A Signature File Algorithm for Large Image Databases

By

Weihua Lu

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree

Master of Science

Department of Computer Science
University of Manitoba,
Winnipeg, Manitoba,Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

A Signature File Algorithm for Large Image Databases

BY

Weihua Lu

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of

Manitoba in partial fulfillment of the requirement of the degree

MASTER OF SCIENCE

Weihua Lu © 2007

Abstract

Signature file, which is an indexing technique, has been extensively studied in text retrieval. It acts as a filtering mechanism which is able to screen out the most non-qualifying documents, thus, confining document searches to smaller relevant candidate sets. Many methods for organizing signature files have been proposed to improve searching speed since querying a large signature file sequentially is very time consuming. However, these methods have limitations when they are applied to image databases since the distinct characteristics of image databases have not been taken into account.

The goal of this research is to design an indexing algorithm for large image databases. This proposed algorithm, called the Image Signature Tree (IST), retrieves an image in a database based on image objects and spatial relations between the objects contained in an image, as well as image sizes and formats. Signature file technique is adopted in this algorithm.

Performance evaluation is conducted both analytically and experimentally. The analytical study of the performance in term of signature reduction ratio was conducted based on probability theory. The retrieval cost, signature reduction ratio, storage cost and update cost are studied by simulation.

Acknowledgements

I would like thank for my thesis supervisor Dr. Yang Jun Chen. Thank you for giving me the opportunity to work with you, leading me going through each phase of this research, encouraging me when I felt frustrated.

I would like to thank Dr. Peter Graham and Dr. Neil Arnason for helping me solve many issues happened during the graduate study at the UM these years.

Many thanks go to the members of my thesis examining committee, Dr. Carson Kai-Sang Leung, Dr. Attahiru S. Alfa for giving me valuable comments and suggestions for improving the quality of my thesis work.

My parents, how can I express my gratitude to you in words! "you raise my up. I can fly because you give me the wings." Without your support, I don't think I can finish this study. My brothers, you always set good examples for me to follow.

I would thank for Miriam and Irma who taught me Bible. My soul finally finds her family which she pursued all these years. In the Christian family, she finds that so many people share the same belief with her, therefore she doesn't feel lonely any more. She becomes more peaceful, more courageous, more confident. Next, I would like to thank my roommates and classmates, I feel so lucky that I have many friends in Winnipeg to share joys and tears together. I would also extend my gratitude to my colleagues in my workplace. Your smiles melt the snow of the Winnipeg. I would like to thank for all the eyes that care about me. How I wish could name you all. Thank you everyone!

I would like to thank the department of computer science of the University of Manitoba for the departmental fellowship. I also like to express my deep appreciation to Dr. Yang Jun Chen for the financial support from his Natural Sciences and Engineering Research Council (NSERC) research grant.

Contents

1 Introduction **1**

1.1 Motivation and Problem Definition 3

1.2 Preliminaries 4

1.2.1 Signature File Technique 4

1.2.2 Image Retrieval 7

1.3 Objective of this research 9

1.4 Organization 10

2 Literature Review **13**

2.1 Signature Algorithms 13

2.1.1 Sequential Signature File 13

2.1.2 Bit Slice File 13

2.1.3 Multi Level Signature File 14

2.1.4 Partitioned Signature File 15

2.1.5 Signature Tree 16

2.1.6 Balanced Signature Tree 17

2.2 Signature File in Image Research 19

2.3 Partitioned Signature File Research 21

3 Image Signature Tree Algorithm **23**

3.1 Image Model 23

3.2 Image Signature Tree Algorithm 24

3.2.1 Partitioning Algorithm for a *BST* 26

3.2.2 Query Processing 30

3.2.3 Insertion 31

3.2.4 Deletion 32

4 Analysis **33**

4.1 Control Parameters 33

4.2 Performance Metrics 35

4.2.1 Storage Cost 35

4.2.2 Signature Reduction 36

4.2.3 Retrieval Cost(RC) 37

4.2.4 Update Cost(UC) 38

5 Implementation **39**

5.1 Prototype 39

5.1.1 Introduction 39

5.1.2 System Overview 41

5.1.3 Test Data 42

5.1.4 Working Platform 42

5.1.5 Implementation of *IST* 43

5.2 Partitioned Signature Tree 47

6 Experimental Results **55**

6.1 Working Platform 55

6.2 Test Data 55

6.3 Storage Overheads 56

6.4 Signature Reduction 56

6.4.1 Signature Reduction with Weight 57

6.4.2 Signature Reduction with Data Size 58

6.5 Partition Access against Query Weight 59

6.6 Retrieval Cost 59

6.7 Update Cost 62

6.7.1 Insertion Cost 62

6.7.2 Deletion Cost 63

7 Discussion and Conclusions **65**

7.1 Summary of Contributions 65

7.2 Future Work 66

A An Example for Analyzing a Balanced Signature Tree **69**

List of Tables

1.1 Signature construction and comparison 5

4.1 Control parameters 34

4.2 Signature drops against weight 37

6.1 Comparison of storage overheads of BST against PST 56

6.2 Signature reduction ratio against weight 57

6.3 Signature reduction ratio against the size 58

6.4 Retrieval cost 60

6.5 Retrieve time of the PST 61

6.6 Insertion time of PST 62

6.7 Usage of partitions 63

6.8 Deletion time of PST 63

A.1 Signature Drops against Weight 71

List of Figures

1.1 Spatial relationship 8

2.1 Signature file and bit slice signature file 14
2.2 Multi level signature file 14
2.3 Partitioned signature file 15
2.4 Signature tree 17
2.5 Basic signature tree structure 18
2.6 Weight-based method 19
2.7 Balanced signature tree 20

3.1 Image model 23
3.2 Image signature tree 25
3.3 Spatial relation tree 26
3.4 Partitioning scheme 1 27
3.5 Partitioning scheme 2 28
3.6 Partition splitting 29

5.1 An image retrieval system 41
5.2 Query interface 42
5.3 The folder tree 43
5.4 The folder data file 44
5.5 Image signature in a partition 45

5.6 How to construct size signature 46
5.7 Schema diagram for the database 46
5.8 Report 47
5.9 Class diagram for *PST* 48
5.10 The flow chart for *PST* 49
5.11 Store the tree to an array 51
5.12 Search procedure 52
5.13 Load an array from disk 53
5.14 Generate the key tree from an array 53

6.1 Signature reduction of the *PST* against weight 57
6.2 Signature reduction ratio of the *PST* 58
6.3 Partition access against query weight 59
6.4 Retrieval time 61
6.5 Retrieval time vs. processes 61
6.6 Insertion time of *PST* 62
6.7 Deletion time 64

A.1 A Balanced Signature Tree 69
A.2 Case 1: Query Signature Weight is 4 70
A.3 Case 2: Query Signature Weight is 3 71
A.4 Case 3: Query Signature Weight is 2 72
A.5 Case 4: Query Signature Weight is 1 73
A.6 Case 5: Query Signature Weight is 0 73

Chapter 1

Introduction

Signature file is an indexing technique that has been investigated extensively in the document access area. It has been widely applied in office automation [14], software libraries [29] and database application systems [26].

A signature is a binary bit string that represents a word in an abstract format. The main idea of signature technique is that a document is considered as a list of words and is decomposed into n blocks. The words in each block are transformed into signatures with a hashing function. These signatures are superimposed (i.e. bitwise OR) together to form a block signature, and these block signatures are stored in a file named a signature file. The nature of this superimposed mechanism implies that a block signature covers its member signatures. If a query signature is not matched with a block signature, it must not match with any of its member signatures either. Therefore, the signature file technique is able to function as a filter that can discard non-qualified information.

If signatures are stored sequentially in a signature file, searching signatures one by one in a large database is still time consuming. Therefore, many efforts have been undertaken to improve searching performance. Examples are the bit-slice method [25], the multilevel tree [21], signature trees [4] and partitioning signature files [22, 30, 33].

Bit-slice adopts a column-wise storage method to avoid the large amount of unnecessary database accessing. Multilevel tree uses a tree structure to filter nonqualified data from root to leaf level by level. Signature tree organizes a signature file into a binary tree to improve the searching speed. Partitioning signature files divide the signature file into various parts to speed up the searching process.

Recently, there has been increasing interests in applying the signature file technique to the multimedia technology. John Eakins and Margaret Graham [8] classify image queries into three levels of abstraction: "primitive features such as color or shape, logical features such as the identity of objects shown, and abstract attributes such as the significance of the scenes depicted". Obtaining logical and abstract features of an image automatically is still in the research stage. Automatic retrieval of images from databases based on their primitive features is called the *Content Based Image Retrieval* ($CBIR$) that has developed rapidly during the last two decades. Available commercial systems are $QBIC$, $Virage$ and Image Scape. Existing indexing algorithms are R tree [15], SR tree [17] and $K{-}D{-}B$ tree [2]. However, these algorithms become inefficient in large image databases because comparing images one by one needs a large amount of time. The goal of this research is to design an efficient indexing algorithm that can be used to speed up image retrieval. In this research, I extend a model which is known as Retrieval by Spatial Similarity (RSS) [28]. This model searches images based on not only objects, but also spatial relationships among the objects contained in a picture. For example, suppose there is an image with a man sitting in a car. Then, this image will be considered as having two objects: "a man and a car". The spatial relationship between the two objects is a "containing" relationship. The proposed indexing algorithm can answer the query such as "Find all the images including a car and a man", as well as "Find all the images with a man in a car".

The proposed algorithm, called the Image Signature Tree (IST), consists of three parts: Folder Signature Tree (FST), Balanced Signature Tree (BST) and Spatial Relation Tree (SRT). FST is created based on the paths where images are stored. The goal of FST is to filter out the folders that do not contain the target images. BST is a binary-tree-like signature file algorithm aiming at the retrieval of images with target objects. SRT is introduced to locate the images with the constraints by spatial relationships between the target objects.

1.1 Motivation and Problem Definition

An indexing algorithm for large image databases should have an efficient search capability. However, image data are not well-defined as the key words used in traditional textual databases, which makes it more challenging to devise a powerful indexing mechanism over images.

During the last two decades, much research has been done on this issue. For instance, $MPEG-7$ provides a standard for describing contents of multimedia, which includes colors, textures, objects as well as events. Retrieval by spatial similarity (RSS) [28] is a significant approach which searches images based on certain spatial relationships among the objects contained in a pictures. In addition, several modeling approaches [28] have been developed to extract different visual contents of images, such as the color based model, texture based model and shape-based model, which can be used specify searching scopes and objects for the image database retrieval. All these methods are also called content-based image retrieval approaches. The basic idea of content-based image retrieval is to transform the important feature of the image into a high dimensional points(feature vector) and perform the nearest neighbor search, which refers to search the close to a given query feature vector.

Existing multimedia indexing algorithms such as R tree [15], and SR tree [17] established on the objects' spatial location. They use either bounding rectangle or bounding sphere to define the shape of the regions within one image. Therefore, they are only able to retrieve the target image by comparing the images in the database one by one. This will cause an I/O bottleneck if the database is large. To relieve the problem, it is necessary to screen out unrelated images before doing similarity comparison with $CBIR$ [28] technique.

In this research, I try a new way to index images, by which the so called signatures file technique is combined with RSS to discard unrelated images as early as possible. Consequently, the following two problems are investigated.

- How to utilize the filtering power of signature file technique in image retrieval.
- How to speed up the image retrieval, insertions and deletions by using the combination of the signature file technique and RSS model.

Experiments have also been done to show how efficiently my method can screen out non-qualified images for the image retrieval.

1.2 Preliminaries

1.2.1 Signature File Technique

The signature file approach [11] was originally introduced for textual retrieval. A signature is a binary bit string that represents a word in an abstract format. The main idea of signature technique is that a document is considered as a list of words and is decomposed into n blocks. Each word of a block is transformed into a signature with a hashing function. These signatures are superimposed (i.e. bitwise OR, denotes as $\vee$) together to form a block signature, and these block signatures are then stored sequentially in a file named a signature file. Table 1.1 illustrates the construction of a block signature using the superimposed coding method, where a block of a document consists of three words, "sun", "moon" and "star". The signature "001010110"

Text		**Signature**		
sun		001	010	110
moon		101	100	100
star		000	110	101
block signature	$\vee$	101	110	111

Query	Query signature		Block signature		Result	
sun	001010110	$\wedge$	101110111	=	001010110	match
football	010000011	$\wedge$	1011110111	=	000000011	no match
building	100000011	$\wedge$	101110111	=	100000011	false drop

Table 1.1: Signature construction and comparison

is generated using a hashing function to represent "sun", the signature "101100100" represents "moon" and "000110101" represents "star". The block signature is created by superimposing these three signatures. Each 1 in a block signature implies that the bit at the same position in its member signatures should also be one. In other words, the block signature covers its member signatures. If a query signature does not match a block signature, it must not match any of its member signatures. Therefore, signature file technique functions as a filter that is able to discard non-qualified information.

Table 1.1 shows a typical query processing with a signature file: given a query, the query signature is compared with each block signature in a signature file. Those unmatched signatures are screened out from candidates for further checking. The remaining candidate signatures are called drops, and their corresponding text are further examined against the query text. The resulting drop set falls into two categories: those seemingly matched, but actual non-qualified signatures, are called false drops, while those exactly matched signatures are called true drops. Therefore, when querying a signature file, there are three types of outcomes, as shown in table 1.1:(1) match($s \wedge s_q = s_q$), that is, for every bit set to one in the query signature s_q, the corresponding bit in object signature s is also set to one. This means that document contains the real query text. In this case, the query signature is "001010110", and the query result is also "001010110", so these two signatures are matched. (2)unmatch ($s \wedge s_q \neq s_q$), that is, there are some bits in s_q not same with the object signature s. In this case, the query signature is "010000011", and the query result is "000000011", these two signatures are apparently not matched. (3)false drop($s \wedge s_q = s_q$). In this case, the query signature is "100000011", and the query result is also "100000011". These two signatures seem to be matched. However, the corresponding document does not have word "building". This signature is called the false drop.

Superimposed method increases the density of 1's in a block signature, resulting in the degraded filtering ability. As shown in table 1.1, each member signature has 4 bits set 1, while the block signature has seven bits set 1. As more 1's appear in s, the false drop probability increases. Intuitively, using longer signature can decrease false drop probability. However, it will increase the storage overhead.

The false drop probability [9, 21] is crucial in evaluating signature file performance, because it affects the number of database accesses and the query processing time. This probability can be calculated by using the following equation: $P^f(x, w) = (1 - (1 - \frac{w}{m})^x)^\alpha$, where x is the number of distinct words a signature block contains. w is the weight of a member signature, i.e., the number of 1's the signature contains, m is the length of a signature, and α denotes the weight of block signature.

False drop is an inherent problem associated with the signature file method. There are two reasons for the false drop occurring in signature file. One is that the superimposed method does not provide one-to-one mapping for the block signatures to the corresponding sets of words, in other words, two different sets of texts might map to the same block signatures, this will result in a signature seems to in an expected block signature, actually, it is not included. Another is that the hash function during signature generation does not provide one-to-one mapping for the signatures to the corresponding word either. That is, two different texts might map to the same signature. This will cause false drops.

The hash function [10] is a method that turns some data (hash field) into a (relatively) small number(hash address) with a function or an algorithm. One common hash function is the $h(K) = K \ mod \ M$ function, which returns the remainder of an integer hash field value K after division by M, the hash address is between 0 to M, which is smaller than original scope from 0 to K . Some hash functions involve to apply arithmetic or logical function on hash field to generate the hash address. Other hash functions select some digits in the hash fields to form the hash address. Some signature files use triplet hashing algorithm to generate the signature. For example: the signature for the word "research" is expected to be with a length 16 and a weight 4. Then "research" is decomposed into a set of triplets: "res", "ese", "sea", "ear", "arc", "rch". Using a hash function, we have h(res) = 4, h(ese) = 8, h(sea) = 9, h(ear) = 10. Since the expected signature only has 4 bits, so we neglect the rest of two triplets: "arc", "rch". Thus, we establish a signature for "research" which is "0001000111000000".

