eyeShot Multimedia Search Engine
"Using text patterns from the WWW to characterize Multimedia Objects” *

Demetris Zeinalipour-Yazti  Theodoros Foliis
Dept. of Computer Science
University of California
Riverside, CA 92507, U.S.A
{osyiazti,folias}@cs.ucr.edu

Abstract. The evolution of the Multimedia Technology and the advancement in public networks in the last recent years has increased vastly the volume of the WWW with multimedia resources such as images, audio files and video files. On the other hand the limited capabilities of the HTML Markup language to describe efficiently these resources has made the task of finding them a difficult endeavor. Moreover the meta-information encapsulated in most multimedia objects is too poor to characterize resources. As a result, most multimedia objects are uncovered and unreferenced by most search engines. Although research has been conducted on that area for quite a long period, most solutions tend to be too expensive in terms of processing and are not scalable in any terms, since they make heavy use of content-based analysis. Another problem that arises from the use of proprietary content-based analysis solutions is that they are not reusable on different file formats. Only recently we have seen commercial search engines such as Google making efforts of indexing efficiently such Objects, without the use of "heavy" content-based analysis. In this project, we present the architecture and the implementation of eyeShot, a multimedia Search Engine which only makes use of the textual properties which are found in Documents in the WWW. eyeShot describes effectively the content of Multimedia Objects by applying a group of algorithms with which we can identify the interrelation of an Multimedia Object and the three different types of information that can be found in WWW Documents. It uses the In-Page, Surrounding and Citation Properties of an Object in order to describe and index it. We discuss our Processing Model and then provide an in-depth explanation of various systems' components design and implementation.

1 Introduction

The increased use of the Internet over the past decade has proven to be the stimulant factor that led to an immense amount of information currently available on the World Wide Web. Furthermore, the advent steps of technology as far as multimedia are concerned, turned out to be a key issue in providing better interactivity and user interfaces aiming at a greater volume of information to become easily accessible and digestible by most users. Although until the mid 90s all efforts were concentrated in providing textual information on the web, the repetitive and consecutive use of extensions in supported multimedia objects over the web changed the content analogy of an html document. In the beginning of 2001, according to Inktomi [12], the total number of pages on the web was estimated to be 1 billion. Currently, the total number of pages on the Internet has surpassed 2.1 billions, according to Cyveillance.com [10], and more than 7 million new pages are being added each day. According to the same report, the average web page is 10KB in size, contains 23 internal links and 5.6 external links, and 14.4 images. This overwhelming growth of the www has caused many difficulties to internet users since there is a lack of search engines capable of answering most of users requests efficiently. According to a study commissioned by Keen.com [11], search engines are essential to daily life. The study has found that Americans need answers to an average of four questions every day and that the Internet is the top source of answers to those questions. About 31 percent of those surveyed said they turned to the Internet and search engines first in their quest for answers. Unfortunately, those who used search engines said they were often frustrated as about half of their searches failed to return useful information. According to the same study, Americans would be willing to pay USD14.50 per week to find the right answers to questions.

* Project under development at the University of California - Riverside, Dept. of Computer Science , Fall 2001.
2 Related work

The very first efforts in the development of search engines that would provide users with the ability to retrieve specific web resources were based on text retrieval. In 1996 a new wave of search engines was introduced with the implementation of Webser [2] and Webseek [3], two image search engines that combined both the textual information surrounding an image in an html document and the results of an image content analysis to characterize and classify various images. Webser system was supplementing keywords related to an image with information gathered through an image analysis procedure in which different algorithms were applied on the content of each image. The system featured classification of images between photos and computer drawings and further used specific algorithms for face recognition in a photo. Using the results of these algorithms along with information regarding file size, image dimensions, color depth, file name etc. of an image, Webser provided parameterized search options in image retrieval over the web. Based on the same notion, that text keywords may not accurately or completely describe image content, ImageRover [5] was introduced in 1997. ImageRover provided a more content specific approach in which searches of web images could be based directly on the image content. The system was consisted by a series of image analysis sub-modules each of which performed a different content analysis: color distribution, texture, orientation, faces and other image properties. Additionally, the system supported image queries by example, using calculated image properties for a specific image as a guideline in the process of identifying similar patterns. As far as audio and video retrieval is concerned, recent advances in speech recognition resulted in solutions like the SpeechBot search engine which was presented in 2001 by Compaq's Cambridge Research Laboratory [1]. SpeachBot indexes streaming media files based on their content using automatic speech recognition technology to transcribe and index documents that do not have transcripts or other content information.

