We test our method with synthetic data sets in Gaussian distribution. It is because many distributions can be approximated by using Gaussian distribution. Let $\mu = (\mu_1, \mu_2, \ldots, \mu_n)$ and $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$, we generated the input distribution of the feature vectors from the mixture of n Gaussian distributions $N(\mu, \sigma^2)$ with the generating function defined as $g(x) = 1/(\sigma\sqrt{2\pi})\exp[-[(x-\mu)^2/2\sigma^2]]$, $-\infty < x < \infty$. In our experiments, we used a constant 0.05 for σ. We check for different σ and found that, for larger σ, the mixture groups will be mixed together. For smaller σ, all the data objects in a mixture group will crowd round the group center. Therefore, we select 0.05 for σ which is suitable to generate data sets in Gaussian distribution. Moreover, we let $n = 2, 4, 8, 16$, and 32, but it is not necessary to set n to these values. Besides, we do not let n to more than 32 because each of the mixture groups will become very small in size. Finally, for each input distribution, different numbers of cluster partitions are generated for the input feature vectors for testing.

For the Gaussian distributed database objects, Equations 4.1 and 4.2 can be rewritten as below. Given the set of *a priori* clusters, $C = \{c\}_1^n$ and the set of cluster partitions generated by the tested methods, $C' = \{c'\}_1^m$, the performance measurements Recall and Precision are defined as:

$$\text{Recall} = \sum_{c_i \in C \wedge c'_j \in C'} \frac{c_i \cap c'_j}{\# c_i} , \tag{4.3}$$

$$\text{Precision} = \sum_{c_i \in C \wedge c'_j \in C'} \frac{c_i \cap c'_j}{\# c'_j} , \tag{4.4}$$

where $\# c_i$ denotes the number of elements in the cluster c_i.

2. **Synthetic Data in Uniform Distribution:**

   We also use synthetic data sets in uniform distribution or random distribution as uniform represents the data distribution opposite to Gaussian.

3. **Real Data:**

   Apart from synthetic data, we also use real data in the experiments to test our method in a real world situation. For our experiments, the real data features are the feature vectors extracted from real images. Basically, we firstly find some real images from different kinds of catalogs. By considering the global color information of each image, we calculate an 8-bucket color histogram form the image and transform it into a feature vector. All of the output feature vectors form the real data set for testing.

4.2.2 Experiment 1: Test for Recall and Precision Performance

In the first experiment, we evaluate the accuracy and efficiency of the four tested methods: RPCL, CL, k-means, and VP-tree to build indexing structures for retrieval. We measure the Recall and Precision performance of these methods for accuracy. Moreover, we also kept the time used for pre-processing which includes clustering and indexing of the feature vectors for efficiency. The main aim of this experiment is to find out which tested method has the best overall performance for locating natural clusters for indexing and retrieval.

We use three different kinds of data sets in this experiment: (1) synthetic data in Gaussian distribution, (2) synthetic data in uniform distribution, and (3) real data (see Section 4.2.1 for details). Each of the data sets consists of 2048 8-dimensional feature vectors. This is not a large data set because it is very time consuming for k-means to locate clusters from a large feature vector set. Therefore, we use a relative small data set here for better comparison of these four methods. Besides, we use 8-D feature vectors here because it is not too high and too low for testing the four tested methods and the real data are also in 8-D. Moreover, for each input data set, different numbers of cluster partitions are generated for the input feature vectors by the four tested methods respectively. We conducted 20 trails with different initial starting points of the centers of the to-be generated cluster partitions for these methods to calculate their average Recall and Precision Performance and the average time used for building indexing structure.

We use several tables and figures to present the experimental results of the three different data sets. For the data sets in Gaussian distribution with different mixture groups, Tables 4.1 and 4.2 show the Recall and Precision results. Since

	No. of Generated Clusters (RPCL, CL, k-means, VP-tree)				
$\#MG$	**2**	**4**	**8**	**16**	**32**
2	1.0, 1.0, 1.0, 1.0	.51, .45, .66, .50	.27, .25, .57, .25	.15, .14, .53, .13	.22, .09, .51, .06
4	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	.52, .58, .80, .50	.39, .43, .77, .25	.53, .31, .76, .13
8	1.0, .91, 1.0, .87	1.0, 1.0, 1.0, .71	1.0, 1.0, .94, .56	.75, 1.0, .89, .31	.73, .56, .88, .17
16	.96, .95, 1.0, .90	1.0, .99, 1.0, .86	1.0, 1.0, 1.0, .76	.99, .98, .96, .65	.93, .83, .94, .41
32	.96, .98, .99, .93	.98, .96, 1.0, .86	.97, .87, .99, .80	.98, .87, 1.0, .69	.98, .87, .94, .63

Table 4.1: Recall table for the data sets in Gaussian distributions in Experiment 1. $\#MG$ is the number of Gaussian mixture groups.

	No. of Generated Clusters (RPCL, CL, k-means, VP-tree)				
$\#MG$	**2**	**4**	**8**	**16**	**32**
2	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0
4	.50, .50, .50, .50	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0
8	.25, .23, .25, .15	.46, .50, .50, .20	1.0, 1.0, .93, .36	1.0, 1.0, .97, .63	1.0, 1.0, 1.0, .79
16	.11, .12, .13, .10	.26, .24, .27, .16	.54, .54, .61, .21	.97, .93, .88, .39	.98, .85, .97, .68
32	.06, .05, .06, .05	.11, .11, .14, .08	.25, .18, .26, .13	.51, .39, .56, .21	.94, .71, .87, .41

Table 4.2: Precision table for the data sets in Gaussian distributions in Experiment 1. $\#MG$ is the number of Gaussian mixture groups.

the time used for pre-processing is independent to the input distribution of the feature vector space, we simply show only the time used for pre-processing 2048 feature vectors with 16 Gaussian mixtures of the input distribution in Figure 4.4. For the data set in uniform distribution and the real data set, we can simply use Figures 4.5 and 4.6 to present their results respectively. Moreover, we use Tables 4.3, 4.4, and 4.5 here to show the main observations of this experiment.

Based on the above experimental results, a brief discussion of the performance of the four methods for content-based retrieval in multimedia databases is given below:

Measures	**RPCL**	**CL**	**k-means**	**VP-tree**
Recall	high	middle	highest	lowest
Precision	high	middle	highest	lowest
Preprocessing Speed	highest	high	lowest	middle

Table 4.3: Comparison of the average performance of the four methods for indexing and retrieval with data sets in Gaussian distributions.

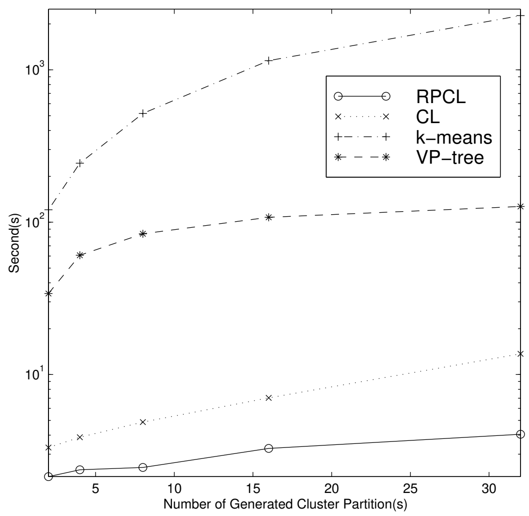


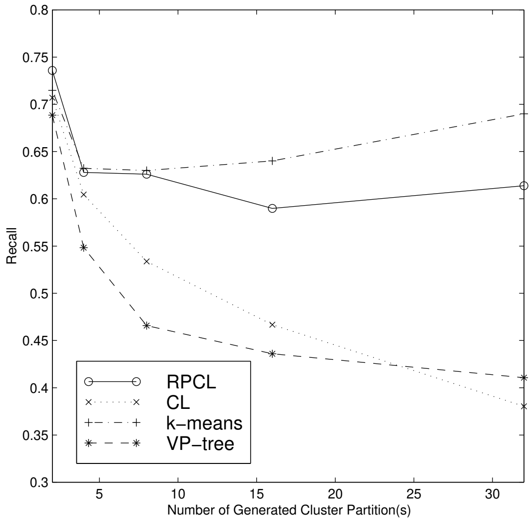
Figure 4.4: Time used for the pre-processing of the 2048 feature vectors with 16 Gaussian mixtures of the input distribution.

Measures	RPCL	CL	k-means	VP-tree
Recall	high	low	highest	low
Precision	high	low	highest	low
Preprocessing Speed	highest	high	lowest	middle

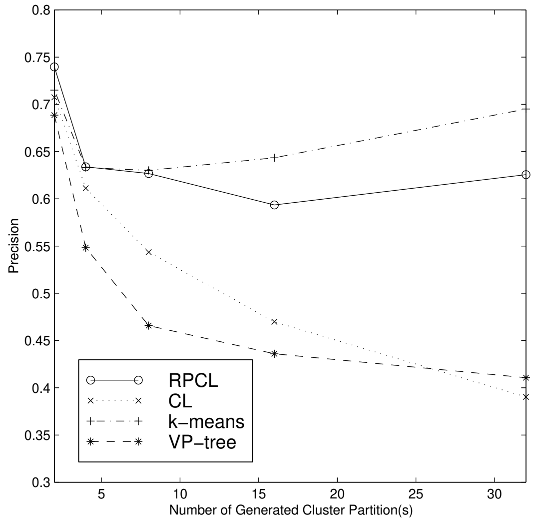
Table 4.4: Comparison of the average performance of the four methods for indexing and retrieval with the uniform data set.

Measures	RPCL	CL	k-means	VP-tree
Recall	high	low	high	high
Precision	high	low	high	high
Preprocessing Speed	highest	middle	lowest	middle

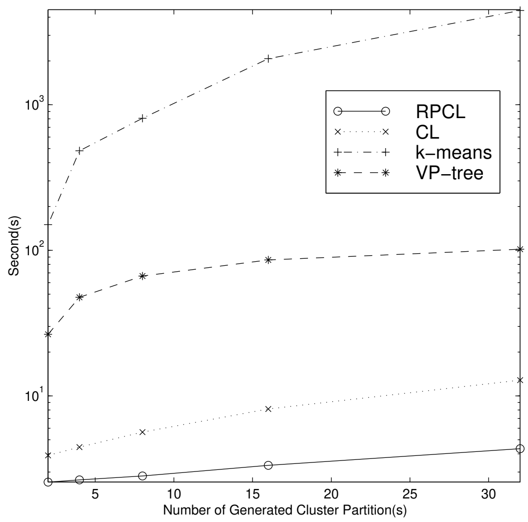
Table 4.5: Comparison of the average performance of the four methods for indexing and retrieval with a given real data set.



(a)

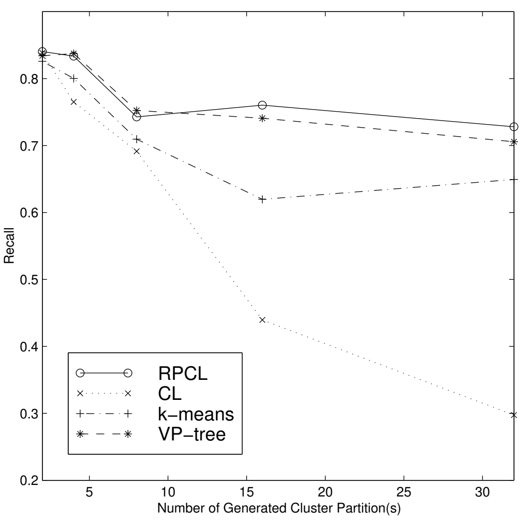


(b)

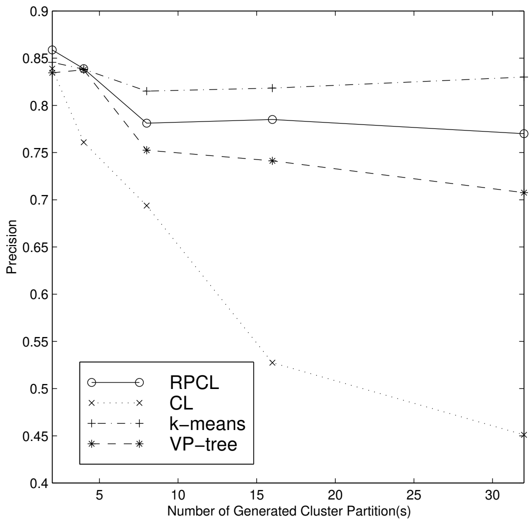


(c)

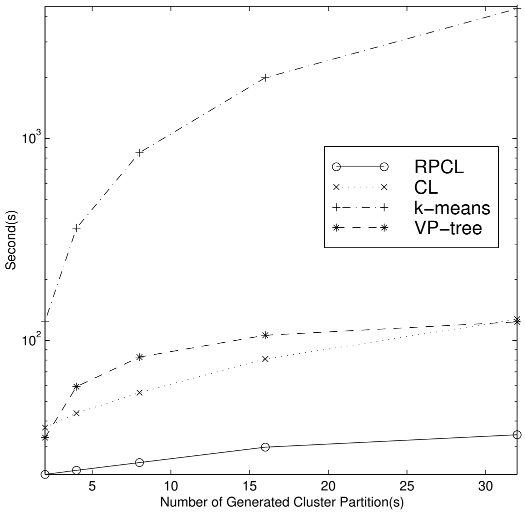
Figure 4.5: Results for the uniform data set in Experiment 1. (a) The Recall results. (b) The Precision results. (c) The pre-processing time.



(a)



(b)



(c)

Figure 4.6: Results for the real data set in Experiment 1. (a) The Recall results. (b) The Precision results. (c) The pre-processing time.

1. **Boundary Problem:**

   For the data sets in Gaussian distribution, the Recall and Precision values of RPCL, CL, and k-means are higher than VP-tree. The reason is that VP-tree cannot handle the boundary problem well. This is a problem when a requested nearest-neighbor query falls near the cluster partition boundary. Since VP-tree does not pay attention to the input distribution, a similar feature vector near to the query may be clustered into another cluster partition. However, the other tested methods try to locate natural clusters from the input data set so that they may handle the boundary problem better. As a result, the Recall and Precision values of the VP-tree method are lower than the other tested methods.

2. **Low Recall Performance When $\#GC > \#MG$:**

   There is a problem for the k-means, CL, and RPCL methods when the number of generated clusters ($\#GC$) is greater than the number of Gaussian mixture groups ($\#MG$) of the input distribution. We may find that the Recall values are relatively low in this case. It is because multiple generated cluster partitions may be bunched together spatially. This leads to an incorrect assessment of cluster partitions since only a few target database objects can be retrieved.

3. **Low Precision Performance When $\#GC < \#MG$:**

   When the number of generated clusters ($\#GC$) is less then the number of Gaussian mixture groups ($\#MG$), we find that the precision values of the tested methods are relatively low. The main reason is two or more mixture groups may be clustered into the same cluster. Therefore, the cluster containing the target database objects may contain many non-target database objects. The precision for retrieving this cluster as the result of a query will be low.

4. **Performance for Data in Uniform Distribution:**

   For the data set in uniform distribution, the Recall and Precision values are not very high. It is because there are no explicit natural clusters found in the distribution of the data set and clustering is not quite useful for this case. As a result, RPCL, CL, and k-means do not have high Recall and Precision Performance.

5. **Performance for the Real Data Set:**

   For the real data set, it is expected that the Recall and Precision values are in between those for data sets in two extreme data distributions: Gaussian and uniform. It is because the distribution of the real data set is usually in between Gaussian and uniform. Therefore, we find that the overall Recall and Precision values are higher than the uniform one, but lower than the Gaussian one in general.

6. **K-means - an Accurate but Slow Method:**

   K-means gives the best average Recall and Precision performance among the four tested methods, but it is the slowest. It is because the k-means algorithm often recomputes the centers of the cluster partitions when clustering. Hence, it is computationally intensive.

7. **RPCL - a Fast Method with Satisfactory Recall and Precision Performance:**

   RPCL gives satisfactory results and it is much faster than k-means for preprocessing. It produces a good cluster center approximation so that it can locate natural clusters well and gives good Recall and Precision results. Moreover, it is a heuristic algorithm for clustering so that it is faster than k-means for more than several hundred times in general.

In summary, RPCL gives the best overall performance among the four tested methods for indexing and retrieval in this experiment. Therefore, we will concentrate on RPCL in the following three experiments to evaluate the efficiency and accuracy of RPCL for indexing and retrieval with other different parameters such as size and dimensionality and find out how these parameters affect the efficiency and accuracy of our method.