1.2.2 Image Retrieval

Image Extraction

The complexity of image retrieval lies in visual feature extraction. John Eakins and Margaret Graham [8] defined three levels of image features: primitive features, logical features and abstract features.

Most research work focuses on abstracting an image automatically by using its primitive features such as color, texture, shape and spatial relationships. The idea is to employ a mathematical high dimensional vector to depict the visual feature of an image. For example, a color could be digitalized into a scale of color level according to its three basic color elements: red, green and blue. Therefore, an image can be represent by a collection of pixels with various gray levels labeled as digital numbers. In other words, an image can be regarded as a matrix with various elements. Thus, an image can be abstracted into a dimensional feature vector. Image retrieval is performed through comparing the query image color feature against the color feature of the images stored in databases.

Obtaining a logical feature means extracting a semantic feature of an image such as identifying objects in an image. Automatic retrieval for this high level visual feature of an image is a challenging topic. The idea of most research is to define a model for each object based on its primitive features, and use data mining techniques to identify the objects that an image contains. Kherfi and Ziou [18] think there is a need to define standards for image description. Up to now, there is no significant research on retrieving abstract features reported. Retrieval by spatial similarity (RSS) [28]

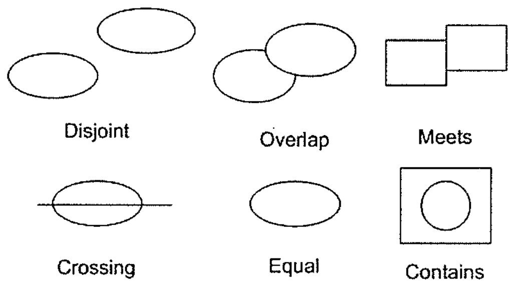


Figure 1.1: Spatial relationship

is a technique that searches images based on certain spatial relationships among the objects contained in pictures [12]. Possible topological relationships between objects are classified into six categories, which are: disjoint, overlap, meets, contains, crossing and equal. Due to the current state of research in image retrieval, I use keywords to represent the identified objects in an image, and created a signature for each keyword in this research.

Image Storage

There are two major approaches [7] for storing images in database. One is to embed images directly in an *OLE* Object field in a database table. The other is to store images on disk or on the network and specify image locations and file names in a database.

Object Linking and Embedding (*OLE*) is the technology developed by Microsoft Corporation for sharing files among different applications By which a picture can be stored as a word document. The main idea of storing images with *OLE* technology is to accommodate each image in a database as an *OLE* Object. Therefore, images can be stored in a database just like other data types, such as text, number and so on. This method is convenient for migration because the images are stored in a database. However, this method can increase the database size because it creates an additional data to render object, which are normally greater than the size of the objects themselves.

Since the size of a database supported by any commercialized system is limited by database size [3]. For example, the maximum size of database that Microsoft access supports is 2 GB, while Microsoft *SQL* 2005 database does not exceed 4 GB. The first method is not suitable for large amount of images. So, I choose the second method to handle the images. The proposed algorithm functions as a filter that cuts off the non-qualified images to reduce the number of disk accesses.

1.3 Objective of this research

Firstly, this research uses an image model to identify image features to facilitate querying. This model describes images based on Retrieval by Spatial Similarity(RSS) model that represent the images by the domain objects and their relationships, and it extracts the image features by their formats and pixel sizes.

Secondly, a signature file algorithm, known as an Image Signature Tree (IST), is presented to retrieve images in a database. The IST consists of three parts: Folder Signature Tree (FST), Balanced Signature Tree (BST) and Spatial Relation Tree (SRT). FST is created based on the paths where images are stored. The goal of FST is to filter out the folders that do not contain the target images. BST is a signature file algorithm aiming at the retrieval of images containing the target objects. SRT is introduced to locate all those images satisfying constraints specified by the spatial relationships between the target objects.

Thirdly, this research presents an efficient partitioning scheme for BST. The BST organizes the signatures in the signature file into a binary tree-like structure. When the image database gets very large, a balanced signature tree could exceed the size of main memory, resulting in a memory bottleneck. Therefore, the index may need to be tailored smaller to fit into main memory. For this reason, an extra partition scheme is proposed to divide the BST into smaller parts with each part holding the same key. This scheme saves the searching space and supports the parallel programming. Furthermore, this scheme provides a dynamic structure that facilitate image insertions, deletions.

Finally, this research provides an in-depth analysis for the balanced signature tree based on probability theory. Simulations are conducted to compare the theoretical value and experimental results.

1.4 Organization

The rest of the thesis is divided into six chapters. They are organized as follows:

- *Chapter 2: Literature Review.* This chapter discuss the related work and provides the background for this research. It begins with an overview of the signature file technique, followed by an introduction of some signature file algorithms. Then it presents some basic knowledge of image retrieval and image storage. Finally, it discusses the application of the signature file technique on signature file, with focus on the related work done on partitioning of signature files in the literature.

- *Chapter 3: Image Signature Tree Algorithm.* This chapter introduces the main algorithm, that is called as Image Signature Tree (IST). First, it presents an image model that aims at defining searching scope and features of an image. Next, it gives a detailed description of the image signature tree algorithm. This chapter ends up with a discussion of an efficient partitioning scheme for BST.

- *Chapter 4: Analysis.* This chapter presents a probabilistic analysis for the signature reduction ratio of Partitioned Signature Tree (PST) algorithm. It describes the performance metrics, including the update cost, the retrieval cost, and a list of control parameters.

- *Chapter 5: Implementation.* This chapter shows implementation details. It starts with a description of the implementation of a prototype system with the IST algorithm, which can be used to retrieve an image or several images from the database. Then, it describes how to implement the PST.

- *Chapter 6: Experimental Results.* This chapter presents and analyzes the experimental results. First, a theoretical analysis is conducted on the performance of BST and PST using methodology in the chapter 4. Then test results are demonstrated to show the difference between these two strategies. It also provide the analysis for the simulation results.

- *Chapter 7: Discussion and Conclusions.* This chapter summarizes the contributions of this research and suggests the future research directions.

Chapter 2

Literature Review

This chapter provides a broad overview of the background material on which the thesis builds. The first section describes various signature file algorithms. Following this, a selection of related work on image retrieval is presented in 2.2. In 2.3, the research efforts to improve the performance of the sequential signature file algorithms by the partitioning approach are reviewed.

2.1 Signature Algorithms

2.1.1 Sequential Signature File

Sequential signature file is the simplest way for organizing signatures. Searching a signature is done by comparing signatures one by one. The performance of this method depends the size of the given signature file and the length of each signature in it. Therefore, this method is not efficient when applied to a large database.

2.1.2 Bit Slice File

Roberts [25] proposed a bit slice (BS) method which regards a signature file with n signatures with length m as a table with n rows and m columns. Such table is then

Signature file:

C1	C2	C3	C4	C5	C6	C7	C8	
0	0	1	0	1	0	0	1	S1
0	1	1	1	0	0	1	0	S2
0	0	1	0	1	1	0	1	S3
1	1	0	0	1	0	1	0	S4
0	0	1	1	1	0	0	1	S5
1	1	1	0	0	1	1	0	S6
0	0	0	1	0	0	1	1	S7
0	1	0	1	0	1	1	0	S8

bit-slice file:

C1	C2	C3	C4	C5	C6	C7	C8	
0	0	1	0	1	0	0	1	S1
0	1	1	1	0	0	1	0	S2
0	0	1	0	1	1	0	1	S3
1	1	0	0	1	0	1	0	S4
0	0	1	1	1	0	0	1	S5
1	1	1	0	0	1	1	0	S6
0	0	0	1	0	0	1	1	S7
0	1	0	1	0	1	1	0	S8

Figure 2.1: Signature file and bit slice signature file

divided into m slices. Each slide corresponds to a column and stored in a different file. When answering a query signature with w 1's, only w corresponding columns of files are checked. Compared to sequential signature file, this method can save $m - w$ unnecessary accessing time. However, the maintenance cost is very large. For example, if we update a signature in BS, m bit-slice files need to be accessed.

2.1.3 Multi Level Signature File

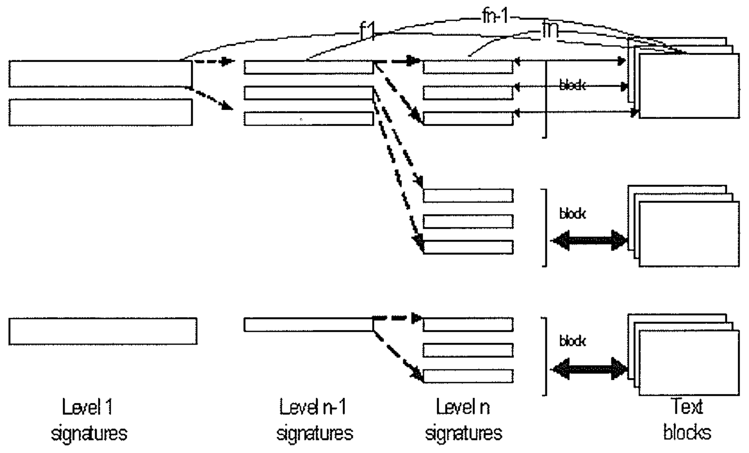


Figure 2.2: Multi level signature file

Multi-level signature file [21, 33], as shown in figure 2.2, is a b-ary tree structure with n levels. Different from binary tree with each parent having two children, this b-nary tree has b children for each parent. The signatures in the lowest level, nth level, are generated from text blocks. Each signature of $n-1$th level is a parent node with b children of signatures at level n, and is generated from a group of text blocks which create its b children signatures at level n. The advantage of this method is that it filters non-qualified documents at higher level, reducing unnecessary accesses at lower level. However, it creates extra storage overheads for the high-level signatures.

2.1.4 Partitioned Signature File

The idea of partitioning scheme [22] is to divide a signature file into small parts in which the signatures hold the same key. Figure 2.3 shows an example of a partition scheme. It involves two levels. The first level is a key table; the second level contains all the signatures that are partitioned into several parts. The signatures in a part have the same key. When a query signature comes, it was first checked against the key table to find all the parts with a matching key. Then all the signatures in these parts will be further examined. The partitioning scheme skips the further checks with unmatched partitions. Therefore, it improves the performance by reducing I/O accessing time.

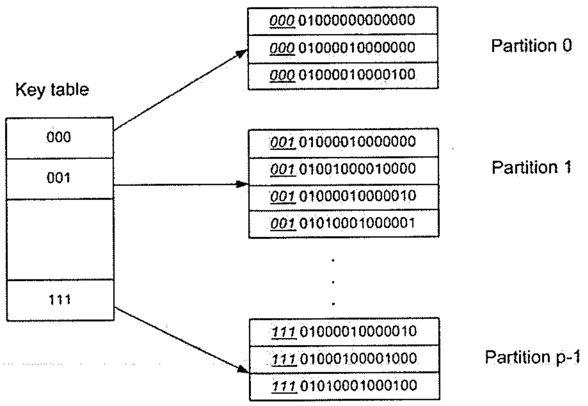


Figure 2.3: Partitioned signature file

2.1.5 Signature Tree

A signature tree (ST) [4] is constructed based on the so-called signature identifiers which is defined as follows:

Definition (Signature identifier) Let $S = s_1.s_2...s_n$ denote a signature file. Consider $s_i(1 \le i \le n)$. If there exists a sequence: $j_1, ...j_h$ such that for any $k \ne i(1 \le k \le n)$,we have $s_i(j_1, ...j_h) \ne s_k(j_1, ...j_h)$, then we say $s_i(j_1, ...j_h)$ identify the signature s_i.

Definition (Signature tree) A signature tree for a signature file $S = s_1 s_2...s_n$, where $s_i \ne s_j$ for $i \ne j$ and $|s_k| = m$ for $k = 1, ..., n$, is a binary tree T such that

1. For each internal node of T, the left edge leaving it is always labeled with 0 and the right edge is always labeled with 1.

2. T has n leaves labeled 1,2,...n, used as pointers to n different positions of s_1, s_2... and s_n in S. Let v be a leaf node. Denote $p(v)$ the pointer to the corresponding signature.

3. Each internal node v is associated with a number, denoted $sk(v)$, to tell which bit will be checked.

4. Let $i_1, ...i_h$ be the numbers associated with the nodes on a path from the root to a leaf v labeled i (then, this leaf node is a pointer to the ith signature in S, i.e $p(v) = i$). Let $p_1, ...p_h$ be the sequence of labels of edges on this path. Then,$(j_1, p_1)...(j_h, p_h)$ make up a signature identifier for s_i, $s_i(j_1, ...j_h)$.

Figure 2.4 illustrates a process for constructing a signature tree from a signature file. A successful signature search works as follows: given a query signature, we start from the root of the ST and get internal node value n, check the query signature at position n. If it is 1, we only need to further query the right subtree; if it is 0, we continue to search both left and right subtrees. The above process is performed recursively until the leaf level is reached. All the reached signatures are the candidates which satisfy the users' query except for false drops.

Signature tree is different from ordinary Binary Tree (BT) in two sides: First,

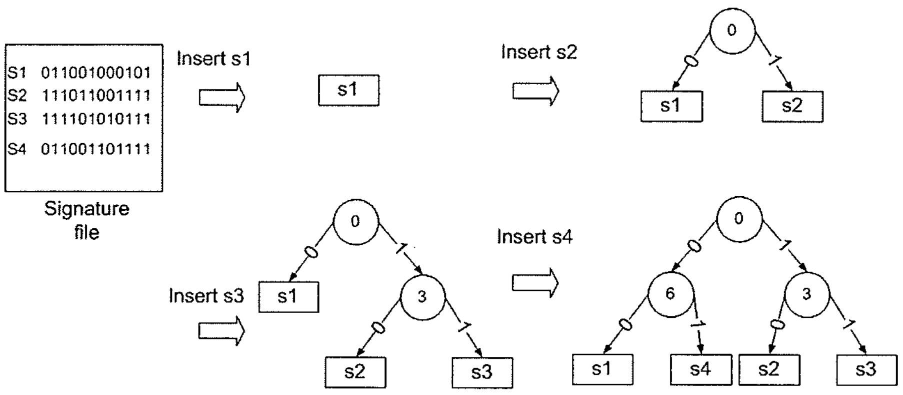


Figure 2.4: Signature tree

it has more basic structures than the binary tree does. Figures 2.5 shows its four different basic structures. While the ordinary binary tree only has the one type structure which is similar to the type A. Type A has three internal nodes; Type B has two signatures and one internal node; Type C and D has three signatures and two internal nodes. This complex structure makes signature tree more complicated than binary tree in term of construction, storage and retrieval. Since signature size is usually longer than internal nodes, it requires more storage space. This feature requires that signature tree should have different storage and retrieval scheme than the binary tree. Second, the internal nodes in *ST* act as not only link roles as in binary tree, but also as position pointers. Once the *ST* is constructed, the connections among the nodes are fixed. Any changes to the internal nodes will lead to a totally different *ST*. So, the balancing tree to make the left and right sub-tree have equal or almost equal depth by rotation in the *BT* is not applicable to the *ST*.

2.1.6 Balanced Signature Tree

In order to avoid the worst case of *ST*, *Chen* [5] proposed a weight-based method to organizes a signature file into an approximately balanced binary tree.