In all these approaches content analysis and classification is based on algorithms and techniques specially designed for each particular media type. Given the exponential growth of new resource formats targeting the www, the development of such systems offers proprietary solutions that are resource dependent. Furthermore, and as far as content analysis is concerned the process of information retrieval focused only on the content of a specific resource tends to be the bottleneck of these systems. Especially when scalability along with efficiency are targets of the system, content analysis increases the cost of indexing and in many cases this cost overrides the benefits of such an implementation. Recently, Google [9] presented an image search engine based on the successful PageRank [6] rating method. It provides efficient results limiting content analysis only on very few attributes like the image size, type and color, saving time and cost during indexing and searching.

In our approach we argue that the www was designed for text and we consider that all other resources have a supplementary role. Furthermore, we argue that if it is possible to build a system with the same efficiency as one of the proprietary systems described above but without the cost of a content analysis, then it is preferable to skip the content analysis step and built a resource-independent system capable of satisfying any new multimedia resources that may appear in the future.

3 The Architecture of eyeShot

In this section we present the architecture. eyeShot Architecture (see Figure 1) consisted of a) eyeShotBot, a high performance Crawler written in JAVA which recursively fetches resources located on the WWW and b) eyeShot Index Server which is in charge of processing downloaded documents, identifying significant patterns and relations of multimedia objects and text keywords in a page. According to that information
the Index Server can assign a weight to this multimedia object. The Index Server is moreover responsible for building and maintaining various persistent data structures and hashtables. The final goal of the Index Server is to come up with an Inverted Index where each keyword points to one or more related multimedia objects. The Inverted Index will be generated with the use of the offline Index Generator. Finally web users may be able to access and query the system by performing searches through the online search interface of eyeShot.

**Fig. 1.** eyeShot Architecture.

4 eyeShotBot - A High Performance Crawler

A crawler also known as spider, robot, or bot is a program that traverses the hypertext structure of the Web automatically, starting from an initial hyper-document and recursively retrieving all documents accessible from that document. Typically every crawler takes a list of "seed" URL's as its input, and repeatedly executes a set of steps, which are: It initializes the list of seed URL's and pops a URL out of the URL list. Then, it determines the IP address of the chosen URL's host name, opens a socket connection to the corresponding server, asks for the particular document, parses the HTTP response header and decides if this particular document should be downloaded. If this is so, the crawler downloads the corresponding document and extracts the links contained in it; otherwise, it proceeds to the next URL. The crawler ensures that each extracted link corresponds to a valid and absolute URL, invoking a URL-Normalizer to "de-relativize" it, if necessary. Then the normalized URL is appended to the list of URLs scheduled for download, provided that this URL has not been fetched earlier. Web crawlers are the key component of services running on Internet and providing searching and indexing support for the entire Web, for corporate Intranets and large portal sites. More recently, crawlers have also been used as tools to conduct focused Web searches and to gather data about the characteristics of the WWW. eyeShotBot is a High Performance Crawler which is based on the WebRACE Crawler [7]. WebRACE Crawler is comprised of two basic components, a *Mini-crawler* which downloads, processes and stores a page and the *Object Cache* which indexes downloaded resources. The WebRACE Crawler has the ability to notify a Filtering Processor,
through a permanent socket link, every time a page is downloaded and cached. This gives us the opportunity to implement a task specific Annotation Engine which operates independently and asynchronously (see Figure 1). We have added several new features to the WebRACE Crawler, such as URLfetcher Number Adjacoment Mechanism, a Monitor Thread which supervises the operation of the URLfetchers and provides basic resource management as well as several others improvements. With our optimizations the crawler became more resistant to system failures, while they also increased the total throughput. More specifically we were able to complete the crawl of approximately 2,000 pages in 1 minutes resulting in a total throughput of 33 pages/sec while the initial throughput of the WebRACE crawler was approximately 10 pages/sec. Describing the performance improvements applied on the WebRACE Crawler is out of the scope of this paper.

