



Jaypee University of Information Technology
Solan (H.P.)

LEARNING RESOURCE CENTER

Acc. Num. *SP07013* Call Num:

General Guidelines:

- ◆ Library books should be used with great care.
- ◆ Tearing, folding, cutting of library books or making any marks on them is not permitted and shall lead to disciplinary action.
- ◆ Any defect noticed at the time of borrowing books must be brought to the library staff immediately. Otherwise the borrower may be required to replace the book by a new copy.
- ◆ The loss of LRC book(s) must be immediately brought to the notice of the Librarian in writing.

Learning Resource Centre-JUIT



SP07013

SEARCH ENGINE

Anshul Gautam **071237**

Dev Vrat Arya **071258**

Ishan Aggarwal **071297**

Under the Supervision of

Mr Yashwant Singh



MAY - 2011

Submitted in partial fulfillment

of the requirements for the degree of

**BACHELOR OF TECHNOLOGY
(CSE)**

**JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY
WAKNAGHAT
SOLAN , HIMACHAL PRADESH
INDIA
2011**



(I)

Table of Contents

CHAPTER NO.	TOPICS	PAGE NO
	CERTIFICATE FROM THE SUPERVISOR	II
	ACKNOWLEDGEMENT	III
	SUMMARY	IV
CHAPTER 1	Introduction	6
1.1	What is Search Engine	6
1.2	Need of Search Engine	7
1.3	Commercial Aspects	7
1.4	History	8
1.5	Working	10
1.6	Market Share	12
1.7	Search Engine Bias	13
CHAPTER 2	System Architecture	14
CHAPTER 3	Methodologies	21
3.1	Crawler	21
3.2	Indexer	30
3.3	Page Rank	35
CHAPTER 4	Sample Codes	44
4.1	Crawling	44
4.2	Indexing	49
4.3	Retrieval	56
RESULTS		58
LITERATURE AND REFERENCES		59

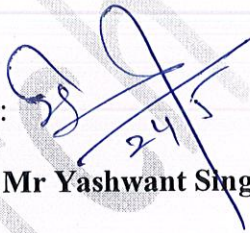
[2]

(II)

CERTIFICATE

This is to certify that the work titled *Search Engine* submitted by **Anshul Gautam, Dev Vrat Arya and Ishan Aggarwal** in partial fulfillment for the award of degree of Bachelor of Technology (CSE) of Jaypee University of Information Technology, Wazirpur has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma. .

Signature in full of Supervisor:



Name in Capital block letters: **Mr Yashwant Singh**

Designation: **Lecturer, CSE& IT**

Date:

(III)

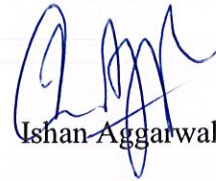
ACKNOWLEDGEMENT

While developing the project, we have learnt a lot. This will be an unforgettable experience.

While developing this project, a lot of difficulties were faced by us. But it was the help of some special people that we have gained much confidence and developed the project quite well. We shall like to thank everyone who in anyway helped us in this project.

We are thankful to our H.O.D(CSE & IT) (Retd) Brig. S. P. Ghrera and our project in charge Mr. Yashwant Singh. These are the people who helped us in providing the required infrastructure, good work culture, mad us learn a lot of new things and giving us the required guidance, which helped us in gathering the requirements.

Signature of Students:



Name of Students: Anshul Gautam

Dev Vrat Arya

Ishan Aggarwal

Enrollment No: 071237

071258

071297

Date:

(IV)

SUMMARY

The objective of the project was to design an automated Search Engine which searches the web for answer of the query by the user. There are billions of pages on the Web. And every minute of the day, folks are posting more. Search Engines helps us to find the specific information we need from these billions of pages.

A search engine can be basically divided into three parts each have its own function:

- a. Crawler-A crawler crawls the web for pages and stores them in a page dump which are further indexed.
- b. Indexer-All the web pages stored in the dump after crawling are read and data is stored in indexes so that it becomes easier to search when data is to be retrieved.
- c. Retrieval-According to the query of the user, data is searched from the stored indexes Along with this different ranking techniques are used so that user gets the best possible result.

Chapter 1. Introduction

1.1 What is a Search Engine

A program that searches documents for specified key words and returns a list of the documents where the keywords were found. Although *search engine* is really a general class of programs, the term is often used to specifically describe systems like Google, Alta Vista and Excite that enable users to search for documents on the World Wide Web and USENET newsgroups.

Typically, a search engine works by sending out a spider to fetch as many documents as possible. Another program, called an *indexer*, then reads these documents and creates an index based on the words contained in each document. Each search engine uses a proprietary algorithm to create its indices such that, ideally, only meaningful results are returned for each query.

Different Search Engine Approaches

- Major search engines such as Google, Yahoo (which uses Google), AltaVista, and Lycos index the content of a large portion of the Web and provide results that can run for pages - and consequently overwhelm the user.
- Specialized content search engines are selective about what part of the Web is crawled and indexed. For example, TechTarget sites for products such as the AS/400 (<http://www.search400.com>) and CRM applications (<http://www.searchCRM.com>) selectively index only the best sites about these products and provide a shorter but more focused list of results.
- Ask Jeeves (<http://www.ask.com>) provides a general search of the Web but allows you to enter a search request in natural language, such as "What's the weather in Seattle today?"
- Special tools and some major Web sites such as Yahoo .let you use a number of search engines at the same time and compile results for you in a single list.
- Individual Web sites, especially larger corporate sites, may use a search engine to index and retrieve the content of just their own site. Some of the major search engine companies license or sell their search engines for use on individual sites.

1.2 Need of Search Engine

For the same reason you need a card catalogue in a library. There is lots of great and useful information in a library, but it's physically impossible to examine all the books personally. Not even the most indefatigable web-surfer could hyperlink to all the documents in the aptly named World Wide Web.

Similarly there are billions of pages on the Web. And every minute of the day, folks are posting more. Search Engines helps us to find the specific information we need from these billions of pages.

The search engines and directories help you shift through all those 1's and 0's to find the specific information you need.

1.3 Commercial Aspects Of Search Engine

In the modern era as the number of internet users are increasing day by day so a Search Engine which provides service efficiently and fastly is commercially very profitable. More the users uses the Search Engine more money it will make for the company.

If we consider the example of Google Search Engine, when we search something on Google then on right hand side there are some advertisements and from these advertisements the Search Engine makes money.

1.4 HISTORY-

During the early development of the web, there was a list of web servers edited by Tim Berners-Lee and hosted on the CERN web server. One historical snapshot from 1992 remains. As more web servers went online the central list could not keep up. On the NCSA site new servers were announced under the title "What's New!"

The very first tool used for searching on the Internet was Archie. The name stands for "archive" without the "v." It was created in 1990 by Alan Emtage, Bill Heelan and J. Peter Deutsch, computer science students at McGill University in Montreal. The program downloaded the directory listings of all the files located on public anonymous FTP (File Transfer Protocol) sites, creating a searchable database of file names; however, Archie did not index the contents of these sites since the amount of data was so limited it could be readily searched manually.

The rise of Gopher (created in 1991 by Mark McCahill at the University of Minnesota) led to two new search programs, Veronica and Jughead. Like Archie, they searched the file names and titles stored in Gopher index systems. Veronica (*Very Easy Rodent-Oriented Net-wide Index to Computerized Archives*) provided a keyword search of most Gopher menu titles in the entire Gopher listings. Jughead (*Jonzy's Universal Gopher Hierarchy Excavation And Display*) was a tool for obtaining menu information from specific Gopher servers. While the name of the search engine "Archie" was not a reference to the Archie comic book series, "Veronica" and "Jughead" are characters in the series, thus referencing their predecessor.

In the summer of 1993, no search engine existed yet for the web, though numerous specialized catalogues were maintained by hand. Oscar Nierstrasz at the University of Geneva wrote a series of Perl scripts that would periodically mirror these pages and rewrite them into a standard format which formed the basis for W3Catalog, the web's first primitive search engine, released on September 2, 1993.

In June 1993, Matthew Gray, then at MIT, produced what was probably the first web robot, the Perl-based World Wide Web Wanderer, and used it to generate an index called 'Wandex'. The purpose of the Wanderer was to measure the size of the World Wide Web, which it did until late 1995. The web's second search engine Aliweb appeared in November 1993. Aliweb did not use a web robot, but instead depended on being notified by website administrators of the existence at each site of an index file in a particular format.