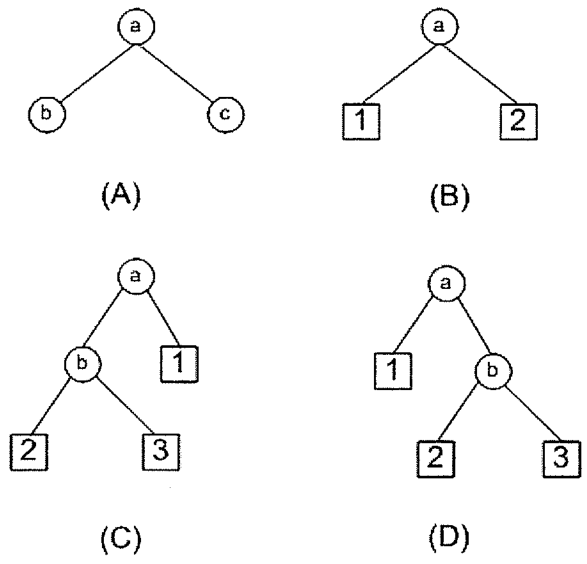


Figure 2.5: Basic signature tree structure

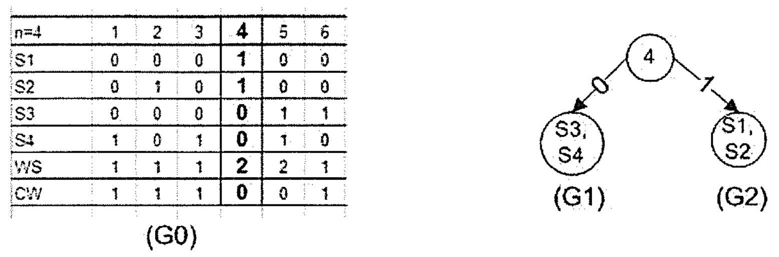
Weight-based method

A signature file $S = S_1S_2...S_n$ can be considered as a Boolean matrix. Let $WS[i]$ be the sum of 1's in the ith column, and $S_{i,j}$ be the bit of signature S_i in the jth column. Define $CW[i] = |WS[i] - \frac{1}{2} * n|(i = 1, 2...n)$, and then choose the first column j where $CW[j]$ is the minimum as a pivot column. The signatures are divided into two groups: G_1 where $S_{i,j} = 0$, and G_2 where $S_{i,j} = 1$.

For example, suppose we have a set of signatures $G0 = \{s1, s2, s3, s4\}$, here, $s1 = 000100, s2 = 010100, s3 = 000011, s4 = 101010$. According to the weight-based method, $G0$ can be divided into two groups $\{s3, s4\}$ and $\{s1, s2\}$. This procedure is shown in Figure 2.6.

Balanced signature tree

Constructing a signature tree is a recursive procedure. First, the original signature sets G_0 is split into two groups G_1 and G_2 according to the weight-based method by



n=4	1	2	3	4	5	6
S1	0	0	0	1	0	0
S2	0	1	0	1	0	0
S3	0	0	0	0	1	1
S4	1	0	1	0	1	0
WS	1	1	1	2	2	1
CW	1	1	1	0	0	1

Figure 2.6: Weight-based method

the pivot column d. Next, the pivot column d is formed as a internal node P_0 with value of d. The left child of P_0 is the group where $S_{i,d}$ equals zero, while The right child is the group where $S_{i,d}$ equals one. Then, subgroup G_1 and G_2 are split as in the first step, generating a position node P_1 and its subgroups G_{11} and G_{12}, as well as P_2 and G_{21} and G_{22}, respectively. At this stage, P_0 updates its left child to P_1 and right child to P_2. Next, subgroups G_{11}, G_{12}, G_{21} and G_{22} continue the first step until all the descendant subgroups contain only one signature. Figure 2.7 shows a process for generating a BST.

2.2 Signature File in Image Research

Nascimento and Chitkara [24] proposed a new signature file technique based on color similarity to retrieve image databases. The main idea of this algorithm is to use signature bit-strings to represent the high-dimensional features extracted from images. They compare this new signature algorithm against a SR-tree in terms of storage overhead and querying process, and claims that storage overheads for 2000 binary images with signature method are 97% smaller than SR-tree [17], while query saving is over 80%. The weak point of this research is that they have not used any signature file technique to improve querying speed.

Kwae and Kabuka [9] introduced a two signature multi-level signature file ($2SMLSF$)

n=4	1	2	3	**4**	5	6
S1	0	0	0	**1**	0	0
S2	0	1	0	**1**	0	0
S3	0	0	0	**0**	1	1
S4	1	0	1	**0**	1	0
WS	1	1	1	**2**	2	1
CW	1	1	1	**0**	0	1

(G0)

n=2	1	**2**	3	4	5	6
S1	0	**0**	0	1	0	0
S2	0	**1**	0	1	0	0
WS	0	**1**	0	2	0	0
CW	1	**0**	1	1	1	1

(G1)

n=2	**1**	2	3	4	5	6
S3	**0**	0	0	0	1	1
S4	**1**	0	1	0	1	0
WS	**1**	0	1	0	2	1
CW	**0**	1	0	1	1	0

(G2)

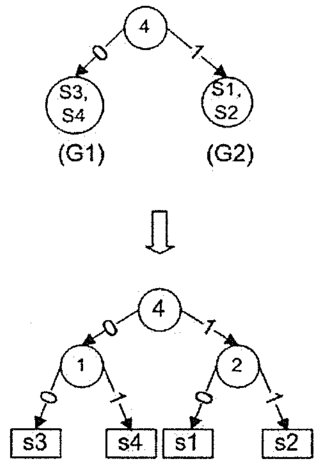


Figure 2.7: Balanced signature tree

as an indexing method for image databases. This algorithm is an extension of multi-level signature file. It consists of two types of signatures. The first type is an exact multi-level tree which is used to retrieve the objects that an image database contains, while the second type is a sequential signature file which is used for retrieving images based on the image objects and their spatial relationships. The advantage of this method is that it can address different query requests, such as, finding all images including a given set of objects or finding all images with specified spatial constraints. Furthermore, an analytical result demonstrates that $2SMLSF$ has smaller storage requirement and more efficient querying speed than the existing signature file techniques. However, the author has not provided useful experimental results to verify the analytical estimation.

Lee and Huang [20] implemented a signature algorithm based on image objects and spatial relationships between objects. This algorithm differs from $2SMLSF$ in two aspects: 1) signatures are created with only one object in an image and the spatial relationships between an object and the rest of objects in the image; 2) signature files are organized by a Hierarchical Relation (HR) graph instead of multi-level signatures. HR graph speeds up the signature searching process by introducing virtual signatures that are not associated with real images. Those virtual signatures, however, increase the storage requirement.

2.3 Partitioned Signature File Research

Zezula [30] proposed a partition scheme named quick filter using linear hashing to group similar signatures into one page. The advantage of this method is that it supports dynamic storage structure that allows a large amount of insertions into databases. Moreover, Zezula and Rabitti studied the so called quick filter [33], and concluded that this method is superior to sequential signature file and multilevel signature file when querying weights are high. Lee and Leng [22] presented three partitioning schemes: the Fixed Prefix method, the Extended Prefix method and the Float Key method. These three methods share a common feature, that is, the original signatures are converted into a partition key table and the corresponding signature partitions. The difference between these three methods lies in the scheme of generating the partition keys from signatures. Fixed Prefix method extracts partition key with a fixed length and a fixed starting position. Extended Prefix method selects a key with variable length but with a fixed starting position. Float Key method chooses a key with a variable position but with a fixed length. The major advantages of these methods are their simple structure. However, they do not support dynamic storage structure. When one partition overflows, the schemes become inefficient.

Gradi, et al. [13] suggested a frame slice partitioned scheme to reduce the update cost caused by bit-sliced storage scheme. Lin [23] implemented the frame-slice signature on Unix workstations and studied the performance in term of false drop probability and response time. Zezula et al. [32] introduced an approach that combines the key-based partitioned method and bit-sliced signature file. This hybrid method shows good searching performance, however, the update cost is still high. They further suggested integrating key-based partition scheme with frame-slice approach. Kim and Chang [19] integrated hashing method and frame slice technique together and proposed a new scheme called a horizontally divided parallel signature file. The approach shows superior performance compared to frame-slice by Gradi and other parallel signature file algorithms published in the literature.

Zezula and Tiberio [31] introduced a Hamming filter that extends quick filter. However, it fails to distribute signatures evenly into partitions, causing unbalanced work loads among partitions. To solve this problem, Shin et al. [27] proposed a dynamic signature file declustering method based on the signature difference. It adopted hamming filter when partitioning a signature file. This is different from my proposed partition scheme, which is based on data structure key and signature difference.

Chapter 3

Image Signature Tree Algorithm

This chapter focuses on the methods and techniques used in this thesis. This chapter first introduces an image model designed to describe the images features. Next, it describes the Image Signature Tree algorithm in detail by introducing its three basic operations, such as query process, insertion and deletion. Finally, it introduces a partitioned signature tree algorithm that is the core component of *IST* algorithm.

3.1 Image Model

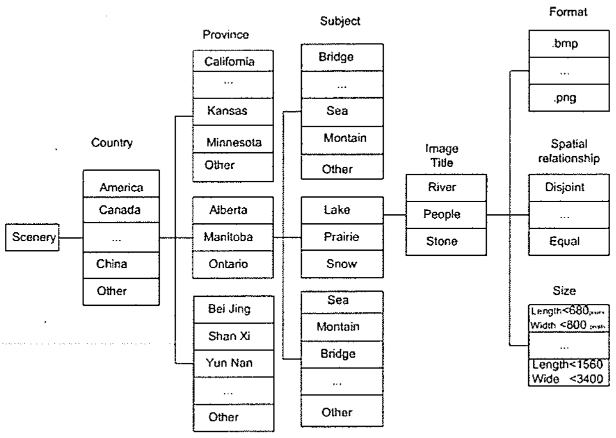


Figure 3.1: Image model

The model designed to describe an image is illustrated as shown in Figure 3.1. It consists of a set of abstract features, a set of object features and a set of primitive features.

The abstract features include:

- type: such as wallpaper, scenery, sports, car, animal and so on.
- subject: such as fish, bird and so on.
- title: such as salmon, catfish and so on.
- size: such as 680×1200, 800×780 and so on.
- format: such as JPG, GIF, BMP, PNG and so on

The object features include a set of keywords which describe an image. The primitive features include: DISJOINT, OVERLAP, MEET, CONTAIN, CROSSING and EQUAL.

3.2 Image Signature Tree Algorithm

According to the image model, an image signature can be defined. It consists of five fields: a folder field, an object field, a relation field, a size field and a file extension type field. Folder signature is based on the abstract feature of the image model. For example, in the above image model, the "scenery" signature is generated by superimposing its subfolders. The subfolder "country" is generated by superimposing its subfolders. The length of folder signature is chosen based on the number of its subfolders for the purpose of minimizing the false drop probability. These folder signatures finally form a multilevel signature tree.

The object signature is obtained based on the objects in an image. For an image with k objects, the object signatures are generated by a hash function, and these k

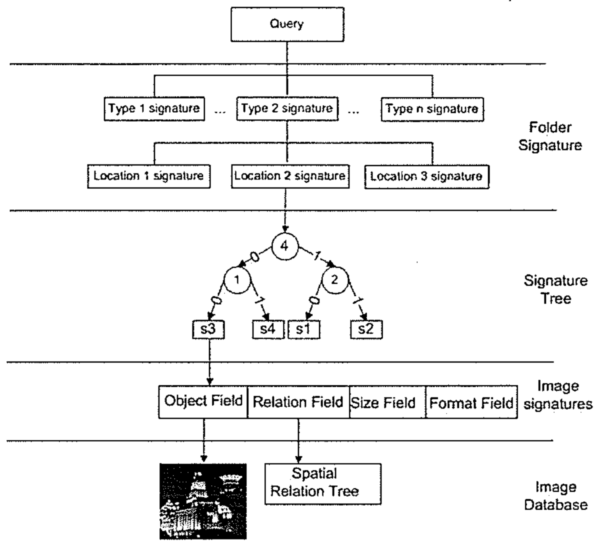


Figure 3.2: Image signature tree

object signatures are superimposed together to form a block signature, which is the object field of an image signature. k objects in an image have $k*(k-1)$ relationships, which fall into six categories: DISJOINT, OVERLAP, MEET, CONTAIN, CROSSING and EQUAL. The relation signatures of an image are superimposed together to generate the relation field. Since there are only six spatial relationships defined in this research, I use a six-bit string to represent relation signatures. For example, I use "100000" to stand for the "DISJOINT" relation and "000010" to represent the "CONTAIN" relation. If all the object pairs in an image belong to the "DISJOINT" and the "CONTAIN" relationships, then the image relationship signature is "100010". The size field and the format field are generated similar to the relation field.

An IST, as illustrated in Figure 3.2, includes three different types of trees: folder signature tree, balanced signature tree and spatial relation tree.

The FST is created by superimposing the subfolders level by level. The leaf of a folder tree is pointed to the root of balanced signature tree. The leaf of a signature tree is the object field of image signatures, it also points to the relation field, the size

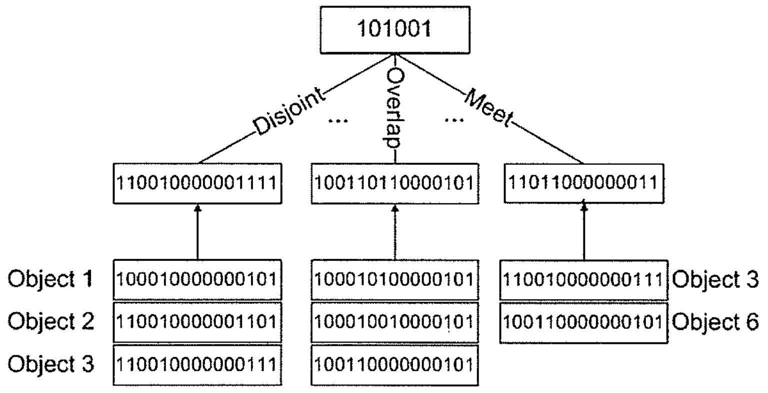


Figure 3.3: Spatial relation tree

field as well as the format field.

To further locate the images queried by spatial relationships between the target objects, I introduced a Spatial Relation Tree (*SRT*), as shown in Figure 3.3. The root of *SRT* is a relation field of an image. It has six children which represent six spatial relationships of object pairs in an image. The leaf nodes of *SRT* are the block signatures of objects within the same relationships.

3.2.1 Partitioning Algorithm for a *BST*

If the image database is very large, a *BST* could exceed the size of main memory. To solve this problem, I proposed two partitioning schemes for a *BST*.

Scheme 1

Since the *BST* is constructed from top to down, the first scheme I present also works in top-down order. The idea is to specify a fixed partition size and start to construct the first partition as the *BST* does. Once the partition is oversized, new children partitions are generated to continue the uncompleted splitting process. Figure 3.4 illustrates this method. The advantage of this method is that the tree in every partition is balanced. This balanced tree structure in turn benefits the signature retrieval. However, waste of partition space caused by a partition is a big problem. As shown in Figure 3.4, some partitions only contain one signature. If these small partitions are merged into one partition to save storage space, the tree structure in each partition will be damaged, and the signature search will use more time. Another problem is that more space may be wasted when the larger partition size is used. The larger partition size allows one partition to accommodate more nodes or signatures. But this will result in more number of nodes at the bottom level for each partition tree. These nodes are connected to the children partitions by parenting of the root of the partition trees. As the partition number increases, the inherent inefficient storage use for each partition will cause more space waste. Therefore, this scheme is not a robust way to partition the balanced signature tree.

