2004

Development and Performance Evaluation of a Real-Time Web Search Engine

Burr S. Watters IV
University of North Florida

DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME WEB SEARCH ENGINE

by

Burr S. Watters IV

A thesis submitted to the
Department of Computer and Information Sciences
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES

December, 2004

The thesis "Development and Performance Evaluation of a Real-Time Web Search Engine" submitted by Burr Sells Watters IV in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

Signature Deleted

12-02-04

Dr. Behrooz Seyed-Abbassi
Thesis Adviser and Committee Chairperson

Signature Deleted

12/2/04

Dr. Yap Chua

Signature Deleted

12/3/04

Dr. Arturo Sanchez-Ruiz

Accepted for the Department of Computer and Information Sciences:

Signature Deleted

12/6/04

Dr. Judith Solano
Chairperson of the Department

Accepted for the College of Computing, Engineering, and Construction:

Signature Deleted

1/11/05

Dr. Neal Coulter
Dean of the College

Accepted for the University:

Signature Deleted

12/15/04

Dr. Thomas Serwatka
Dean of Graduate Studies

ACKNOWLEDGEMENT

I wish to thank my family for their loving support throughout this last accomplishment of my academic career. In addition, I thank my friends, both here and abroad, for their constant words of encouragement through some of the difficult and tedious times. I thank Dr. Behrooz Seyed-Abbassi for his directions and guidance to a goal which I thought was unattainable.

Lastly, I dedicate this thesis in honor and in memory of my father, Burr S. Watters III.

Contents

Figures

Tables

Abstract

As the World Wide Web continues to grow, the tools to retrieve the information must develop in terms of locating web pages, categorizing content, and retrieving quality pages. Web search engines have enhanced the online experience by making pages easier to find. Search engines have made a science of cataloging page content, but the data can age, becoming outdated and irrelevant.

By searching pages in real time in a localized area of the web, information that is retrieved is guaranteed to be available at the time of the search. The real-time search engines intriguing premise provides an overwhelming challenge. Since the web is searched in real time, the engine's execution will take longer than traditional search engines. The challenge is to determine what factors can enhance the performance of the real-time search engine.

This research takes a look at three components: traversal methodologies for searching the web, utilizing concurrently executing spiders, and implementing a caching resource to reduce the execution time of the real-time search engine. These components represent some basic methodologies to improve performance. By determining which implementations provide the best response, a better and faster real-time search engine can become a useful searching tool for Internet users.

Chapter 1

Introduction

During the early 1990s, the World Wide Web (WWW) received its start from many organizations as a means to communicate with the public. For the previous twenty years, the government, its agencies, and large universities relied on the network to relay data across the country in a matter of minutes, and eventually seconds. As the Internet became more commercially and publicly based, Internet users had very few resources to find information on the web in a timely fashion. The resulting solution was the creation of online search companies that allowed people to traverse the web for business or pleasure to search the web on any topic.

In its most basic format, the WWW is a massive repository of semi-organized information. There is not a day that goes by where someone is not adding, updating, or changing its content. With so much information available at our finger tips, search engines provide a way to search this massive repository of information. To take advantage of the vast amount of knowledge and data available on the WWW, researchers are continuously exploring and developing new methodologies of searching for content relative information.

Through web search engines, companies, such as Lycos, AltaVista, and WebCrawler, have enhanced the online experience by making websites easier to find. In describing the development of Lycos, Michael Mauldin pointed out that the major functions in the operation of a web engine include gathering hyperlinked pages across the web into an index, storing the pages in a database repository, and returning the hyperlinks to web pages based on keywords, phrases, and ideas matched against the database of indexed pages [Knoblock97].

According to Filman and Pant, "the search engine industry has evolved two dominate ways of finding information: directories and spiders" [Filman98, page 21]. Directories are maintained by people who categorize web sites, which are primarily submitted by outside sources, into different subject areas based on their content. It can be "labor-intensive" to process the pages and to change categories if the page content is modified [Filman98]. Spiders (or bots) are an alternative to directories. Most of the spiders that scour the web work in completely different ways, though they accomplish the same task of searching for information [Knoblock97].

The goal of the search engine is to allow the user to quickly find a web page that pertains to the user's topic of interest. More often than not, search results from a search query could contain many web pages that are not relevant to the search. Either the search engine returns a link that is dead, or the contents have been changed from the time the page was indexed.

A solution to address these problems is to create a dynamic search engine whose purpose is to search for web content in real time. The objective is to provide the user with web content that is consistent and relevant to a user's needs by searching a dynamic area of web pages in real time. The concept also revolves around searching in an area of the web containing pages with similar content. The process would start by choosing a page to become the authority page of the group of pages. The authority page, referred to as a root page, should be a reference page to other pages of similar content. The process would continue by searching the authority page for content and then traversing the authority's hyperlinks for additional searching. This would allow the user to find information across several pages that are related via hyperlinks. Searching in real time would eliminate the problems of retrieving dead links and provide a better chance of retrieving more relevant content for the web surfer.

The purpose of this research is to provide the user with web content that is consistent and relevant to a user's needs by searching a dynamic domain of web pages in real time. The focus is on the development of a real-time search engine and measuring the performance of the different implementations of the engine. Testing the best performance relies on a couple of component implementations.

The first component is the method of traversing the hyperlinks to discover new content. This looks at such techniques as breadth-first and depth-first searching. A more unique method known as a pruning searching is also examined.

The second component focuses on two implementations of quickly retrieving and searching information from a web page in real time. The first method is a total web search approach in which every page is downloaded from the web. The second method is derived from a web caching solution.

The third component looks at the effectiveness of threads in a real-time engine. It would not be unexpected that the multithreaded design will provide better results over a single thread, so the research will focus on the difference in performance using various numbers of threads.

Overall, the mission of the research focuses on what factors in these three implementation areas will have a greater impact on the performance of a real-time search engine. The remainder of the thesis is organized as follows. Chapter 2 discusses some of today's current uses of search engine technology. Chapter 3 describes the concepts and the processes behind a real-time search engine. Chapter 4 highlights the design of the proposed application. Chapter 5 delves into the methodologies of traversing web content. Chapter 6 focuses on caching web content and determining what is deemed as a query hit. Chapter 7 explains how threads play an important role in real-time performance. Chapter 8 describes the processes involved in executing a real-time search engine. The research wraps up with an analysis of the results in Chapter 9 and concluding research overview in Chapter 10.

Chapter 2

Contemporary Search Engines

"People think searching is a monolithic activity," said Werbach. "But it isn't. The reality is that sometimes you will want to use different techniques for different reasons" [Greenberg99, page 11]. His quote sums up the essence of this chapter. On the outside, search engines appear to perform in the same manner. However the implementation and research that goes into developing different contemporary search engines tells a different story.

In an article titled, "Searching the World Wide Web," Michael L. Mauldin explained how he developed Lycos into the search engine wonder that it is today [Knoblock97]. He briefly touched upon the operation of a web engine by pointing out its major functions. A forging program would first gather hyperlinked pages across the web into an index. Next, the pages are stored in a database repository. Finally, a retrieval system returns hyperlinks to web pages based on keywords, phrases, and ideas matched against the database of indexed pages [Knoblock97]. On a fairly simplistic level, Mauldin was describing the process of web query retrieval. It can be quite a daunting task, but companies like Lycos, AltaVista, and WebCrawler have enhanced the online experience by making web pages easier to find.

Web search and retrieval architectures can be categorized loosely as traditional search engines, metasearch engines, and distributed architecture search engines [Pokorny04]. All three categorizations make up the general-purpose search engines that most people are familiar with on the web. Essentially, a user enters a topic into a graphical web interface in the form of a word, phrase, or groups of phrases. The engines return a set of ranked web pages that pertain to the user's general topic. Ranking web pages is a useful tool for search engine users as it recommends the best possible page that pertains to the search topic.

2.1 Traditional Search Engines

Traditional search engines, also known as centralized search engines, consist of the following components: crawlers, page repositories, indexers, query engines, and ranking algorithms [Pokorny04]. Crawlers, known as spiders or bots, perform the basic task of scouring the web for web content. Their sole task is to download content from the pages they traverse in the web via hyperlinks. The interconnecting web of hyperlinks creates a web map which the crawler's control module uses to determine where the crawler has been and will search. A visual representation can be seen in Figure 2-1.

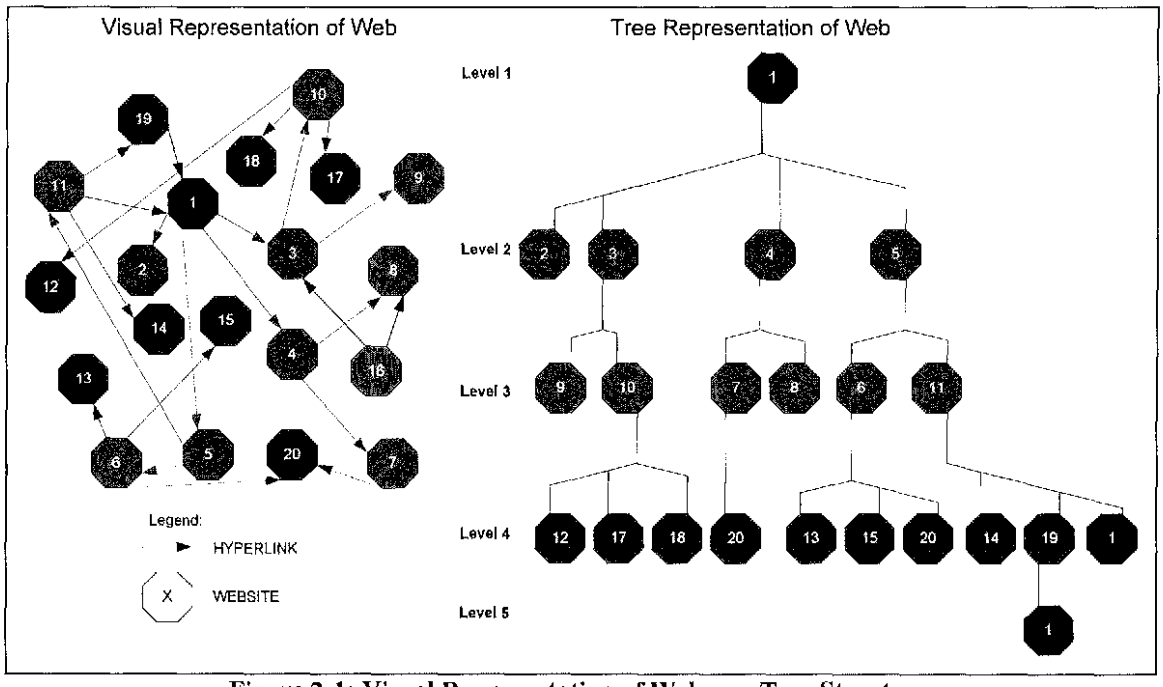


Figure 2-1: Visual Representation of Web as a Tree Structure

The content retrieved from the crawler is stored in page repositories for further processing by the traditional search engine. Indexers are processes that parse the downloaded page content within page repositories. The indexers are responsible for determining how the page should be categorized within the search engine.

The last major part of the traditional search engine is the query engine. The query engine's responsibility is to locate which pages in the page repository match the user's query. The query engine uses the indexed content in a database system to discover the pages and displays them according to a ranking scheme. The ranking algorithm concludes the relevance of a page to a user's particular search topic. The page's ranking can be based on query-independent criteria, such as lists of related pages or similarity indexes, or it can be based on query-dependent criteria, like the page's length and vocabulary, the publication date, or the number of pages connecting to a particular page.

Of course, a mixture of query dependent and independent methodologies can be used to contribute to a more refined ranking.

2.2 Meta Search Engines

Another facet of the web is a concept known as the hidden web. This term describes the vast amount of web pages that are not reachable by traditional search engines [Pokorny04]. Such information typically resides in text databases. In some cases, the data is located in the "deep web" where it is only reachable through web form interfaces or user authentication [Hafri04]. The metasearch architecture is a searching methodology that utilizes the strengths of text-based and web-based search engines to discover pages that satisfy a query. It achieves this by searching multiple databases simultaneously. Examples of such an architecture would be Metacrawler and Dogpile. One disadvantage of this architecture is that the referenced databases must have a descriptive tag or summary about its contents. Many databases do not publicly announce this information, so the number of resources for a metasearch architecture are limited. Research has been in development to automate the extraction of content from these databases to form a database summary.

A metasearch engine has three major components: a database selection process, a query translation process, and a result compilation process. The database selection process is critical to the entire application, as it is responsible for determining which third party databases would yield the best results. This is accomplished using the summaries

provided by each search database. The query translation process must modify the original metasearch query into a query format that is compatible with the third party database. Lastly, the metasearch has to merge the results of the responding databases and rank them accordingly for the user.

2.3 Distributed Search Architecture

The third search engine architecture is described as a distributed search architecture. The web in general is a large peer-to-peer network that contains distributed information and processes. With traditional and metasearch engines, the crawling and querying processing takes place in a centralized area or application. Pokorny highlighted two potential issues with centralized search engines: high computational costs and potentially poor rankings. The distributed search architecture aims to divide the tasks of searching and categorizing the web among several search engine systems, thus relieving the burden and increasing the effectiveness of the search engine process on centralized systems or applications [Pokorny04].

Pokorny states that this architecture is not a new one, but the research in recent years has started to develop some very interesting techniques. A system known as Apoidea manages the load in a decentralized system by distributing the crawling process based on the geographical proximity to web resources [Singh03]. Another technique uses decentralized crawlers to find web content based on the document rankings of pages

[Aberer03]. A more common method of distributed searching utilizes a decentralized group of metasearchers to query and obtain results from specialized search engines.

2.4 Content Ranking

Searching, retrieving, and matching web pages are not the only functions that are performed by search engines. Search engines provide the extra service of recommending which pages will most likely satisfy the search topic. This process is known as ranking. Ranking is the search engine's attempt at measuring the importance of a page in regards to the search topic. The ranking is typically rated on the following three approaches: linking, content, and anchors [Pokorny04].

2.5 Directories and Crawlers

Directories are maintained by people categorizing web pages into different subject areas based upon their content. This assures the user that what they locate in a category only pertains to the search topic. Most pages that are classified by directories are submitted from outside sources. The disadvantage is the processing of pages can be "labor-intensive" [Filman98]. If a page changes content, it will take time before a classifier can review the page so it can be moved to another category. Filman and Pant also mention that if a page is not found within a catalog, there is a good chance that it will not be in the directory at all [Filman98].

An alternative to directories are bots or spiders. Lycos[1] creator Michael Maudlin says that most spiders that scour the web searching for information function in different ways, though they accomplish the same task [Knoblock97]. The idea behind the spider's functionality came from observing how the typical web surfer visits web pages by jumping from one page to another via hyperlinks. Essentially, spiders scour the web by traversing through the same hyperlinks [Szymanski01].

Mauldin modified John Leavitt's LongLegs search engine [December94] and added the Pursuit retrieval engine [Knoblock97] to create the Lycos search engine. The basic goal of the spider was to catalog the information on the web by reviewing a web page, extracting its hyperlinks, cataloging the page contents, and then moving on to process the page's hyperlinks. Maudlin mentions that Lycos' early spiders were written in Perl. This allowed the storage queues, which the spider populated, to be stored in a Database Management System (DBMS) for effective access [Knoblock97]. Presently, Lycos is a proprietary spider written in C for quicker processing.

2.6 The Methods of Traversing the Web

The next issue is how to explore web sites. The depth-first search is a good traversal methodology, but this can cause a tremendous load on some servers. The breadth-first search is an alternative to reduce the load, but Mauldin says that the algorithm favors "smaller web servers over large mega sites such as universities and Internet service

[1] Lycos was named for the wolf spider, Lycosidae lycosa, which catches its prey by pursuit, rather than in a web.

providers" [Knoblock97, page 9]. Lycos settled on a popularity heuristic approach. This algorithm basically notes the number of external servers linking to one web page. The more links found would determine popularity of the page and the direction of the search.

2.7 Web Page Content Extraction

Most web documents are written in a language known as Hypertext Markup Language (HTML). Although other file formats have become popular over the years, HTML remains the foundation for formatting web documents. HTML's purpose is to notify a web client about how a document should be displayed in a web browser. The HTML code is made up of a series of tags, end tags, and data content. Some of the functions performed by HTML can dictate the title of the page, give a summary of the page's content, and inform how the content should be laid out in the page. This is achieved by placing the data content between a HTML tag and HTML end tag, as displayed as below:

```
<title> This is an example </title>
```

Because HTML tags are used for formatting, it is difficult to use HTML to describe the data content of a web page. As a result, Extensible Mark Language (XML) was developed for the purpose of describing a document's content. Fortunately, this does not mean that content cannot be retrieved from an HTML document.

When a page is retrieved from the web, the informational content needs to be processed from the HTML tags. One technique is to copy the entire content into a database. Bun

and Ishizuka's Emerging Topic Tracking System (ETTS) provides a good example [Bun01]. By using HTML tags as a reference to where data can be located in a document, ETTS uses a parser to remove and replace HTML tags with sentence delimiters. After the HTML tags are removed, the result is raw text in a semi-sentence structured format. ETTS compares this text to a previously recorded text for changes and updates in an index database system with those modifications.

Lycos' Maudlin points out there could be a copyright violation with such techniques, so Maudlin chose to parse the HTML pages based on "weighty" terms [Knoblock97]. Through statistical sampling, the spider is able to determine how often certain words are used in a page. The top one quarter of keywords with high statistical scores are chosen as "representatives" of the page and written to the database repository. Eventually, these keywords and scores are used to approximate how useful a page might be to a Lycos user.

Unlike Bun and Ishizuka's ETTS, Lakshumanan, Sadri, and Subramanian developed WebLog, a query language which categorized web pages based on a keyword's relationship to the page's HTML code [Lakshmanan96]. In essence, important keywords are obtained by their association with HTML tags. This relationship, known as rel-infon pairs, is used during processing to match a query to a web page. One example of WebLog's syntax would be

```
<url> [<rid> : <attr> → <val>]
```

where <url> is the URL of the page, <rid> is the name of the rel-infon, <attr> is the HTML tag, and <val> is the value being searched. One disadvantage of the query language is the cryptic syntactical structure for a simple query.

When a subset of XML-QL, called XQL, was prototyped by Ishikawa et al. of Fujitsu Laboratories, a simple query language design was created to search XML documents [Ishikawa99]. Although XML is not intended to be used as a document structure language like HTML, the principals behind designing a query language were similar. The creators noticed that a relational database's capacity to relate categorize data was very similar to XML's ability to classify data.

The popularity of XML stems from its ability to allow designers to create a document that can self-describe the content it contains. XML allows for the development of personalized tags to define, transmit, validate, and interpret data between different sources [Webopedia]. XML is assisted by a document known as the document type definition (DTD) that describes the customized tags and format of an XML document. XML has gained recognition with system designers by allowing different systems to communicate via a common data format.

Developing a language to query XML documents based on relational database's Structured Query Language (SQL) was not a difficult step to make. Ishikawa and company proposed using an SQL-like language to query XML documents. The format appeared as:

```
SELECT $book.author
FROM bib URI "www.a.b.c/bib.xml", book $bib.book
WHERE $book.publisher.name ="Addison-Wesley"
```

The DTD for this statement appears in the Appendix A. In this statement, the query statement is satisfied if the publisher's name is "Addison-Wesley" from the Universal Resource Identifier (URI) "www.a.b.c/bib.xml. If a match is found, then the book's author would be displayed. Modeling the XQL language after SQL has some real advantages. The structured of the XQL language is similar to SQL's SELECT statement. This provides an easy transition for users who are familiar with standard SQL to create XQL queries. Another advantage is derived from the similarities on how data is retrieved. Through the convenience of a single select statement, the user can specify the data needed, where to find the data, and what criteria the data needs to satisfy in order to be retrieved.

2.8 Providing a Direction for Web Traversing

No matter which spider/crawler/bot implementation is used, there are still a couple of issues that need to be addressed when searching the web. First and foremost, what determines which page should be downloaded? How does the crawler know which hyperlink in the page will provide the most useful and comprehensive results? Pokorny states that certain measures can be used, such as interested-driven metrics, popularity-driven metrics, and location-driven metrics, which can assist a web searching system with finding meaningful pages [Pokorny04].

The second issue is how often searched pages should be refreshed. Most search engines rely on database management systems to house page content downloaded from the crawlers. Twenty-three percent of web pages change on a daily basis [Pokorny04]. Most updates occur on a monthly schedule. Such a time frame can cause many page listings in the search database to become obsolete and stale within a relatively short time frame. One suggestion is known as a proportional refresh, which tracks the update frequency of a particular web page and schedules the crawler to coincide with the frequency of the update.

Pokorny mentions the final issue as developing a parallelized crawling process without duplicating efforts [Pokorny04]. If a set of crawlers start categorizing the web, paying no attention to the work efforts of its peers, there is little doubt that several crawlers could download the same page causing duplication of work. The crawler would have to be aware of what pages have been parsed by its siblings in order to ensure time and resources are not wasted on duplicated work.

Greenberg addresses some new approaches to searching and ranking that could be integrated into new projects [Greenberg99]. Greenberg mentions the concept of human annotation, which bases search results off of the behavior of previous user web searches. In essence, as the search engine is used, it tracks which pages are popular for a particular topic. Those pages are related to the topic or keywords and used the next time another user searches on the same topic. The downside, according to critics, is that the more

popular pages will overcome less popular ones and diminish the usefulness of searching for content on rarely visited pages.

Direct Hit is another approach known as a popularity engine [Greenberg99]. Cofounder Gary Culliss cites that his application anonymously tracks a user's surfing habits to determine which content is of interest to the user. The tracking habits are calculated through a proprietary algorithm to provide for ranking the relevance of web pages and sites.

Another methodology is the Googlebot used by Google [Greenberg99]. The creators, Sergey Brin and Larry Page, describe the process as a mixture of keyword and human-annotation based approaches. Googlebot uses the topological structure of the web to determine the importance of a page within a web of hyperlinked content. During keyword searches in Google, text-matching operations, as well as some other algorithms, are performed on neighboring pages via hyperlinks. The search returns the web location addresses with similar content within a web of interconnected sites and pages.

Clever [Charkabrti99] was developed in 1996 as a means of determining the validity of a page as an authority of its content. Clever provides the method of determining "a topic's most definitive and authoritative web pages" [Charkabrti99, page 9]. In essence, these pages become the most relevant and qualified resource, or authority, that a user can find on a particular topic. Once a page is determined to be authority, its ranking becomes much more important based on its quality and relevance to the topic. The second area of

focus in the Clever project is reviewing the hyperlinks that connect the vast amount of web pages together. The creators of Clever contend that the hyperlinks from one page act as an endorsement to another page. Thus, mining and processing the web for a collection of these endorsements can lead to the discovery of authorities. The collections of hyperlinks are referred to as hubs.

Granted, on several web pages, hyperlinks can refer to content that is considered invalid or useless. The idea is that through processing link structure between interconnecting pages, Clever can create an aggregated list of links (i.e. a hub) that point to one page as the authority of all pages within that topical area. By analyzing the density, direction, and clustering of links, the Clever application can determine which pages will provide the most beneficial content based on a user's search needs [Szymanski01].

In summary, the basic search engine needs to crawl through the web and retrieve data content that it believes is relevant to the topic of the page. That data must be categorized and weighted in an index to determine the quality and relevance of the topic. A query language is used to query the index and retrieve URL addresses that pertain to the user's query. To create a real-time search engine, the design will have to utilize features of contemporary search engines, such as to provide a simplified query language, categorize semi-structured data into meaningful relationships, and traverse the web in a manner that will not exceed the user's patience.

Chapter 3

Real-Time Search Engines

In the world of the web, the content that embodies it is constantly changing. Jarolav Pokorny states there are two types of changes on the web: persistence of web pages and page content modification [Pokorny04]. Persistence of web pages refers to the lifespan of the page, or in other words, how long does a web page stay on the web. One study would suggest that the web page lifespan seems to be a little under two years [Koehler99]. The site that houses a page seems to average a little over two years. In a web of about 3 billion web pages by today's estimate [Hafri04], having a page only exist for two years creates a lot of dead links and content turn over. In addition to a page's existence on the web, the page also undergoes content changes. The page and its web address still exist, but the content in it changes. Twenty-three percent of all web pages change every day [Pokorny04]. Constant change can render completely new content within just a couple of days. Web addresses gathered by spiders and search bots might no longer reference the same content that might have been previously present.

The challenge is to provide fast and efficient methods for searching the web. Real time search engines provide one type of solution to the dynamic nature of the web. By searching pages in real time in a localized area of the web, information that is retrieved is guaranteed to be available for the user at the time of the search. With traditional catalog-

based search engines, the data can age, becoming outdated and irrelevant to the user's needs. One symptom of searching an index of cataloged content is the presence of dead hyperlink addresses in the result set. For example, as previously mentioned, nearly a quarter of the web pages change daily. This means that the page must be constantly monitored and the catalog updated to maintain continuity with the current web page.

A spider or bot would have to refresh its catalog content constantly just to keep the addresses up-to-date. Constantly monitoring a change in a web page can be time consuming for a web crawling system and place undue stress on a web server. As a result, a spider may only visit a page a couple times per month. Visits may occur more if the page is highly visited by web traffic. With infrequent updates from web spiders, the catalog is bound to contain web addresses to content that may not exist.

The reliance on indexes can also give a different set of results based on the traditional search engine that is used. As these engines search the web independently of each other, the engines' spiders could catalog data using different processing algorithms or crawl to pages that competitor spiders never encountered. Unlike metasearch engines, a traditional search engine cannot utilize a competitor's index to process a query. As a result, different traditional search engines will return different result sets.

In another scenario, the server hosting the page may be experiencing technical problems which keep it from serving web pages. Such a case would prevent the spider from visiting the page, thus allowing the indexed information for the page to further age.

Since a search is performed in real time, the result set contains live and up-to-date content that satisfies the user's needs. Real-time search engines can eliminate inactive webpages by dropping those web addresses that it cannot reach.

One of the technicalities of a real-time search engine can also allow it to become one of its greatest weaknesses. Since pages are searched in real time, it is possible that the search could take several seconds to complete. Essentially, the real-time search engine has its crawler download and parse web pages as the user waits. Since traditional web crawlers can take several hours to find, download, and categorize the pages, a reasonable assumption is that the real-time search engine crawler would have the same type of performance. In order to be effective, the real-time search engine must search an area of the web that is much smaller than the traditional search engines.

Some research has been done in the area of real-time search engines. In one paper, the authors A. Iekji and F. Fotouth developed an adaptive real-time search engine to address the issues with traditional search engines [Ikeji99]. The Ikeji's real-time engine allows the user to specify where the searching should begin by giving an URL address. The engine will review the contents of the address and determine if it satisfies the query given by the user. Anchor tags or hyperlinks that are found in the page's HTML are stored so they are searched next.

Due to the time and space constraints, the process cannot continue to search the web indefinitely. Only a partial sample of the web can be searched. The process will

continue until the search time expires or the engine has traversed a certain amount of links from the original page. The user will have specified both the time and number of hyperlinks in a statement that is very similar to relational database's SQL and to WebSQL [Mendelzon96]. In addition to searching content, the engine also searches for web page attributes, such as the last time the page was modified, the title, the size of the document, and images contained within the HTML document.

Traditional search engines utilize an index for categorizing page content. With Ikeji's real-time engine, no index is needed. The information is retrieved directly from the web page and processed, allowing the results to be current and accurate. Because time is limited in real-time searches, the engine must decide the appropriate path to take in a real-time search. This algorithm can vary among search engines, and may remain confidential in the case of commercialized engines. Ikeji's engine utilizes a scoring system to determine which traversal path might present the best results. An alternative method would be the brute-force approach, also known as a linear search. The web traversing alternative would basically download and process every page that is linked from the original page. Brute-force technique lacks the finesse of estimating the pockets of web pages that would likely satisfy the query. Consequently, the brute-force search would expend more system resources and time to execute.

Ikeji's methodology is encouraging, as it closely resembles the direction of this body of research. The advantage to Ikeji's real-time engine is that by focusing search efforts on an area of the web that is more likely to return results, there is a favorable outcome that

the result set will contain information that is valid and valuable to the user [Ikeji99]. The results of the engine were mentioned in terms of how many pages satisfied the query during the authors' testing. There were no indications of processing time or comparisons against other engines.

Another real-time search engine is called Orase. Although very little research could be found on this engine, a brief description of the engine's abilities can be found at its website[2] [Franz04]. The application was written by Markus Franz. The engine "guesses" different URLs based on the query entered into the system. Those URLs are then searched for content.

Other research into real-time search engines focused on dividing the massive search content onto a distributed architecture [Hafri04] and determining how to categorize real-time content for presentation to the user [Chau01]. Although the implementations were not discussed, the distributed architecture and presentation research is beyond the scope of this paper.

The real-time engine is not meant to replace traditional search engines, but to enhance the user's ability to find additional information in a quicker amount of time. This would be analogous to the "FIND" function in a word processing application. One web page is used as a primer address for the search engine. The address should be selected by the user and should have some relevance to the search criteria. The real-time search engine will search through a web of connected hyperlinks starting at the primer address. This

[2] Orase can be found at http://www.orase.com.

methodology provides two advantages: the result set content from the search should be relevant to the subject of the primer page and the amount of time needed to execute the search should be minimal as it only searches the immediate web pages of the primer page.

Chapter 4

Proposed Application

The concept of a real-time search engine stemmed from the development of a web-based query language graduate project. Varieties of web query languages have been, and still are being, developed to retrieve relevant results and/or provide an easier language for web querying. The purpose of the research is to review the performance of a real-time search engine in three different component implementations. The web query language models the semantics and syntax structure of the web. The application is designed to query web pages and retrieve the results of the query, but do so in a real-time execution of the query language.

4.1 Searching Without a Catalog Index

Current web search research relies on database management systems as the source of the web page content. A database schema, modeling the structure of web pages, would create a more accurate picture of how data is represented on the Internet. With a schema in place, a more descriptive query language could be used to retrieve information from that database which houses details on web pages and their informational content. As described in Chapter 2, the structure of the web content makes it very difficult to find

some way of relating data to its location on the Internet. This includes multimedia content, dynamic content, content hidden behind password secured sites, and static HTML-based content.

In the development of the language and its engine, the thought of retrieving the data was of great concern. One logical course of action was to search the web itself instead of peering into a database of semi-related, and possibly outdated, web content. Thus, the implementation of a real-time search engine was built as a method of finding content that is as current as possible. The prototype function according to its design, and the query language provided the needed semantics of retrieving HTML-based web content. The issue at hand was the performance of the crawler and data content retrieval system.

4.2 The Approach for Improving Performance

In this research, three approaches are used to enhance the researcher's previously developed query language engine prototype. The application dynamically searches web pages and their adjacent pages for a result set that satisfies the query. A series of feature enhancements are being implemented to increase the performance of the search engine. The prototype has been in development for two years to test the theory behind a real-time web query language for searching a dynamic web environment. The prototype had encouraging results, but suffered in performance when the number of pages that needed to be searched grew at an exponential rate.

The basic functionality behind the real-time search engine is no different from traditional search engines with respects to the crawler and the query processor. The distinction is seen in the implementation and integration of these two processes into a real-time environment. Basically, users submit their search topic in the form of a query to the real-time search engine. The user is defined as the person who gives queries to the system for the purpose of searching for web pages that suits a topic of interest. The query is the statement that a user gives to the system to state the intention of the search. The query contains a web address, specified by the user, which acts as the first page to be downloaded, parsed, and processed against the query. Any hyperlink addresses found in the page are stored in memory. The content of the page is parsed and matched against the conditions stated in the user's query. If the page contains keywords that match the condition, the web address is placed in the result set. If not, the web address is dropped. The result set is described as the set of web addresses that are returned to the user after the search has been completed.

The second part of this process is the repetitive search of pages in real time. Once a page has been processed, the spider downloads the next page from the list of hyperlink addresses that were stored earlier. The page is parsed and processed. The entire repetitive process ends when the spider has reached its search domain boundary specified by the user. The search domain is the boundary which keeps the engine from endlessly searching the web. The boundary is set by the user in the form of a depth-level number, which represents a current web page's position within a search domain.

The query processing is complete when there are no more pages to be matched. Lastly, the result set, which is a list all URLs that satisfy the user's query, is displayed to the user. A query is satisfied when a webpage matches the user's query statement.

This describes the basic process of a search engine. Traditional search engines and the proposed real-time search engine differ in the execution timing between the crawler and the query processor. Traditional engines download and parse web content in advance of a user's need to search the web. The query processor searches the database of indexed content and retrieves those pages that might be of interest. The wait time between the start of query processing and receiving a list of URL addresses can take as little as a tenth of a second, depending on the search engine[3]. With real-time search engines, the crawlers traverse the web immediately after the user submits the query and the query processor starts matching pages after the first page has been downloaded by the crawler. Due to the fact that the process downloads and search pages in real time, the execution takes longer than traditional search engines. It is important to note that the enhancements mentioned in this research are being applied to improve the real-time performance.

The development of a query language was the goal of the prototype. As this research is based on developing a better performing real-time search engine, the query language structure does not have an actual impact on performance. It allows the user to specify under what conditions the search engine will operate. Performance results of the real-time search engine will be compared to the various operations the user can set in the query statement. It is important to highlight that some changes in the language structure

[3] Most search engines will return how long it takes to execute the query.

are needed to allow the user to make performance-based enhancements to the search engine.

4.2.1 Approach 1: Traversing the Web

First and foremost, the performance of a web crawler will depend on the method of web traversal and the amount of pages that will be visited. The more time spent traversing the web, the longer the user will have to wait for some sort of search results. Essentially the web is a massive interconnected hive of web pages. By traversing from one page to another page, a traversal path is created. In some instances, a traversal path could revisit pages that have been visited. Such an event would create an infinite traversal loop. In a real-time search engine, this can have disastrous effects on performance. Thus, the engine must track and avoid pages already visited.

The second issue involving performance with real-time search engines is coping with the great amount of pages on the web. Commercial crawlers could take days to traverse just a section of the web. With a real-time search engine, this time has to be drastically reduced to be of any benefit to the users. Hence, the number of pages a real-time search engine processes will be much more limited than its traditional search engine counterparts. As a result, the real-time engine would be more suited to searching within a domain name or a small group of related web pages. The simplest analogue is to compare the real-time search engine to the "FIND" feature for a large word processing application. The content is limited, but the size of the document is too great to review by

the human eye. Thus, a boundary must be in place to prevent the engine from traversing too far and executing too long. The boundary can be specified in the query statement as a maximum search depth. By using a specified depth, the user can control the amount of pages visited and how deep into the web the engine will search.

With a search boundary specified, the application must have a traversal method. Web traversal may be accomplished basically via three approaches: a breadth-first approach, a depth-first approach, and a selective path variation of both the breath-first and depth-first search. The goal for implementing these traversal methods is determining which method provides the largest result set in the shortest amount of time. The depth-first and breath-first searches are referred to as global searches, which means they will search every page within the search domain. The selective path variation of these two traversal approaches, developed for this research, is known as pruned branch traversal, which selectively chooses one traversal path over another based on the content of the pages.