5 eyeShot Annotation Engine

The eyeShot Annotation Engine is the core and most important part of the eyeShot System Architecture. In subsection 5.1 we are going to give the concept behind our implementation and in section 5.2 we are going to provide implementation details. Although we are going to expand more on the indexing techniques we would like to mention that our ranking solution is based on a simple cost-effective "keyword weight" Scheme. More specifically we intend to build an Object Index (See Figure 2), where each URL will be described by several (keywords, weight) sets. Every time a new keyword that may be related to a particular resource is found, it is added along with the appropriate weight into the table. If the keyword already exists in the table then the value of the weight is increased by the value of the weight of the Object we are trying to add. For efficiency the keywords of each URL are added in another hashtable in order to access and modify its content easily. More details will be given in subsection 5.2.

![Object's Index (Image's URLs)](image_url)

**Fig. 2.** Object Index is the Output of the Annotation Engine.

5.1 The Concept behind Processing

The main idea behind our approach is that since every Object that is linked from a document inherits the document's textual properties, we will be able to describe that object in terms of keywords. Our solution is based only on the textual properties which may be extracted from 3 different categories: a) In-Page Properties, b) Surrounding Properties and c) Citation Properties.

a) In-Page Properties, are properties found in the document which contains the links to the Objects. These properties in fact give as a general idea of the context under which these Objects are referred. For this reason the In-Page Properties are inherited by all the Objects which are referred from a particular Document. Usually In-Page Properties can be found in the following text properties of a Document:

- All Keywords that appear in the Document.
- *The Meta Keywords* that may appear in the source code of the Document.

- *The Title* of that particular Document which usually describes concisely the "Entity" that is present in that Document.

- *Captions & Emphasized Text*, will usually generate a different visual effect to a human eye. They also many times create a hierarchy in a Document, since they describe the most important aspects of that Document. This information is potentially useful and in fact is utilized only by a few search engines. These properties must be taken into consideration in order to characterize the Objects of a given Document in the best way.

Consider the example of figure 3. In this Document we can clearly observe that the text is the property that provides information relevant to the Entity "Michael Jordan". We can give a concise description of the Object contained in that page (his image) by making use of the In-Page Properties. Some of the identifying attributes are the title and the captioned text. In that way, the Objects (images) of this page could be described by the "important" keywords of that page. Of course the other two images, which are in fact advertisements, will also inherit these properties. To resolve this problem we will have to make use of the Surrounding Properties, which are the textual properties located in the vicinity of each Object. Although our example refers to Image-Objects we could actually extend this idea to any kind of other Object such as *Audio-Object* (e.g. an interview) or *Video Object* (e.g. a part of a game).

![Fig. 3. Example of a Document which has references to Objects related to the Entity "Michael Jordan".](image)

*b) Surrounding Properties*, are the properties that will certainly differentiate the description of a particular resource. Surrounding Properties are found in the surrounding area in which the Object appears in a given Document (See Figure 4). We recognize that the extraction of the correct attributes may be difficult many times especially if the Objects are placed in different positions with the use of HTML tables. On the other hand though a "smart" parser may identify what text is actually near to the extracted objects. Surrounding Properties include keywords found in the following text properties of a page:
- **The Object Filename**, is many times a good way to identify the key property that characterizes that resource. Many times though it may also become an overhead, since the filename is often chosen arbitrary and is not useful at all.

- **Link Text**, is the text that appears on the text link (if any) that links to that Object, e.g. "My friend Chris in New Jersey". This information is extremely valuable since it gives a text description of the linked Object. Moreover since the author of the given document expects the visitor to understand something potentially useful, which may also lead him to click the link, we must utilize this information.

- **Other Description Fields**, such as the "Alt" property in a citation to an Image-Object. These properties are also very important since they usually represent a concise description, by the author of the Document, of the particular resource.

- **The Surrounding Text**, within a pre-specified range area, such as 20 words before and after that image, would give us a good understanding of the context in which the particular resource was cited.