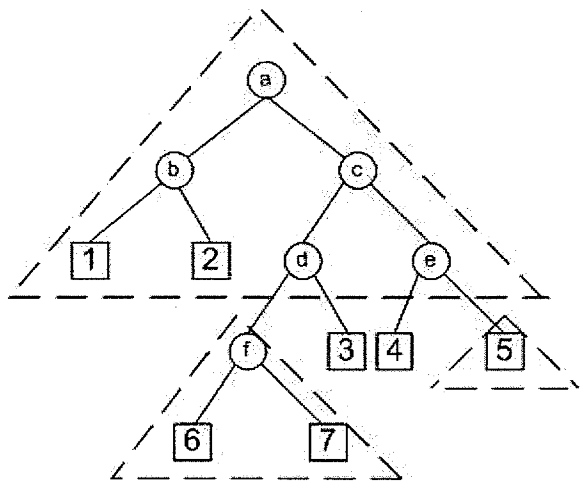


Figure 3.4: Partitioning scheme 1

Scheme 2

Scheme 2 is proposed to solve the problems occurring in scheme 1. Like other partitioned signature file schemes, the signatures in a partition are organized into a balanced signature tree. Figure 3.5 illustrates the structure of scheme 2. The partitioned signature tree involves two levels. The first level is a key tree; the second level is the partitions that are a set of balanced signature trees. The key tree represents the set of keys: $\{00, 010, 011, 10, 11\}$, the partition 1 has a key $\{00\}$. This key is comprised of two bits, the left child of the node with value of 4 determines the first bit "0" and the left child of the node with value of 1 determines the second bit "0". Therefore, in this partition, all the signatures with a common feature, which is to a 0 bit at the position 4 and the position 1. Partition 5 has a key $\{011\}$. This key consists of three bits, the left child of node with value of 4 determines the first bit "0", the right child of node with value of 1 determines the second bit "1", the right child of node with value of 6 determines the third bit "1". All the signatures with a "0" bit at position 4, a "1" bit at position 1 and a "1" bit at position 6 are organized into this partition. Leaves of the key partition tree contain the partition identification number and the total signature numbers of that partition.

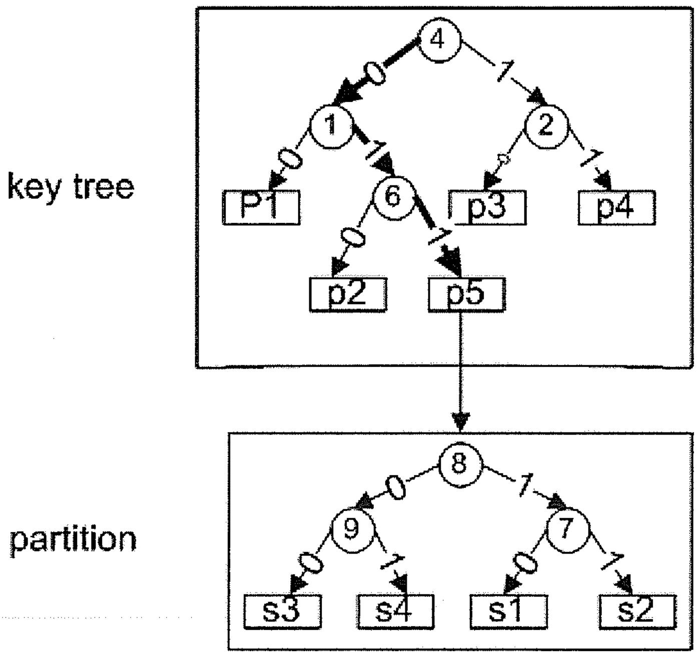


Figure 3.5: Partitioning scheme 2

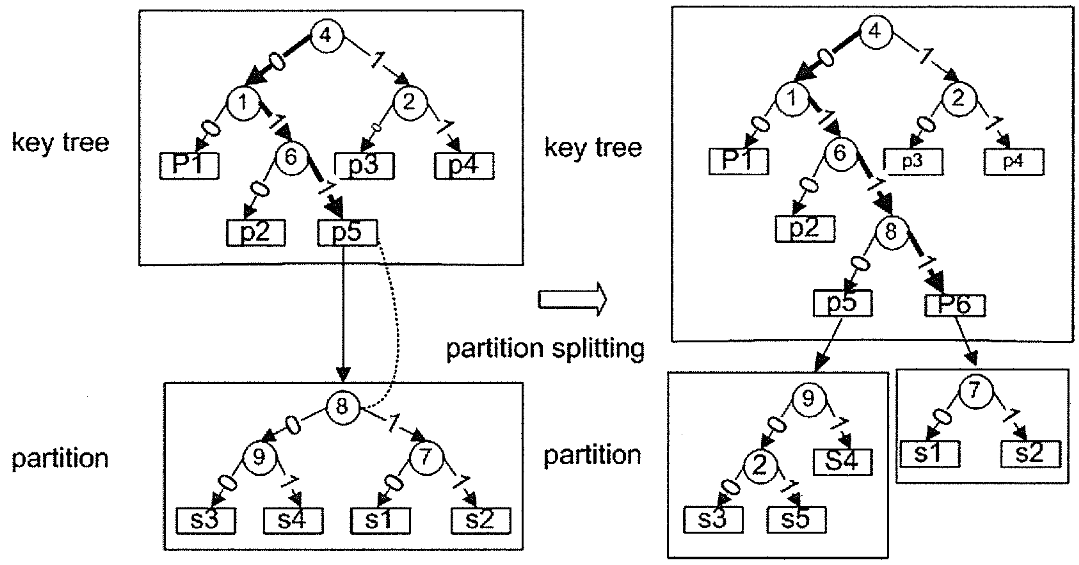


Figure 3.6: Partition splitting

Contrary for scheme 1, scheme 2 uses a signature tree in its key tree part which sacrifices a little balance. However, Scheme 2 improves on the scheme 1 in partition space use. When a new inserted signature happens to cause a partition over its size, instead of splitting at the bottom, the partition breaks from the root. As shown in Figure 3.6, the root (the node with value 8) of the signature tree in that partition will move to the key partition tree. The left subtree (the root is the node with value 9) and right subtree (the root is the node with value 7) will form into two new partitions. Each new partition is nearly half the size of the old partition. Once these new partitions grow to be oversized, they can be divided into smaller ones.

There are several advantages of this partitioning scheme. First, it distributes the signatures into partitions with a uniform size, alleviating the I/O bottleneck caused by over accessing in one partition. Second, it allows dynamic partitioning, which facilitates large numbers of insertions in a database while sacrificing only some equilibrium of the signature tree.

3.2.2 Query Processing

Once the folder signature tree, signature tree and spatial tree are created, IST can answer the following three types of queries :

Algorithm: IST retrieval type 1

Objective: Search for all images including object O_i and O_j with subject F_k

1. Construct a query signature for an image I

2. Start from root of IST, go to folder signature tree. If there is no F_k directory, algorithms stops. Otherwise, go to step 3

3. Go to the key partition tree, search the partitions containing the query signature, if the result is zero, algorithm stops. Otherwise, go to step 4

4. Go to the signature tree in each candidate partition, search for target signatures matched with the query signature.

Algorithm: IST retrieval type 2

Objective: search for Image I containing object O_i and O_j, and O_i and O_j meets

1. Generate query signature of image I and the relationships signature between O_i and O_j

2. Start from root of IST, go through FST and enter into key partitions tree

3. Search the partitions containing the query signature. If the result is zero, algorithm stops. Otherwise, go to step 4

4. Go to the root of signature tree in each candidate partition, search for target signatures matched with query signature. If the result is zero, algorithm stops. Otherwise, go to step 5

5. Enter into relation field. If it contains *"meets"*, search the relationships signature for O_i and O_j. Otherwise, algorithm stops.

Algorithm: IST retrieval type 3

Objective: search for all JPG images containing object O_i and O_j

1. Generate a query signature for an image I and a relationship signature between O_i and O_j

2. Start from root of IST, go through FST and enter into key partitions tree

3. Search the partitions containing the query signature. If the result is zero, algorithm stops. Otherwise, go to step 4

4. Go to the root of signature tree in each candidate partition, search for target signatures matched with query signature. If the result is zero, algorithm stops. Otherwise, go to step 5

5. Enter into format field, if it contains JPG, retrieve this image from disk. Otherwise, algorithm stops.

3.2.3 Insertion

Algorithm: Insertion

Objective: insert an image I with objects O_i and O_j

1. Assign the inserted image an identifier and generate an insert signature for an image I and a relationship signature between O_i and O_j

2. Start from root of IST, search the corresponding folder

3. Go to the PST. Allocate the image I into the corresponding partition based on its signature, increase the signature count at the leaf node. If the count number is less than the partition size, go to step 6

4. Go into signature tree, and split the signature into two partitions, the root of signature tree migrates to the key partition tree as new leaf node

5. Update the partition number of key partition tree and signature count information at new leaf node

6. Go into signature tree, insert image I signature at the leaf level, and establish a pointer to image, fill the relation field, format field and size field, and so on

7. Construct an SRT tree based on relationship between O_i and O_j.

3.2.4 Deletion

Algorithm: Deletion

Objective: Delete Image I with objects O_i and O_j

1. Generate the supposed to be deleted signature of image I

2. Start from root of IST, go to the PST

3. Go to the corresponding partition based on the deleted signature. Decrease the corresponding signature count at the leaf node

4. Enter into the signature tree, find the target signature, delete the SRT tree and delete the signature, including relation field, size field and so on.

Chapter 4

Analysis

Since the performance of the signature tree depends upon the distributions of "1"s and "0"s of a signature file, the probabilistic (average-case) analysis is selected as a tool for evaluating the filtering ability of the algorithm. This chapter provides a probabilistic analysis for the partitioned signature tree in term of signature reduction ratio, retrieval cost as well as theoretical analysis in terms of update cost. The control parameters that affect the performance of the PST are listed in section 4.1. Furthermore, the performance metrics are explained in section 4.2. The focus of this chapter is the analysis of the signature reduction ratio of the balanced tree.

4.1 Control Parameters

Table 4.1 lists several variables that will affect the performance measures. They include: the total number of images, the size of the partition, the size of key tree, and so on. Since the number of edge traversed through the key tree by each search is variable, the average number of edge($\overline{k}$), as well as the average weight in the query key $\overline{w_{qkey}}$ are used to simplify the analysis.

n	total number of images in a database
P_i	partition i
p	size of a partition
s	number of distinct objects in a signature file
s_i	number of distinct objects in partition P_i
$\overline{k}$	average length of key tree
$\overline{w_{qkey}}$	weight in a query key signature
w	number of 1's in an object signature
w_i	number of 1's in an image I object signature
w_q	number of 1's in a query signature
m	length (in bits) of a signature
mko	length (in bits) of a key tree node
mpo	length (in bits) of a balanced tree internal node
M	total number of bits in storage

Table 4.1: Control parameters

4.2 Performance Metrics

4.2.1 Storage Cost

Before analyzing the storage cost for the *PST*, let us go back to review the four basic structures in Figure 2.5 in the chapter 2. Type (B),(C) and (D) indicate that every two signatures need one internal node to distinguish the bit difference. Type (A) illustrates that every two internal nodes needs one internal node to retain the parent-children relationships. For a signature tree with n signatures, it has $n-1$ internal nodes. we can conclude that storage cost for a signature tree is the sum of n signatures and $n-1$ internal nodes. For the same reason, the storage cost of a partitioned signature tree is the partition numbers n_p plus $n_p - 1$ key tree nodes.

The storage size of the *PST* is determined by the size of the key tree and the partition, as well as the number of partitions and the storage size of signatures.

The total number of partitions (n_p) can be calculated through dividing the total number of images (n) by the maximum partition size (p), as shown in equation 4.1:

$$n_p = \lceil n/p \rceil \tag{4.1}$$

Thus, the storage size of key tree, M_{key}, is:

$$M_{key} = (n_p - 1) * mko; \tag{4.2}$$

The storage space for the balanced signature tree includes internal nodes M_{node} and signature leaves $M_{signature}$. They are calculated by the following formulas:

$$M_{node} = (n-1) * mpo; \tag{4.3}$$

$$M_{signature} = m * n \tag{4.4}$$

where m is calculated by using the following formula given in [21].

$$m = (1/\ln 2)^2 s \ln(1/P^f) \tag{4.5}$$

and

$$P^f = 0.5^w \tag{4.6}$$

Therefore, the total storage cost of the partitioned signature tree, M, is:

$$M = M_{key} + M_{node} + M_{Signature} \tag{4.7}$$

4.2.2 Signature Reduction

The signature reduction ratio is the total number of signatures for further checked against the query text over the total number of signatures, as shown in equation 4.8. It reflects the filtering ability of a signature file algorithm to cut off the unmatched signatures. The smaller signature reduction ratio is, the better filtering performance a signature file algorithm has. It is affected by the length of a signature, the weight of signatures, and the signature file size [21]. Therefore, for the same size *BST* and *PST*, if signature length and the weight of signature are same, the signature reduction rates are equal.

$$Signature\ reduction\ ratio = \frac{The\ total\ number\ of\ drops}{The\ total\ number\ of\ signatures} \tag{4.8}$$

Assuming a *BST* has the signatures of length m bits. If the given query signature is of a weight i, then the signature drops will be $(0.5)^i * 2^m$ [21].

$$Signature\ reduction\ ratio[weight = i] = \frac{(0.5)^i * 2^m}{2^m} = (0.5)^i \tag{4.9}$$

The average number of signature drops of a *BST* is:

$$Average\ signature\ drops = \sum_{i=0}^{m} X_i P_i = \sum_{i=0}^{m} 2^m * (0.5)^i * \binom{m}{i} / 2^m = \sum_{i=0}^{m} 0.5^i * \binom{m}{i} \tag{4.10}$$

Weight i	Signature Drops $X = 0.5^i * sample\ space(2^m)$	Numbers of Signature with same weights$\binom{m}{i}$	Probability $\binom{m}{i}/2^m$
i=m	$2^m * (1/2)^i = 1$	1	$1/2^m$
i=m-1	$2^m * (1/2)^i = 2$	m	$m/2^m$
i=...	...	...	...
i=1	$(2^m) * (1/2)^i = m$	m	$m/2^m$
i=0	$(2^m) * (1/2)^0 = 2^m$	1	$1/2^m$

Table 4.2: Signature drops against weight

4.2.3 Retrieval Cost(RC)

This research assumes that the partition is the basic unit for data transfer between the main memory and the external storage device, and I/O access cost is the main cost during querying process. Therefore, the retrieval cost of the partitioned signature tree depends on the number of activated partitions.

Partition Activation Probability

The partition activation probability, Pr, can also be derived from the signature reduction analysis. Equation 4.11 gives the formula.

$$\Pr = \binom{\overline{k}}{w_{qkey}}/2^{\overline{k}} \tag{4.11}$$

So, the activated partitions, P_a, can be obtained with equation 4.12.

$$P_a = 2^{\overline{k}} * 0.5^{\overline{w_{qkey}}} * \Pr = 0.5^{\overline{w_{qkey}}} * \binom{\overline{k}}{w_{qkey}} \tag{4.12}$$

Retrieval Cost of Partitioned Signature Tree

The retrieval cost of the key tree, KRC, is:

$$KRC = 1 * I/O\ Cost \tag{4.13}$$

The activated partitions cost, PRC, is as follows:

$$PRC = P_a * I/O\ Cost \tag{4.14}$$

Therefore, the retrieval cost of partitioned signature tree, RC, is:

$$RC = KPC + PRC \tag{4.15}$$

4.2.4 Update Cost(UC)

An update of the partitioned signature tree could be an insertion or a deletion operation. For simplicity, the costs of partition splitting and partition merging are not considered in this research.

The insertion operation, (UCI), includes the retrieval cost and the signature insertion cost.

$$UCI = Retrieval\ Cost + Insertion\ Cost \tag{4.16}$$

The deletion operation, (UCD), involves the retrieval cost and the signature deletion cost.

$$UCD = Retrieval\ Cost + Deletion\ Cost \tag{4.17}$$

For a better understanding how the signature reduction ratio is achieved, I give an example in appendix A to illustrate the analytical estimation.