The depth-first search drills down to the bottom of the search domain tree by searching a parent page and moving to the first available child page. Essentially this creates a vertical search pattern in the tree. One downside is that the depth-first search could cause a tremendous load on some web servers.

The second traversal method is the breadth-first search. This method traverses through the search domain tree by searching one depth level at a time. This would be similar to a horizontal search pattern in the multi-branch tree. The breadth-first search can be used to

reduce the load on servers, but the algorithm could favor "smaller web servers over large mega sites, such as universities and Internet service providers" [Filman98].

The last method utilizes a selective path approach of the depth-first and breadth-first methods. With the previous traversal methods, all pages within the traversal domain are searched. It is reasonable that some pages will link to child pages that have no commonality in topic or content to the user's query. Such pages can be "pruned" from the traversal domain if there is no relevance in topic or content. Pruning refers to the intentional removing of a section of the search domain that has no relevance to the search query. This has the effect of reducing the number of child pages that must be processed, thus reducing execution time of the real-time search engine.

4.2.2 Approach 2: Real-Time Caching

The current traditional search engines do not search the WWW in real time. These search engines scour the web, placing their findings in large database repositories for later retrieval by users via a web graphical user interface. In order to retain the real-time nature of the application, the web search has to be executed when the user wants to perform a web search. Another method used to increase performance among many computing functions is caching. Essentially, the traditional search engine is a large repository of cached web content that is searched during a query. As highlighted earlier, the information can become stale and old after a short period of time. But if constantly

updated, cache can reduce the time of retrieving data from a cached page versus downloading the content over the web.

By consulting when the last time a page was updated during traversal, a crawler can determine if a newly modified page should be downloaded, parsed, and cached or if a cached version of the page will do. Creating a hybrid matching solution utilizing the database as a cache repository and using the memory-based search tree as the matching engine, the goal is to determine if a real-time caching method performs better for the search engine. Using a hybrid of a real-time downloading and caching, the search engine would avoid any network/internet lag that might be encountered while downloading pages.

4.2.3 Approach 3: Concurrency

The final performance enhancement will be developing the search engine into a parallelized threading architecture. Since the goal of the application is to parse pages in real-time, parallelizing the execution through the use of threads will have an influence on performance. In a multi-threaded environment, threads can be employed to execute tasks that are functionally independent of each other. By creating several threads to work together to crawl and process HTML pages, the information will process much faster. The real-time search engine prototype is executed in a single threaded instance. This means that the engine downloaded, parsed, and matched one page at a time to accomplish its task. In a modification of the prototype, the spider and query processor will be

separated into threaded objects, allowing the creation of multiple crawlers to run concurrently.

It is not expected that the real-time engine will be competitive when compared to today's current search engines. The goal is to provide another tool to supplement searching on a smaller section of the web. In doing so, the real-time search engine will provide results of pages that are current and more relevant to the user's search needs.

Chapter 5

Traversing Through the Web

The user gives a query statement that specifies the search conditions. The query supplies a primer URL address, the search conditions, and how deep into the web the program should search. A primer URL address is used as the initial page for starting the search process. In order for the real-time search engine to be effective, the search must begin in an area of the web that is applicable to the topic. For example, if a user wanted to search for the diet habits of the bald eagle, it would be reasonable to physically start a search at a site pertaining to eagles or a site about animal diets or a even a site of a zoo that exhibits eagles. Such information would not be found at sites that do not deal with the subject of eagles or animal diets. Thus, a real-time search engine needs a primer page, or URL address, of a site that relates to the topic of interest to establish the searching process. The value of the real-time search engine comes into play when it returns page results that are linked from the primer page.

5.1 Defining a Search Domain

The Internet is a web of interconnecting web pages. Web pages that are closely associated with each other, through direct hyperlinks, form a community of sites that

typically relate to a specific subject matter. In terms of web searching, a search domain describes those pages that the real-time search engine discovers and processes. The search domain may consist of pages outside of a topic area, or it may simply link to other pages that have not been encountered. The idea is that the group of interconnected web pages creates a searching area for the real-time engine to search simply by being connected to each other.

Starting with the primer URL address, which forms the "root" of the search domain, the search domain can be pictured as the common data structure known as a "tree." The diagram of the search domain tree can be found in Figure 5-1. In the tree structure, the top page (i.e. the page from which the search begins) is known as the root of the tree. The hyperlinks of the root page are known as child pages. In the example in Figure 5-1, the root's child pages would be categorized on level two of the search domain tree. Those level two pages would have hyperlinks to child pages which would be located on level three of the search domain. The remaining level numbers are given accordingly.

In Figure 5-1, it can be seen that the interlinking pages form a tree that has a level assigned to it by the traversal algorithm. This allows the engine to determine how far it has searched into the web. In this diagram, a depth level of 3 was specified by the user in the query statement. When the engine encounters child nodes with a level number greater than 3, the spider discontinues adding links to the hyperlink queue. At the time the hyperlink queue is exhausted, the search engine stops executing.

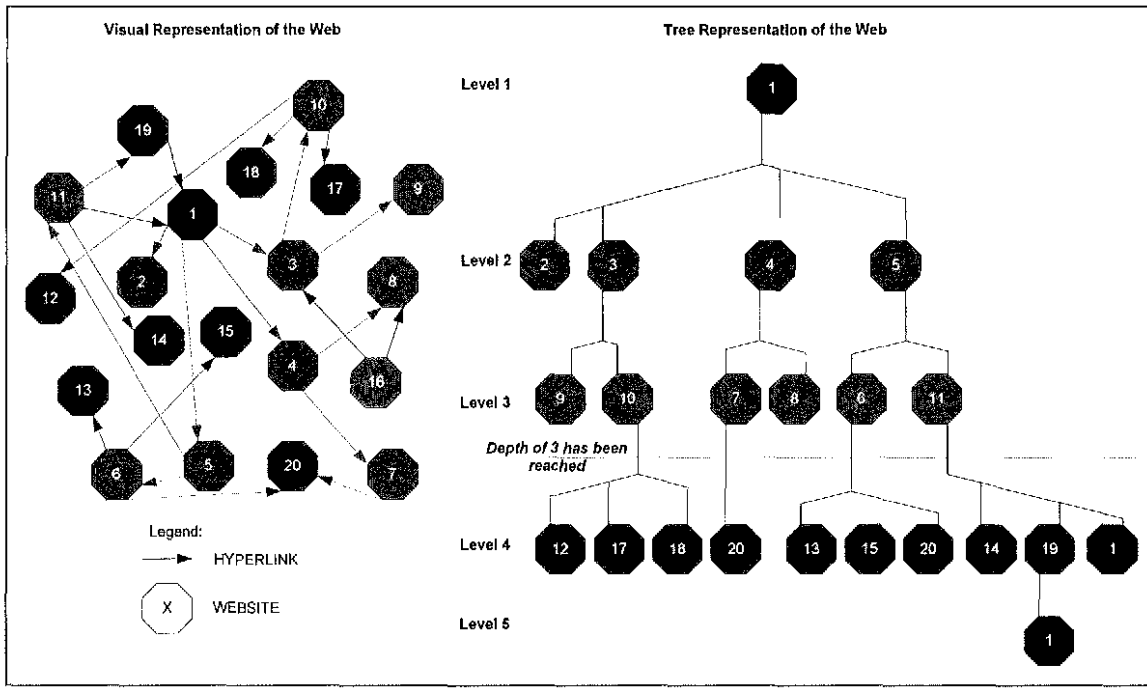


Figure 5-1: Depth Level in a Tree Representation of the Web

As the spider traverses from node to node in the search domain, the content is downloaded and processed by the engine to determine if the page satisfies the user's query. At this point, the spider thread retrieves another URL from a hyperlink queue and repeats until the appropriate depth level is reached.

5.2 Removing Cyclic Loops from Traversal Paths

Pages can refer back to themselves, creating a cyclic loop in the tree that is highly undesirable. In such circumstances, a search engine could find itself looping through multiple pages that it has already visited, thus having disastrous effects on performance. In order to prevent this repetitive process from occurring, a web crawler must retain the knowledge of the pages it has visited. This information is kept in an easily searchable

data structure known as an AA Tree. When a hyperlink in a page is encountered, the link's address is listed in the tree, if not already present. The real-time spider consults the tree to see if the URL address has been visited. If it has not been visited, the address is queued for downloading and processing. If it has been visited, the address is simply discarded.

5.3 Depth-First and Breadth-First Traversal in Search Domain

There are two types of traversal methods mentioned by Mauldin [Filman98] that are implemented in this application: the depth-first search and the breadth-first search. The first type of traversal implemented in the research is the depth-first traversal. The depth-first search method describes the process of traversing down to the leaf node of a branch of a tree. The process is detailed in Figure 5-2. In the figure, the tree has a depth of 3, but on the web, the depth could be limitless. As a result, the search would be unending, which is clearly not an option for real-time searches.

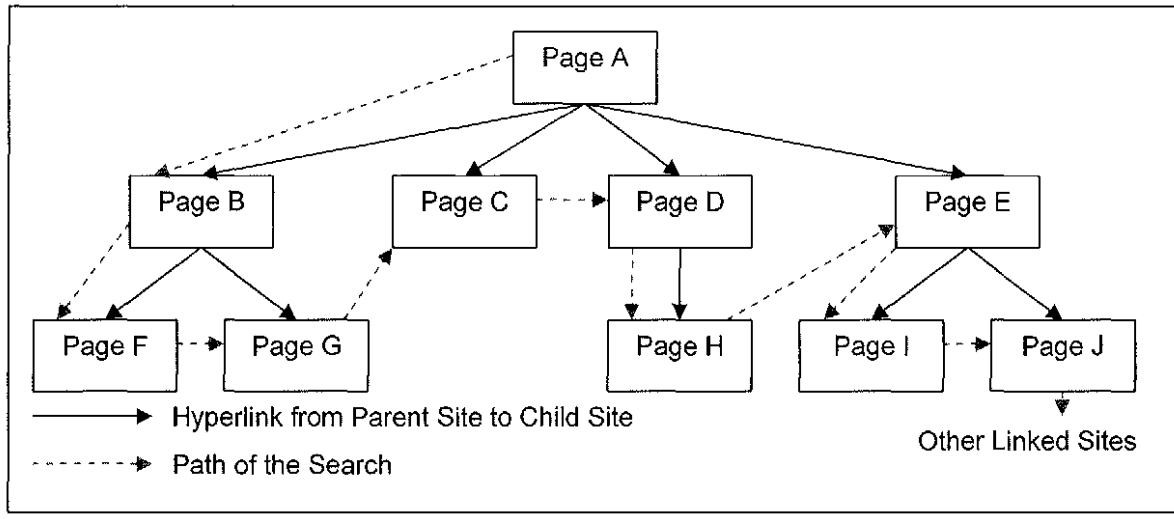


Figure 5-2: Depth-First Traversal Search

With a depth of 3 specified in the query statement and a URL starting at Page A, the depth-first search would take the path of A, B, F. When F is encountered as the depth limit, the program would search for additional children at Page B. In this case, the page would be Page G. The search would continue as G, C, D, H, E, I, and J. This methodology performs well, but there is a disadvantage stated by Mauldin. With a depth-first search, the Hypertext Transfer Protocol (HTTP) server could become overwhelmed with hits from the search engine if the links consistently referred to the same web server.

The second implementation, and an alternative to depth-first searching, is the breadth-first traversal. This search process visits all nodes on the same level before searching any of these nodes' children at the next level. This process is demonstrated in Figure 5-3. When the search begins at Page A with a depth level of 3, the path the algorithm takes is A, B, C, D, and E. These pages are related to Page A by being a direct child of A.

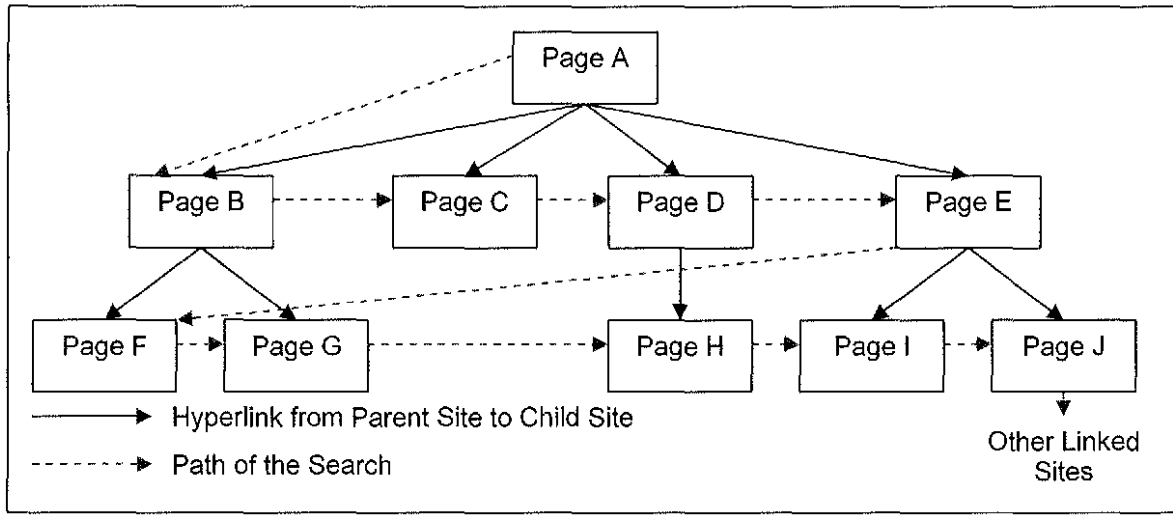


Figure 5-3: Breadth-First Traversal Search

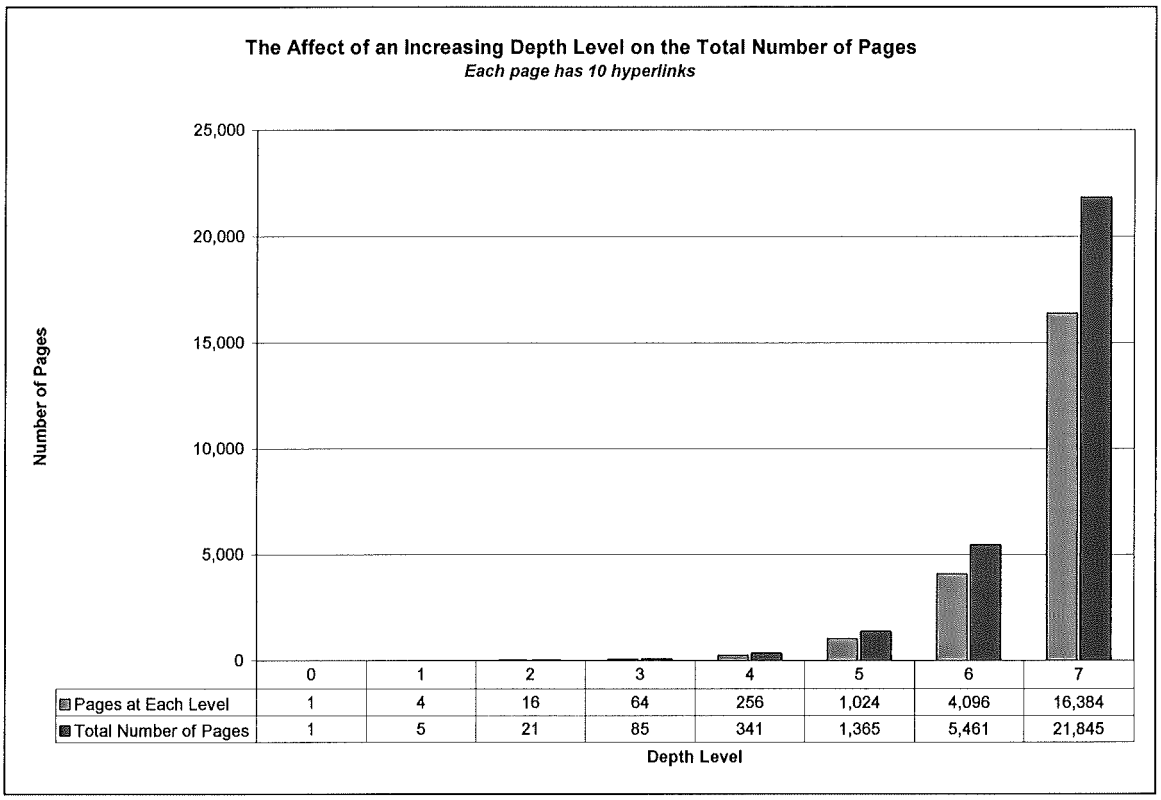
When B, C, D, and E are searched, the algorithm records their children in a queue. The process pulls the front node from the queue, parses it, and inserts any hyperlink URL addresses on that page at the rear of the queue. This gives a level by level traversal during the dynamic search.

The HTTP server could still become burdened by requests if a page only makes link references to pages on that HTTP server. Chances are the page has links to another HTTP server, thus the search engine will not burden one single HTTP server for a great deal of time. With a greater depth level, there is better chance that child pages will refer to links outside the initial root site. In these cases, the request load that the real-time engine makes of the HTTP server should only come in waves, and not all at once. In essence, the breath-first traversal improves the response performance by allowing the server to respond with current requests before burdening it with more hits.

5.4 Pruning the Search Domain

The third traversal technique is an algorithm designed to reduce the amount of pages that are searched by the real-time search engine. The method, developed from this research, is known as the pruned branch traversal. As described earlier, the pages within a search domain can be laid out in a traversal path that is similar to a multi-node tree data structure. At the root level, there is the root page that is the head of the tree.

The root page could have no hyperlinks or several hyperlinks that leads to other pages contained within it. Those pages from the root page could also have no hyperlinks to several hyperlinks embedded in its pages. In a theoretical example, if every page in the search domain had four hyperlinks, at each depth level the number of pages that would have to be searched would quadruple. In Figure 5-4 at a depth level seven, 21,845 pages would have to be search. In reality, the total number of pages to be searched would vary based on the number of hyperlinks on each paged search.



	0	1	2	3	4	5	6	7
Pages at Each Level	1	4	16	64	256	1,024	4,096	16,384
Total Number of Pages	1	5	21	85	341	1,365	5,461	21,845

Figure 5-4: The Effect of Depth Level on Total Number of Searched Pages

The depth-first and breath-first searches must search every page in the search domain before the real-time engine can stop execution. In some cases, a hyperlink might direct the spider to traverse to pages that have very little in common with the search criteria specified by the user. For example, if the topic was based on the feeding habits of eagles and the root page for the search was the front page of a zoo, one of the child pages could easily point the crawler to a page off-topic, such as the zoo sponsor's website or the zoo's human resources website. Such topics do not relate to the topic being search, but are directly linked from the site so they must be searched.

Thus, the issue that resides with depth-first and breath-first traversal methods is that they inherently will search all pages irrelevant of the page's topic. For searches that are specified to reach a shallow depth level, this may not be much of an issue. But certainly for searches that would like to traverse deep into a website or the web itself, some sort of pruning would have to occur to remove the non-topic relate web pages. In an article titled "Local Searching the Internet," the authors describe the concept of just searching one website or the site's neighboring sites [Angelaccio02]. It is similar to commercial search engines allowing a search to take place in one web domain. Localized searching is easy to control by filtering pages according to the web domain in which they reside. The only downside to this technique is that the crawler has a limited search domain according to the web domain specified in the search query.

Instead of eliminating pages because they do not fall within a particular domain, the concept of eliminating individual pages from the search domain based on content was incorporated in this research. This is termed as "pruning" the search domain. Pruning utilizes the breath-first or depth-first search methodology, but filters pages based on content, not according to the web domain to which it belongs. In this methodology, the spider can crawl outside the web domain specified by the root URL address and traverse to neighboring pages on its own free will. As a page is encountered in the traversal, it is processed to determine if it matches the user's criteria. The filtering process is based on whether the page contains any of the keywords given in the user's query statement. A traversal path similar to Figure 5-5 is created with the pruning traversal.

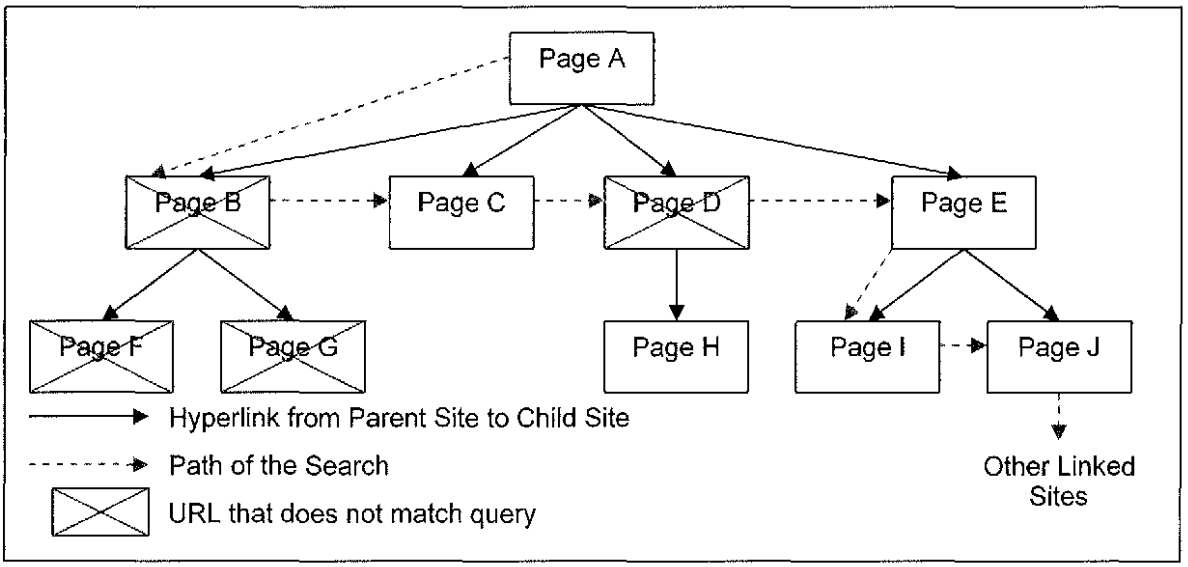


Figure 5-5: Pruning Traversal Path

Thus, those parent pages that have successfully passed through the filter will have their child pages searched. Those pages that do not successfully pass thought the filter are dropped, thus pruning them from the search domain. As branches are pruned from the search domain tree, the size of the search domain will be reduced. Having a reduced search domain decreases the crawlers' traversal and downloading time; thereby improving the performance of the real-time search engine.

An issue of the pruned methodology is determining the number of URL addresses that are returned from a breadth-first/depth-first search and from a pruned search. Since a smaller number of pages are searched in pruned search, it is possible that the result set will be less than the results of depth-first or breath-first traversal techniques. Consequently, the quality of the pruned-based search is tested during execution as well as the performance.

Chapter 6

Real-Time Caching

Every HTTP request to a web server creates traffic on the web. The more data traffic that appears on the web, the longer it takes to send or retrieve data. As a result, high traffic will cause high network latency. Network latency refers to the amount of time it takes for data to travel from one computer to another over a network. To reduce latency, network traffic will have to ease with the decrease of requests, or at least appear to be easing.

6.1 Overview of Caching

Caching is the technique of storing data closer to the requesting computer in an effort to reduce the amount of time it takes to send the data. Caching is implemented in memory systems and hard drives by retaining a copy of highly used data in quick access retrieval systems. As web traffic increases, web caching is used to reduce bandwidth, decrease user perceived latencies, and reduce the load strain on web servers [Davidson01].

As a web page is requested from any web server, the web page may be cached at the web server itself, at the web client, or at any point in the transmittal process on proxy servers.

If another request was made for the same page, the request would only have to travel to the closest cached resource and send it back to the user. Network resources and time are saved as a result of that request not having to be sent directly to the web server. Users notice the improved time it takes to retrieve the requested web page. Web server administrators note that the system is not burdened as much due to a reduction in requests, thus allowing the server to respond to additional requests.

Despite the advantages, there are some potential problems with caching web pages. Just as data indexed in a web search engine can grow old and stale, the pages placed in a caching system can suffer the same fate. In addition, some web resources should never be cached, such as personal information or time-sensitive data.

The second issue can have an increasing effect on latency. When a request comes to a server that performs web caching, response time is decreased if the page is present in cache. This is known as a hit. If the page is not present in cache, it is considered a miss and the request is forwarded to the next cache source or to the web server. Since additional time is taken while verifying a cache miss, the user can perceive an increase in latency as the request is passed from one cached source to another.

6.2 Real-Time Search Cache

Caching, in terms of a web browsing, can take place at the web server, the client's browser, or at a proxy server. Under ideal conditions, caching will decrease network

latency and improve the response time of web requests. If applied to a real-time search engine, web crawlers can utilize a cached resource, when available, instead of retrieving data from the web. If the spider recognizes a change has not be made to a web page that has been visited in the past, the system can opt to have the spider visit a local caching resource to download data. As a result, the network latency is reduced, since data content is not transmitted through the internet via the spider. Response time increases, because the information is retrieved from the local caching resource.

Like web caching, the cache resource for the real-time search engine must be populated with web page data in order to be effective. To resolve this, if the engine is running in a caching mode, any cache misses that are encountered informs the spider to retrieve the page for parsing and processing. In addition, the web page data is stored in cache for future cache requests.

6.3 Implementing Real-Time Search Caching

Through caching, the engine can simply retrieve the parsed data from the DBMS instead of downloading and parsing the page from the web server. The disadvantage is the originator page of a cached page could change over time; thus, the page in the database would not accurately represent the new page. The engine resolves this issue by retrieving the last time the page has been modified from the originator web server. If the page has been modified, it will be downloaded, parsed, and used to update the cached version in

the database. If the page has not been modified, the system will retrieve the page from the DBMS and eliminate the expensive time of downloading and parsing the page.

If the page was encountered during previous searches, the page's last-modified time from the web server is compared to the DBMS's cached copy of the last-modified time. If there is no difference in the time, the page's data content is retrieved from the DBMS and placed in memory for query processing. If the last-modified times are not the same, this means the page has been updated since the cached page was parsed. This requires the DBMS to update the last-modified time and purge all data content associated with the page. Lastly, the system must download the recently updated page, parse it, and cache it back to the DBMS.

On some web pages, the last-modified time field is set to zero. This prevents web browsers and web proxies from caching pages that change frequently. The real-time search engine will have to abide by this convention as well. When such pages are encountered during a cache search, the page data is purged from the DBMS and the page is downloaded, parsed, and cached again[4].

[4] It would be quicker not to cache the page if it changes constantly. But for this research, a database search mode was incorporated into the search engine for testing purposes. Web data must be cached for the database search to function.

Chapter 7

Concurrency

The final method of increasing performance of the web search engine would be in how the application process is executed on the hardware system. Concurrency describes the act of running multiple parts of an application at the same time. The methods about achieving concurrency differ based on language implementation, hardware specifications, and operating systems.

7.1 Overview of Concurrency

Applications can achieve concurrency through multiprocessing or multithreading. A process is made up of a single address space and a single thread of control. The process must maintain its state information such as page tables, swap images, file descriptions, outstanding Input/Output (I/O) requests, and saved register values [Anderson89]. For every process that is executing, a series of these attributes must be created and maintained. In order to generate another execution of process, a new process must be created with all the relevant process state information.

A thread is described as a "single unit of execution within a process" [Gu99, page 34]. Each thread, known as a "lightweight process" [Anderson89], separates the execution portion from the rest of the "definition" of a process. In other words, a single thread is only concerned with its executing instructions and the program counter associated with those instructions. All other thread information is maintained in the process that encapsulates the thread.

Thus, multiple threads would be the several threads executing concurrently within a single address space contained in a process itself. In this scenario, a thread will execute a portion of the program while cooperating with other threads in the use of common processing attributes like page tables, swap images, and outstanding I/O requests. Anderson argues that sharing these common process attributes can provide a reduction in overhead [Anderson89].

On a uni-processor system, multithreading is used as a programming structure aid and to overlap I/O requests with additional processing. The only type of management needed is thread creation and switching between threads during execution, known as context switching. Anderson mentions that locking is not a factor in a uni-processor system because only one thread is executing at a time.

On a multiprocessor system, multithreading can exploit the resources of a multiple processor environment to create a true parallelism. Performance is achieved through thread management as threads are assigned to run on available processors. Locks are

implemented on process resources that the threads share in an effort not to corrupt the execution. Inadequate management of these locks can severely hamper performance.

In an evaluation of multithreading in the Java environment, Gu et al. determined how multiple threads can impact an application. They observed that performance was improved with multithreading over single threading through attempts "to mask communication latency and system overhead operations" [Gu99]. This meant that during the idle time of one thread, another thread can utilize the down processor time to accomplish its tasks. Idle times can occur during requests calls for information from a hard drive or network resource. In an application with a number of these types of communication latencies, the application could continue to execute threads while some threads are waiting for I/O.

Gu et al. noted that execution of multiple threads differed in a Java implementation based on the operating system. This is due in part to the manner in which Java threads are matched to the native running threads of a multithreaded capable operating system. The authors also discovered a threshold at which too many threads impeded the performance of the system. This threshold varied from one operating system to the other, but the results were evident. Heavy use of excessive threading can degrade system performance due to the time wasted in thread management [Gu99].

7.2 Example of Concurrency

Researchers have noted that threads improve performance by allowing functionally independent tasks within a program to execute in parallel in several different applications. For example, Lee et al. described how the Common Object Request Broker Architecture (CORBA) is improved by using multiple threads [Lee01]. CORBA allows clients to access objects running on a remote machine. Without threading, a client would have to wait until the object was not in use in order to perform its task. In a multithreaded environment, the Object Request Broker would spawn a thread to handle an object request from each client. This allows the clients to operate on an object in parallel; therefore, improving response time.

Unlike CORBA, the real-time search engine does not use threads for client connections. Since the goal of the application is to parse pages in real time, dividing the execution tasks can have a great influence on performance. In a multithreaded environment, threads can be employed to execute tasks that are functionally independent of each other. By creating threads that implement retrieving, parsing, and searching HTML pages, the information can process much faster. Essentially, the search engine employs additional threads to accomplish the tasks in a shorter period of time.

Among the many uses of threads, web search spiders can be implemented in a multithreaded fashion to increase retrieval time of web content [Moody03]. Since the main purpose of a web spider is to download web content, the execution of the spider depends on locating and retrieving websites, which is one means of I/O communications.

The spider must wait for an HTTP response from the web server that could take any where from a couple of milliseconds to several seconds. During this wait time, another spider thread can easily be retrieving content and proceeding with parsing the HTML content of the web page.

7.3 Necessity for Concurrency in Real-Time Search Engines

The real-time search engine prototype application was a single threaded application. This means that the application downloaded, parsed, and matched a web page one at a time to accomplish its task. The spider would download content. The hyperlinks of the page would be placed into a queue. The content would be matched against the query conditions. The process would repeat until the application came to the proper depth level and suspended execution.

There were a couple of these functions that could have been performed independently of each other. Due to fact the program ran in a single thread, processing cycles were wasted during network I/O operations and database SQL calls. During this time, it was important to notice benefits of utilizing threads. The application can be modified so when a thread is idle and waiting on an external process to occur, another thread can execute during the wait time. Thus, the entire application is not waiting on one execution event.

In a modification of the original engine prototype, the spider function and matching function were separated into two threaded objects, but the method used ran multiple spiders to download multiple pages at once. After downloading all pages, the multiple matching engines were executed to match the page content to the query statement; thus, retrieving the final result set. As a result of this method of threading, the multiple spider engine threads did not run concurrently with the multiple matching engine threads. Unfortunately in the prototype, it was unclear whether a multithreaded environment had any affect on performance, due to the implementation of an enhanced memory-based, binary tree in the search engine.

7.4 The Multithreaded Real-Time Search Engine

In this research, the real-time search engine is executed on a multiprocessor system. In combination with this hardware and a multiprocessor operating system, more threads can be spawned onto the individual processors in the system and increase the number of pages that are concurrently being downloaded. In traditional search engines, multiple processors and the spawning of several spider threads would improve the number of pages downloaded during any given time period. Since spiders run in real time using a real-time search engine, multiple spider threads running concurrently should reduce the overall wait a user has to experience during a query.

Chapter 8

The Process of the Real-Time Search Engine

The following chapter describes the process of how web content is retrieved from the web and processed in the real-time search engine system.

8.1 Setting the Environment for the Search

The web search needs several pieces of information to direct it on its world wide journey through the web. The information is given to the engine in the form of a modified SQL statement. An SQL-like statement allows a user with a passing familiarity of the relational SQL to easily create statements. The statement is designed as follows:

```
SELECT search_mode [WITH PRUNING]
FROM URL address[, URL address]
WHERE tag = "keyword" [{and, or} tag = "keyword"]
DEPTH search_depth_of_adjacent_sites
THREAD number_of_threads;
```

The engine must first know a mode to run in. This is specified in the SELECT clause. The function of this clause states how the engine should traverse the web and which resource it should search (i.e. web, cache, or database). A listing of the search modes are found in Table 8-1.

Breadth-First Traversal		Depth-First Traversal	
Search Mode	*Description*	*Search Mode*	*Description*
breadth-web	Web search	depth-web	Web search
breadth-cache	Web search with cache search option	depth-cache	Web search with cache search option
breadth-database	Complete search of the cached database with no web search	depth-database	Complete search of the cached database with no web search

Table 8-1: Search Modes for the Real-Time Search Engine

The next phrase is the optional WITH PRUNING clause. This statement informs the search engine to eliminate pages from the search domain if they do not satisfy the search query. The pruning methodology is one of the performance enhancements being examined in this research.

The FROM clause gives the starting point, also known as the root in this application, of the search, just as it would provide the source tables to search in relational databases. These pages should also be the topic authority pages for the search domain.

The WHERE phrase specifies the search conditions of the query. This statement allows the user to string search conditions together with "and" or "or" conjunctions. This feature provides the programmer with greater syntactical flexibility and the ability to create refined conditional statements for searching.

The DEPTH clause provides a stopping point for the web spider. As discussed in the Chapter 5 for traversing the web, a depth level must be specified. If not, the spider will simply continue to traverse through the web until it encounters no more pages to search. The depth level permits the user to specify when to discontinue the search and allows for an eventual end to the searching process.

Lastly, a THREAD clause was added to the statement to allow the user to choose the number of spider threads that should be spawned during execution. The performance effects of this clause are examined in Chapter 9.

8.2 Spider and Parser

To achieve the performance sought, a real-time search engine was developed using threads. During development of the project, the spider/parser and matching routines were separated into two object threads to achieve better performance. While testing the two threaded objects, it was apparent that only one query processing thread (or matching thread) was needed to keep up with 5, 10, and even 20 spider threads. It was decided at this point that the execution time of the parser was too small to warrant a separate thread. In the final version of the search engine, the query processing functionality was merged into the spider thread[5]. In essence, once the page has been downloaded and parsed, the page's content is immediately matched against the query conditions to determine if the page satisfies the query.

[5] This was also implemented to resolve inconsistencies in the parsing algorithm.

Another earlier design consideration was the separation of the spider and parser. The original intent was to separate the spider and the parser into two independently working threads. Since the earlier trial versions of the real-time search engine downloaded the text during the parsing phase, any attempts to separate the processes resulted in longer execution times. As a result, the final release of the real-time engine kept the content retrieval feature and parser feature as one threaded spider object. An overview of the combined processes can be seen in Figure 8-1. In the end, merging the content retrieval function and the parsing function did not remove the multithreading aspect from the application. The multithreaded feature of the application is utilized in spawning several spider threads to concurrently download, parse, and process web pages.