4.2.3 Experiment 2: Test for Different Sizes of Input Data Sets

In this experiment, we test the accuracy and efficiency of RPCL for indexing and retrieval with different sizes of input feature vector sets. We measure the Recall and Precision performance of our method for accuracy and record the time used for pre-processing for efficiency. We use two different kinds of data sets in this experiment: (1) synthetic data in Gaussian distribution and (2) synthetic data in uniform distribution (see Section 4.2.1 for details). The data sets are 8-dimensional feature vector sets with sizes varying from 1024 to 40960. For each input data set, different numbers of cluster partitions are generated for the experiment. We conducted 20 trails with different initial starting points of the centers of the to-be generated cluster partitions for RPCL to calculate its average Recall and Precision Performance and the average time used for building indexing structure.

We use several figures and tables to present the results of this experiment. For the data sets in Gaussian distribution with different mixture groups, Table 4.6 shows the Recall and Precision results. For better illustration, we show the results for the data sets with feature vectors having 16 Gaussian mixture groups by using Figure 4.7. Moreover, it shows the time used for pre-processing this data set. On the other hand, Table 4.7 and Figure 4.8 present all the results for the data sets in uniform distribution.

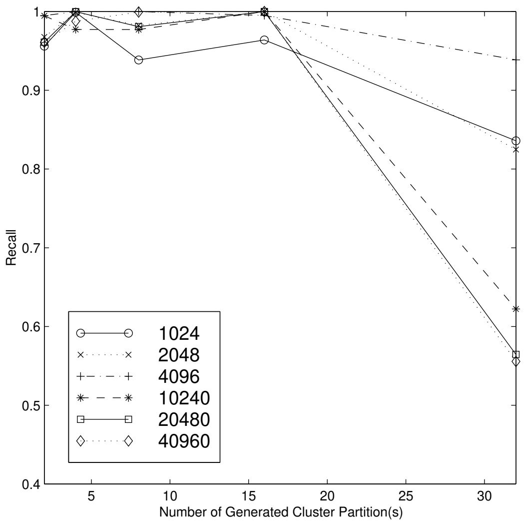
	No. of Generated Clusters				
$\#MG$	**2**	**4**	**8**	**16**	**32**
2	1.0	.47	.28	.23	.17
	1.0	.47	.28	.12	.06
	1.0	.51	.26	.11	.11
	1.0	.47	.30	.12	.06
	1.0	.47	.29	.14	.06
	1.0	.53	.29	.14	.06
4	1.0	1.0	.54	.67	.57
	1.0	1.0	.50	.37	.43
	1.0	1.0	.64	.25	.14
	1.0	1.0	.62	.25	.15
	1.0	1.0	.64	.25	.12
	1.0	1.0	.65	.26	.13
8	1.0	1.0	1.0	1.0	.79
	.99	1.0	1.0	.78	.64
	1.0	.92	1.0	.53	.56
	1.0	1.0	1.0	.57	.31
	1.0	1.0	1.0	.62	.27
	1.0	1.0	1.0	.57	.27
16	.96	1.0	.94	.96	.84
	.97	1.0	.98	1.0	.82
	1.0	1.0	1.0	1.0	.94
	.99	.98	.98	1.0	.62
	.96	1.0	.98	1.0	.56
	.96	.99	1.0	1.0	.56
32	.97	.96	.97	.95	.95
	.98	.95	.97	.98	1.0
	.98	.90	.96	.96	1.0
	.97	.98	.98	.99	1.0
	.99	1.0	.97	1.0	.99
	.99	.98	1.0	.99	1.0

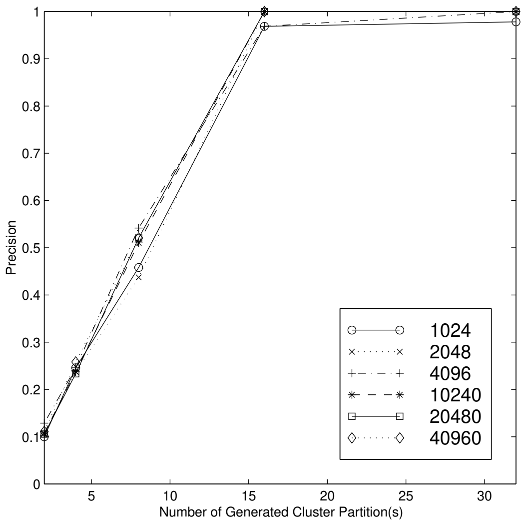
(a)

	No. of Generated Clusters				
$\#MG$	**2**	**4**	**8**	**16**	**32**
2	1.0	1.0	1.0	1.0	1.0
	1.0	1.0	1.0	1.0	1.0
	1.0	1.0	1.0	1.0	1.0
	1.0	1.0	1.0	1.0	1.0
	1.0	1.0	1.0	1.0	1.0
	1.0	1.0	1.0	1.0	1.0
4	.50	1.0	1.0	1.0	1.0
	.50	1.0	1.0	1.0	1.0
	.50	1.0	1.0	1.0	1.0
	.50	1.0	1.0	1.0	1.0
	.50	1.0	1.0	1.0	1.0
	.50	1.0	1.0	1.0	1.0
8	.27	.58	1.0	1.0	1.0
	.23	.58	1.0	1.0	1.0
	.27	.46	1.0	1.0	1.0
	.27	.50	1.0	.97	1.0
	.25	.50	1.0	1.0	1.0
	.25	.58	1.0	1.0	1.0
16	.10	.25	.46	.97	.98
	.11	.24	.44	1.0	1.0
	.13	.25	.54	.97	1.0
	.11	.24	.51	1.0	1.0
	.11	.23	.52	1.0	1.0
	.11	.26	.52	1.0	1.0
32	.06	.10	.20	.44	.83
	.06	.09	.23	.50	.97
	.05	.10	.19	.46	.97
	.05	.11	.23	.42	.97
	.05	.11	.19	.46	.97
	.05	.10	.20	.47	.98

(b)

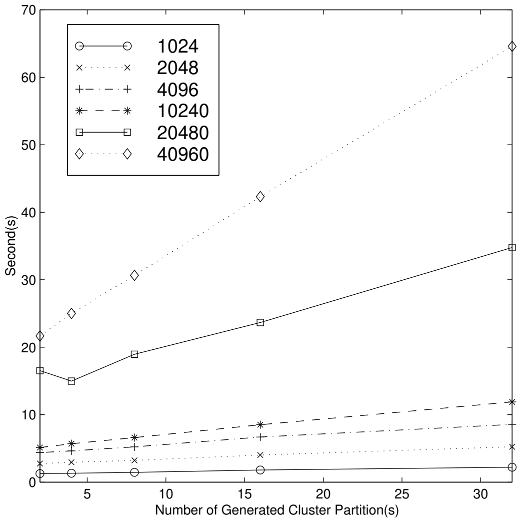
Table 4.6: Results for the data sets in Gaussian distributions in Experiment 2. (a) The Recall table. (b) The Precision table. Each entry of the tables is a column of 6 values for 6 different sizes of the data sets: 1024, 2048, 4096, 10240, 20480, and 40960. $\#MG$ is the number of Gaussian mixture groups.





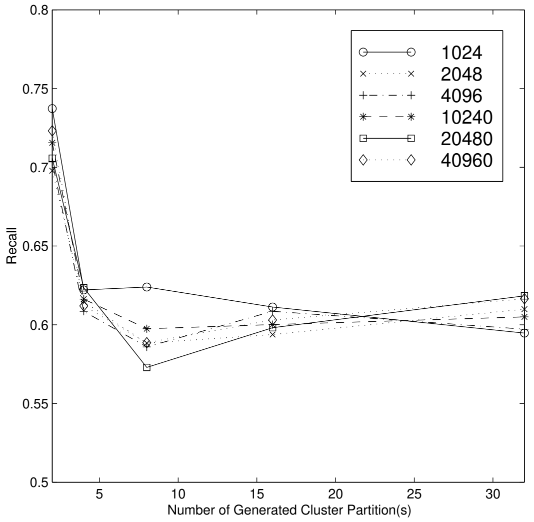
(a)

(b)

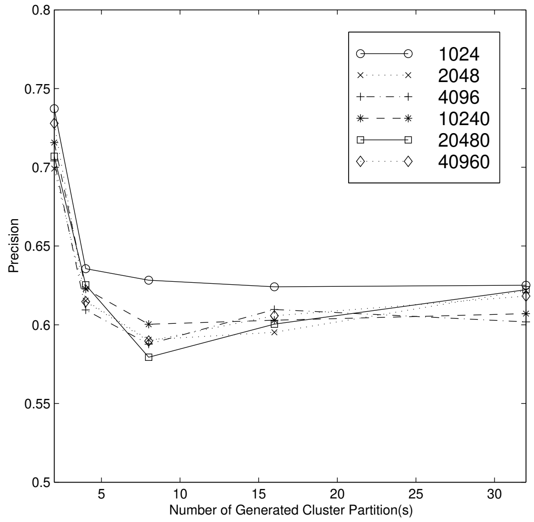


(c)

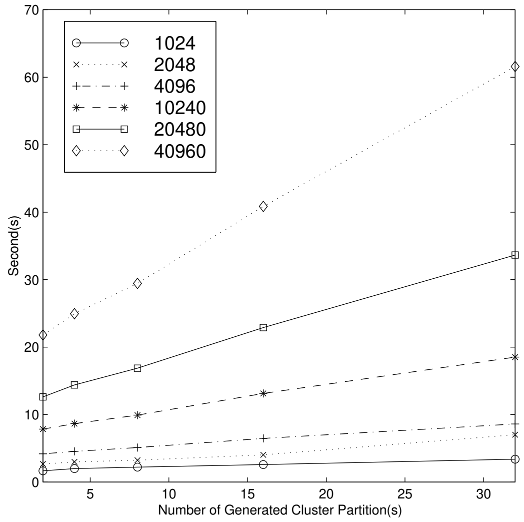
Figure 4.7: Results for the data sets in Gaussian distribution with 16 mixture groups in Experiment 2. (a) The Recall results. (b) The Precision results. (c) The pre-processing time.



(a)



(b)



(c)

Figure 4.8: Results for the uniform data sets in Experiment 2. (a) The Recall results. (b) The Precision results. (c) The pre-processing time.

Size of Data Set	# Generated Clusters 2	4	8	16	32
1024	.73	.62	.62	.61	.59
2048	.70	.61	.59	.59	.61
4096	.70	.61	.59	.61	.60
10240	.72	.62	.60	.60	.61
20480	.71	.62	.57	.60	.62
40960	.72	.61	.59	.60	.62

(a)

Size of Data Set	# Generated Clusters 2	4	8	16	32
1024	.74	.64	.63	.62	.63
2048	.70	.61	.59	.60	.62
4096	.70	.61	.59	.61	.60
10240	.72	.62	.60	.60	.61
20480	.71	.63	.58	.60	.62
40960	.73	.61	.59	.61	.62

(b)

Table 4.7: Results for the data sets in uniform distribution in Experiment 2. (a) The Recall table. (b) The Precision table.

From the experimental results, we find that the accuracy is unaffected by the sizes of the data sets in general. From the tables and figures, we can see that the Recall and Precision values are almost the same for different data set sizes provided that the other parameters are fixed and more pre-processing time is needed for larger data set. Therefore, it is concluded that the accuracy of the non-hierarchical RPCL indexing method is independent to the size of the input data set.

4.2.4 Experiment 3: Test for Different Numbers of Dimensions

Apart from different sizes of data sets, we also test the performance of RPCL for indexing with feature vectors having different numbers of dimensions in terms of Recall and Precision for accuracy and the pre-processing time for efficiency. Two different kinds of data sets are used in this experiment: (1) synthetic data in Gaussian distribution and (2) synthetic data in uniform distribution (see Section 4.2.1 for details). All the data sets are fixed to have 10240 feature vectors with different numbers of dimensions such as 4, 8, 16, and 32. We fixed the size of each data set to 10240 as it is not too large or too small for testing and we do not use data more than 32-D because it is not so efficient for our method to work with

#MG	No. of Generated Clusters (DIM = 4, 8, 16, 32) 2	4	8	16	32
2	1.0, 1.0, 1.0, 1.0	.51, .49, .53, .55	.25, .25, .26, .28	.13, .12, .14, .11	.07, .07, .06, .06
4	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	.61, .48, .62, .62	.27, .27, .27, .26	.14, .14, .13, .12
8	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	.50, .49, .54, .64	.32, .24, .27, .28
16	1.0, .96, .98, 1.0	1.0, 1.0, 1.0, 1.0	.98, 1.0, 1.0, 1.0	.98, 1.0, 1.0, 1.0	.58, .57, .57, .57
32	.94, .99, .99, .98	.93, .97, .99, .99	.93, .95, 1.0, .98	.96, .97, 1.0, 1.0	.97, 1.0, 1.0, 1.0

Table 4.8: The Recall table for the data sets in Gaussian distributions in Experiment 3.

#MG	No. of Generated Clusters (DIM = 4, 8, 16, 32) 2	4	8	16	32
2	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0
4	.50, .50, .50, .50	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0
8	.23, .27, .25, .57	.67, .58, .54, .80	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0, 1.0
16	.11, .09, .12, .21	.23, .20, .22, .25	.48, .48, .50, .54	.80, 1.0, 1.0, 1.0	.92, 1.0, 1.0, 1.0
32	.05, .05, .07, .06	.08, .11, .11, .11	.15, .20, .25, .32	.39, .51, .53, .54	.72, 1.0, 1.0, 1.0

Table 4.9: The Precision table for the data sets in Gaussian distributions in Experiment 3.

such high dimensional data. Moreover, for each input data set, different numbers of cluster partitions are generated for the experiment. We conducted 20 trails with different initial starting points of the centers of the to-be generated cluster partitions for RPCL to calculate its average Recall and Precision Performance and the average time used for building indexing structure.

We use several figures and tables to present the results of this experiment. For the data sets in Gaussian distribution with different mixture groups, Tables 4.8 and 4.9 show the Recall and Precision results. Moreover, we show the results for the data set with feature vectors having 16 Gaussian mixture groups by using Figure 4.9 for better illustration. Furthermore, Table 4.10 and Figure 4.10 present the results for the data sets in uniform distribution.

