EFFICIENT MULTIMEDIA RETRIEVAL USING CUSTOM INDEXING METHODS

By

Malcolm W. Steenburgh

B.Sc. Honours, Queen's University at Kingston, 1996

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTERS OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

1998

© Malcolm W. Steenburgh, 1998

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
Vancouver, Canada

Date DEC 21ST/98.

DE-6 (2/88)

Abstract

One of the largest problems associated with content-based indexing of multi-media documents is the inefficiencies associated with the process. These inefficiencies are witnessed in a number of areas throughout the process, manifesting themselves in the form of high network bandwidth and processor requirements. The following piece of work examines two novel concepts which attempt to minimize these inefficiencies through the reduction in both bandwidth and processor requirements.

Included in this work is a discussion related to the reduction of network bandwidth requirements. The discussion focuses around the network bandwidth currently associated with the development of *indexing schemes*. In order for developers to implement and test *indexing schemes*, they currently have to download and index all of the media to which the new scheme will be applied. In addition to this being time consuming and work intensive, the entire process results in large amounts of network traffic. The proposed solution to this problem is a software interface allowing developers to submit unique *indexing schemes* to remote servers for processing. Upon receipt, the remote server applies the scheme to an arbitrary subset of its indexed media, restricted only by a set of time constraints set by either the developer or server administrator. Within the time constraints, the server will apply the given scheme and return the best results to the user. The result is efficiencies that are orders of magnitude greater than possible through current methodology.

The second concept discussed in the work focusses on the efficient application of known *indexing schemes* in query by example format queries. The algorithm makes use of the triangle inequality to reduce the number of media objects that actually have to be compared during query processing.

In addition to a disussion of the aforementioned concepts, this work includes an examination of an actual implementation of the concepts using the Java programming language. The application shows the viability of both concepts, showing that they indeed reduce the inefficiency currently associated with multi-media indexing systems.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Originally a medium for communication, the Internet has evolved into an extremely large, unsupervised and relatively unstructured multimedia database[1]. Early in this evolution it was realized that in order for this data to be useful and accessible, variations on classic database management system (DBMS) techniques would have to be implemented. In order for these implementations to be successful, they would have to meet two very important requirements - they would have to be both effective and efficient. With very little bandwidth, processing power and time, a user would have to be able to locate data relevant to their requirements.

This realization resulted in several online solutions. These solutions manifested themselves in one of two flavours - indices and search engines.

Indices, such as Yahoo[1], originally attempted to index the Internet by manually or semi-automatically classifying all documents into various categories.

Users are presented with an interface similar to that shown in Figure 1.1 and are expected to manoeuvre their way around a tree-like structure in search of content related to their search[2]. This method presents many problems including the classification of sites based on a single perception and the huge amounts of manual labour associated with doing so.

From the indices was born a second solution; the alpha-numeric search engine[3]. This solution is more true to classic DBMS techniques. Search engines automatically crawl the Internet, accessing and indexing all relevant documents. The result is an index which allows users to post alpha-numeric queries, receiving results in the form of pointers to relevant documents.

[1] Yahoo can be accessed on the Internet at `http://www.yahoo.com`

[2] It should be noticed that the current implementation of the Yahoo interface has been supplemented with the ability to post alpha-numeric queries to the index.

[3] For an example of an alpha-numeric search engine, see `www.altavista.digital.com`.

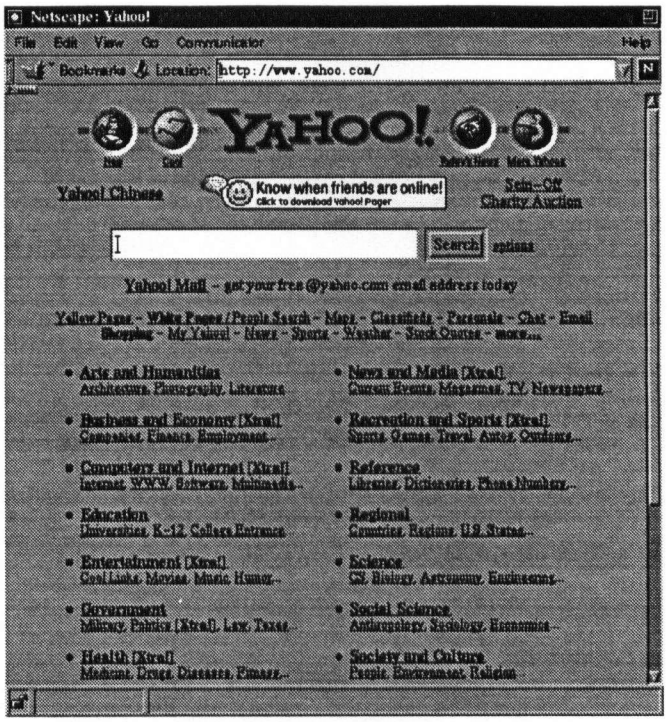

Figure 1.1: The *Yahoo* user interface.

Currently, this paradigm arguably provides the most effective method for accessing alpha-numeric data on the Internet.

Although alpha-numeric search engines provide an adequate means of searching for text based media documents, demand is moving beyond text to richer mediums. Increases in storage capabilities, processing power and network bandwidth have resulted in greater demand for and provision of non-text media objects such as images, audio samples and video clips. Most commercial Internet search engines claim to support various types of media objects[4]. The problem is that this support is strictly alpha-numeric. In most of these cases, the fitness or quality of a media object is based on superficial data such as the text that surrounds it in an HTML document. Although this may provide sufficient results in some applications, it is not reasonable to expect to be able to capture all perceptions of a multimedia object using text alone. Beyond this, it is even more unreasonable to expect web designers, amateur and

[4]For our purposes, a media object is any file containing user observable information including text documents, images, audio samples and video clips

professional alike, to capture all perceptions of the object in the content that they provide.

This deficiency is not sufficient for the requirements of tomorrow's user. In order to provide users with access to non-text media, on par with the alpha-numeric access with which they have become accustomed, new content-based methods for indexing media objects must be developed. The following work examines some of the major obstacles that have to be overcome in achieving this goal. It discusses what has been achieved so far and contributes some insight into methods which bring us closer to providing effective access to non-text media while concentrating on the central issue of efficiency.

1.1 Indexing Multimedia Objects Over a Distributed Network

The realization that text based techniques alone are an insufficient means of indexing non-alphanumeric data has lead to the start of a transition from alpha-numeric to content-based indexing (CBI) techniques.

Until recently, the continued use of alpha-numeric indexing techniques was a result of resource based restrictions. Increases in micro-processor performance, storage capabilities and network bandwidth have removed this once existent restriction, allowing systems to start making use of various content-based techniques as a means of measuring media object fitness.

The removal of these resource based restrictions has resulted in the realization of several new interesting problems. One of the more interesting of these problems involves the development of *indexing schemes*. For our purposes, an *indexing scheme* is defined as a metric that measures the fitness or quality of a given media object. This quality can either be an absolute measure or a measure relative to another media object of the same type. One example of an *indexing scheme* might be a face finding algorithm that takes an image as input and returns a percentage chance that the image contains a human face.

Associated with this problem are questions such as: Given all of the required resources, what is the best way to determine a multimedia object's fitness? Is there a single unifying *indexing scheme* which will provide users with the ability to effectively access desired content?

This work does not attempt to provide an effective means of accessing multimedia data through answers to questions such as these. Instead, the work focuses the elements of effectiveness associated with efficiency. Due to the fact that media objects are so large, it is not sufficient for a user to be able to effectively access relevant data - access must also be efficient. In this case, efficiency is defined by two parameters; time and traffic requirements. Firstly, it is important that a user be able to find relevant multimedia data in real or near to real time. Secondly, efficient access requires that only a minimal amount of network traffic be generated during both the indexing of media and the processing of queries. Even when dealing with the indexing of text documents, the greatest cause of inefficiency arises from the need to access and download each document[8]. The introduction of content-based indexing has resulted in an exponential magnification of this problem as text documents are extremely small relative to other media objects.

1.2 Deficiencies in Other Systems

Most current multimedia indexing systems address at least a subset of the problems illustrated above with varying levels of success. Although there are many exceptions, in general, multimedia indexing systems make use of some form of content-based indexing, acknowledging the fact that alpha-numeric indexing alone is insufficient.

Having recognized this, despite many efforts to address the issues of appropriate indexing schemes and efficiency, all current implementations fall short in at least one primary area.

Current systems provide the user with a "static metric" application. Although the system may allow the user to adjust and modify an inordinate number of parameters, it only allows the application of a single, constant, metric when determining the fitness of a media object. That is, although the user may be able to modify a given set of parameters, they are generally unable to specify exactly how the quality of a media object is judged. In order to apply truly custom indexing schemes on any significant number of media objects, the user would have to have the local resources allowing them to download, store, process and index all candidate media

objects. This is highly inefficient, violating one of the primary requirements of a successful media indexing system.

1.3 Motivation and Contributions

The primary motivation for the work that follows is based on observations made of existing multimedia indexing systems. As previously mentioned, despite the existence of several CBI systems, there are few systems that effectively deal with many of the problems related to efficiency.

The work that follows does not directly attempt to solve the problem of developing optimal *indexing schemes* for various multimedia formats. It does acknowledge the difficulty of the problem and attempts to address several issues of efficiency associated with it. This is done through the provision of an extensible, highly configurable system that allows for the efficient application of metrics with a variety of different levels of uniqueness. The following is a list of the contributions made by the work that follows:

- The work examines an implementation of the triangle inequality that allows for the efficient application of static metrics, providing functionality similar to that available through other systemsi (see Chapter 4).
- A methodology is developed allowing for the efficient application of customized metrics where a customized metric consists of a programmable combination of mulitple static metrics (see Chapter 4).
- Both the concept of the remote execution of truly unique metrics and its application are investigated and shown to be a viable means of scheme development and application (see Chapter 3).
- In addition to the examination of a system developed in order to prove the viability of various concepts examined throughout the work, all of the changes required to make the same system a fully functional product are highlighted (see Chapters 5 through 7).

1.4 Outline

The remaining chapters are organized as follows: Chapter 2 discusses much of the related work that has been done with regards to indexing multimedia databases. It examines another implementation of the *triangle inequality* as well as several web based image DBMS's and content-based indexing scheme implementations. Following this examination, Chapter 3 examines the concept of applying unique metrics to remote databases. Chapter 4 discusses how the *triangle inequality* was implemented so as to allow users to efficiently apply both single and multiple metrics to a large multimedia database. Following this, Chapter 5 examines an implementation of the concepts discribed in previous sections. Chapter 6 then looks at the performance of the system as it steps through several experiments that were performed on the system. The final chapter, Chapter 7, makes several conclusions with regards to the work and examines areas which require further investigation.

Chapter 2

Related Work

There are a variety of areas related to the content-based indexing of media objects that have received varying degrees of attention. Those which have received the greatest attention involve both the development and implementation of efficient indexing methods and the provision of appropriate interfaces to the data once it has been indexed. The following chapter examines a sample of the work done pertaining to the topic of content-based indexing of media objects. It focuses on those areas with the highest relevance to the work which follows. Most importantly, it examines areas such as interfacing with the Internet, indexing scheme development and application and remote processing.

2.1 Interfacing with the Internet

As it continues to grow and permeate various aspects of society, the provision of Internet access has become a necessity to the success and development of numerous products and services. It has repeatedly shown that it is a medium appropriate for the exposure and distribution of products and services.

The Internet, and all that it is comprised of, will be a major contributing factor to the success of content-based indexing for two very important reasons. Firstly, in order for content-based indexing systems to be useful, they have to provide users with robust service. That is, service which gives users effective access to an extremely large, diverse body of data. Presently, the only body of data that ultimately fits this description is the Internet. The second important way that the Internet will contribute to the success of content-based indexing systems is through the audience it provides. The sheer magnitude of the Internet's user base provides for demand

of services in even the smallest of niche markets. As a result of these two factors, one of the major goals pertaining to the development of content-based indexing systems has been their successful integration with the Internet.

2.1.1 Providing an Interface over the World Wide Web

One of the most important aspects of providing web based solutions is a successful user interface. With the development of Internet "friendly" programming and scripting languages such as Java, JavaScript, and Perl and the availability of standardized browsing software such as Netscape Communicator and Internet Explorer, the development of such interfaces has become relatively straight forward. Many interfaces have been developed using a variety of differing styles.

Most current interfaces force the user to make an alpha-numeric query prior to applying any truly content-based indexing techniques[11][21][22][23][24]. One such implementation is the image DBMS developed at Columbia University by John Smith called *WebSEEk*[23]. The interface is such that it initially allows the user to formulate a text-based description of an image through the use of an index very similar to that implemented by Yahoo (see Figure 1.1). The text-based description of the image is then used as a filter prior to the application of any of the system's colour similarity based CBI techniques. Figure 2.2 shows an interesting element of the *WebSEEk* interface. The figure shows the system's ability to take a sample image and increase its relevance or fitness based on adjustments to levels in its colour histogram. Having adjusted the histogram, the new image can then be resubmitted to the server following the standard query-by-example paradigm.