In the process diagram in Figure 8-1 on page 58, the system starts with retrieving a URL address from the hyperlink queue. This queue sustains the life of a spider thread by giving it a URL address to download and parse. When the queue is empty, the thread dies. In a multithreaded implementation, the hyperlink queue must sustain several threads running concurrently. In order to prevent the application from becoming corrupted, a lock is placed on the queue. This allows only one thread to add or remove URL addresses at a time. The queue becomes a communication link between the spider threads, allowing them to know what pages have been downloaded and which pages to download next.

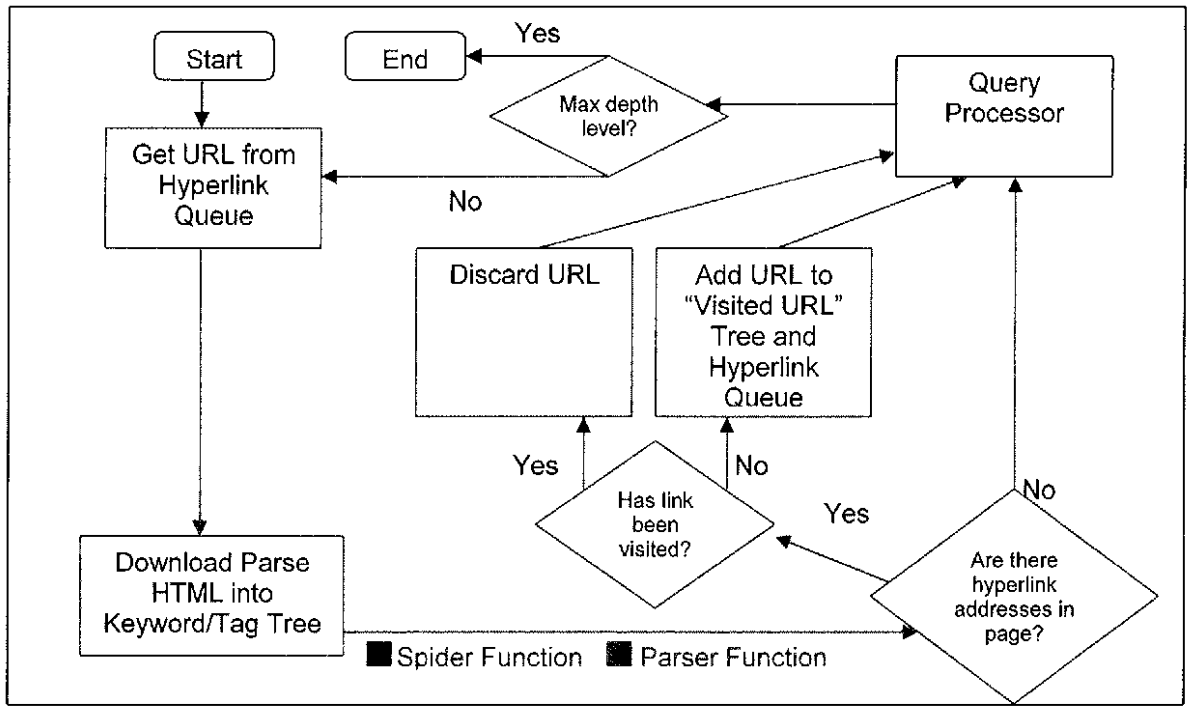


Figure 8-1: Flow Diagram of the Spider and Parser Function

In this process, the program traverses the web starting with the primer URLs which are given by the user in the query statement. The main thread inserts the primer URL addresses, from the FROM clause, into the hyperlink queue. As mentioned earlier, the hyperlink queue houses the links that the real-time search engine must search. This list grows as anchor tags (i.e. hyperlinks) are encountered in the HTML code during parsing.

The final function of the spider described in Figure 8-1 is eliminating web pages which the search engine has already searched. This issue was mentioned earlier as a way to prevent the engine from getting caught in a recursive and infinite loop of interconnected web pages. This only pertains to URLs that were visited during the current execution. Subsequent executions using the same query statement will always process a web page at least once. The process uses an AA binary tree [Wiess99] of URL addresses which were

already downloaded, or visited. In order to make sure the spider does not parse a previously downloaded page, the balanced tree is consulted. If the URL is present in the tree, the address is discarded. If not, the page is downloaded and the URL is added to the tree.

A data searching algorithm can have a tremendous impact on performance. Although the method for "in-application" searching is required in this application, an efficient and speedy implementation was used. Such a need arose when the application had to ensure it was not parsing the duplicate pages. Since a recursion prevention algorithm was needed, it is reasonable to assume that the application would be storing anywhere from one to four thousand hyperlink addresses during a given search. Memory space and time were considerations in determining the proper algorithm. The first option was to try a database management system to house the addresses of visited pages. But it became obvious that there was an overhead to contend with every time the application had to consult the database.

To reduce the overhead time associated with the DBMS, the search engine was modified to use a memory resident, balanced binary tree-based search, using an AA Tree data structure [Wiess99], to match HTML pages to the query statement. The tree-based search algorithm replaced every instance where a search was required. This included the web page recursion prevention and the query processor. In pre-development trials, the memory-based search responded 60% faster than a database-search algorithm. In

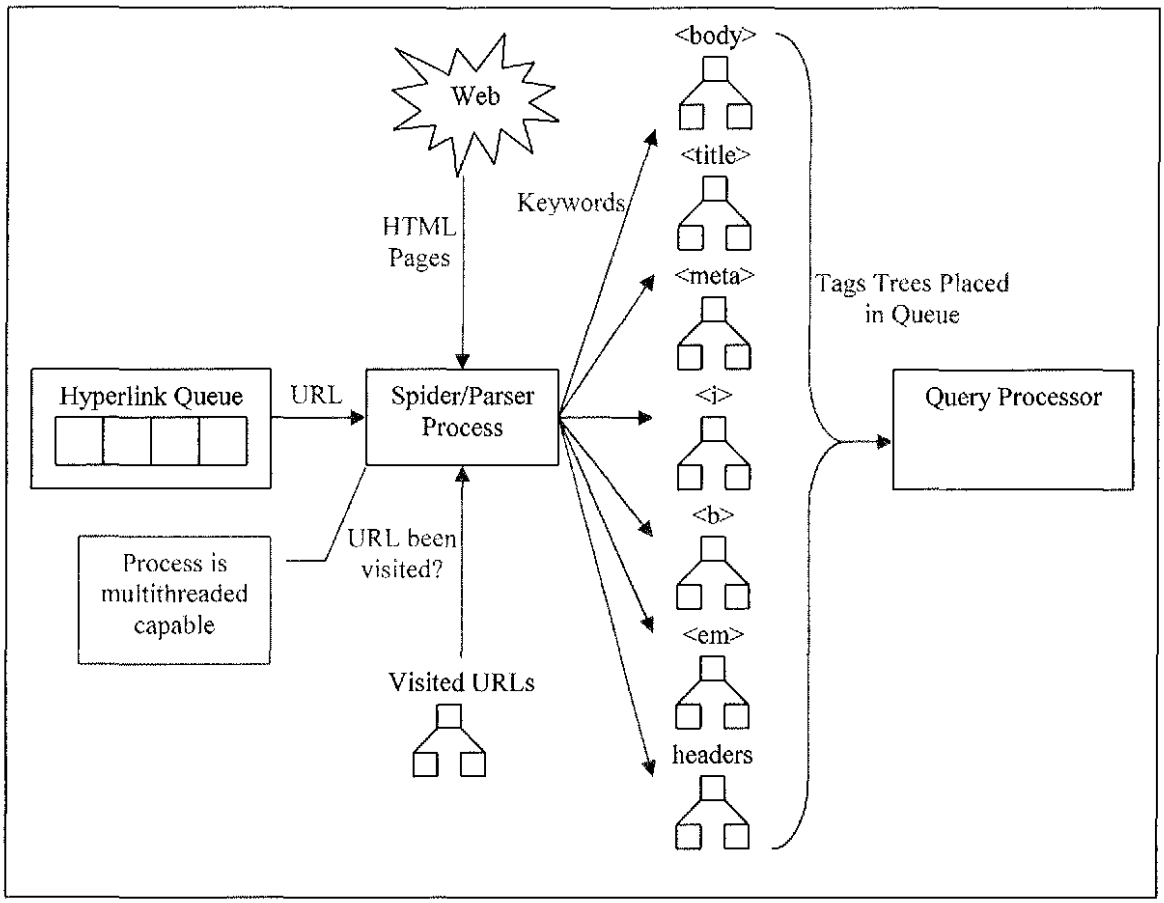


Figure 8-2: HTML Parser Process

essence, the SQL system call was replaced with the AA Tree data structures and significantly improved the search algorithm performance.

The parser function is responsible for decomposing a HTML page into keywords that are associated with HTML tags in the document. This function can be seen in Figure 8-2. The parser first retrieves the HTML text from the HTML text queue. The parser sorts the information into seven valid categories, or "buckets," according to the text's surrounding HTML tags. The valid categories are based on the search HTML tags from the query statement (i.e., <title>, <meta>, <body>, <i>, <b>, , and the header tags).

The category buckets are implemented using the balanced AA Trees. The tree based structure is utilized for its efficient searching features. When the page has been fully parsed, the tag trees are bundled with the page's URL and sent to the query processor. If the depth level has not been reached by the spider, the parser continues to download HTML text from URL addresses given by the spider.

8.3 Conceptual Transversal of the Web

Chapter 5 highlighted the traversal techniques of the breath-first search, depth-first search, and pruning-based search. An overview of the traversal process can be seen in Figure 8-3.

The basic premise behind traversing the web was to parse a page to see if it matched the WHERE clause conditions in the real-time search engine query. As the parsing executes, the application records the page's hyperlinks (links to child pages) in a queue.

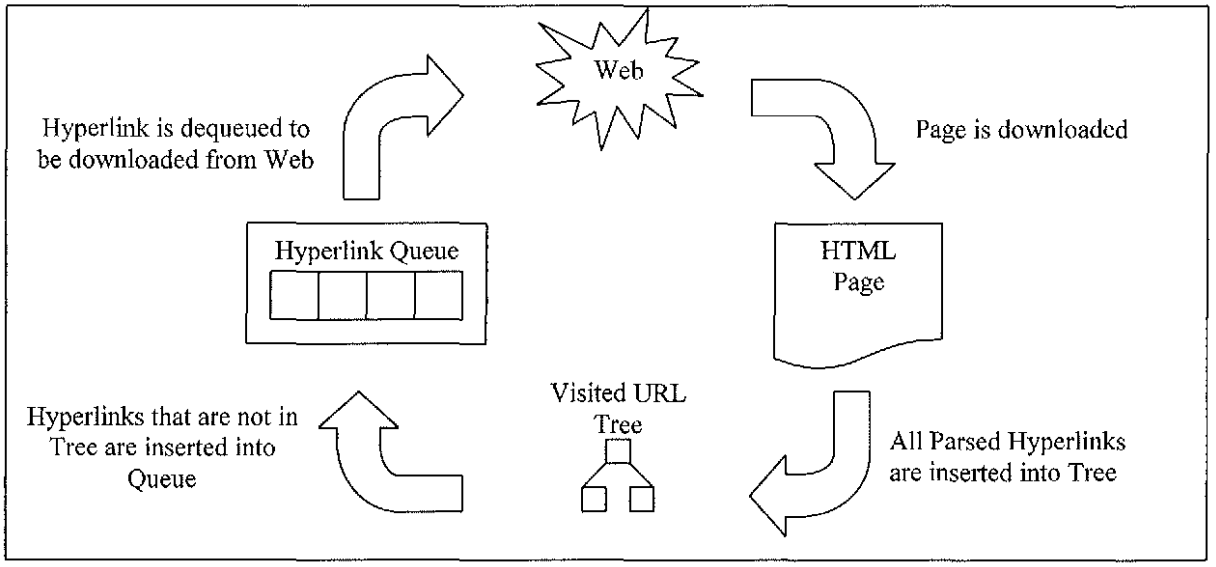


Figure 8-3: Lifecycle of Processing Hyperlinks During Traversal

Depending on how the hyperlinks are stored in the queue, it will affect the search methodology. By inserting the hyperlinks at the end of the queue, the traversal will be breath-first. If the hyperlinks are inserted at the beginning of the queue, the traversal will be depth-first. To achieve pruning, the page must be processed by the query processor. If the page satisfies the conditions in the query, its hyperlinks are stored in the queue for retrieval and processing. If not, the page is discarded with no child pages being queued.

Inserting addresses into the hyperlink queue is not the only part of the traversal process. When a spider is ready to download and parse another page, it retrieves a URL address from the front of the hyperlink queue. Retrieval of the addresses from the hyperlink queue always occurs at the front of the queue in breath-first, depth-first, and pruning traversals.

Due to the linking nature of the web, any search engine spider could easily download pages that have been previously visited. This results in a waste of processor cycles and possible encounters of recursive loops of linked pages. If the issue was not resolved, the best case scenario would have the application parse just a couple of previously encountered pages. Worst case scenario would occur if the application got into an unending loop of linked web pages.

To prevent the problem from occurring, when a page has been downloaded and parsed, all child URL addresses are inserted into an AA Tree designated as the Visited Link tree. The tree is consulted every time a child link is inserted into the queue. If the URL link is present, the child URL link address is discarded. As a result, all recursive hyperlink paths in the search domain are removed from the spider's traversal path. The spider will not traverse a path or parse a page that is it has visited.

As a child URL link enters the queue, it receives a level number which represents its depth in the queue. The depth is calculated by adding one depth level to the current parent page's depth level. As the child pages are pulled from the storage queue for parsing and matching, the page's depth level is analyzed. If the current depth level is greater than the level in the DEPTH clause in the user's query statement, the traversal process ends. If the depth level is less, the entire transversal process continues by parsing the link and discovering its child links and HTML/Keywords tag associations.

8.4 Parsing HTML

For the research, a very simple method for categorizing web content was used. The underlying principle of the categorization was inspired by WebLog's tag/keyword pairing. Just like WebLog [Lakshmanan96] had certain tags that would be used in a tag/keyword pair, the real-time search engine obtains keyword content by associating the word with the HTML tag encompassing it.

The keywords are searched according to which HTML tag they are paired with in the web page. For this research, the HTML tags that can be searched are <title>, <meta>, <b>, , <i>, <body>, and headers (<h1>, <h2>, etc.). While parsing a page, the engine searches for one of these HTML tags. When tags are encountered in the document, the contents residing between the start tag and end tag are associated with the HTML tag. Since the content between the tags can consist of several words, the parser has to separate the content into individual words.

For example, if the title of a web page is "University of North Florida," the words "university," "of," "north," and "florida" will be contained between <title> and </title> in the page's HTML. All four words are paired with the TITLE tag. This pairing will then be used to determine if the page matches the user's search conditions specified in the web query.

If one HTML tag is found inside another (i.e., <body> could have several <i> tags between the <body> and </body> tags), the content is parsed into the inner most tag. For

implementation, a stack data structure is used for association keyword content with HTML tags. The only exception to this pair rule is with the tag <body>. Since the bulk of the web content falls within the BODY tags of a HTML, all content is associated with the BODY tag. If keyword content is contained within other tags, the keyword is associated with that tag word and with the BODY tag. This was done to allow the BODY tag to be a "catch all" tag for the search engine. If it is not between the BODY tags, the keyword is either not in the page or not being displayed to the web surfer.

8.5 Query Processing

The last major component of the real-time searching engine is referred to as the query processor, also known as the matching function. The query processor is used to discover if the page satisfies the user's query. A process overview of the matching thread can be seen in Figure 8-4. The matching process is fed the HTML content of a particular web page from the spider's parser. For each parsed page, the categorized HTML content is passed from the spider to the query processor as a set of "tag" trees, which are AA Tree data structures that house data content associated with a HTML tag. For each condition specified in the query statement, the tag tree is checked to see if a keyword is found. For example, if the user writes:

```
WHERE body = "care" and title = "dog"
```

the matching process searches the BODY tag tree for "care" and TITLE tag tree for

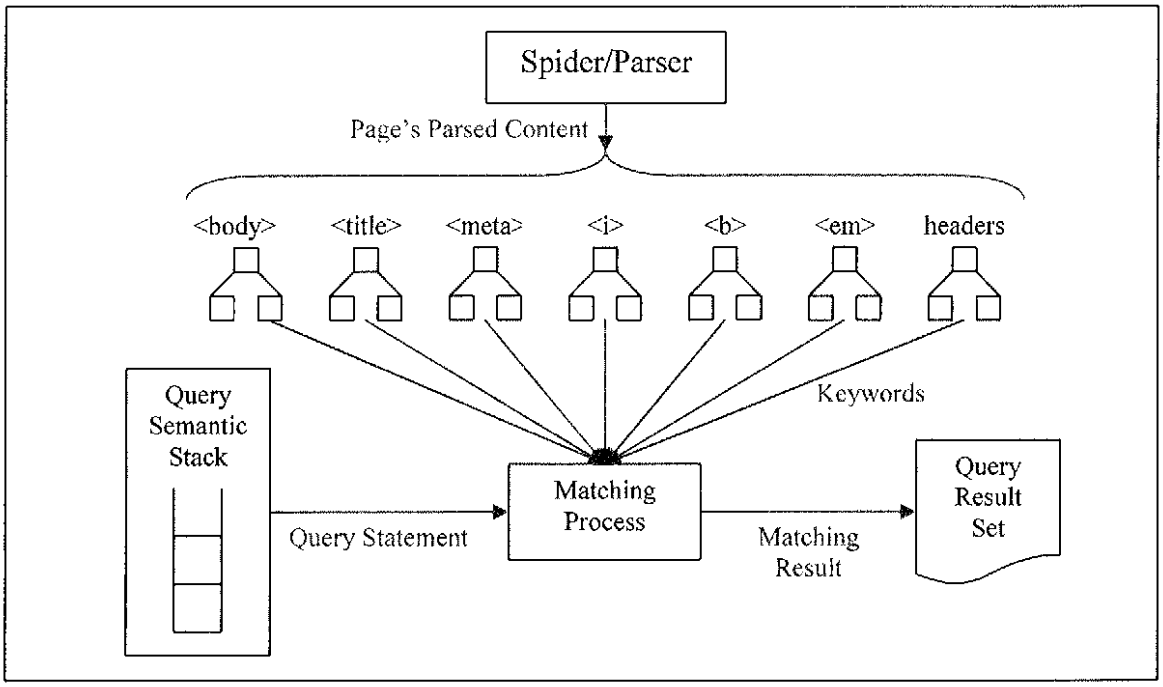


Figure 8-4: The Query Processor

"dog." Both conditions would have to be true in order for the page to be valid and placed into the result set. If there is not a match, the URL is discarded. The process will continue until the depth level is reached.

The tag/keyword pairs from the WHERE clause are placed in a conditions list along with any conjunctions and parentheses. By incorporating conjunctions and using parentheses into the SQL statement, the condition clause can be manipulated and arranged according to the user's search needs. Just like the parser utilized a stack to associate HTML tags with HTML content, the matching thread uses a stack data structure to evaluate the conditional statements. The evaluation order is similar to Mathematics' Order of Operations, which is used to determine how expressions are evaluated. This ensures that conditions in the parentheses are evaluated before the outer conditions. During

evaluation, the tag specified in the condition must be matched up to the tag tree with which it is associated. Once that has occurred, the tag tree will be searched to find the keyword. If the word is not present, a false flag is returned; else, a true flag is returned back to the calling program.

When all tag/keyword pairs have been evaluated to true or false, the conjunctions must be evaluated. If there are no conjunctions, then only one tag/keyword pair was present and the result is returned to the calling function. When conjunctions are present, the query processor uses the same order of operation comparisons as standard SQL. AND conjunctions are evaluated first, followed by the OR conjunctions. As mentioned before, parentheses can also affect the order of comparison. Once all conjunctions between tag/keyword pairs have been evaluated, only one Boolean flag should remain. That flag signals the application as to whether the page satisfies the query statement.

8.6 Web Search vs. Web\Database Search

Before the user starts the real-time search engine execution, they must decide upon one of two different search techniques: the Web-only search or the Web-based search with caching. With the traditional Web search, every page that is examined by the program must be downloaded from the web. This search mode guarantees that the search is as current as possible. The second search technique relies on a combination of the Web search and a DBMS cache search. During this search, if the engine has never

encountered a page, it will be downloaded, parsed, and cached into a DBMS. During the caching process, the page's last-modified time is recorded.

If the page was encountered during a previous search, the page's last-modified time from the web server will be compared to the DBMS's cached copy of the last-modified time. If there is no difference in the time, the page's tag/keyword pairs are retrieved from the DBMS and placed in the HTML "tag" trees (balanced AA Tree data structures). If the last-modified times are not the same, this means the page has been updated since the cached page was parsed. This requires the DBMS to update the last-modified time and purge all tag/keyword pairs associated with the page. Lastly, the engine's spider must download the recently updated page, parse it, and cache it back to the DBMS.

On some web pages, the last-modified time field is set to zero. When these pages are encountered during a cached mode search, the page data is purged from the DBMS and the page is downloaded, parsed, and cached again.

This may seem inefficient to cache a page that indicates it is going to change. As a third search mode option, the application allows for a database search to be executed against the database. The database search mode was created to act as a comparison on how a traditional search engine would perform against the real-time system. It is not surprising that a real-time search engine would take longer to execute than a traditional web search like Google or Yahoo. For the sake of comparison, the database search was developed to

discover the difference between systems that retrieved data from the web versus a system that retrieves information from a database.

Chapter 9

Results and Analysis

The goal of the research is to peruse different techniques to enhance the execution of the real-time web search engine. The enhancements took the form of three modifications to a prototype real-time search engine that was developed by the researcher. The first modification was the method of traversing through the WWW. A breadth-first search, depth-first search, and a pruning search were developed. The second enhancement was the use of threads to create several concurrent spiders to surf the web. The third modification was to develop real-time caching into the search engine.

One important aspect that plays a critical role in the execution of the real-time search engine is the computer hardware system. Because the application is designed to run in real-time, the execution speed of the hardware plays an important factor in performance. As a result, the real-time engine was developed on a server with four 1.5 Gigahertz Xeon Pentium processors with hyperthreading. In addition, the system requires a sizeable amount of memory for housing URL addresses and web page content. The computer hardware contained 8 Gigabytes of memory. The real-time search engine is required to contact internet resources, so an adequate internet connection is warranted. The system was tested over an ATM-based network connection. Lastly, in order to house the cached

version of the web pages, a database management system was utilized. The MYSQL 3.23.58 database system resided on the same hardware as the real-time search engine.

The results collected for this research depended upon the enhancements being tested. In all cases, the timing results of each experiment were the average execution time of five test runs. Performance will be measured upon the number of pages that are returned and the time it took to return those pages. The setup will be discussed in the analysis of each enhancement.

9.1 Limitations of Current Implementation

There are some limitations to the real-time searching engine that was developed. The first issue was the lack of control over the hardware executing the search engine. The engine relied on Java's and the operating system's ability to schedule processing time for each spider. An ideal situation would have been to manage the timing for better performance, but this is not possible under the current implementation. In addition, the system was not solely dedicated to the real-time search engine. System resources were diverted to projects by other researchers.

Since page keywords are stored individually within the engines parsing function, it is not possible to search compound words or phrases. This design decision was made to limit the amount of processing when matching a page to a query. As the process was

occurring in real time, it was important to make sure the matching function performed as quickly as possible.

Due to the reliance on the Internet for real-time searches, many issues developed as a result. First and foremost was the presence of network latency on the Internet. Numerous tests showed that the system performed better when searching a web server on the same network as the machine executing the search. Attempts to search a remote server off site resulted in slower response time, which was not unexpected. Performance is also affected by the amount of traffic that is currently on the network and the workload of the web server. During peak network utilization periods, execution times typically ran longer than expected. As a result, the testing was completed at times when network utilization was typically lower.

Performance was also hindered by non-responsive web servers and inactive domain names. Every hyperlink that is encountered by the system must be checked to see if it is active. In a real-time search, the additional checking resulted in longer wait times for the user. In some cases, the additional processing time only confirmed that a domain was inactive or unreachable.

Another limitation was in how the engine's HTML parser handled poorly coded web pages. In most cases, a poorly written HTML page can still be read by a web browser. In the case of the real-time search engine, only pages that conformed to standard HTML code were parsed properly. A poorly coded web page would be eliminated from the

search. The real-time engine is only able to retrieve data from HTML-based content. Web content such as Word documents, PowerPoint files, and PDF files were not searched by the engine.

Lastly, the engine's spiders are only capable of crawling to openly public and accessible pages. Anything that is located within the "deep web" is inaccessible. This means the crawlers cannot visit any page that requires user authentication or submission of data through a form to get to another part of a web site. In addition, the crawler does not have the capability to search secured pages using the Secured Socket Layer (SSL) protocol. The spiders request the information in plain text, thus eliminating the possibility of collecting data from SSL-based website.

9.2 Analysis of Web Traversal Implementation

The web traversal methods focused on two global traversal searches and one selective path traversal search. The selective path search will be discussed later in this chapter. The two global searches are the breadth-first traversal search and the depth-first traversal search. Both implementations explores the same amount of pages within a given search domain and depth level. Both traversals discover the matching result sets for the same query statement. The difference resides in how quickly the two traversal methods function. As stated in Chapter 5, the depth-first traversal can place a strain on some web servers since it drills down to the bottom of the traversal path. In some cases, that

traversal path will reside on the same web server. If this is the case, then breadth-first will execute faster.

To test the traversal methodologies, two separate queries were executed. One query used a local intranet web server as its primer URL address. The second query used a remote web internet server as its primer address. Due to the proximity of the local web server to the application server, which ran the real-time search engine, it was anticipated that latency would be reduced, thus providing a more accurate performance assessment of the traversal methods. An external web server is also needed to accurately portray an actual search that would occur on web servers throughout the internet.

The tests were arranged to measure how quickly the traversal methods responded to searches at different depth levels. For each depth level, an average of five executions was calculated using 1, 5, 10, 20, 35, 50, 65, and 80 threaded sessions. Not only did this give an idea of how each methodology reacted at a particular depth level, but it also indicated how multithreading affected both methodologies.

In Figure 9-1 on page 76, the results of a breadth-first and depth-first search traversals at depth 1 using an intranet web server as the primer URL are shown. Figure 9-2 on page 76 displays a breadth-first and depth-first traversal at depth 1 using an internet web server as the primer URL. In both figures, the execution trends for both traversals are relatively the same. As the traversal continues down into depth level 2 and 3 in Figures 9-3 through 9-6 on pages 77 and 78, the trends of both methods match each other in performance. As

the trend continues in both Figures 9-1 and 9-2, the depth-first line closely follows the breadth-first trend, and at some points, slips below the breadth-first traversal in terms of response time.

A cause for some slight changes in the execution could be a result of network latency. But in most cases, it is believed that poor server response time is the cause. As mentioned before, the main difference between the breadth-first traversal and depth-first traversal is the order in which the engine encounters web pages. The breadth-first visits all pages on the same depth level before continuing to the next lower depth-level. The depth-level search will drill down to the bottom of the search domain and traverse in a vertical pattern up and down the search domain.

The time it takes to process a page is highly dependent on responsiveness of a web server to the crawler's request. If the crawler encounters a page early in the traversal process, the slow response time appears not to be a factor as the engine is crawling over several pages after it. If the crawler encounters a slow response page late into the traversal process, the overall execution time will lengthen as the engine waits for the final web page to be processed.

As a result, the traversal process is dependent on the response time and the location of the web page in the search domain. In the end, the traversal process that encounters the slow response pages at the beginning of the search will perform better.

Despite this issue, the traversal graphs in Figures 9-1 through 9-6 indicate there is not much of a difference in traversal performance between the breadth-first and depth-first searches in ideal conditions. In the worst case scenario, poor response time from a web server near the end of the traversal can lengthen the execution time of both traversal methods.

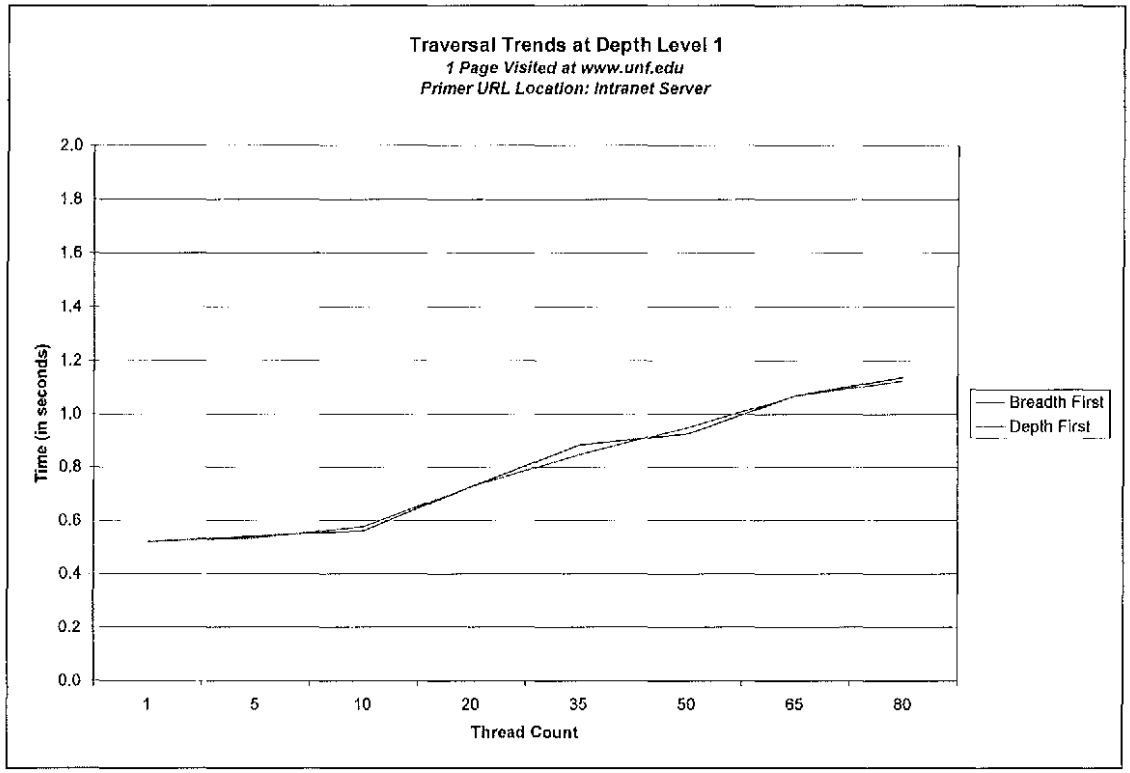


Figure 9-1: Execution of Global Traversal Methods at Depth Level 1 (Intranet)

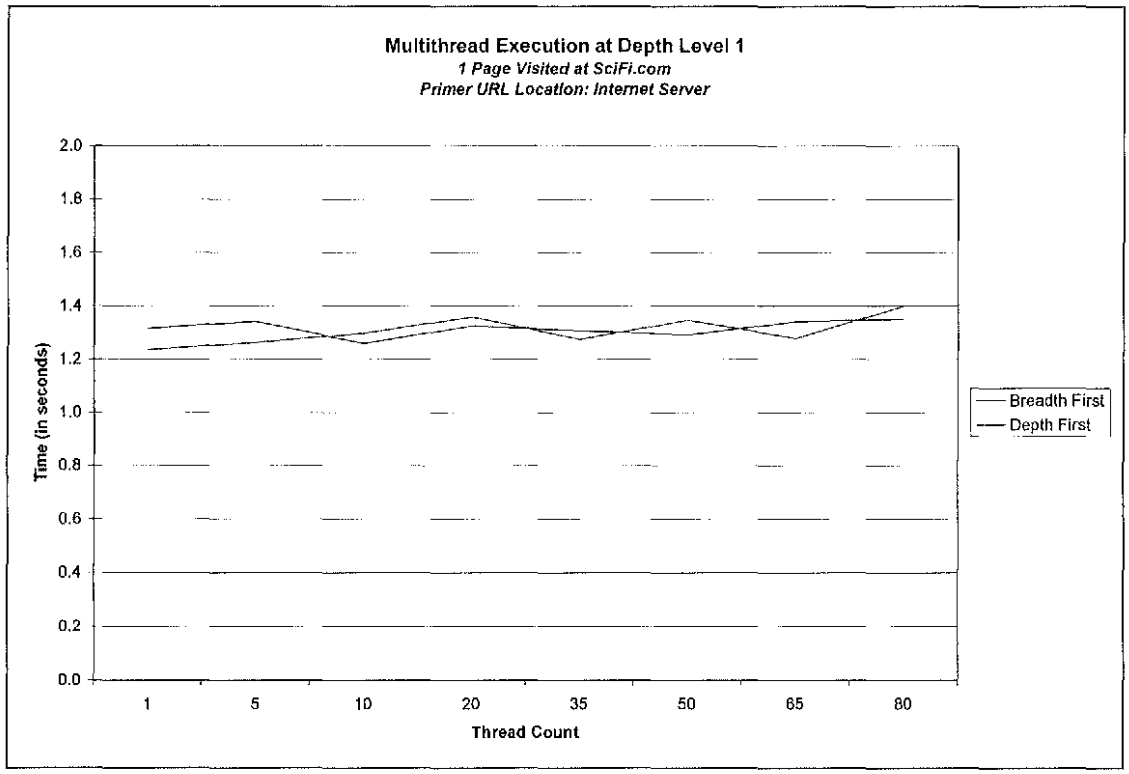


Figure 9-2: Execution of Global Traversal Methods at Depth Level 1 (Internet)