Chapter 5

Implementation

This chapter describes the prototype of the Image Retrieval System using IST. Due to the lack of real test data, we created a set of bit strings to represent signatures and map them to folder texts and image objects. The purpose of implementing the prototype was to demonstrate how to locate the desired image without loading it into memory. This chapter begins with the discussions of the file system and $DBMS$, proceeds with the system design and ends up with some implementation details. Finally, the implementation of the PST algorithm is presented using the flow chart and the class relationships diagram.

5.1 Prototype

5.1.1 Introduction

A database is a collection of data stored in a computer in such a way that information can be retrieved from it [10]. Before the emergence of database, the file system was the dominant way for people to organize data and it is still in use in our daily life. In the file system, the application programs control the data directly, this make it efficient in dealing with small numbers of items. However, file system suffers from the separation and isolation of data when people need to cross-refer the related information [6]. For example, assume that in an online store, the customer order file and the product stock file are accessed and maintained by two programs respectively. If a customer buys a product from the store, then, the quantity of this product in store should be decreased by one. Since the product stock file is totally separated from the customer order file, some work need to be updated the product stock file. To make the matter worse, the problem with data inconsistency becomes more severe. Moreover, isolated files produce a huge amount of data redundancy which wastes both time and space. In addition, program-data dependence, the inherent feature of the file system, restricts the portability and reusability of the application system.

The limitations of the file system have led to the introduction of databases. The database approach solves the problems by separating application program from data, organizing the related data logically and sharing data among the applications.

The main component of the database approach is the Database Management System($DBMS$). A $DBMS$ is a software system that supports the creation, maintenance and access to the database. It provides a Data Definition Language (DDL) for users to define a database and a Data Manipulation Language (DML) for users to retrieve and update data from database. Through controlling access to the database, $DBMS$ offers a number of advantages including security, data consistence, data sharing and so on.

Among the variety of $DBMS$s, the Relational Database Management System ($RDBMS$) is most extensively used, such as Microsoft Access, Microsoft SQL and *Oracle*. The relational database organizes the data based on the relations or say tables with rows and columns. The rows represent the collection of records, and columns represent the attributes contained in each record.

The indexing technique is used to speed up the retrieval of the records in databases. Normally, there are two types of indexes. The first type is called secondary access

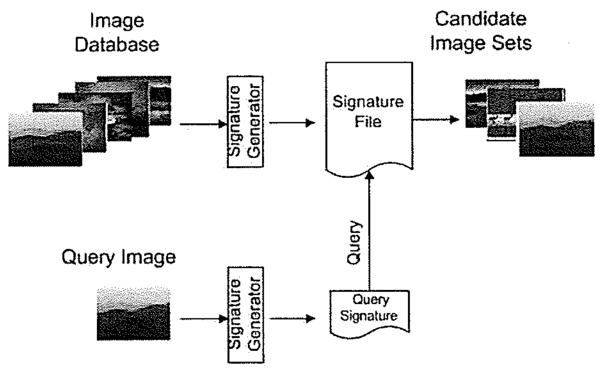


Figure 5.1: An image retrieval system

paths. That is, the indexes are separated from data files which contain the set of records, but hold one field of the data which links to the corresponding records. Based on that field, an index file can be sorted or ordered so that searching can be efficiently executed without touching the original data file. *DBMS* can create and delete an index. Examples of such indexes are single-level index, clustering, *B*-tree and so on. The second type of index integrates the index and data file together. In this thesis, the *IST* was implemented as the first type.

5.1.2 System Overview

Figure 5.1 provides a simple view of the system. A signature file is created to represent images in the database. When a user queries an image from the database, he/she must specify expected the features of the image, these features are transformed into a query signature and delivered to the system. The *IRS* searches the signature file for the query signature. The searching result is a collection of candidate images that the user desires.

Figure 5.2 shows the query interface for the system. This interface allows users to define the target image features, such as image categories, image objects, the relationships between objects, image format, the size of images, and so on. After defining all the parameters, the user can run the query by pressing the "Query" button.

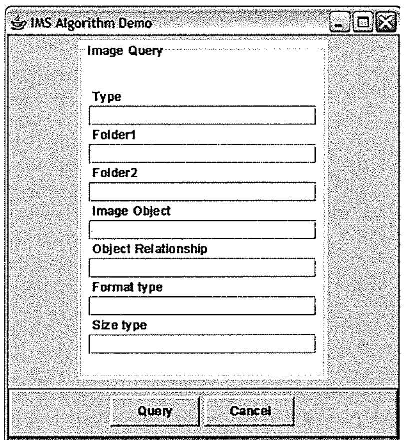


Figure 5.2: Query interface

5.1.3 Test Data

The test data is synthetic. With the proposed image model, an image can be transformed into a textual record with a special structure. In traditional document retrieval systems, the signature for the text is a bit string generated with a hashing method. Hashing methods can be applied to image databases. For the prototype test data, I generated a list of bit strings. Each bit string with a length l and w 1's corresponds to an image object. The image signature is achieved by superimposing all the object signatures that an image contains.

5.1.4 Working Platform

The working platform will be based on a Toshiba Satellite M35X-S161 notebook with a 1.30GHz Intel *Celeron* Processor, a 60GB disk and 512M bytes of memory. Microsoft *SQL* 2000 is selected as a database system. *JDBC* technology is utilized

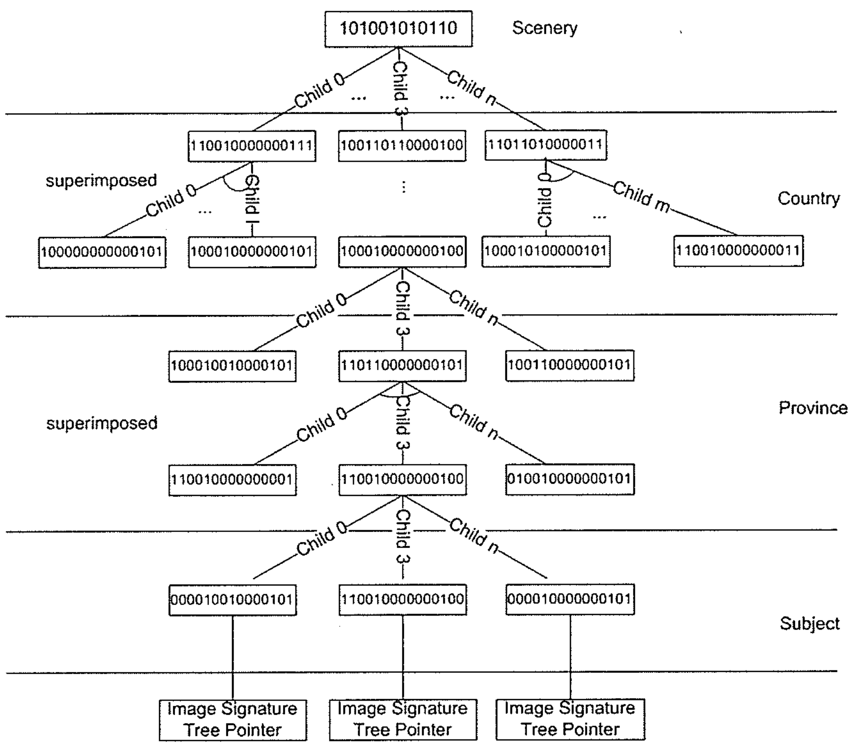


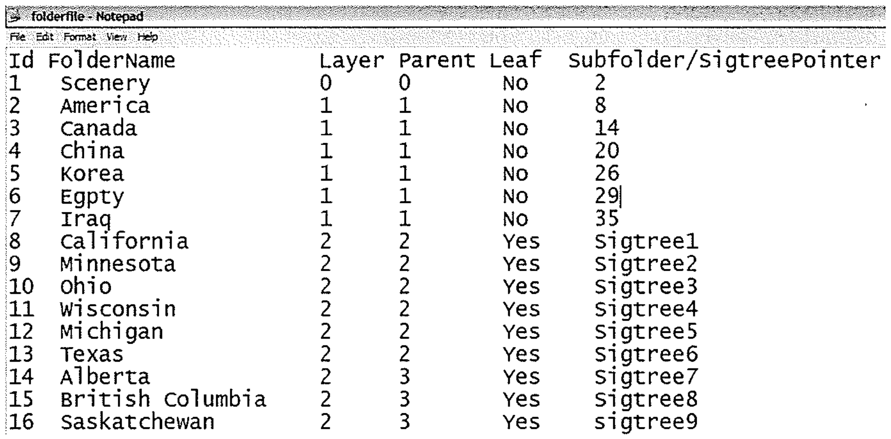
Figure 5.3: The folder tree

for Java programs to access the database.

5.1.5 Implementation of *IST*

This section summarizes how to implement each component of *IST*.

Figure 5.3 illustrates the structure of the folder tree. Retrieving the folder at each layer is equivalent to searching words in the document. Sometimes one folder contain many subfolders, if superimposing all the signatures for each subfolder without extending the signature length, the false drop probability will increase. In order to lower the false drop probability of signature file, this folder needs to be decomposed into groups so that each group contains small number of distinct words. As shown in Figure 5.3, the first subfolder is divided into n groups, the signatures of group member are superimposed together to form a new signature.



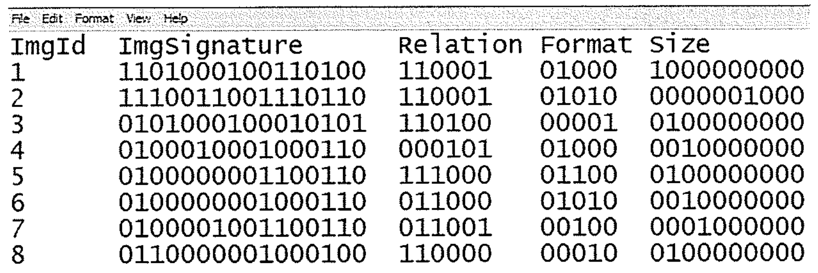
```
Id FolderName          Layer Parent Leaf  Subfolder/SigtreePointer
1   Scenery            0     0      No     2
2   America            1     1      No     8
3   Canada             1     1      No     14
4   China              1     1      No     20
5   Korea              1     1      No     26
6   Egpty              1     1      No     29
7   Iraq               1     1      No     35
8   California         2     2      Yes    Sigtree1
9   Minnesota          2     2      Yes    Sigtree2
10  Ohio               2     2      Yes    Sigtree3
11  Wisconsin          2     2      Yes    Sigtree4
12  Michigan           2     2      Yes    Sigtree5
13  Texas              2     2      Yes    Sigtree6
14  Alberta            2     3      Yes    Sigtree7
15  British Columbia   2     3      Yes    Sigtree8
16  Saskatchewan       2     3      Yes    sigtree9
```

Figure 5.4: The folder data file

Figure 5.4 shows the folder data file which is used for the construction of the folder tree. The folder data file includes folder name, subfolder, a signature tree pointer and so on. The *Id* column holds the identification number for each folder. The second column, *FolderName*, gives the category information by which images are classified to store in the disk. The *Layer* column identify the folder layer. *Parent* column identify the folder's parent. *Leaf* column indicates whether its folder points to a signature tree. The folder is a tree structure itself. The children of each folder is indicated by *Subfolder/SigatureTreePointer* column. This column also points to the corresponding signature tree. The given example only has three layers. Both layer 0 and layer 1 have subfolders, while layer 2 does not have. Instead, the layer 2 links to the signature tree.

Searching a folder works as follows: 1) when a querying folder comes, the program check the signature at the corresponding layer in the index file; 2) If the result is matched, the program can further check the folder data to find the exact matched folder; 3) If there is not exact matched result, the program stops. Otherwise, the program is prepared to further compare the signature at the next layer.

Figure 5.5 shows the structure of an image signature file. It consists of five



ImgId	ImgSignature	Relation	Format	Size
1	1101000100110100	110001	01000	1000000000
2	1110011001110110	110001	01010	0000001000
3	0101000100010101	110100	00001	0100000000
4	0100010001000110	000101	01000	0010000000
5	0100000001100110	111000	01100	0100000000
6	0100000001000110	011000	01010	0010000000
7	0100001001100110	011001	00100	0001000000
8	0110000001000100	110000	00010	0100000000

Figure 5.5: Image signature in a partition

columns. The first column, *Imgid*, indicates the identification number of images. The second column, *ImgSignature*, is object signature, which is obtained by superimposing the objects in an image. The *Relation* column stores the relation signature. For an image with k objects, object signatures are generated by a hash function, and these k object signatures are superimposed together to form a block signature, which is the object field of an image signature. k objects in an image have $k * (k - 1)$ relationships, which fall into six categories: DISJOINT,OVERLAP, MEET, CONTAIN, CROSSING and EQUAL. The relation signatures of an image are superimposed together to generate the relation field. Since there are only six spatial relationships defined in this research, I use a six-bit string to represent relation signatures. For example, I use "100000" to stand for the "DISJOINT" relation and "000010" to represent the "CONTAIN" relation. If all the object pairs in an image are belonged to the "DISJOINT" and the "CONTAIN" relationships, then the image relationship signature is "100010". The size filed and the format field are generated similar to the relation field. In this research, the variety of images formats, such as .bmp, .jpg, .jif,.tiff,.png, are constructed into a signature. I use one signature to stand for *.bmp* type, and different signature to represent another format. The format signatures for an image are superimposed together to form a signature. Figure 5.6 gives an example how to construct a size signature.

The *DBMS*s use so called schema to describe the database and the table layout.

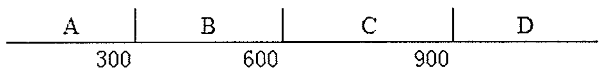


Image	length	width	string	signature
263 * 300	263 < 300	300=300	AA	10010000
350 * 650	300<350 <600	300< 650<900	BC	00010100
800 * 950	600<800 <900	950 >900	CD	00000101
1200 * 1100	1200 >900	1100 >900	DD	00010010

Figure 5.6: How to construct size signature

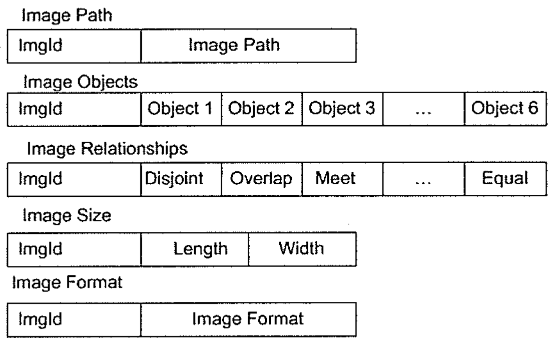


Figure 5.7: Schema diagram for the database

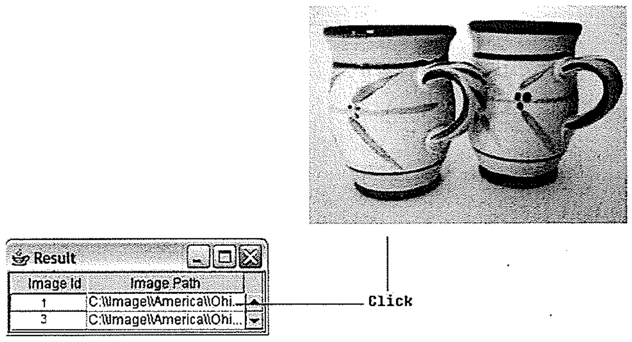


Image Id	Image Path
1	C:\\Image\\America\\Ohi...
3	C:\\Image\\America\\Ohi...

Figure 5.8: Report

The database schema provides the information for the specific tables within a database as well as the relationships between tables. It is specified during the database design and is not expected to change frequently. Figure 5.7 displays the schema diagram of this prototype database System. The signature indexing file is linked to real database records through *ImageId* field.

Figure 5.8 shows the example of a query result. The image can be displayed by clicking the table cell with a mouse.