By increasing the number of dimensions, the experimental results show that the accuracy is not affected for the data sets in Gaussian distributions, but it may be lowered for the data sets in uniform distribution. The relatively lower Recall and Precision results found for uniform data because there are no explicit natural

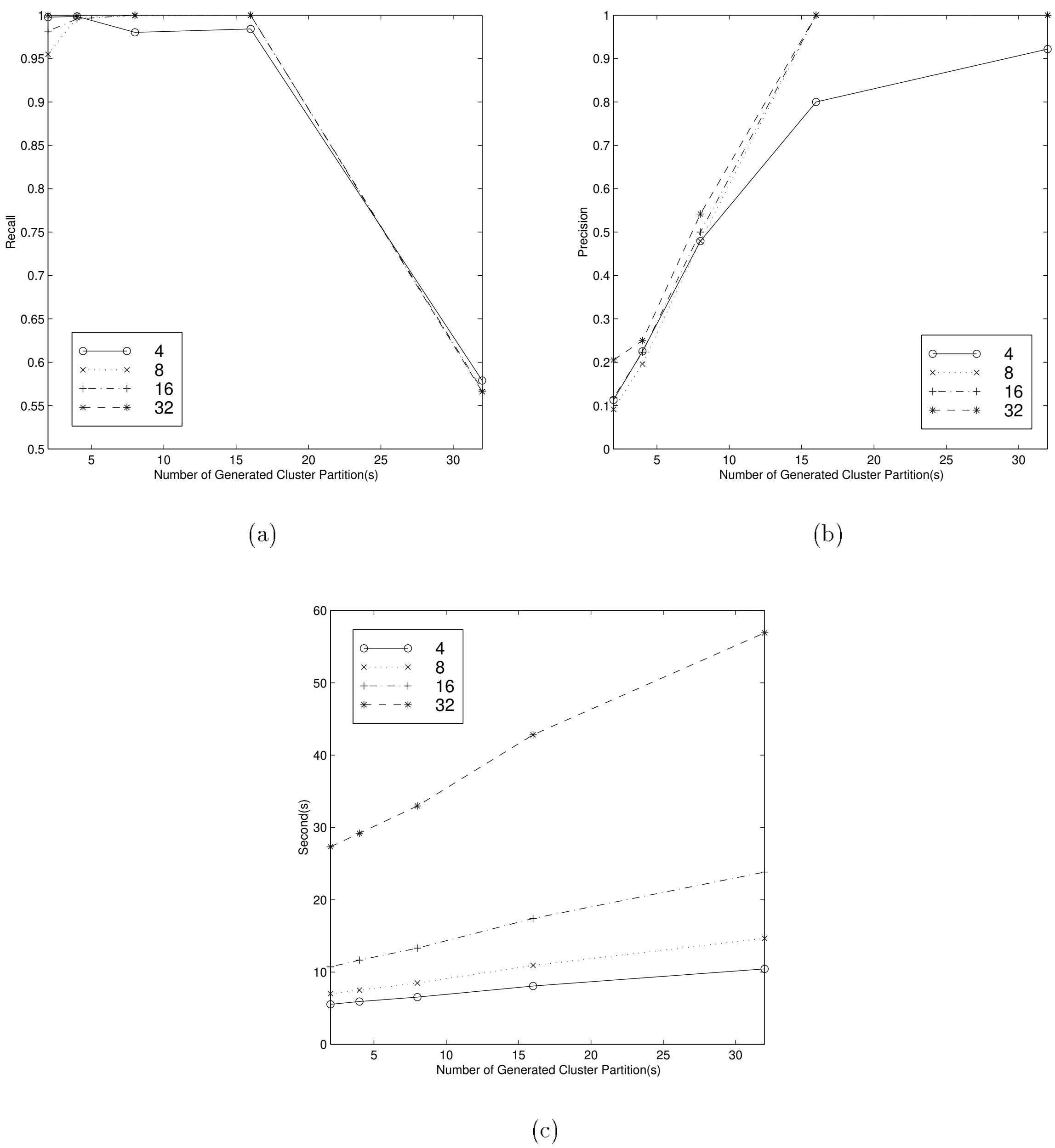
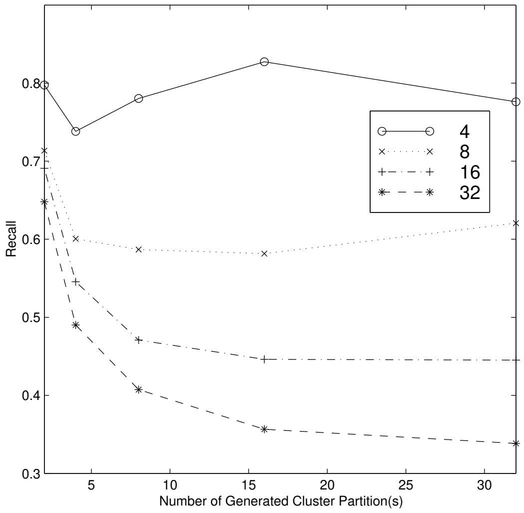
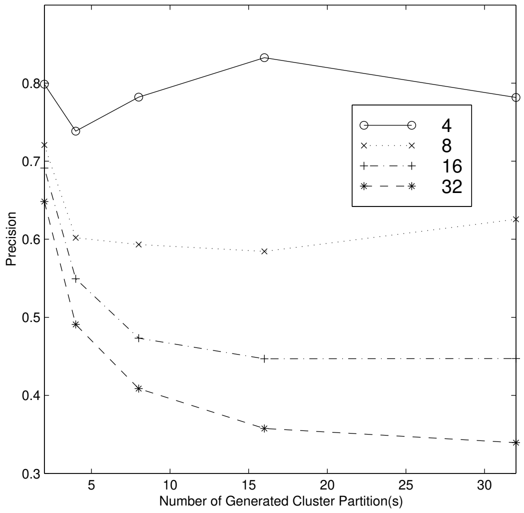


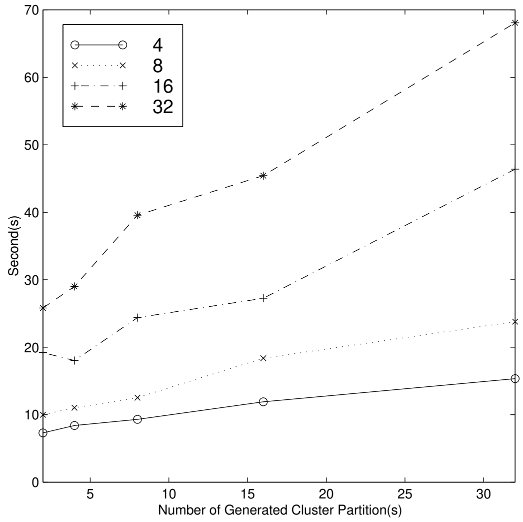
Figure 4.9: Results for the data sets in Gaussian distribution with 16 mixture groups in Experiment 3. (a) The Recall results. (b) The Precision results. (c) The pre-processing time.



(a)



(b)



(c)

Figure 4.10: Results for the uniform data sets in Experiment 3. (a) The Recall results. (b) The Precision results. (c) The pre-processing time.

DIM	# Generated Clusters 2	4	8	16	32
4	.80	.74	.78	.83	.78
8	.71	.60	.59	.58	.62
16	.69	.55	.47	.45	.45
32	.65	.49	.41	.36	.34

(a)

DIM	# Generated Clusters 2	4	8	16	32
4	.80	.74	.78	.83	.78
8	.72	.60	.59	.58	.63
16	.69	.55	.47	.45	.45
32	.65	.49	.41	.36	.34

(b)

Table 4.10: Results for the uniform data sets in Experiment 3. (a) The Recall table. (b) The Precision table.

clusters for RPCL to locate. Therefore, we can conclude that our method is more suitable for data sets with distributions similar to Gaussian distribution.

4.2.5 Experiment 4: Compare with Actual Nearest-neighbor Results

In this experiment, we compare the results given by our method with the actual nearest-neighbor results in order to check the actual accuracy of our method. In the first three sets of experiments, we mainly evaluate the Recall and Precision performance of the tested methods. We want to find out the (accuracy) percentage of the database objects retrieved by our method can also be found in the actual nearest-neighbor results in the experiment for accuracy.

We use three different kinds of data sets in this experiment: (1) synthetic data in Gaussian distribution, (2) synthetic data in uniform distribution, and (3) real data (see Section 4.2.1 for details). Each of the data sets contains 8-dimensional 10240 feature vectors. Moreover, for each input data set, different numbers of cluster partitions are generated for the experiment. We conducted 20 trails with different initial starting points of the centers of the to-be generated cluster partitions for RPCL to find out the results of the given queries. The results of this experiment are presented by Tables 4.11, 4.12, and 4.13.

	No. of Generated Clusters				
$\#MG$	**2**	**4**	**8**	**16**	**32**
2	88.14	56.14	40.72	33.40	29.55
4	73.42	84.34	56.58	40.85	29.30
8	65.40	60.97	79.10	54.41	39.58
16	62.14	54.14	57.22	75.34	45.04
32	64.05	49.82	49.42	49.88	73.08

Table 4.11: Accuracy percentages for the data sets in Gaussian distributions in Experiment 4. $\#MG$ is the number of Gaussian mixture groups.

No. of Generated Clusters				
2	**4**	**8**	**16**	**32**
59.67	42.43	35.09	32.08	28.91

Table 4.12: Accuracy percentages for the uniform data set in Experiment 4.

There are several observations for the accuracy percentages of the three different kinds of data sets similar to those in Experiment 1. For data sets in Gaussian distributions, when the number of generated clusters ($\#GC$) is the same as the number of Gaussian mixture groups ($\#MG$) of the input distribution, the percentages are higher than the others. The reason is the same as the one in Experiment 1. Another observation with the same reason as the one in Experiment 1 is that the percentages for the uniform data set are the lowest and those for the real data set are in the middle. These same observations show that Recall and Precision are good measurements for accuracy.

From the experimental results, the accuracy percentages (for first cluster retrieval) are relatively high (73%-88%) for the data sets in Gaussian distribution when $\#GC = \#MG$, but we find that the larger the number of generated cluster partitions, the lower the accuracy percentage. It is because the chance of the occurrence of the boundary problem is higher when there are many generated

No. of Generated Clusters				
2	**4**	**8**	**16**	**32**
67.72	56.97	48.39	44.76	37.51

Table 4.13: Accuracy percentages for the real data set in Experiment 4.

clusters. It shows that our method can lessen the boundary problem, but it still cannot solve it completely.

4.3 Chapter Summary

In summary, we propose to use RPCL to produce cluster partitions in a non-hierarchical fashion for content-based indexing. From the experimental results, we show that our method (RPCL) gives good searching performance and it is the fastest method to build for indexing among the tested methods.

Our method using the non-hierarchical approach for indexing seems to be a good method, but there are still some limitations. First, it is not so efficient to perform insertion and deletion in our indexing method. Since our method uses a non-hierarchical indexing structure, there is no relationship in between two different levels' nodes. We have to find the target node at each level individually for insertion and deletion. Second, we find that our method still cannot solve the boundary problem completely. It does not give 100% nearest-neighbor result for a query in general. In order to lessen the above problems, we propose a hierarchical approach of our method in Chapter 5.

Chapter 5

Hierarchical RPCL Indexing

5.1 The Hierarchical Approach

In this chapter, we are going to present the hierarchical approach of our indexing method. This method uses a hierarchical structure for indexing so that relationship can be found in the nodes between two levels and it helps us to update the indexing structure. Moreover, we can perform backtracking in the hierarchical indexing structure so that 100% nearest-neighbor results can be obtained. In short, we propose this hierarchical approach here to solve the limitations found in the non-hierarchical RPCL indexing method.

The hierarchical approach transforms a feature vector space into a sequence of nested clusters. It clusters the vectors which are in a cluster of the previous level (see Figure 5.1).

The hierarchical clustering approach can be formulated as follows. Let the feature vector set X with n vectors be

$$X = \{x_i\}_{i=1}^{n} \ .$$

A cluster, C, of X breaks X into subsets $C_1, C_2, \ldots, C_m$ satisfying the following:

$$C_i \cap C_j = \emptyset, \quad 1 \leq i, j \leq m, i \neq j \ ,$$

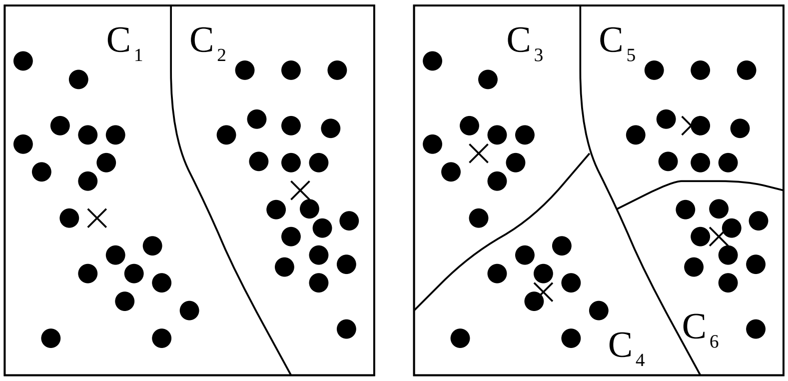


Figure 5.1: Hierarchical clustering. C_3 and C_4 are the clusters inside C_1. C_5 and C_6 are the clusters inside C_2. The dots represent the database objects (feature vectors). The crosses represent the centers.

$$C_1 \cup C_2 \cup \ldots \cup C_m = X \ .$$

Cluster B is nested into cluster C if every component of B is a proper subset of a component of C. A hierarchical clustering is a sequence of clusters in which each cluster is nested into the previous cluster in the sequence.

After clustering, there exists a mapping function that maps the generated clusters to a binary indexing structure. For example, all the feature vectors are in one cluster at the root level and there are 2^i subtrees (clusters) at depth i (see Figure 5.2).

At the top level, a nearest-neighbor query q is compared to the centers of the clusters in the immediate lower level. The cluster with center closest to the query point q is selected. The elements in the selected cluster will be the result of the query if they satisfy the criteria of the nearest-neighbor search. Otherwise, the search will proceed to the lower levels. In Section 5.5, we will present how to make use of a branch-and-bound method to speed up the searching.

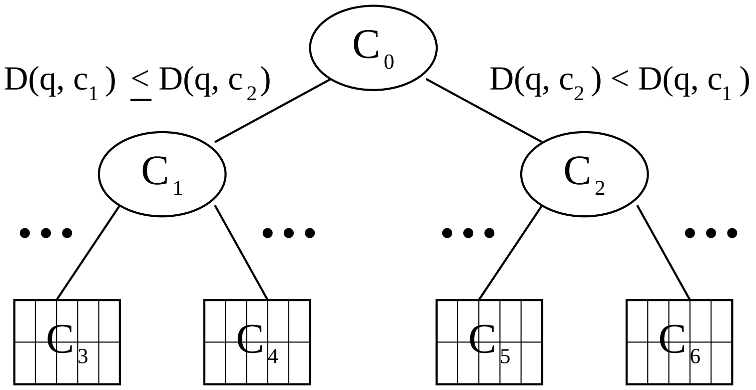


Figure 5.2: The indexing structure for the hierarchical clustering in Figure 5.1. C_0 is the root node which contains all the dots in the data set used in Figure 5.1. $D(q, c_i)$ means the L_2-norm distance between nearest-neighbor query q and the center c_i of the cluster C_i.

5.2 The Hierarchical RPCL Binary Tree (RPCL-b-tree)

In this section, we will introduce the hierarchical RPCL binary tree (RPCL-b-tree). We will also outline the procedure of building the tree structure from the RPCL clusters.

Given a set of data, we can perform top-down RPCL clustering and build a RPCL-b-tree based on the RPCL clusters. The basic idea is that we apply RPCL to cluster the data set into two sub-clusters each time and then continue to do RPCL clustering hierarchically to each of the sub-clusters until each of the final sub-clusters contains less than a pre-specified number of data points. With these RPCL clusters, we can build a RPCL-b-tree easily.

RPCL-b-tree is a hierarchical RPCL binary tree structure. There are two kinds of nodes in the tree: *leaf node* and *non-leaf node*.

Definition 5.1 (Leaf Node) *A leaf node contains a cluster of at most M data points calculated by RPCL clustering. M is the maximum number of data in a leaf node.*

Definition 5.2 (Non-leaf Node) *A non-leaf node contains 2 entries of the form,*

$$(F_i, ChildPtr_i) \ ,$$

where $i = 1$ and 2, $ChildPtr_i$ is a pointer to its i-th child node, and F_i is a tuple summarizing the information of the cluster of the i-th child node,

$$F_i = (N_i, LS_i, RpclCenter_i) \ ,$$

where

1. *N_i is the number of data points in the cluster,*
2. *LS_i is the linear sum of the N_i data points (i.e. $LS_i = \sum_{i=1}^{N_i} X_i$, $\{X_i\}_{i=1}^{N_i}$ is the cluster of data points), and*
3. *$RpclCenter_i$ is the cluster center calculated by RPCL clustering.*

The tuple F_i is similar to the *Clustering Feature* (CF) in [72]. We keep this tuple in each non-leaf node because it can help us to calculate the centroid of the cluster for retrieval. The centroid C of a cluster $\{X_i\}_{i=1}^{N}$ is defined as: $C = (\sum_{i=1}^{N} X_i)/N$, which can be easily computed from information in the tuple.

Based on Definitions 5.1 and 5.2, RPCL-b-tree satisfies the following properties.