Another system which makes use of an interface somewhat similar to that employed by *WebSEEk* is the *ImageRover* system[21][22]. This system is similar in that it requires the user to enter an alpha-numeric query. Based on textual relevance, a number of images are returned to the user. These images are then used as part of a *query-by-example* mechanism. This mechanism allows the user to select any subset of the displayed images and resubmit them as a query to the server. This process can go on for as many iterations as the user desires,

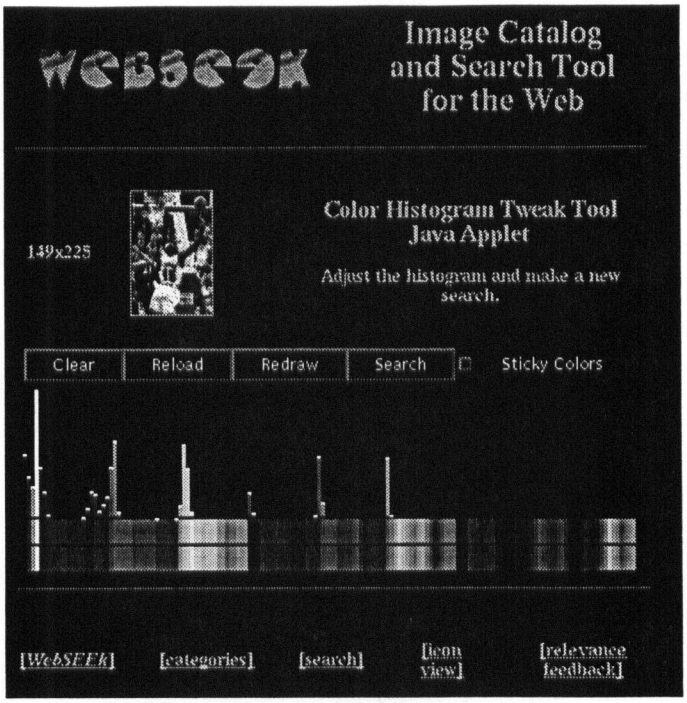

Figure 2.2: The *WebSEEk* colour histogram modification interface.

allowing them to constantly modify their query, including or discounting any previously selected images. The interface is interesting from the point of view that it allows the user to constantly refine their query while maintaining all of the images that were determined to have desirable properties along the way. The main problem with the interface is that the user does *not have* any access to the metrics being used to determine the fitness of the images.

One of the few systems which allows the user to make queries totally based on CBI techniques is the *QBIC* system[18]. Developed by researchers at IBM's Almaden Laboratories, the *QBIC* system has been applied to several application specific domains. Each implementation has required slight modifications to fit the requirements of the given domain. In its generic, demonstration version, the user is presented with a set of random images and several options as to how they may refine or develop a query. Among these options are the abilities to formulate custom colour queries, custom colour percentage queries, custom paint queries or to look at a further series of random images. The most interesting of these interfaces is the ability to create

custom paint queries. This interface allows a user to paint a custom colour image and use it to query the database. Figure 2.3 shows an example query with a blue area near the top and and

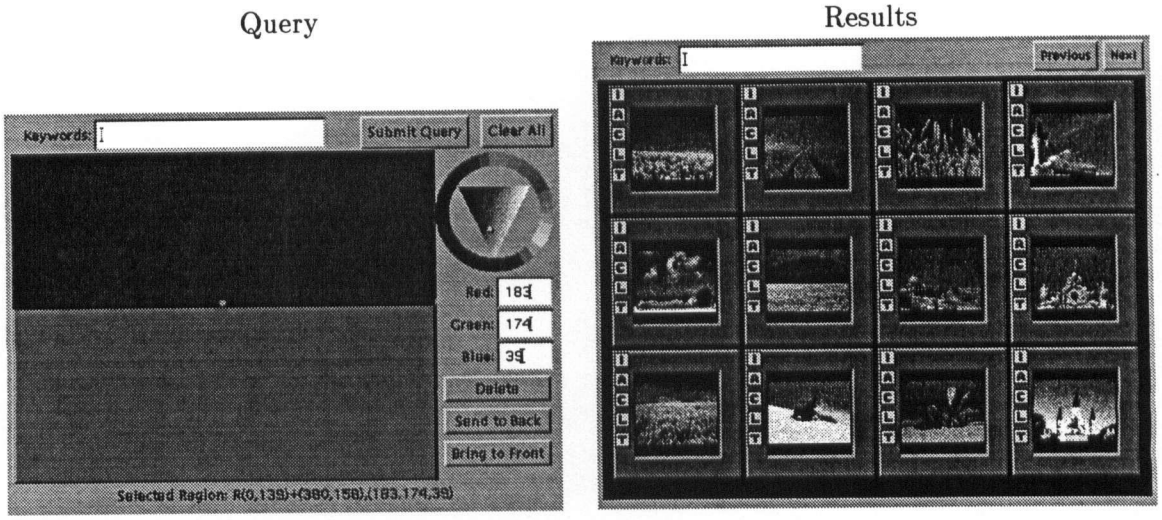

Figure 2.3: The QBIC custom paint interface: A sample query and set of results.

brown area near the bottom. As might be expected, the results show several images with blue sky, a horizon and some ground area.

2.1.2 Indexing World Wide Web Content

The implementation of an appropriate user interface does not insure the success or effectiveness of a CBI system. In order to make full use of the breadth of the Internet, users have to have access to tools and services which provide them with efficient and effective methods for querying a large portion if not all of the network's content. As mentioned before, there are several systems that provide users with effective user interfaces to their media databases allowing queries posted over the World Wide Web (WWW)[4][11][18][21][22][23][24]. Despite this faculty, there are few implementations which actually provide the user with the ability to query any substantial portion of the network's content. Instead, they provide the user with the ability to query static image databases containing perhaps thousands of images rather than the millions that are actually available[4][18]. Although this may be a sufficient solution for the interim, it

completely disregards the network as the resource it is.

Admittedly, the task of creating and maintaining an index of the multimedia content of the Internet is daunting. Estimations have been made stating that a system containing a complete index of the WWW would contain references to between 10 and 30 million images amongst the 50 to 100 million total documents[22]. Despite these inordinate numbers, there are several systems which are attempting to provide users access as extensive as that provided by current text based search engines. Among those engines attempting provide such access are *WebSeer*[11][24], *ImageRover*[21][22] *WebSEEk*[23], Yahoo's Image Surfer and the Lycos media search tool.

Stan Sclaroff's *ImageRover* is one of the more interesting implementations. This system claims to have been the first in a wave of content-based WWW image search engines, preceding its peers[21][22]. In describing his system, Sclaroff breaks the process of indexing the image contents of the Internet into three major challenges[22][1].

The first major challenge is the collection of images. Based on an experimentally determined gathering rate of approximately one image every 82 seconds, Sclaroff estimates that it would take 25 years to collect all of the image content available on the net with the use of a single robot. An implementation which allows for a high degree of parallelism provides performance such that 32 robots are able to collect over 1 million images monthly.

The second major challenge was overcoming the hurdle of indexing all of the retrieved content - *image digestion* as Sclaroff refers to it[22]. Having addressed the problem of image collection, it was found that a similar, highly parallel solution allows for images to be digested at a rate on par with which they are collected.

The third major hurdle that had to be overcome in the implementation of *ImageRover* was that of index navigation. Due to the fact that the index of a database is proportional in size to that of the database, an index referencing the entire WWW would be extremely large. In order

[1]Sclaroff actually discusses four major challenges including image collection, image digestion, image index and search and a user interface. The user interface is not discussed here because it does not present any Internet specific challenges.

to process queries in a reasonable amount of time (i.e. less than a second) several constraints had to be placed on the amount of processing required at run time. Sclaroff's solution involved the implementation of several techniques that allowed for a large amount of precomputation to be performed during the image digestion stage.

Overall, the three solutions, or at least variations of them, have become the generally accepted means of handling the immense size of the Internet. In fact, many of the techniques implemented by Sclaroff are also used in implementations which access small and medium sized databases. More importantly, the benefits gained from high degrees of parallelism and maximal amounts of precomputation have been recognized and consequently applied throughout this work.

2.2 Indexing Scheme Design

One of the most important and difficult problems to be addressed with regards to content-based media indexing is the question of developing metrics to test the fitness of media objects. Perhaps a result of the problem's difficulty, a number of differing approaches have been put forth. These approaches can be classified into one of two abstract categories. These categories are based on whether or not the content of a media object has to be analyzed during the application of the scheme. The two classifications are *superficial* and content-based indexing schemes.

2.2.1 Superficial Indexing Schemes

For our purposes, the classification of an indexing scheme as being superficial requires that the scheme is unaware of the contents of the media object - any indexing statistics that it extrapolates are derived from information external to the actual file data. There are two basic forms that superficial indexing schemes currently take. They can either make use of objective or subjective media object features.

Perhaps the most rudimentary indexing schemes are those based on objective media object features. Objective media object features are considered any quantitative data directly

associated with a file, external to the file body. This data can only be derived through the examination of a file's header or its storage attributes. For example, Frankel et al.[11][24] make use of objective media object features (in particular image features) in their *WebSeer* implementation[24]. The *WebSeer* implementation makes use of such attributes as whether or not an image is colour or grayscale, the size of the image, the image format, the size of the file containing the image and the date on which the file was created. With absolutely no image analysis, all of these attributes can be derived and used as a means of indexing. These features can be successfully used to filter undesirable media objects based on their qualities but are limited in that they do not make any consideration for the true content of the object.

In addition to its recruitment of objective media object features, the *WebSeer* implementation makes use of the second type of superficial indexing scheme - subjective media object features. Subjective media object features are those features that have been associated with a media object as a result of someone or something's perception of the media object. These features normally take the form of alpha-numeric data surrounding a media object.

WebSeer makes use of this data by applying an alpha-numeric filter to the text surrounding images in their respective HTML pages prior to performing any content-based indexing. This text filter approach is a good idea in that text processing can be done relatively quickly and inexpensively. In fact, due to the shear size of the WWW, this method might suffice as long as the user is looking for "any" related image and not "every" image or one image in particular.

The argument for the application of subjective features is based on the relevance of text surrounding media objects. The problem with the idea is that although the text may be relevant, it is difficult to consciously describe all perceptions of a media object let alone effectively describe it solely with the content that surrounds it on a web page.

2.2.2 Content Based Indexing Schemes

The implementation of superficial indexing schemes was a natural step in the development of media indexing systems. They provided a level of access to non-text media that was previously

unavailable. This access was provided through the re-use of many of the techniques used for standard text indexing. In addition to the relative ease with which they were implemented, they also maintained a relatively constant resource requirement. That is, aside from increased storage requirements, the resources required to perform superficial indexing schemes is nearly the same as that for indexing text documents.

Increases in storage capabilities, processing power and network bandwidth have provided developers with the ability to move beyond the requirements of superficial indexing schemes to content-based indexing schemes. Content-based indexing involves the examination and perhaps interpretation of the actual content of a media object. In general, these schemes can be classified into one of four areas based on their functionality. The four areas are *Object Classification*, *Structure Recognition*, *Region Discrimination* and *Comparative Analysis.*

Object Classification

The process of object classification involves examining the content of a media object and placing it in one of a number of discrete classifications. For example when dealing with a database of audio samples, it might be useful to divide the elements into groups based on their content. Some useful classifications might divide samples into those containing human voices, music or other specialized sounds such as gun shots.

Another example is the classification algorithms utilized by *WebSeer.* Algorithms developed by Vassilis Athitsos and Michael J. Swain allow *WebSeer* to classify images. The algorithms are based on several statistical observations about images and allow the system to efficiently and accurately classify images as being photographs or computer generated[3].

The usefulness of such classifications can vary widely depending on the users requirements. In cases where the user is searching for images of a given class, however, the use of object classification facilitates the discrimination between desired and unwanted objects with relatively little runtime processor requirements.

Structure Recognition

The desire to determine the presence or absence of various structures in media objects gives rise to a second type of indexing scheme. Differing from object classification in that it operates at a finer level, the usefulness of structural recognition is obvious. It provides systems with the ability to further discriminate media objects based on the structures they contain. There have been many examples of structural recognition especially with respect to images. Two of the more prominent structures recognized are human faces and landscape horizons.

- *Human Face Location and Recognition in Images and Video:* A large amount of work has been done in the development[17][19][25] and application[4][11][24] of facial location and recognition algorithms. The goal of the algorithms is to locate facial structures in images and possibly attempt to match them with previously located faces. The applicability of such indexing schemes is very wide spread. Aside from being useful for casually finding both random and specific people from image DBMSs, there are many applications of such algorithms especially in the area of security.

- *Horizon Detection:* Another popular structure recognition algorithm allows the user to detect horizons in images. Horizon detection is often desirable in that it aids with the ability to determine where a picture was taken. For instance was the picture taken indoors or outside? There are several implementations that make use of horizon detection as part of their indexing schemes[11][24].

In general, it is possible to develop highly accurate, and consequently successful, structure recognizers. The problem with the paradigm is that it is very difficult, if not impossible, to develop a generic structure recognizer without developing individual recognizers for all structures.

Region Discrimination

Yet another method of distinguishing media objects is through various regions that they may contain. In terms of image and video indexing, this area has benefited tremendously as a result of related work done in the field of computer vision[5]. Region discrimination with images is generally based on qualities such as their colour, texture, and shape[9][11][24].

Another area where region discrimination techniques are applied is in the digital video domain. Regions have been segmented and classified based on shot boundaries[2][7][16], camera motion[2][20], object motion[2], DCT coefficients[2] and frame features[2].

One example of region discrimination has been demonstrated by Drew Saur et al.[20]. Due to the difficulty associated with generic content-based video indexing, the group has concentrated on the annotation and analysis of basketball video. The goal of the project was to extract meaningful, high level information from given MPEG sequences of basketball video. Using various techniques based on estimated camera motion, they were able to successfully discriminate amongst regions containing "fast-breaks"[2], attempted shots, steals or loose balls.