JumpStation (released in December 1993) used a web robot to find web pages and to build its index, and used a web form as the interface to its query program. It was thus the first WWW resource-discovery tool to combine the three essential features of a web search engine (crawling, indexing, and searching) as described below. Because of the limited resources available on the platform on which it ran, its indexing and hence searching were limited to the titles and headings found in the web pages the crawler encountered.

One of the first "full text" crawler-based search engines was WebCrawler, which came out in 1994. Unlike its predecessors, it let users search for any word in any webpage, which has become the standard for all major search engines since. It was also the first one to be widely

known by the public. Also in 1994, Lycos (which started at Carnegie Mellon University) was launched and became a major commercial endeavor.

Soon after, many search engines appeared and vied for popularity. These included Magellan, Excite, Infoseek, Inktomi, Northern Light, and AltaVista. Yahoo! was among the most popular ways for people to find web pages of interest, but its search function operated on its web directory, rather than full-text copies of web pages. Information seekers could also browse the directory instead of doing a keyword-based search.

In 1996, Netscape was looking to give a single search engine an exclusive deal to be their featured search engine. There was so much interest that instead a deal was struck with Netscape by five of the major search engines, where for \$5Million per year each search engine would be in a rotation on the Netscape search engine page. The five engines were Yahoo!, Magellan, Lycos, Infoseek, and Excite.

Search engines were also known as some of the brightest stars in the Internet investing frenzy that occurred in the late 1990s. Several companies entered the market spectacularly, receiving record gains during their initial public offerings. Some have taken down their public search engine, and are marketing enterprise-only editions, such as Northern Light. Many search engine companies were caught up in the dot-com bubble, a speculation-driven market boom that peaked in 1999 and ended in 2001.

Around 2000, the Google search engine rose to prominence. The company achieved better results for many searches with an innovation called PageRank. This iterative algorithm ranks web pages based on the number and PageRank of other web sites and pages that link there, on the premise that good or desirable pages are linked to more than others. Google also maintained a minimalist interface to its search engine. In contrast, many of its competitors embedded a search engine in a web portal.

By 2000, Yahoo was providing search services based on Inktomi's search engine. Yahoo! acquired Inktomi in 2002, and Overture (which owned AlltheWeb and AltaVista) in 2003. Yahoo! switched to Google's search engine until 2004, when it launched its own search engine based on the combined technologies of its acquisitions.

Microsoft first launched MSN Search in the fall of 1998 using search results from Inktomi. In early 1999 the site began to display listings from Looksmart blended with results from Inktomi except for a short time in 1999 when results from AltaVista were used instead. In 2004, Microsoft began a transition to its own search technology, powered by its own web crawler (called msnbot).

Microsoft's rebranded search engine, Bing, was launched on June 1, 2009. On July 29, 2009, Yahoo! and Microsoft finalized a deal in which Yahoo! Search would be powered by Microsoft Bing technology.

1.5 HOW DO A SEARCH ENGINE WORKS

Web search engines work by storing information about many web pages, which they retrieve from the html itself. These pages are retrieved by a Web crawler (sometimes also known as a spider) — an automated Web browser which follows every link on the site. Exclusions can be made by the use of robots.txt. The contents of each page are then analyzed to determine how it should be indexed (for example, words are extracted from the titles, headings, or special fields called meta tags). Data about web pages are stored in an index database for use in later queries. A query can be a single word. The purpose of an index is to allow information to be found as quickly as possible. Some search engines, such as Google, store all or part of the source page (referred to as a cache) as well as information about the web pages, whereas others, such as AltaVista, store every word of every page they find. This cached page always holds the actual search text since it is the one that was actually indexed, so it can be very useful when the content of the current page has been updated and the search terms are no longer in it. This problem might be considered to be a mild form of linkrot, and Google's handling of it increases usability by satisfying user expectations that the search terms will be on the returned webpage. This satisfies the principle of least astonishment since the user normally expects the search terms to be on the returned pages. Increased search relevance makes these cached pages very useful, even beyond the fact that they may contain data that may no longer be available elsewhere.

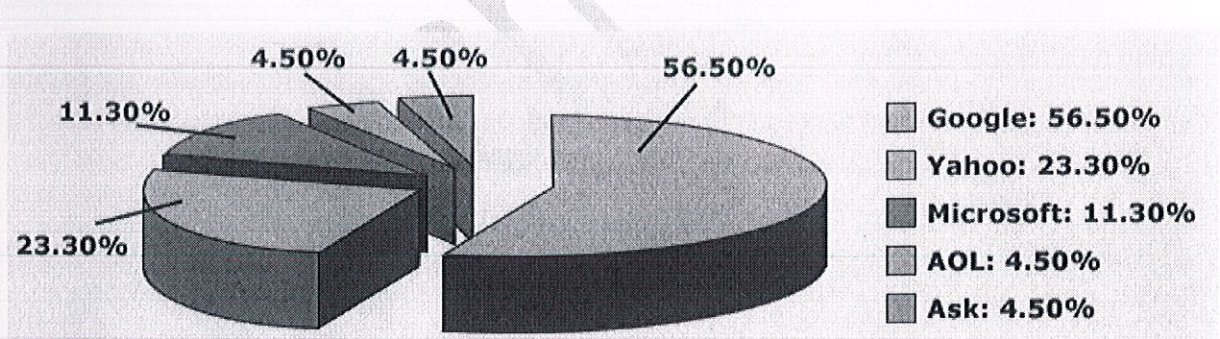
When a user enters a query into a search engine (typically by using key words), the engine examines its index and provides a listing of best-matching web pages according to its criteria, usually with a short summary containing the document's title and sometimes parts of the text. The index is built from the information stored with the data and the method by which the information is indexed. Unfortunately, there are currently no known public search engines that allow documents to be searched by date. Most search engines support the use of the boolean operators AND, OR and NOT to further specify the search query. Boolean operators are for literal searches that allow the user to refine and extend the terms of the search. The engine looks for the words or phrases exactly as entered. Some search engines provide an advanced feature called proximity search which allows users to define the distance between keywords. There is also concept-based searching where the research involves using statistical analysis on pages containing the words or phrases you search for. As well, natural language queries allow the user to type a question in the same form one would ask it to a human. A site like this would be ask.com.

The usefulness of a search engine depends on the relevance of the **result set** it gives back. While there may be millions of web pages that include a particular word or phrase, some pages may be more relevant, popular, or authoritative than others. Most search engines employ methods to rank the results to provide the "best" results first. How a search engine decides which pages are the best matches, and what order the results should be shown in, varies widely from one engine to another. The methods also change over time as Internet usage changes and new techniques evolve. There are two main types of search engine that have evolved: one is a system of predefined and hierarchically ordered keywords that humans have programmed extensively. The other is a system that generates an "inverted index" by analyzing texts it locates. This second form relies much more heavily on the computer itself to do the bulk of the work.

Most Web search engines are commercial ventures supported by advertising revenue and, as a result, some employ the practice of allowing advertisers to pay money to have their listings ranked higher in search results. Those search engines which do not accept money for their search engine results make money by running search related ads alongside the regular search engine results. The search engines make money every time someone clicks on one of these ads.