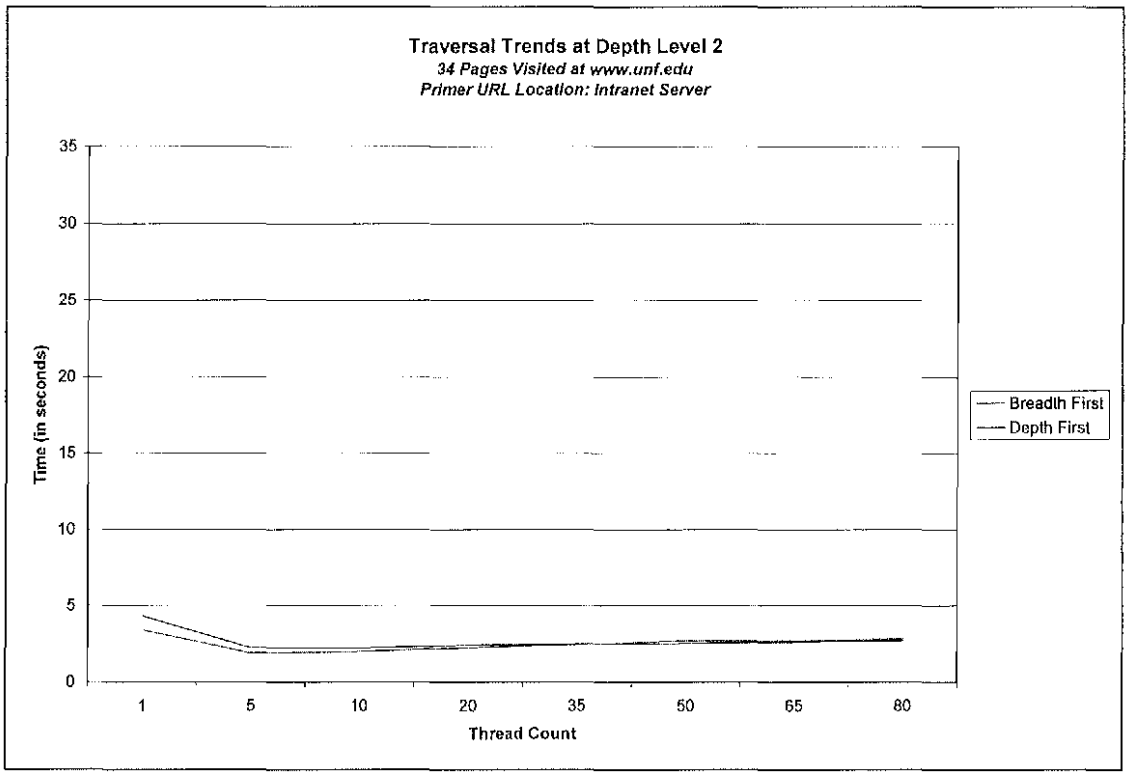


Figure 9-3: Execution of Global Traversal Methods at Depth Level 2 (Intranet)

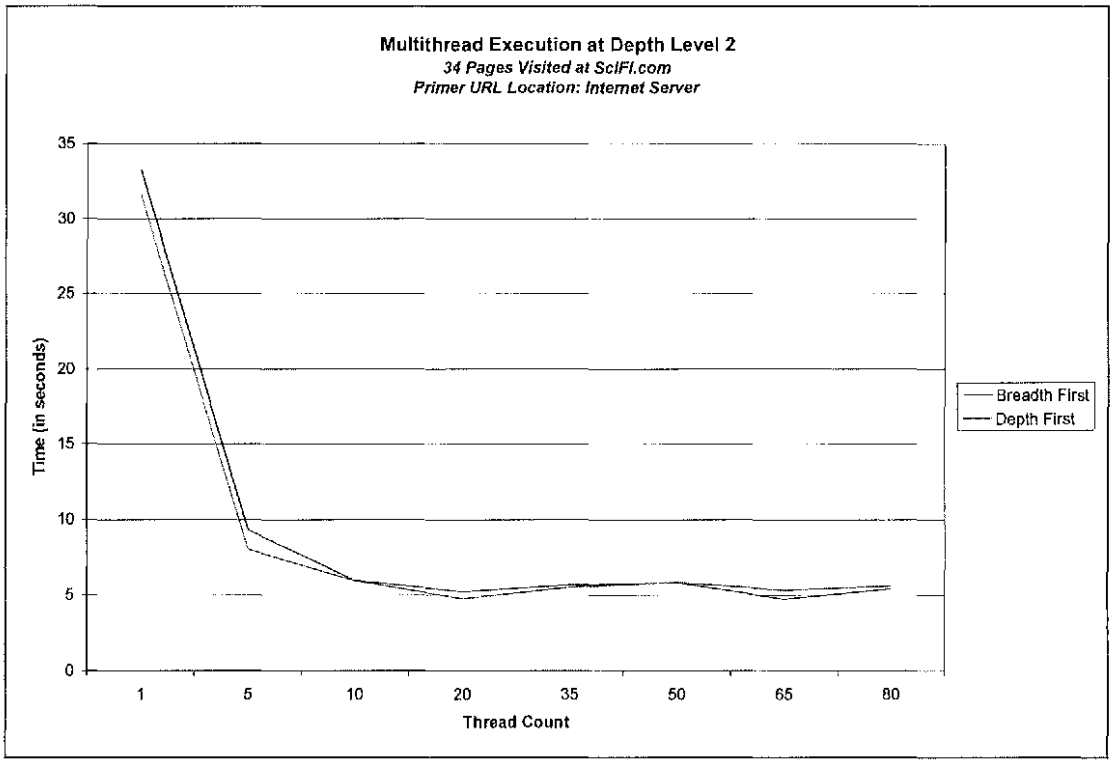


Figure 9-4: Execution of Global Traversal Methods at Depth Level 2 (Internet)

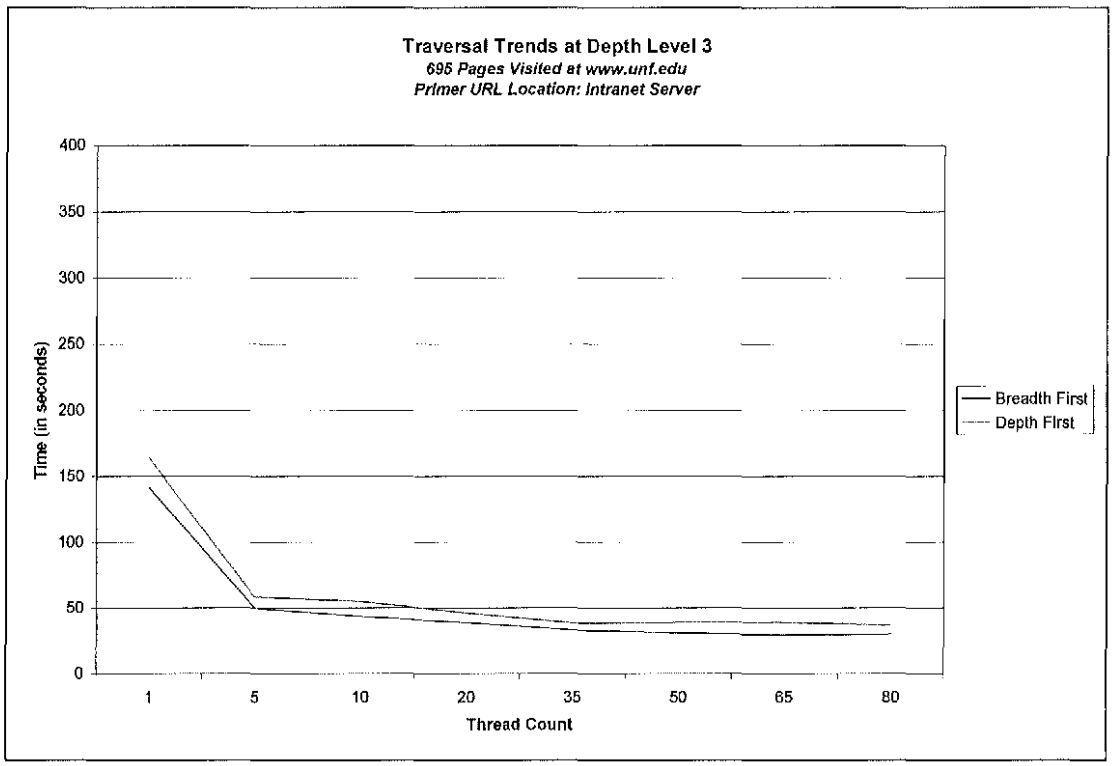


Figure 9-5: Execution of Global Traversal Methods at Depth Level 3 (Intranet)

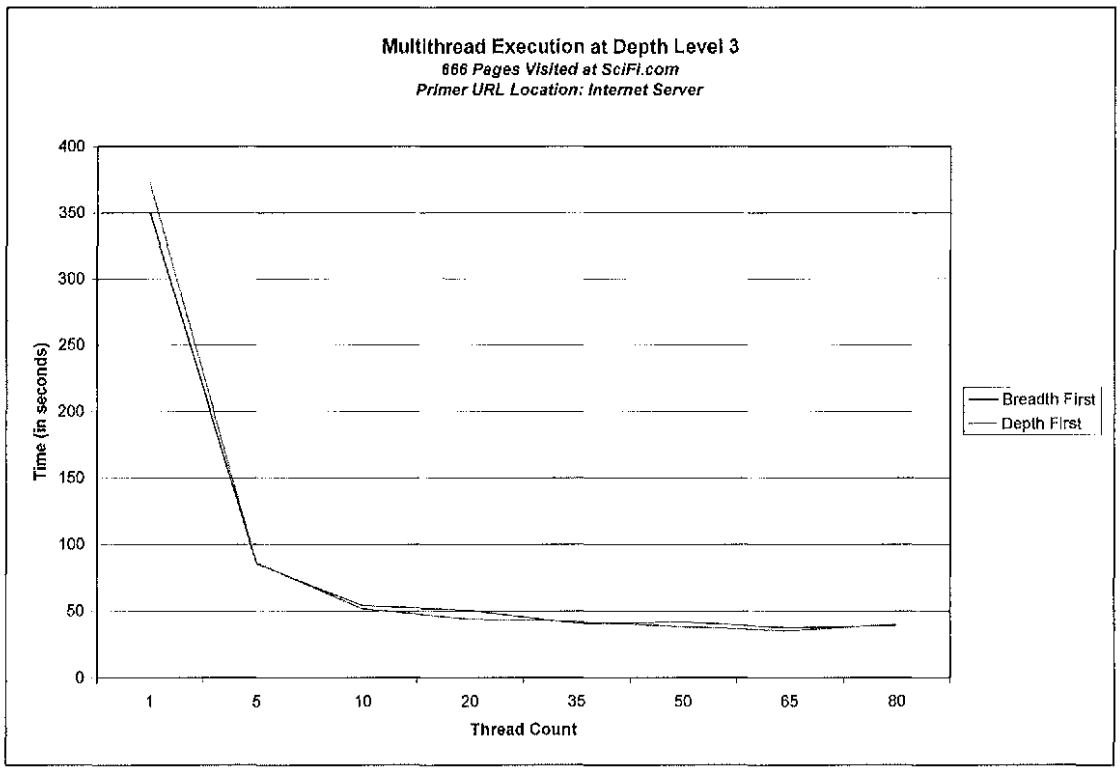


Figure 9-6: Execution of Global Traversal Methods at Depth Level 3 (Internet)

9.3 Analysis of Multithreaded Implementation

With breadth-first and depth-first traversals implemented, the next step was to test the effects of multithreading of the real-time search engine. The test was setup similar to the traversal testing procedures. Two URL primer addresses, an intranet web server address and an internet web server address, were used to examine how geographic proximity affected the engine's response time. Both URL addresses were searched at depth levels 1, 2, and 3 using thread counts of 1, 5, 10, 20, 35, 50, 65, and 80. An average of five test runs at each depth level using the same number of threads were conducted. The results can be seen for the local web server in Figures 9-1, 9-3, and 9-5 on pages 76 to 78. The results for the remote web server are in Figures 9-2, 9-4, and 9-6 on pages 76 to 78.

Multiple depth levels tested how the engine threads responded under different loads of web pages. The lower the depth number meant the fewer pages that would have to be parsed. Fewer pages meant the search engine would complete faster, but the execution time would depend on the number of threads specified in the user's query. It is believed that the system would respond faster at a lower depth level with few threads due to less overhead in managing fewer threads. The higher depth level would draw in more web pages to process, and thus, require additional threads to process them.

In Figure 9-1, the local network web server at depth level 1 clocked at about .55 seconds utilizing one thread. The trend gradually increased going from using 5 threads at .55 seconds to using 80 threads at almost 1.2 seconds. With the remote web server network in Figure 9-2, the results were consistent at 1.3 seconds while utilizing 1 thread to a group

of 80 threads. Clearly, multithreading had an effect on the local network web server at lower thread counts. Since only one page was parsed at depth level 1, the additional overhead with starting multiple threads and then stopping them began to take its toll. Since the local web server response time was so quick, lower thread counts fared better. As the thread count increased, the performance started to decrease as the additional threads hindered progress. With the remote web server at depth level 1, the transmission time of the web page to the spider had a tremendous effect on performance, as the time clocked in at almost .7 seconds slower. It is believed that in conjunction with the slower network response time, the operating system had a better handle on managing the threads. As a result, the performance tended to stay consistent through to the 80 thread count. It is believed that execution time would have risen as more threads were added.

Figures 9-3 and 9-4 at depth level 2 and Figures 9-5 and 9-6 at depth level 3, gave much more expected results with starting at a high execution time with 1 thread and then dropping and leveling out at around 5 or 10 threaded executions. It is interesting to note how much of an improvement occurred running a 5-threaded execution over a 1-threaded execution. Furthermore, the additional threaded executions did not seem to make the dramatic improvement in time as the initial jump from 1 to 5 threads. There is a slight downward trend in all four figures, with a slight upward trend at an 80-threaded execution.

The initial drop in time from 1 thread to 5 threads can be contributed to the additional threads making use of idle time while spider threads are waiting for network I/O. The

slight downward trend from a 10-threaded execution to a 50- or 65-threaded execution is the system trying to utilize the additional idle time that might exist between threads. The slight trend upward at an 80-threaded execution in Figures 9-3, 9-4, and 9-6 could indicate a saturation point where too many threads are starting to impact performance due to thread management or thread thrashing. It is believed that further extensions of the graphs into 100 or 150 threads would demonstrate a loss in execution time and performance.

9.4 Analysis of Pruning Traversal Implementation

The pruning traversal methodology is designed to be a simple selective path traversal search. Unlike the breadth-first and depth-first search, the pruning traversal eliminates the hyperlinks of a page from the searching process if the page does not satisfy the user's query. The effect is similar to pruning a tree in order to cut down the number of branches. By selectively pruning a search domain tree based on page content, the search domain will become much smaller and take less time to search.

The testing was accomplished in four phases. The four phases were a full breadth-first search, a full depth-first search, a pruned breath-first search, and a pruned depth-first search. The goal is to compare the pruned search results with the full searches in terms of execution time and the total number of pages that satisfies the query. Each testing phase was divided into test runs at depth levels 1, 2, 3, and 4. The time for each depth level is the average time of five executions at each of the specified depth levels.

Figure 9-7 on page 83 displays the result of the four testing phases. Figure 9-8 on page 83 displays the page count at each depth level according to the number of pages which satisfied the query and the number of pages processed during the search engine execution. The almost identical results at depth level 1 for all four phases were not unexpected as each traversal method processed only one page. At depth level 2, the results were very similar as two pages matched the query and eleven pages were processed under the four phases. This was expected as well, since the first depth level node was not pruned and its ten hyperlinks were processed.

The difference became clear at depth level 3 where the full search methods encountered ninety-nine pages and matched six pages to the query between nine and ten seconds. The pruning search methods encountered sixty pages and matched the exact same six pages between seven and eight seconds. To have both the full and pruning search methods match the same pages to a query in the same depth is unusual, but possible.

At depth level 4, the pruning search methods executed two times faster compared to the full search methods. The pruning search discovered 363 pages and matched thirty-nine pages to the query. On the other hand, the full search methods visited 1,202 and matched fifty-two pages to the query.

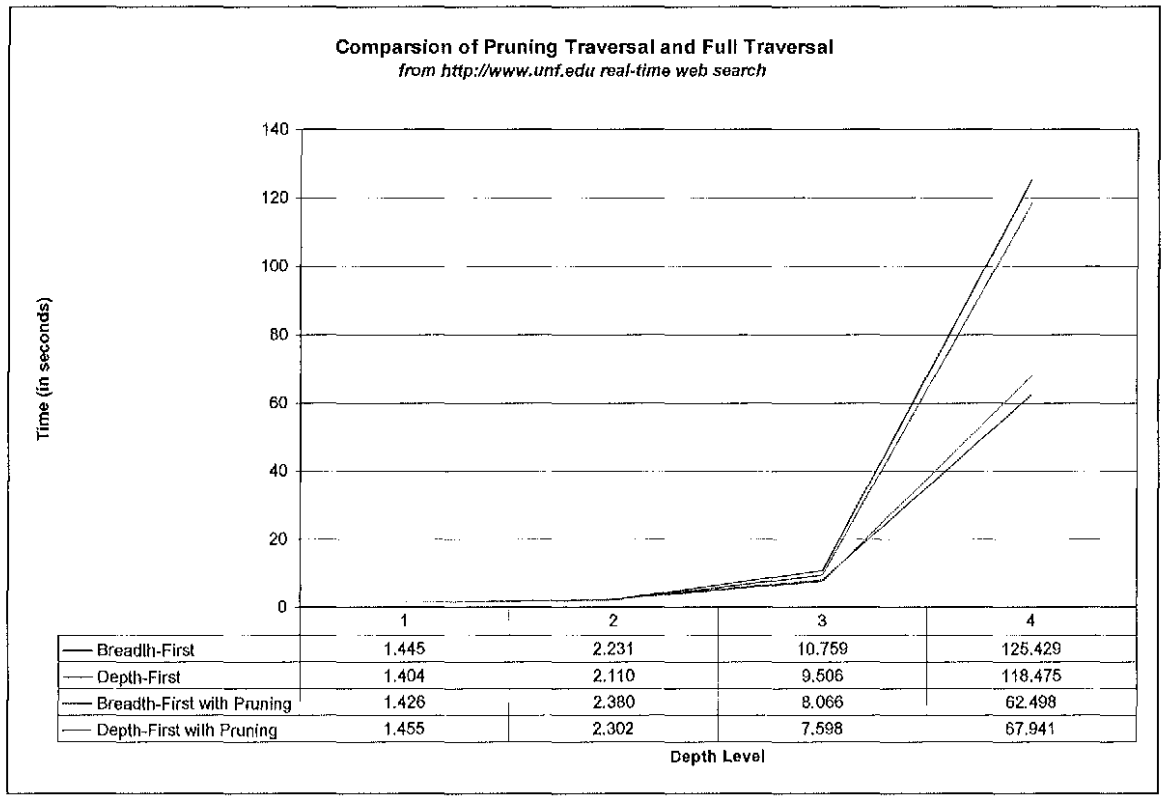


Figure 9-7: Results of Pruning Traversal

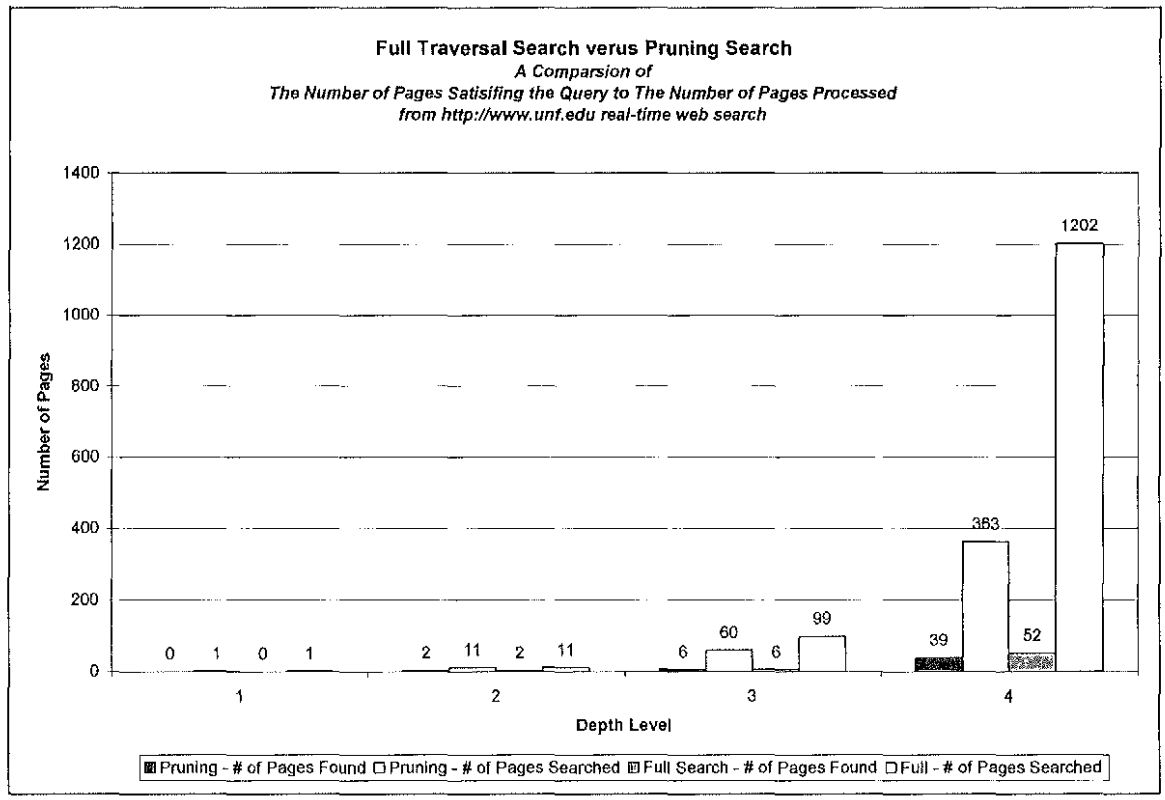


Figure 9-8: Number of Pages Reviewed During Pruning Traversal

Unfortunately, the comparison of results is subjective based on the user's needs. As a result of pruning, the events of the depth level 4 searches are not uncommon. The pruning search does give a faster execution time, but the performance comes at a price. As hyperlinks are pruned from the search domain, it is likely that some valid web pages will be dropped. This causes the number of query satisfying pages to be lower when compared to a full breadth-first or depth-first search.

The issue is to determine whether the increase in performance is worth the cost of losing the additional URL addresses from a full search result set. The quality of the search result set also plays a factor in the success of the pruned traversal. If a pruned traversal found a page, called Page X, six levels deep in the search domain, it is reasonable to assume that the quality of Page X would satisfy the user. This assumption is based on the fact that the pruning search discovered four pages between the primer page and Page X which satisfied the pruning conditions. In short, the "pedigree" of Page X's parents could greatly affect the quality of Page X in the perspective of the user.

In conclusion, the success of a pruned search depends on the needs of the user. If the user is satisfied with a smaller set of quality pages, then pruning would be a valid option. If the user determines quantity is better than quality and wants to take the extra time to perform the search, then a full breadth-first or depth-first search is warranted.

9.5 Analysis of Caching Implementation

Web searching with caching permits the system to retrieve web content from a web resource or from a local cached resource. A test was devised to examine the caching at three different depth levels in the search domain. The cached mode was compared against the results of a web only retrieval search. As a measure to understand how quickly data was retrieved from the local cache resource, a search based completely on cached data was tested. This is known as the "database search."

For each search mode, the average execution time of five execution runs was taken at depth level 1, 2, and 3. The average execution of each mode was compared to determine which methodology excelled. The results of the cached tests can be seen in Figure 9-9.

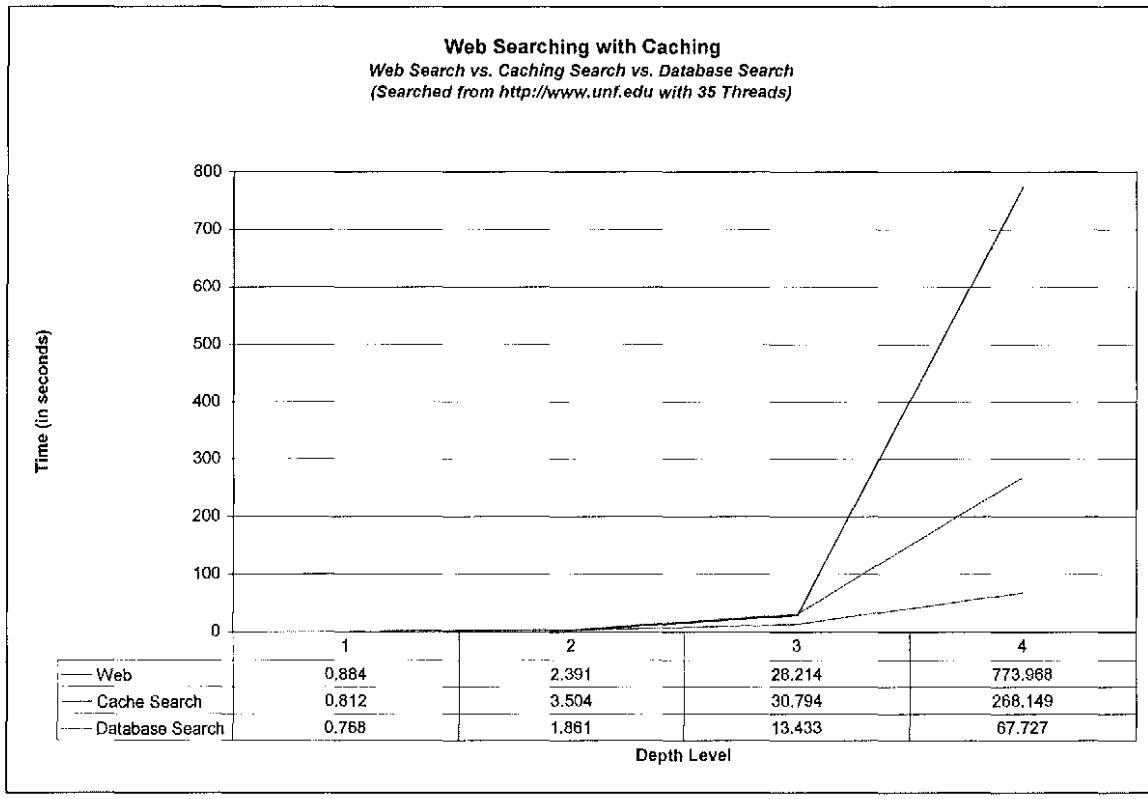


Figure 9-9: Results of Web Searching with Caching

At depth level 1, the execution times for all three modes were almost identical. This was expected as only one page was being processed by the system. At depth 2, thirty-four pages were parsed in each mode. Web search gave a better performance than caching by over a full second. The caching method performed as such because of the additional overhead of caching (database) maintenance on such a rather small number of parsed pages. The results proved the same at depth 3 as 695 pages were parsed. The web search performed better than the cache mode by about 2.5 seconds. At depth 4, where 5,265 pages were processed, the cached search executed almost three times faster than the web search. With this many pages, the web search would have encountered issues with slow servers, network lag, and bad URLs. The cached routine excelled as most of its pages were locate in cache and avoided the "data transportation" issues that the web search faced.

The interesting note in this particular experiment was the database search. At all four depth levels, the database search constantly outperformed the cached and web based searches. In this process, the engine did not request any information from web servers (i.e. a request for a last modified date or download for a page). All information was retrieved from the cached resource, just as if a traditional search engine would request information from a catalog index.

9.6 Comparison with Current Search Engines

The results would not be complete without a brief look at the results of current search engines using similar search criteria from the real-time search engine experiments. In most of the testing, the University of North Florida's homepage (http://www.unf.edu) was used as the primer URL address. The search conditions specified that the page had to have the words "blackboard" and "news" located somewhere in the body of the HTML document.

For testing purposes, two different search features were utilized on each engine (if available). The first feature used the standard form for general content search with no restrictions across the web. The second feature was a general content search within one domain with no restrictions.

On November 16, 2004, a domain-based search was executed on Google, Lycos, Yahoo!, AltaVista, and Ask Jeeves to find pages from http://www.unf.edu with the content "blackboard" and "news" in a web page. Lycos, Yahoo!, AltaVista, and Ask Jeeves could not locate any page from that domain within their index. Google was the only engine to discover 270 hits in an execution time of about .31 seconds.

In Table 9-1, the query for "blackboard," "news," and "unf" was performed using the engines' general purpose search to examine how fast the pages were retrieved and the number of pages retrieved.

Search Engine	Execution Time	Number of Pages
Altavista	*	397
Ask Jeeves	*	89
Lycos	*	123
Google	0.49	516
Yahoo!	0.34	404

* (Execution time was not available, but results appeared in under a second.)

Table 9-1: Execution Times for Commercial Search Engines

The information in Table 9-1 indicates that today's current search engines provide a faster service than the real-time search engine. At a depth level 3, the real-time search engine could locate 47 pages in about 30 seconds. This is expected as the real-time search engine crawls and processes its pages in real time. The pages that were displayed by the web engines were mostly URL's not associated with http://www.unf.edu.

In addition, a domain only search of http://www.unf.edu revealed no results from the search engines reviewed except for Google. This would give an indication that most engines do not perform a search as thorough as the real-time search engine.

9.7 Result Set Quality

Today's search engines rank the quality of web pages according to how well the page matches the query. High quality pages are ranked first in the result lists for the user. In this research, ranking did not play a role in the engine.

In terms of a real-time search engine, quality is affected by searching local areas of the web. By searching an area of neighboring pages with a similar topical focus, the user will be able to locate higher quality web pages. The result sets were examined to confirm that the engine was retrieving pages that satisfied the query. The examination occurred by physically checking the URLs to ensure the query's keywords were in the document.

Chapter 10

Conclusion

The real-time search engine has promise as a localized searching tool. The ability to discover web content in real time can assist users in finding the most up-to-date content in any area of the web. Traditional search engines, such as Google, Yahoo!, and AltaVista, rely on categories of indexed web content for their searches. The content is gathered, parsed, and categorized before the user ever submits a request. As a result, the indexed content can become stale and outdated if not regularly maintained.

Real-time search engines perform the search in real time as the user waits. The benefit is, instead of relying on an index for web content, the web becomes the data repository. Information is retrieved, parsed, and matched directly from web pages. If the page satisfies the user's query, it is displayed for the user.

Another benefit of real-time search engines is the quality of the search results. Because information is retrieved in real time, the search engine must be pointed to a primer web page to perform the search. The content of the primer web page should have significance to the topic the user wishes to search. If the search begins in an area of pages that relate to each other based on similar content, any pages that are found within that search area will give better quality pages that pertain to the user's query.

10.1 Implementing Better Performance

This research focused on the performance of a real-time search engine by examining three different component implementations. The components included traversal methodologies for searching the web, utilizing concurrently executing spiders, and implementing a caching resource to reduce the execution time of a real-time search engine.

10.1.1 Traversing the Web

The first method focuses on a breadth-first search, which searches all pages on a single depth level of the search domain before continuing to the next level. The search pattern resembles a horizontal search through the tree starting at the top node and working down through the different layers.

The second method is known as the depth-first search, which starts at the top of the tree and drills directly down to the bottom node of the tree. The search proceeds to search up and down through the hyperlinks for web content. This search is similar a vertical search pattern as spiders traverse up and down through the tree of web pages.

The third method is a variation of the breadth-first and depth-first traversals. It is a selective path based search which eliminates branches in the search domain by pruning

pages that do not match the user's search criteria. The pruning process reduces execution time, but at the cost of a reduced number of pages that satisfies the web search.

Based on performance, the difference between breadth-first and depth-first traversals was negligible. As expected, both traversals parsed the same number of pages when given the same query. Performance was affected by the placement of poorly responding web pages within the search domain tree. Pruning traversal executed at a faster rate than the full searches, but came at the cost of a reduced result set. In the end, the pruning perform depends on the needs of the individual user.

10.1.2 Concurrency

A single spider can certainly handle the job of searching through a web domain. The execution would be time consuming, but the task would be completed. In a real-time search engine, a time consuming spider has a negative impact on performance and the user's patience. As an enhancement, the real-time search engine was built on the capability to create any number of spider threads specified by the user.

The results of concurrency had a tremendous impact on performance as the execution time dropped 50% to 80% as the engine went from a 1-thread execution to a 5-thread execution at depth level 2 and higher. The impact was less noticeable as the thread count increased above 10. As the number of threads grew, the execution time was constant.

Slight decrease in performance was noticeable around 80 threads with some queries. Overall, the threads had a definite impact on execution time.

10.1.3 Web Searching with Caching

A traditional search index can be viewed as a large repository of categorized cached web data. Since caching is successful in the web browsing market, it is conceivable that it would make an impact on the real-time search engine performance.

The performance of the caching search at lower depth levels was disappointing as the execution time was nearly the same as a full real-time web search. Only after depth level 3 did the caching search provide better execution time. The performance short fall came when the caching mode had to make a request to the web server to find when the page was last modified. This I/O request would certainly impact execution time as the spider would have to wait for a response. In a small search domain with smaller web pages, the performance of a caching search would be similar to a full search. If the HTML pages are larger, then a caching search would certainly provide a better response time.

10.2 Practicality of the Real-Time Search Engine

The web searching performance depends on the conditions of the query and the placement of web pages in the search domain. For the best response time, it is

recommended to utilize multiple spider threads whenever possible. The number of spiders should vary based on network latency and the size of the search domain.

A pruning traversal will out perform a full breadth-first or depth-first search, but it comes with the price of a reduced result set. In most of the test runs, the pruning had only dropped a few URL addresses from results set. If the execution time could be cut in half as a result of pruning, this would seem to be a favorable trade off. But the analysis is very subjective and the conclusion is based on the needs of the user.

Under ideal conditions, caching is a valuable time saving method. When there are a large number of pages to be searched, relying on a cache resource to provide the bulk of the page content can greatly improve performance. In a practical sense, the overwhelming disadvantage with caching is that the content has to be inserted into cache first before the performance can be realized. This means the caching resource would have to be built up over a period of time. Since the cache is only updated when a real-time search is executed, the same queries would have to be executed to keep the content updated; otherwise, the cache becomes stale and a more time consuming request to the web server will have to be made. In such cases, a web search would provide better results than a cached search.

10.3 Future Enhancements

The design of the real-time search engine was based around the performance implementations described in this research. During the development, many improvements were encountered, but due to time and resource constraints were not applied in the application. The syntax of the query statement could be improved to allow searches on page attributes such as the page creation date, the last modified date, and size of the page. A timeout clause could also be added to the query statement to stop the search engine after it has executed for a specified amount of time. In addition, the WHERE clause could be extended to search for compound strings and phrases.

In terms of adding additional functionality, the spider can be adapted to crawl to non-HTML documents such as Word documents, PowerPoint files, text files, and PDF documents. In the current implementation, the result set did not rank the quality of the web page. Adding such a feature might hamper performance, but would provide a meaningful list of pages to the user.

Lastly, the selective path approach for the pruning search was based on if the page matched the web query. A more selective path heuristic approach would provide a better search path; and possibly a higher return in quality web pages.

10.4 Impressions

The real-time search engine cannot compare to the search engines of Lycos, AltaVista, and Google. These search engines have massive servers that scour the web 24 hours a day searching for content to store in efficient catalogs of databases. At its best, the real-time search engine can perform a three level search in about one minute. The deeper the search, the more pages the engine has to download and parse. The real-time search engine may not be able to compete with today's search engines, but it can allow a single user to quickly and easily perform a real-time dynamic search of a page and its neighboring hyperlinks. In the end, it is better than the browser's single page "FIND" command.

References

[Aberer03]
Aberer, K. and J. Wu, "A Framework for Decentralized Ranking in Web Information Retrieval," Proc Asia Pacific Web Conference 2642 (2003), pp. 213-226.

[Anderson89]
Anderson, T. E., D. D. Lazowska, and H. M. Levy, "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors," Proceedings of the 1989 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (1989), pp. 49-60.

[Angelaccio02]
Angelaccio, M. and B. Buttarazzi, "Local Searching the Internet," IEEE Internet Computing 6, 1 (January/February 2002), pp. 25-33.

[Bun01]
Bun, K. K. and M. Ishizuka, "Emerging Topic Tracking System," Third International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (June 2001), pp. 2-11.

[Charkabrti99]
Chakrabarti, S., et al, "Mining the Web's Link Structure," Computer 32, 8 (1999), pp. 60-67.