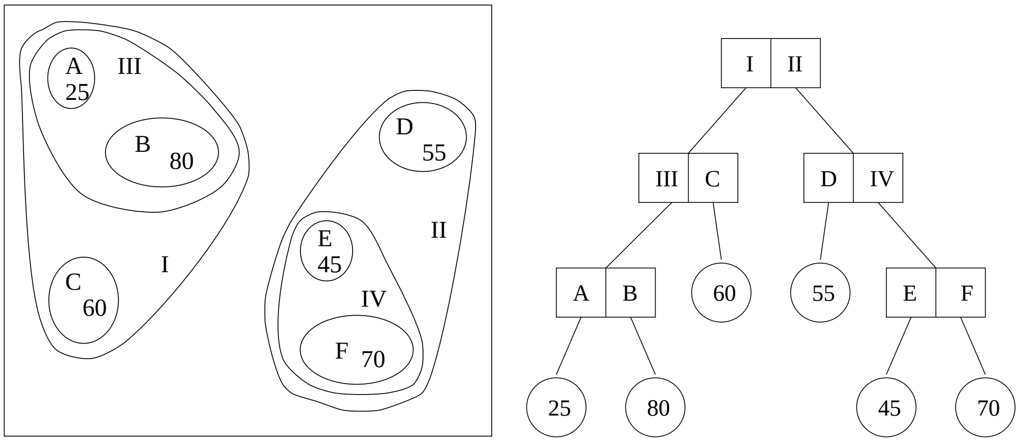
Property 5.1 *Each leaf node contains between 1 and M data point(s).*

Property 5.2 *Each non-leaf node has two children.*

Property 5.3 *It has been proven in [72] that N and LS for a non-leaf node can be easily calculated from the clustering information of its child nodes with $F_1 = (N_1, LS_1, RpclCenter_1)$ and $F_2 = (N_2, LS_2, RpclCenter_2)$ as:*

$$N = N_1 + N_2 \ ,$$

$$LS = LS_1 + LS_2 \ .$$



(a) (b)

Figure 5.3: An example of RPCL-b-tree. (a) shows the input data whereas (b) shows the corresponding RPCL-b-tree. A to F are the RPCL clusters for leaf nodes. The number in each cluster indicates its size. I to IV are the intermediate RPCL clusters for non-leaf nodes. Note that the node size is 100 in this RPCL-b-tree.

After introducing the RPCL-b-tree, we now present the algorithm for building the hierarchical binary tree by using RPCL clustering (Figure 5.3 shows an example of RPCL-b-tree).

Algorithm 5.1 BuildTree(D, P, M)

▷ *Input: A set of data objects D, a RPCL-b-tree node P (P is empty at the first time), and the maximum node size M*

▷ *Output: A RPCL-b-tree*

if D's size is greater than M **then do**
  create a non-leaf node Q
  add Q as a child node of P if any
  use RPCL to cluster D into two sub-sets D_1 and D_2
  BuildTree(D_1, Q, M)
  BuildTree(D_2, Q, M)
  return Q
else
  create a leaf node L for D
  add L as a child node of P if any
  calculate the clustering information of D and store it in the corresponding entry of P

return L
end if

5.3 Insertion

Our method not only works in a batch mode, but also allows us to insert data to the indexing structure. The algorithm for inserting a single data point p to the tree is shown as follows.

Algorithm 5.2 Insert(T, p, M)

▷ *Input: A RPCL-b-tree T, a to-be inserted data object p, and the maximum node size M*
▷ *Output: An updated RPCL-b-tree*

$N \leftarrow$ the root node of T
while N is not a leaf node **do**
    $N \leftarrow$ the node with center closest to p among its child nodes if any
end while
associate p to N
update the center of N according to the RPCL clustering rules
if N's size is larger than M **then do**
    split the node into 2 sub-nodes by using RPCL
end if
update the information of N's ancestors if necessary

The performance of the indexing tree for searching may be reduced after some individual data point insertions. The more the insertions, the worse the performance. The reason is that the insertion algorithm dose not fully consider the overall distribution of the inserted data point and the original data so that it cannot guarantee to keep the natural clusters. The searching performance will then be worse (See Table 5.1 and Figure 5.4 for an example. The more the number of distance computations, the worse the searching performance. See Section 5.6.1 for details.). As a result, we may have to rebuild the indexing structure after a certain amount of data points have been inserted.

# points pre-processed	# points inserted	avg. # of distance computations
10000	0	25.8
9000	1000	33.1
8000	2000	33.95
6000	4000	37.75
4000	6000	44.80
2000	8000	41.65
0	10000	41.60

Table 5.1: The average searching performance of 20 nearest-neighbor queries on the RPCL-b-trees built in different ways of data insertions with the same 10000 data objects.

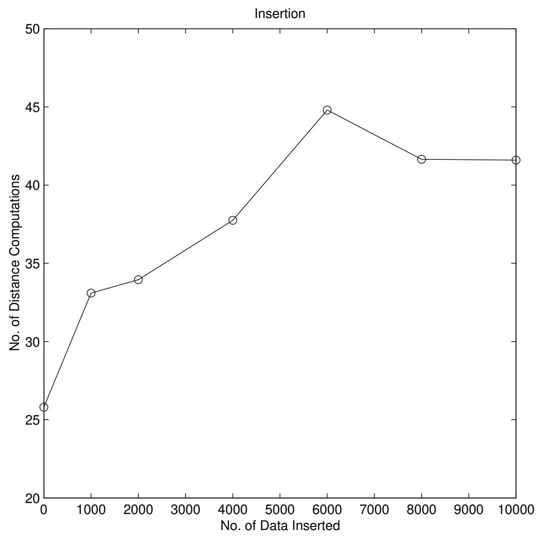


Figure 5.4: Searching performance for insertion.

5.4 Deletion

Apart from insertion, we can also delete an individual data point from a RPCL-b-tree. The algorithm for deleting a single data point from the tree is shown as follows.

Algorithm 5.3 Delete(T, q, M)

▷ *Input: A RPCL-b-tree T, a to-be deleted data object q, and the maximum node size M*

▷ *Output: An updated RPCL-b-tree*

```
N ← the root node of T
while N is not a leaf node do
    N ← the node with center closest to q among its child nodes if any
end while
if q is associated with N then do
    remove q form N
    update the center of N according to RPCL clustering rules
    update the information of Q's ancestors if necessary
    if the size of Q's parent node less than M then do
        merge all Q's parent node's child nodes
    end if
end if
```

The deletion algorithm makes the searching performance worse. When the number of deletions increases, the searching performance will decrease because node merging will change the original indexing structure (see Table 5.2 and Figure 5.5 as an example). Sometimes, the resultant indexing tree will give better searching results especially when the number of deletions is relatively small. It is because only a few points are removed from the indexing tree and it does not affect the natural clusters and the indexing structure any more.

5.5 Searching

In our work, we make use of the branch-and-bound algorithm proposed in [38] to compute the k nearest neighbors to a given query. The method is designed

# points pre-processed	# points deleted	avg. # of distance computations
5000	0	28.8
5100	100	31.75
5200	200	22.75
5300	300	28.10
5400	400	32.05
5600	600	32.25
5800	800	30.20
6000	1000	37.65
10000	5000	43.40

Table 5.2: The average searching performance of 20 nearest-neighbor queries on the RPCL-b-trees built in different ways of data deletions with the same final 5000 data objects.

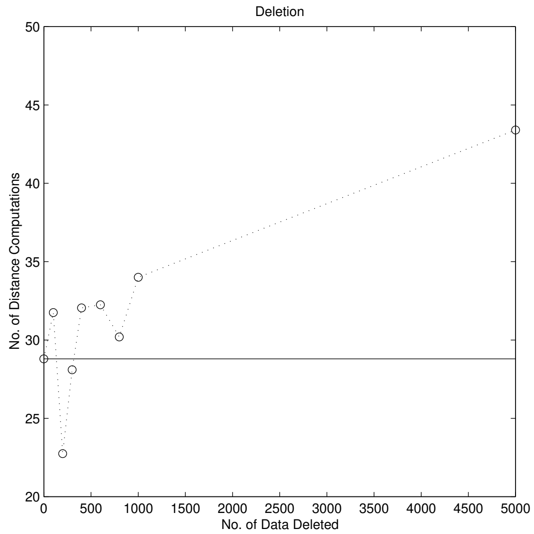


Figure 5.5: Searching performance for deletion.

specially for a tree structure which represents a hierarchical decomposition of a set of data points by using clustering techniques. Our RPCL-b-tree has exactly the same structure as the one described. Therefore, we apply the method for effective and efficient nearest-neighbor search.

The basic idea of the branch-and-bound method consists of two stages. First, the feature set is hierarchically decomposed into disjoint subsets (by RPCL in our method). The results of this decomposition are represented by a tree structure (RPCL-b-tree). Second, the resultant tree is searched by the branch-and-bound algorithm.

For the branch-and-bound algorithm, each node is tested to determine whether or not the nearest neighbor to a query by the following 4 rules. Let X be the query, B be the distance to the current nearest neighbor of X among the features considered up to the present, S_p be the set of features associated with node p, N_p be the number of samples associated with node p, M_p be the sample mean of S_p, $r_{pmax} = \max_{X_i \in S_p} D(X_i, M_p)$, and $r_{pmin} = \min_{X_i \in S_p} D(X_i, M_p)$, the 4 rules (see also Figure 5.6) are:

Rule 5.1 (General Inclusion Rule) *No $X_i \in S_p$ can be the nearest neighbor to X, if*

$$D(X, M_p) > B + r_{pmax} \ .$$

Proof: The proof of Rule 5.1 follows. For $X_i \in S_p$, by triangle inequality,

$$D(X, X_i) + D(X_i, M_p) \geq D(X, M_p) \ .$$

By definition, $D(X_i, M_p) \leq r_{pmax}$, we have

$$D(X, X_i) \geq D(X, M_p) - r_{pmax} \ .$$

Therefore, no X_i can be nearest neighbor to X, if

$$D(X, X_i) \geq D(X, M_p) - r_{pmax} > B \ ,$$

$$D(X, M_p) > B + r_{pmax} \; .$$

Rule 5.1 follows immediately.

Rule 5.2 (Specific Inclusion Rule) *X_i cannot be the nearest neighbor to X, if*

$$D(X, M_p) > B + D(X_i, M_p) \; ,$$

where $X_i \in S_p$.

Proof: The proof of Rule 5.2 is similar to the one for Rule 5.1.

Rule 5.3 (General Exclusion Rule) *No $X_i \in S_p$ can be the nearest neighbor to X, if*

$$B + D(X, M_p) < r_{pmin} \; .$$

Proof: The proof of Rule 5.3 follows. For $X_i \in S_p$, by triangle inequality,

$$D(X_i, X) + D(X, M_p) \geq D(X_i, M_p) \; .$$

By definition, $D(X_i, M_p) \geq r_{pmin}$, we have

$$D(X_i, X) \geq r_{pmin} - D(X, M_p) \; .$$

Therefore, no X_i can be nearest neighbor to X, if

$$D(X_i, X) \geq r_{pmin} - D(X, M_p) > B \; ,$$

$$B + D(X, M_p) < r_{pmin} \; .$$

Rule 5.3 follows immediately.

Rule 5.4 (Specific Exclusion Rule) *X_i cannot be the nearest neighbor to X, if*

$$B + D(X, M_p) < D(X_i, M_p) \; ,$$

where $X_i \in S_p$.

Proof: The proof of Rule 5.4 is similar to the one for Rule 5.3.

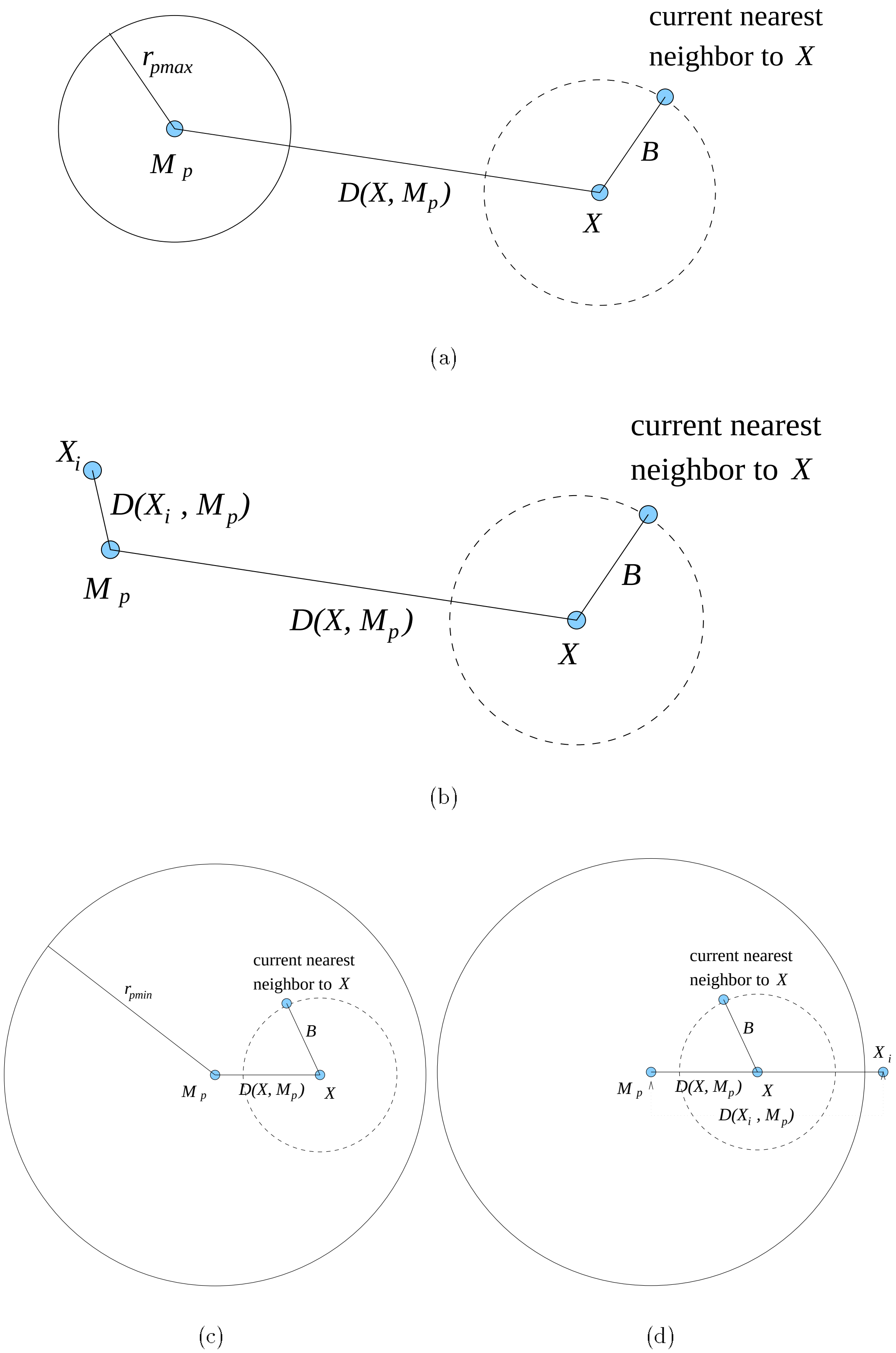


Figure 5.6: (a) General inclusion rule. (b) Specific inclusion rule. (c) General exclusion rule. (d) Specific exclusion rule.

With the above 4 rules, the branch-and-bound algorithm to find the nearest neighbor to a given query X is given below:

Algorithm 5.4 Search(T, X)

$\triangleright$ *Input: a RPCL-b-tree T and a query X*
$\triangleright$ *Output: the nearest neighbor to X*
$\triangleright$ *Initialization*
1 $B \leftarrow \infty$
2 $L \leftarrow 1$ $\triangleright$ *L is the current level*
3 $N \leftarrow$ root node $\triangleright$ *N is the current node*
$\triangleright$ *Expansion of current node*
4 $ActiveList \leftarrow$ all nodes that are immediately direct successors of N at level L
5 compute and store the $D(X, M_p)$'s for the nodes in $ActiveList$
$\triangleright$ *Test for Rules 5.1 and 5.3*
6 **for** each node p in $ActiveList$ at level L **do**
7   **if** $B + r_{pmax} < D(X, M_p)$ or $B + D(X, M_p) < r_{pmin}$ **then do**
8   remove p from the $ActiveList$ at level L
9   **end if**
10 **end for**
$\triangleright$ *Backtracking*
11 **if** no nodes left in $ActiveList$ at level L **then do**
12     $L \leftarrow L - 1$
    $\triangleright$ *backtrack to the previous level*
13   **if** $L = 0$ **then do**
14   terminate the algorithm
15   **else**
16   go to Line 6
17   **end if**
18 **end if**
$\triangleright$ *Choose the nearest node for expansion*
19 $p \leftarrow$ the node yielding the smallest $D(X, M_p)$ among the nodes in $ActiveList$ at level L
20 $N \leftarrow p$
21 remove p from the $ActiveList$ at level L
22 **if** L is not final level **then do**
23     $L = L + 1$
24   go to Line 4
25 **end if**
$\triangleright$ *Test for Rules 5.2 and 5.4*
26 **for** each X_i in p **do**
27   **if** $B + D(X_i, M_p) < D(X, M_p)$ or $B + D(X, M_p) < D(X_i, M_p)$ **then do**
28   do not compute $D(X, X_i)$
        $\triangleright$ *X_i cannot be the nearest neighbor to X*
29   **else**
30   compute $D(X, X_i)$

```
31          if D(X, X_i) < B then do
32              N = node i                    ▷ current nearest neighbor to X
33              B = D(X, X_i)
34          end if
35      end if
36  end for
37  go to Line 6
```

The above algorithm is easy to be extended to search k nearest neighbors for a query X. It can be done by keeping a sorted list of k current nearest neighbors instead of the current nearest neighbor and changing B to be the distance to the k-th nearest neighbor of X among the features considered up to the present.