Comparative Analysis

Also referred to as *query-by-example*[1][18][22], *relevance feedback*[23], *like-this*[10] or *distance measures*[6] the fundamental concept of comparative analysis is that a media object's fitness is judged based on its similarity to another of the same type. In terms of our definition of an indexing scheme the comparative analysis paradigm is really a *meta*-indexing scheme. That is, most indexing schemes can be applied using comparative analysis methodology. Consider the concept of region discrimination discussed earlier. Given two media objects, it is possible to apply various region discrimination techniques to them and then make some judgement of their similarity based on the results of the region discrimination techniques.

Very quickly, we can see how this methodology can be useful in the formulation of queries. One example of an interaction with an image indexing system that utilizes comparative analysis

[2]In basketball, a "fast-break" refers to the rapid movement of the ball from one end of the court to the other.

might be as follows.

For whatever reason, a user desires images of landscapes. They have an image that is close but not quite what they are looking for. The user formulates a query to the image indexing system using the image that they have in conjunction with a description of what properties of make it desirable (i.e. the blue colour of the sky). The system then accepts the query, applies various content-based techniques to the submitted image and responds with a list of images.

There are several systems that provide this functionality along with other variations of it. For example, the *QBIC* system allows users to create queries in several ways based on colour, texture, and image layout[18]. As discussed earlier (See 2.1.1), the *QBIC* system also allows the user to create queries based on custom colour(s) and user defined images essentially allowing the user to paint their own images and submit them to the query mechanism.

Yet another system worth examining is the *EXQUISI* project. This system, developed by Dwi Faulus [10] shows an implementation of a query interface language that allows the user to make *like-this* and *like-this-in-what* queries to large image databases. The system is interesting in that it allows the user to make queries to databases of similar images, based on sub-image level features.

2.3 Efficient Application of Indexing Schemes

As important as the development of indexing schemes is their efficient application. The reason that this is important is that despite continuous advances, processor cycles and speed remain finite. Due to the high dimensionality of media objects, runtime processing of individual media objects is unreasonable. In order to meet the requirement that a user receive a response in real or near to real time a great deal of work has been done on optimized indexing structures and algorithms. Two of the more interesting discussions on the efficient application of indexing schemes can be found in papers by White[26] and Berman[6].

White and Jain's work [26] focuses efforts on the Vector Space Model - a model which reduces all objects to fixed dimensional vectors. The application of the Vector Space Model

to the domain of media objects, in general, is difficult. This difficulty is a direct result of the high dimensional vectors required in order to accurately represent media objects. The paper discusses how this hindrance can be handled through the implementation of optimized R-trees and k-d trees. Included in the paper are several similarity retrieval algorithms including an algorithm which operates optimally when used with a combination of an inexpensive primary filter and a more expensive secondary filter.

Berman describes an image indexing system that makes use of the *triangle inequality* (See Appendix A). In addition to its library of images, the implementation uses several key images and various comparative analysis techniques which Berman refers to as distance measures. The basic principle behind the system is that the distance measures are used to determine the distance between each of the key images and the library images. Having calculated these distances, it is possible to determine a lower and upper bound on the distance of query images to library images. The result of this is a significant reduction in the number of images that have to be examined in order to find the closest matching library image. Most importantly, these techniques are not restricted to the image domain. Changing the actual fitness metrics allows them to be applied to all types of media. The results of this work are fairly impressive despite the fact that Berman has missed out on several key optimizations. These optimizations are a key element of this work and will be examined later in chapter 4.

2.4 Remote Processing

One of the largest problems associated with content-based indexing over a distributed network is the traffic associated with the transfer of media objects from one system to another. Both clients and servers contribute to this problem but a much larger percentage of the traffic is generated by the servers as they create and maintain their indices.

The general principle on which these servers work is based on what will be referred to as the Internet robot paradigm. The concept of Internet robots has been around for a while. The general idea behind them is that they automatically scour the web indexing its content. Thus,

given a starting point, they recursively traverse all of the possible links available, processing all interesting data. Usually, the result of a robot's actions is an index of all of the locations that it has visited and the media it has found there. These robots have reached a state where they have become very successful. They are powerful enough to generate and maintain indices spanning substantial portions of the World Wide Web.

There are many systems which implement this robot paradigm as a means for indexing World Wide Web content[1][11][21][22][24]. All of them face one common problem - the amount of traffic resulting from the generation and maintenance of their index. In order to perform indexing, all of the systems require all media to be downloaded from the remote server. The result is inordinate amounts of network traffic and local CPU requirements.

The obvious answer to the problem is the avoidance of any significant downloads. The more difficult question is how to avoid them. One possible solution can be found in the idea of remote processing. There may be many different definitions of remote processing, but for the purposes of our work, the idea of remote processing involves the packaging and shipment of jobs for processing at a remote machine.

The generally accepted model for remote processing is *RPC*(remote procedure calls). This model, developed in the 1970's, allows for a machine to call procedures located on a remote machine. The paradigm involves the client packaging up arguments, sending them to the server for processing and receiving a result. This model is advantageous in that it provides a viable mechanism for distributed computing. The main hindrance of the RPC paradigm is that the interaction has to conform to a strict protocol. Each procedure's arguments and results must be agreed upon prior to any interaction.

More recently, there have been developments that attempt to overcome the shortfalls witnessed with RPC. Most applicably are General Magic's *Odyssey* API[12] and classes associated with the Java programming language.

The *Odyssey* API is a set of Java class libraries which allow the user to implement *Mobile Agents*[12]. The *Mobile Agent* concept is discussed in Jim White's *Mobile Agents* white

paper[27]. In describing them, White discusses how they implement the *remote programming* paradigm. *Remote programming* differs from *remote procedure calls* in that computer-to-computer communication is no longer limited to sending arguments to procedures which already exist on a remote machine. *Remote programming* requires that a language be defined in such a way as to allow for entire state variant procedures to be sent from one machine to another for remote processing. A *Mobile Agent* is defined as both the procedure that is being sent and the state that it contains. More specifically, it is defined as "a process that acts autonomously on behalf of a person or organization...[with] its own thread of execution so that it can perform tasks on its own initiative"[12]. According to the documentation, the *Odyssey* API is an agent system, available from General Magic which allows users to define Mobile Agents.

Similar functionality is provided by the Java programming language. Newer versions of the programming language provide two alternatives for remote programming. Firstly, there is Java's Remote Method Invocation (RMI). This part of Java is similar in many ways to RPC and other RMI systems except that it has been optimized for the Java environment. That is, it is able to take advantage of the Java object model as a result of its assumption that it is working in a homogeneous system[15].

The second, more relevant, ability of Java comes from a combination of its ability to serialize objects[14] and reflect upon them[13]. Java provides programmers with the ability to serialize objects, and then send or receive them along a data channel. The only stipulation that is placed on serialized objects is that the receiver has to be aware of the object's class. This interface provides a great deal of functionality and is extremely useful for tasks such as storage and communication amongst machines.

The requirement that the receiving machine must be aware of the class from which an object was instantiated initially seems to be quite limiting. Further investigation, however, reveals the Reflection API[13] which provides us with some insight into avoiding this limitation. The Reflection API provides developers with the functionality required to examine all elements of a given class including attributes such as superclass, fields, methods and constructors. Chapter 3

discusses the importance of this capability, showing how it was applied in a system that allowed for the submission of truly unique *indexing schemes.*

This shows an implementation combining both serialization and reflection. The implementation allows for the users to apply unique indexing methods across a distributed network. The methodology is interesting in that it has the ability to virtually eliminate the traffic problems associated with current distributed indexing systems.

2.5 Summary

The work reviewed above, although not complete, is certainly a representative sample of that directly related to this work. It illustrates the fact that the problems of context-based indexing are well recognized and considered relevant to study. We can see that many advances have been made away from alpha-numeric systems towards a viable content-based solution and yet there are still a number of problems which need to be solved before a viable system is developed.

The following problems illustrated by previous implementations are dealt with in the work to follow:

- *Static Query Mechanism:* Although many of the systems provide interfaces which are available over the World Wide Web, most if not all of them are static. The following system attempts to define an architecture which allows for a variety of interfaces to act as the front end for the same database. The advantage of such a system is that the interface can be implementation dependent, providing a variety of options dependent on user requirements.

- *Indexing Scheme Application:* All of the systems address the application of *indexing schemes.* Some of them allow for combinations of multiple schemes[6] but, in general, only one, hidden scheme is applicable. This work addresses this issue by providing a mechanism to allow for the application of indexing schemes varying in uniqueness with variable levels of efficiency.

- *Internet Robot Inefficiencies:* Finally, a suggestion is made that may help avoid the traffic generated by current implementations of Internet robots. The solution discussed is similar to the Mobile Agent[27] paradigm and allows for a significant decrease in the traffic associated with indexing.

Chapter 3

Applying Unique Indexing Schemes

An *indexing scheme* is said to be unique if the media server applying it has no knowledge of it prior to query submission. One of the primary deficiences of current media indexing systems are the restrictions that they place on the application of unique *indexing schemes*. Currently, there is not a single system that allows a user to submit, in addition to their query, a unique means by which media object fitness should be measured. There are some systems that attempt to address the problem through the implementation of configurable metrics. Such systems provide interfaces that allow the user to "tune" their query through the alteration of various parameters associated with a single metric. Other systems address the problem through the provision of a variety of different metrics. For the most part, however, media indexing systems are quite limited in terms of the variety of metrics that they can apply. Most systems only provide the user with the ability to submit standard queries for processing via a completely static method.

This lack of functionality is significant for a number of reasons. Perhaps the most significant is its effect on the progress and efficiency of *indexing scheme* development. The overhead currently associated with the development of *indexing schemes* is costly enough to inhibit progress in the general area of CBI. Currently, in order to develop, test and apply new and unique *indexing schemes* a user must first develop an entire media indexing system of their own. From the point of view of efficiency, this is not acceptable at all. In addition to development time, in order to make the system useful, a great deal of traffic is generated as the user downloads and indexes all of the media to be included.

One of the primary goals of this work was to overcome this problem, effectively decreasing the overhead associated with scheme development, removing the deficiencies associated with

current systems and improving overall efficiency. This goal was met through the provision of a software interface. The interface allows users to develop and package their own unique metrics in the form of a Java class and submit them to a server. No longer are users constrained to toggling parameters of fixed metrics located on remote machines. The system allows the user to submit and apply unique *indexing schemes* located on any Web server. The only predetermined aspect of the submitted scheme is the interface and naming conventions with which it must comply. In addition to the various other steps associated with standard queries, the general process of applying unique schemes simply involves notifying the server of the location of the scheme in question via the client interface.

3.1 The Interface

In order for the media server to successfully access and apply the correct methods contained within a given class, the class must conform to a specific interface. This interface is simple, well defined and yet non-restrictive. Aside from the fact that they must be written in Java, unique schemes need only conform to at most four major guidelines in order to comply with the interface.

The first stipulation of the interface is that all unique schemes must be subclasses of the `IndexingScheme` class. In Java syntax, they must `extend` the `IndexingScheme` class. This is done to insure that the functionality provided by the `IndexingScheme` class is available during both development and application.

Secondly, the user has to implement a `static` method named `index`. The intent of this method is to contain the actual metric expressed by the class. As illustrated below, this method can be implemented in one of two flavors.

```
public static Double index(MediaDocument)
public static Double index(MediaDocument, MediaDocument)
```

The first implementation of the method takes a single object which is a member of the

`MediaDocument`[1] class and returns a `Double` - the Java class representing double precision floating point numbers. The lower the value of returned, the more desirable the object. An implementation of the first type is intended to determine media object quality based on a non-comparative calculation. For example, a scheme of this type could return how close the average magnitude of motion vectors contained in a video clip are to a given value. This differs from the second format which is intended for use when implementing comparative *IndexingSchemes*. For example, in the case where a user wished to determine the similarity of two images based on the qualities of their respective color histograms, this would be the desirable implementation.

An important feature of the system is that the methodology of the `IndexingScheme` does not have to be constrained to the `index` method or the class in which it is contained. That is, the system's design allows the `index()` method to make reference to other methods within the current class as well as external methods contained in other classes as long as the other classes are located at the same code base.

Any `IndexingScheme` conforming to the above restrictions will pass all compliance tests implemented by the server. In order for a scheme to be maintained on a server, taking advantage of the efficiencies provided by the triangle inequality, it must also conform to two additional restrictions. Firstly, all values returned from the `index` methods must be within the range from 0.0 to 1.0. Again, the lower the value returned, the more desirable the object is assumed to be. In addition to this, as will be discussed in Chapter 4 comparative schemes implementing the second interface must calculate linear distances between objects.

3.2 Flagging Problems

A common problem associated with any client-server architecture is compliance. It is often very difficult to insure that the information being shared between both the client and server meet certain standards. In terms of providing a user with an interface that allows the submission of unique *indexing schemes* these problems generally manifest themselves in the form of naming

[1] The `MediaDocument` class encapsulates all of the properties associated with the various types of media objects. A full description is provided in section 5.4.2

errors. In some cases, the `IndexingScheme` class is valid and will compile but gets rejected by the server. This rejection is a result of the fact that the methods contained within the class do not comply with the naming conventions defined by the server.

One possibility for handling these types of errors is to abort the query and return a meaningless response. Instead, this system has implemented a series of checks that insure that any submitted classes are valid *IndexingSchemes* prior to attempting to apply them. If they are not void, a meaningful error is returned to the user. There are many advantages to implementing such a system. The most important advantage is the ease with which it is possible to debug otherwise functional schemes.

All of the error checking is done using the Java Reflection Application Programming Interface (API). A standard part of the Java programming language, this API allows the runtime examination of classes. Functionality is provided allowing access to all aspects of a given class including all of the fields, methods and constructors. Basically, Java supports several key classes which are required to represent classes. Among these are the `Class`, `Constructor`, `Field` and `Method` classes. Each of these classes support methods which, when combined, provide all of the functionality required to determine the exact interface of a class.