[Chau01]
Chau, M., D. Zeng, and H. Chen, "Personalized Spiders for Web Search and Analysis," First ACM/IEEE-CS Joint Conference on Digital Libraries (June 2001), pp. 79-87.

[Davidson01]
Davidson, B., "A Web Caching Primer," IEEE Internet Computing 5, 4 (July/August 2001), pp. 38-45.

[December94]
December, J., "New spiders roam the Web," Computer-Mediated Communication Magazine 1, 5 (September 1994), pp. 3-4.

[Filman98]
Filman, R. and S. Pant, "Searching the Internet," IEEE Internet Computing 2, 4 (July 1998), pp. 21-23.

[Franz04]
Orase was developed by Markus Franz as a real-time web search engine.
http://www.orase.com

[Greenberg99]
Greenberg, I. and L. Garber, "Searching for New Search Technologies," Computer 32, 8 (August 1999), pp. 4-11.

[Gu99]
Gu, Y., B. S. Lee, and W. Chai, "Evaluation of Java Thread Performance on Two Different Multithreaded Kernels," ACM SIGOPS Operating Systems Review 33, 1 (1999), pp. 34-46.

[Hafri04]
Hafri, Y. and C Djeraba, "High Performance Crawling System," Proceedings of the 6th ACM SIGMM International Workshop on Multimedia Information Retrieval (October 2004), pp. 299-306.

[Ikeji99]
Ikeji, A. and F. Fotouhi, "An Adaptive Real-Time Web Search Engine", Proceedings of the 2nd International Workshop on Web Information and Data Management (1999), pp. 12-16.

[Ishikawa99]
Ishikawa, H., K. Kubota, and Y. Kanemasa, "The Design of a Language for XML Data," 10th International Workshop on Database & Expert Systems Applications (September 1999), pp. 919-922.

[Koehler99]
Koehler, W., "Digital Libraries and World Wide Web Sites and Page Persistence," Information Research 4, 4 (July 1999), http://informationr.net/ir/4-4/paper60.html.

[Knoblock97]
Knoblock, C., "Searching the World Wide Web," IEEE Expert 12, 1 (January/February 1997), pp. 8-14.

[Lakshmanan96]
Lakshmanan, L., I. Subramanian, and F. Sadri, "A Declarative Language for Querying and Restructuring the Web," 6th International Workshop on Research Issues in Data Engineering (RIDE '96) Interoperability of Nontraditional Database Systems (February 1996), pp. 12-23.

[Lee01]
Lee, C., et al, "Using Threading and Factory Model to Improve the Performance of Distributed Object Computing System," 21st International Conference on Distributed Computing Systems Workshops (April 2001), pp. 371-378.

[Mendelzon96]
Mendelzon, A., G. Mihalia, and T. Milo, "Querying the World Web Wide," 4th International Conference on Parallel and Distributed Information Systems (December 1996), pp. 80-91.

[Moody03]
Moody, K. and M. Palomino, "SharpSpider: Spidering the Web through Web Services," First Latin American Web Congress (November 2003), pp. 219-221.

[Pokorny04]
Pokorny, J., "Web Searching and Information Retrieval," Computing in Science and Engineering 6 (July/August 2004), pp. 43-48.

[Singh03]
Singh, A., et al., "Apoedica: A Decentralized Peer-to-Peer Architecture for Crawling the World Wide Web," Proceedings SIGIR 2003 Workshop on Distributed Information Retrieval (2003), pp. 126-142.

[Szymanski01]
Szymanksi, B. K. and M. Chung, "A Method for Indexing Web Pages Using Web Bots," Proceeding Int. Conference on Info-Tech & Info-Net ICCII'2001 (November 2001), pp. 1-6.

[Webopedia]
XML - http://webopedia.com/TERM/X/XML.html

[Wiess99]
Weiss, M. A., Data Structures & Problem Solving Using Java. (March 1999), Addison-Wesley Longman Inc, Reading, Massachusetts.

Additional Resources

[AltaVista]
AltaVista is considered by many to be the premier search engine. The engine consistently returns the same results for the same query. It has also been noted as returning irrelevant data due to the fact that information is not screened. Altavista can perform both web and Usenet searches and features a wide variety of search tools. http://www.altavista.com

[Askjeeves]
Ask Jeeves is well know for is natural language approach to querying the web. The catalog was maintained by a human staff. Concurrently, it is using a crawler-based service to search the web.
http://www.ask.com

[Dogpile]
Dogpile is a metasearch engine that crawls other search engines to combine the power of those search engines.
http://www.dogpile.com

[Google]
Google is considered by many to be the first search engine to visit when looking for information on the web. It has recently searched up to 8 billion websites. Google started as a Stanford University project by Larry Page and Sergey Brinn.
http://www.google.com

[Lycos]
Lycos is one of the first search engines on the Web. It is not as sophisticated as the newer search engines, but it allows both keyword and subject searches. It excels in speed, ease of use, and large indices. Lycos does not support Boolean searches.
http://www.lycos.com

[Metacrawler]
Metacrawler is a metasearch engine that crawls other search engines to combine the power of those search engines.
http://www.metacrawler.com

[Vaughan-Nichols03]
Vaughan-Nichols, S., "Seeking Better Web Search Technologies," Computer 36, 8 (August 2003), pp.19-21.

[Webcrawler]
Webcrawler is a much more powerful engine for advanced searches when compared to AltaVista. It includes search tools for proximity operators and preclassified search catalogs. Webcrawler also supports natural language queries.
http://www.webcrawler.com

[Yahoo]
Yahoo was started in 1994 and is the web's oldest search engine. It was directory based until 2002 when it started to use spiders to search the web. It still maintains its original directory based service.
http://www.yahoo.com

Appendix A

DTD for Book Example

```
<!ELEMENT book (author+, title, publisher)>
<!ATTLIST book year CDATA>
<!ELEMENT article (author, title, year?)>
<!ATTLIST article type CDATA>
<!ELEMENT publisher (name, address)>
<!ELEMENT author (firstname?, lastname, office+)>
<!ELEMENT office (CDATA | longoffice)>
<!ELEMENT longoffice (building, room)>
<!ELEMENT watch (name, brand)
```

Appendix B

Examples of Traversal and Multithreaded Implementations

```
Welcome to the Real-Time Search Engine
A Master Thesis Project by Burr Watters
DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME WEB SEARCH ENGINE

Search Engine Environmental Elements
-----------------------------------

Enter Query:
SELECT breadth-web
FROM http://www.unf.edu
WHERE body = "business" and body = "news"
DEPTH 3
THREAD 20;

Search Mode is Web Mode with Breadth-First Traversal.
Pruning Traversal is OFF.
Search is running with 20 threads.
Maximum Depth Level is set to 3.


Real-Time Search Engine Workspace
---------------------------------
1 hyperlinks in storage queue remains to be parsed.
Now parsing URL http://www.unf.edu in Thread 0.
http://www.unf.edu was processed in 0.18 seconds in Thread 0.
36 hyperlinks in storage queue remains to be parsed.
Now parsing URL http://www.unf.edu/view/prospective in Thread 14.
35 hyperlinks in storage queue remains to be parsed.
Now parsing URL http://www.unf.edu/view/current/students.html in Thread
12.
{Edited for brevity}
http://www.unf.edu/compserv/info/dept/its/security was processed in
17.544 seconds in Thread 7.
http://www.unf.edu/compserv/info/dept/its/webpub was processed in
15.095 seconds in Thread 17.
Thread 8 has completed in 56.55800000000001 seconds.
Thread 1 has completed in 57.549 seconds.
Thread 15 has completed in 60.804 seconds.
Thread 16 has completed in 59.623999999999995 seconds.
Thread 3 has completed in 56.762 seconds.
Thread 13 has completed in 68.695 seconds.
Thread 4 has completed in 60.066 seconds.
```

```
Thread 12 has completed in 59.706 seconds.
Thread 7 has completed in 67.35 seconds.
Thread 10 has completed in 67.685 seconds.
Thread 2 has completed in 55.389 seconds.
Thread 9 has completed in 64.46300000000001 seconds.
Thread 14 has completed in 58.403 seconds.
Thread 6 has completed in 64.785 seconds.
Thread 19 has completed in 67.681 seconds.
Thread 18 has completed in 58.362 seconds.
Thread 0 has completed in 54.065999999999995 seconds.
Thread 11 has completed in 56.305000000000014 seconds.
Thread 17 has completed in 47.756 seconds.
Thread 5 has completed in 67.78200000000001 seconds.


URL Result Set from Real-Time Search Engine: 22 URL(s) were discovered.
------------------------------------------------------------------------
http://www.unf.edu/sitemap
http://dir.yahoo.com/Regional/U_S__States/Florida/Cities/Jacksonville
http://www.unf.edu/development/news/expertsguide/index.php
http://www.unf.edu/development/news/pressreleases/index.php
http://www.unf.edu/campus/maps_AtoZ_B.html
http://www.unf.edu/alumni/address_update.html
http://www.unf.edu/campus/maps_numbers.html
http://www.jacksonville.com
http://www.amsouth.com
http://www.firstcoastcommunity.com
http://www.sptimes.com/2004/07/06/Tampabay/Lazy_Fridays_a_luxury.shtml
http://www.jacksonville.com/tu-online/stories/092704/pat_16746411.shtml
http://jacksonville.com/tu-online/stories/053104/bus_15721014.shtml
http://www.jacksonville.com/tu-online/stories/111904/met_17230702.shtml
http://jacksonville.com/tu-online/stories/111504/met_17187547.shtml
http://www.jacksonville.com/tu-online/stories/080704/met_16302553.shtml
http://jacksonville.com/tu-online/stories/062104/pat_15910874.shtml
http://jacksonville.com/tu-online/stories/102704/ner_17002416.shtml
http://jacksonville.bizjournals.com/jacksonville/stories/2004/10/04/edi
torial2.html
http://jacksonville.bizjournals.com/jacksonville/stories/2004/06/21/dai
ly25.html
http://www.unf.edu/library/info/whatsnew.html
http://www.fldcu.org

Real-Time Search Statistics
---------------------------
Pages Processed Per Thread:
Thread 8: 41
Thread 3: 28
Thread 2: 59
Thread 1: 36
Thread 6: 40
Thread 10: 34
Thread 11: 56
Thread 7: 26
Thread 12: 43
Thread 9: 24
Thread 14: 29
Thread 15: 30
```

```
Thread 19: 26
Thread 18: 36
Thread 0: 59
Thread 16: 35
Thread 5: 24
Thread 17: 21
Thread 13: 23
Thread 4: 24

Total Search Time: 69.804
Summation of Crawling Time for ALL threads: 1209.7910000000002
Total Pages Searched: 694

Caching and Database Statistics
------------------------------
DB Download: 0
DB No Download: 0
Web Download: 0
```

```
Welcome to the Real-Time Search Engine
A Master Thesis Project by Burr Watters
DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME WEB SEARCH ENGINE

Search Engine Environmental Elements
-----------------------------------

Enter Query:
SELECT depth-web
FROM http://www.unf.edu
WHERE body = "business" and body = "news"
DEPTH 3
THREAD 20;

Search Mode is Web Mode with Depth-First Traversal.
Pruning Traversal is OFF.
Search is running with 20 threads.
Maximum Depth Level is set to 3.


Real-Time Search Engine Workspace
---------------------------------
1 hyperlinks in storage queue remains to be parsed.
Now parsing URL http://www.unf.edu in Thread 0.
http://www.unf.edu was processed in 0.17 seconds in Thread 0.
36 hyperlinks in storage queue remains to be parsed.
Now parsing URL http://www.unf.edu/compserv/info/gnulinux.html in
Thread 16.
{Edited for brevity}
http://www.amsouth.com was processed in 6.955 seconds in Thread 1.
http://www.unf.edu/dept/cdc was processed in 3.89 seconds in Thread 9.
http://www.gojacksonville.com is a dead link in Thread 2.
Thread 5 has completed in 53.088 seconds.
Thread 7 has completed in 52.23599999999999 seconds.
Thread 8 has completed in 54.58200000000001 seconds.
Thread 12 has completed in 53.10199999999999 seconds.
Thread 19 has completed in 53.31900000000001 seconds.
Thread 4 has completed in 51.69499999999999 seconds.
Thread 6 has completed in 52.028 seconds.
Thread 0 has completed in 53.30799999999999 seconds.
Thread 17 has completed in 53.021 seconds.
Thread 3 has completed in 49.71399999999999 seconds.
Thread 15 has completed in 52.907999999999994 seconds.
Thread 16 has completed in 52.384 seconds.
Thread 13 has completed in 52.464 seconds.
Thread 1 has completed in 54.427 seconds.
Thread 14 has completed in 52.18099999999999 seconds.
Thread 18 has completed in 52.817000000000014 seconds.
Thread 11 has completed in 51.434999999999995 seconds.
Thread 9 has completed in 56.70900000000001 seconds.
Thread 2 has completed in 43.867999999999995 seconds.
Thread 10 has completed in 53.135 seconds.


URL Result Set from Real-Time Search Engine: 22 URL(s) were discovered.
```

```
--------------------------------------------------------------------------
http://www.unf.edu/sitemap
http://www.unf.edu/library/info/whatsnew.html
http://www.unf.edu/campus/maps_numbers.html
http://www.unf.edu/campus/maps_AtoZ_B.html
http://www.sptimes.com/2004/07/06/Tampabay/Lazy_Fridays_a_luxury.shtml
http://jacksonville.com/tu-online/stories/053104/bus_15721014.shtml
http://jacksonville.com/tu-online/stories/062104/pat_15910874.shtml
http://www.jacksonville.com/tu-online/stories/080704/met_16302553.shtml
http://www.jacksonville.com/tu-online/stories/111904/met_17230702.shtml
http://www.jacksonville.com/tu-online/stories/092704/pat_16746411.shtml
http://jacksonville.com/tu-online/stories/111504/met_17187547.shtml
http://jacksonville.com/tu-online/stories/102704/ner_17002416.shtml
http://www.unf.edu/development/news/expertsguide/index.php
http://jacksonville.bizjournals.com/jacksonville/stories/2004/10/04/edi
torial2.html
http://jacksonville.bizjournals.com/jacksonville/stories/2004/06/21/dai
ly25.html
http://www.unf.edu/development/news/pressreleases/index.php
http://www.fldcu.org
http://dir.yahoo.com/Regional/U_S__States/Florida/Cities/Jacksonville
http://www.unf.edu/alumni/address_update.html
http://www.jacksonville.com
http://www.firstcoastcommunity.com
http://www.amsouth.com

Real-Time Search Statistics
--------------------------
Pages Processed Per Thread:
Thread 12: 29
Thread 5: 23
Thread 0: 48
Thread 7: 26
Thread 11: 53
Thread 8: 15
Thread 14: 48
Thread 13: 39
Thread 17: 30
Thread 3: 92
Thread 19: 32
Thread 15: 24
Thread 16: 38
Thread 4: 35
Thread 1: 26
Thread 18: 30
Thread 6: 51
Thread 9: 20
Thread 2: 15
Thread 10: 20

Total Search Time: 66.188
Summation of Crawling Time for ALL threads: 1048.421
Total Pages Searched: 694

Caching and Database Statistics
------------------------------
DB Download: 0
```

DB No Download: 0
Web Download: 0

Appendix C

Example of Pruning Traversal Implementation

```
Welcome to the Real-Time Search Engine
A Master Thesis Project by Burr Watters
DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME WEB SEARCH ENGINE

Search Engine Environmental Elements
------------------------------------

Enter Query:
SELECT breadth-web WITH PRUNING
FROM http://www.unf.edu
WHERE body = "business" and body = "news"
DEPTH 3
THREAD 20;

Search Mode is Web Mode with Breadth-First Traversal.
Pruning Traversal is ON.
Search is running with 20 threads.
Maximum Depth Level is set to 3.


Real-Time Search Engine Workspace
---------------------------------
1 hyperlinks in storage queue remains to be parsed.
Now parsing URL http://www.unf.edu in Thread 0.
http://www.unf.edu was processed in 0.173 seconds in Thread 0.
36 hyperlinks in storage queue remains to be parsed.
Now parsing URL http://www.unf.edu/view/prospective in Thread 11.
{Edited for brevity}
http://www.unf.edu/compserv/info/dept/its/tisoli was processed in 18.52
seconds in Thread 15.
http://www.unf.edu/compserv/info/dept/its/webpub was processed in
18.611 seconds in Thread 8.
Thread 13 has completed in 28.118000000000002 seconds.
Thread 4 has completed in 26.461000000000002 seconds.
Thread 10 has completed in 31.301000000000002 seconds.
Thread 14 has completed in 28.519 seconds.
Thread 17 has completed in 28.857999999999997 seconds.
Thread 8 has completed in 32.486000000000004 seconds.
Thread 2 has completed in 31.823999999999998 seconds.
Thread 1 has completed in 26.122 seconds.
Thread 5 has completed in 32.346000000000004 seconds.
Thread 3 has completed in 26.658 seconds.
Thread 0 has completed in 29.220999999999997 seconds.
Thread 6 has completed in 5.468999999999999 seconds.
Thread 12 has completed in 30.096 seconds.
```

```
Thread 11 has completed in 31.268 seconds.
Thread 15 has completed in 33.528999999999996 seconds.
Thread 19 has completed in 32.394 seconds.
Thread 18 has completed in 32.724 seconds.
Thread 7 has completed in 26.987 seconds.
Thread 9 has completed in 32.218 seconds.
Thread 16 has completed in 27.954000000000004 seconds.


URL Result Set from Real-Time Search Engine: 15 URL(s) were discovered.
------------------------------------------------------------------------
http://www.unf.edu/sitemap
http://www.unf.edu/development/news/expertsguide/index.php
http://www.unf.edu/development/news/pressreleases/index.php
http://dir.yahoo.com/Regional/U_S__States/Florida/Cities/Jacksonville
http://www.jacksonville.com
http://www.jacksonville.com/tu-online/stories/111904/met_17230702.shtml
http://jacksonville.com/tu-online/stories/111504/met_17187547.shtml
http://jacksonville.com/tu-online/stories/102704/ner_17002416.shtml
http://www.jacksonville.com/tu-online/stories/092704/pat_16746411.shtml
http://www.sptimes.com/2004/07/06/Tampabay/Lazy_Fridays_a_luxury.shtml
http://www.jacksonville.com/tu-online/stories/080704/met_16302553.shtml
http://jacksonville.bizjournals.com/jacksonville/stories/2004/10/04/edi
torial2.html
http://jacksonville.com/tu-online/stories/062104/pat_15910874.shtml
http://jacksonville.com/tu-online/stories/053104/bus_15721014.shtml
http://jacksonville.bizjournals.com/jacksonville/stories/2004/06/21/dai
ly25.html

Real-Time Search Statistics
---------------------------
Pages Processed Per Thread:
Thread 17: 7
Thread 13: 47
Thread 3: 13
Thread 10: 17
Thread 16: 21
Thread 8: 16
Thread 0: 5
Thread 6: 72
Thread 1: 23
Thread 4: 21
Thread 2: 5
Thread 5: 13
Thread 11: 9
Thread 12: 10
Thread 15: 10
Thread 19: 15
Thread 18: 12
Thread 14: 11
Thread 7: 48
Thread 9: 13

Total Search Time: 35.615
Summation of Crawling Time for ALL threads: 574.5529999999999
Total Pages Searched: 388
```

```
Caching and Database Statistics
-------------------------------
DB Download: 0
DB No Download: 0
Web Download: 0
```

Appendix D

Example of Web Search with Caching Implementation

```
Welcome to the Real-Time Search Engine
A Master Thesis Project by Burr Watters
DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME WEB SEARCH ENGINE

Search Engine Environmental Elements
------------------------------------

Enter Query:
SELECT breadth-cache
FROM http://www.unf.edu
WHERE body = "business" and body = "news"
DEPTH 3
THREAD 20;

Search Mode is Caching Mode with Breadth-First Traversal.
Pruning Traversal is OFF.
Search is running with 20 threads.
Maximum Depth Level is set to 3.


Real-Time Search Engine Workspace
---------------------------------
1 hyperlinks in storage queue remains to be parsed.
Now parsing URL http://www.unf.edu in Thread 0.
CACHE: URL http://www.unf.edu has not been modified since last
download.
http://www.unf.edu was processed in 0.059 seconds in Thread 0.
{Edited for brevity}
http://www.unf.edu/compserv/helpdesk/real was processed in 5.206
seconds in Thread 9.
http://www.gojacksonville.com is a dead link in Thread 13.
Thread 16 has completed in 21.13 seconds.
Thread 13 has completed in 14.849999999999998 seconds.
Thread 18 has completed in 19.578 seconds.
Thread 5 has completed in 16.344 seconds.
Thread 0 has completed in 16.713000000000005 seconds.
Thread 4 has completed in 16.163999999999998 seconds.
Thread 7 has completed in 16.344 seconds.
Thread 11 has completed in 15.375999999999998 seconds.
Thread 19 has completed in 20.918 seconds.
Thread 17 has completed in 21.082 seconds.
Thread 6 has completed in 21.490000000000002 seconds.
Thread 2 has completed in 21.488999999999997 seconds.
Thread 8 has completed in 16.470999999999993 seconds.
Thread 12 has completed in 21.266 seconds.
```

```
Thread 1 has completed in 21.110999999999997 seconds.
Thread 3 has completed in 21.279000000000003 seconds.
Thread 9 has completed in 17.977000000000004 seconds.
Thread 10 has completed in 14.146999999999998 seconds.
Thread 15 has completed in 16.012 seconds.
Thread 14 has completed in 21.282 seconds.


URL Result Set from Real-Time Search Engine: 23 URL(s) were discovered.
---------------------------------------------------------------------
http://www.unf.edu/sitemap
http://www.fema.gov
http://www.unf.edu/campus/maps_numbers.html
http://www.unf.edu/campus/maps_AtoZ_B.html
http://www.unf.edu/alumni/address_update.html
http://www.fldcu.org
http://www.unf.edu/development/news/expertsguide/index.php
http://www.unf.edu/development/news/pressreleases/index.php
http://www.sptimes.com/2004/07/06/Tampabay/Lazy_Fridays_a_luxury.shtml
http://jacksonville.com/tu-online/stories/053104/bus_15721014.shtml
http://jacksonville.com/tu-online/stories/102704/ner_17002416.shtml
http://www.jacksonville.com/tu-online/stories/092704/pat_16746411.shtml
http://jacksonville.com/tu-online/stories/062104/pat_15910874.shtml
http://www.jacksonville.com/tu-online/stories/111904/met_17230702.shtml
http://jacksonville.com/tu-online/stories/111504/met_17187547.shtml
http://www.jacksonville.com/tu-online/stories/080704/met_16302553.shtml
http://www.jacksonville.com
http://www.amsouth.com
http://www.firstcoastcommunity.com
http://jacksonville.bizjournals.com/jacksonville/stories/2004/10/04/edi
torial2.html
http://jacksonville.bizjournals.com/jacksonville/stories/2004/06/21/dai
ly25.html
http://dir.yahoo.com/Regional/U_S__States/Florida/Cities/Jacksonville
http://www.unf.edu/library/info/whatsnew.html

Real-Time Search Statistics
---------------------------
Pages Processed Per Thread:
Thread 13: 30
Thread 6: 18
Thread 10: 40
Thread 0: 28
Thread 16: 13
Thread 4: 76
Thread 18: 23
Thread 14: 9
Thread 11: 51
Thread 17: 15
Thread 2: 14
Thread 8: 65
Thread 1: 12
Thread 12: 18
Thread 3: 22
Thread 5: 68
Thread 9: 46
Thread 7: 61
```

```
Thread 15: 70
Thread 19: 20

Total Search Time: 36.126
Summation of Crawling Time for ALL threads: 371.023
Total Pages Searched: 699

Caching and Database Statistics
------------------------------
DB Download: 1 4.374
DB No Download: 402 0.9120621890547264
Web Download: 0
```

Appendix E

Java Source Code for Real-Time Search Engine

Search.java

```
/*********************************************************************
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

SEARCH.JAVA IS THE CONTROLLING OBJECT THAT EXECUTES THE REAL-TIME
ENGINE.  IT CREATES THE SPIDER OBJECTS AND STARTS THE CORRECT NUMBER
OF SPIDER THREADS SPECIFIED BY THE USER.  AFTER THE THREADS HAVE
COMPLETED, IT GATHERS THE STATS INFORMATION, PRINTS THE STATS, AND
PRINTS THE URLS THAT SATISIFY THE QUERY.
*********************************************************************/

import java.io.*;
import java.util.*;
import tree.*;

public class Search {
    public static void main(String[] args) {
        // CREATES A SPIDER THREAD GROUP
        ThreadGroup spiderThreadGroup =
            new ThreadGroup("Spider Group");
        // CREATES A QUERY PROCESSOR THREAD GROUP
        ThreadGroup matchingThreadGroup =
            new ThreadGroup("Matching Group");

        // HOUSES THE CONDITIONS LIST
        LinkedList condition_list = new LinkedList();

        // CREATES STATS OBJECT
        Stats stats = new Stats();

        // CREATES THE TREE TO HOUSE THE URLS THAT SATISFY THE QUERY
        AATree matched_url_tree = new AATree();

        // CREATES OBJECT FOR THE TRAVERSAL METHODS
        TraversalMethods traversal_method;
```

```
// CREATES THE TREE TO HOUSE THE VISITED URLS
AATree visited_url_tree = new AATree();

// SPECIFIES THE SEARCH MODE
int option;
// SPECIFIES THE NUMBER OF THREADS
int threads;
// SPECIFIES WHETHER PRUNING WILL OCCUR
boolean pt;
// SPECIFIES THE DEPTH
int depth;
// SPECIFIES THE MODE OF TRAVERSING
String traversal_mode;

System.out.println("\n\nWelcome to the Real-Time
    Search Engine");
System.out.println("A Master Thesis Project by Burr Watters");
System.out.println("DEVELOPMENT AND PERFORMANCE EVALUATION
    OF A REAL-TIME WEB SEARCH ENGINE");
System.out.println("\nSearch Engine Environmental Elements");
System.out.println("-----------------------------------");

// TEST QUERY STRINGS
String SQL = "SELECT cache WITH PRUNING\nFROM
    http://www.unf.edu\nWHERE body = \"blackboard\" or body =
    \"news\"\nDEPTH 2\nTHREAD 10;";
SQL = "SELECT depth-web\nFROM http://www.unf.edu\nWHERE body =
    \"business\" and body = \"advisor\"\nDEPTH 2\nTHREAD 5;";

try {
   // ALLOWS THE QUERY TO BE READ IN FROM THE COMMAND INTERFACE
   BufferedReader br = new BufferedReader(new
          InputStreamReader(System.in));

   System.out.println("\nEnter Query:");
   SQL = br.readLine();

   System.out.println("");

   // PARSES THE QUERY STATEMENT
   ParsedStatementNode psn = new ParsedStatementNode();
   psn = StatementParser.parse(SQL);

   // SETS THE SEARCH MODE OF THE REAL-TIME ENGINE
   if ( (psn.search_mode).compareTo("breadth-web") == 0 ) {
      option = 2;
      traversal_mode = "breadth";
      System.out.println("Search Mode is Web Mode with Breadth-
          First Traversal.");
   }
   else if ( (psn.search_mode).compareTo("breadth-cache") == 0)
   {
      option = 1;
      traversal_mode = "breadth";
      System.out.println("Search Mode is Caching Mode with
          Breadth-First Traversal.");
   }
```

```
else if ( (psn.search_mode).compareTo("breadth-database") ==
        0 ) {
   option = 3;
   traversal_mode = "breadth";
   System.out.println("Search Mode is Database Mode with
        Breadth-First Traversal.");
}
else if ( (psn.search_mode).compareTo("depth-web") == 0 ) {
   option = 2;
   traversal_mode = "depth";
   System.out.println("Search Mode is Web Mode with Depth-
        First Traversal.");
}
else if ( (psn.search_mode).compareTo("depth-cache") == 0 )
{
   option = 1;
   traversal_mode = "depth";
   System.out.println("Search Mode is Caching Mode with
        Depth-First Traversal.");
}
else {
   option = 3;
   traversal_mode = "depth";
   System.out.println("Search Mode is Database Mode with
        Depth-First Traversal.");
}

// SETS IF PRUNING SHOULD OCCUR
pt = psn.pruning_mode;

if ( pt )
   System.out.println("Pruning Traversal is ON." );
else
   System.out.println("Pruning Traversal is OFF." );

// SETS THE THREAD COUNT
threads = psn.thread_count;
System.out.println("Search is running with " + threads + "
        threads.");

// SETS THE MAX DEPTH LEVEL
depth = psn.max_depth_level;
System.out.println("Maxium Depth Level is set to " + depth +
        ".\n\n");

// SETS THE TRAVERSAL MODE INTERFACE TO BREADTH-FIRST SEARCH
// OR DEPTH-FIRST SEARCH
if ( traversal_mode.compareTo("breadth") == 0 ) {
   traversal_method = new
        BreadthFirstTraversal(spiderThreadGroup, depth,
              threads);
}
else {
   traversal_method = new
        DepthFirstTraversal(spiderThreadGroup, depth,
              threads);
}
```

```
// CREATES A SYNCH QUEUE FOR MATCHED URLS AND THE PARSED HTML
// QUEUE
 SynchQueue matched_url_list = new
        SynchQueue( traversal_method );
 SynchQueue parsed_html_queue = new
        SynchQueue( traversal_method );

 // CREATES OBJECTS TO HOUSE THE TIME STATS
 TimeNode total_time = new TimeNode();
 TimeNode total_time2 = new TimeNode();

 // INSERT THE PRIMER URLS INTO THE TRAVERSAL METHODS
 traversal_method.insertURL( psn.primer_urls, 0, "M1");

 // SETS THE CONDITIONAL STATEMENTS
 condition_list = psn.condition_list;

 System.out.println("Real-Time Search Engine Workspace");
 System.out.println("--------------------------------");

 // STARTS THE EXECUTION TIMER FOR THE ENTIRE APPLICATION
 long start = System.currentTimeMillis();

 // CREATES A SPIDER THREAD ACCORDING TO THE COUNT IN THE
 // QUERY STATEMENT
 for (int i = 0; i < threads; i++) {
    Spider a1 = new Spider(traversal_method, depth, option,
        pt, Integer.toString(i), total_time,
        spiderThreadGroup, stats, condition_list,
        matched_url_list, total_time2, matched_url_tree,
        visited_url_tree );
    new Thread(spiderThreadGroup, a1).start();
 }

 // LOOPS UNTIL ALL THE SPIDER THREADS HAVE ENDED
 while ( spiderThreadGroup.activeCount() != 0 ) {}

 // ENDS THE EXECUTION TIMER FOR THE ENTIRE APPLICATIONS
 long end = System.currentTimeMillis();

 System.out.println("\n\nURL Result Set from Real-Time Search
        Engine: " + matched_url_list.size() + " URL(s) were
        discovered.");
 System.out.println("----------------------------------------
        -------------------------------");

 // LISTS THE MATCH URLS TO THE SCREEN
 while ( matched_url_list.size() > 0 ) {
    MatchedNode matched_node = new MatchedNode();

    matched_node = (MatchedNode) matched_url_list.getNode();

    System.out.println( matched_node.url );
 }

 System.out.println("\nReal-Time Search Statisics");
```

```
            System.out.println("-------------------------");
            System.out.println("Pages Processed Per Thread:");

            // LISTS OF THE NUMBER OF PAGES PROCESSED PER THREAD
            while ( !stats.isThreadStatsEmpty() ) {
               Node node = new Node();

               node = stats.getThreadStats();
               System.out.println("Thread " + node.content + ": " +
                   node.occurance );
            }

            // CALCULATES AND DISPLAYS THE TOTAL EXECUTION TIME
            double parse_execution_time = (end - start) / 1000.0;
            System.out.println("\nTotal Search Time: " +
                   parse_execution_time);
            System.out.println("Sumation of Crawling Time for ALL
                   threads: " + stats.getTime() );
            System.out.println("Total Pages Searched: " +
                   stats.getCount() );

            // DISPLAYS THE STATS OF THE CACHING AND DATABASE
            // METHODOLOGIES
            System.out.println("\nCaching and Database Statisics");
            System.out.println("------------------------------");
            if ( stats.getCachedWebViewCount() != 0 )
               System.out.println("DB Download: " +
                   stats.getCachedWebViewCount() + " " +
                   stats.getCachedWebViewTime() /
                   stats.getCachedWebViewCount() );
            else
               System.out.println("DB Download: " +
                   stats.getCachedWebViewCount() );

            if ( stats.getCachedViewCount() != 0 )
               System.out.println("DB No Download: " +
                   stats.getCachedViewCount() + " " +
                   stats.getCachedViewTime() /
                   stats.getCachedViewCount() );
            else
               System.out.println("DB No Download: " +
                   stats.getCachedViewCount() );

            if ( stats.getWebViewCount() != 0 )
               System.out.println("Web Download: " +
                   stats.getWebViewCount() + " " +
                   stats.getWebViewTime() / stats.getWebViewCount() );
            else
               System.out.println("Web Download: " +
                   stats.getWebViewCount() );
         }
         catch (Exception e) {
            System.out.println(e);
         }
      }
   }
```