Search Engine

1.6 Market Share



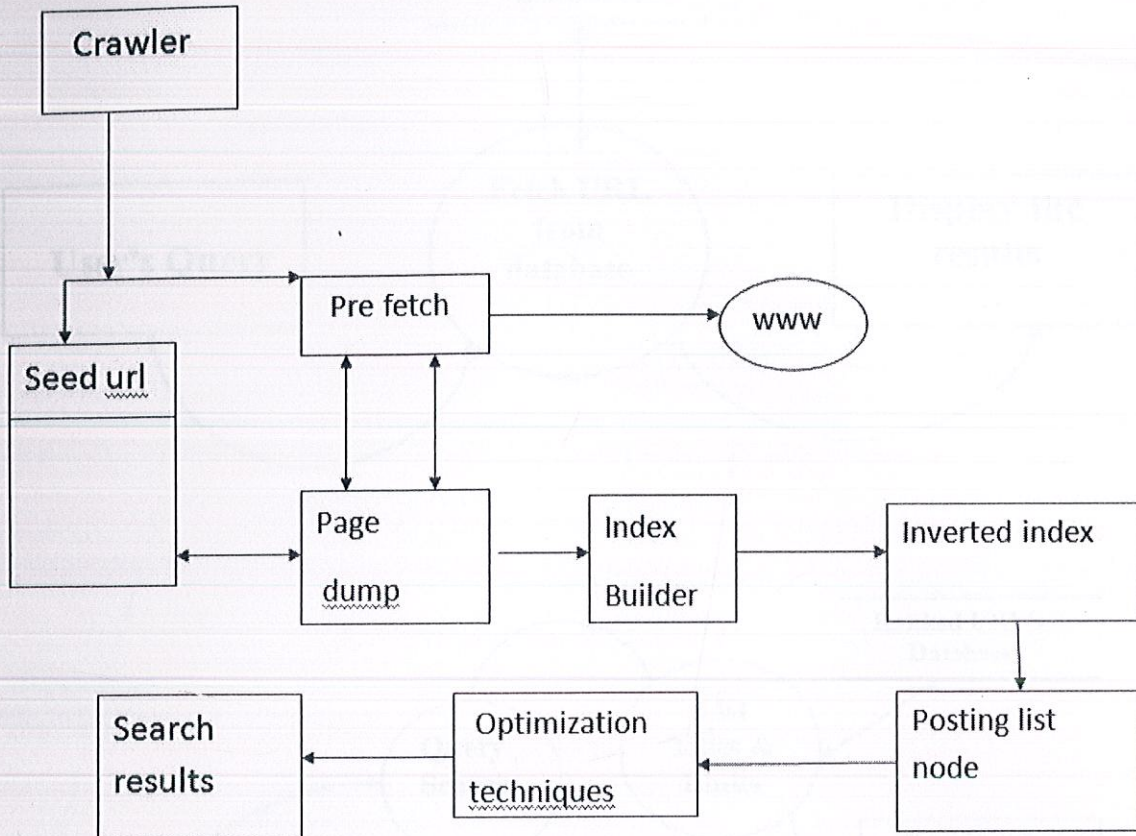
In the United States, Google held a 63.2% market share in May 2009, according to Nielsen NetRatings. In the People's Republic of China, Baidu held a 61.6% market share for web search in July 2009.

1.7 SEARCH ENGINE BIAS

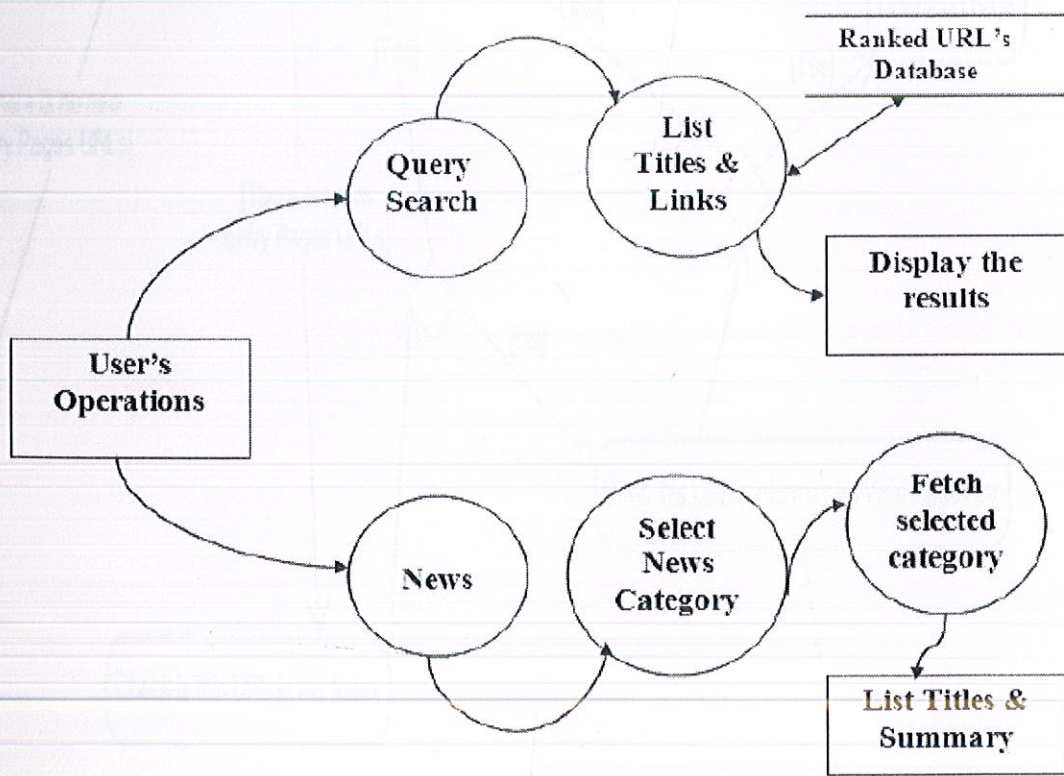
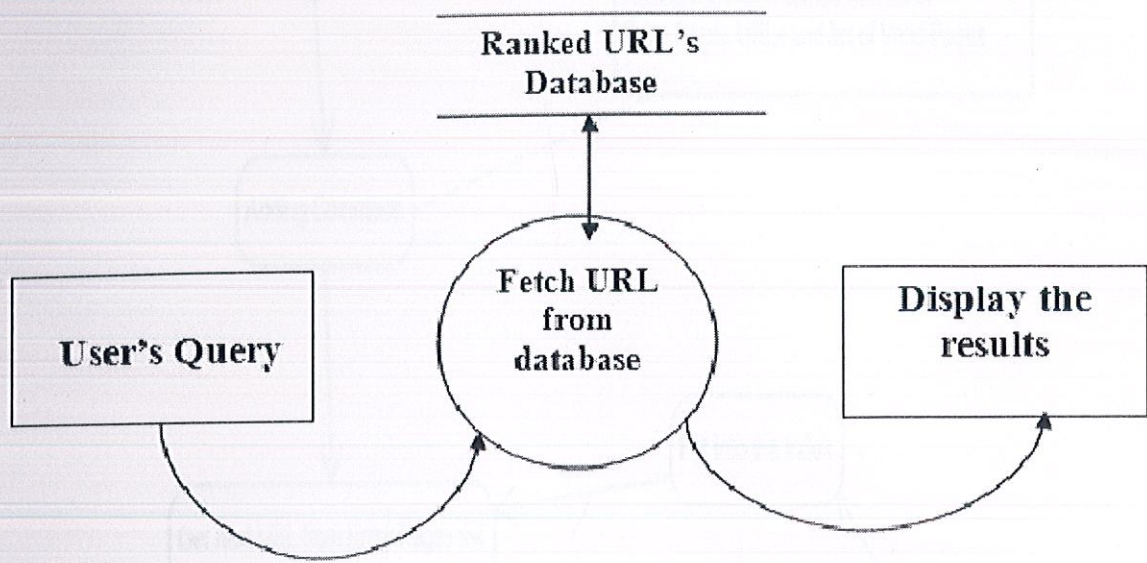
Although search engines are programmed to rank websites based on their popularity and relevancy, empirical studies indicate various political, economic, and social biases in the information they provide. These biases could be a direct result of economic and commercial processes (e.g., companies that advertise with a search engine can become also more popular in its organic search results), and political processes (e.g., the removal of search results in order to comply with local laws). Google Bombing is one example of an attempt to manipulate search results for political, social or commercial reasons.