5.6 Experiments

In this section, we present several experiments and their results for the performance of the RPCL-b-tree method with the branch-and-bound algorithm for retrieval. We use different kinds of data together with different parameters in the experiments in order to measure the efficiency of the method for 100 % nearest-neighbor search.

5.6.1 Experimental Setup

We conducted 6 different experiments to measure the efficiency of the RPCL-b-tree method for 100% nearest-neighbor search. All the experiments were conducted on an Ultra Sparc 1 machine and the RPCL-b-tree was implemented using C++.

Unlike the experiments presented in Chapter 4, we use a new measurement for efficiency here instead of Recall and Precision. It is because Recall and Precision are used for accuracy not efficiency and we do not need to test the accuracy for 100% nearest-neighbor search. For efficiency, it is hard to tell how efficient a method is, thus we define an efficiency measurement for the experiments. In a nearest-neighbor retrieval, the most time-consuming part is to find the distances between a query and the feature vectors. Therefore, the efficiency of an indexing method is almost proportional to these distance computations. The efficiency of our method is defined based on the efficiency of the linear search because it has the worst efficiency in searching than other methods. The efficiency measurement is defined as:

Definition 5.3 (Efficiency Measurement)

$$efficiency = 1 - \frac{\#\text{ of distance computations for the checked method}}{\#\text{ of distance computations in linear search}}. \quad (5.1)$$

We know that the efficiency of linear search is 0 because it needs to compute the distance between every feature vector and the query. Moreover, the total number of distance computations is equal to the size of the data set. As a result, Equation 5.1 becomes:

$$efficiency = 1 - \frac{\#\text{ of distance computations for the checked method}}{\text{size of the data set}}. \quad (5.2)$$

We use an example here to illustrate what is the practical meaning for this efficiency. For example, if the searching efficiency of a method is 0.8, the method needs only approximately 20% of the searching time needed by the linear search for retrieval.

We use three different kinds of data sets in the experiments:

1. **Clustered Data:** We test our method with synthetic data sets in Gaussian distribution. Section 4.2.1 shows the generating formula for this kind of data. In the following experiments, we simply used a constant 0.05 for σ and let n = 10, 100, and 1000 for the generating formula of the clustered data with dimensions varying from 2 to 16.

2. **Uniform Data:** We use also synthetic data sets in uniform distribution (see Section 4.2.1 for details). In the following experiments, the uniform data sets have dimensions varying from 2 to 16.

3. **Real Data:** We use a real data set for testing our method in a real world situation. The real data set is obtained in the same way as the one described in Section 4.2.1. The only difference is that the size of the set is 10000 here because we need not test for k-means in the following experiments so that a larger data set is more appropriate. Unlike synthetic data sets, this real data set is fixed with 10000 8-D feature vectors. Therefore, we use the set in Experiments 5, 7, 9, and 10 only.

In Experiments 5-10, we try to test our method's efficiency with different parameters as below:

- **Experiment 5:** Test for different node sizes.
- **Experiment 6:** Test for different sizes of the data sets.
- **Experiment 7:** Test for different data distributions.
- **Experiment 8:** Test for different numbers of dimensions.
- **Experiment 9:** Test for different numbers of database objects retrieved in nearest-neighbor search.
- **Experiment 10:** Test with VP-tree for comparing their efficiency.

From these experiments, we want to find out how these parameters affect the efficiency of our method. Moreover, we will work out a relationship formula for describing the relationship between the efficiency and these parameters in Section 5.8.

We first build a RPCL-b-tree in a batch mode for each of the testing data sets and then perform nearest-neighbor searches to calculate the efficiency of the RPCL-b-tree with the Equation 5.2. Finally, we give a brief discussion on the results and try to come out some conclusions. All the results and the conclusions will be presented in the following sections.

Node Size	100, 200, 500, 1000, and 2000.
Size of Data Set	10000.
Data Type	Clustered data with 100 Gaussian mixtures, uniform data, and real data.
Dimensionality	8.
Number of Database Objects Retrieved	1, 5, 20, 50, and 100.

Table 5.3: Detail of the parameters in Experiment 5.

5.6.2 Experiment 5: Test for Different Node Sizes

In Experiment 5, we try to test the efficiency of the RPCL-b-tree with different node sizes. In a RPCL-b-tree, each node contains no more than a certain number of data or each node has a maximum size. We want to find out how the node size affects the efficiency of the RPCL-b-tree and then figure out the most suitable node size for the tree from the experimental results.

We test the efficiency of the method for 100% nearest-neighbor retrieval. That means the searching result will be exactly the same as the linear search one. We use different numbers of node sizes in this experiment together with several other parameters. Table 5.3 shows the detail of the parameters. We test with node sizes actually from 1% to 20% of the size of data set. It is meaningless to test with node sizes more than 20% as the indexing tree may have only a few leaf nodes and each of them may become very large in size. Moreover, we retrieve no more than 100 database objects in the experiment because in practice the result of a query is not a too large set so that we can pick out the desired database objects manually from the set. Finally, for each set of parameters, 10 different nearest-neighbor searches are performed and the average results are used for analysis.

We present three main kinds of figures from the experiments: *the Indexing Structure Construction Time*, *the Searching Time* and *the Searching Efficiency*. The time used in building the indexing structure is shown in Table 5.4 and Figure

Node Size	**100**	**200**	**500**	**1000**	**2000**
Clustered Data	8.40	7.09	4.44	3.92	3.18
Uniform Data	7.64	5.54	3.84	3.03	3.21
Real Data	8.22	6.65	5.37	4.05	3.00

Table 5.4: Time used (in seconds) for building the indexing structures in Experiment 5.

Node Size		**100**	**200**	**500**	**1000**	**2000**
Clustered Data	$k = 1$	0.007	0.013	0.042	0.063	0.076
	$k = 20$	0.010	0.019	0.052	0.082	0.102
	$k = 100$	0.026	0.050	0.077	0.118	0.123
Uniform Data	$k = 1$	0.017	0.023	0.045	0.060	0.092
	$k = 20$	0.049	0.059	0.081	0.092	0.120
	$k = 100$	0.084	0.101	0.119	0.137	0.139
Real Data	$k = 1$	0.012	0.017	0.018	0.023	0.035
	$k = 20$	0.022	0.020	0.032	0.040	0.049
	$k = 100$	0.037	0.039	0.058	0.058	0.074

Table 5.5: The average time used (in seconds) for searching the k nearest-neighbors to 10 different queries in Experiment 5.

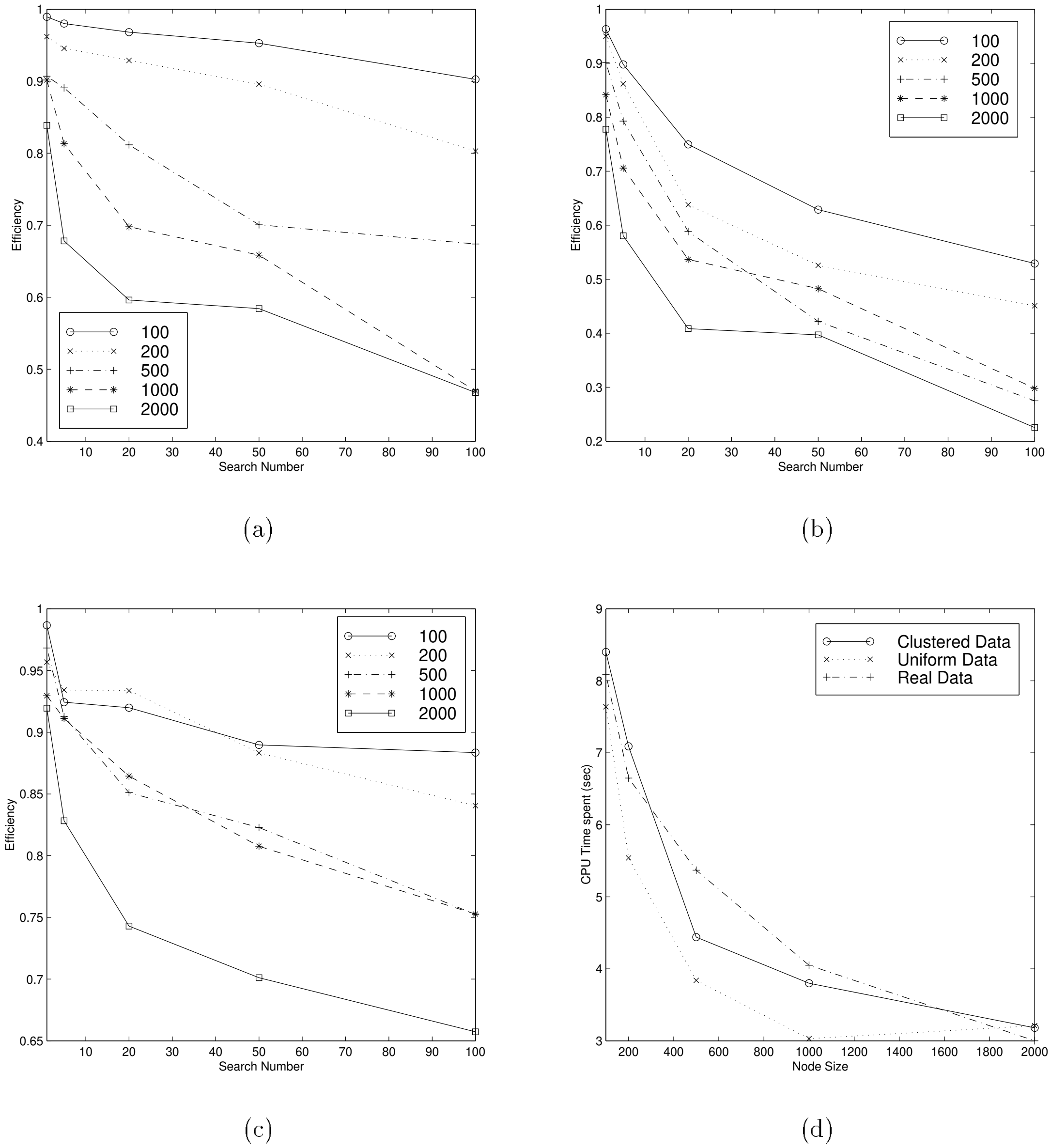


Figure 5.7: Results of Experiment 5. (a), (b), and (c) are the average efficiency for searching different numbers of nearest neighbors to 10 different queries for clustered data, uniform data, and real data respectively. (d) is the time used (in seconds) for building the indexing structure.

5.7(d). Table 5.5 shows the searching time. The efficiency of nearest-neighbor search with the RPCL-b-tree is presented in Figures 5.7(a), (b), and (c). Here are the observations from the figures and tables:

1. The smaller the node size, the better the efficiency is.
2. The larger the number of database objects retrieved, the worse the efficiency is.
3. The smaller the node size, the more time is needed to construct the indexing structure.
4. The larger the node size, the more the searching time is needed.

From the experimental results, we find that the smaller the node size, the better the efficiency is. For example, when node size is 1% (size = 100) or 2% (size = 200) of the size of the data set, the efficiency is better than those for other node sizes. However, in term of the indexing structure construction time, the smaller the node size, the more time is needed. Since the construction of the indexing structure is a pre-processing part and we only need to do it once, a little bit more construction time is acceptable. Therefore, we would rather have a better searching efficiency instead. We conclude that 1% to 2% of the size of the data set is a suitable node size for the RPCL-b-tree.

5.6.3 Experiment 6: Test for Different Sizes of Data Sets

In Experiment 6, we test the efficiency of the RPCL-b-tree with different sizes of the data sets. We want to find out if our indexing method is suitable for a large set of data.

We test the efficiency of the method for 100% nearest-neighbor retrieval. We use different sizes of the data sets in this experiment together with several other parameters. Table 5.6 shows the detail of the parameters. We fix the node size to

Node Size	200.
Size of Data Set	1000, 2000, 5000, 10000, 20000, and 50000.
Data Type	Clustered data with 100 Gaussian mixtures and uniform data.
Dimensionality	8.
Number of Database Objects Retrieved	1, 5, 20, 50, and 100.

Table 5.6: Detail of the parameters in Experiment 6.

Data Set Size	**1000**	**2000**	**5000**	**10000**	**20000**	**50000**
Clustered Data	0.42	1.09	3.10	7.09	14.48	47.88
Uniform Data	0.37	1.03	3.34	5.54	11.11	30.47

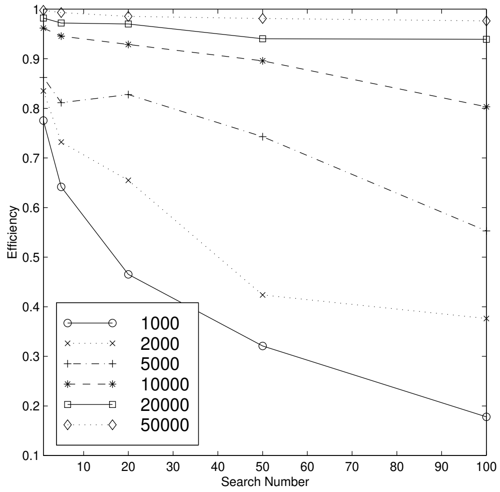
Table 5.7: Time used (in seconds) for building the indexing structures in Experiment 6.

200 as we find in the last experiment that 2% of the size of the data set (10000) is a suitable value for node size. For each set of parameters, 10 different nearest-neighbor searches are performed and the average results are used for analysis.

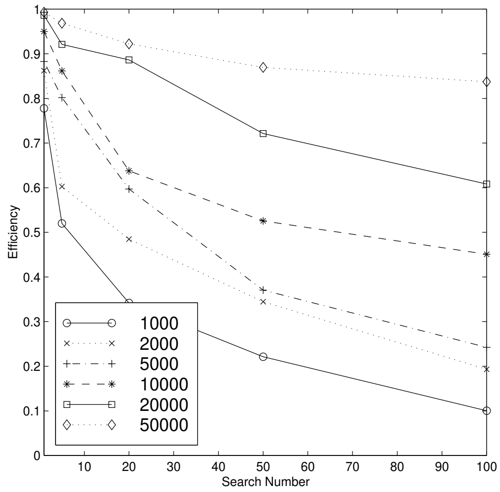
We use some tables and figures to present the experimental results. The time used in building the indexing structure is shown in Table 5.7 and Figure 5.8(c). Moreover, Table 5.8 shows the searching time. The efficiency of nearest-neighbor search with the RPCL-b-tree is presented in Figures 5.8(a) and (b). Here are the

Data Set Size		**1000**	**2000**	**5000**	**10000**	**20000**	**50000**
Clustered Data	k=1	0.009	0.028	0.017	0.013	0.012	0.008
	k=20	0.010	0.037	0.021	0.019	0.017	0.010
	k=100	0.023	0.058	0.044	0.050	0.039	0.036
Uniform Data	k=1	0.011	0.015	0.018	0.023	0.032	0.033
	k=20	0.014	0.028	0.042	0.059	0.057	0.079
	k=100	0.023	0.038	0.066	0.101	0.138	0.111

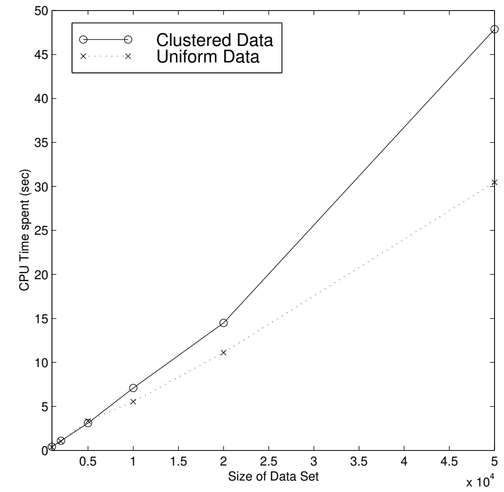
Table 5.8: The average time used (in seconds) for searching the k nearest-neighbors to 10 different queries in Experiment 6.