Currently, several, fairly strict, checks are made by the server to insure that submitted classes are compliant. Upon retrieving a remote class, several of its properties are checked using the `IndexingScheme static, final` class method `boolean checkCompliance(Class scheme)`.

The method has been declared a `static` member of the `IndexingScheme` class to insure that all scheme developers have the ability to check the compliance of their schemes prior to submitting them to a server. Secondly, the method has been declared as `final` to prevent it from being overridden by any sub-class. Having defined the method in such a way, it takes a Java `Class` object as its only argument and performs a number of tests verifying appropriate subclassing and the presence of several key fields and methods.

3.3 The *IndexingScheme* Base Class

All of the *i*ndexing scheme functionality discussed thus far is actually contained in the form of `static` methods in the `IndexingScheme` base class. This next section discusses the additional properties of the class, including member variables and other, user accessible, methods.

Firstly, the `IndexingScheme` class contains two `public` member variables as shown below.

```
public static String name
public static String description
```

The first, `name`, has been implemented to allow for the storage of a name free of any Java context. The second, `description`, contains a meaningless default value but was designed to store a detailed description of the workings of the scheme. All subclasses should override these member variables with meaningful values. They become important when the scheme is made publicly available and users are deciding whether or not it meets their requirements.

In addition to the member variables, the class contains several `public static final` methods.

```
int checkCompliance(Class)
String getName(Class)
String getDescription(Class)
double applyScheme(Class, MediaDocument)
double applyScheme(Class, MediaDocument, MediaDocument)
```

The `checkCompliance(Class)` method, as discussed earlier (see section 3.2), can be accessed by the user to make sure that a scheme complies with the server's requirements. In the case that the given scheme does not comply, the method will throw a `SchemeComplianceException` containing information regarding the compliance violation. It should be noted that this is the same function used by the server to check compliance before applying the scheme to any MediaDocuments. Both the `getName(Class)` and `getDescription(Class)` have been implemented for easy access to both the `name` and `description` member variables respectively. Finally, both flavors of the `applyScheme()` return the result of applying the given *IndexingScheme* to the

provided *MediaDocuments*. Again, it is important to note that, in addition to allowing the user to test developmental schemes locally, it is the same method used by the server when applying *IndexingSchemes*.

3.4 A Sample Scheme

The following section discusses a sample *IndexingScheme* called `BinToBinScheme`. The `BinToBinScheme` encompasses a metric that takes two images and determines their similarity based on the values contained in their RGB color histograms. The measure is based on a sum of the binwise absolute differences for each of the red, green and blue bands indexed by the histogram.

In terms of its implementation, `BinToBinScheme` is quite simple. It illustrates an implementation of an *IndexingScheme* that conforms to all of the restrictions discussed earlier. The source code for the scheme can be seen in Figure 3.4. The implementation of the actual metric is encompassed in the `ColourHistogram` class's method `double binToBinDifference(ColourHistogram)`. In order to access the metric, the scheme's `index` method accesses the appropriate method in one of the image's `ColourHistogram` objects and returns the result in the form of a `Double`.

This scheme is a good example because it illustrates several important points while hiding unnecessary details. Firstly, it illustrates how to override both the `name` and `description` member variables. Additionally, it illustrates an implementation of the `index` method in its two argument format. Finally, the ability to access methods available in other classes is displayed via the call made to the `binToBinDifference` method contained in the external `ColourHistogram` class. The only restriction that it does not directly illustrate is the fact that the values returned by the `index` method are constrained between 0.0 and 1.0. The constraint is implicit in that the `binToBinDifference` method is defined such that it returns values in the appropriate range.

```
package thesis.schemes;

import java.awt.Color;
import thesis.media.ColourHistogram;
import thesis.media.ImageDocument;

/**
 * This <CODE>IndexingScheme</CODE> operates on a pair of images determining
 * how close they are in terms of their colouring. This is done by taking a
 * bin-to-bin difference of their RGB colour histograms.
 * The <CODE>index</CODE> method returns a number between 0.0 and 1.0 where
 * 0.0 indicates an exact match and 1.0 indicates a complete dissimilarity.
 */

public class BinToBinScheme extends IndexingScheme {
   /**
    * This meaningful name of this indexing scheme
    */
   public static String name = "BinToBinScheme";

   /**
    * This string stores a description of how the <CODE>IndexingScheme</CODE>
    * operates.
    */
   public static String description = "This IndexingScheme operates on a "+
      "pair of images and determines how close they are in terms of their "+
      "RGB colour histograms";

   /**
    * This method takes two <CODE>ImageDocument</CODE>s and determines how
    * similar they are based on their RGB colour histograms.  The result is
    * returned as a <CODE>Double</CODE> value between 0.0 and 1.0.  Simply
    * stated, it applies the <CODE>binToBinDifference()</CODE> method
    * implemented by the <CODE>ColourHistogram</CODE> class.
    *
    * @param imageA   The first of two <CODE>ImageDocument</CODE>s to compare.
    * @param imageB   The second of two <CODE>ImageDocument</CODE>s to compare.
    * @return         A <CODE>Double</CODE> indicating how similar these images
    *                 are based on their average colours;
    */
   public static Double index(ImageDocument imageA, ImageDocument imageB){
      return new Double(
        imageA.colourHistogram.binToBinDifference(imageB.colourHistogram));
   }
}
```

Figure 3.4: Java source code for `BinToBinScheme.java`

Chapter 4

Using the Triangle Inequality to Improve Efficiency

One of the primary goals specified in the design of the media indexing system was that it had to be efficient. This efficiency was required of various aspects of the system related to both the creation and querying of an index of a large database of media objects. The provision of such efficiency is an important requirement in that it allows for a large number of users to access an even larger database. This chapter examines a method for providing efficient handling of user queries to a very large media database. It discusses an application of the triangle inequality optimized for handling *best-n* queries. *Best-n* queries are those queries which require the n best matches in the database to be returned as a result of the query. These results are generally sorted with respect to how closely they match the query requirements.

As discussed in Appendix A the *triangle inequality* states that the distance between any two objects cannot be greater than the sum or less than the absolute difference of their individual distances to a third object. The following is a description of a method which uses this law as a means of facilitating efficient queries to large media databases. Before continuing, it is important to note that all metrics should be commutative. That is, $|AB| = |BA|$. If this requirement is not met, then there can be problems with the application of the triangle inequality.

4.1 A Basic Application of the Triangle Inequality

The first step in the application of the *triangle inequality* is the assumption of the *Vector Space Model* (VSM). As mentioned earlier, the VSM reduces all objects to fixed dimensional vectors. That is, various attributes and properties of the objects are quantized and stored as the dimensions of their associated vector. The methods by which this reduction is performed

are important in that they determine the effectiveness of any indexing to follow. It determines what information and attributes, originally possessed by the object, shall contribute to the indexing process. More importantly, this step determines the amount of high level information to be derived from the object.

Perhaps the most interesting aspect of the object reduction process is that it can encompass a variety of different measures providing further indexing with access to both high and low level object attributes. For example, one dimension of the vector could be a low-level attribute of the object like the number of pixels in an image. Similarly, another dimension might represent a higher level attribute of the object such as the number of human faces stored in the same image. This capacity is interesting in that it will help facilitate the implementation of custom queries. Again, it should be noted that it is not a goal of the system to provide the user with a robust set of metrics, but rather to provide them with the capabilities to implement and apply arbitrarily complex indexing schemes as they require.

Having assumed the Vector Space Model, we can now examine how the definition of *indexing schemes* fit with the model. Earlier, *indexing schemes* were defined as metrics which measure the fitness or quality of given media objects. This definition fits in with the *VSM* on a number of different levels. Firstly, *indexing schemes* are used to process and interpret the raw data contained within media objects. They extract and synthesize the information to be stored in each dimension of the object vectors. Similarly, the definition of an *indexing scheme* can be applied at a higher level of abstraction. Having reduced a media object to its representative vector through the application of various *indexing schemes*, higher level schemes can be designed. These higher level schemes indirectly measure a media object's fitness through the interpretation of the *meta*-data stored in its representative vector. That is, they indirectly measure an object's fitness through various permutations and combinations of the data contained in the vector.

The assumption of the VSM facilitates the implementation of comparative *indexing schemes*. Again, a comparative *indexing scheme* is a scheme that, given two `MediaDocument` objects,

judges the quality of one of the **MediaDocuments** based on its similarity or dissimilarity to the other. As we move towards an implementation of the triangle inequality, these comparative schemes will become important, acting as distance measures, providing an estimate of how similar various media objects are.

We define $I(a, b)$ as the value returned when the comparative *indexing scheme* I is applied to the MediaDocuments a and b. This value is between 0 and 1 where 0 indicates a perfect match between object a and b and 1 indicates complete dissimilarity. Next we define the three different types of **MediaDocument** objects that will exist and interact with the database. Library documents (l) are defined as all **MediaDocument** objects contained within the media database. These are the objects that query results will be drawn from. Their defining property is the fact that they are available prior to the submission of any queries. This is important in that it allows for preprocessing to be performed. Key documents (k) are defined as a representative subset of the library documents. The goal for the selection of Key documents is to have as small a number of them as possible while maintaining an accurate representative sample of the library documents. In the ideal case, they are evenly distributed amongst the vector space of the library documents as defined by the VSM. Finally, Query documents (q) are defined as any document that is being submitted as part of a comparative analysis query. The important characteristic of Query documents is that the system has no prior knowledge of their existence. This deficiency makes it impossible to perform any sort of preprocessing. Without the *triangle inequality* this would result in unreasonable runtime processing requirements thus limiting access to the database.

Having defined the various types of objects associated with the system we shall consider a media database that consists of a set of library objects $L = \{l_1, ..., l_N\}$, a set of key objects[1], $K = \{k_1, ..., k_M\}$, and, for now, a single *indexing scheme* I. Prior to the submission of any queries, the objects are reduced to a single dimensional vector through the pre-calculation of $I(k_v, l_w) \forall \{1 \leq v \leq M\}$ and $\{1 \leq w \leq N\}$.

[1] Although it is not necessary, this set of objects is generally a small subset of the library objects.

Upon the submission of a query image, q, the values of $I(k_v, q)$ are calculated. Based on the *triangle inequality* it is then possible to place a bound on both the minimum and maximum distance between the query object and any library object, l.

$$max_{1 \leq v \leq M} |I(k_v, l) - I(k_v, q)| \leq I(l, q) \leq min_{1 \leq v \leq M} (I(k_v, l) + I(k_v, q)) \tag{4.1}$$

Equation 4.1 shows how the maximum and minimum bounds on library-query object distances can be efficiently calculated based on pre-calculated key-library object distances. It shows that the maximum value of $I(l, q)$ is less than or equal to the minimal sum of the distance between the library object, l, and any key object, k_v and the query object, q and the same key object. More importantly, it shows that a lower bound can be placed on library-query distances. This lower bound is the maximal value that can be obtained from the absolute value of the difference between the distances separating the library object, l, and any key object, k_v, and the query object, q and the same key object.

Based on these bounds, finding the library object that best matches the query object according to the indexing scheme can be done in a manner similar to Berman's[6] as follows. First, B_1 is defined as a variable that will be used to store the distance between the query object, q, and its current best match in the library, l'. By its nature, B_1 will always be less than or equal to its value on the previous iteration.

Having defined B_1, we can move linearly through the index, calculating $I(l, q)$, the distance between the library and query objects only when the minimum possible distance between them, as defined by equation 4.1, is less than the current value of B_1. During this process, the library object associated with the current value of B_1 is maintained and then returned as the best match upon completion of the traversal.

Ideally, at least one of the key objects shares properties identical to those of the query image. The result of such a condition would be that the value of the distance lower bound illustrated by equation 4.1 would perfectly represent the actual distance between the query object and the library object. Although that situation does not insure a minimal number of

distance calculations, it does prevent any calculations resulting from poor lower bound values. Accurate representation such as this, however, is rarely the case. The fact that query objects do not generally share properties which are identical to the key objects provides for a very inefficient worst case scenario. This scenario involves the runtime calculation of all of the distances between the query object and each of the library objects - obviously an unacceptable situation when dealing with large numbers of media objects and computationally expensive metrics.

Even in the best scenario, the minimum possible distance between all but one library object and the query object must be examined. Although this comparison is relatively inexpensive, the potential magnitude of the database requires all possible optimizations to be made.

Finally, this algorithm does not perform very well under time or computational restrictions. That is, in the case where restrictions have been placed on the amount of time or compute cycles allowed to handle a query, the algorithm makes no effort to examine the best candidates first. In order to circumvent all of these problems and allow for *best-n* queries to be processed, several modifications have been made to the algorithm.

4.2 An Optimized Version

The following section examines several modifications that have been made to the previous algorithm allowing for more efficient query handling. The most important modification to the algorithm affects pre-calculation. In this case, not only are $I(k_v, l_w)$ calculated for all key and library objects, the results are also sorted. The results of measuring the distance from each library object to each key object are sorted so that the library object most similar to the key object has the lowest index and the most dissimilar object has the highest. The results of these calculations are stored in an array of doubly sorted vectors. That is, each key object has a vector of library object-distance pairs sorted by both the object and the distance value associated with it. The nice property of this amendment is that, in addition to assisting with a solution to the problems mentioned above, no further requirements are placed on runtime

resources.

Having made the modification, the submission of a query requiring the best n library objects to be returned is handled by, again, calculating the actual distances from all of the key objects to the query objects: $I(k_v, q)$ for $\{1 \leq v \leq M\}$. Based on this calculation, the theoretically best possible candidates are those library objects that are the same distance from any of the key objects as the query object.