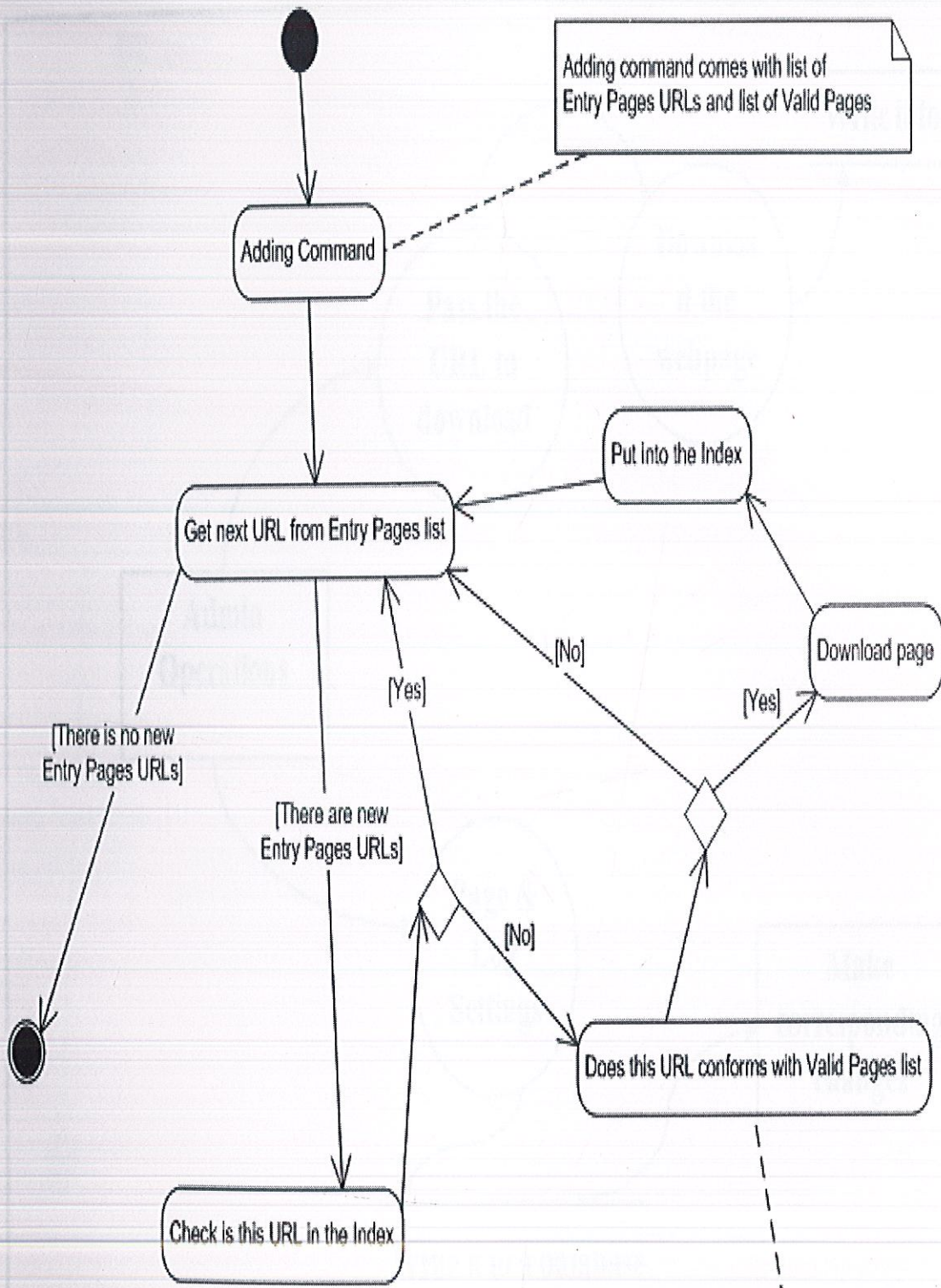
Chapter 2.

System Architecture



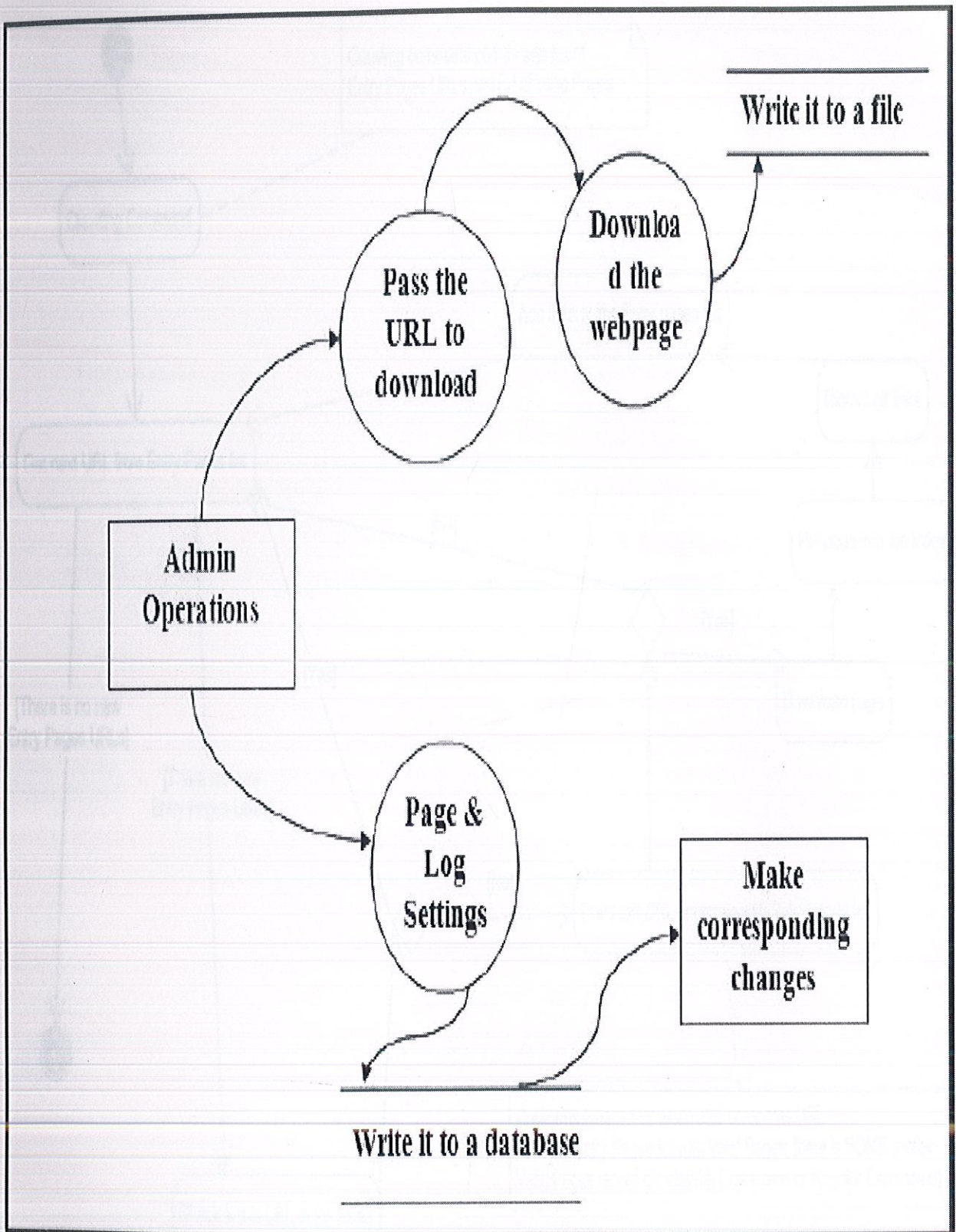
- **Crawling the web:** Search engines run automated programs, called "robots" or "spiders", that use the hyperlink structure of the web to "crawl" the pages and documents that make up the World Wide Web.
- **Indexing documents:** Once a page has been crawled, its contents can be "indexed" - stored in a giant database of documents that makes up a search engine's "index".
- **Processing queries:** When a request for information comes into the search engine, the engine retrieves from its index all the document that match the query.
- **Ranking results:** Once the search engine has determined which results are a match for the query, the engine's algorithm (a mathematical equation commonly used for sorting) runs calculations on each of the results to determine which is most relevant to the given query