(a)



(b)



(c)

Figure 5.8: Results of Experiment 6. (a) and (b) are the average efficiency for searching different numbers of nearest neighbors to 10 different queries for clustered data and uniform data respectively. (c) is the time used (in seconds) for building the indexing structure.

observations from the figures and tables.

1. The larger the data set, the better the efficiency is.

2. The larger the number of database objects retrieved, the worse the efficiency is.

3. The larger the data set, the more time is needed to construct the indexing structure.

From the experimental results, the main finding is that the efficiency increases when the size of the data set increases. It is not hard to explain this observation with node size. Given a fixed node size, the ratio of the node size to the size of the data set decreases as the data set size increases. From Experiment 5, we know that the smaller the ratio, the better the efficiency of the method is. Again, it may need a little bit more time to construct the indexing structure for a large data set, but we do not mind to pay more pre-processing time for efficient retrieval.

5.6.4 Experiment 7: Test for Different Data Distributions

The objective of Experiment 7 is to test the searching performance of the RPCL-b-tree with different kinds of data distributions. We concentrate mainly on the Gaussian distribution with different numbers of Gaussian mixtures. We try to find out how the number of Gaussian mixtures affects the searching performance of our indexing method.

We test the efficiency of the method for 100% nearest-neighbor retrieval. We use clustered data with different numbers of Gaussian mixtures together with an uniform data set in which each feature vector itself can be treated as a Gaussian mixture and the real data set for reference. Table 5.9 shows the detail of the parameters. For each set of parameters, 10 different nearest-neighbor searches are performed and the average results are used for analysis.

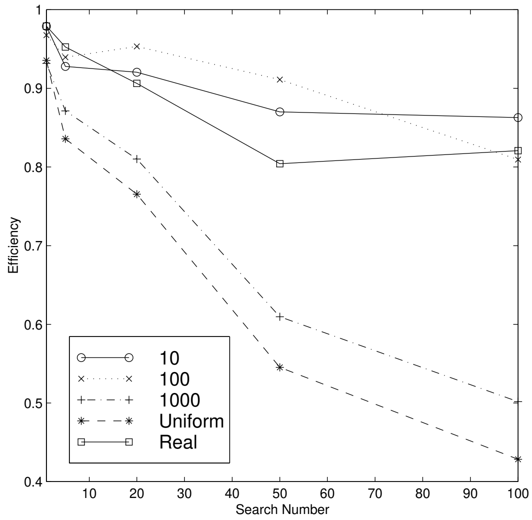


Figure 5.9: The average efficiency for searching different numbers of nearest-neighbors to 10 different queries for the clustered and uniform data in Experiment 7.

Node Size	200.
Size of Data Set	10000.
Data Type	Clustered data with 10, 100, and 1000 Gaussian Mixtures together with uniform data and real data.
Dimensionality	8.
Number of Database Objects Retrieved	1, 5, 20, 50, and 100.

Table 5.9: Detail of the parameters in Experiment 7.

No. of Gaussian Mixtures	**10**	**100**	**1000**	**Uniform**	**Real**
Construction Time	6.36	7.93	7.09	5.60	6.78

Table 5.10: Time used (in seconds) for building the indexing structures in Experiment 7.

No. of Gaussian Mixtures		**10**	**100**	**1000**	**Uniform**	**Real**
Searching Time	k=1	0.006	0.010	0.029	0.032	0.022
	k=20	0.012	0.016	0.048	0.061	0.025
	k=100	0.032	0.041	0.098	0.100	0.039

Table 5.11: The average time used (in seconds) for searching the k nearest-neighbors to 10 different queries in Experiment 7.

We use some tables and figures to present the experimental results. The time used in building the indexing structure and in the searching are shown in Table 5.10 and Table 5.11 respectively whereas the efficiency of nearest-neighbor search with the RPCL-b-tree is presented in Figure 5.9. Here are some of the main observations:

1. More Gaussian mixtures seem to give worse searching performance.
2. The larger the number of database objects retrieved, the worse the efficiency is.
3. The number of Gaussian mixtures is independent to the indexing structure construction time.
4. The more the Gaussian mixtures, the more time is needed in the searching.

From the experimental results, we find that the more the Gaussian mixtures, the worse the efficiency. It is because more small clusters are found when building the indexing structure. As a result, the indexing structure may have more nodes which will worsen the searching performance because more decisions have to be made for determining whether the node is going to be examined. We can treat the uniform data set as a 10000-sized data set with 10000 Gaussian mixtures or each individual feature itself is a one-point Gaussian mixture. Hence, we find that the efficiency for uniform data is not very good relatively. As expected, the efficiency of real data is in between the one for clustered data with 10 Gaussian mixtures and the one for uniform data.

5.6.5 Experiment 8: Test for Different Numbers of Dimensions

In this experiment, we try to test the efficiency of the RPCL-b-tree for the data in different numbers of dimensions. We know that many existing methods are not

Node Size	200.
Size of Data Set	10000.
Data Type	Clustered data with 100 Gaussian mixtures and uniform data.
Dimensionality	2, 4, 8, and 16.
Number of Database Objects Retrieved	1, 5, 20, 50, and 100.

Table 5.12: Detail of the parameters in Experiment 8.

Dimensionality	**2**	**4**	**8**	**16**
Clustered Data	1.95	4.02	8.79	17.95
Uniform Data	1.84	3.07	5.78	21.80

Table 5.13: Time used (in seconds) for building the indexing structures in Experiment 8.

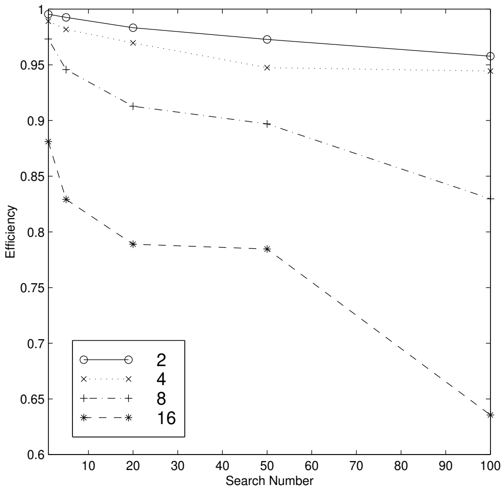
quite applicable for high dimensional data. Therefore, we want to find out if our method is suitable for high dimensional data.

We test the efficiency of the method for 100% nearest-neighbor retrieval. We use the data with different numbers of dimensions together with several other parameters. Table 5.12 shows the detail of the parameters. For each set of parameters, 10 different nearest-neighbor searches are performed and the average results are used for analysis.

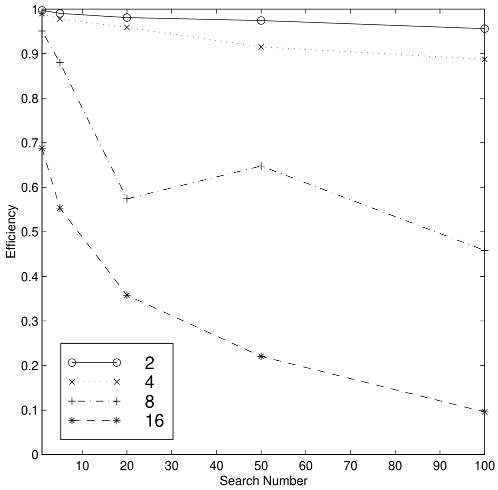
We use some tables and figures to present the experimental results. The time

Dimensionality		**2**	**4**	**8**	**16**
Clustered Data	k=1	0.002	0.002	0.012	0.059
	k=20	0.002	0.008	0.018	0.082
	k=100	0.006	0.023	0.043	0.133
Uniform Data	k=1	0.001	0.006	0.031	0.218
	k=20	0.001	0.005	0.072	0.225
	k=100	0.008	0.015	0.101	0.254

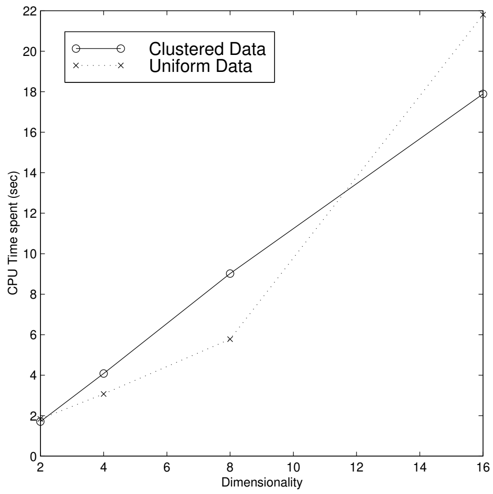
Table 5.14: The average time used (in seconds) for searching the k nearest-neighbors to 10 different queries in Experiment 8.



(a)



(b)



(c)

Figure 5.10: Results of Experiment 8. (a) and (b) are the average efficiency for searching different numbers of nearest neighbors to 10 different queries for clustered data and uniform data respectively. (c) is the time used (in seconds) for building the indexing structure.

used in building the indexing structure is shown in Table 5.13 and Figure 5.10(c). Moreover, Table 5.14 shows the searching time. The efficiency of nearest-neighbor search with the RPCL-b-tree is presented in Figures 5.10(a) and (b). Observations from the figures and tables include:

1. The higher the dimensionality, the worse the searching performance.
2. The larger the number of database objects retrieved, the worse the efficiency is.
3. The higher the dimensionality, the more time is needed for indexing.
4. The higher the dimensionality, the more time is needed in the searching.

From the experimental results, we find that our method works fine for low dimensional data. The efficiency is up to 0.9 for both clustered and uniform data. On the other hand, the searching performance of our method is acceptable for 16-dimensional clustered data. The efficiency is approximately 0.6. However it is only 0.1 for 16-dimensional uniform data which is not a good performance. For a real data set, it may be assumed to have an underlying distribution and we can usually approximate it by using Gaussian mixtures. Therefore, the efficiency is still acceptable for real data. We can check back Figure 5.7(c) for the searching performance with 10000 8-D real data and the efficiency is up to at least 0.85 when the node size is 200.

5.6.6 Experiment 9: Test for Different Numbers of Database Objects Retrieved

In this experiment, we test the RPCL-b-tree method for retrieving different numbers of database objects. We try to find out how the number of retrieved database objects affects the efficiency of the method.

Node Size	200.
Size of Data Set	10000.
Data Type	Clustered data with 100 Gaussian mixtures, uniform data, and real data.
Dimensionality	8.
Number of Database Objects Retrieved	1, 2, 5, 10, 20, 50, and 100.
Accuracy	10% to 100%.

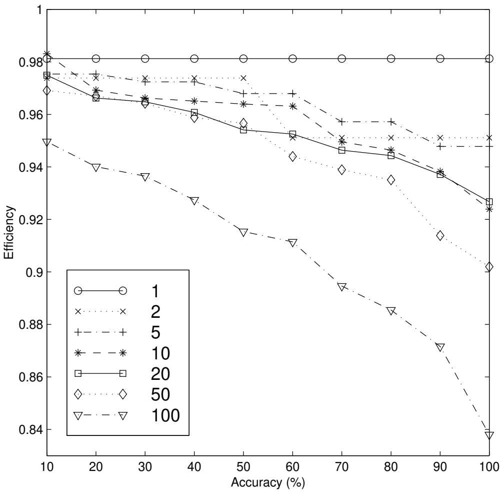
Table 5.15: Detail of the parameters in Experiment 9.

Data Distribution	**Clustered Data**	**Uniform Data**	**Real Data**
Construction Time	7.05	5.51	6.78

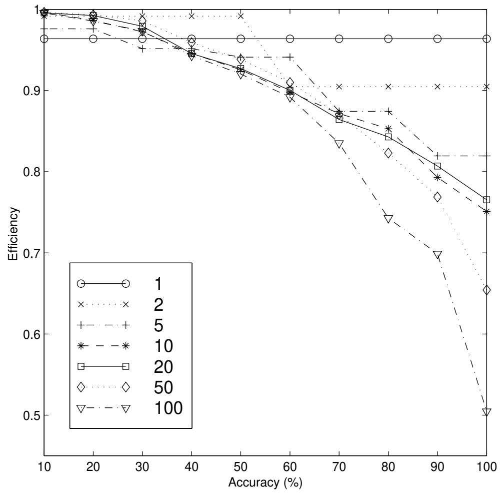
Table 5.16: Time used (in seconds) for building the indexing structures in Experiment 9.

We test the efficiency of the method for nearest-neighbor search with different numbers of database objects retrieved. In the Experiments 5-8, the number of database objects retrieved is used on the x-axis in the figures. For this experiment, we are exactly testing this number. Therefore, we change the x-axis' attribute to accuracy so that we can check also the efficiency for approximate nearest-neighbor search. The accuracy ranges from 10% to 100% and it simply indicates how accurate the nearest-neighbor retrieval result is. For example, given a top-10 nearest-neighbor retrieval result, accuracy equal to 80% means 8 out of the 10 database objects are correctly retrieved. Table 5.15 shows the detail of the parameters. For each set of parameters, 10 different nearest-neighbor searches are performed and the average results are used for analysis.

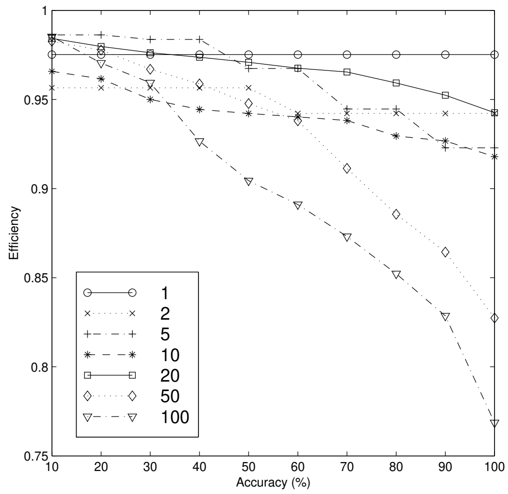
We use some tables and figures to present the experimental results. The time used in building the indexing structure and the searching time are shown in Table 5.16 and 5.17 respectively. Moreover, the efficiency of nearest-neighbor search with the RPCL-b-tree is presented in Figure 5.11. Here are some of the observations:



(a)



(b)



(c)

Figure 5.11: Results of Experiment 9. (a), (b), and (c) are the average efficiency for searching different numbers of nearest neighbors to 10 different queries for clustered data, uniform data, and real data respectively.

No. of objects retrieved	**1**	**2**	**5**	**10**	**20**	**50**	**100**
Clustered Data	0.008	0.014	0.016	0.020	0.017	0.022	0.046
Uniform Data	0.027	0.039	0.043	0.048	0.057	0.069	0.091
Real Data	0.017	0.019	0.014	0.021	0.019	0.039	0.050

Table 5.17: The average time used (in seconds) for searching different numbers of nearest-neighbors to 10 different queries in Experiment 9.

1. The more the database objects retrieved, the worse the searching efficiency.

2. The more accurate the nearest-neighbor retrieval, the worse the efficiency.

3. The more the database objects retrieved, the more searching time is needed.

From the experimental results, we find that the efficiency decreases when the number of database objects retrieved increases because more nodes and more candidate features are needed to be examined.

5.6.7 Experiment 10: Test with VP-tree

After testing our method with different parameters, we try to test the RPCL-b-tree method with VP-tree for comparing their searching efficiency. We consider VP-tree because it works similar to our method. First, it uses a hierarchical structure for indexing. Second, it splits the input data set into small sub-sets hierarchically in a top-down approach to form the indexing tree. In this experiment, we want to show that RPCL-b-tree is better than VP-tree for indexing and retrieval.

We test the efficiency of RPCL-b-tree and VP-tree for 100% nearest-neighbor retrieval. For fair comparison, backtracking is allowed for searching in VP-tree like RPCL-b-tree. We test the two methods with the following parameter sets which are similar to those in Experiments 5-8.

- **Parameter Set 1.** Different node sizes: 200, 1000, and 2000 for 10000 synthetic feature vectors with 100 Gaussian mixtures.

Node Size	**200**	**1000**	**2000**
Clustered Data (RPCL)	7.09	3.92	3.18
Clustered Data (VP)	144.21	94.76	70.68

Table 5.18: Time used (in seconds) for building the indexing structures for parameter set 1.