Consider again the example of figure 3, we can see that the importance of the two advertisements would fade out since there are only a few and irrelevant (to Michael Jordan) Surrounding Text. Furthermore the Link Text, the Other Description Fields and the Object Filename are not adding anything towards the Entity "Michael Jordan". Hence, they will receive less importance than the Entity’s picture which in fact satisfied most of the above attributes.

The problem that arises now is that there might exist another Document on the WWW which contains information regarding the same Entity we are describing in the example. This page might belong to a portal with less importance than the CNN.com network. Thus, their image of "Michael Jordan" should also have a smaller importance than the image of this Document. For this reason we are going to need the Citation Properties which are described in the next subsection.

![Fig. 4. The Analysis and the Weighting Scheme of an HTML page.](image)

\[\text{The funny cat} \]

b) **Citation Properties**, are the text properties that can be found in the vicinity of a Link that cites to a Document which contains the Objects we are trying to describe. Our purpose is to define a simple and cost-effective mechanism with which a certain Document’s authority can be inherited by other Documents that it is citing. In this way the Objects cited from well-accredited Documents, would also become important. As far as solutions are concerned there are several approaches, like the PageRank algorithm. The Citation Properties will be inherited again by all the objects that are referenced from a certain Document. We haven’t done any progress on the Citation Properties and hence they are out of the scope of this paper.
It is important to mention that the more documents cite to a particular Object (e.g. an image) the better description we will get for that Object since the most important keyword description will fade out the irrelevant keywords that were added accidentally. So if 100 pages have a link to the image of Michael Jordan in the previous example, then we would have a very clear picture of what is the exact content of that image.

5.2 Implementation Description.

The eyeShot Annotation Engine processes documents that have been downloaded and cached in the Object Cache of eyeShotBot. Its purpose is to process gathered documents and try to describe, in terms of keywords, the various Objects of the given page. Figure 5 illustrates the set of steps that are taken every time a notification arrives from eyeShotBot.

Fig. 5. The eyeShot Annotation Engine.

In step 1, we are illustrating the communication message sent by eyeShotBot. The communication messages arrive through a permanent socket communication link that is established during the initialization of eyeShotBot. Every message includes two attributes: a) Link, which is the url that was downloaded and b) Mime-Type, which represents the Mime-Type of the particular document. The Link property is needed by the Annotation engine so that it can fetch the appropriate document from the eyeShotBot Object Cache. Since addressing is achieved with hashing, a URL fetcher can easily compute the hashcode of the url and find the filename of the appropriate url. The Mime-Type on the other hand allows the Filtering Processors of the Annotation Engine to select and apply the appropriate filter. So if for example the Annotation Engine receives a "text/html" request, it will then identify that it has to apply a Filtering Processor which processes HTML. In our current implementation we have only implemented a "text/html" Filtering Processor which is capable of extracting semantics out of an html pages. We could extend our idea and implement new filters such as an "application/pdf" (Acrobat PDF) or "application/msword" (Microsoft Word) which would extract semantics information out of these file formats. The Annotation Engine loads the available filters from secondary storage /conf/filters.conf during startup. The Mime Types that are not identified in /conf/filters.conf are not processed any further since for the particular processing task we are not interested in analyzing these resources.

In step 1 we can also see that on each message arrival the system updates the Object’s Validation Index (OVI). Finally, OVI allows us to identify which of the Objects, we are describing, are not broken or bad links. This happens because during the extraction phase of URL’s we can’t know a’ priori if the links exist in reality. OVI is used by the Inverted Index Generator which we will describe later.
In step 2, the Annotation Engine en-queues the request in a SafeQueue Data Structure [7], since the rate of incoming requests might be larger than the rate of serviced requests. SafeQueue is a JAVA implementation of a typical FIFO queue which incorporates support for persistency, overflow control, disk caching, multi-threaded access and fast indexing to avoid the insertion of duplicate QueueNode entries. As soon as one of the working filtering processor becomes available it pops a QueueNode out of the Queue, loads the appropriate file from the filesystem into main memory and proceeds to step 3.

In step 3, we can see the involvement of several concurrent Filtering Processors. A Filtering Processor (FP) is the component responsible for loading a document from cache, uncompressed the document and analyzing a document according to the three different Processing Techniques (In-Page, Surrounding and Citation Properties). As we have mentioned before, we have content specific FPs, which means that a different processing technique will apply to different content types. Each FP works as a pipe (See Figure 6).