Spider.java

```
/**************************************************************************
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

SPIDER.JAVA IS THE MAIN HTML PARSER.  IT TAKES A URL AND DECOMPILES
THE HTML INTO SPECIFIC TAGS NEEDED FOR THE QUERY.  IT'S SECOND FUNCTION
IS TO PARSE HTML KEYWORDS INTO A DATABASE AND RETREIVE WHEN THE WEBSITE
HAS BEEN STATIC.
**************************************************************************/

import java.net.*;
import java.io.*;
import java.util.*;
import java.sql.*;
import tree.*;

public class Spider implements Runnable
{
   // HOUSES DATA STRUCTURES FOR HTML KEYWORDS
   private LinkedList HLinkTag = new LinkedList();
   private AATree TitleTree = new AATree();
   private AATree MetaTree = new AATree();
   private AATree BoldTree = new AATree();
   private AATree EmTree = new AATree();
   private AATree ItalicTree = new AATree();
   private AATree BodyTree = new AATree();
   private AATree HeaderTree = new AATree();
   private Stack ParserStack = new Stack();

   // COUNTS THE NUMBER OF LINKS
   private int linkCnt = 0;

   // DECLARES THE GROUP OF SPIDER THREADS
   ThreadGroup spiderThreadGroup;

   // DETECTS IF THE SITE HAS BEEN FOUND
   private boolean website_found;

   private TraversalMethods traversal_method;
   AATree visited_url_tree;

   // DECLARES THE VARIABLE FOR THE MAX DEPTH
   int max_depth;

   // DECLARES CONNECTION VARIABLE
   Connection con;

   // SPECIFIES IF SEARCH MODE SHOULD BE WEB, CACHED, OR DATABASE
   int option;
   // SPECIFIES IF THE SEARCH SHOULD USE PRUNING
```

```
boolean pruning_traversal;
// SPECIFIES THREAD NAME
String threadname;

// DECLARES THE TIME VARIABLES
TimeNode total_time;
double parse_execution_time;
int parse_count = 0;
double total_parse_time;
int db_download = 0;
int db_no_download = 0;
int web_download = 0;
double db_download_time = 0;
double db_no_download_time = 0;
double web_download_time = 0;
Stats stats;
int links = 0;
int redirect = 0;

// GIVES THE CONDITION LIST FROM THE QUEUE
LinkedList condition_list;
// GIVES THE MATCHED URL LIST
SynchQueue matched_url_list;
// SPECIFIES TIME NODE
TimeNode time_node2;
// HOUSES THE MATCHING URL TREE
AATree matched_url_tree;


// CONSTRUCTOR FOR THE HTMLPARSER
public Spider( TraversalMethods traversal_method, int max_depth,
   int option, boolean pruning_traversal, String thread, TimeNode
   total_time, ThreadGroup spiderThreadGroup, Stats stats,
   LinkedList condition_list, SynchQueue matched_url_list,
   TimeNode time_node2, AATree matched_url_tree,
   AATree visited_url_tree ) throws IOException
{
   this.traversal_method = traversal_method;
   this.max_depth = max_depth;
   this.option = option;
   this.pruning_traversal = pruning_traversal;
   this.spiderThreadGroup = spiderThreadGroup;
   this.stats = stats;

   this.condition_list = condition_list;
   this.matched_url_list = matched_url_list;
   this.time_node2 = time_node2;

   threadname = thread;
   this.total_time = total_time;

   this.matched_url_tree = matched_url_tree;
   this.visited_url_tree = visited_url_tree;

   // LOADS MySQL DRIVER
   try
   {
```

```
            Class.forName("com.mysql.jdbc.Driver").newInstance();
        }
        catch(Exception e)
        {
            System.out.println("Failed to load mySQL driver.");
        }
    }

    // RUNNER FOR THE HTMLPARSER
    public void run()
    {
        // CALLS SPIDER FUNCTION
        spider();

        // TRACKS HOW LONG SEARCH HAS BEEN RUNNING AND HOW MANY SITES
        // HAVE BEEN PARSED
        total_time.execution = total_parse_time;
        total_time.page_count = parse_count;

        // INSERTS STATS INFORMATION OF THE THREAD AFTER IT HAS COMPLETED
        stats.insert( threadname, parse_count, total_parse_time,
                db_download, db_no_download, web_download,
                db_download_time, db_no_download_time, web_download_time,
                links, redirect );
        System.out.println("Thread " + threadname + " has completed in "
                + total_parse_time + " seconds." );
    }

    // SPIDER IS THE WORKHORSE OF THE PROGRAM.  THIS FUNCTION WILL
    // PARSER ALL URLS AND MATCH THEM TO THE QUERYBASE...IF THE URL
    // SATIFIES THE QUERY, IT IS PRINTED TO THE WEBPAGE, IF NOT
    // IT IS DISCARDED.
    private void spider( )
    {
        String search_method;
        String link;
        boolean dup_link = false;
        String childs_content;
        int line_break_cnt = 0;
        LinkedList valid_urls = new LinkedList();

        try {
            // OPENS DATABASE CONNECTION
            con = DriverManager.getConnection("jdbc:mysql://localhost:3306
                /watters?user=watb0003&password=unfunf");


            // REMOVES THE PARENT URL FROM THE LINKED LINK, THIS PARENT
            // WILL MOST LIKELY SPAWN SEVERAL CHILDERN.
            Node parent = (Node) traversal_method.removeURL(threadname);

            // REPEATS UNTIL NO HYPERLINKS ARE LEFT
            while ( !traversal_method.traversalComplete() ) {
                try {
                    website_found = true;

                    // RESETS THE TAG TREES
```

```
TitleTree = new AATree();
MetaTree = new AATree();
BoldTree = new AATree();
EmTree = new AATree();
ItalicTree = new AATree();
BodyTree = new AATree();
HeaderTree = new AATree();
HLinkTag = new LinkedList();

System.out.println("Now parsing URL " + parent.content +
   " in Thread " + threadname + ".");

long start = System.currentTimeMillis();

parse_count++;

// THIS IF STATEMENT WILL DETERMINE WHICH PARSER TO RUN,
// DATA OR WEB IF THE USER JUST WANTS TO PARSE THE WEB,
// THEN THE FIRST PART OF THE IF STATEMENT IS RUN.
if ( option == 2 ) {
      int response = 0;
      URL url = new URL(parent.content);
      // OPENS CONNECTION TO WEB SERVER
      URLConnection connection = url.openConnection();
      if (connection instanceof HttpURLConnection) {
         // RETRIEVES HEADER INFORMATION OF PAGE
         HttpURLConnection httpConnection =
               (HttpURLConnection)connection;
         httpConnection.setRequestMethod("HEAD");
         httpConnection.connect();
         response = httpConnection.getResponseCode();
      }
      links++;
      // ONLY PARSES PAGES IF THE WEB SERVER RESPONSES
      // AND THE PAGES CONTENT IS NOT OVER 100000
      if ( connection.getContentLength() < 100000
         && response < 400 ) {
            WebParser( parent.content, 0 );
      }
      else {
         throw new IOException("URL " + parent.content +
               " was not found or was too large.");
      }
      search_method = "web";
}
else {
   // THE USER HAS CHOOSEN TO USE THE DATABASE DURING
   // THE PARSING
   Statement select = con.createStatement();

   // POLLS THE DBMS TO SEE IF THE SITE HAS BEEN PARSED
   ResultSet query_result = select.executeQuery(
         "select time from url_time_table where url =
         \"" + parent.content + "\"");

   // IF THE ABOVE SQL STATEMENT RETURNED A ROW, THEN
   // THE PAGE HAS BEEN PARSED
```

```
if ( query_result.next() ) {
   URL url;
   long db_time = 0;
   long url_time = 0;

   if ( option < 3 ) {
      url = new URL( parent.content );

      // PREPARES THE PAGES CURRENT MODIFICATION TIME
      // AND THE DATABASE'S LAST MODIFICATION TIME
      URLConnection url_conn = url.openConnection();
      db_time = Long.parseLong(
            query_result.getString(1) );

      // CHECKS THE LAST TIME THE HTML PAGE WAS
      // MODIFIED
      url_time = url_conn.getLastModified();
   }

   // IF THE PAGE HAS NOT BEEN CHANGED SINCE IT WAS
   // LAST DOWNLOAD AND THE PAGE DOES NOT HAVE A 0
   // AS THE MODIFIED TIME...
   if ( option == 3 || url_time == db_time ) {
      System.out.println("CACHE: URL " +
            parent.content + " has not been modified
            since last download.");

      // THE DBMS IS USED TO RETRIEVE THE KEYWORDS
      DataParser( parent.content );
      search_method = "db_no_download";
      db_no_download++;
   }
   else {
      System.out.println("CACHE: URL " +
            parent.content + " has been modified or
            does not have a modified since date.");

      // ELSE DOWNLOADS THE SITE FROM THE WEB
      int response = 0;
      url = new URL(parent.content);
      // OPENS CONNECTION TO WEB SERVER
      URLConnection connection =
            url.openConnection();
      if (connection instanceof HttpURLConnection) {
         // RETRIEVES HEADER INFORMATION OF PAGE
         HttpURLConnection httpConnection =
            (HttpURLConnection)connection;
         httpConnection.setRequestMethod("HEAD");
         httpConnection.connect();
         response = httpConnection.getResponseCode();
      }
      links++;
      // ONLY PARSES PAGES IF THE WEB SERVER
      // RESPONSES AND THE PAGES CONTENT IS NOT OVER
      // 100000
      if ( connection.getContentLength() < 100000
            && response < 400 ) {
```

```
                WebParser( parent.content, 1 );
            }
            else {
                throw new IOException("CACHE: URL " +
                    parent.content + " was not found or was
                    too large.");
            }
            search_method = "db_download";
            db_download++;
        }
    }
    else {
        if ( option < 3 ) {
            System.out.println("CACHE: URL " +
                    parent.content + " is being downloaded
                    for the first time.");
            // THIS OPTION IS RAN IF QUERYBASE2 DETERMINES
            // THIS IS THE FIRST

            int response = 0;
            URL url = new URL(parent.content);
            // OPENS CONNECTION TO WEB SERVER
            URLConnection connection =
                    url.openConnection();
            if (connection instanceof HttpURLConnection) {
                // RETRIEVES HEADER INFORMATION OF PAGE
                HttpURLConnection httpConnection =
                    (HttpURLConnection)connection;
                httpConnection.setRequestMethod("HEAD");
                httpConnection.connect();
                response = httpConnection.getResponseCode();
            }
            links++;
            // ONLY PARSES PAGES IF THE WEB SERVER
            // RESPONSES AND THE PAGES CONTENT IS NOT OVER
            // 100000
            if ( connection.getContentLength() < 100000
                    && response < 400 ) {
                    WebParser( parent.content, 0 );
            }
            else {
                throw new IOException("URL " +
                    parent.content + " was not found or was
                    too large.");
            }
            search_method = "web_download";
            web_download++;
        }
        else {
            System.out.println("CACHE: URL " +
                    parent.content + " not found.");
            website_found = false;
            search_method = "database - link not found";
        }
    }
}
```

```
// THROWS ERROR IF DATABASE IMPLEMENATION COULD NOT FIND
// SITE, THIS DOES NOT STOP THE PARSER, IT JUST SKIPS IT
if ( !website_found )
   throw new IOException(parent.content + " was not
         found was not found in database.\n");

// ADDS PARSED HTML TREES TO A NODE AND SUBMITS NODE TO
// PARSED HTML QUEUE
ParsedHtmlNode parseNode = new ParsedHtmlNode();

// BUNDLES NEWLY PARSED KEYWORD TREES (BUCKETS) INTO ONE
// NODE
parseNode.BodyTree = BodyTree;
parseNode.TitleTree = TitleTree;
parseNode.EmTree = HeaderTree;
parseNode.BoldTree = BoldTree;
parseNode.MetaTree = MetaTree;
parseNode.HeaderTree = HeaderTree;
parseNode.ItalicTree = ItalicTree;
parseNode.url = parent.content;
parseNode.p_time = parse_execution_time;
parseNode.origin = search_method;

// ADDS THE NODE TO A QUEUE FOR THE MATCHING THREAD
Matching matching = new Matching(condition_list,
   matched_url_list, threadname, time_node2,
   traversal_method, pruning_traversal,
   matched_url_tree);
matching.setContent(parseNode, threadname);
int result = matching.run();

// CALCULATES THE TIME IT TOOK TO PROECESS THE PAGE
long finish = System.currentTimeMillis();
parse_execution_time = (finish - start) / 1000.0;

// CALCULATES DOWNLOADING TIME
if ( search_method.compareTo("db_no_download") == 0 )
   db_no_download_time = db_no_download_time +
         parse_execution_time;
else if (search_method.compareTo("db_download") == 0 )
   db_download_time = db_download_time +
         parse_execution_time;
else if (search_method.compareTo("web_download") == 0 )
   web_download_time = web_download_time +
         parse_execution_time;

// TIME STATS FOR PARSING EACH PAGE
total_parse_time = total_parse_time +
   parse_execution_time;

System.out.println(parent.content + " was processed in "
   + parse_execution_time + " seconds in Thread " +
   threadname + ".");

// THIS LOOP WILL PLACE THE LINKS OF THE PARENT IN A
// QUEUE FOR PARSING.
if ( parent.occurance < max_depth ) {
```

```
                // WILL ADD LINKS TO TRAVERSAL IF PRUNING IS NOT SET
                if ( pruning_traversal == false ) {
                   traversal_method.insertURL( HLinkTag,
                      parent.occurance, threadname );
                }
                else if ( result > 0 ) {
                   // ADDS LINKS TO TRAVERSAL IF PARENT PAGE
                   // SATISIFIES THE QUERY
                   traversal_method.insertURL( HLinkTag,
                      parent.occurance, threadname );
                }
             }
          }
          catch (IOException ioe) {
             // SAME STATUS CODE AS ABOVE BUT ASTERISK IS PRINTED FOR
             // DEAD LINKS.
             System.out.println(parent.content + " is a dead link in
                Thread " + threadname + "." );
          }
          catch ( NumberFormatException nfe ) {
             System.out.println("Error: Number Format Exception" +
                nfe);
          }
          catch ( SQLException sqle ) {
             System.out.println("Error: SQL Exception " + sqle);
          }

          // GETS NEXT NODE FROM THE HYPERLINK QUEUE FOR PARSING
          parent = (Node) traversal_method.removeURL(threadname);
       }
    }
    catch ( SQLException sqle ) {
       System.out.println("Error: SQL Exception" + sqle);
    }
}

// THIS FUNCTION PARSERS QUERIES THAT ARE RECIEVED FROM THE
// DATABASE.  IT THEN PUTS THE CONTENT FOR THOSE DATABASES INTO
// LINKED LIST SO THE MAIN CLASS AND RUN THE QUERIES.
private void DataParser(String url) throws SQLException
{
    int records_found = 0;
    String tag;
    Statement select = con.createStatement();
    String keyword;

    // QUERIES FOR LINKS
    ResultSet result = select.executeQuery("SELECT tag, keyword FROM
          html_parse_table WHERE url = \"" + url + "\"");

    while ( result.next() ) {
       tag = result.getString(1);
       tag = tag.trim();

       keyword = new String ( result.getString(2) );

       if ( tag.compareTo("a") == 0 )
```

```
            // ADDS LINKS TO LINKED LIST
            HLinkTag.add( keyword );
         else if ( tag.compareTo("title") == 0 )
            // ADDS TITLE KEYWORDS TO TITLE TREE
            TitleTree.insert( keyword.toLowerCase() );
         else if ( tag.compareTo("meta") == 0 )
            // ADDS META KEYWORDS TO META TREE
            MetaTree.insert( keyword.toLowerCase() );
         else if ( tag.compareTo("body") == 0 )
            // ADDS BODY KEYWORDS TO BODY TREE
            BodyTree.insert( keyword.toLowerCase() );
         else if ( tag.compareTo("header") == 0 ) {
            // ADDS HEADER KEYWORDS TO HEADER TREE
            HeaderTree.insert( keyword.toLowerCase() );
            BodyTree.insert( keyword.toLowerCase() );
         }
         else if ( tag.compareTo("b") == 0 ) {
            // ADDS BOLD KEYWORDS TO BOLD TREE
            BoldTree.insert( keyword.toLowerCase() );
            BodyTree.insert( keyword.toLowerCase() );
         }
         else if ( tag.compareTo("em") == 0 ) {
            // ADDS EMPHASIS KEYWORDS TO EM TREE
            EmTree.insert( keyword.toLowerCase() );
            BodyTree.insert( keyword.toLowerCase() );
         }
         else if ( tag.compareTo("i") == 0 ) {
            // ADDS ITALIC KEYWORDS TO ITALIC TREE
            ItalicTree.insert( keyword.toLowerCase() );
            BodyTree.insert( keyword.toLowerCase() );
         }
         records_found = 1;
      }

      // NOTES THAT THE DBMS DID NOT CONTAIN THIS PAGE
      if ( records_found == 0 ) {
         website_found = false;
      }

      select.close();
   }

   // THIS FUNCTION IS RESPONIBLE FOR CALLING THE PARSER, BUT IT ALSO
   // WEEDS OUT NON-HTML BASED PAGES.
   private void WebParser(String urlLoc, int site_has_been_modified )
            throws IOException, SQLException
   {
         Statement select = con.createStatement();

         // REMOVES OLD KEYWORD INFORMATION IN DATABASE IF SITE HAS
         // BEEN MODIFIED
         if ( site_has_been_modified == 1 ) {
            select.executeUpdate(
                  "DELETE FROM html_parse_table WHERE url = \"" +
                  urlLoc + "\"");
            select.executeUpdate("
                  DELETE FROM url_time_table WHERE url = \"" + urlLoc +
```

```
                "\"");
        }

        String url_path;
        URL url = new URL(urlLoc);

        // TRIMS PATH NAME
        url_path = url.getPath();
        url_path = url_path.trim();


        // THESE IF STATEMENTS TRY TO ELIMINATE NON-HTML BASED PAGES
        if ( url_path.indexOf('.') == -1 || url_path.endsWith(".html")
           || url_path.endsWith(".htm") || url_path.endsWith(".asp")
           || url_path.endsWith(".jsp") || url_path.endsWith(".shtml")
           || url_path.endsWith(".php") ) {
           // ADDS URL TO DBMS IF THE USER CHOOSE TO USE THE DBMS TO
           // SEARCH
           if ( option == 1 ) {
              URLConnection url_conn = url.openConnection();
              long url_last_modified = url_conn.getLastModified();

              select.executeUpdate("INSERT INTO url_time_table
                 VALUES (\"" + urlLoc  + "\",\"" + url_last_modified +
                 "\")" );
           }

           parse(url);
        }
        else
           System.out.println("Possible HTML attachment.");
}

// THIS FUNCTION DOES THE PARSING OF ACTUAL WEB PAGES INTO ITS
// SPECIFIC TAGS.  THOSE KEYWORDS ARE THEN PLACE IN LINKED LISTS.
private void parse(URL url) throws IOException, SQLException
{
   //ParserStack = new Stack();
   InputStream in = url.openStream();
   BufferedInputStream bufIn = new BufferedInputStream(in);

   String word = "";
   String content = "";
   boolean endtag;
   int searchindex;
   char previous_char;
   int prefix_index;
   String prefix;
   String postfix;
   boolean end_download = false;

   String[] valid_tag = { "<a", "<b", "<body", "<em", "<h1", "<h2",
         "<h3", "<h4", "<h5", "<h6", "<i", "<meta", "<title" };
   String[] invalid_tag = { "<!--", "<!doctype", "<abbr",
         "<acronym", "<address", "<applet", "<area", "<base",
         "<basefont", "<bdo", "<big", "<blockquote", "<br",
         "<button", "<caption", "<center", "<cite", "<code", "<col",
```

```
        "colgroup", "<dd", "<del", "<dfn", "<dir", "<div", "<dl",
        "<dt", "<fieldset", "<font", "<form", "<frame",
        "<frameset", "<head", "<hr", "<html", "<iframe", "<img",
        "<input", "<ins", "<isindex", "<kdb", "<label", "<legend",
        "<li", "<link", "<map", "<menu", "<noframes", "<noscript",
        "<object", "<ol", "<optgroup", "<option", "<p", "<param",
        "<pre", "<q", "<s", "<samp", "<script", "<select","<small",
        "<span", "<strike", "<strong", "<style", "<sub", "<sup",
        "<table", "<tbody", "<td", "<textarea", "<tfoot", "<th",
        "<thead", "<tr", "<tt", "<u", "<ul", "<var"};

Arrays.sort(valid_tag);
Arrays.sort(invalid_tag);

String url_string = url.toString();

// STARTS THE PARSING PROCESS
for (;;)
{
   // READS IN CHARACTERS FROM THE HTML PAGE
   int data = bufIn.read();

   if (data == -1 || end_download )
      break;
   else if ( (char) data == '<' )
   {
      // THIS IF STATEMENT CATCHES THE FIRST BRACKET TAGS AND ALL
      // CONTENT IN THE BRACKET
      String previous_word = word;
      word = "<";
      endtag = false;
      for (;;)
      {
         previous_char = (char) data;
         data = bufIn.read();

         if ( data == -1 || end_download )
            break;
         else if ( previous_char == '<' && (char) data == '/' )
         {
            // SIGNALS THAT THIS TAG IS AN END TAG
            endtag = true;
         }
         else if ( (char) data == '>' )
         {
            // BREAKS LOOP WHEN END TAG IS FOUND
            break;
         }
         // ADDS THE CHARACTERS TOGETHER TO CREATE WORD
         word = word + (char) data;
      }

      // FINDS THE TAG'S NAME AND PARSERS THE TAGS NAME FROM THE
      // ATTRIBUTES
      prefix_index = word.indexOf(" ");
      if ( prefix_index == -1 )
      {
```

```
      prefix = word;
      postfix = "";
   }
   else
   {
      prefix = word.substring(0, prefix_index);
      postfix = word.substring(prefix_index);
   }

   if ( endtag == true )
   {
      // IF THE TAG IS AN END TAG, IT TRIES TO SEE IF THE TAG
      // IS A VALID
      // TAG AND THEN POPS IT OFF THE STACK IF SO.
      if ( word.compareTo("</html") == 0 ||
         word.compareTo("</HTML") == 0 )
         end_download = true;

      if ( ParserStack.empty() )
      {
         if ( previous_word.length() != 0 )
         {
            addContent(previous_word, url.toString() );
            word = "";
         }
         word = "";
      }
      else
      {
         if ( content.length() != 0 )
         {
            addContent(content, url.toString() );
            content = "";
         }
         String peektag = (String) ParserStack.peek();
         String subword = word.substring(0,
               word.indexOf('/'));
         word = subword + word.substring(
               word.indexOf('/') + 1 );

         // IF WORD IS AN END TAG, IT POPS IT OFF THE STACK.
         if ( word.equalsIgnoreCase( peektag ) )
         {
            String endword = (String) ParserStack.pop();
            word = "";
         }
      }
   }
   else if ( ( searchindex = Arrays.binarySearch(valid_tag,
          prefix.toLowerCase() ) ) >= 0 )
   {
      // THIS IF STATEMENT SIGNALS THAT THE WORDS IS A VALID
      // TAG

      // PUSHES TAG ON TO STACK.
      ParserStack.push( prefix.toLowerCase() );
```

```
            word = "";

            // SENDS THE TAG OFF TO EXTRACT CONTENTS IF THERE ARE
            // CONTENTS IN THE TAG.
            if ( prefix_index == -1 )
               extractTagContents( " ", url );
            else
               extractTagContents( postfix, url);
            word = "";
         }
         else if ( Arrays.binarySearch(invalid_tag, prefix) >= 0 )
         {
            // THIS IF STATEMENT SIGNALS THAT THE WORD IS A TAG, BUT
            // NOT ONE THAT WE ARE
            // LOOKING FOR

            // MAKES SURE SCRIPT TAGS DON'T INTERFERE WITH THE
            // PROCESS
            if ( prefix.equalsIgnoreCase("<script") )
            {
               word = "";
               for (;;)
               {
                  data = bufIn.read();

                  word = word + (char) data;
                  if ( (char) data == '>' )
                  {
                     if ( word.indexOf("</script") == -1
                           || word.indexOf("</SCRIPT") == -1 )
                        break;
                  }
               }
            }
            word = "";
         }
         else
         {
            // REMOVES COMMENTS IN HTML SCRIPT
            word = "";
         }
      }
      else if ( !ParserStack.empty() )
      {
         // THIS IF STATEMENT CATCHES ALL WORDS WHEN THERE IS STILL
         // SOMETHING ON THE STACK

         // THIS SEE IF THE CHARACTER COMING END IS ANYTHING BUT A
         // LETTER OR NUMBER.
         if ( Character.isWhitespace( (char) data ) ||
               (char) data == ',' || (char) data == ';' ||
               (char) data == '\\'  || (char) data == '/' ||
               (char) data == '.' || (char) data == '\"' ||
               (char) data == '-' || (char) data == '©' ||
               (char) data == '!' || (char) data == '?' ||
               (char) data == '|' || (char) data == '(' ||
               (char) data == ')' || (char) data == '\'' ||
```

```
                (char) data == '[' || (char) data == ']' ||
                (char) data == '{' || (char) data == '}')
            {
               // CHECKS TO MAKE SURE THE CONTENT IS NOT A COMMON WORD,
               // OR IS NOT BLANK.
               if ( content.length() != 0 && isNotCommon( content ) )
               {
                  // SENDS THE WORD OFF TO BE ADDED TO A LIST.
                  addContent(content, url.toString() );
               }
               content = "";
            }
            else
               // THIS STATEMENT ABOVE ADDS THE CHARACTER TO THE
               // WORD...THUS BUILDING THE WORD
               content = content + (char) data;
         }
         else
         {
            // IF THE STACK IS EMPTY, CATCHES THE HTML CHARACTERS THAT
            // COME THROUGH

            // THIS SEE IF THE CHARACTER COMING END IS ANYTHING BUT A
            // LETTER OR NUMBER.
            if ( Character.isWhitespace( (char) data ) ||
                  (char) data == ',' || (char) data == ';' ||
                  (char) data == '\\'  || (char) data == '/' ||
                  (char) data == '.' || (char) data == '\"' ||
                  (char) data == '-' || (char) data == '©' ||
                  (char) data == '!' || (char) data == '?' ||
                  (char) data == '|' || (char) data == '(' ||
                  (char) data == ')' || (char) data == '\'' ||
                  (char) data == '[' || (char) data == ']' ||
                  (char) data == '{' || (char) data == '}')
            {
               // MAKES SURE THE WORDS ARE NOT BLANK
               if ( word.length() != 0 )
               {
                  //System.out.println("--> " + word );
                  addContent(word, url.toString() );
                  word = "";
               }
            }
            else
            {
               // THIS STATEMENT ADDS THE CHARACTER TO THE WORD...THUS
               // BUILDING THE WORD
               word = word + (char) data;
            }
         }
      }
      //Closes stream
      bufIn.close();
      in.close();
   }

   // THIS FUNCTION IS EXTRACTS THE CONTENTS THAT FALLS IN THE TAG
```

```
   // ITSELF...EXAMPLES ARE ANCHOR TAGS AND META TAGS
   private void extractTagContents(String content, URL url)
            throws SQLException
   {
      boolean pound_sign = false;
      String word = "";
      int i;

      String tag = (String) ParserStack.peek();

      // IF THE TAG IS A ANCHOR, IT TRIES TO PULL THE INFORMATION OUT
      // OF THE ANCHOR
      if ( tag.equalsIgnoreCase("<a") )
      {
         content = content.trim();
         String lowercase_content = content.toLowerCase();
         char[] cont = content.toCharArray();

         // REMOVES POSSIBLITIES OF MAIL, JAVASCRIPT, OR FTP LINKS
COMING THROUGH PROCESS
         if ( lowercase_content.indexOf("href=\"mailto:") > -1 ||
            lowercase_content.indexOf("href=\"javascript") > -1 ||
            lowercase_content.indexOf("href=\"ftp:") > -1 )
         {
               int junk = 0;
         }
         else if ( ( i = lowercase_content.indexOf("href=\"") ) > -1
            || ( i = lowercase_content.indexOf("href=") ) > -1 )
         {
            // THIS IF STATEMENT CATCHES THE 'HREF' LINKS NEEDED TO
            // RECORD LINKS
            try
            {
               // SKIPS A QUOTE THAT MIGHT COME AFTER THE "HREF="
               // ATTRIBUTE
               if ( cont[i + 5] == '\"' )
                  i = i + 6;
               else
                  i = i + 5;

               // CHECKS IF THE LINK IS RELATIVE, IF SO, THEN THE HOST
               // IS PUT IN FRONT OF THE PATH
               if ( lowercase_content.indexOf("href=\"http://") < 0
                  && lowercase_content.indexOf("href=http://") < 0 )
                  word = "http://" + (String) url.getHost() +
                        url.getPath();

               // I FORGET WHAT THIS IS SUPPOSE TO DO
               if ( word.endsWith(".html") || word.endsWith(".htm") )
               {
                  int last_slash = word.lastIndexOf("/");
                  word = word.substring(0, last_slash + 1);
               }

               //CHECKS TO SEE IF THE END OF PATH HAS A FORWARD SLASH
               if ( word.length() != 0 )
                  if ( word.charAt( word.length() - 1 ) != '/' )
```

```
      //ADDS A BACKSLASH TO THE END OF THE PATH
      word = word + '/';

// IF FIRST CHARACTER IN LINK IS A FORWARD SLASH, THIS
// IS REMOVED BY SKIPPING IT
if ( cont[i] == '/' )
   i++;

// BUILDS THE REST OF THE LINK UNTIL ONE OF THE BELOW
// CHARACTERS IS INCOUNTERED
for (; i < content.length() && cont[i] != '\"'
   && cont[i] != '\r' && cont[i] != '\n' ;i++)
{
      word = word + cont[i];
}

// CHECKS TO SEE IF THE LINK IS ALREADY IN THE LIST. IF
// NOT, IT IS ADDED.

try {
   URL url_link = new URL(word);

   // TRIMS PATH NAME
   String url_path = url_link.getPath();
   url_path = url_path.trim();

   if ( word.indexOf('#') == -1 ) {
      // THESE IF STATEMENTS TRY TO ELIMINATE NON-HTML
      // BASED PAGES
      if ( url_path.indexOf('.') == -1 )
         HLinkTag.add(word);
      else if ( url_path.endsWith(".html") ||
          url_path.endsWith(".htm") ||
          url_path.endsWith(".asp") ||
          url_path.endsWith(".jsp") ||
          url_path.endsWith(".shtml") ||
          url_path.endsWith(".php") )
         HLinkTag.add(word);
   }

   if ( option == 1 && word.indexOf('#') == -1 ) {
      try {
         if ( url_path.indexOf('.') == -1 ||
               url_path.endsWith(".html") ||
               url_path.endsWith(".htm") ||
               url_path.endsWith(".asp") ||
               url_path.endsWith(".jsp") ||
               url_path.endsWith(".shtml") ||
                url_path.endsWith(".php") ) {
            Statement select = con.createStatement();
            select.executeUpdate("INSERT INTO
               html_parse_table VALUES(\"" +
               url.toString() + "\",\"a\",\"" +
               word + "\")" );
         }
      }
      catch ( SQLException sqle ) {
```

```
                }
            }
        }
        catch (MalformedURLException e) {}
    }
    catch ( ArrayIndexOutOfBoundsException e )
    {
        // THIS JUNK STATEMENT IS ADDED TO CATCH ARRAY
        // EXCEPTIONS
        int junk = 0;
    }
}

// REMOVES THE ANCHOR LINK FROM THE STACK
ParserStack.pop();
}
else if ( tag.equalsIgnoreCase("<meta") )
{
// THIS RUNS IF THE TAG IS A META TAG

// REMOVES THE META TAG FROM THE STACK
ParserStack.pop();
content = content.trim();

content = content.toLowerCase();

// MAKES SRUE THE META TAG ATTRIBUTES ARE KEYWORDS OR
// DESCRIPTION
if ( content.startsWith("name=\"keywords\" ") ||
    content.startsWith("name=\"description\" ") )
{
    // SETS THE INDEX TO THE START OF THE WORDS
    if ( content.startsWith("name=\"keywords\" ") )
        i = 25;
    else
        i = 28;

    char[] cont = content.toCharArray();

    // THIS WILL REPEAT UNTIL THE END OF THE CHARACTER ARRAY
    for (; i < content.length(); i++)
    {
        // IF NON-LETTER OR NON-NUMBER CHARACTER IS ENCOUNTERED,
        // THIS WILL TRY TO ADD THE WORD TO THE LIST
        if ( Character.isWhitespace(cont[i]) ||
            cont[i] == ',' || cont[i] == ';' ||
            cont[i] == '\\'  || cont[i] == '/'  ||
            cont[i] == '.' || cont[i] == '\"' ||
            cont[i] == '-' || cont[i] == '©' ||
            cont[i] == '!' || cont[i] == '?' ||
            cont[i] == '|' || cont[i] == '(' ||
            cont[i] == ')' || cont[i] == '\'' ||
            cont[i] == '[' || cont[i] == ']' ||
            cont[i] == '{' || cont[i] == '}' )
        {
            // IF THE WORD IS NOT COMMON AND IS NOT BLANK, IT
            // WILL BE ADDED TO THE LIST
```

```
                if ( word.length() != 0 && isNotCommon( word ) )
                {
                   MetaTree.insert( new String (word) );
                   if ( option == 1 ) {
                      try {
                         Statement select = con.createStatement();
                         select.executeUpdate( "INSERT INTO
                            html_parse_table VALUES(\"" +
                            url.toString() + "\",\"meta\",\"" +
                            word + "\")" );
                      }
                      catch ( SQLException sqle ) {
                      }
                   }
                }
                word = "";
             }
             else
                word = word + cont[i];
          }

          // THIS PICKS UP THE LAST WORD
          if ( word.length() != 0 && isNotCommon( word ) )
          {
             MetaTree.insert( new String (word) );
             if ( option == 1 ) {
                try {
                   Statement select = con.createStatement();
                   select.executeUpdate( "INSERT INTO
                      html_parse_table VALUES(\"" + url.toString() +
                      "\",\"meta\",\"" + word + "\")" );
                }
                catch ( SQLException sqle ) {
                }
             }
          }
       }
    }
}

// THIS FUNCTION PICKS UP THE WORDS THAT ARE NOT IN TAGS, BUT ARE
// ASSOCIATED WITH VALID TAGS
private void addContent(String content, String website )
          throws SQLException
{
    Statement select = con.createStatement();

    content = content.toLowerCase();

    // IF THE STACK IS EMPTY, ANY WORDS WILL BE ADDED TO THE BODY
    // LIST.
    if ( ParserStack.empty() )
    {
       BodyTree.insert( new String (content) );

       // IF THE USER CHOOSE TO USE THE DMBS SEARCH, THIS WILL INSERT
       // KEYWORD INTO DMBS TABLE
```

```
   if ( option == 1 ) {
      try {
         select.executeUpdate( "INSERT INTO html_parse_table
            VALUES(\"" + website + "\",\"body\",\"" + content
            + "\")" );
      }
      catch ( SQLException sqle ) { }
   }
}
else
{
   // THIS WILL CHECK TO SEE WHICH TAG LIST TO PUT THE WORD IN
   String tagpeek = (String) ParserStack.peek();

   // THIS TAG IS FOR BOLD
   if ( tagpeek.equalsIgnoreCase( "<b" ) )
   {
      BoldTree.insert( new String (content) );
      BodyTree.insert( new String (content) );

      // IF THE USER CHOOSE TO USE THE DMBS SEARCH, THIS WILL
      // INSERT KEYWORD INTO DMBS TABLE
      if ( option == 1 ) {
         try {
            select.executeUpdate( "INSERT INTO html_parse_table
                  VALUES(\"" + website + "\",\"body\",\"" +
                  content + "\")" );
            select.executeUpdate( "INSERT INTO html_parse_table
                  VALUES(\"" + website + "\",\"bold\",\"" +
                  content + "\")" );
         }
         catch ( SQLException sqle ) { }
      }
   }
   else if ( tagpeek.equalsIgnoreCase( "<body" )
            && isNotCommon( content ) )
   {
      // THIS TAG IS FOR BODY
      BodyTree.insert( new String (content) );

      // IF THE USER CHOOSE TO USE THE DMBS SEARCH, THIS WILL
      // INSERT KEYWORD INTO DMBS TABLE
      if ( option == 1 ) {
         try {
            select.executeUpdate( "INSERT INTO html_parse_table
                  VALUES(\"" + website + "\",\"body\",\"" +
                  content + "\")" );
         }
         catch ( SQLException sqle ) { }
      }
   }
   else if ( tagpeek.equalsIgnoreCase( "<em" ) )
   {
      // THIS TAG IS FOR EMPHASIS
      EmTree.insert( new String (content) );
      BodyTree.insert( new String (content) );
```

```
      // IF THE USER CHOOSE TO USE THE DMBS SEARCH, THIS WILL
      // INSERT KEYWORD INTO DMBS TABLE
      if ( option == 1 ) {
         try {
            select.executeUpdate( "INSERT INTO html_parse_table
                  VALUES(\"" + website + "\",\"body\",\"" +
                   content + "\")" );
            select.executeUpdate( "INSERT INTO html_parse_table
                  VALUES(\"" + website + "\",\"em\",\"" + content
                  + "\")" );
         }
         catch ( SQLException sqle ) { }
      }
   }
   else if ( tagpeek.equalsIgnoreCase( "<h1" ) ||
      tagpeek.equalsIgnoreCase( "<h2" ) ||
      tagpeek.equalsIgnoreCase( "<h3" ) ||
      tagpeek.equalsIgnoreCase( "<h4" ) ||
      tagpeek.equalsIgnoreCase( "<h5" ) ||
      tagpeek.equalsIgnoreCase( "<h6" ) )
   {
      // THIS TAG IS FOR HEADERS
      HeaderTree.insert( new String (content) );
      BodyTree.insert( new String (content) );

      // IF THE USER CHOOSE TO USE THE DMBS SEARCH, THIS WILL
      // INSERT KEYWORD INTO DMBS TABLE
      if ( option == 1 ) {
         try {
            select.executeUpdate( "INSERT INTO html_parse_table
                  VALUES(\"" + website + "\",\"body\",\"" +
                  content + "\")" );
            select.executeUpdate( "INSERT INTO html_parse_table
                  VALUES(\"" + website + "\",\"header\",\"" +
                  content + "\")" );
         }
         catch ( SQLException sqle ) { }
      }
   }
   else if ( tagpeek.equalsIgnoreCase( "<i" ) )
   {
      // THIS TAG IS FOR ITALICS
      ItalicTree.insert( new String (content) );
      BodyTree.insert( new String (content) );

      // IF THE USER CHOOSE TO USE THE DMBS SEARCH, THIS WILL
      // INSERT
      // KEYWORD INTO DMBS TABLE
      if ( option == 1 ) {
         try {
            select.executeUpdate( "INSERT INTO html_parse_table
                  VALUES(\"" + website + "\",\"body\",\"" +
                  content + "\")" );
            select.executeUpdate( "INSERT INTO html_parse_table
            VALUES(\"" + website + "\",\"i\",\"" + content +
            "\")" );
         }
```

```
                catch ( SQLException sqle ) { }
            }
        }
        else if ( tagpeek.equalsIgnoreCase( "<title" ) )
        {
            // THIS TAG IS FOR TITLES
            TitleTree.insert( new String (content) );

            // IF THE USER CHOOSE TO USE THE DMBS SEARCH, THIS WILL
            // INSERT
            // KEYWORD INTO DMBS TABLE
            if ( option == 1 ) {
                try {
                    select.executeUpdate( "INSERT INTO html_parse_table
                          VALUES(\"" + website + "\",\"title\",\"" +
                          content + "\")" );
                }
                catch ( SQLException sqle ) { }
            }
        }
    }
}

// THIS FUNCTION IS TO CHECK THAT THE WORD IS NOT A COMMON WORD
private boolean isNotCommon(String content)
{
    if ( content.equalsIgnoreCase( "a" ) ||
         content.equalsIgnoreCase( "the" ) ||
         content.equalsIgnoreCase( "an" ) ||
         content.equalsIgnoreCase( "and" ) ||
         content.equalsIgnoreCase( "or" ) ||
         content.equalsIgnoreCase( " " ) ||
         content.equalsIgnoreCase( "of" ) ||
         content.equalsIgnoreCase( "to" ) ||
         content.equalsIgnoreCase( "is" ) )
        return false;
    else
        return true;
}
}
```