Node Size		**200**	**1000**	**2000**
Clustered Data	$k = 1$ (RPCL)	0.013	0.063	0.076
	$k = 1$ (VP)	0.114	0.112	0.113
	$k = 20$ (RPCL)	0.019	0.082	0.102
	$k = 20$ (VP)	0.115	0.113	0.114
	$k = 100$ (RPCL)	0.050	0.118	0.123
	$k = 100$ (VP)	0.127	0.128	0.133

Table 5.19: The average time used (in seconds) for searching the k nearest-neighbors to 10 different queries for parameter set 1.

- **Parameter Set 2.** Different sizes of data sets: 1000, 10000, and 50000 for synthetic feature vectors with 100 Gaussian mixtures.
- **Parameter Set 3.** Data sets of 10000 feature vectors with different data distributions: clustered data with 100 Gaussian mixtures, uniform data, and real data.
- **Parameter Set 4.** Data sets of 10000 feature vectors in 100 Gaussian mixtures with different numbers of dimensions: 2, 4, 8, and 16.

For each set of parameters, 10 different nearest-neighbor searches are performed and the average results are used for analysis. We present the experimental results are in Tables 5.18-5.25 and Figure 5.12.

From the experimental results, we find that RPCL-b-tree is more efficient than VP-tree for 100 % nearest-neighbor retrieval. For building the indexing structure, RPCL-b-tree is faster than VP-tree for each of the tested input data sets. It is because RPCL-b-tree uses a very fast clustering method, RPCL, to locate natural

Data Set Size	**1000**	**10000**	**50000**
Clustered Data (RPCL)	0.42	7.09	47.88
Clustered Data (VP)	7.43	145.19	966.17

Table 5.20: Time used (in seconds) for building the indexing structures for parameter set 2.

Data Set Size		**1000**	**10000**	**50000**
Clustered Data	k=1 (RPCL)	0.009	0.013	0.008
	k=1 (VP)	0.014	0.117	0.585
	k=20 (RPCL)	0.010	0.019	0.010
	k=20 (VP)	0.013	0.117	0.574
	k=100 (RPCL)	0.023	0.050	0.036
	k=100 (VP)	0.024	0.131	0.601

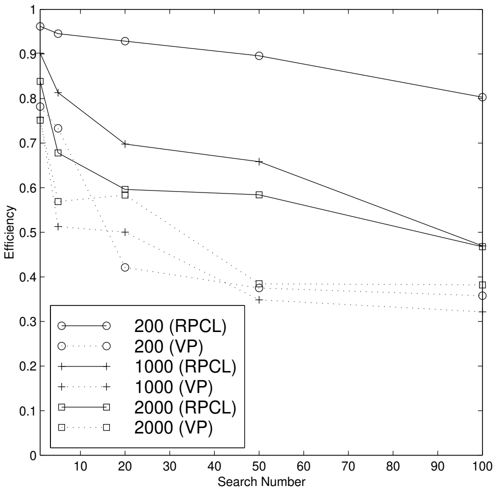
Table 5.21: The average time used (in seconds) for searching the k nearest-neighbors to 10 different queries for parameter set 2.

No. of Gaussian Mixtures	**100**	**Uniform**	**Real**
Construction Time (RPCL)	7.93	5.60	6.78
Construction Time (VP)	145.11	144.51	144.51

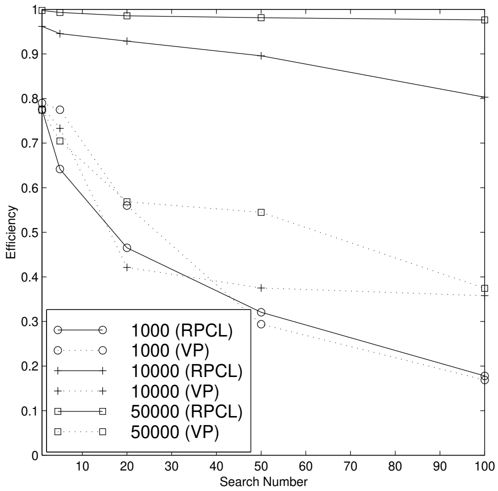
Table 5.22: Time used (in seconds) for building the indexing structures for parameter set 3.

No. of Gaussian Mixtures		**100**	**Uniform**	**Real**
Searching Time	k=1 (RPCL)	0.010	0.032	0.022
	k=1 (VP)	0.114	0.113	0.117
	k=20 (RPCL)	0.016	0.061	0.025
	k=20 (VP)	0.116	0.117	0.124
	k=100 (RPCL)	0.041	0.100	0.039
	k=100 (VP)	0.130	0.126	0.178

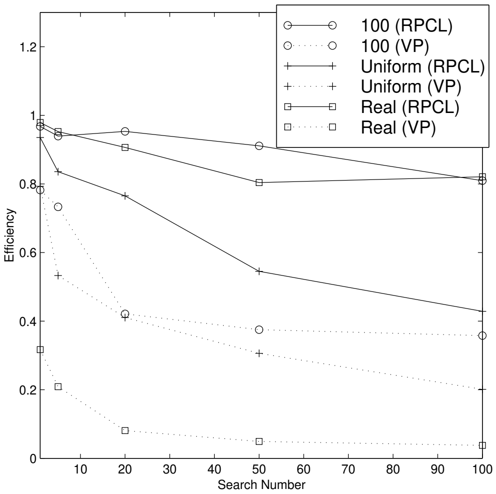
Table 5.23: The average time used (in seconds) for searching the k nearest-neighbors to 10 different queries for parameter set 3.



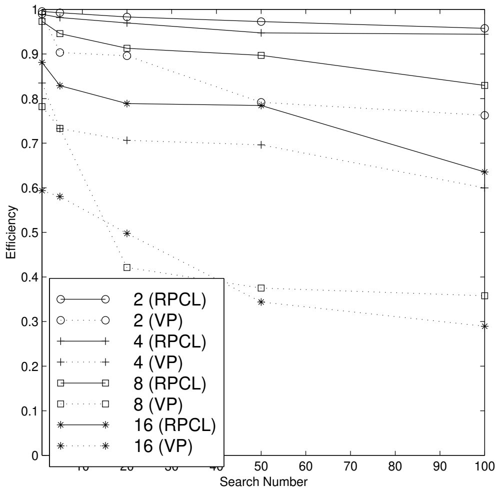
(a)



(b)



(c)



(d)

Figure 5.12: Results of Experiment 10. (a), (b), (c), and (d) are the average efficiency for searching different numbers of nearest neighbors to 10 different queries for parameter sets 1, 2, 3, and 4 respectively.

Dimensionality	**2**	**4**	**8**	**16**
Clustered Data (RPCL)	1.95	4.02	8.79	17.95
Clustered Data (VP)	49.73	81.08	145.19	276.95

Table 5.24: Time used (in seconds) for building the indexing structures for parameter set 4.

Dimensionality		**2**	**4**	**8**	**16**
Clustered Data	k=1 (RPCL)	0.002	0.002	0.012	0.059
	k=1 (VP)	0.036	0.062	0.117	0.218
	k=20 (RPCL)	0.002	0.008	0.018	0.082
	k=20 (VP)	0.039	0.063	0.117	0.224
	k=100 (RPCL)	0.006	0.023	0.043	0.133
	k=100 (VP)	0.042	0.073	0.131	0.252

Table 5.25: The average time used (in seconds) for searching the k nearest-neighbors to 10 different queries for parameter set 4.

clusters from the input data set for indexing. For retrieval, RPCL-b-tree is also more efficient than VP-tree. From Figure 5.12, we can see that the searching efficiency values of RPCL-b-tree are higher than the corresponding values of VP-tree. The main reason is that our method makes use of the branch-and-bound algorithm to speed up the searching. Moreover, we can also figure out a specific advantage from Figure 5.12(b). You can find that RPCL-b-tree is much more efficient than VP-tree for large data sets. In fact, this is due to the power of RPCL to locate clusters from large data set. In short, RPCL-b-tree outperforms VP-tree for indexing and retrieval.

5.7 Discussion

After introducing the RPCL-b-tree for indexing and retrieval, we would like to have a short discussion on its performance.

1. **Efficient Nearest-neighbor Retrieval:**

From most the experimental results, the efficiency is up to at least 0.8 for 100%-accuracy nearest-neighbor retrieval in general. It is because RPCL gives us natural clusters and we try to keep each of the natural clusters in a node. Therefore, most of the data in the node can be retrieved together as the result of a requested nearest-neighbor query and hence improve the effectiveness of retrieval. Besides, we make use of the branch-and-bound algorithm on the RPCL-b-tree for retrieval so that the efficiency of nearest-neighbor search can be increased. We also find that the searching time is proportional to efficiency so that most of the retrievals can be finished in no more than 1 second. Moreover, we have shown in Experiment 10 that our RPCL-b-tree outperforms a common used indexing method VP-tree for efficient retrieval.

2. **Solving the Boundary Problem for 100% Nearest-neighbor Result:**

   By using the branch-and-bound algorithm on the RPCL-b-tree, the boundary problem described in Section 2.4 can be solved. For example, there is a nearest-neighbor query lying on a boundary of two cluster partitions. With the backtracking mechanism and the 4 elimination rules, the branch-and-bound algorithm gives us 100% result of the query which may contain data objects on different clusters on both sides of the boundary efficiently.

3. **Short Indexing Structure Construction Time Needed:**

   The time used for building the indexing structure is short in the experiments. It is no more than 10 seconds for most of the cases. RPCL is a very fast clustering method so that it can locate the natural clusters efficiently. As a result, less time is needed for constructing the indexing structure.

4. **Good Performance for Real Data:**

   According to the experiments for the real data set, the efficiency is relatively high. The reason is that the distribution of the real data set can usually be approximated by using Gaussian mixtures. So, its efficiency is often similar to the efficiency of the clustered data or just a little bit worse.

5. **Insertion and Deletion:**

   Unlike the non-hierarchical approach presented in Chapter 4, the RPCL-b-tree has a hierarchical indexing structure which helps us to perform data insertion and deletion. From Sections 5.3 and 5.4, we know that a single data object can be inserted to or deleted from the indexing structure easily because there is a clear relationship among the internal nodes. Starting from the root node, we can find the target leaf node without any problem and then update the indexing structure for data insertion and deletion.

6. **Choice of Node Size:**

   We know from the experimental results that the smaller the node size, the better the efficiency. Therefore, we tend to choose a small node size although a little bit more indexing time have to spend. However, RPCL is a stochastic heuristic clustering method and it is computational efficient. It works fine for clustering a large set of data, but it may not be so efficient for a set with only a few data objects, say 50. Therefore, we have to choose a little bit larger node size, say 200, in most of the experiments for the RPCL-b-tree. This may worsen the effectiveness of the retrieval in a small multimedia database. However, in practice we often use multimedia database to manage a large number of data objects so that the node size is still relatively small in this case and it does not lessen the effectiveness of retrieval much.

7. **Branching Factor:**

   RPCL-b-tree is a binary tree so its branching factor (the maximum number of children for a non-leaf node) is 2. Since RPCL clustering can calculate any number of clusters from a set of data, it is easy to change the binary tree to a general tree with branching factor more than 2.

8. **Limitation:**

   The RPCL-b-tree works fine for low dimensional data and its efficiency is relatively high, but its performance is not very good for high dimensional data. In order to improve the searching performance, techniques that can reduce dimensionality of the features can be used before indexing. For example, we may use *KL Transform* [17, 24] or *Fast Fourier Transform* [10] to calculate the most important features out of a high dimensional feature vectors and then produce a low dimensional one for indexing.

5.8 A Relationship Formula

Based on all the experimental results, we want to find out the relationship between the efficiency and the tested parameters. In this section, we try to use a formula to describe their relationship. Recall the efficiency formula (Equation 5.2),

$$efficiency = 1 - \frac{\text{\# of distance computations for the checked method}}{\text{size of the data set}}.$$

We can rewrite it as:

$$efficiency = 1 - \%\text{ of direct distance computations needed.} \tag{5.3}$$

From Equation 5.3, we find that it is much straight forward to use the percentage of direct computations needed ($\%\ distcomp$) in our relationship formula. Once we obtain $\%\ distcomp$, we can get the corresponding efficiency by Equation 5.3 easily. Let M be the node size, S be the size of the data set, D be the dimensionality, and Q be the number of database objects retrieved. The relationship formula is defined as:

$$\%\ distcomp = (k_1 \cdot M + k_2 \cdot S + k_3 \cdot D + k_4 \cdot Q) \cdot 100\% \tag{5.4}$$

	k_1	k_2	k_3	k_4
Gaussian-100	$1.216 \cdot 10^{-4}$	$-6.6429 \cdot 10^{-6}$	$1.7437 \cdot 10^{-2}$	$2.0251 \cdot 10^{-3}$
Uniform	$9.5601 \cdot 10^{-5}$	$-9.4841 \cdot 10^{-6}$	$3.5376 \cdot 10^{-2}$	$3.9351 \cdot 10^{-3}$
Real	$6.6159 \cdot 10^{-5}$	$-3.3335 \cdot 10^{-4}$	$4.2030 \cdot 10^{-1}$	$1.3369 \cdot 10^{-3}$

Table 5.26: The values of the factors k_1, k_2, k_3, and k_4 for different data distributions.

where k_1, k_2, k_3, and k_4 are real-valued factors and different data distributions may have different factors. We simply use a linear model here for the first version of the formula. In fact, more research is needed to find a better model to describe the relationship of the parameters.

From the results of the Experiments 5-9, we make use of the linear regression technique to find the values of the factors: k_1, k_2, k_3, and k_4 of the three relationship formula for clustered data with 100 Gaussian mixtures, uniform data, and real data respectively (see also Table 5.26). They are:

$$\begin{aligned} \%\ distcomp_{Gau-100} = (&1.216 \cdot 10^{-4} \cdot M \ - \ 6.6429 \cdot 10^{-6} \cdot S \\ &+ \ 1.7437 \cdot 10^{-2} \cdot D \ + \ 2.0251 \cdot 10^{-3} \cdot Q) \cdot 100\% \end{aligned} \tag{5.5}$$

$$\begin{aligned} \%\ distcomp_{uni} = (&9.5601 \cdot 10^{-5} \cdot M \ - \ 9.4841 \cdot 10^{-6} \cdot S \\ &+ \ 3.5376 \cdot 10^{-2} \cdot D \ + \ 3.9351 \cdot 10^{-3} \cdot Q) \cdot 100\% \end{aligned} \tag{5.6}$$

$$\begin{aligned} \%\ distcomp_{real} = (&6.6159 \cdot 10^{-5} \cdot M \ - \ 3.3335 \cdot 10^{-4} \cdot S \\ &+ \ 4.2030 \cdot 10^{-1} \cdot D \ + \ 1.3369 \cdot 10^{-3} \cdot Q) \cdot 100\% \end{aligned} \tag{5.7}$$

From the formula, we can not only get an estimated efficiency value by giving the values of the parameters, but also easily find out the relationships between each of the tested parameters and the efficiency. Here are these relationships:

- **Node Size (M):** k_1 is positive. Therefore, the larger the node size, the greater the % *distcomp* and the worse the efficiency.

- **Size of Data Set** (S)**:** k_2 is negative. Therefore, the larger the data set, the less the % *distcomp* and the better the efficiency.

- **Dimensionality** (D)**:** k_3 is positive. Therefore, the higher the dimensionality, the greater the % *distcomp* and the worse the efficiency.

- **Number of Database Objects Retrieved** (Q)**:** k_4 is positive. Therefore, the more the database objects retrieved, the greater the % *distcomp* and the worst the efficiency.

Besides, form Table 5.26, we find that the magnitude of k_3 is the largest factor among the four. It means that k_3 is the dominate factor. A little bit change of the number of dimensions will affect the efficiency much.

Moreover, we can generalize the formula to other indexing methods for comparing their efficiency. For the indexing methods having corresponding relationship formula, we can find out which method gives us the best efficiency easily for a given set of parameters. For example, from the results of Experiment 10, we can work out the relationship formula of VP-tree for clustered data with 100 Gaussian mixtures as:

$$\begin{aligned}\%\ distcomp_{vp-Gau-100} = (&6.2544 \cdot 10^{-5} \cdot M \ + \ 1.9647 \cdot 10^{-7} \cdot S \\ &+ \ 3.1051 \cdot 10^{-2} \cdot D \ + \ 3.6359 \cdot 10^{-3} \cdot Q) \cdot 100\% \end{aligned} \tag{5.8}$$

Using Equations 5.5 and 5.8, we can compare the efficiency of RPCL-b-tree and VP-tree for the clustered data. Given the following parameters, $M = 500$, $S = 20000$, $D = 8$, and $Q = 20$, RPCL-b-tree's estimated efficiency is 0.8921 whereas VP's estimated efficiency is 0.6437. Therefore, we know that RPCL-b-tree is most likely more efficient than VP-tree for the given set of parameters.