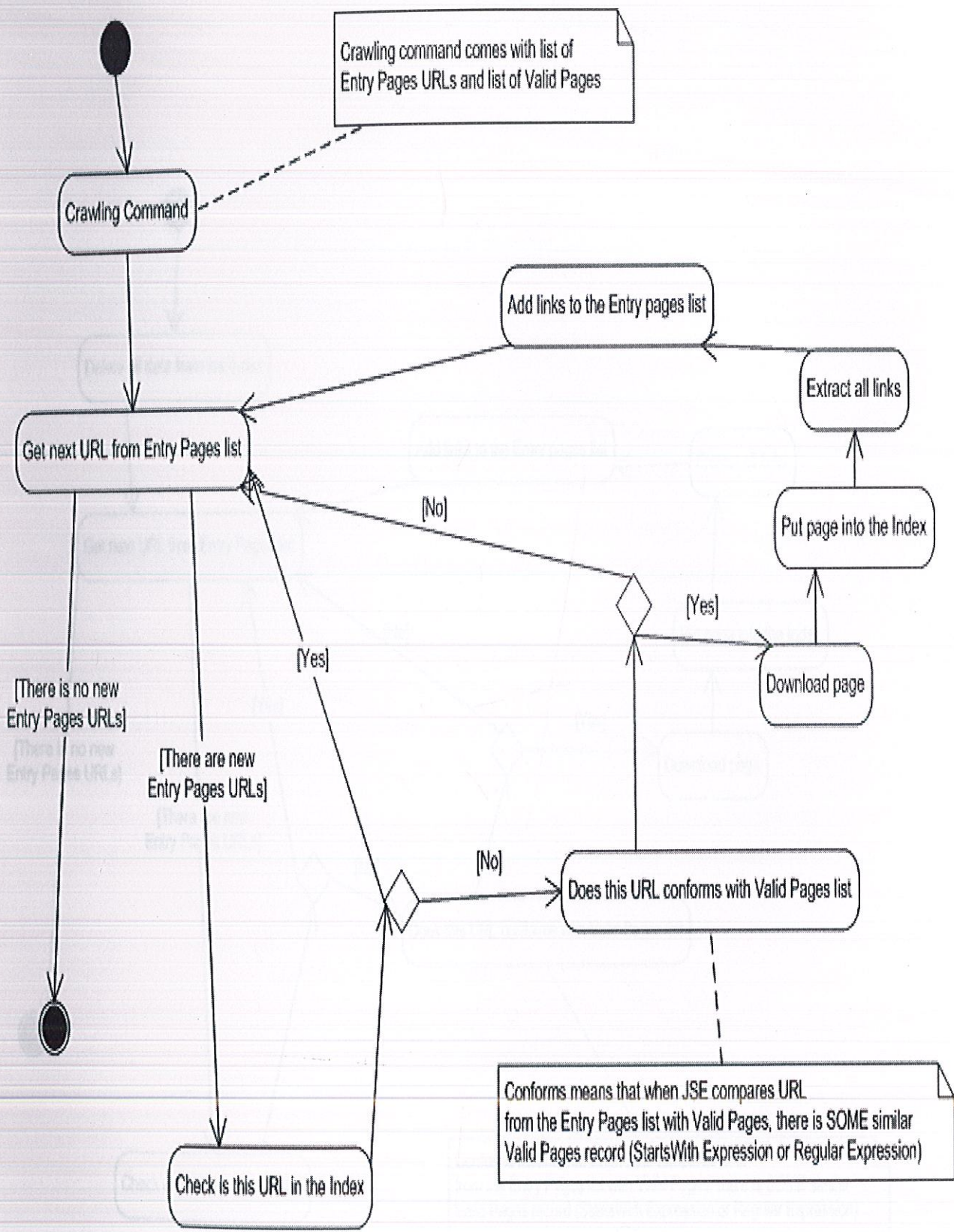


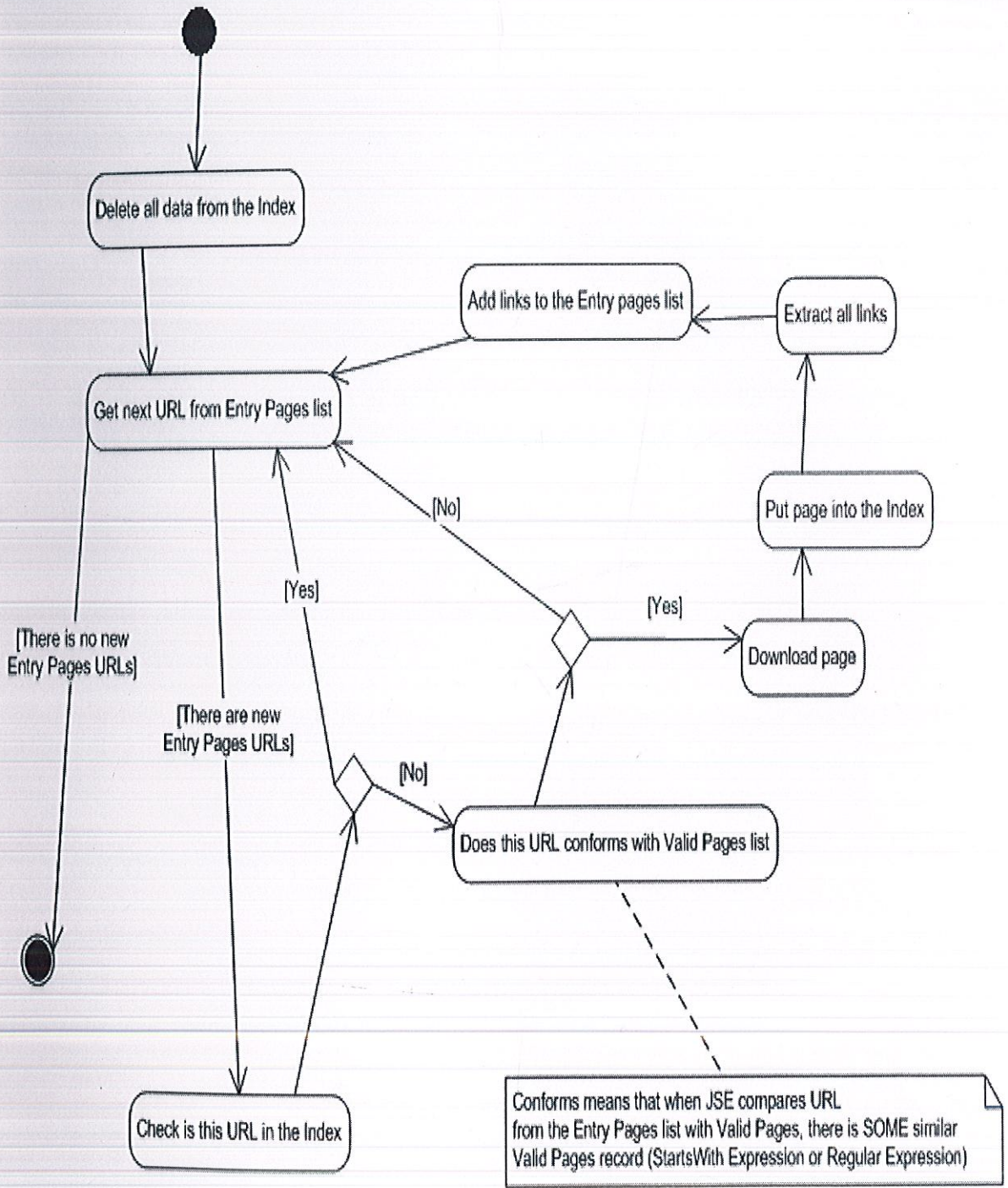


Adding command comes with list of Entry Pages URLs and list of Valid Pages

Conforms means that when JSE compares URL from the Entry Pages list with Valid Pages, there is SOME similar Valid Pages record (StartsWith Expression or Regular Expression)







3.1.1 Crawler

A Web crawler is a computer program that browses the World Wide Web in a systematic, automated manner. It is used by search engines, indexers, bots, scrapers, and web spiders.

This process is a cycle that repeats itself. It starts with a crawler that visits a page and extracts all links from it. It then checks if the URL is in the index. If not, it adds it to the entry pages list. The process then repeats for the next URL in the list.

The crawler starts by deleting all data from the index. It then gets the next URL from the entry pages list. It checks if there are new entry pages URLs. If not, it ends. If yes, it checks if the URL is in the index. If not, it adds it to the entry pages list. If yes, it downloads the page, puts it into the index, and extracts all links.

The flowchart shows the following steps: 1. Delete all data from the Index. 2. Get next URL from Entry Pages list. 3. Decision: [There is no new Entry Pages URLs] -> End. [There are new Entry Pages URLs] -> Check is this URL in the Index. 4. Decision: Does this URL conforms with Valid Pages list. [No] -> Add links to the Entry pages list. [Yes] -> Download page. 5. Download page -> Put page into the Index -> Extract all links -> Add links to the Entry pages list.