Matching.java

```
/*************************************************************************
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

MATCHING IS A THREADED CLASS THAT MATCHES INCOMING PARSED URL
ADDRESSES (THEIR KEYWORD BUCKETS) TO THE QUERY STATEMENT PROVIDED
BY THE USER.  ALL MATCHED URLS ARE RETURNED TO THE MAIN THREAD FOR
DISPLAY.
*************************************************************************/

import java.util.*;
import java.io.*;
import tree.*;

// MATCHING CLASS IS DESIGN TO SEE IF THE URL SATISIFIES THE
// QUERY STATEMENT
public class Matching {
   Stack stack_buffer;
   String condition_word;

   // TAG PUSHED OR PULLED FROM THE STACK
   String stack_tag;
   // KEYWORD PUSHED OR PULLED FROM THE STACK
   String stack_keyword;
   String stack_item;
   // ITEM FROM THE CONDITIONAL STATEMENT
   String conditional;
   // RESULT OF A CONDITION PHRASE
   String first_condition_result;
   // RESULT OF A CONDITION PHRASE
   String second_condition_result;
   // BOOLEAN RESULT OF THE FIRST AND SECOND CONDITION
   String comparsion_result;
   // FLAG FOR INFORMING THE APP IF A SINGLE CONDITION IN THE WHERE
   // CLAUSE HAS BEEN SATISIFIED
   boolean singular_result;
   // FLAG TO KEEP THE PROGRAM LOOPING UNTIL FINISHED
   boolean keep_looping;

   // TREE TO STORE URLS THAT MATCH THE QUERY
   AATree matched_url_tree;

   // CONDITION LIST FROM THE MAIN THREAD
   LinkedList condition_list;

   // NODE TO HOUSE THE PAGE'S PARSED INFORMATION
   ParsedHtmlNode parsed_node;

   // QUEUE OF URLS THAT SATIFY THE QUERY
   SynchQueue matched_url_list;
```

```
// DEBUGGING TOOL - NAME OF THREAD
String name;
// USED FOR TRACKING THE OVERALL TIME OF EXECUTION
TimeNode time_node;

// TRAVERSAL METHOD
TraversalMethods traversal_method;
// TELLS QUERY PROCESSOR IF PRUNING IS BEING USED
boolean pruning_traversal;
String threadname;

// CONSTRUCTOR FOR MATCHING CLASS
public Matching( LinkedList condition_list,
         SynchQueue matched_url_list, String name,
         TimeNode time_node, TraversalMethods traversal_method,
         boolean pruning_traversal, AATree matched_url_tree)
{
   this.condition_list = condition_list;
   this.matched_url_list = matched_url_list;
   this.name = name;
   this.time_node = time_node;
   this.traversal_method = traversal_method;
   this.pruning_traversal = pruning_traversal;
   this.matched_url_tree = matched_url_tree;
}

// SETS THE PAGE CONTENT THAT WILL BE MATCHED
public void setContent(ParsedHtmlNode parsed_node,
         String threadname) {
   this.parsed_node = parsed_node;
   this.threadname = threadname;
}

// RUNNER FOR MATCHING THREAD
public int run()
{
   LinkedList working_condition_list;
   String result;

   long start = System.currentTimeMillis();

   // CALLS MATCHING FUNCTION
   result = matching( parsed_node );

   long finish = System.currentTimeMillis();
   double execution_time = (finish - start) / 1000.0;

   time_node.execution = time_node.execution + execution_time;

   // IF THE URL SATISIFIES THE QUERY, THEN IT IS PULLED ONTO THE
   // MATCHED QUEUE
   if ( result.compareTo( "true" ) == 0 ) {
         MatchedNode matched_node = new MatchedNode();
     matched_node.url = parsed_node.url;
     matched_node.p_time = parsed_node.p_time;
     matched_node.m_time = execution_time;
```

```
      matched_node.origin = parsed_node.origin;

      matched_url_list.addNode( matched_node );
   }

   // SETS THE IF PART OR ALL OF THE QUERY WAS SATISFIED
   if ( result.compareTo("false") == 0 && singular_result == false )
      return 0;
   else if ( result.compareTo("false") == 0 &&
         singular_result == true )
      return 1;
   else
      return 2;
}

// THIS MATCHING CLASS EXAMINES THE PARTS OF THE CONDITIONAL
// STATEMENT TO EVALUATE IF THE URL ADDRESS SATISIFIES THE QUERY
// STATEMENT
public String matching( ParsedHtmlNode parsed_node ) {
   // ITERATOR FOR MOVING THROUGH THE CONDITIONAL LIST.
   ListIterator iter = condition_list.listIterator(0);

   // CREATES THE STACK TO FOR EVALUATION
   stack_buffer = new Stack();

   // SETS THE LOOPING FLAG
   keep_looping = true;
   stack_buffer.empty();

   // WILL KEEP LOOPING UNTIL THE END OF THE CONDITIONAL LIST IS
   // REACHED
   while ( keep_looping ) {

      // PULLS CONDITIONAL ITEMS FROM THE CONDITIONAL LIST
      if ( iter.hasNext() )
         condition_word = (String) iter.next();
      else
         condition_word = "";

      // IF STATEMENT WILL CATCH THE OPEN PARATHESISES.
      if ( condition_word.compareTo( "(" ) == 0 ) {
         stack_buffer.push( "(" );
      }
      else if ( condition_word.compareTo( ")" ) == 0 ) {
         // IF THE CODNTIONAL ITEM IS A CLOSE PARATHESISES,
         // MATCHING HAS TO EVALUATE THE ITEMS ON THE STACK
         // UNTIL IT REACHES THE OPEN PARATHESIS

         while ( true ) {

            // POPS FIRST CONDITION FROM THE TOP OF THE STACK
            first_condition_result = (String) stack_buffer.pop();
            stack_item = (String) stack_buffer.pop();

            // IF SECOND ITEM IS A PARENTHESIS...
            if ( stack_item.compareTo( "(" ) == 0 ) {
```

```
            // THEN THE END OF THE PARENTHESIS BLOCK IS HERE,
            // POPS THE NEW RESULT INTO THE STACK AND BREAKS FROM
            // STACK.

            stack_buffer.push( first_condition_result );
            break;

         }
         else {

            // IF SECOND ITEM IS AN "OR" OR AN "AND"...
            // SAVES STACK_ITEM
            conditional = stack_item;

         }

         second_condition_result = (String) stack_buffer.pop();

         // EVALUATE RESULTS FROM FIRST AND SECOND CONDITION
         // USING CONDITIONAL
         comparsion_result = evaluateConditional(
            first_condition_result, second_condition_result,
            conditional);

         // PUSH RESULTS BACK ON STACK
         stack_buffer.push( comparsion_result );
      }
   }
   else if ( ! iter.hasNext() ) {
      // IF THE CONDITIONAL LIST DOES NOT HAVE ANY CONDITIONAL
      // ITEMS LEFT, THIS WILL EVALUATE TO THE BTOOM OF THE LIST.
      while ( true ) {

         // A STACK OF SIZE 1 MEANS THE BOTTOM OF THE STACK
         // HAS BEEN REACHED

         if ( stack_buffer.size() == 1 ) {
            keep_looping = false;
            break;
         }

         // POPS TWO ITEMS FROM THE TOP OF THE STACK
         first_condition_result = (String) stack_buffer.pop();
         conditional = (String) stack_buffer.pop();

         second_condition_result = (String) stack_buffer.pop();

         // EVALUATE RESULTS FROM FIRST AND SECOND CONDITION
         // USING CONDITIONALS
         comparsion_result = evaluateConditional(
            first_condition_result, second_condition_result,
            conditional);

         // PUSH RESULTS BACK ON STACK
         stack_buffer.push( comparsion_result );
      }
   }
```

```
else if ( condition_word.compareTo( "and" ) == 0 ) {
   // IF AN "AND" IS ENCOUNTERED AS PART OF THE CONDITIONAL
   // LIST, THEN
   // THE FOLLOWING KEYWORD AND TAG ARE EVALUATED, AND THEN
   // THOSE RESULTS ARE EVALUATED AGAINST THE PREVIOUS
   // KEYWORD/TAG USING THE "AND" OPERATOR

   // POPS FIRST CONDITION BOOLEAN FROM STACK
   first_condition_result = (String) stack_buffer.pop();

   // PULLS SECOND CONDITION STATEMENT FROM QUEUE
   stack_tag = (String) iter.next();

   if ( stack_tag.compareTo( "(" ) == 0 ) {
      // "AND" FALLS BEFORE A OPEN PARENTHESE, SO "AND"
      // AND "(" IS SAVED TO THE STACK.

      stack_buffer.push( first_condition_result );
      stack_buffer.push( "and" );
      stack_buffer.push( "(" );
   }
   else {
      // "AND" DOES NOT FALL BEFORE AN OPEN PARENTHESE, THUS,
      // KEYWORD/TAG PAIRS ARE EVALUTED

      stack_keyword = (String) iter.next();

      // EVALUATES SECOND SET OF ITEMS
      second_condition_result = evaluateTagAndKeyword(
         stack_tag, stack_keyword, parsed_node );

      if ( second_condition_result.compareTo("true") == 0 )
         singular_result = true;

      // EVAULATES FIRST CONDITION AND SECOND CONDITION USING
      // "AND"
      comparsion_result = evaluateConditional(
         first_condition_result, second_condition_result,
         "and");

      // PUSHES THE COMPARSION RESULTS OF THESE TWO CONDTIONS
      // ONTO THE STACK.
      stack_buffer.push( comparsion_result );
   }
}
else {
   if ( condition_word.compareTo("or") == 0 ) {
      // IF "OR" IS ENCOUNTERED, IT IS PUSHED ON TO THE STACK
      stack_buffer.push("or");
   }
   else {
      // KEYWORD/TAG PAIR ARE ENCOUNTER AND EVALUATED AND
      // PUSHED ONTO THE STACK

      stack_keyword = (String) iter.next();

      comparsion_result = evaluateTagAndKeyword(
```

```
                condition_word, stack_keyword, parsed_node );

            if ( comparsion_result.compareTo("true") == 0 )
                singular_result = true;

            // PUSH RESULTS BACK ON STACK
            stack_buffer.push( comparsion_result );
        }
    }
}

// RETURNS THE LAST BOOLEAN EXPRESSION IN THE STACK.  THIS
// EXPRESSION REVEALS IF THE URL STATISFIES THE QUERY.
return ( (String) stack_buffer.pop() );
}

// THIS METHOD IS EVALUATES THE TAG AND KEYWORD PAIR.
private String evaluateTagAndKeyword( String tag, String keyword,
    ParsedHtmlNode parsed_node ) {

    if ( tag.compareTo("b") == 0 ) {
        // IF TAG IS BOLD, THEN CHECKS IF THE KEYWORD IS IN THE BOLD
        // TREE
        if ( parsed_node.BoldTree.find( keyword ) != null )
            return("true");
        else
            return("false");
    }
    else if ( tag.compareTo("body") == 0 ) {
        // IF TAG IS BODY, THEN CHECKS IF THE KEYWORD IS IN THE BODY
        // TREE
        if ( parsed_node.BodyTree.find( keyword ) != null ) {
            return("true");
        }
        else {
            return("false");
        }
    }
    else if ( tag.compareTo("title") == 0 ) {
        // IF TAG IS TITLE, THEN CHECKS IF THE KEYWORD IS IN THE TITLE
        // TREE
        if ( parsed_node.TitleTree.find( keyword ) != null )
            return("true");
        else
            return("false");
    }
    else if ( tag.compareTo("header") == 0 ) {
        // IF TAG IS HEADER, THEN CHECKS IF THE KEYWORD IS IN THE
        // HEADER TREE
        if ( parsed_node.HeaderTree.find( keyword ) != null )
            return("true");
        else
            return("false");
    }
    else if ( tag.compareTo("i") == 0 ) {
        // IF TAG IS ITALIC, THEN CHECKS IF THE KEYWORD IS IN THE
        // ITALIC TREE
```

```
      if ( parsed_node.ItalicTree.find( keyword ) != null )
         return("true");
      else
         return("false");
   }
   else if ( tag.compareTo("em") == 0 ) {
      // IF TAG IS EMPHASIS, THEN CHECKS IF THE KEYWORD IS IN THE EM
      // TREE
      if ( parsed_node.EmTree.find( keyword ) != null )
         return("true");
      else
         return("false");
   }
   else if ( tag.compareTo("meta") == 0 ) {
      // IF TAG IS META, THEN CHECKS IF THE KEYWORD IS IN THE BOLD
      // TREE
      if ( parsed_node.MetaTree.find( keyword ) != null )
         return("true");
      else
         return("false");
   }
   else {
   //if ( tag.compareTo("*") == 0 ) {
      // IF TAG IS *, THEN CHECKS IF IN ONE OF THE TAG TREES

      if ( parsed_node.BoldTree.find( keyword ) != null )
         return("true");
      if ( parsed_node.TitleTree.find( keyword ) != null )
         return("true");
      if ( parsed_node.BodyTree.find( keyword ) != null )
         return("true");
      if ( parsed_node.HeaderTree.find( keyword ) != null )
         return("true");
      if ( parsed_node.ItalicTree.find( keyword ) != null )
         return("true");
      if ( parsed_node.EmTree.find( keyword ) != null )
         return("true");
      if ( parsed_node.MetaTree.find( keyword ) != null )
         return("true");

      return ("false");
   }
}

// FUNCTION WILL TEST TO BOOLEAN CONDITIONS AGAINST EACH OTHER.
//  THESE BOOLEAN
// CONDITIONS REPRESENT KEYWORD/TAG PAIRS
private String evaluateConditional( String first_condition_result,
   String second_condition_result, String conditional) {

   // CONVERTS BOTH CONDITIONS TO BOOLEAN EXPRESSIONS.
   Boolean first_condition =
         Boolean.valueOf(first_condition_result);
   Boolean second_condition =
         Boolean.valueOf(second_condition_result);

   if ( conditional.compareTo("and") == 0 ) {
```

```
            // COMPARES USING THE "AND"
            if ( first_condition.booleanValue() )
                if ( second_condition.booleanValue() )
                   return ("true");
                else
                   return ("false");
            else
               return ("false");
        }
        else {
            // COMPARES USING THE "OR"
            if ( first_condition.booleanValue() )
               return("true");
            else
                if ( second_condition.booleanValue() )
                   return ("true");
                else
                   return ("false");
        }
    }
}
```

TraversalMethods.java

```
/**************************************************************************
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

INTERFACE FOR THE TRAVERSAL METHODS.  ALLOWS THE USER TO CHOOSE ONE
OF TWO DIFFERENT IMPLEMENTATIONS TO TRAVERSE THE WEB.
**************************************************************************/

import java.util.*;

interface TraversalMethods {
   void insertURL(LinkedList valid_urls, int parent_level,
      String thread);
   Node removeURL(String thread);
   boolean isEmpty();
   boolean traversalComplete();
   boolean threadsWaiting();
   public int getTotalInsertedLinks();
   public int getTotalRemovedLinks();
}
```

BreadthFirstTraversal.java

```
/**************************************************************************
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

BREADTHFIRSTTRAVERSAL.JAVA ALLOWS THE SPIDER TO SEARCH THE WEB IN A
BREADTH-FIRST TRAVERSAL.
**************************************************************************/

import java.util.LinkedList;
import tree.*;
import java.lang.Exception;
import java.net.*;
import java.io.*;

public class BreadthFirstTraversal implements TraversalMethods {
   // CONTAINS THREAD GROUP
   private ThreadGroup spiderThreadGroup;
   // HOUSES THE LINKS THAT WILL BE SEARCHED
   private LinkedList URLQueue = new LinkedList();
   // HOUSES THE LINKS THAT HAVE ALREADY BEEN SEARCHED
   private AATree visited_url_tree = new AATree();
   // SETS THE MAX DEPTH LEVEL
   private int max_depth_level = 0;
   // SETS THE INITIAL THREAD COUNT
   private int thread_number = 0;

   // COUNTERS FOR HYPERLINKS
   int links;
   int de_links;
   int redirects;

   // COUNTER FOR THE NUMBER OF THREADS WAITING
   private int thread_waiting;
   // LETS THE TRAVERSAL KNOW WHEN IT HAS COMPLETED
   private boolean traversal_complete;


   // CONSTRUCTOR
   public BreadthFirstTraversal(ThreadGroup spiderThreadGroup,
            int max_depth_level, int thread_number) {
      this.spiderThreadGroup = spiderThreadGroup;
      this.max_depth_level = max_depth_level;
      this.thread_number = thread_number;

      links = 0;
      de_links = 0;

      traversal_complete = false;
      thread_waiting = 0;
   }
```

```
// METHOD WHICH INSERTS URLS INTO THE HYPERLINK QUEUE
synchronized public void insertURL(LinkedList valid_urls,
        int parent_level, String thread) {
   Node url_address_node;

   // LOOPS THROUGH A PAGES HYPERLINKS FOR INPUT INTO THE HYPERLINK
   // QUEUE
   while ( valid_urls.size() != 0 ) {
        // REMOVES LINK FROM QUEUE
        String url_address = (String) valid_urls.removeFirst();

        url_address_node = new Node();

        if (  url_address.endsWith( "/" ) )
           url_address_node.content =
              url_address.substring(0, url_address.length() - 1 );
        else
           url_address_node.content = url_address;

        // CHECKS CHILD URL AGAINST USED LIST FOR DUPS
        if ( visited_url_tree.find( url_address_node.content ) ==
              null ) {

           url_address_node.occurance = parent_level + 1;
           links++;

           // ADDS LINK TO VISITED LIST
           visited_url_tree.insert( url_address_node.content );

           // ADDS LINK TO THE HYPERLINK LIST
           URLQueue.addLast(url_address_node);
        }
   }
   // UNLOCKS THE OBJECT
   notifyAll();
}

// METHOD WHICH REMOVES URLS FROM THE HYPERLINK QUEUE
synchronized public Node removeURL(String thread) {
   Node first_node = new Node();

   // CATCHES THE THREAD INCASE THERE ARE NO LINKS TO PARSE
   while ( URLQueue.isEmpty() && !traversal_complete ) {
      thread_waiting++;
      try {
         wait(1,0);
         // IF ALL THREADS ARE WAITING, SETS THE TRAVERSAL TO
         // COMPLETE
         if ( thread_number <= thread_waiting ) {
            traversal_complete = true;
         }
      } catch (InterruptedException e) { }
      thread_waiting--;
   }

   // PRINTS THE NUMBER OF HYPERLINKS THAT NEED TO BE PARSED
```

```
      if ( traversal_complete == false )
         System.out.println(URLQueue.size() + " hyperlinks in storage
            queue remains to be parsed.");

      // CHECKS TO MAKE SURE THE QUEUE IS NOT EMPTY BEFORE DE-QUEUING
      if ( !isEmpty() && !traversal_complete ) {
         de_links++;
         // REMOVES LINK FROM QUEUE
         first_node = (Node) URLQueue.removeFirst();
      }
      else {
         first_node = null;
      }

      // UNLOCKS OBJECT
      notifyAll();
      return first_node;
   }

   // RETURNS IF THE HYPERLINK QUEUE IS EMPTY
   public boolean isEmpty() {
      if ( URLQueue.size() == 0 )
         return true;
      else
         return false;
   }

   // RETURNS IF THE TRAVERSAL IS COMPLETED
   public boolean traversalComplete() {
      return traversal_complete;
   }

   // RETURNS THE NUMBER OF THREADS WAITING FOR A HYPERLINK IN THE
   // OBJECT
   public boolean threadsWaiting() {
      return ( thread_number <= thread_waiting );
   }

   // RETURNS THE TOTAL INSERTED LINKS IN QUEUE
   public int getTotalInsertedLinks() {
      return links;
   }

   // RETURNS THE TOTAL REMOVED LINKS FROM THE QUEUE
   public int getTotalRemovedLinks() {
      return de_links;
   }
}
```

DepthFirstTraversal.java

```
/**************************************************************************
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

DEPTHFIRSTTRAVERSAL.JAVA ALLOWS THE SPIDER TO SEARCH THE WEB IN A
DEPTH-FIRST TRAVERSAL.
**************************************************************************/

import java.util.LinkedList;
import tree.*;
import java.lang.Exception;

public class DepthFirstTraversal implements TraversalMethods {
   // CONTAINS THREAD GROUP
   private ThreadGroup spiderThreadGroup;
   // HOUSES THE LINKS THAT WILL BE SEARCHED
   private LinkedList URLQueue = new LinkedList();
   // HOUSES THE LINKS THAT HAVE ALREADY BEEN SEARCHED
   private AATree visited_url_tree = new AATree();
   // SETS THE MAX DEPTH LEVEL
   private int max_depth_level = 0;
   // SETS THE INITIAL THREAD COUNT
   private int thread_number = 0;

   // COUNTERS FOR HYPERLINKS
   int links;
   int de_links;

   // COUNTER FOR THE NUMBER OF THREADS WAITING
   private int thread_waiting;
   // LETS THE TRAVERSAL KNOW WHEN IT HAS COMPLETED
   private boolean traversal_complete;

   // CONSTRUCTOR
   public DepthFirstTraversal(ThreadGroup spiderThreadGroup,
            int max_depth_level, int thread_number) {
      this.spiderThreadGroup = spiderThreadGroup;
      this.max_depth_level = max_depth_level;
      this.thread_number = thread_number;

      links = 0;
      de_links = 0;

      traversal_complete = false;
      thread_waiting = 0;
   }

   // METHOD WHICH INSERTS URLS INTO THE HYPERLINK QUEUE
   synchronized public void insertURL(LinkedList valid_urls,
            int parent_level, String thread) {
```

```
   Node url_address_node;

   // LOOPS THROUGH A PAGES HYPERLINKS FOR INPUT INTO THE HYPERLINK
   // QUEUE
   while ( valid_urls.size() != 0 ) {
      // REMOVES LINK FROM QUEUE
      String url_address = (String) valid_urls.removeFirst();

      url_address_node = new Node();

      if (  url_address.endsWith( "/" ) )
         url_address_node.content =
               url_address.substring(0, url_address.length() - 1 );
      else
         url_address_node.content = url_address;

      // CHECKS CHILD URL AGAINST USED LIST FOR DUPS
      if ( visited_url_tree.find(url_address_node.content) == null )
      {

         url_address_node.occurance = parent_level + 1;
         links++;

         // ADDS LINK TO VISITED LIST
         visited_url_tree.insert( url_address_node.content );

         // ADDS LINK TO THE HYPERLINK LIST
         URLQueue.addFirst(url_address_node);
      }
   }
   // UNLOCKS THE OBJECT
   notifyAll();
}

// METHOD WHICH REMOVES URLS FROM THE HYPERLINK QUEUE
synchronized public Node removeURL(String thread) {
   Node first_node = new Node();

   // CATCHES THE THREAD INCASE THERE ARE NO LINKS TO PARSE
   while ( URLQueue.isEmpty() && !traversal_complete ) {
      thread_waiting++;
      try {
         wait(1,0);
         // IF ALL THREADS ARE WAITING, SETS THE TRAVERSAL TO
         // COMPLETE
         if ( thread_number <= thread_waiting ) {
            traversal_complete = true;
         }
      } catch (InterruptedException e) { }
      thread_waiting--;
   }

   // PRINTS THE NUMBER OF HYPERLINKS THAT NEED TO BE PARSED
   if ( traversal_complete == false )
      System.out.println(URLQueue.size() + " hyperlinks in storage
         queue remains to be parsed.");
```

```
        // CHECKS TO MAKE SURE THE QUEUE IS NOT EMPTY BEFORE DE-QUEUING
        if ( !isEmpty() && !traversal_complete) {
            de_links++;
            // REMOVES LINK FROM QUEUE
            first_node = (Node) URLQueue.removeFirst();
        }
        else {
            first_node = null;
        }

        // UNLOCKS OBJECT
        notifyAll();
        return first_node;
    }

    // RETURNS IF THE HYPERLINK QUEUE IS EMPTY
    public boolean isEmpty() {
        if ( URLQueue.size() == 0 )
            return true;
        else
            return false;
    }

    // RETURNS IF THE TRAVERSAL IS COMPLETED
    public boolean traversalComplete() {
        return traversal_complete;
    }

    // RETURNS THE NUMBER OF THREADS WAITING FOR A HYPERLINK IN THE
    // OBJECT
    public boolean threadsWaiting() {
        return ( thread_number <= thread_waiting );
    }

    // RETURNS THE TOTAL INSERTED LINKS IN QUEUE
    public int getTotalInsertedLinks() {
        return links;
    }

    // RETURNS THE TOTAL REMOVED LINKS FROM THE QUEUE
    public int getTotalRemovedLinks() {
        return de_links;
    }
}
```

Node.java

```
/**************************************************************************
*
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

Node.java is a houses the keyword and its score as QueryBase transfers

its content in the application
**************************************************************************
/

public class Node{

   public String content;

   public int occurance;


   public Node( )

   {

      content = "";

      occurance = 0;

   }


   // TEST TO SEE IF THE CONTENT IS EQUALED

   public boolean equals( String word )

   {

      if ( content.equalsIgnoreCase( word ) )

         return true;

      else

         return false;

   }
```

```
    // INCREMENTS THE OCCURRANCE OF THE CONTENT

    public void increment()
    {
        occurance++;
    }
}
```