5.9 Chapter Summary

In summary, our method has good searching performance in general. The branch-and-bound algorithm solves the boundary problem and makes nearest-neighbor search on the RPCL-b-tree more efficient and effective. We have shown that our method outperforms VP-tree for indexing and retrieval. Moreover, the hierarchical structure helps us to update the RPCL-b-tree simply. From the experimental results, we work out a relationship formula for efficiency estimation and efficiency comparison with other indexing methods.

Chapter 6

Conclusion

6.1 Future Works

1. **Indexing in Montage:**

   One of the key issues in Montage (see Section 1.1 for details) is the implementation of a good indexing structure for efficient and accurate image retrieval of a large amount of images. Since Montage uses image contents or features for retrieval, traditional indexing methods are not particularly suitable for the system. Therefore, we plan to implement our RPCL indexing method into the system for efficient and accurate content-based indexing and retrieval.

2. **A Hybrid Method for RPCL-b-tree:**

   We plan to use a hybrid method to build the RPCL-b-tree. In this hybrid method, we use RPCL to cluster large data sets only. It is because RPCL is not so efficient for small data set (size ≤ 100). On the other hand, we use another suitable clustering method for small data sets. As a result, we can use a smaller node size in the tree and it will increase the efficiency of retrieval.

3. **The Relationship Formula:**

   The relationship formula presented in Section 5.8 is in a linear model currently. More research is needed to find out a better model for the relationship formula.

4. **A Bottom Up Approach for RPCL Indexing:**

   Apart from building the indexing structure in a top-down approach, we can also use RPCL to find out the actual number of natural clusters form the input feature vector set and then build an indexing structure in a bottom up fashion. After all the natural clusters located by RPCL, we can merge the neighbor clusters to form a bigger cluster for its parent level. The process repeats until the root cluster containing the whole feature vector set is formed. We will then get the indexing structure for retrieval. This indexing method seems to be better than RPCL-b-tree because it actually keeps the natural clusters at the leaf nodes.

6.2 Conclusion

We have used an efficient clustering algorithm Rival Penalized Competitive Learning (RPCL) to locate natural clusters for content-based indexing and retrieval. Based on the located clusters, we have presented two approaches to build up indexing structures: *Non-hierarchical approach* and *Hierarchical approach*. For non-hierarchical approach, we have analyzed the performance of our method by using some experiments and have found that the overall performance of our method for nearest-neighbor retrieval is better then other tested methods. For hierarchical approach, we have proposed to build a hierarchical binary tree (RPCL-b-tree) for retrieval. We also make use of a branch-and-bound algorithm to solve the boundary problem. From the experimental results, it is concluded that: (1) RPCL is a very fast method to locate natural clusters for indexing, (2) the non-hierarchical

RPCL indexing have high recall and precision performance for producing good approximate retrieval result quickly, and (3) RPCL-b-tree is efficient to produce 100% nearest-neighbor search results and it outperforms VP-tree for indexing and retrieval. According to the experimental results for the RPCL-b-tree, we also work out a relationship formula for finding estimated searching efficiency and comparing the efficiency with other indexing methods.

Bibliography

[1] G. W. Adamson and D. Bawden. "Comparison of hierarchical cluster analysis techniques for automatic classification of chemical structures". *Journal of Chemical Information and Computer Sciences*, 21(4):204–209, 1981.

[2] S. C. Ahalt, A. K. Krishnamurthy, P. Chen, and D.E. Melton. 'Competitive learning algorithms for vector quantization". *Neural Networks*, 3(3):227–290, 1990.

[3] M. R. Anderberg. *"Cluster Analysis for Applications"*. Academic Press, New York, 1973.

[4] J. Ashley, R. Barber, M. Flickner, J. Hafner, D. Lee, W. Niblack, and D. Petkovic. "Automatic and semi-automatic methods for image annotation and retrieval in QBIC". In *Proceedings of Storage and Retrieval for Image and Video Databases III*, volume 2420, pages 24–35, February 1995.

[5] R. Bayer. "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms". *Acta Informatica*, 1(4):290–306, 1972.

[6] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. "The R*-tree: an efficient and robust access method for points and rectangles". *ACM SIGMOD Record*, 19(2):322–331, 1990.

[7] J. L. Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". *Communications of the ACM*, 18(9):509–517, 1975.

[8] S. Berchtold, D. A. Keim, and H. P. Kriegel. "The X-tree: An Index Structure for High-Dimensional Data". In *Proceedings on the 22th VLDB Conference*, pages 28–39, 1996.

[9] Tolga Bozkaya and Meral Ozsoyoglu. "Distance-Based Indexing for High-dimensional Metric Spaces". *SIGMOD Record*, 26(2):357–368, June 1997.

[10] E. O. Brigham. *"The Fast Fourier Transform"*. Prentice Hall, 1974.

[11] M. Bruynooghe. "Large data set clustering methods using the concept of space contraction". In *COMPSTAT 1978 Proceedings in Computational Statistics*, volume 3, pages 239–245, 1978.

[12] W. A. Burkhard and R. M. Keller. "Some approaches to best-match file searching". *Communications of the ACM*, 16(4):230–236, 1973.

[13] T. C. Chiueh. "Content-Based Image Indexing". In *Proceedings of the 20th VLDB Conference*, pages 582–593, September 1994.

[14] W. W. Chu, I. T. Ieong, R. K. Taira, and C. M. Breant. "A temporal evolutionary object-oriented data model and its query language for medial image management". In *Proceedings of 18th VLDB Conference*, pages 53–64, 1992.

[15] Wesley W. Chu, Alfonso F. Cardenas, and Ricky K. Taira. "KMeD: A knowledge-based Multimedia Medical Distributed Database System". *Information Systems*, 20(2):75–96, 1995.

[16] D. Comer. "The Ubiquitous B-tree". *ACM Computing Surveys*, 11(2):121–137, June 1979.

[17] R. Duda and P. Hart. *"Pattern Classification and Scene Analysis"*. Wiley, New York, 1973.

[18] Brian S. Everitt. *"Cluster Analysis"*. Halsted Press, New York, 1993.

[19] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. "Efficient and effective querying by image content". *Journal of Intelligient Information Systems*, 3(3/4):231–262, July 1994.

[20] C. D. Feustel and L. G. Shapiro. "The nearest neighbor problem in an abstract metric space". *Pattern Recognition Letters*, 1(2):125–128, December 1982.

[21] R. A. Finkel and J. L. Bentley. "Quad Trees: A Data Structure for Retrieval on Composite Keys". *Acta Informatica*, 4(1):1–9, 1974.

[22] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. "Query by image and video content: The QBIC system". *IEEE Computer*, 28(9):23–32, September 1995.

[23] K. Fukunaga and P. M. Narendra. "A branch and bound algorithm for computing K-nearest neighbors". *IEEE Transactions on Computers*, 24(7):750–753, July 1975.

[24] Keinosuke Fukunaga. *"Introduction to Statistical Pattern Recognition"*. Academic Press, 2nd edition, 1990.

[25] C. A. Goble, M. H. O'Dochery, P. J. Crowther, M. A. Ireton, C. N. Daskalakis, J. Oakley, S. Kay, and C. S. Xydeas. "The Manchester Multimedia Information System". *Multimedia Systems, Interaction and Applications, 1st Eurographics Workshop*, pages 269–282, March 1991.

[26] C. A. Goble, M. H. O'Dochery, P. J. Crowther, M. A. Ireton, J. Oakley, and C. S. Xydeas. "The Manchester Multimedia Information System". *Advances in Database Technology EDBT'92, Third International Conference on Extending Database Technology*, pages 39–55, March 1992.

[27] Y. H. Gong, C. H. Chuan, and X. Y. Guo. "Image Indexing and Retrieval Based on Color Histograms". *Multimedia Tools and Applications*, 2(2):133–156, March 1996.

[28] A. D. Gordon. "A review of hierarchical classification". *J. Roy. Statist. Soc.*, 150:119–137, 1987.

[29] A. D. Gordon. *"Hierarchical Classification in Clustering and Classification"*. World Scientific Press, 1992.

[30] W. I. Grosky, R. Jain, and R. Mehrotra. *"The Handbook of Multimedia Information Management"*. Prentice-Hall, 1997.

[31] V. N. Gudivada and V. V. Raghavan. "Content-Based Image Retrieval Systems". *Computer*, 28(9):18–22, September 1995.

[32] A. Gupta, T. Weymouth, and R.Jain. "Semantic queries with pictures: The VIMSYS model". In *Proc. 17th VLDB*, pages 69–79, 1991.

[33] A. Guttman. "R-trees: A Dynamic Index Structure for Spatial Searching". *ACM SIGMOD*, 14(2):47–57, June 1984.

[34] S. Haykin. *"Neural networks: a comprehensive foundation"*. Macmillan, New York, 1994.

[35] K. Hirata and T. Kato. "Query by visual example - content based image retrieval". In *Advances in Database Technology EDBT'92, Third International Conference on Extending Database Technology*, pages 56–71, March 1992.

[36] C. E. Jacobs, A. Finkelstein, and D. H. Salesin. "Fast Multiresolution Image Querying". In *Proceedings of SIGGAPH 95*, pages 277–286, August 1995.

[37] H. V. Jagadish. "A Retrieval Technique for Similar Shapes". In *Proc. of the ACM SIGMOD Internation Conference on the Management of Data*, pages 208–217, June 1991.

[38] B. Kamgar and L. N. Kanal. "An improved branch and bound algorithm for computing k-nearest neighbors". *Pattern Recognition Letters*, 3(1):7–12, January 1985.

[39] Norio Katayama and Shinichi Satoh. "The SR-tree: an index structure for high-dimensional nearest neighbor queries". *SIGMOD Record*, 26(2):369–380, June 1997.

[40] T. Kato. "Database architecture for content-based image retrieval". In *SPIE*, volume 1662, pages 112–123, 1992.

[41] I. King, A. Fu, L.W. Chan, and L. Xu. "Montage: An Image Database for the Hong Kong's Textile, Fashion, and Clothing Industry", 1995. http://www.cse.cuhk.edu.hk/~viplab.

[42] Irwin King and Tak Kan Lau. "A Feature-Based Image Retrieval Database for the Fashion, Textile, and Clothing Industry in Hong Kong". In *Proceedings of the 1996 International Symposium on Multimedia Information Processing (ISMIP'96)*, 1996.

[43] Irwin King and Tak Kan Lau. "Comparison of Several Partitioning Methods for Information Retrieval in Image Databases". In *Proceedings of the 1997 International Symposium on Multimedia Information Processing (ISMIP'97)*, pages 215–220, 1997.

[44] Irwin King and Tak Kan Lau. "Competitive Learning Clustering for Information Retrieval in Image Databases". In *Proceedings of the 1997 Internation Conference on Neural Information Processing (ICONIP'97)*, pages 906–909, 1997.

[45] Tak Kan Lau and Sui Tak Hung. "Montage: An Image Database for Effective Digital Image Management". In *Hong Kong International Computer Conference 1997 (HKICC'97)*, volume 1, pages 83–88, 1997.

[46] Tak Kan Lau and Irwin King. "Montage: An Image Database for the Fashion, Textile, and Clothing Industry in Hong Kong". In *Proceedings of the Third Asian Conference on Computer Vision (ACCV'98)*, volume 1, pages 410–417, 1998.

[47] Tak Kan Lau and Irwin King. "Performance Analysis of Clustering Algorithms for Information Retrieval in Image Databases". In *Proceedings to the International Joint Conference on Neural Networks (IJCNN'98)*, pages 932–937, 1998.

[48] W. Y. Ma and B. S. Manjunath. "Texture-based Pattern Retrieval from Image Databases". *Multimedia Tools and Applications*, 2(1):35–51, January 1996.

[49] J. B. MacQueen. "Some Methods for Classification and Analysis of Multivariate Observations". In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[50] B. S. Manjunath and W. Y. Ma. "Texture Features for Browsing and Retrieval of Image Data". *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(8):837–842, August 1996.

[51] R. Mehrotra and J. E. Gray. "Similar-shape Retrieval in Shape Data Management". *Computer*, 29(9):57–62, September 1995.

[52] B. M. Mehtre, M. S. Kankanhalli, and W. F. Lee. "Shape Measures for Content-based Image Retrieval - a Comparison". *Information Processing and Management*, 33(3):319–337, May 1997.

[53] S. A. Nene and S. K. Nayar. "A Simple Algorithm for Nearest Neighbor Search in High Dimensions". *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(9):989–1003, September 1997.

[54] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin. "The QBIC project: querying images by content using color, texture, and shape". In *Proceedings of the SPIE - The International Society for Optical Engineering*, volume 1908, pages 173–187, 1993.

[55] H. Nishizawa, T. Obi, M. Yamaguchi, and N. Ohyama. "Hierarchical clustering method for the analysis of large amount of data". In *Proceedings of the SPIE - The International Society for Optical Engineering*, volume 2824, pages 183–190, 1996.

[56] M. H. O'Dochery, C. N. Daskalakis, P. J. Crowther, C. A. Goble, M. A. Ireton, J. Oakley, and C. S. Xydeas. "The design and implementation of a multimedia information system with automatic content retrieval". *Information Services and Use*, 11:345–385, 1991.

[57] V.E. Ogle and M. Stonebraker. "Chabot: Retrieval from a Relational Database of Images". *Computer*, 28(9):40–48, September 1995.

[58] S. Openshaw. "Monothetic divisive algorithms for classifying large data sets". *COMPSTAT*, 4:419–425, 1981.

[59] A. Pentland, R. W. Picard, and S. Sclaroff. "Photobook: Content-Based Manipulation of Image Databases". *International Journal of Computer Vision*, 18(3):223–254, June 1996.

[60] N. Ramesh and I. K. Sethi. "Feature Identification as an Aid to Content-based Image Retrieval". In *SPIE*, volume 2420, pages 2–11, 1995.

[61] N. Roussopoulos, S. Kelley, and F. Vincent. "Nearest Neighbour Queries". *ACM SIGMOD*, 24(2):71–79, June 1995.

[62] D. E. Rumelhart and D. Zipser. "Feature discovery by competitive learning". *Cognitive Science*, 9:75–112, 1985.

[63] T. Sellis, N. Roussopoulos, and C. Faloutsos. "The R^+-tree: a dynamic index for multidimensional objects". In *Proceedings of the 13th VLDB Conference*, pages 507–518, 1987.

[64] J.R. Smith and S. F. Chang. "VisualSEEk: a fully automated content-based image query system". In *ACM multimedia - international conference - 1996*, pages 87–98, November 1996.

[65] F. Tomita and S. Tsuji. *"Computer analysis of visual textures"*. Kluwer Academic Publishers, Boston, 1990.

[66] L.H. Tung and I. King. "A two-stage framework for polygon retrieval using minimum circular error bound". In *Proceedings to the 9th International Conference on Image Analysis and Processing (ICIAP'97)*, pages 567–574, 1997.

[67] L.H. Tung, I. King, P.F. Fung, and W.S. Lee. "Two-Stage Polygon Representation for Efficient Shape Retrieval in Image Databases". In *Proceedings of the First International Workshop on Image Databases and Multi-Media Search*, pages 146–153, 1996.

[68] D. A. White and R. Jain. "Similarity Indexing with the SS-tree". In *Proceedings of the 12th International Conference on Data Engineering*, pages 516–523, February 1996.

[69] J. K. Wu, A. D. Narasimhalu, B. M. Mehtre, C. P. Lam, and Y. P. Gao. "CORE: A content-based retrieval engine for multimedia information systems". *ACM Multimedia Systems*, 3(1):25–41, February 1995.

[70] L. Xu, A. Krzyżak, and E. Oja. "Rival penalized competitive learning for clustering analysis, RBF net, and curve detection". *IEEE Transactions on Neural Networks*, 4(4):636–649, July 1993.

[71] P. N. Yianilos. "Data structures and algorithms for nearest neighbor search in general metric spaces". In *Proc. of the 4th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 311–321, 1993.

[72] T. Zhang, R. Ramakrishnan, and M. Livny. "BIRCH: A New Data Clustering Algorithm and Its Applications". *Data Mining and Knowledge Discovery*, 1(2):141–182, 1997.