The flowchart includes decision diamonds and process ovals. The first decision diamond checks for new entry pages URLs. The second decision diamond checks if the URL conforms with the valid pages list. The process ovals represent the actions taken at each step.

The flowchart starts with a solid black circle at the top, leading to the 'Delete all data from the Index' process oval. It ends with a solid black circle at the bottom left, reached from the '[There is no new Entry Pages URLs]' path.

The flowchart uses standard UML notation: ovals for processes, diamonds for decisions, and arrows for flow. A dashed line connects the 'Does this URL conforms with Valid Pages list' process to a note box.

Conforms means that when JSE compares URL from the Entry Pages list with Valid Pages, there is SOME similar Valid Pages record (StartsWith Expression or Regular Expression)

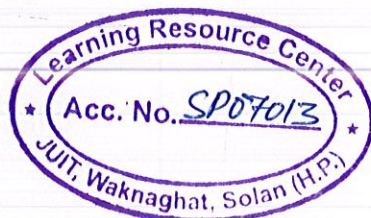
Chapter 3

3.1 Crawler

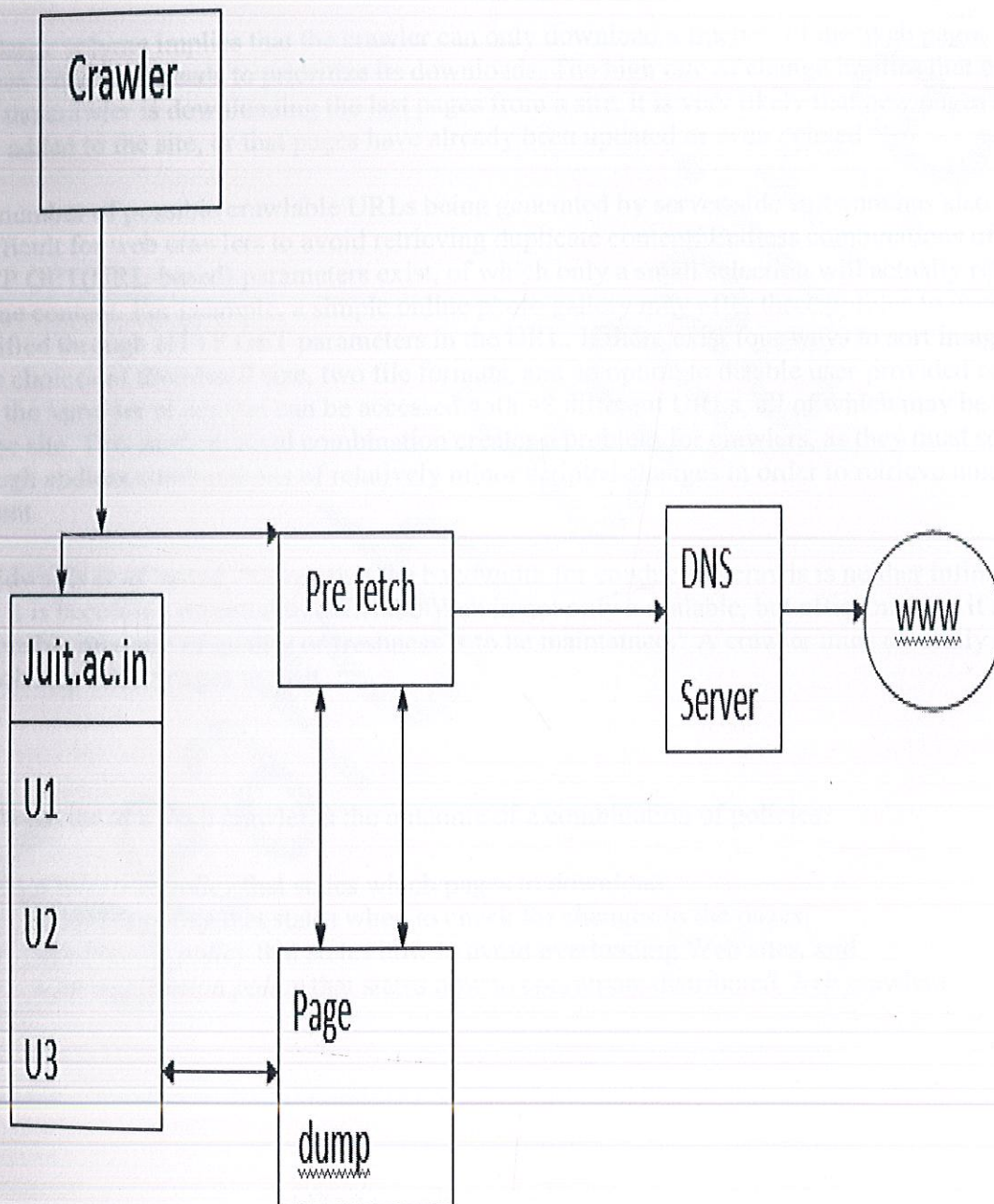
A **Web crawler** is a computer program that browses the World Wide Web in a methodical, automated manner or in an orderly fashion. Other terms for Web crawlers are *ants*, *automatic indexers*, *bots*, or *Web spiders*, *Web robots*, or—especially in the FOAF community—*Web scutters*.

This process is called *Web crawling* or *spidering*. Many sites, in particular search engines, use spidering as a means of providing up-to-date data. Web crawlers are mainly used to create a copy of all the visited pages for later processing by a search engine that will index the downloaded pages to provide fast searches. Crawlers can also be used for automating maintenance tasks on a Web site, such as checking links or validating HTML code. Also, crawlers can be used to gather specific types of information from Web pages, such as harvesting e-mail addresses (usually for spam).

A Web crawler is one type of bot, or software agent. In general, it starts with a list of URLs to visit, called the *seeds*. As the crawler visits these URLs, it identifies all the hyperlinks in the page and adds them to the list of URLs to visit, called the *crawl frontier*. URLs from the frontier are recursively visited according to a set of policies



3.1.1 Architecture of web crawler



3.1.2 Crawling Policies

There are important characteristics of the Web that make crawling very difficult:

- its large volume,
- its fast rate of change, and
- dynamic page generation.

The large volume implies that the crawler can only download a fraction of the Web pages within a given time, so it needs to prioritize its downloads. The high rate of change implies that by the time the crawler is downloading the last pages from a site, it is very likely that new pages have been added to the site, or that pages have already been updated or even deleted.

The number of possible crawlable URLs being generated by server-side software has also made it difficult for web crawlers to avoid retrieving duplicate content. Endless combinations of HTTP GET(URL-based) parameters exist, of which only a small selection will actually return unique content. For example, a simple online photo gallery may offer three options to users, as specified through HTTP GET parameters in the URL. If there exist four ways to sort images, three choices of thumbnail size, two file formats, and an option to disable user-provided content, then the same set of content can be accessed with 48 different URLs, all of which may be linked on the site. This mathematical combination creates a problem for crawlers, as they must sort through endless combinations of relatively minor scripted changes in order to retrieve unique content.

As Edwards *et al.* noted, "Given that the bandwidth for conducting crawls is neither infinite nor free, it is becoming essential to crawl the Web in not only a scalable, but efficient way, if some reasonable measure of quality or freshness is to be maintained." A crawler must carefully choose at each step which pages to visit

The behavior of a Web crawler is the outcome of a combination of policies:

- a *selection policy* that states which pages to download,
- a *re-visit policy* that states when to check for changes to the pages,
- a *politeness policy* that states how to avoid overloading Web sites, and
- a *parallelization policy* that states how to coordinate distributed Web crawlers

Selection policy

Given the current size of the Web, even large search engines cover only a portion of the publicly-available part. A 2005 study showed that large-scale search engines index no more than 40%-70% of the indexable Web; a previous study by Dr. Steve Lawrence and Lee Giles showed that no search engine indexed more than 16% of the Web in 1999. As a crawler always downloads just a fraction of the Web pages, it is highly desirable that the downloaded fraction contains the most relevant pages and not just a random sample of the Web.