The next step of the algorithm involves the placement of the best candidates according to each key object onto a sorted queue. Again, the quality of the candidates is determined by the lower bound on their distance to the query object. The sorted queue was designed such that elements are always inserted in sorted order, lowest values near the front of the queue and removed from the front. As elements are removed from the queue any of their neighbors from their associated key vectors are added.

Having placed the library candidates with minimal lower bound distances to the query object on the queue, we begin to process. Elements are continuously removed from and added to the queue until the lower bound on the library object currently being processed is higher than B_n, the actual distance from the query object to the nth best element found thus far as measured by the applied *indexing scheme*. Once the lower bound of the first object on the queue exceeds this value, the search is complete. This is a valid indication of completion because we know, as a result of maintaining the objects in sorted order, that no object is closer to the query object than those that have already been examined.

Although the above method avoids a great deal of redundant calculations, it doesn't make optimal use of the information made available through the lower bounds pre-calculations. The naive method that accompanies the above sorted queue model would simply pull objects from the queue and calculate the actual distance between the library object and the query object. This method can be improved through further exploitation of the pre-calculation described earlier. Theoretically, the library objects that are being added to the queue are those which are closest to any particular key object. The advantage of having multiple key objects is that the

lower bound on the distance between a library object and the query object calculated relative to one key object is not necessarily the lower bound relative to all of the key objects. Consequently, a further reduction in the number of actual library-query object comparisons can be made. This reduction can be made by finding the maximal lower bound on the distance between the library and query objects in question and only calculating the actual distance if this maximal bound is less than the nth best. Thus, library objects are removed from the sorted queue and if their maximal lower bound has a value less than the nth best actual distance calculated so far, the actual distance between it and the query object is calculated. If this value is better than the nth best, the object is inserted into the results list R. Again, once the lower bound for any library object in the queue is greater than the nth best, the query has been completed and the results are returned.

Not only does this algorithm avoid the redundant runtime calculations and comparisons of Berman's method, it also provides a better solution to the resource restriction problem discussed earlier. The algorithm attempts to examine the best possible candidates first and in doing so provides better results in time or processor limited situations. In addition to this, it also provides for query extensions. If the user initially asks for the n-best matches, the query can be extended to provide results for the $n + x$-best matches without having to reprocess the initial query elements.

4.3 An Example

4.3.1 The Query Conditions

The following example is a simple illustration of the method described above. For simplicity, we will consider a very small image database where the set of key images, K, are distinct from the set of library images, L. Our example will consider an image database management system (IDBMS). The system will consist of a set of three key images, $K = \{k_0, k_1, k_2\}$, ten library images, $L = \{l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8, l_9\}$ and a single *indexing scheme*, I. The query to be submitted will consist of a query image, q and a request for the best 2 matching images from

the image library.

The indexing scheme, I, will be very simple. It will judge image similarity based on the number of pixels contained in both the x and y dimensions of the images.

$$I(a,b) = \frac{\sqrt{(a_x - b_x)^2 + (a_y - by)^2}}{1000} \tag{4.2}$$

Equation 4.2 shows the formula that will be implemented. The value of the divisor is 1000 purely for the sake of cosmetics. it resulted in a set of proximity values ranging between 0.0 and 1.0 as required by the *indexing scheme* definition. When this scheme is applied to a pair of images, the closer two images are in terms of the number of pixels they contain in each dimension, the lower the value that will be returned. This scheme was selected for a number of reasons. Firstly, it is really just a Euclidean distance measure with a scaling factor. This makes it both easy to comprehend and express mathematically. Secondly, it encompasses a two-dimensional problem space. This is advantageous in that two dimensional problem spaces are easy to visualize.

Table 4.1 shows the actual dimensions of all of the images to participate in the example query. The images were arbitrarily selected from the images that were members of a larger IDBMS used during the actual implementation and testing of the system described earlier. They range in size from approximately 100 to 400 pixels in both the x and y dimensions.

Figure 4.5 shows how all of the images, key, library and query are distributed over the two dimensional space of x and y pixels. We can see that despite the fact that they were selected randomly, the distribution of the key images is somewhat even over the problem space. Similarly, the library images are also well distributed. Visually, they are clustered around the key images. In fact, they can be clustered into three groups each relatively close to a distinct key image. This is a nice feature in that it will allow for the key images to accurately represent members from the various groups. Finally, and most importantly, the query image, q, is located very close to k_1. This is a highly desirable feature in that the closer the key images represent the query image, the faster the algorithm should, theoretically, be able to find the best match.

Image Set	Width	Height
Key Images		
k_0	308	209
k_1	195	270
k_2	142	195
Library Images		
l_0	388	160
l_1	399	187
l_2	282	187
l_3	141	109
l_4	150	144
l_5	172	251
l_6	108	150
l_7	226	282
l_8	361	239
l_9	208	255
Query Images		
q	188	265

Table 4.1: Sample IDBMS image dimensions.

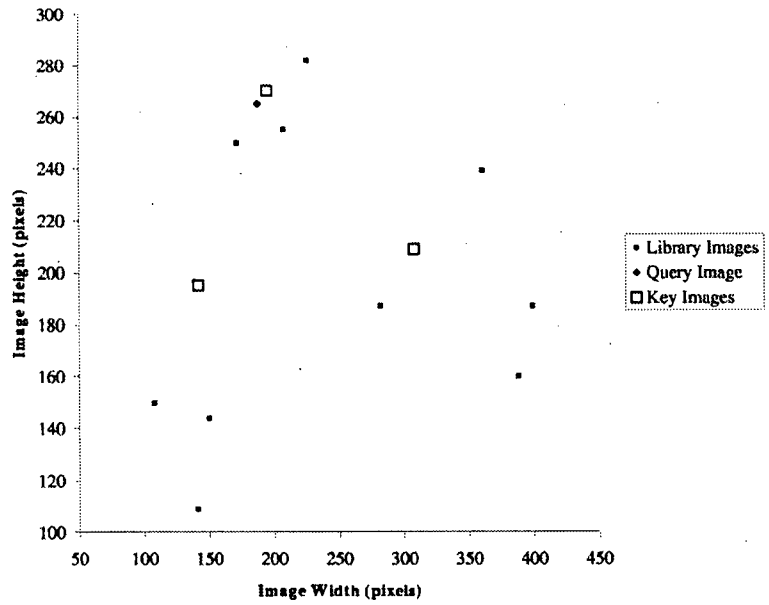

Figure 4.5: Distribution of images involved with query over 2D pixel space

4.3.2 Handling the Query

Upon the submission of the query image, q, all of the key-query similarities are calculated. These similarities are calculated through the application of $I(k_v, q)$ for each of the key images, $0 \leq v \leq 2$. Table 4.2 shows the results of this application.

Image	$I(k_v, q)$
k_0	0.132
k_1	0.009
k_2	0.084

Table 4.2: Key-Query image similarity measures.

The calculation of all of the key-query distances, combined with the pre-calculated key-library distances allow for lower bounds to be placed on all of the query-library similarities. These lower bounds are calculated through the application of the triangle inequality. More specifically, as shown by equation 4.3, the similarity of the query object, q, and any library image, l_x, is no less than the absolute value of the maximum difference between the distance from any key image to the library image and the same key image to the query image.

$$I(l_w, q) \geq max_{0 \leq v \leq 2} |I(k_v, l_w) - I(k_v, q)| \tag{4.3}$$

Given that the distances between all of the key and library images have been pre-calculated and sorted, for each key image, a library image is selected as the candidate with the best possibility of matching the query image. These candidates are selected based on their distances to the key images. A key image's best candidate is that image that has been shown to have a similarity to the key image that most closely matches that of the query image. More specifically, the best candidate for each key image is $min_{0 \leq w \leq 9} |I(k_v, l_w) - I(k_v, q)|$.

Table 4.3 shows both the pre-calculated and sorted key-library image distance measures and the lower bounds on library-query distances for each key image. Based on calculations involving the key images, $\{k_0, k_1, k_2\}$, the table shows that the best candidates to match the query image, are $\{l_5, l_9, l_3\}$ respectively.

	l_2	l_8	l_1	l_0	l_7	l_9	l_5	l_4	l_3	l_6
$I(k_0, l_x)$	0.034	0.061	0.094	0.094	0.110	0.110	0.142	0.171	0.195	0.209
$\|I(k_0, l_x) - I(k_0, q)\|$	0.098	0.072	0.039	0.039	0.023	0.022	0.010	0.038	0.062	0.076
	l_9	l_5	l_7	l_2	l_4	l_6	l_8	l_3	l_1	l_0
$I(k_1, l_x)$	0.020	0.030	0.033	0.120	0.134	0.148	0.169	0.170	0.220	0.222
$\|I(k_1, l_x) - I(k_1, q)\|$	0.011	0.021	0.025	0.112	0.125	0.140	0.160	0.161	0.212	0.214
	l_4	l_6	l_5	l_3	l_9	l_7	l_2	l_8	l_0	l_1
$I(k_2, l_x)$	0.052	0.056	0.064	0.086	0.089	0.121	0.140	0.223	0.248	0.257
$\|I(k_2, l_x) - I(k_2, q)\|$	0.032	0.027	0.020	0.002	0.005	0.037	0.056	0.140	0.165	0.173

Table 4.3: Key-library image similarity and lower bounds on library-query image similarity.

Based on their similarity to the key images, these three will be the initial images to be placed on the sorted queue, Q. Using the notation, $l_y^{x.xxx}$ where $x.xxx$ indicates the library images distance measure to the key image and y identifies the library image, the queue will initially have the following order $Q = \{l_3^{0.002}, l_5^{0.010}, l_9^{0.011}\}$.

Since the result list, R, is empty, the first element on the queue is automatically removed for inspection. Given the above order, $l_3^{0.002}$ is removed from Q, $I(l_3, q)$ is calculated yielding a distance measure of 0.1629, leaving $R = \{l_3^{0.1629}\}$. As per the method described earlier, both $l_5^{0.020}$ and $l_9^{0.005}$ are placed on the queue. They are placed on the queue as a result of the fact that they are neighbors of l_3 in the k_2 list of proximities. Thus, $Q = \{l_9^{0.005}, l_5^{0.010}, l_9^{0.011}, l_5^{0.020}\}$. As will be seen, the fact that both l_5 and l_9 are on the queue in two different positions doesn't matter.

Again, since $|R| < n$ the first element, $l_9^{0.005}$, is removed, $I(l_9, q) = 0.0224$ is calculated, its neighbor $l_7^{0.037}$ is placed on the queue. Now, $R = \{l_9^{0.0224}, l_3^{0.1629}\}$, $B_n = 0.0224$, and $Q = \{l_5^{0.010}, l_9^{0.011}, l_5^{0.020}, l_7^{0.037}\}$.

It is at this point, when $|R| = 2$, that the algorithm becomes interesting. The next element on the queue is $l_5^{0.010}$. This element is pulled off of the queue. Its actual proximity to q has never been calculated, therefore, its maximum lower bound on the distance to q is determined to be 0.021 (as indicated by the k_1 proximity list). Since this value is less than B_n, we calculate $I(l_5, q) = 0.0213$. This iteration is finished by placing $l_9^{0.022}$ and $l_4^{0.038}$ on to the queue. The result

list now contains $R = \{l_5^{0.0213}, l_9^{0.0224}\}$, $B_n = 0.0213$ and $Q = \{l_9^{0.011}, l_5^{0.021}, l_9^{0.022}, l_7^{0.037}, l_4^{0.038}\}$.

As a result of the fact that $I(l_9, q)$ has already been calculated, $l_9^{0.011}$ is now removed from the queue and its neighbor $l_5^{0.021}$ is added without any further calculation. This process continues, for three iterations, removing elements that have already been calculated and placing their neighbors on until a state is reached where the queue is $Q = \{l_7^{0.023}, l_7^{0.025}, l_6^{0.027}, l^{0.037} l_7, l_4^{0.038}\}$.

4.3.3 Completion

At this point, the algorithm finishes. Completion is realized due to the fact the lower bound on the next element in the queue, $l_7^{0.023}$, is higher than the current value of $B_n = 0.0213$. We know that we have examined all of the documents required due to the fact that, all of the best candidates were initially placed on the queue. That is, the images with the lowest lower bounds from each of the key images. As these elements were processed, elements with continuously poorer lower bounds were placed onto the queue, maintaining their sorted order and always processing the candidate with the minimal lower bound. This process insures that the best candidate will always be at the head of the queue, thus insuring that no candidates are overlooked.

Having returned the n-best elements, it is very easy to see that through maintenance of the queue, we could extend the results to include the $n+x$-best elements without any re-calculation of the initial n-best.

If we examine the overall process, we see that despite the fact that we placed 11 elements (5 unique elements) onto the we queue, distance measures were only calculated for $\{l_3, l_9, l_5\}$. This is excellent considering that the optimal case requires 2 elements be compared. In terms of quality, the results of the query were $R_{final} = \{l_5, l_9\}$. Closer inspection will reveal that, indeed, these two images are the best matches to the query object as both differed from the original by at most 20 and 14 pixels in the respective dimensions. This verification can be done by either calculating the distances to all of the other images or through inspection of the graph shown in Figure 4.5.

4.4 Extending the Algorithm

The previous example shows an implementation of the *triangle inequality* that allows the user to apply a single *indexing scheme* to a large body of data. This method has shown excellent results in terms of its efficiency but is lacking in at least one key area. This deficiency is spawned from the fact that it requires prior knowledge of the metric. This requirement makes the method quite limiting in terms of query formulation. It effectively limits the user to the same constraints exhibited by most current multimedia indexing systems. That is, it allows the user to submit a wide range of queries to the database but does not provide any control over the metric by which object similarity is measured.