5.2 Partitioned Signature Tree

Figure 5.9 displays the class diagram for the *PST*. Class *PartitionedSignatureTree*, *BalancedSignatureTree* and *SignatureTree* are three main components in *PST*. The roles and their relationships of these classes are defined as follows:

- PartitionSignatureTree. This class is responsible for dividing the signature file into partitions and creates a key tree, which hold the key information for all the partitions. It inherits from class *BinTree*, which is a tree structure for a collection of class objects, called *BTreeNode*. *PartitionSignatureTree* invokes *pEasyQueue* class when it saves the key tree by the level order into the disk.

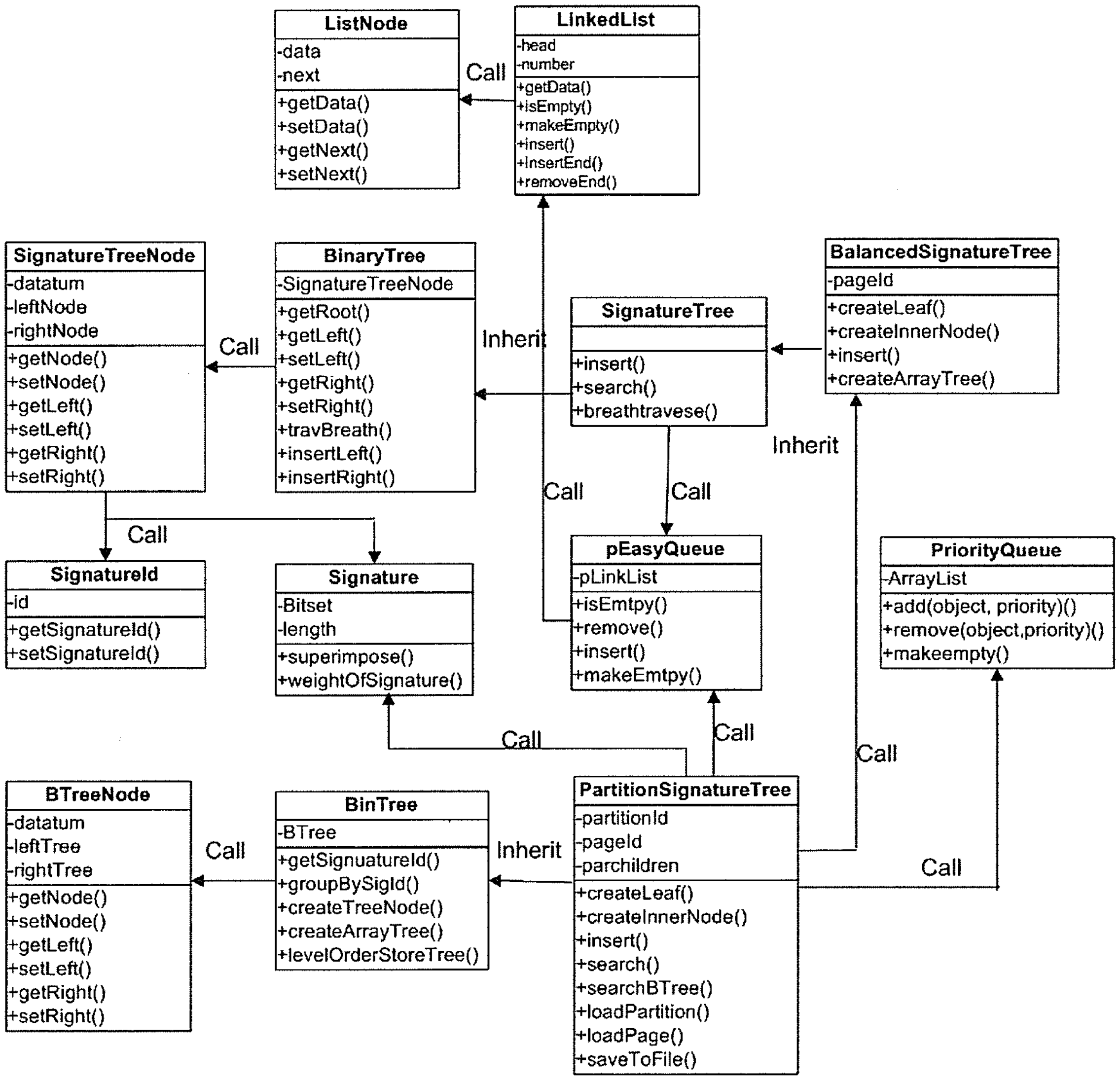


Figure 5.9: Class diagram for *PST*

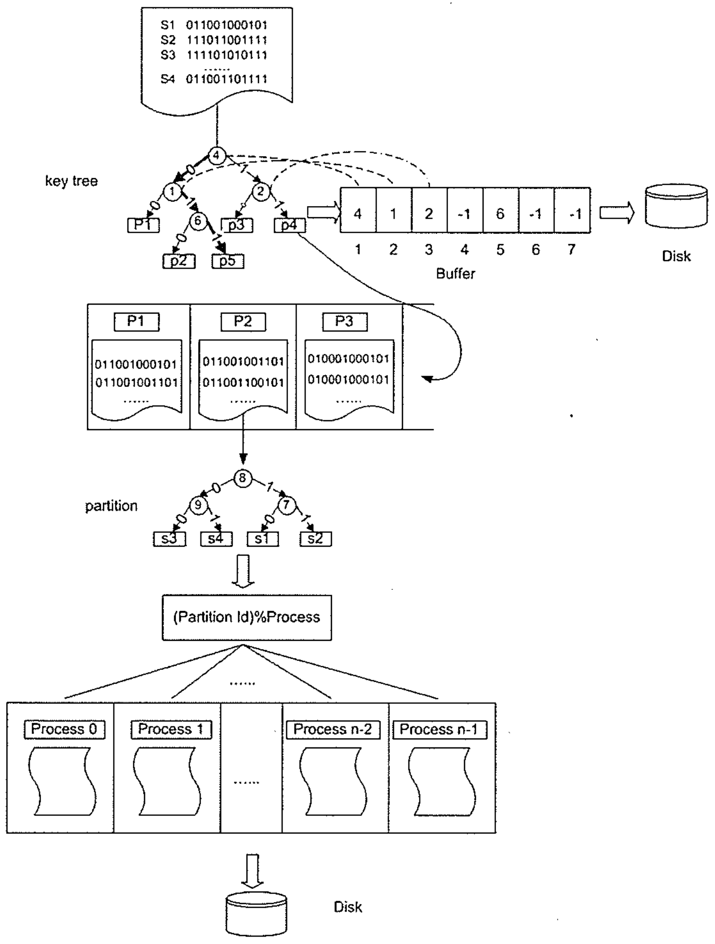


Figure 5.10: The flow chart for *PST*

After storing the key tree, it further construct the partitioned signatures into a tree by calling *BalancedSignatureTree*. When searching a query signature, it uses *Signature* class to create a query signature, and invokes *PriorityQueue* to preserve the ID information for the activated partitions.

- BalancedSignatureTree. This class constructs signatures into a balanced signature tree. This class inherits from class *SignatureTree*, with some methods overridden, such as *insert*,*createInnerNode*, etc. These overridden methods implement the weight-based method to ensure the generation of the balanced signature tree.

- SignatureTree. This class constructs signatures into a signature tree. Although *SignatureTree* and *PartitionSignatureTree* have some similar methods, such as search, insertion, deletion, etc, the inherit operation is quite different. The internal node of the *Keytree* in the *PartitionSignatureTree* is generating by a splitting process as the *BST* does, while the internal node of *SignatureTree* is generated by comparing the signature difference between two signatures. Therefore, class *SignatureTree* inherited from class *BinaryTree* which has a different tree node structure with *BinTree*. *SignatureTree* also invokes *pEasyQueue* when it traverses tree by level order.

Figure 5.10 shows the procedure of constructing the *PST*. First, the signature file is scanned and divided into a number of partitions. These partitions are inserted into an *Arraylist* for further constructing into a balanced signature tree. At the same time, a key tree is created and the data values of tree nodes are buffered into an array by the level order. This array is further stored into disk. By doing this, a key tree is successfully stored into the disk for further retrieval. Figure 5.11 shows the algorithm for storing tree to an array. Next, the partitions stored in the *Arraylist* are constructed into balanced signature trees. Finally, these balanced signature trees

```
/* Purpose: store a balanced tree into a buffer
   Input parameter:  the root of the tree, array
   Output: void */
public void levelOrderStoreTree(BTree root,int[] b)
{
 pEasyQueue q = new pEasyQueue();
 BTree tmp;
 Int i =0;
 q.insert(root);
  if(root != null) {
   if( i <= b.size) {
    while(!q.isEmpty())
    {
      tmp = (BTree)(q.remove());
      if(tmp.getLeft() != null) q.insert(tmp.getLeft());
      if(tmp.getRight() != null) q.insert(tmp.getRight());
        b[i] = tmp.getNode();
        i++;
    }
   }
  }
}
```

Figure 5.11: Store the tree to an array

are saved into the disk. If the algorithm is implemented with multi-threads, a set of files should be created for storing partitions to avoid potential file accessing conflictions during the partition retrieval. The number of files should be the same as that of threads. Then, the partitions are distributed into those files according to their partition identification numbers. For example, assuming there are 4 threads executing computation. In this case, 4 files are generated to store the partitions and one file is generated to store the key partition. On the other hand, if the algorithm is implemented in shared-memory parallel environment, only one file is needed.
Figure 5.12 illustrates a searching procedure for the *PST*. When a query signature comes, the *PST* loads the key tree from the disk. Figure 5.13 shows the algorithm for loading the array for key tree from the disk, and Figure 5.14 shows the algorithm for generating the key tree from an array. Starting from the root of the key tree, the query is executed and result is a set of I.D. numbers for the activated partitions. These I.D. numbers are pushed into a priority queue. In a parallel environment, the number of queues is the same as that of processes. When a query comes, the different

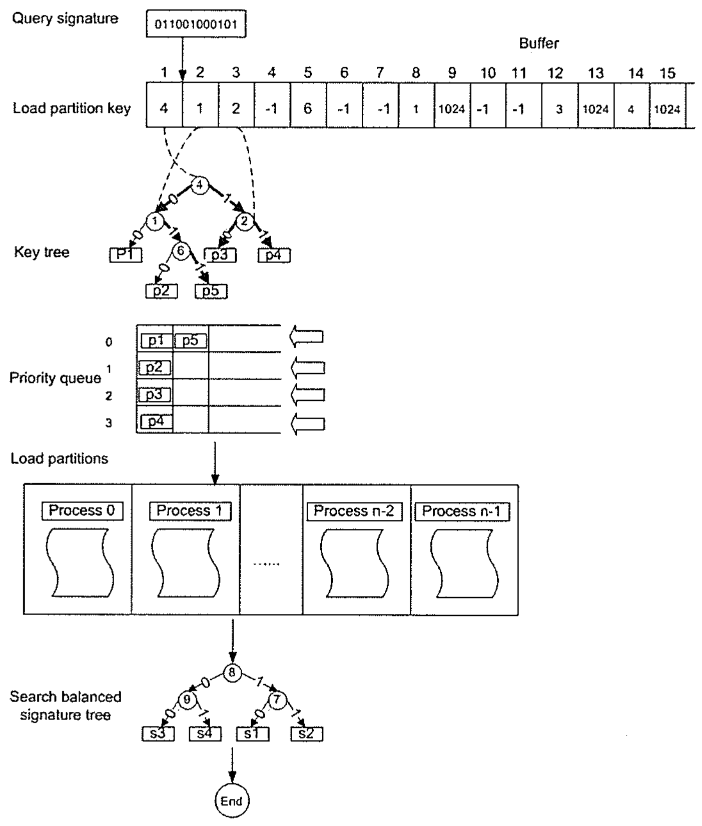


Figure 5.12: Search procedure

```
/*  Purpose: load a key tree from the disk
    Input parameter:  the filename for storing the key tree, Array
    Output: void
*/
     static void loadPartition(String filename, int[] readBuf)
     {
       File fIn;
       RandomAccessFile raf;
       int length;
       int offset = 0;  // key tree's offset in the file.
       Try {
           fIn = new File(filename);
           raf = new RandomAccessFile(fIn,"r");
           System.out.println("\nRestoring Partition signature tree...");
           raf.seek(ofset);
           for ( int i=0; i<length; i++) {
                  readBuf[i] = raf.readInt();
           }
           raf.close();
           System.out.println("\n Page restored successfully.");
       }
       catch(Exception e) {
         e.printStackTrace( );
       }
      }
```

Figure 5.13: Load an array from disk

```
/* Purpose: create a balanced tree from an array
   Input parameter:  array, array index, array size
   Output: Tree
*/
public BTree CreateArrayTree(int[] a, int i, int size) {
   if (i >= size)
      return null;
  else
      try {
         return (
            CreateTreeNode(a[i],
                   CreateArrayTree(a, 2 * i + 1, size),
                   CreateArrayTree(a, 2 * i + 2, size)));
    }
    catch (Exception ex) {
      return null;
    }
 }

 public BTree CreateTreeNode(int d1,BTree p1, BTree p2) {
   BTree t;
   t = new BTree(d1,p1,p2);
   t.setLeft(p1);
   t.setRight(p2);
   return t;
 }
```

Figure 5.14: Generate the key tree from an array

processes can retrieve the activated partitions from the queues and its corresponding files concurrently.

Chapter 6

Experimental Results

In order to test scalability of balanced signature tree algorithm as well as consistency with the theoretical analysis in chapter 4, this chapter will present the simulation results followed by the evaluation of performance.

6.1 Working Platform

The working platform will be based on a Toshiba Satellite M35X-S161 notebook with a 1.30GHz Intel *Celeron* Processor, a 60GB disk and 512M bytes of memory.

6.2 Test Data

We have written a program to generate several signature files as the test data, with each containing 10000, 40000, 60000, 80000, and 100000 signatures respectively. Generating a signature file includes the following two steps: First, construct a signature by setting a binary string with length of l and weight of w. Second, superimpose n signatures to form a block signature, and then store these block signatures into a file.

Signature file 32 *bits*	*BST*		*PST*	
	Experimental Value	Theoretical Value	Experimental Value	Theoretical Value
10000	8K	5.998K	9K	5.998K
40000	32K	23.998K	33K	24.004K
60000	48K	35.998K	49K	36.022K
80000	64K	47.998K	65K	48.030K

Table 6.1: Comparison of storage overheads of *BST* against *PST*

6.3 Storage Overheads

Signature tree uses position nodes to specify its search path. These position nodes are additional data structures to the signature file. Therefore, signature tree requires additional disk space for these position nodes. Table 6.1 shows the storage overhead for the *BST* and the *PST* with signatures of 32 bits in length. As can be seen, the partitioned signature tree occupies more space than the balanced signature tree. The reason is that the key table requires an extra space. The simulation value is 33% larger than the theoretical value. This ratio is constant as the signature file increases.

6.4 Signature Reduction

Signature Reduction performance is conducted with *SSJ* code (Stochastic Simulation in Java). A query process will be repeated 100 times. For each process, a signature will be generated randomly , and the number of drops (including true drops and false drops) will be collected. Finally, the drops will be obtained by averaging the sum of drops in each process. The signature reduction rate is examined in two directions.

Weight	Signature reduction ratio	
	Theoretical	Simulation
10	0.9765E-05	0.0063
16	1.52588E-05	0.0013
20	9.53674E-07	0.0005
24	5.96046E-08	0.0003
32	2.32803E-10	0.0001

Table 6.2: Signature reduction ratio against weight

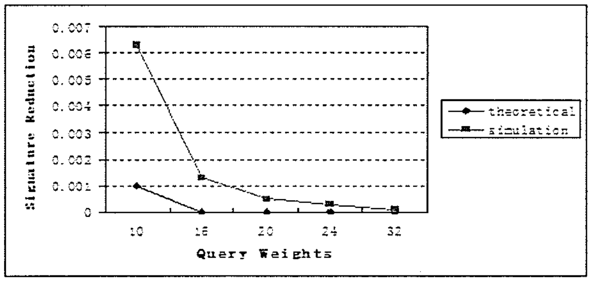


Figure 6.1: Signature reduction of the *PST* against weight

First, I perform the test by changing query weights. Second, I investigate the performance on different signature file sizes. The analysis in chapter 4 indicates that *BST* and *PST* have the same reduction rate, I only list the test results for *PST*.