![Diagram](https://via.placeholder.com/150)

**Fig. 6.** The "text/html" Filtering Processor Data Flow Diagram.

of several filters, where each FP, works completely independently and asynchronously from its co-workers. We can see that the "text/html" FP will on-the-fly remove any unnecessary items (step 1-4), such as Comments, Tags, Special Characters such as spaces and finally convert the file to a lower-case character set. After that it will tokenize the remaining text (step 5) according to several characters (e.g. ’ ’, ', ', '#', '+', '-'). This wide-based tokenization may give as a wider set of keywords. For instance the keyword "Mary-Ann" would be splitted into 2 keywords "Mary", "Ann". Step 6 illustrates the Common Words Remover which will eliminate keywords that are too common in general and which shouldn't be included in indexes. Such keywords are for instance "is", "a", "there". Our common-words dictionary includes approximately 400 words and characters which were identified as too common. These keywords can be configured through /conf/blockexicon.txt. Finally in step 7 we are counting the occurrence of each of the keyword. Each keyword occurrence is multiplied by an offset in order to specify at runtime how important should a "general-keyword" become during processing. For our experiments we have configured the offset of the "general-keywords" to 1.

After extracting the general-keywords from a given document we are going to locate and extract the URL of all the Objects which are cited from this document (step 8). All these URLs are going to inherit the In-Page Properties. In this way an object receives a general characterization. Right after this step we locate the vicinity of each URL and extract the Surrounding Properties. The Surrounding Properties are weighted in a different way, than the In-Page Properties, since they provide a more precise description of the object we are trying to describe. In our experiments we have configured the Filtering Processor to
multiply the weight of an image "Alt" property by 50, the Surrounding text in a length of 80 keywords (back and forward) by 20 and the Image Name by 100. All the weights are then, in step 10, merged with the In-Page Properties. This means that we will come up with a small subset of the Object's Index (See Figure 2). Finally this subset will be merged to the Object Index, which is the point where all the Filtering Processors submit their results. All operations on the Object Index are blocking, since we have multi-threaded access, but in a newer version we will provide a non-blocking version of the Index. Although resource management and thread scheduling of each FP is left to Java's runtime system and the underlying operating system, we have implemented a simple monitoring scheme which allows us to supervise the operation of the various FPs. The number of active FPs can be configured during the initialization of the Annotation Engine /conf/fprocessor.conf.

6 eyeShot Inverted Index Generator

The Inverted Index Generator is a simple Java program, of approximately 150 lines, which loads into main memory the Object's Index and converts it to an Inverted Index. As we can see in figure 7, the Object's Index consists of a hashtable where each index (or URL) points to another hashtable of \{keywords,weight\}. It is obvious that such a structure is not so helpful when we are trying to search for a particular keyword. Hence, we are transforming it into the Inverted Index, where each index (keyword) points to the URL that actually contained the keyword. Each index of the Inverted Index consists of a \{url, weight\} sets. In this way it is easy to find the URLs that contain a specific keyword. The transformation of the Object's Index to the Inverted Index is an offline procedure, which invokes that it is not done during the crawling and the indexing phase, but only when we finish with these operations. Another detail, is that the Inverted Index Generator makes use of the Objects Validation Index (OVI) in order to fine-grain the indexed results, by identifying and discarding URLs that are either bad or broken. Bad or broken URLs are simply identified if they do not exist in OVI. In our largest experimentation which was conducted on the cs.aer.edu domain we crawled approximately 30,000 documents. The Image Index resulted in a total size of 5,000 images (gif, jpeg, png, tif). This experimentation lasted approximately 25 minutes, since the crawling was done at a rate of 30 requests/second. The transformation of this index into the Inverted Index took less than 1 minute and resulted in an Inverted Index of averagely 16,000 case-insensitive words. The total size of the Inverted Index, on secondary storage, was 8.5Mbytes compressed or 120Mbytes uncompressed. Of course the Inverted Index size is prohibitive for real deployment but we have to mention that no optimization was done on it. We are currently working on shrinking the Inverted Index size by the use of the secondary storage.