The following section suggests an algorithm that still maintains the efficiency provided by the method described earlier while allowing the user more freedom in terms of object similarity metrics. Although it does not allow the user complete autonomy in metric formulation, it does allow for arbitrarily complex polynomial combinations of known schemes. Again, the idea is similar to that proposed by Berman[6]. Several optimizations have been made to aid with the application of the algorithm to large databases.

The general algorithm is similar to that presented in the previous section except that object similarity calculations have been extended and made more flexible. This improvement in the similarity calculations allows for an arbitrary number of *indexing schemes* to be combined. They are combined through a series of multipliers and exponents creating, what will be referred to as, a *meta-indexing scheme*, $I_M(a,b)$.

$$I_M(a,b) = m_1 I_1(a,b)^{e_1} + \cdots + m_n I_n(a,b)^{e_n} \tag{4.4}$$

Equation 4.4 shows one method for combining an arbitrary sized series of schemes to form a single metric. It shows that the *meta*-distance between any two objects can be calculated via the application of a series of *indexing schemes*. Before being added to the total *meta*-distance, the result of applying the current scheme, I_n, has an exponent, e_n, applied and then is multiplied by another factor, m_n. The advantage of this type of a calculation is that it allows for iterative

refinement of the final value.

A query to a system that implemented this method would, in addition to the query image required of the previous algorithm, consist of a series of real valued multipliers and exponents. Having received a query of this type, the first step of its processing involves the ordering of the various schemes. Firstly, any schemes that have zero valued multipliers or exponents are ignored as they will contribute a constant value of either -1, 0, or 1 to the similarity measure between all objects in question. Having disregarded any irrelevant schemes, the second step involves ordering the schemes. This ordering explains the reason for requiring that *indexing schemes* return a value constrained to be a real value between 0.0 and 1.0. Ordering involves the determination of the maximal or minimal possible contributions to the meta-distance of each of the individual *indexing schemes.*

$$\begin{aligned} d_x^{max} &= m_x^{e_x} \qquad if,\ m \geq 0 \\ &= -1 \times |m_x^{e_x}| \quad if \quad m < 0 \end{aligned} \tag{4.5}$$

For each multiplier-exponent pair, we calculate d^{max}, the maximal possible value that an individual scheme can contribute to the sum. Having made this calculation, an ordered list of the schemes is created. This ordering indicates the order in which the schemes will be applied. The ordering is as follows. The first half of the list consists of all of those *indexing schemes* with $d^{max} \leq 0$ from smallest to largest. The second half consists of those schemes with $d^{max} > 0$ sorted from largest to smallest.

$$I_M(a,b) = -3I_3(a,b)^2 - 3I_5(a,b)^{0.5} + 4I_6(a,b)^{.2} + 2I_1(a,b)^3 + 3I_4(a,b)^3 + 4I_2(a,b)^5 \tag{4.6}$$

Equation 4.6 shows an example combination with the schemes listed in the order in which they will be applied.

Based on this ordering, it is now possible to iteratively calculate the distance between two objects, refining the value at each step. The reason for sorting the schemes based on their d^{max} values as described above is that it ensures that, for every application of a scheme with

a $d^{max} > 0$, the total value of $I_M(a, b)$ will only increase. Further, schemes with higher d^{max} values are applied first due to the fact that they have a higher potential to increase the value of the distance between the two objects in question beyond the current value of B_n. This is advantageous in that, if it any point during the calculation of the similarity between a library image and a query image, $I_M(l_x, q)$, the current value of the distance exceeds B_n we can stop calculation. Stopping calculation via this method is valid as a result of the fact that on each iteration, we are adding a positive value to the distance, making it more positive.

Chapter 5

A Media Indexing System

In order to demonstrate an application of the ideas discussed in Chapter 3 and Chapter 4, a content based image indexing system was implemented. The following chapter examines this implementation. It describes a number of the system's features as well as several of the design decisions that were made both prior to and during its implementation.

The approach taken during the implementation of the image indexing system was similar to that taken by Lycos and other first generation text-based search engines. These engines parsed HTML documents, extracting keywords and using standard statistical based algorithms to determine text document fitness. These first generation algorithms provided a tool very similar to the UNIX Grep tool. Given a query, they attempted to return a document with similar words or phrases without attempting to attach any meanings or biases to words and phrases. Similarly, the following implementation provides the user with the ability to calculate and access statistical measures on images without attaching any meaning to the statistics. The system consists of both a client and server that communicate via a protocol based on the exchange of data objects.

5.1 Server

The server is a stand alone implementation that handles two major classifications of queries. The first type of query is based on the model discussed in Chapter 3. It allows the user to submit and apply unique *indexing schemes* to the database of images recognized by the server. The second type of query implements the model discussed in Chapter 4. It provides functionality allowing for the efficient application of server resident *indexing schemes*. This application, as

discussed earlier, makes use of the triangle inequality.

In order to support both types of queries, the system maintains three lists. It maintains a list of library images, a list of key images and a list of known *indexing schemes*. Both image lists contain the Internet locations of a series of GIF and JPG formatted images. In addition to this series of lists, the server maintains a matrix of sorted vectors. This matrix is indexed by each possible key image and *indexing scheme* pair. It contains the pre-calculated distances from each key image to each of the library objects based on the respective distance measure.

The server provides a number of methods through which it can be made aware of the images that it will be dealing with. In addition to methods which allow the user to submit both key and library images individually, the server provides funtionality allowing for HTTP references to HTML documents containing images. Upon submission, these documents are parsed and any relevant images are indexed.

5.2 Client-Server Communication

Access to the server is provided through a client interface. This interface defines a communication protocol which supports the exchange of objects between the client and server. There are two types of objects exchanged between the client and server. The client submits a `Query` object to the server and the server responds with a `Response` object. The `Query` object contains all of the information required for the server to process the query and produce `Response` object. Similarly, the `Response` object contains all of the information pertaining to the results of the query including any relevant error messages.

5.3 Client

Unlike the server, the client is not defined beyond the communication interface it shares with the server. A Java `Client` class has been defined but it only contains the functionality required to create a connection and interact with the server. It makes certain assumptions in terms of the functionality of an attached user interface. For instance, it assumes that the user interface

will provide all of the necessary information required to create the `Query` object. This type of implementation was chosen in order to allow implementation specific interfaces to be added as required.

5.4 Design Decisions

Prior to the implementation of the system as described above, several design decisions were made. These decisions focused on the development of an implementation that could demonstrate both the validity and effectiveness of the theory discussed in previous chapters. The decisions were influenced by three dominant factors; time and effort, portability and expandability.

5.4.1 The Java Programming Language

The first decision that was made with regards to the system's implementation was the choice of an appropriate development environment. Having examined a set of alternatives, the system was implemented using the Java programming language. The following list summarizes the features of the language that contributed to it being chosen.

- *Thread Library:* An important factor involved in the choice of Java as the implementation platform was its high level, standardized thread library. The functionality provided by the library assists with both the speed at which indexing will occur and the system's expandability. Due to amounts of traffic, bandwidth restrictions and server response times, delays can often be witnessed during the transfer and indexing of media objects. Multi-threaded implementations can diminish the effects of these delays through context switching. In addition to the performance improvements observed as a result of context switching, multi-threaded applications can witness drastic performance improvements when run on larger machines. More specifically, they demonstrate greater perfomance improvements when run on multi-processor hardware architectures than their single threaded counterparts.

- *Internet Connectivity:* One of the factors that contributed to the choice of Java as the implementation languages was its ability to interface with the Internet. The language is equipped with the full suite of high level Internet protocols. The capabilities provided by the protocols aid in the speed and ease with which applications can be developed and integrated with the Internet. This attribute was highly desirable during the selection due to the close relationship the system has with the Internet.

- *Portability:* The Internet consists of a diverse collection of hardware architectures. Java reduces the problems associated with this heterogeneity through its ability to operate seamlessly over a variety of architectures. This ability is important for a number of reasons. Most importantly it insures that both client and server software will be executable on different systems, maintaining seamless interaction over a wide range of hardware combinations.

- *Object Oriented Paradigm:* One of the most important features of Java is the fact that it strictly adheres to an object oriented paradigm. This feature was important during the design of the system for a number of reasons. Firstly, one of the most important properties of the system was that it had to be extensible. Adhering to strict object oriented design principles aided in insuring that this would be possible due to the fact that it allowed for abstract interfaces to be defined.

  In addition to the fact that the object oriented paradigm aided in the system's extensibility, it also provided a great deal of functionality which allowed for the system's distributed design. Java version 1.1 provides the user with the ability to serialise objects. Object serialisation is the process of converting an object into a form that can be transmitted to a data sink. That is, a form that can be saved to a file or transmitted to a remote destination. This functionality was useful during the implementation of index persistence and various aspects of communication.

- *Acceptance:* In addition to all of the previously mentioned attributes, one of Javas strongest is its acceptance. Although it is still at the point where it is supported to varying degrees, most popular Internet browser software including Netscape Navigator, Internet Explorer and the HotJava browser claim to support Java. It should be noted that one of the major problems encountered during development was a result of this varying support. For example, at the time of this writing, Netscape Navigator 4.0 supported only portions of version 1.1 of the Java AWT (Abstract Windowing Toolkit). This toolkit is responsible for all graphical displays - one can imagine the difficulty of implementing an acceptable GUI that will operate over multiple platforms with varying degrees of support. Despite this drawback, all major operating systems provide native Java support which allowed us to circumvent the browsers where ever necessary.

5.4.2 The `MediaDocument` and Related Classes

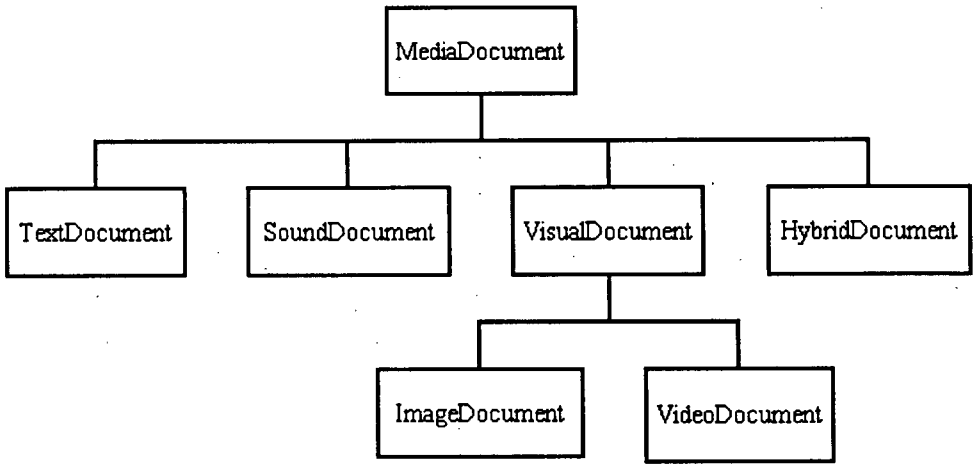

Figure 5.6: The `MediaDocument` Hierarchy

Having chosen a language in with which to develop the system, a representation for the various different types of media had to be developed. This representation had to be simple and extensible allowing for a variety of different media all sharing some common attributes. The solution to this problem resulted in the definition of an `abstract` Java class referred

to as the `MediaDocument` class. This class is the super class of all media objects to be indexed or involved with the system. The class contains information and methods common to all media objects such as location, size and name. Figure 5.6 shows the relationship between the various `MediaDocument` subclasses. Document specific information is stored at the highest applicable level of the hierarchy. For example, colour histogram information is stored in the `ImageDocument` class. Dimensional information, on the other hand, is stored in the `VisualDocument` class. The reasoning behind this organizational paradigm is that it allows for the possibility of *indexing schemes* being applied to a variety of different `MediaDocument` sub-classes.

MediaDocumentObserver

This interface is defined as a means of monitoring the status of media objects. Similar in functionality to the Java `ImageObserver` interface, when implemented, this interface allows a class to monitor the status of registered `MediaDocument` objects. The implementation of this interface provides developers with a key element of functionality when dealing with `MediaDocument` objects. It allows for the downloading of media to be done asynchronously. The interface makes use of the `documentUpdate` functions in order to notify the main thread of execution that downloading is either complete or an error has occured. Without this functionality, a great deal of time could be wasted waiting for the initialization of `MediaDocuments` to complete. The implementation of the interface allows `MediaDocument` object creation to occur inline with the current thread of execution and the actual downloading and indexing to occur elsewhere. This effectively masks any waiting that may occur, allowing the application to perform other tasks. In the case where an error occurs, the application is notified via the interface it has implemented.

5.4.3 Query Interface

One of the most important aspects of any media indexing system is the query interface. In order for a system to be used, it must have an intuitive, simple-to-use interface. The definitive

qualities of such an interface are hard to quantify. Requirements change from application to application.

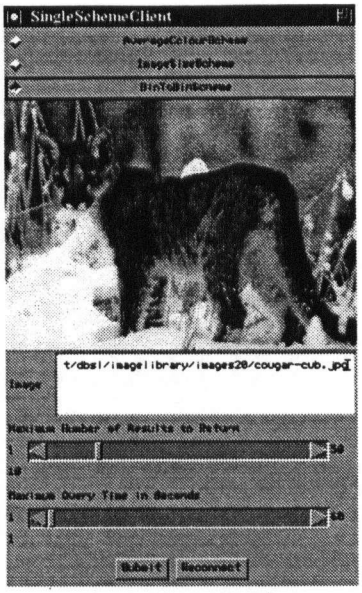

Figure 5.7: A sample user interface

Figure 5.7 shows one of the client side user interface used with the media indexing system. It is a very simple system that allows the user to select a single scheme and easily manipulate various parameters. This interface worked well for the experimentation associated with this work. It may, however not work so well in other applications.