6.4.1 Signature Reduction with Weight

Table 6.2 gives the results of the signature reduction ratio against the query signature weight for *PST*. Figure 6.1 shows that the signature reduction ratio decreases as query weight increases. The simulation results also agree with the theoretical value. This indicates that query weight has a big impact on signature reduction rate.

Weight	Signature Reduction Ratio			
	Theoretical	Simulation 1	Simulation 2	Simulation 3
10	0.9765E-05	0.0063	0.0052	0.004
16	1.52588E-05	0.0013	0.0009	0.0007
20	9.53674E-07	0.0005	0.0005	0.000225
24	5.96046E-08	0.0003	0.0005	0.000125
32	2.32803E-10	0.0001	0.0005	0.00005

Table 6.3: Signature reduction ratio against the size

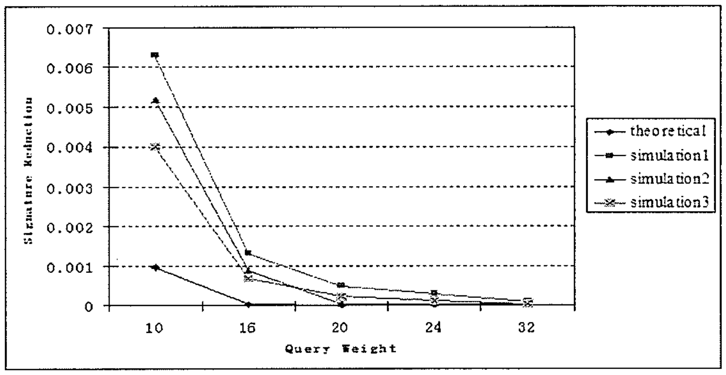


Figure 6.2: Signature reduction ratio of the *PST*

6.4.2 Signature Reduction with Data Size

Table 6.3 shows how the signature reduction ratio varies with different data sizes. As shown in Figure 6.2, as signature file size increases, the signature reduction ratio decreases. The larger the file size is, the closer to the theoretical value. This is due to the fact that statistical variation decreases as the file size increases.

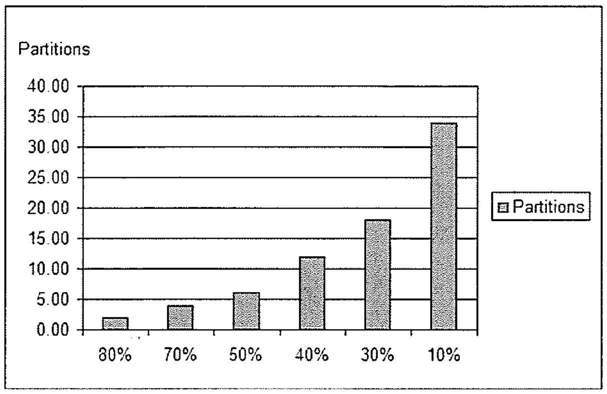


Figure 6.3: Partition access against query weight

6.5 Partition Access against Query Weight

Query weight affects partition accesses. Figure 6.3 shows that the number of partitions accessed when 80000 signatures are divided into 80 partitions with each containing 1000 signatures. In this chart, the X axis stands for the ratio of query weight to the signature length, while Y axis represents partition accesses. As can be seen, the number of partition accesses increases as the ratio of query weight to the signature decreases. This result is in accordance with the analytical prediction for a *PST*.

6.6 Retrieval Cost

For the small size *BST* and *PST*, the later does not show much advantage over the former. The retrieval cost of the balanced signature tree is less than the partitioned signature tree. This is due to I/O cost for the partitioned signature tree. According to my test, an operation to load a partition from disk usually takes 290 $\sim$ 320 *msc*.

So, the test was conducted in a multitask environment to determine if *PST* shows more advantages in these circumstances. Initially, the experiments were carried out using Java threads. Table 6.4 provides a set of results of the retrieval cost for the

Signature file size	Execution time (msc)		
	Threads 1	Threads 2	Threads 3
10000	391	383	425
20000	1026	624	1290
40000	1722	879	1940
60000	2273	1465	2690
80000	3077	1917	3993

Table 6.4: Retrieval cost

partitioned signature tree which is plotted as a graph in Figure 6.4. In these tests, the partition size was set to 10000 signatures. As can be seen, the multithread program yields fairly good performance when the thread number is 2. However, when the thread number is over three, the search time is longer than that of the one thread program. The reason for this is that the Java multithread is not true parallel processing. Java multithreaded mechanism is designed for efficiently utilizing CPU and other resources within a process by keeping the processes to run continuously [16]. By assigning a time slot for each thread and swapping the active thread during the execution, *JVM* makes the execution of several threads appear to be simultaneous. There is only one thread accessing one physical processor at one moment in Java. The true parallel processing requires two or more physical processors.

Since current Java program for a single processor can not run on different physical processors straightforwardly. I implemented the partitioned signature tree with *Java OpenMP* on the multiprocess machine. The parallel machine I used is an *IBM Netfinity* 8500 at the University of Manitoba. This computer has a shared-memory processor (*SMP*) system with 8 Intel Xeon 700 Mhz processors and 7.5 GB of memory. Figure 6.5 shows the retrieval time against different processes.

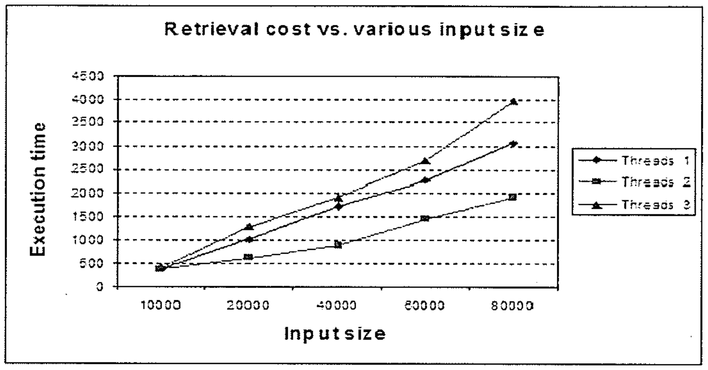


Figure 6.4: Retrieval time

Number of Processes	The Number of Processes				
	10	2	4	6	8
10000	396	353	326	318	316
20000	1013	665	632	625	618
40000	1625	976	637	625	618
60000	2239	980	940	624	622
80000	2859	1588	940	632	625

Table 6.5: Retrieve time of the *PST*

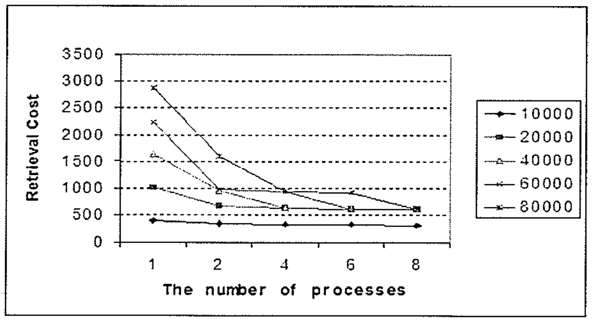


Figure 6.5: Retrieval time vs. processes

The Number of Insertion	Insertion Time(msc)				
	1000	1500	2000	2500	4000
Signature file 10000	416	453	489	609	1154
Signature file 20000	1008	1040	1101	1226	1792
Signature file 40000	1622	1679	1710	2106	2373
Signature file 60000	2267	2264	2400	2770	2925
Signature file 80000	2705	2917	3248	3438	3623

Table 6.6: Insertion time of *PST*

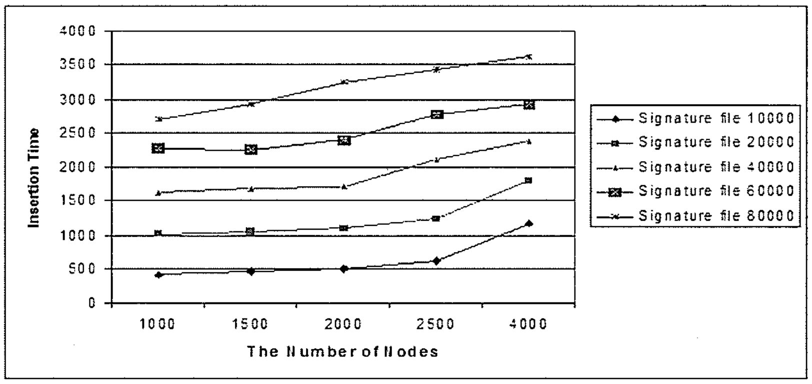


Figure 6.6: Insertion time of *PST*

6.7 Update Cost

6.7.1 Insertion Cost

Table 6.6 presents the test results for insertion cost which is plotted as graph in Figure 6.6. As can be seen, the insertion cost increases as the signature file increases.

When insertion causes a partition to be oversized, a new partition should be generated to accommodate new signatures. The worst case for scheme 1 in chapter 3 is when new signature is about to be inserted into a partition that is at its maxi-

Partition size	Partition Efficiency	
	Scheme 1	Scheme 2
256	0.004	0.45
512	0.002	0.47
1024	0.001	0.48

Table 6.7: Usage of partitions

The Number of Insertion	Deletion Time(msc)				
	1000	1500	2000	2500	4000
Signature file 10000	410	424	446	448	451
Signature file 20000	733	1038	1059	1088	1074
Signature file 40000	1588	1645	1688	1711	1714
Signature file 60000	2144	2239	2244	2144	2340
Signature file 80000	2704	2753	2880	2885	2912

Table 6.8: Deletion time of *PST*

mum size, an internal node is generated to identify the different bit between the new inserted signature and one existing signature with the partition. The splitting partition occurred on this internal node. Its left and right children should go to two new partitions. The usage of two partitions is rather low being one child signature per partition. The usage of this partition is rather low 1/partition size. This situation will never occur with the scheme 2 because it splits partitions from the root and Table 6.7 gives the comparison.

6.7.2 Deletion Cost

Table 6.8 gives the test results for the deletion cost which is plotted in Figure 6.7.

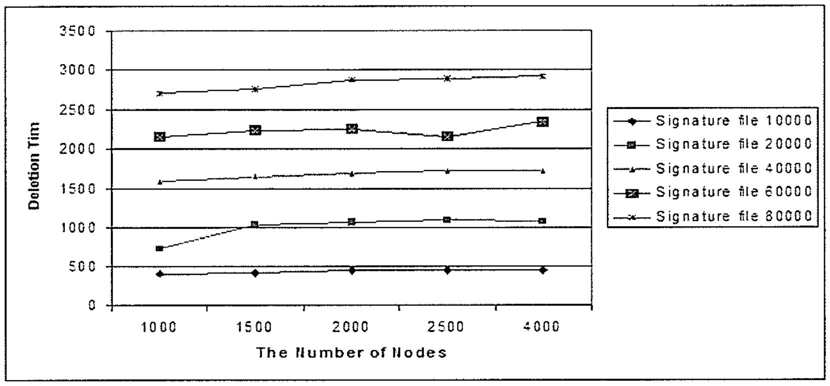


Figure 6.7: Deletion time

As can be seen, the deletion cost increases as the signature file increases. Compared with the insertion cost, the deletion cost is much lower. This is because generating the new internal nodes using the weight method costs more time.

Chapter 7

Discussion and Conclusions

This chapter presents a summary of in this thesis, and provides suggestions for future work in this area.

7.1 Summary of Contributions

In this research, we presented an image indexing algorithm for large image databases. This indexing algorithm uses the signature file technique that eliminates unnecessary searches to improve retrieval speed. The motivation for this algorithm is to overcome the limitation of the current research on image signature file methods that suffer some serious problems, such as not utilizing the filtering feature of signature file, lacking of experimental results or neglecting some import image features. We analyzed the performance of the major components of this algorithm and evaluated it by experiments. The primary contributions of this research are summarized as below. First, this thesis introduced an image model that specifies the image features by extending the *RSS* model. Previous image signature algorithms only considered image objects and their spatial relationships. This model improves on this by providing image storage path information, image pixel size as well as image format. With this model, an image is transformed into a textual record with a special structure, which can be efficiently handled using signature files and signature trees. Secondly, this thesis presented the Image Signature Tree algorithm that filters images level by level. Some previous research on image signature algorithm suffered from either lacks of filtering ability or did not use advanced signature file technique. This approach is based on a balanced signature tree algorithm that exhibits high performance. An image retrieval system is developed to demonstrate the feasibility of IST algorithm. Thirdly, a new partition scheme is proposed to divide the balanced signature tree into partitions where each partition holds the same key. The motivation for this scheme is to separate the signature tree into smaller ones in the case of large number of signatures. Different from other partitioning schemes, this scheme uses a signature tree structure as the key. This saves search space and mitigates the false drop probability significantly. It also supports the parallel programming. The significant feature of this scheme is that it is able to provide a good workload balance and partition usage. Finally, this research provides an in-depth analysis for signature reduction ratio for the balanced signature tree. Experiments show that simulation results are close to the theoretical estimation.

7.2 Future Work

The following describes the possible extensions to this thesis:

1. We would like to implement more signature algorithms. Through comparing the performance with various algorithms, we can better understand the advantages and weakness of the signature tree algorithm, the balanced signature tree algorithm as well as the partitioned signature tree algorithm.

2. This thesis presented a prototype system with *IST* method. However, the lack of a large amount of test data for *IST* algorithm is the weakness of this research. Also, this thesis has not provided a sufficient analysis for the folder tree, which should be conducted to prove the efficiency of *IST* method.

3. The image model we designed has not captured all image features. More image attributes, such as color, shape, etc, can be added to the image model to provide more accurate description of images. If so, more query condition can be defined when performing image searches. As query criteria increase, the accuracy of expected image will increase.

4. This research assumes image objects can be identified within images. In reality, extracting image objects from images is rather challenging. In the future, automatically abstracting objects from an image will be a significant research topic in image retrieval.

Appendix A

An Example for Analyzing a Balanced Signature Tree

Figure A.1 illustrates a balanced signature tree generated from 16 signatures. The signature file contains the equal number of "0" bits and "1" bits. That is, there are 32 "0" bits and 32 "1" bits in the file. The balanced signature tree consists of 15 internal nodes and 16 leaves which stand for 16 signatures.

When a query signature comes, the search is performed and the results can be classified into five categories according to its weight:

Case 1 ($w_q = 4$): the query signature is "1111". Starting from the root of the balanced signature tree, the first bit is checked. After that, it goes to the right subtree

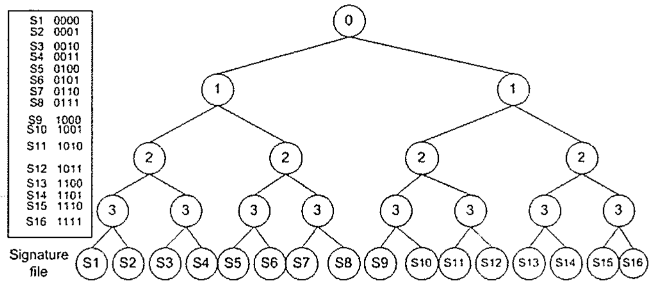


Figure A.1: A Balanced Signature Tree

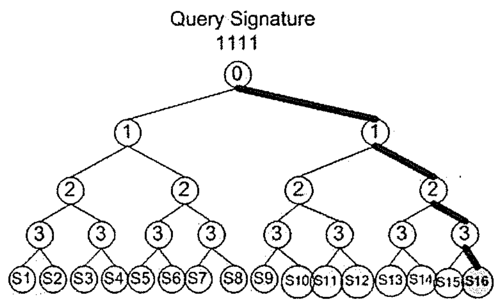


Figure A.2: Case 1: Query Signature Weight is 4

to check the second bit. The highlighted edges in figure A.2 show the search path. The resulting signature is the shadowed leaf node ($S16$) in figure A.2.