StatementParser.java

```
/**************************************************************************
*
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

STATEMENTPARSER.JAVA IS THE QUERY PARSER FOR THE REAL-TIME SEARCH
ENGINE.
**************************************************************************
/

import java.io.*;
import java.util.*;
import java.net.*;

public class StatementParser {
   public static ParsedStatementNode parse(String SQLStatement) {

      LinkedList condition_list = new LinkedList();
      LinkedList primer_urls = new LinkedList();
      LinkedList req_tags_list = new LinkedList();
      int max_depth_level = 0;
      boolean search_database = false;

      String search_mode = "";
      boolean pruning_mode = false;
      int thread_count = 0;

      condition_list.clear();

      // PREPARES VALID TAG ARRAY
      String[] valid_tag = { "*", "b", "body", "em", "header", "i",
            "meta", "title" };
      Arrays.sort(valid_tag);

      char letter;
      String line = "";
      String tag = "";
      int i = 0;

      // RETRIEVES THE INCOMING QUERY STATEMENT AND CONVERTS TO
      // LOWERCASE
      line = SQLStatement;

      if ( line.length() == 0 ) {
            System.out.println("*** Error: QueryBase query must start
                  with a \"FROM\" statement. ***");
            return (null);
      }

      // IF THE USER FORGETS TO PUT A SEMI-COLON AT THE END OF THE
```

```
      // STATEMENT, THIS WILL ADD IT
      if ( line.charAt(line.length() - 1) != ';')
         line = line + ";";

      line = line.toLowerCase();

      char[] array = line.toCharArray();

//*********************************************************************
         // ENSURES THE STATEMENT STARTS WITH A FROM CLAUSE
         if ( !line.startsWith( "select " ) )
         {
            System.out.println("*** Error: QueryBase query must start
                  with a \"SELECT\" statement. ***");
            markError(0, array, line.length(), 0 );
            return (null);
         }

//*********************************************************************
         // PARSES SELECT CLAUSE
         tag = "";
         search_database = false;
         i = 7;
         Node node;

         while ( array[i] != ' ' && array[i] != '\n' && array[i] != ';'
)
         {
            tag = tag + array[i];
            i++;
         }

         // CATCHES THE SEARCH MODE IN THE SELECT CLAUSE
         // IF WORD IS VALID TAG IT WILL ADD THE TAG TO BE SEARCHED
         if ( tag.length() != 0 && ( tag.compareTo("breadth-web") == 0
            || tag.compareTo("breadth-cache") == 0 ||
            tag.compareTo("breadth-database") == 0 ||
            tag.compareTo("depth-web") == 0 ||
            tag.compareTo("depth-cache") == 0 ||
            tag.compareTo("depth-database") == 0 ) )
         {
            search_mode = tag;
            tag = "";
         }
         else
         {
            // PRINTS ERROR IF WORD IS NOT A VALID TAG AND EXITS
            // FUNCTION
            System.out.println("*** Last Error: Invalid search mode
                  specified in Select statement ***");

            if ( array[i] == '\n' )
               markError(i-1, array, line.length(), tag.length() );
            else
               markError(i, array, line.length(), tag.length() );
            return(null);
         }
```

```
i++;

// CHECKS FOR THE WITH PRUNING CLAUSE
while (array[i] != ' ' && array[i] != '\n' && array[i] != ';')
{
   tag = tag + array[i];
   i++;
}

//System.out.println(tag);

if ( tag.compareTo("from") != 0 ) {
   if ( tag.compareTo("with") != 0 ) {
      // PRINTS ERROR IF WORD IS NOT A VALID TAG AND EXITS
      // FUNCTION
      System.out.println("*** Last Error: Expecting WITH
         PRUNING or FROM clause ***");

      if ( array[i] == '\n' )
         markError(i-1, array, line.length(), tag.length() );
      else
         markError(i, array, line.length(), tag.length() );
      return(null);
   }
   i++;

   // CLEARS TAG FOR "PRUNING" WORD
   tag = "";

   while (array[i] != ' ' && array[i] != '\n' && array[i] !=
         ';' )
   {
      tag = tag + array[i];
      i++;
   }
   //System.out.println(tag);

   if ( tag.compareTo("pruning") == 0 ) {
      pruning_mode = true;
      tag = "";
   }
   else
   {
      // PRINTS ERROR IF WORD IS NOT A VALID TAG AND EXITS
      // FUNCTION
      System.out.println("*** Last Error: Expecting WITH
         PRUNING clause ***");

      if ( array[i] == '\n' )
         markError(i-1, array, line.length(), tag.length() );
      else
         markError(i, array, line.length(), tag.length() );
      return(null);
   }
   i++;

   //PULLS NEXT WORD IN THE STATEMENT
```

```
   while ( array[i] != ' ' && array[i] != '\n' && array[i] !=
         ';' )
   {
      tag = tag + array[i];
      i++;
   }
}

//System.out.println(tag);

if ( tag.compareTo("from") != 0 ) {
   // PRINTS ERROR IF WORD IS NOT A VALID TAG AND EXITS
         FUNCTION
   System.out.println("*** Last Error: Expecting FROM clause
         ***");

   if ( array[i] == '\n' )
      markError(i-1, array, line.length(), tag.length() );
   else
      markError(i, array, line.length(), tag.length() );
   return(null);
}
// CLEARS TAG
tag = "";
i++;

// RETRIEVES ALL HYPERLINKS IN FROM CLAUSE

// LOOP WHILE SPACE IS NOT ENCOUNTERED OR SEMI-COLON
while ( i < line.length() &&  array[i] != ' ' && array[i] !=
   '\n' && array[i] != ';' )
{
   // IF COMMA IS ENCOUNTERED, IT WILL TRY TO READ THE URL,
         AND THEN ADD TO HYPERLINK LIST
   // IF URL IS READABLE.
   if ( array[i] == ',' && search_database == false )
   {
      try
      {
         URL url = new URL(tag);
         url.openStream();

         //node = new Node( );
         //node.content = tag;
         //node.occurance = 0;
         System.out.println(tag);

         primer_urls.add( tag );
         tag = "";
      }
      catch (MalformedURLException mue)
      {
         // ERRORS IF URL IS NOT VALID AND EXITS FUNCTION
         System.out.println("*** Error: Invalid URL specified
               in FROM statement ***");
         markError(i, array, line.length(), tag.length() );
         return(null);
```

```
            }
            catch (IOException ioe)
            {
               // ERRORS IF URL IS UNREACHABLE AND EXITS FUNCTION
               System.out.println("*** Error: Unable to connect to
                     URL specified in FROM statement ***");
               markError(i, array, line.length(), tag.length() );
               return(null);
            }
         }
         else if ( array[i] == '*' )
            // IF ASTERISK IS ENCOUNTERED, IT WILL SET THE PROGRAM
               TO SEARCH IN THE URL INDEX INSTEAD OF
            // RUNNING QUERIES
            search_database = true;
         else
            // BUILDS THE TAGS CHARACTERS AT A TIME.
            tag = tag + array[i];
         i++;
      }
      //System.out.println(tag);

      // CATCHES THE LAST HYPERLINK IN THE FROM CLAUSE
      tag = tag.trim();
      if ( tag.length() != 0 && search_database == false )
      {
         try
         {
            URL url = new URL(tag);
            url.openStream();

            //node = new Node( );
            //node.content = tag;
            //node.occurance = 0;
            primer_urls.add( tag );
            tag = "";
         }
         catch ( MalformedURLException mue )
         {
            // ERRORS IF URL IS NOT VALID AND EXITS FUNCTION
            System.out.println("*** Error: Invalid URL specified in
               FROM statement ***");
            markError(i, array, line.length(), tag.length() );
            return(null);
         }
         catch ( IOException ioe )
         {
            // ERRORS IF URL IS UNREACHABLE AND EXITS FUNCTION
            System.out.println("*** Error: Unable to connect to URL
               specified in FROM statement ***");
            markError(i, array, line.length(), tag.length() );
            return(null);
         }

         if ( search_database == false && primer_urls.size() == 0 )
         {
```

```
                // ERRORS IF NO URL IS SPECIFIED AND EXITS FUNCTION
                System.out.println("*** Error: No URL's where specified
                   in FROM statement ***");
                markError(i, array, line.length(), tag.length() );
                return(null);
             }

             // CLEARS THE HYPERLINKS LINKED LIST TO SIGNIFY THAT THE
                URL INDEX IS SUPPOSE TO BE RUN.
             if ( search_database == true )
                primer_urls.clear();
          }
//*************************************************************************
          // CHECKS FOR THE "WHERE" OR "DEPTH OF" CLAUSE
          tag = "";
          i++;
          for (; i < line.length() && array[i] != ' '; i++ )
             tag = tag + array[i];

          // IF THE SELECT STATEMENT ENDS BEFORE THE WHERE OR DEPTH OF
          // CLAUSE, THIS WILL CATCH IT
          if ( i == line.length() )
          {
             // ERRORS WHEN DEPTH OR WHERE STATEMENT IS NOT AVAILABLE
             // AND EXITS FUNCTION
             System.out.println("*** Error: QueryBase query must have a
                   \"DEPTH\" clause with optional \"WHERE\" statement.
                   ***");
             markError(i-1, array, line.length(), 0 );
             return(null);
          }

          tag = tag.trim();

          // IF CLAUSE IS NOT A DEPTH CLAUSE, IT CHECKS IF IT IS A WHERE
          // CLAUSE
          if ( !tag.equalsIgnoreCase( "depth" ) )
          {
             // CHECKS IF CLAUSE IS A WHERE CLAUSE
             if ( !tag.equalsIgnoreCase( "where" ) )
             {
                // ERRORS IF QUERYBASE DOES NOT HAVE A WHERE STATEMENT
                // AND EXITS FUNCTION
                System.out.println("*** Error: QueryBase query must have
                   a \"WHERE\" statement. ***");
                markError(i, array, line.length(), tag.length() );
                return(null);
             }

//*************************************************************************
             // RETRIEVES THE KEYWORDS FROM THE WHERE STATEMENT
             boolean keywords_present = false;
             int num_of_parenthesis = 0;
             boolean more_conditions = true;
             tag = "";
             i++;
             // LOOPS UNTIL A WHITESPACE, NEW LINE CHARACTER, RETURN
```

```
// CHARACTER OR END OF STATEMENT IS REACHED
while ( more_conditions )
{
   // SINCE THE KEYWORDS ARE IN QUOTES, THIS SEGEMENT OF
   // CODE WILL EXTRACT THE KEYWORDS USING THE COMMAS AND
   // QUOTES AS GUIDES
   keywords_present = true;

   // MAKES SURE THAT THE CONDITION STATEMENT IS NOT CUT
   // SHORT BY A INCOMPLETE CONDITION STATEMENT
   if ( i >= line.length() ) {
         // ERRORS IF WHERE FORMAT IS NOT CORRECT AND EXITS
         // FUNCTION
         System.out.println("*** Error: Invalid condition
            statement format in \"WHERE\" statement.");
         markError(i, array, line.length(), 0 );
         return(null);
   }

   // TESTS IF THERE ARE PARENTHESIS
   if ( array[i] == '(' ) {
      i++;
      num_of_parenthesis++;
      condition_list.add( "(" );
   }
   else {
      while ( array[i] != ' ' ) {
         tag = tag + array[i];
         i++;
      }

      //out.println("Another Tag: " + tag + "<br>");
      // TESTS TO SEE IF THE TAG IS A VALID TAG
      if ( Arrays.binarySearch(valid_tag, tag) >= 0 ) {
         condition_list.add( tag );
         req_tags_list.add( tag );
         tag = "";
      }
      else {
         // ERRORS IF WHERE FORMAT IS NOT CORRECT AND EXITS
         // FUNCTION
         System.out.println("*** Error: Invalid format in
            \"WHERE\" statement.  Invalid TAG format in
            conditional statement.");
         markError(i, array, line.length(), 0 );
         return(null);
      }

      // THE NEXT CHARACTER SHOULD BE AN EQUALS SIGN.  TIME
      // TO MAKE SURE.
      i++;
      if ( array[i] != '=' ) {
         // ERRORS IF WHERE FORMAT IS NOT CORRECT AND EXITS
         // FUNCTION
         System.out.println("*** Error: Invalid format in
            \"WHERE\" statement.  Expecting an EQUAL sign
            in conditional statement.");
```

```
            markError(i, array, line.length(), 0 );
            return(null);
        }

        // THE NEXT CHARACTER AFTER EQUALS SIGN SHOULD BE A
        // SPACE.  TIME TO MAKE SURE.
        i++;
        if ( array[i] != ' ' ) {
            // ERRORS IF WHERE FORMAT IS NOT CORRECT AND EXITS
            // FUNCTION
            System.out.println("*** Error: Invalid format in
                \"WHERE\" statement.  Expecting an SPACE in
                conditional statement after EQUAL sign.");
            markError(i, array, line.length(), 0 );
            return(null);
        }

        // THE NEXT STRING SHOULD BE KEYWORDS ENCLOSED IN
        // DOUBLE QUOTES.  THIS WILL EXTRACT KEYWORDS.
        i++;
        if ( array[i] == '"' )
        {
            i++;
            while ( array[i] != '"' )
            {
                if ( array[i] == ' ' || array[i] == '\n' ||
                        array[i] == '\r' )
                {
                    // ERRORS IF WHERE FORMAT IS NOT CORRECT AND
                    // EXITS FUNCTION
                    System.out.println("*** Error: Invalid
                            format in \"WHERE\" statement.  No
                            whitespaces are allowed in
                            keyword.");
                    markError(i, array, line.length(), 0 );
                    return(null);
                }

                tag = tag + array[i];
                i++;
            }

            // ENSURES THAT THE TAG IS NOT A LENGTH OF ZERO
            // (IE. THERE IS SOMETHING ACUTALLY IN THE TAG
            if ( tag.length() != 0 )
            {
                // MAKES SURE THE USER CAN NOT SEARCH ON
                // KEYWORD "AND" OR "OR"
                if ( tag.compareTo("and") == 0 ||
                        tag.compareTo("or") == 0 ) {
                    // ERRORS IF WHERE FORMAT IS NOT CORRECT AND
                    // EXITS FUNCTION
                    System.out.println("*** Error: Detected an
                            \"and\" or an \"or\" in the where
                            clauses.");
                    markError(i, array, line.length(), 0 );
                    return(null);
```

```
                    }
                    else {
                        i++;
                        condition_list.add( tag );
                        tag = "";
                        keywords_present = true;
                    }
                }
                else {
                        // ERRORS IF WHERE FORMAT IS NOT CORRECT AND
                        // EXITS FUNCTION
                        System.out.println("*** Error: Invalid
                            format in \"WHERE\" statement.  Nothing
                            in keyword.");
                        markError(i, array, line.length(), 0 );
                        return(null);
                }
            }
            else {
                // ERRORS IF WHERE FORMAT IS NOT CORRECT AND EXITS
                // FUNCTION
                System.out.println("*** Error: Invalid format in
                    \"WHERE\" statement.  Keyword needs to be in
                    double quotes.");
                markError(i, array, line.length(), 0 );
                return(null);
            }

            // TESTS END OF LINE AND IF THERE IS AN EXTRA
            // PARENTHESIS
            if ( i < line.length() && array[i] == ')' ) {
                // MAKES SURE THAT THEIR ARE MATCHING PARENTHESIS
                // IN THE WHERE STATEMENT
                if ( --num_of_parenthesis == -1 ) {
                    // ERRORS IF WHERE FORMAT IS NOT CORRECT AND
                    // EXITS FUNCTION
                    System.out.println("*** Error: Invalid format
                            in \"WHERE\" statement.  Unexpected
                            parenthesis.");
                    markError(i, array, line.length(), 0 );
                    return(null);
                }
                condition_list.add( ")" );
                i++;
            }

            i++;
            for (; i < line.length() && Character.isWhitespace(
array[i] ); i++ );

            // CHECKS IF THE QUERY STATEMENT ENDS AFTER THE FIRST
            // CHARACTER.
            if ( i == line.length() ) {
                // ERRORS IF WHERE FORMAT IS NOT CORRECT AND EXITS
                // FUNCTION
                System.out.println("*** Error 1: QueryBase query
                    must have a \"DEPTH\" clause. ***");
```

```
                    markError(i, array, line.length(), 0 );
                    return(null);
                }

                for (; i < line.length() &&
                      !Character.isWhitespace(array[i]); i++ )
                    tag = tag + array[i];

                if ( i == line.length() ) {
                    // ERRORS IF WHERE FORMAT IS NOT CORRECT AND EXITS
                    // FUNCTION
                    System.out.println("*** Error 3: QueryBase query
                       must have a \"DEPTH\" clause. ***");
                    markError(i, array, line.length(), 0 );
                    return(null);
                }

                //out.println("Condition Tag: " + tag + "<br>");
                // MAKES SURE THAT THEIR IS AN "AND" OR AN "OR AFTER
                // KEYWORD/TAG PAIR
                if ( tag.compareTo("and") == 0 ||
                      tag.compareTo("or") == 0 ) {
                    condition_list.add(tag);

                    // CLEARS TAG VARIABLE AND INCREMENTS ARRAY
                    // POSITION i TO NEXT POSITION
                    tag = "";
                    i++;
                }
                else if ( tag.compareTo("depth") == 0 )
                    more_conditions = false;
                else {
                    // ERRORS IF WHERE FORMAT IS NOT CORRECT AND EXITS
                    // FUNCTION
                    System.out.println("*** Error: Invalid format in
                       \"WHERE\" statement.  Unexpected word. ***");
                    markError(i, array, line.length(), 0 );
                    return(null);
                }
            }
        }

        if ( num_of_parenthesis != 0 ) {
            // ERRORS IF WHERE FORMAT IS NOT CORRECT AND EXITS
            // FUNCTION
            System.out.println("*** Error: Invalid format in
               \"WHERE\" statement.  There is a mismatched
               parenthesis. ***");
            markError(i, array, line.length(), 0 );
            return(null);
        }

//*********************************************************************
        // IF NO KEYWORDS WERE FOUND, THE PARSER WILL ERROR AND
        // EXIT FUNCTION
        if ( keywords_present == false ) {
            System.out.println("*** Error: \"WHERE\" clause present
```

```
                without keywords. ***");
            markError(i, array, line.length(), 0 );
            return(null);
         }

         // IF THE SELECT STATEMENT ENDS BEFORE THE DEPTH OF CLAUSE,
         // THIS WILL CATCH IT
         if ( i == line.length() ) {
            System.out.println("*** Error: QueryBase query must have
               a \"DEPTH\" clause. ***");
            markError(i-1, array, line.length(), 0 );
            return(null);
         }
      }

//*********************************************************************
      // THIS CODE SEGEMENT WILL TRY TO EXTRACT THE INTEGER FOR THE
      // DEPTH OF STATEMENT
      tag = "";
      i++;
      while ( array[i] != ' ' && array[i] != '\n'
         && array[i] != ';' )
      {
         tag = tag + array[i];
         i++;
      }

      try
      {
         // MAES SURE THE DEPTH OF NUMBER IS A INTEGER
         max_depth_level = Integer.parseInt( tag );
         //out.println("Depth Level: " + max_depth_level );

         if ( max_depth_level < 0 ) {
            // ERROR SIF DEPTH OF NUMBER IS NOT INTEGER AND EXITS
            // FUCNTION
            System.out.println("*** Error: \"DEPTH\" cannot have
               negative number. ***");
            markError(i, array, line.length(), tag.length() );
            return(null);
         }
      }
      catch (NumberFormatException e)
      {
         // ERRORS IF DEPTH OF NUMBER IS NOT INTEGER AND EXITS
         // FUCNTION
         System.out.println("*** Error: \"DEPTH\" statement does not
         // have integer value. ***");
         markError(i, array, line.length(), tag.length() );
         return(null);
      }
      i++;

//*********************************************************************
      // IF THE SELECT STATEMENT ENDS BEFORE THE DEPTH OF CLAUSE,
      // THIS WILL CATCH IT
      if ( i == line.length() ) {
```

```
   System.out.println("*** Error: Expecting a \"THREAD\"
         clause. ***");
   markError(i-1, array, line.length(), 0 );
   return(null);
}

//System.out.println(i + " Length:" + line.length() );

// THIS CODE SEGEMENT WILL TRY TO EXTRACT THE THREAD CLAUSE OF
// STATEMENT
tag = "";
while (array[i] != ' ' && array[i] != '\n' && array[i] != ';')
{
   tag = tag + array[i];
   i++;
}

if ( tag.compareTo("thread") != 0 ) {
   // PRINTS ERROR IF WORD IS NOT A VALID TAG AND EXITS
   // FUNCTION
   System.out.println("*** Last Error: Expecting THREAD clause
         ***");

   if ( array[i] == '\n' )
      markError(i-1, array, line.length(), tag.length() );
   else
      markError(i, array, line.length(), tag.length() );
   return(null);
}
i++;

//System.out.println(tag);

// THIS CODE SEGEMENT WILL TRY TO EXTRACT THE INTEGER FOR THE
// THREAD STATEMENT
tag = "";
while (array[i] != ' ' && array[i] != '\n' && array[i] != ';')
{
   tag = tag + array[i];
   i++;
}

try
{
   // MAKES SURE THE THREAD NUMBER IS A INTEGER
   thread_count = Integer.parseInt( tag );
   //out.println("Depth Level: " + max_depth_level );

   if ( thread_count < 0 ) {
      // ERROR SIF DEPTH OF NUMBER IS NOT INTEGER AND EXITS
      // FUCNTION
      System.out.println("*** Error: \"THREAD\" cannot have
         negative number. ***");
      markError(i, array, line.length(), tag.length() );
      return(null);
   }
}
```

```
            catch (NumberFormatException e)
            {
               // ERRORS IF DEPTH OF NUMBER IS NOT INTEGER AND EXITS
               // FUCNTION
               System.out.println("*** Error: \"THREAD\" statement does
                     not have integer value. ***");
               markError(i, array, line.length(), tag.length() );
               return(null);
            }

//*************************************************************************

            // RETURNS ZERO TO CALLING FUNCTION FOR OK STATUS OF SQL
            // PARSER
            ParsedStatementNode psn = new ParsedStatementNode();

            psn.condition_list = condition_list;
            psn.primer_urls = primer_urls;
            psn.req_tags_list = req_tags_list;
            psn.max_depth_level = max_depth_level;
            psn.search_mode = search_mode;
            psn.pruning_mode = pruning_mode;
            psn.thread_count = thread_count;

            return psn;
/*       }
         catch (ArrayIndexOutOfBoundsException e) {
         catch(IOException e) {
            out.println("*** Error: The required \"FROM\", \"WHERE\", or
               \"DEPTH\" clauses are missing from statement. ***");
            markError(i, array, line.length(), tag.length() );
            return(null);
         }*/
      }

      // THE MARKERROR FUNCTION PRINTS OUT THE QUERYBASE STATEMENT AND
      // TRIES TO SHOW THE USE WHERE AN ERROR MIGHT BE IN THE LANGUAGE.
      public static void markError(int index, char[] array, int length,
          int word_length)
      {
         int line_index = 0;

         // PRINTS OUT THE FIRST LINE UNTIL ERROR
         for (int i = 0; i < length; i++)
         {
            if ( i == index )
            {
               while( i < length && array[i] != '\n' )
               {
                  System.out.print( array[i] );
                  i++;
               }

               // PRINTS POINTER TO THE ERROR
               System.out.println("");
               for (int j = 0; j < line_index - word_length; j++)
                  System.out.print(" ");
```

```
            System.out.print("^");
         }

         // PRINTS OUT THE REST OF THE STATEMENT
         if ( i < length )
         {
            if ( array[i] == '\n' )
            {
               System.out.println("");
               line_index = -1;
            }

            System.out.print( array[i] );
            line_index++;
         }
      }
   }
}
```

Node.java

```
/**************************************************************************
*
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

Node.java is a houses the keyword and its score as QueryBase transfers

its content in the application
**************************************************************************
/

public class Node{

   public String content;

   public int occurance;


   public Node( )

   {

      content = "";

      occurance = 0;

   }


   // TEST TO SEE IF THE CONTENT IS EQUALED

   public boolean equals( String word )

   {

      if ( content.equalsIgnoreCase( word ) )

         return true;

      else

         return false;

   }
```

```
    // INCREMENTS THE OCCURRANCE OF THE CONTENT

    public void increment()
    {
        occurance++;
    }
}
```

ParsedHtmlNode.java

```
/**************************************************************************
*
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

PARSEDHTMLNODE IS A CLASS THAT HOUSES THE PARSED KEYWORD AS TREES
UNTIL THEY ARE CHECKED BY THE MATCHING THREADS.  THIS CLASS ALSO
HOUSES THE URL ADDRESS AND THE TIME IT TOOK TO PARSE THE SITE.
**************************************************************************
/

import tree.*;
import java.util.*;

public class ParsedHtmlNode{
   // HTML TAG TREES
   public AATree BodyTree;
   public AATree TitleTree;
   public AATree HeaderTree;
   public AATree EmTree;
   public AATree BoldTree;
   public AATree MetaTree;
   public AATree ItalicTree;

   // URL ADDRESS
   public String url;
   // PARSE TIME
   public double p_time;
   // PARENT URL ADDRESS
   public String origin;
   // CHILD LINKS FROM PAGE
   public LinkedList hyperlinks;
   // DEPTH OF PARENT
   public int parent_depth_level;

   public ParsedHtmlNode() {
   }
}
```

ParsedStatementNode.java

```
/*************************************************************************
*
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

THIS CLASS ACTS A NODE FOR STORING URLS AND THEIR PARSE TIMES AFTER
THEY HAVE BEEN MATCHED TO THE QUERY STATEMENT
*************************************************************************
/

import java.util.LinkedList;

public class ParsedStatementNode{
   // LIST OF CONDITIONS
   public LinkedList condition_list;
   // LIST OF PRIMER URLS
   public LinkedList primer_urls;
   // NOT USED
   public LinkedList req_tags_list;
   // MAX DEPTH LEVEL
   public int max_depth_level;
   // GIVES THE SEARCH MODE OF THE SPIDER
   public String search_mode;
   // GIVES THE PRUNING MODE
   public boolean pruning_mode;
   // GIVES THE NUMBER OF THREADS
   public int thread_count;
}
```

Stats.java

```
/**************************************************************************
*
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

STATS.JAVA TRACKS THE STATS FOR THE INDIVIDUAL THREADS AS OBJECTS
**************************************************************************
/

import java.util.*;
import java.io.*;

public class Stats
{
   int total_count;
   double total_time;
   int db_download;
   int db_no_download;
   int web_download;
   double db_download_time;
   double db_no_download_time;
   double web_download_time;
   int links;
   int redirect;
   LinkedList ParsedCountPerThread = new LinkedList();

   public Stats () {
      total_count = 0;
      total_time = 0;
      db_download = 0;
      db_no_download = 0;
      web_download = 0;
      db_download_time = 0;
      db_no_download_time = 0;
      web_download_time = 0;
   }

   // RETURNS THE TOTAL NUMBER OF PAGES PROCESSED
   public int getCount() {
      return total_count;
   }

   // GETS THE TOTAL TIME FOR THE PAGES PARSED
   public double getTime() {
      return total_time;
   }

   // GETS THE NUMBER OF PAGES FROM WEB DURING CACHE MODE
   public int getCachedWebViewCount() {
      return db_download;
```

```
}

// GETS THE NUMBER OF PAGES CACHED
public int getCachedViewCount() {
   return db_no_download;
}

// GETES THE NUMBER OF PAGE FROM WEB
public int getWebViewCount() {
   return web_download;
}

// GETS THE TIME IT TOOK TO PROCESS PAGES FROM WEB DURING CACHE
public double getCachedWebViewTime() {
   return db_download_time;
}

// GETS THE TIME IT TOOK TO PROCESS PAGES FROM CACHE
public double getCachedViewTime() {
   return db_no_download_time;
}

// GETS THE TIME IT TOOK TO PROCESS PAGES FROM WEB
public double getWebViewTime() {
   return web_download_time;
}

// GETS THE THREAD STATS
public Node getThreadStats() {
   return (Node) ParsedCountPerThread.removeFirst();
}

// GETS THE NUMBER OF LINKS
public int getLinkCount() {
   return links;
}

// GETS THE NUMBER OF REDIRECTS
public int getRedirectCount() {
   return redirect;
}

// CHECKS TO SEE IF THE STATS QUEUE IS EMPTY
public boolean isThreadStatsEmpty() {
   if ( ParsedCountPerThread.size() > 0 )
      return false;
   else
      return true;
}

// INSERTS STAT INFORMATION FROM ALL THREADS INTO QUEUE SO THE
// MAIN PROGRAM AND DISPLAY STATS
synchronized public void insert( String name, int thread_count,
         double thread_time, int db_download, int db_no_download,
         int web_download, double db_download_time,
         double db_no_download_time, double web_download_time,
         int links, int redirect ) {
```

```
        total_count = total_count + thread_count;
        total_time = total_time + thread_time;
        this.db_download += db_download;
        this.db_no_download += db_no_download;
        this.web_download += web_download;
        this.db_download_time += db_download_time;
        this.db_no_download_time += db_no_download_time;
        this.web_download_time += web_download_time;
        this.links += links;
        this.redirect += redirect;

        Node node = new Node();
        node.content = name;
        node.occurance = thread_count;
        ParsedCountPerThread.add(node);
    }
}
```

SynchQueue.java

```
/**************************************************************************
*
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

SYNCHQUEUE IS A QUEUE CLASS THAT IS DESIGNED TO BE THREAD SAFE.
**************************************************************************
/

import java.util.*;
import java.io.*;

public class SynchQueue extends LinkedList
{
   private Object node;
   private TraversalMethods traversal_method;
   boolean complete;

   public SynchQueue ( TraversalMethods traversal_method ) {
      this.traversal_method = traversal_method;
      complete = false;
   }

   // ALLOWS THE CALLING OBJECT TO RETRIEVE THE LINKED LISTS CONTENT
   synchronized public Object getNode() {
      // IF THE QUEUE IS BEING USED OR THE SIZE OF THE QUEUE IS
      // ZERO, THEN THE ENTERING THREAD IS PUT ON HOLD
      while ( this.isEmpty() ) {
         try {
            wait(1, 0);
            if (this.isEmpty() && traversal_method.threadsWaiting()) {
               complete = true;
               break;
            }
         }
         catch ( InterruptedException e ) {}
      }

        // REMOVES NODE FROM QUEUE
        if ( !this.isEmpty() )
           node = this.removeFirst();
        else
           node = null;

        // NOTIFIES WAITING THREADS
        notifyAll();

        // RETURNS NODE TO CALL OBJECT
        return node;
   }
```

```
   // ALLOWS THE CALLING OBJECT TO ADD CONTENT TO THE LINKED LIST
   synchronized public void addNode( Object node )
   {
      // ADDS NODE TO LIST
      this.addLast( node );

      // NOTIFIES WAITING THREADS
      notifyAll();
   }

   // CHECKS IF THE QUEUE IS EMPTY
   public boolean isEmpty() {
       if ( this.size() == 0 )
           return true;
       else
           return false;
   }

   // CHECKS IF THE PROCESS IS COMPLETED
   public boolean isComplete() {
      return complete;
   }

}
```

TimeNode.java

```
/*********************************************************************
*
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

TIMENODE IS A CLASS THAT HOUSES THE TOTAL EXECUTION TIME AND NUMBER

OF PAGES PARSED BY QUERYBASE DURING THE SEARCH SESSION.  THIS IS

ONLY USED TO PASS THE TIME AND COUNT FROM THE MAIN THREAD TO THE

HTML THREADS AND BACK.
*********************************************************************
/


public class TimeNode{

   public double execution = 0;

   public int page_count = 0;


   public TimeNode() {

   }

}
```

MatchedNode.java

```
/**************************************************************************
*
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

THIS CLASS ACTS A NODE FOR STORING URLS AND THEIR PARSE TIMES AFTER

THEY HAVE BEEN MATCHED TO THE QUERY STATEMENT
**************************************************************************
/

public class MatchedNode{

   // URL ADDRESS
   public String url;

   // PARSING TIME
   public double p_time;
   // QUERY PROCESSING TIME

   public double m_time;
   // PARENT URL ADDRESS

   public String origin;



   public MatchedNode() {

   }

}
```

AATree.java

```
/**************************************************************************
*
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

THIS IS AN OBJECT OF THE AATREE.

THIS CODE WAS OBTAIN FROM THE FOLLOWING RESOURCE.  I AM CREDITING
THE AUTHOR WITH THIS CODE.

Weiss, M. A. (March 1999).  Data Structures & Problem Solving Using
Java.
Addison-Wesley Longman, Inc: Massachusetts.  530-537.

NOTE: I DID MAKE A COUPLE OF CHANGES AT THE END OF THIS CODE.
**************************************************************************
/

// AATree class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// Exceptions are thrown by insert and remove if warranted

/**
 * Implements an AA-tree.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
package tree;

public class AATree
{
    /**
     * Construct the tree.
     */
    public AATree( )
    {
        root = nullNode;
    }
```

```
/**
 * Insert into the tree.
 * @param x the item to insert.
 * @throws DuplicateItemException if x is already present.
 */
synchronized public void insert( Comparable x )
{
    try {
       root = insert( x, root );
    }
    catch (DuplicateItemException e) {
    }
}

/**
 * Remove from the tree.
 * @param x the item to remove.
 * @throws ItemNotFoundException if x is not found.
 */
public void remove( Comparable x )
{
    deletedNode = nullNode;
    root = remove( x, root );
}

/**
 * Find the smallest item in the tree.
 * @return the smallest item or null if empty.
 */
public Comparable findMin( )
{
    if( isEmpty( ) )
        return null;

    AANode ptr = root;

    while( ptr.left != nullNode )
        ptr = ptr.left;

    return ptr.element;
}

/**
 * Find the largest item in the tree.
 * @return the largest item or null if empty.
 */
public Comparable findMax( )
{
    if( isEmpty( ) )
        return null;

    AANode ptr = root;

    while( ptr.right != nullNode )
        ptr = ptr.right;

    return ptr.element;
```

```
}

/**
 * Find an item in the tree.
 * @param x the item to search for.
 * @return the matching item of null if not found.
 */
synchronized public Comparable find( Comparable x )
{
    AANode current = root;
    nullNode.element = x;

    for( ; ; )
    {
        if( x.compareTo( current.element ) < 0 )
            current = current.left;
        else if( x.compareTo( current.element ) > 0 )
            current = current.right;
        else if( current != nullNode )
            return current.element;
        else
            return null;
    }
}

/**
 * Make the tree logically empty.
 */
public void makeEmpty( )
{
    root = nullNode;
}

/**
 * Test if the tree is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    return root == nullNode;
}

/**
 * Internal method to insert into a subtree.
 * @param x the item to insert.
 * @param t the node that roots the tree.
 * @return the new root.
 * @throws DuplicateItemException if x is already present.
 */
private AANode insert( Comparable x, AANode t )
{
    if( t == nullNode )
        t = new AANode( x );
    else if( x.compareTo( t.element ) < 0 )
        t.left = insert( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = insert( x, t.right );
```

```
        else
            throw new DuplicateItemException( x.toString( ) );

        t = skew( t );
        t = split( t );
        return t;
    }

    /**
     * Internal method to remove from a subtree.
     * @param x the item to remove.
     * @param t the node that roots the tree.
     * @return the new root.
     * @throws ItemNotFoundException if x is not found.
     */
    private AANode remove( Comparable x, AANode t )
    {
        if( t != nullNode )
        {
            // Step 1: Search down the tree and set lastNode and
            // deletedNode
            lastNode = t;
            if( x.compareTo( t.element ) < 0 )
                t.left = remove( x, t.left );
            else
            {
                deletedNode = t;
                t.right = remove( x, t.right );
            }

            // Step 2: If at the bottom of the tree and
            //         x is present, we remove it
            if( t == lastNode )
            {
                if( deletedNode == nullNode ||
                  x.compareTo( deletedNode.element ) != 0 )
                    throw new ItemNotFoundException( x.toString( ) );
                deletedNode.element = t.element;
                t = t.right;
            }

            // Step 3: Otherwise, we are not at the bottom; rebalance
            else
                if( t.left.level < t.level - 1 ||
                  t.right.level < t.level - 1 )
                {
                    if( t.right.level > --t.level )
                        t.right.level = t.level;
                    t = skew( t );
                    t.right = skew( t.right );
                    t.right.right = skew( t.right.right );
                    t = split( t );
                    t.right = split( t.right );
                }
        }
        return t;
    }
```

```
/**
 * Skew primitive for AA-trees.
 * @param t the node that roots the tree.
 * @return the new root after the rotation.
 */
private static AANode skew( AANode t )
{
    if( t.left.level == t.level )
        t = rotateWithLeftChild( t );
    return t;
}

/**
 * Split primitive for AA-trees.
 * @param t the node that roots the tree.
 * @return the new root after the rotation.
 */
private static AANode split( AANode t )
{
    if( t.right.right.level == t.level )
    {
        t = rotateWithRightChild( t );
        t.level++;
    }
    return t;
}

/**
 * Rotate binary tree node with left child.
 */
private static AANode rotateWithLeftChild( AANode k2 )
{
    AANode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
}

/**
 * Rotate binary tree node with right child.
 */
private static AANode rotateWithRightChild( AANode k1 )
{
    AANode k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    return k2;
}

private static class AANode
{
        // Constructors
    AANode( Comparable theElement )
    {
        element = theElement;
        left    = right = nullNode;
```

```
            level   = 1;
        }

        Comparable element;      // The data in the node
        AANode     left;         // Left child
        AANode     right;        // Right child
        // THIS LEVEL NUMBER WAS ADDED BY ME, NOT THE AUTHOR
        int        level;        // Level
    }

    private AANode root;
    private static AANode nullNode;

    static         // static initializer for nullNode
    {
        nullNode = new AANode( null );
        nullNode.left = nullNode.right = nullNode;
        // THIS LEVEL NUMBER WAS ADDED BY ME, NOT THE AUTHOR
        nullNode.level = 0;
    }

    private static AANode deletedNode;
    private static AANode lastNode;

    public void traverseTree() {
       traverse(root);
    }

    private void traverse(AANode current) {

       if ( current != nullNode ) {
          traverse(current.left);
          System.out.println(current.element);
          traverse(current.right);
       }
    }
}
```

DuplicateItemException.java

```
/**************************************************************************
*
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

THIS CODE WAS OBTAIN FROM THE FOLLOWING RESOURCE.  I AM CREDITING

THE AUTHOR WITH THIS CODE.


Weiss, M. A. (March 1999).  Data Structures & Problem Solving Using
Java.

Addison-Wesley Longman, Inc: Massachusetts.  530-537.

**************************************************************************
/


/**
 * Exception class for duplicate item errors
 * in search tree insertions.
 * @author Mark Allen Weiss
 */
package tree;


public class DuplicateItemException extends RuntimeException
{
    /**
     * Construct this exception object.
     */
    public DuplicateItemException( )
    {
        super( );
```

```
    }
    /**
     * Construct this exception object.
     * @param message the error message.
     */
    public DuplicateItemException( String message )
    {
        super( message );
    }
}
```

ItemNotFoundException.java

```
/**************************************************************************
*
Name: Burr Watters
Date: 12/1/2004
Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-TIME
WEB SEARCH ENGINE
Advisor: Dr. Seyed-Abbassi - University of North Florida

THIS CODE WAS OBTAIN FROM THE FOLLOWING RESOURCE.  I AM CREDITING

THE AUTHOR WITH THIS CODE.


Weiss, M. A. (March 1999).  Data Structures & Problem Solving Using
Java.

Addison-Wesley Longman, Inc: Massachusetts.  530-537.

**************************************************************************
/


/**
 * Exception class for duplicate item errors
 * in search tree insertions.
 * @author Mark Allen Weiss
 */
package tree;


public class ItemNotFoundException extends RuntimeException
{
    /**
     * Construct this exception object.
     */
    public ItemNotFoundException( )
    {
```

```
        super( );
    }
    /**
     * Construct this exception object.
     * @param message the error message.
     */
    public ItemNotFoundException( String message )
    {
        super( message );
    }
}
```

SQL Table Defintions for Cache Database

```
--*********************************************************************
-- Name: Burr Watters
-- Date: 12/1/2004
-- Thesis Project: DEVELOPMENT AND PERFORMANCE EVALUATION OF A REAL-
-- TIME
-- WEB SEARCH ENGINE
-- Advisor: Dr. Seyed-Abbassi - University of North Florida

-- SQL SOURCE CODE FOR THE MYSQL DATABASE.  SETS UP THE DATABASE FOR
-- CACHING.
--*********************************************************************

use watters;

drop table html_parse_table;
drop table url_time_table;

create table html_parse_table (
url     CHAR(255)  NOT NULL,
tag     CHAR(7)    NOT NULL,
keyword CHAR(238)   NOT NULL,
PRIMARY KEY ( url, tag, keyword ) );

create table url_time_table (
url     CHAR(255)  NOT NULL,
time    BIGINT     NOT NULL,
PRIMARY KEY ( url ) );
```

Vita

Burr S. Watters IV has a Bachelor of Science degree in Computer and Information Sciences from the University of North Florida, December 2001. He expects to receive a Master of Science in Information Systems from the University of North Florida, December 2004. Dr. Behrooz Seyed-Abbassi of the University of North Florida is Burr's thesis advisor. Burr is currently working for the Coggin College of Business at the University of North Florida as a Technology Coordinator. His duties range from server administration to technical support to technical staff management. He has been employed with the University for 7 years.

Burr has had experience working with both Windows and Linux in the fields of networking, software development, server administration, and application deployment. He is knowledgeable in programming languages as Java, PHP, C, Visual Basic, and ASP. He has an interest in developing web applications for electronic communication and business commerce. Burr currently resides in Jacksonville, Florida.