In order to accomodate this constant variability, the `Client` object has been defined without a user-interface. Instead, it defines a class on which a user interface can easily be added. Thus, it is possible to define custom user interfaces on a per application basis.

Chapter 6

Experimental Results

One of the most important questions with regards to the methods contained within this work is how well they perform under "real world" conditions. The following chapter examines the performance of the implementation described earlier. Due to the speed at which hardware performance is currently advancing, system performance is judged based on the number of direct library-query object comparisons that have to be made for a given query. The examination is done through a series of experiments which test various aspects of the system showing exactly how well the methodology performs under a variety of conditions.

6.1 Operating Environment

All of the experiments illustrated in the following sections were performed on a single processor Intel, Pentium II running at 300MHz under RedHat Linux Version 5.0. The system had 500Mb of RAM and 9.0Gb of hard drive space. As mentioned earlier, the software system was developed entirely using the Java programming language. More specifically, using the Java Development Kit version 1.1.5 (jdk1.1.5).

6.2 Experiments

6.2.1 Media Library Size

Due to a lack of both time and resources, it was not possible to develop an index of the entire media content of the Internet. Instead, a very small subset of images from the network were downloaded and indexed. This first experiment was designed to estimate how the system behaves as the media library grows. It is believed that the portion of the database required to

be examined in order to fulfill the query requirements will decrease as the library size increases.

During this experiment, a total of 1610 distinct queries were made to the system using a single, 16 dimensional *indexing scheme* and a variety of different key images, number of images to be returned and library sizes. 5 sets of 23 queries were made requesting 5, 10, 20, 30, and 40 results with both 1 and 2 key images. These queries were made to a variety of different libraries containing 89, 190, 276, 522, 623, 809 and 913 distinct images respectively.

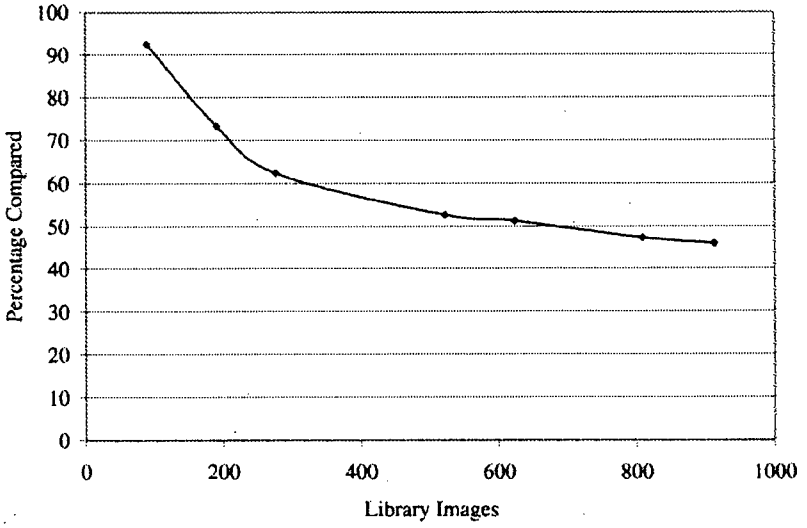

Figure 6.8: Graph showing the effects of varying the number of library images

Figure 6.8 shows a summary of all of the results obtained. The figure shows the percentage of images that are required to be examined under the various different library sizes. These percentages were obtained by calculating the average percentage of images compared under each of the conditions listed above. Thus, the percentage of comparisons made for the library with a size of 89 was obtained by taking the average of the number of images actually compared when 5, 10, 20, 30 and 40 images were requested using both 1 and 2 key images.

Examination of the graph shows a distinct trend indicating a decrease in the percentage of library images that have to be compared as the library size grows. Moreover, the graph appears to indicate an almost logarithmic relationship between the percentage of comparisons and the size of the library.

6.2.2 Number of Key Documents

One of the factors that contributes to the effectiveness of the triangle inequality is the number of key documents involved. It is believed that an increase in the number of key documents will result in changes to both the number of lower bounds checks and the number actual document comparisons made.

Firstly, a marked decrease in the number of actual similarity calculations or document comparisons that will be required to retrieve the best library documents is expected. Secondly, it is expected that as the number of key documents is increased, the number of lower bounds checks that will be made will increase relative to the number of actual comparisons. That is, the difference in the number of lower bounds checks and the number of comparisons made will become larger. The reasoning behind these hypotheses is as follows. As the number of key images increases, so does the accuracy with which they represent the data base. In terms of the algorithm itself, it would be expected that as the number of key documents was increased, the accuracy of the maximal lower bounds on the distance between the library and query document would also increase. This increase in accuracy will result in a decrease in the necessary library-query comparisons and a subsequent increase in the number of checks required to find the maximal lower bound.

In order to test the above hypothesis, a library of 522 images was compiled and indexed using an *indexing scheme* that encompassed 16 dimensions and a variable number of randomly selected key images ranging from 1 to 20. [1] A series of queries were then submitted to the server. The queries varied in both the query image and the number of images to be returned.

[1] These images were randomly selected to help illustrate a "real-world" situation where a set of keys will be used with a variety of differing indexing schemes.

Requests for 5, 10, 20, 30 and 40 images were made to the server for each of 23 different query images.

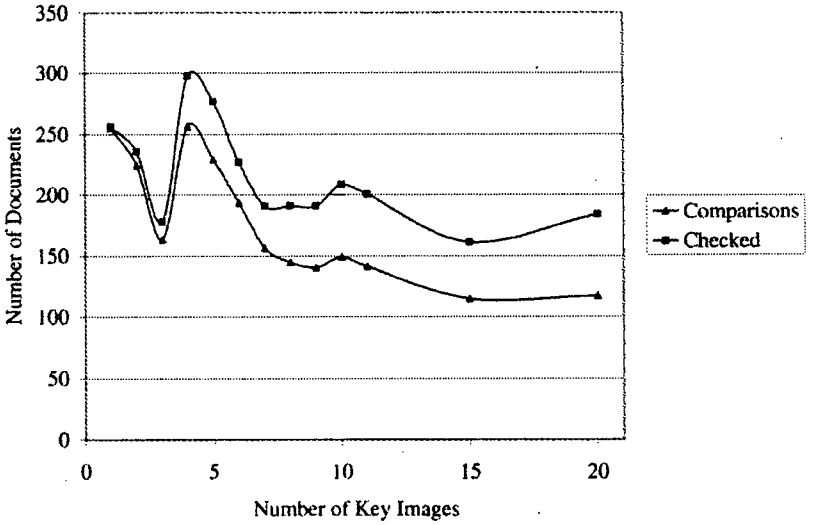

Figure 6.9: Graph showing the effects of varying the number of key images

Figure 6.9 shows a graphical summary of the results of all of the queries. Again, the values plotted on the graph represent the average values for all of the different queries made under the varying numbers of key images.

Inspection of the graph appears to show two definite trends that correspond with the original hypotheses. Firstly, it can be seen that, in general, as the number of key images is increased, the number of comparisons required to fulfill a query decreases. Closer examination reveals that the use of 9 key images resulted in a decrease in comparisons by 45% and the use of 15 resulted in a reduction of 65% over the case where only a single key image was used.

Contrary to the decreasing trend, there are also several peaks in the graph. These peaks can be noticed at the forth, tenth and twentieth key images. Upon initial inspection, one

might be lead to believe that these are examples showing that the initial theories were wrong; additional key images do not aid in the efficiency of the triangle inequality. In fact, what it does show is that although the addition of key images generally improves the performance of the methodology, poor selection of keys can result in degenerated performance.

6.2.3 *Indexing Scheme* Complexity

Another element that contributes to the efficiency of the triangle inequality is the dimensionality of the space in which it is being applied. In the domain of this system, dimensionality is a property of the *indexing scheme*. It is defined by the number media attributes encompassed by the metric. That is, for each element of a given media object encompassed by the calculation, another dimension is incurred. It is believed that, in general, the higher an *indexing scheme*'s dimensionality, the less effective the triangle inequality will be at reducing the number of required comparisons. This hypothesis is derived from the fact that the higher the dimensionality of the problem space, the lower the chance that the lower bound generated is a good approximation of the actual distance. The intuition bethind this is that every dimension involved with the metric provides another "direction" that can contribute to the distance between two objects. As the dimensionality rises, the chance that two objects are located in the same direction at the same distance from a third decreases exponentially.

Figure 6.10 summarizes the results obtained by making 920 queries to the system. The same set of 230 queries were made to each of 4 different metrics that encompassed 2, 3, 16 and 48 dimensions. The figure displays how the percentage of the library images that must be compared to any given query image varies with the complexity of the metric.

As expected, increasing the *indexing scheme*'s dimensionality increases the percentage of the library that has to be examined. Quite unexpectedly, however, the trend seems to indicate that dimensionality has less and less of an effect as it continues to increase. That is, increasing from 5 to 10 dimensions will show a greater detriment to performance than the increase from 20 to 25 dimensions.

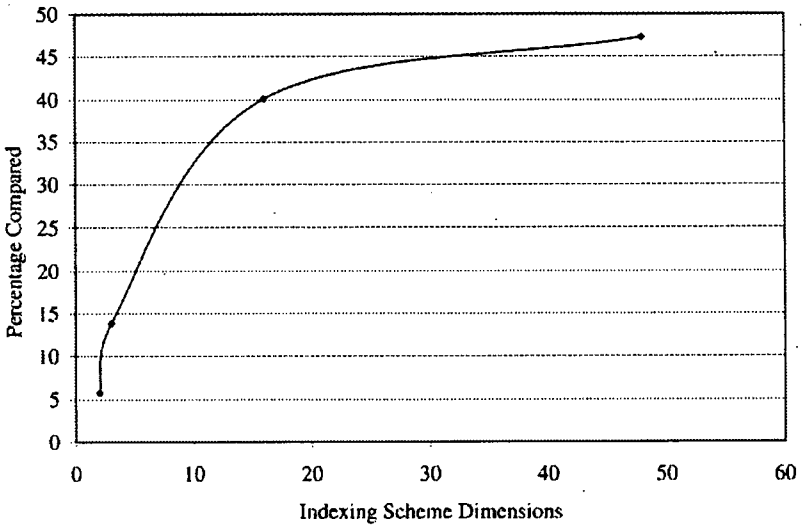

Figure 6.10: Graph showing the effects of varying the metric complexity

6.2.4 Number of Results

The following experiment examines how varying the number of results returned to the user affects the number of comparisons that have to be made between the query and library images. It is expected that an increase in the number of comparisons required to return the best images will be witnessed in conjunction with an increase in the images being requested.

A library of approximately 800 images was used for this experiment. A 16 dimensional *indexing scheme* and 2 key images were used as 23 distinct query images were submitted to the server with requests made for the best 1, 2, 4, 8, 16, 24, 32, 48 and 96 matches.

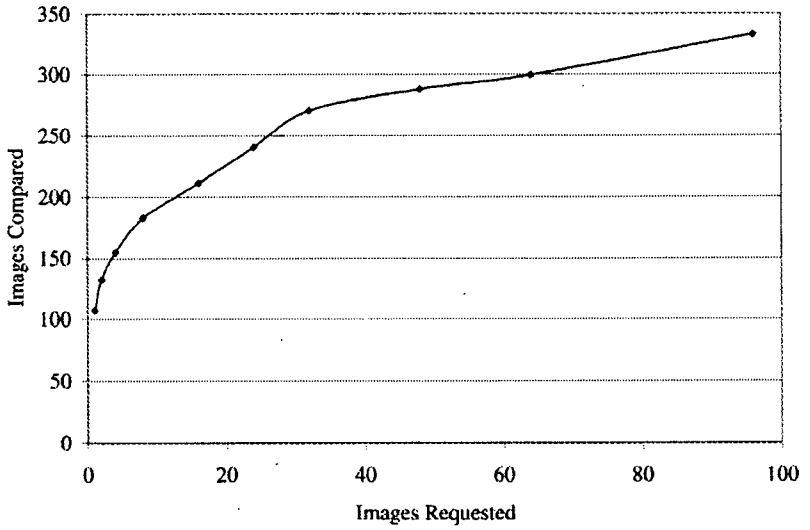

Figure 6.11: Graph showing the effects of varying the number of results requested.

Figure 6.11 shows the relationship between the number of images requested and the number of images that were compared. The graph shows a very rapid increase in the number of comparisons required as the number of images requested increases initially. In addition, it shows that this trend is not maintained as the request size continues to grow. Instead, the number of

comparisons flattens out to what appears to be an almost linear relationship.

6.3 Summary

All of these experiments have displayed that there are four key factors that affect the performance of the system; the number of elements in the media library, the number and quality of the keys used during the indexing process, the complexity of the *indexing scheme*, and the number of results requested by the user. In all cases, it was witnessed that the more that was demanded of the system, the greater the number of comparisons required to fulfill the query.

Chapter 7

Conclusions and Further Developments

7.1 Conclusions

Recently, there has been a great deal of work in the area of content based media indexing. This work has been motivated by the need to have effective and efficient access to the large bodies of media data available today, in particular, the data contained throughout the Internet. Despite all of the work that has been done, all current systems fall short in several areas. This thesis has tried to recognize and perhaps contribute to the process of overcoming these shortcomings. Where an *indexing scheme* was defined as a metric that measures either the absolute or relative quality of a given multimedia object. It explains and demonstrates an application of two pivotal concepts related to the efficient application of *indexing schemes.*

The application of unique *indexing schemes* is currently very inefficient. Generally, in order for a user to apply a unique scheme, they must, at some point, download and index all of the media that the metric will be applied to. The main problem with this is that it is inefficient in terms of time, storage and most importantly network bandwidth. The first concept discussed by the work examined the possibility of decreasing this inefficiency. The idea was that a large portion of the inefficiency could be eliminated through the provision of a software interface that allowed for the submission of unique *indexing schemes* for remote processing.