This requires a metric of importance for prioritizing Web pages. The importance of a page is a function of its intrinsic quality, its popularity in terms of links or visits, and even of its URL (the latter is the case of vertical search engines restricted to a single top-level domain, or search engines restricted to a fixed Web site). Designing a good selection policy has an added difficulty: it must work with partial information, as the complete set of Web pages is not known during crawling

Focused crawling

The importance of a page for a crawler can also be expressed as a function of the similarity of a page to a given query. Web crawlers that attempt to download pages that are similar to each other are called **focused crawler** or **topical crawlers**. The concepts of topical and focused crawling were first introduced by Menczer and by Chakrabarti *et al.*

The main problem in focused crawling is that in the context of a Web crawler, we would like to be able to predict the similarity of the text of a given page to the query before actually downloading the page. A possible predictor is the anchor text of links; this was the approach taken by Pinkerton in a crawler developed in the early days of the Web. Diligenti *et al* propose to use the complete content of the pages already visited to infer the similarity between the driving query and the pages that have not been visited yet. The performance of a focused crawling depends mostly on the richness of links in the specific topic being searched, and a focused crawling usually relies on a general Web search engine for providing starting points.

Restricting followed links

A crawler may only want to seek out HTML pages and avoid all other MIME types. In order to request only HTML resources, a crawler may make an HTTP HEAD request to determine a Web resource's MIME type before requesting the entire resource with a GET request. To avoid making numerous HEAD requests, a crawler may examine the URL and only request a resource if the URL ends with certain characters such as .html, .htm, .asp, .aspx, .php, .jsp, .jspx or a slash. This strategy may cause numerous HTML Web resources to be unintentionally skipped.

Some crawlers may also avoid requesting any resources that have a "?" in them (are dynamically produced) in order to avoid spider traps that may cause the crawler to download an infinite number of URLs from a Web site. This strategy is unreliable if the site uses URL rewriting to simplify its URLs.

URL normalization

Crawlers usually perform some type of URL normalization in order to avoid crawling the same resource more than once. The term *URL normalization*, also called *URL canonicalization*, refers to the process of modifying and standardizing a URL in a consistent manner. There are several types of normalization that may be performed including conversion of URLs to lowercase, removal of "." and ".." segments, and adding trailing slashes to the non-empty path component.

Path-ascending crawling

Some crawlers intend to download as many resources as possible from a particular web site. So *path-ascending crawler* was introduced that would ascend to every path in each URL that it intends to crawl. For example, when given a seed URL of `http://llama.org/hamster/monkey/page.html`, it will attempt to crawl `/hamster/monkey/`, `/hamster/`, and `/`. Cothey found that a path-ascending crawler was very effective in finding isolated resources, or resources for which no inbound link would have been found in regular crawling.

Many path-ascending crawlers are also known as Web harvesting software, because they're used to "harvest" or collect all the content — perhaps the collection of photos in a gallery — from a specific page or host

Re-visit policy

The Web has a very dynamic nature, and crawling a fraction of the Web can take weeks or months. By the time a Web crawler has finished its crawl, many events could have happened, including creations, updates and deletions.

From the search engine's point of view, there is a cost associated with not detecting an event, and thus having an outdated copy of a resource. The most-used cost functions are freshness and age.

Freshness: This is a binary measure that indicates whether the local copy is accurate or not. The freshness of a page p in the repository at time t is defined as:

$$F_p(t) = \begin{cases} 1 & \text{if } p \text{ is equal to the local copy at time } t \\ 0 & \text{otherwise} \end{cases}$$

Age: This is a measure that indicates how outdated the local copy is. The age of a page p in the repository, at time t is defined as:

$$A_p(t) = \begin{cases} 0 & \text{if } p \text{ is not modified at time } t \\ t - \text{modification time of } p & \text{otherwise} \end{cases}$$

Coffman *et al.* worked with a definition of the objective of a Web crawler that is equivalent to freshness, but use a different wording: they propose that a crawler must minimize the fraction of time pages remain outdated. They also noted that the problem of Web crawling can be modeled as a multiple-queue, single-server polling system, on which the Web crawler is the server and the Web sites are the queues. Page modifications are the arrival of the customers, and switch-over times are the interval between page accesses to a single Web site. Under this model, mean waiting time for a customer in the polling system is equivalent to the average age for the Web crawler.

The objective of the crawler is to keep the average freshness of pages in its collection as high as possible, or to keep the average age of pages as low as possible. These objectives are not equivalent: in the first case, the crawler is just concerned with how many pages are out-dated, while in the second case, the crawler is concerned with how old the local copies of pages are.

Two simple re-visiting policies were studied by Cho and Garcia-Molina:

Uniform policy: This involves re-visiting all pages in the collection with the same frequency, regardless of their rates of change.

Proportional policy: This involves re-visiting more often the pages that change more frequently. The visiting frequency is directly proportional to the (estimated) change frequency.

(In both cases, the repeated crawling order of pages can be done either in a random or a fixed order.)

Cho and Garcia-Molina proved the surprising result that, in terms of average freshness, the uniform policy outperforms the proportional policy in both a simulated Web and a real Web crawl. The explanation for this result comes from the fact that, when a page changes too often, the crawler will waste time by trying to re-crawl it too fast and still will not be able to keep its copy of the page fresh.

To improve freshness, the crawler should penalize the elements that change too often. The optimal re-visiting policy is neither the uniform policy nor the proportional policy. The optimal method for keeping average freshness high includes ignoring the pages that change too often, and the optimal for keeping average age low is to use access frequencies that monotonically (and sub-linearly) increase with the rate of change of each page. In both cases, the optimal is closer to the uniform policy than to the proportional policy: as Coffman *et al.* note, "in order to minimize the expected obsolescence time, the accesses to any particular page should be kept as evenly spaced as possible". Explicit formulas for the re-visit policy are not attainable in general, but they are obtained numerically, as they depend on the distribution of page changes. Cho and Garcia-Molina show that the exponential distribution is a good fit for describing page changes, while Ipeirotis *et al.* show how to use statistical tools to discover parameters that affect this distribution. Note that the re-visiting policies considered here regard all pages as homogeneous in terms of quality ("all pages on the Web are worth the same"), something that is not a realistic scenario, so further information about the Web page quality should be included to achieve a better crawling policy

Politeness policy

Crawlers can retrieve data much quicker and in greater depth than human searchers, so they can have a crippling impact on the performance of a site. Needless to say, if a single crawler is performing multiple requests per second and/or downloading large files, a server would have a hard time keeping up with requests from multiple crawlers.

As noted by Koster, the use of Web crawlers is useful for a number of tasks, but comes with a price for the general community. The costs of using Web crawlers include:

- network resources, as crawlers require considerable bandwidth and operate with a high degree of parallelism during a long period of time;
- server overload, especially if the frequency of accesses to a given server is too high;
- poorly-written crawlers, which can crash servers or routers, or which download pages they cannot handle; and
- personal crawlers that, if deployed by too many users, can disrupt networks and Web servers.

A partial solution to these problems is the robots exclusion protocol, also known as the robots.txt protocol that is a standard for administrators to indicate which parts of their Web servers should not be accessed by crawlers. This standard does not include a suggestion for the interval of visits to the same server, even though this interval is the most effective way of avoiding server overload. Recently commercial search engines like Ask Jeeves, MSN and Yahoo are able to use an extra "Crawl-delay:" parameter in the robots.txt file to indicate the number of seconds to delay between requests.

The first proposal for the interval between connections was given in and was 60 seconds. However, if pages were downloaded at this rate from a website with more than 100,000 pages over a perfect connection with zero latency and infinite bandwidth, it would take more than 2 months to download only that entire Web site; also, only a fraction of the resources from that Web server would be used. This does not seem acceptable.

Cho uses 10 seconds as an interval for accesses, and the WIRE crawler uses 15 seconds as the default. The MercatorWeb crawler follows an adaptive politeness policy: if it took t seconds to download a document from a given server, the crawler waits for $10t$ seconds before downloading the next page. Dill *et al* use 1 second.

For those using Web crawlers for research purposes, a more detailed cost-benefit analysis is needed and ethical considerations should be taken into account when deciding where to crawl and how fast to crawl.

Anecdotal evidence from access logs shows that access intervals from known crawlers vary between 20 seconds and 3–4 minutes. It is worth noticing that even when being very polite, and taking all the safeguards to avoid overloading Web servers, some complaints from Web server

administrators are received. Brin and Page note that: "... running a crawler which connects to more than half a million servers (...) generates a fair amount of e-mail and phone calls. Because of the vast number of people coming on line, there are always those who do not know what a crawler is, because this is the first one they have seen."