7 eyeShot Querying System

The eyeShot Querying System consists of a web page where a user can enter several keywords and receive search results that satisfy his query. The current implementation includes only search in conjunction, where

![Fig. 7. The Inverted Index (Lexicon).](image-url)
all keywords have to be satisfied by the search results. We are currently also working on advanced search capabilities where the system will actually be able to answer several other types of queries, such as search in disjunction where one or more of the keywords must be satisfied as well as search extract-phrase and search case-sensitive keywords. All these different searches can easily be deployed on the top of our current design, since additional information that might be required to satisfy the above query types can be fetched from the secondary storage.

Figure 8 presents the query analysis process that takes place every time the system is asked to answer a specific query. The Inverted Index is loaded in main memory and is shared among the several concurrent HTTP requests that may exist. The process again works as a pipe of filters where we will finally come up with the exact keywords we have to lookup in the index. In step 1, we will split the query into tokens.

![Diagram of Query Processor]

Fig. 8. The Query Analysis Process and the Generation of Search Results.

The split-characters used are identical to the characters that were used during the indexing phase. In this way we are sure that if somebody searches again for "Mary-Ann" then the actual query performed will be ('"mary", "ann") In step 2, we are removing common words, exactly as we did in the indexing phase, since anyway these keywords won't exist in our index. Step 3, is the actual keywords lookup which will return us several Inverted Index records. Each record contains a keyword and a hashtable of (url, weight). The Results merger in step 4 will join the results and sort them based on the weight field. The merger uses quicksort for sorting the results. Finally in step 6 the results selector will decide which records should be returned. In the case the user requests to see the second page of the results and each page contains actually 5 results then the Results Selector will format into html the 5th until the 10th record. Although the whole process might seem as a quite lengthy procedure, we must mention that a query on our 5,000 images index requires always less than 5 milliseconds. Moreover it is important to mention that even if the index contained a very larger set of images we could satisfy the requests in similar time. This happens due to the fact that a hashtable lookup takes usually only O(1) if the hashing function produces satisfactory hashcodes with low collision. In fact the JAVA v1.3 String.hashCode() function produces very satisfactory hash-codes in contrast with previous versions where too many identical hashcodes where generated.

7.1 System Configuration.

The eyeshot system is currently running on our Department Apache Web Server and can be found at (http://www.cs.ucr.edu/~csyazi/eyeshot/). The system is running along with a Tomcat Servlet Engine [8]. The Servlet Engine allows us to serve dynamic web pages (i.e. query results). The machine that hosts the Tomcat Server is a Quad Xeon 550MHz with 2GB of RAM. Initially we had our system configured on a Allaire JRUN 3.0 Servlet Engine but the evaluation license allowed us to have only 3 concurrent
requests on the server, which subsequently implies that our system would not be available to a large number of users. Furthermore the Tomcat Servlet Engine has been proven much faster than the Allaire JRUN Engine, since it could instantly response to our requests without delaying them, while JRUN on the other hand delayed for several seconds’ requests if the server was somehow loaded. We definitely believe that our system is an open system since it is based on only open source code projects and architectures.

8 Conclusions & Future Work

In this paper, we presented the architecture and implementation of eyeShot, a Multimedia Search Engine. eyeShot that uses WebRACE high performance crawler to gather, save and index resources from the WWW. eyeShot further consists of a filtering processor, an Inverted Index Generator and the eyeShot Web Querying Interface. The whole system is written entirely in Java. Our experiments showed that Objects that are cited from web pages (such as images), can efficiently described only by the In-Page, Surrounding and Citation textual Properties of an html page.

To assess the robustness and success of our system we have performed several crawls on many different domains. Our System worked efficiently and with no failures when crawling local Webs in our LAN and University WAN, and the global Internet. We are currently working on optimizing several modules of our System in order to perform a large scale crawling and indexing. More specifically we are optimizing the Object Index and the InvertedIndex, since they reside solely in memory. We moreover intend to move several parts of the index into secondary storage and make use of Loader-ThreadGroups, which will provide us concurrent access to secondary storage, every time we might need access to one or more pieces of information stored on disk.

References

8. The Jakarta Project http://jakarta.apache.org/