The second concept examined by the work attempted to improve the efficiency of applying *indexing schemes* from another angle. It examined an application of the triangle inequality, optimized for use with large multimedia databases. It examined methods which allowed the application of resident *indexing schemes*, both static and custom. Unlike static schemes which

encompass only a single metric, custom *indexing schemes* were defined as the polynomial combination of several distinct schemes.

The application of these two concepts were demonstrated through the implementation of a multi-threaded image indexing system developed using Java. The system implemented a software interface that allowed users to develop and submit their own, unique *indexing schemes.* The two major requirements placed on the schemes were that they had to be written in the Java programming and implement the interface defined by the `IndexingScheme` class. The system demonstrated the viability of this technique, allowing users to submit queries to a large database of images. Although the response time was highly dependent on the complexity of the scheme, responses were generally posted within 2 seconds of submission requiring orders of magnitude lower bandwidth requirements. Secondly, the system demonstrated that the described application of the triangle inequality decreases the number of comparisons required to determine optimal matches in large databases.

7.2 Further Developments

Originally, the design and implementation of the system associated with this work concentrated more on testing and data collection than on the production of a usable product. Statistics derived from the experiments performed in Chapter 6 show that there is definite room for improvement. This section discusses several modifications that would be required in order for the system to reach its full potential and develop into a fully functional product.

7.2.1 C/C++ Implementation

The entire implementation of the image indexing system was done using the Java programming language. Java was selected as the development environment of choice for a number of reasons that have already been discussed in section 5.4.1. Despite its many advantages, however, there are several drawbacks to using the Java development environment.

One of the most important drawbacks that accompanies the language is a result of the fact

that it is portable. All applications written in Java can, theoretically, be compiled once and run on any hardware and software platform. This portability is a result of the implementation of Java's just-in-time (JIT) compiler. Initially, Java source code is compiled into byte code. This byte code is then submitted to the the JIT which actually compiles the code just prior to execution. Although it has improved tremendously since it was first released, the compiler still doesn't provide the performance provided by native implementations in C/C++.

This lack of performance was not a limiting factor during the implementation of the prototype system described in this work. If, on the other hand, this system were to be implemented to its fullest extent, attempting to provide indices to millions of media documents of varying formats, the inefficiencies associated with Java would be considerable. Thus, at the cost of further development time, a useful extension to the system would be a native C/C++ server implementation that could communicate with Java clients.

7.2.2 Larger than Memory Implementation

Currently, the system was designed so that all of the data is stored in memory. The machine used during implementation and testing had 500Mb of RAM allowing for several up to one thousand images to be stored at a time. Again, in order to expand the size of the system so that it encompasses larger, real world environments, it would be necessary to develop a larger than memory implementation. That is, an implementation that automatically stored various portions of the data to disk.

7.2.3 Additional Preprocessing

One means of increasing the performance of the system is to perform additional preprocessing. Currently, the image implementation keeps track of rudimentary properties of images. Additional preprocessing such as orientation analysis, texture measures, and face detection would enhance both the performance of the system and the ease with which additional *indexing schemes* could be developed.

7.2.4 Duplicate Elimination

As has been observed by others [1], there a large number of images and other media that exist in a variety of locations across the WWW. There are two main problems with the existence of duplicates in the system. First, it is undesirable for a user to obtain multiple copies of the same media object as a result of their query. In addition to this, the existence of duplicates is expensive in terms of space and processing time. A desirable extension to the existing system might be to add some functionality to help avoid indexing the same object multiple times.

In order to implement such a system, three basic cases would have to be handled. The first type of duplication witnessed involved multiple references a single media document located at a single location on the network. This case, in fact, has been handled already. One of the attributes of the *MediaDocument* object is `location`. This attribute stores the original location of the document allowing for the prevention of further indexing.

The second and third cases are not so trivial and it is not so clear as to whether or not duplicates of this type should be eliminated. The second case involves multiple references to a single media document located in multiple locations. That is, the same file is located in different locations on the network. The third case involves the existence of similar documents in a variety of locations on the network.

The solution implemented by Agnew et al.[1] involves the local storage of thumbnails. Images are considered duplicate if their thumbnails are identical. This method has several shortcomings and questions could be posed as to how it would extend to non-image media. It does, however, present a good starting point for work that may follow.

7.2.5 Distributed Cataloging

As mentioned earlier in Section 1.2, one of the primary contributors to Internet traffic associated with media indexing is the production of global catalogs. In order for a system to maintain an index of remote media, it must constantly index and download media. This paradigm is highly inefficient in terms of the amount of network traffic that it generates and could be aided through

the implementation of a distributed cataloging system. Such a system might implement a set of protocols that allowed for media to be indexed locally as required and uploaded to a central server. The advantage of this is the elimination of useless downloads of media that hasn't changed and avoids the transmission of the media objects themselves. Only the indexed form of the object would now be transmitted.

An extension to this system might allow for the distributed processing of queries. For example a user could submit a query to a central server which would relay the query to a number of known servers each of which would process and return the results of the query. The advantage of such a system is lower processor requirements on systems as well as even lower network traffic as only media objects selected by the user as being desirable would ever have to be transmitted across the network.

7.2.6 Improving Key Object Selection

The experiments in Chapter 6 demonstrated that four main factors affect the number of comparisons required to process a query. Of these four, properties associated with only two of these factors can be altered without affecting the services provided to the user. The first of these is the process of selecting key objects.

Performance of the system is highly dependent upon the quality of the key objects selected. Currently, key object selection is done arbitrarily. This was done as it is very difficult to select keys that are applicable under a variety of different *indexing schemes.* One possibility for improving the system would be to implement an automatic key selection algorithm to be executed after all of *indexing schemes* have been selected. Another, perhaps better solution, might be to remove the existence of actual keys all together and replace them with generated keys. These generated keys would only possess the properties associated with selected schemes. A method for generating these keys could be implemented as follows. Objects could be grouped in to any number of clusters based on the qualities by which they are being indexed. Keys could then be generated based on the average values of the attributes of the elements contained in

the clusters. Generating keys in this fashion would insure their appropriate distribution and aid in providing closer to optimal performance.

7.2.7 Reducing the Search Space

As observed in Section 6.2.1, the second factor that can be altered without the end user being directly affected is the size of the media library. Although the relationship between library size and the number of comparisons is less than a linear relation, if the system were to index a body the size of the Internet, the increase in the number of required comparisons would be significant. Consequently, methods for reducing the search space would have to be installed. One possible method for reducing the space would be through the implementation of object clustering as discussed earlier. Having created the clusters, the result candidates could be reduced from the entire database to a subset based on the clustering. For example, candidates may have to meet the criteria of being at most one cluster away from the cluster which most closely represents the query object.

Another solution might be a hierarchical clustering system. Such a system might have several levels of clusters; each level containing a more specific set of elements than its parent and each cluster containing a representative (perhaps generated) key. Thus, with only a few comparisons to keys, the algorithm could reduce the size of the cluster to a reasonable size prior to applying the triangle inequality.

Bibliography

[1] Brent Agnew, Christos Faloutsos, Zhengyu Wang, and Don Welch. Multi-media indexing over the web. In *Storage and Retrieval for Image and Video Databases*, volume 5, pages 72–83, San Jose, California, February 1997.

[2] G. Ahanger and T.D.C. Little. A survey of technologies for parsing and indexing digital video. Technical Report MCL Technical Report 11-01-95, Department of Electrical and Computer Systems Engineering, Boston University, 44 Cummington Street, Boston University, Boston Massachusetts 02215, USA, November 1995.

[3] Vassilis Athitsos and Michael J. Swain. Distinguishing photographs and graphics on the world wide web. In *IEEE Workshop on Content-Based Access of Image and Video Libraries*, Chicago, Illinois, March 1997.

[4] Jeffrey R. Bach, Chris Fuller, Amarnath Gupta, Arun Hampapur, Bradley Horowitz, Rich Humphrey, Ramesh Jain, and Chiao fe Shu. The virage image search engine: An open framework for image management. In *Storage and Retrieval for Image and Video Databases*, volume 4, pages 76–87, San Jose, California, February 1996.

[5] Dana H. Ballard and Christopher M. Brown. *Computer Vision.* Prentice Hall, 1982.

[6] Andrew Berman and Linda Shapiro. Efficient image retrieval with multiple distance measures. In *Storage and Retrieval for Image and Video Databases*, volume 5, pages 12–21, San Jose, California, February 1997.

[7] John S. Boreczky and Lawrence A. Rowe. Comparison of video shot boundary detection techniques. In *IS&T/SPIE 1996 International Symposium on Electronic Imaging: Science and Technology*, San Jose, CA 94720, February 1996.

[8] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, Michael F. Schwartz, and Duane P. Wessels. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, University of Colorado - Boulder, March 1995.

[9] C.Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3:231–262, 1994.

[10] Dwi Faulus. Design and implementation of an expressive query interface/language for image database. Master's thesis, The University of British Columbia, 1996.

[11] Charles Frankel, Michael J. Swain, and Vassilis Athitsos. Webseer: An image search engine for the world wide web. Technical Report TR-96-14, The University of Chicago, Computer Science Department, 1100 East 58th Street, Chicago, Illinois 60637, August 1996.

[12] General Magic Inc. Introduction to the odyssey api. API Documentation http://www.genmagic.com/agents/odysseyIntro.ps, General Magic, Inc., General Magic, Inc., 420 N. Mary Avenue, Sunnyvale, CA, 94086, USA, 1997.

[13] Sun Microsystems Inc. *Java Core Reflection: API and Specification.* Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View CA 94043 U.S.A., January 1997.

[14] Sun Microsystems Inc. *Java Object Serialization Specification.* Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View CA 94043 U.S.A., February 1997.

[15] Sun Microsystems Inc. *Java Remote Method Invocation Specification.* Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View CA 94043 U.S.A., February 1997.

[16] Jianhao Meng, Yujen Juan, and Shih-Fu Chang. Scene change detection in a mpeg compressed video sequence. In *IS&T/SPIE Symposium Proceedings*, volume 2419, San Jose, CA, February 1995.

[17] Baback Moghaddam, Wasiuddin Whaid, and Alex Pentland. Beyond eigenfaces: Probabilistic matching for face recognition. Technical Report 443, MIT Media Laboratory, 20 Ames St. Cambridge, MA 02139, USA, April 1998.

[18] Wayne Niblack, Ron Barber, Qill Equitz, Myron Flickner, Eduardo Glasman, Dragutin Petkovic, Peter Yanker, and Christos Faloutsos. The qbic project: Querying images by contenet using color, texture and shape. Research Report RJ 9203 (81511), IBM Research Division, IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California, 95120-6099, February 1993.

[19] Henry A. Rowley, Shumeet Baluja, and Takeo Kanade. Rotation invariant neural network-based face detection. Technical Report CMU-CS-97-201, Carnegie Mellon University, Pittsburgh, PA 15213, December 1997.

[20] Drew D. Saur, Yap-Peng Tan, Sanjeev R. Kulkarni, and Peter J. Ramadge. Automated analysis and annotation of basketball video. In *Storage and Retrieval for Image and Video Databases*, volume 5, Princeton, NJ 08544, February 1997.

[21] Stan Sclaroff. World wide web image search engines. In *NSF Workshop on Visual Information Management*, Boston, MA, June 1995.

[22] Stan Sclaroff, Leonid Taycher, and Marco LaCascia. Imagerover: A content-based image browser for the world wide web. In *IEEE Workshop on Content-based Access of Image and Video Libraries*, Boston, MA, June 1997.

[23] John R. Smith and Shih-Fu Chang. An image and video search engine for the world-wide web. In *Storage and Retrieval for Image and Video Databases 5*, pages 84–95, San Jose, California, February 1997.

[24] Michael J. Swain, Charles Frankel, and Vassilis Athitsos. Webseer : An image search engine for the world wide web. Technical report, The University of Chicago, Computer Science Department, 1100East 58th Street, Chicago, Illinois 60637, August 1996.

[25] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3:71–86, 1991.

[26] David A. White and Ramesh Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California, San Diego, 9500 Gilman Drive, Mail Code 0407, La Jolla, CA 92093-0407, July 1996.

[27] Jim White. Mobile agents white paper. White Paper http://www.genmagic.com/agents/Whitepaper/whitepaper.html, General Magic, 420 N. Mary Avenue, Sunnyvale, CA 94086, 1996.

Appendix A

Definition of the Triangle Inequality

The triangle inequality states that the distance between any two points cannot be greater than the sum or less than the difference of their individual distances to a third point.

Thus, given three points A, B, and C.

$$|AC| - |BC| \leq |AB| \leq |AC| + |BC|$$

Appendix B

Locations of Various Online Media Databases

- *QBIC:* Created by a group of researchers at IBM's Almaden Laboratory, this system can be accessed via its home page at `http://wwwqbic.almaden.ibm.com/`.

- *ImageRover:* Stan Sclaroff developed the *ImageRover* at Boston University. There are a number of papers and a working demonstration that can be accessed at `http://www.cs.bu.edu/group` There is a number of things available from this link including access to related papers as well as the online demonstration.

- *WebSEEk:* Based at Columbia University, this system was developed by John R. Smith and Shih-Fu Chang and is accessible at `http://disney.ctr.columbia.edu/webseek/`.

- *WebSeer:* Originally based at the University of Chicago and implemented by Michael J. Swain, Charles Frankel, and Vassilis Athitsos, the *WebSeer* project is no longer accessible.