Case 2 ($w_q = 3$): the query signature has four possible cases, which are 0111,1011,1101,1110. The search processes are shown in figure A.3(a),(b),(c) and (d) respectively. No matter what case is, each of them has two signature drops.

Case 3 ($w_q = 2$): the query signature has six possible combinations, which are 0011,0101,0110,1001,1010,1100. Figure A.4(a),(b),(c),(d),(e) and (f) illustrate the search path for each case respectively. All of the cases share one common feature, that is, every case results four signature drops.

Case 4 ($w_q = 1$): the query signature has four possible cases, which are 0001,0010,0100,1000, as shown in figure A.5. All of the cases have eight signature drops.

Case 5 ($w_q = 0$): the query signature is 0000, and the result contains sixteen signature drops, as shown in figure A.6.

The above illustration shows that the average number of signature drops depends on the weight of query signature, rather than the position of "1"s in the query signature. It occurs in the space of events [1] with certain discipline. Table A.1 shows the relationships between signature weight, the number of drops and drops occurrence probability.

According to the table A.1, let X be the random variable of signature drops. The

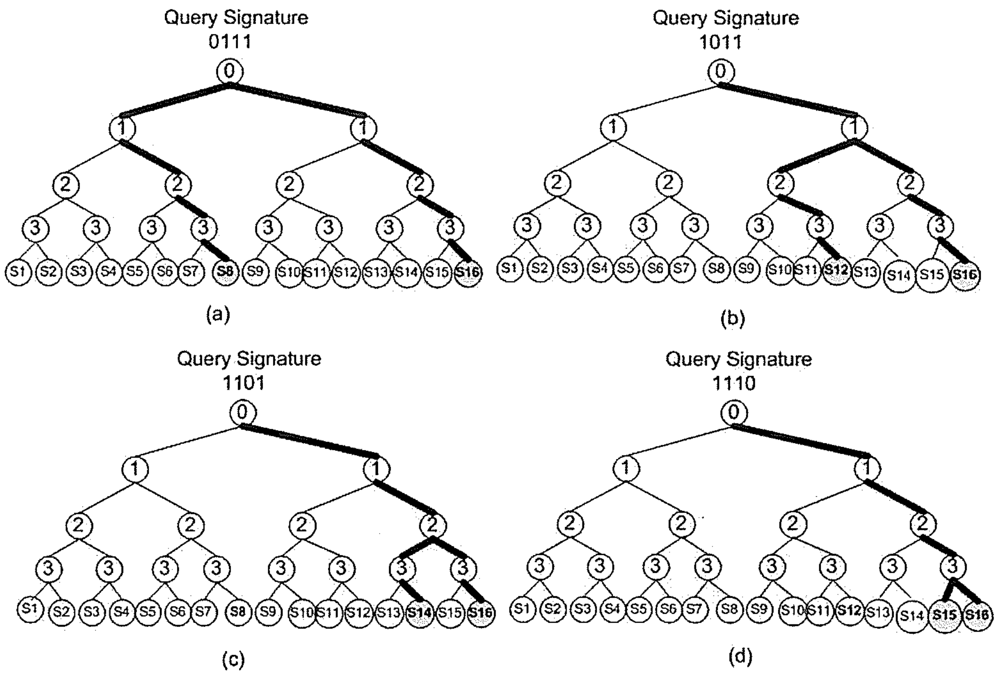


Figure A.3: Case 2: Query Signature Weight is 3

Weight i	**Signature Drops** $X = 0.5^i * sample\ space(2^m)$	**Numbers of Signature with Same Weights** $\binom{m}{i}$	**Occurrence Probability**$\binom{m}{i}/2^m$
i=4	$16 * (1/2)^4 = 1$	1	1/16
i=3	$16 * (1/2)^3 = 2$	4	4/16
i=2	$16 * (1/2)^2 = 4$	6	6/16
i=1	$16 * (1/2)^1 = 8$	4	4/16
i=0	$16 * (1/2)^0 = 16$	1	1/16

Table A.1: Signature Drops against Weight

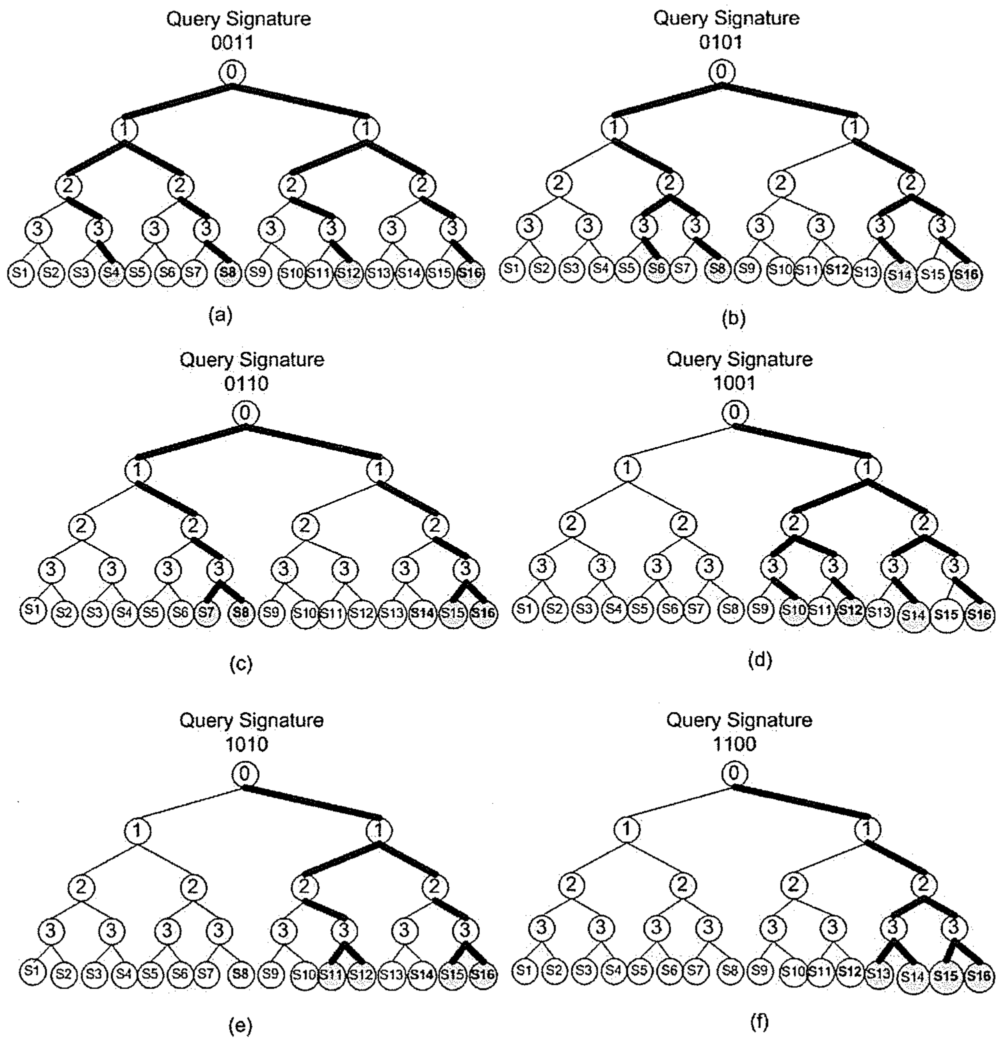


Figure A.4: Case 3: Query Signature Weight is 2

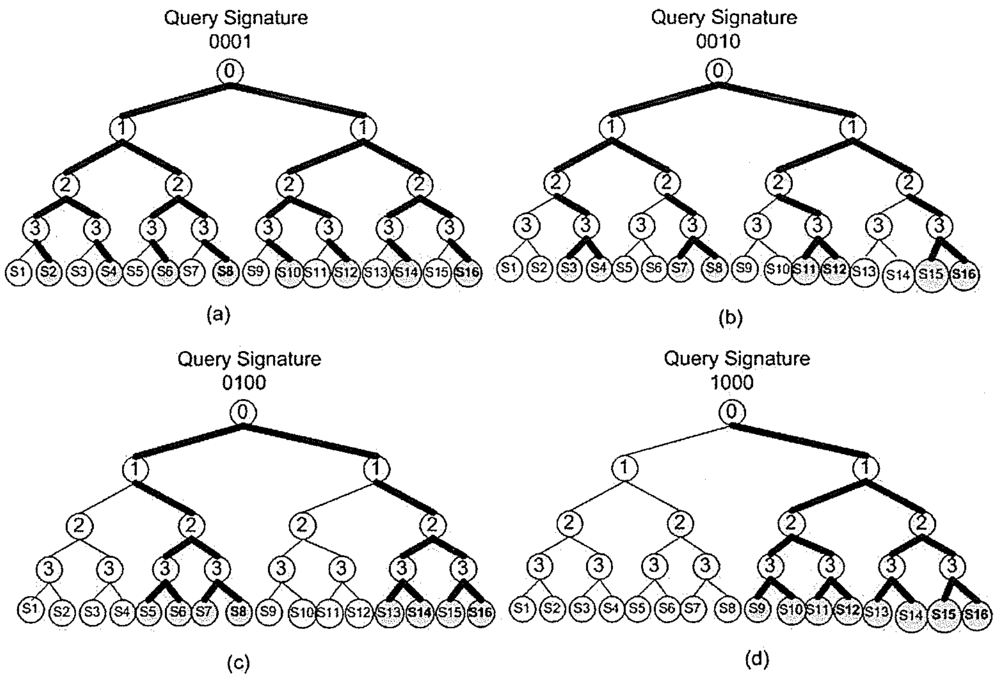


Figure A.5: Case 4: Query Signature Weight is 1

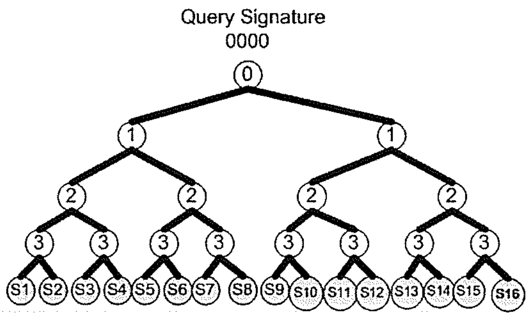


Figure A.6: Case 5: Query Signature Weight is 0

average number of signature drops (ASD), is calculated as follows:

$$ASD = \sum_{i=0}^{4} X_i P_i = 5.0625 \tag{A.1}$$

Compared to the sequential signature file, the signature drops are reduced from 16 to 5.0625.

Bibliography

[1] A. O. Allen. *Probability, Statistics and Queuing Theory with Computer Science Applications.* Academic Press, New York, USA, 1990.

[2] J. L. Bently. Multidimensional binary search in database applications. *IEEE transactions on software engineering*, 4(5):333–340, 1979.

[3] A. Bertrand. Should I store images in the database or the file system. http://databases.aspfaq.com/database/should-i-store-images-in-the-database-or-the-filesystem.html, 2006.

[4] Y. Chen. Signature files and signature trees. *Information Processing Letters*, 82(4):213–221, 2002.

[5] Y. Chen. On the signature trees and balanced signature trees. In *Proceedings of the 21st International Conference on Data Engineering*, pages 742–753, Tokyo, Japan, April 2005.

[6] T. M. Connolly and C. E. Begg. *Database Systems.* Addison Wesley, New York, USA, 2002.

[7] Microsoft Corporation. Store images in a database. http://office.microsoft.com/en-us/assistance/HP052802251033.aspx, 2006.

[8] J. P. Eakins. Retrieval of still images by content. In *Lectures on information retrieval*, pages 111–138, New York, USA, September 2001.

[9] E. A. El-Kwae and M. R. Kabuka. Efficient content-based indexing of large image databases. *ACM Transactions on Information Systems*, 18(2):171–210, 2000.

[10] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addision-Wesley, California, USA, 1994.

[11] C. Faloutsos and S. Christodoulakis. Description and performance analysis of signature file methods for office filing. *ACM Transactions on Office Information Systems*, 5(3):237–257, 1987.

[12] T. Gevers and A. W. M. Smeulders. Image search engines an overview. http://staff.science.uva.nl/ gevers/pub/overview.pdf, 2003.

[13] F. Grandi, P. Tiberio, and P. Zezula. Frame-sliced partitioned parallel signature files. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 286–297, Copenhagen,Denmark, June 1992.

[14] C. J. Guarin. Access by content of documents in an office information system. In *SIGIR '88: Proceedings of the 11th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 629–644, New York, USA, May 1988.

[15] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the SIGMOD Conference*, pages 47–57, Boston, USA, June 1984.

[16] I. Horton. *Beginning Java 2*. Wrox Press Ltd, Birmingham, UK, 2002.

[17] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *1997 ACM SIGMOD International Conference on Management of Data*, pages 13–15, New York, USA, June 1997.

[18] M. L. Kherfi, D. Ziou, and A. Bernardi. Image retrieval from the world wide web: Issues, techniques, and systems. *ACM Computing Surveys*, 36(1):35–67, 2004.

[19] J. K. Kim and J. W. Chang. A new parallel signature file method for efficient information retrieval. In *CIKM '95, Proceedings of the fourth International Conference on Information and Knowledge Management*, pages 66–73, Baltimore, USA, November 1995.

[20] C. H. Lee and P. W. Huang. Image indexing and similarity retrieval based on key objects. In *ICME*, pages 819–822, Taipei, Taiwan, June 2004.

[21] D. L. Lee, Y. M. Kim, and G. Patel. Efficient signature file methods for text retrieval. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):423–435, 1995.

[22] D. L. Lee and C. W. Leng. Partitioned signature files: Design issues and performance evaluation. *ACM Transaction on Information Systems*, 7:158–180, 1989.

[23] Z. Lin and C. Faloutsos. Frame-sliced signature files. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):281–289, 1992.

[24] M. A. Nascimento and V. Chitkara. Color-based image retrieval using binary signatures. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 687–692, Madrid, Spain, March 2002.

[25] C. S. Roberts. Partial match retrieval via the method of the superimposed codes. *Proc. IEEE*, 67(12):1624–1642, 1979.

[26] R. Sacks-Davis, Member, IEEE, A. Kent, K. Ramamohanarao, J. Thom, and J. Zobel. Atlas: A nested relational database system for text applications. *IEEE Transactions on Knowledge and data engineering*, 7(3):454–470, 1995.

[27] J. R. Shin, C. B. Son, J. S. Yoo, and B. M. Im. A dynamic signature file declustering method based on the signature difference. *International Journal of Information Technology*, 8(1):5, 2002.

[28] E. Vicario. *Image Description and Retrieval.* Plenum Publishing Corporation, New York, USA, 1998.

[29] A. M. Zaremski and J. M. Wing. Signature matching: A tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.

[30] P. Zezula. Linear hashing for signature files. In *the IFIP TC6 and TC8 International Symposium on Network Information Processing Systems*, pages 192–196, Sofia, Bulgaria, May 1998.

[31] P. Zezula, P. Ciaccia, and P. Tiberio. Hamming filters: A dynamic signature file organization for parallel stores. In *19th International Conference on Very Large Data Bases*, pages 314–327, Dublin, Ireland, August 1993.

[32] P. Zezula, P. Ciaccia, and P. Tiberio. Key-based partitioned bit-sliced signature file. *SIGIR Forum*, 29(2):20–34, 1995.

[33] P. Zezula and F. Rabitti. Dynamic partitioning of signature files. *ACM Transactions on Information Systems*, 9(4):336–369, 1991.