Parallelization policy

A parallel crawler is a crawler that runs multiple processes in parallel. The goal is to maximize the download rate while minimizing the overhead from parallelization and to avoid repeated downloads of the same page. To avoid downloading the same page more than once, the crawling system requires a policy for assigning the new URLs discovered during the crawling process, as the same URL can be found by two different crawling processes.

Examples of Web crawlers

The following is a list of published crawler architectures for general-purpose crawlers (excluding focused web crawlers), with a brief description that includes the names given to the different components and outstanding features:

- **Yahoo! Slurp** is the name of the Yahoo Search crawler.
- **Msnbot** is the name of Microsoft's Bing webcrawler.
- **FAST Crawler** is a distributed crawler, used by Fast Search & Transfer, and a general description of its architecture is available.
- **Googlebot** is described in some detail, but the reference is only about an early version of its architecture, which was based in C++ and Python. The crawler was integrated with the indexing process, because text parsing was done for full-text indexing and also for URL extraction. There is a URL server that sends lists of URLs to be fetched by several crawling processes. During parsing, the URLs found were passed to a URL server that checked if the URL have been previously seen. If not, the URL was added to the queue of the URL server.
- **Methabot** is a scriptable web crawler written in C, released under the ISC license.
- **arachnode.net** is an open-source .NET web crawler written in C# using SQL 2005/SQL 2008 and Lucene.
- **PolyBot** is a distributed crawler written in C++ and Python, which is composed of a "crawl manager", one or more "downloaders" and one or more "DNS resolvers". Collected URLs are added to a queue on disk, and processed later to search for seen URLs in batch mode. The politeness policy considers both third and second level domains (e.g.: `www.example.com` and `www2.example.com` are third level domains) because third level domains are usually hosted by the same Web server.
- **RBSE** was the first published web crawler. It was based on two programs: the first program, "spider" maintains a queue in a relational database, and the second program "mite", is a modified www ASCII browser that downloads the pages from the Web.
- **WebCrawler** was used to build the first publicly-available full-text index of a subset of the Web. It was based on lib-WWW to download pages, and another program to parse

and order URLs for breadth-first exploration of the Web graph. It also included a real-time crawler that followed links based on the similarity of the anchor text with the provided query.

- **World Wide Web Worm** was a crawler used to build a simple index of document titles and URLs. The index could be searched by using the grep Unix command.
- **WebFountain** is a distributed, modular crawler similar to Mercator but written in C++. It features a "controller" machine that coordinates a series of "ant" machines. After repeatedly downloading pages, a change rate is inferred for each page and a non-linear programming method must be used to solve the equation system for maximizing freshness. The authors recommend to use this crawling order in the early stages of the crawl, and then switch to a uniform crawling order, in which all pages are being visited with the same frequency.
- **WebRACE** is a crawling and caching module implemented in Java, and used as a part of a more generic system called eRACE. The system receives requests from users for downloading web pages, so the crawler acts in part as a smart proxy server. The system also handles requests for "subscriptions" to Web pages that must be monitored: when the pages change, they must be downloaded by the crawler and the subscriber must be notified. The most outstanding feature of WebRACE is that, while most crawlers start with a set of "seed" URLs, WebRACE is continuously receiving new starting URLs to crawl from.

3.2 Indexing

Web indexing (or "Internet indexing") includes back-of-book-style indexes to individual websites or an intranet, and the creation of keyword metadata to provide a more useful vocabulary for Internet or onsite search engines. With the increase in the number of periodicals that have articles online, web indexing is also becoming important for periodical websites.

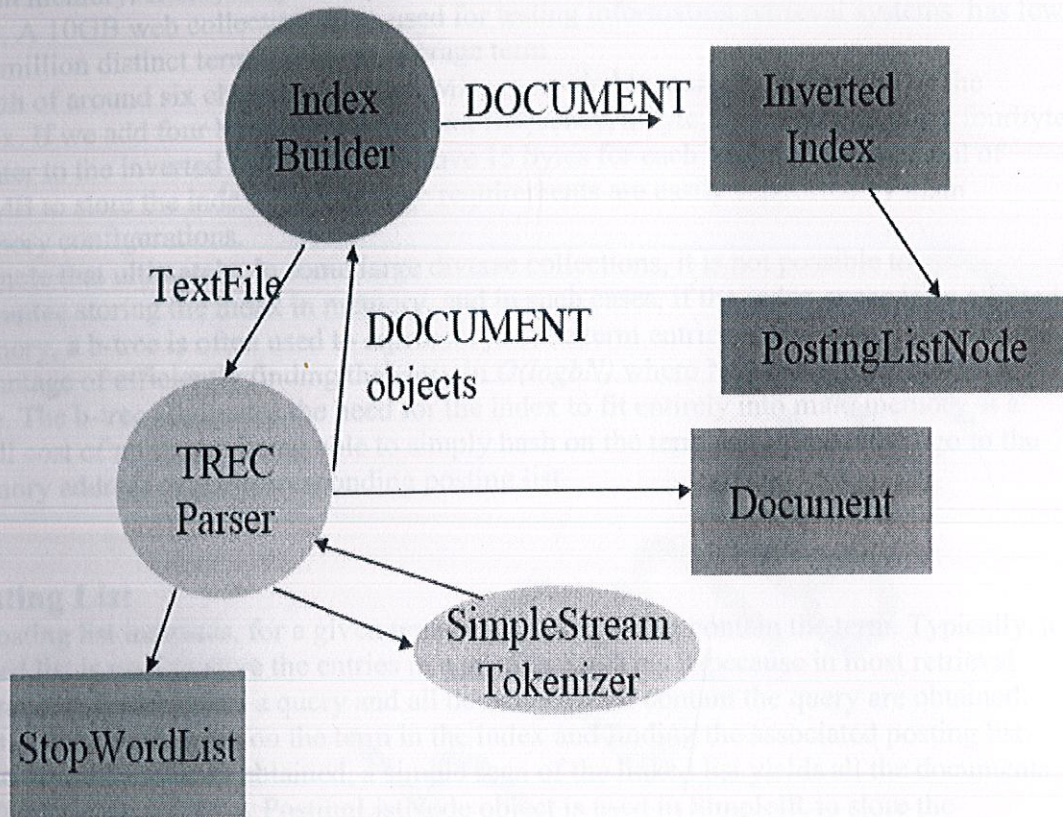
Back-of-the-book-style web indexes may be called "web site A-Z indexes." The implication with "A-Z" is that there is an alphabetical browse view or interface. This interface differs from that of a browse through layers of hierarchical categories (also known as a taxonomy) which are not necessarily alphabetical, but are also found on some web sites.

Web site A-Z indexes have several advantages over Search Engines - Language is full of homographs and synonyms and not all the references found will be relevant. For example, a computer-produced index of the 9/11 report showed many references for George Bush, but did not distinguish between GB senior and George W. In an environmental text, the phrase "lead users" will refer, not only to users of the metallic element, but also to early adopters of technology. Some hits will be time-wasting references, such as looking up "teaching children" and finding only the statement saying "... the above is not relevant when teaching children". Possibly more concerning, Search Engines may miss information – looking up the Dardanelles and missing references to the Hellespont or Çanakkale Boğazı, or seeking information about population and missing discussions about the number of people living in the area. A human-produced index has someone check each and every part of the text to find everything relevant to the search term, while a Search Engine leaves the responsibility for finding the information with the enquirer.

Although an A-Z index could be used to index multiple sites, rather than the multiple pages of a single site, this is unusual.

Metadata web indexing involves assigning keywords or phrases to web pages or web sites within a meta-tag field, so that the web page or web site can be retrieved with a search engine that is customized to search the keywords field. This may or may not involve using keywords restricted to a controlled vocabulary list.

Inverted Index



IndexBuilder drives the indexing process. When instantiated, the *build* method in this object actually parses documents using the Parser. As mentioned in Chapter 2, the parser returns a list of **Document** objects that contain parsed terms. The parsed documents objects are then sent to the *add* method in the inverted index. Once the index is built, users are able to efficiently submit queries against the text. When completed, the index is written to disk with the *write* method.

Index

The index is simply a list of terms. Since we rarely traverse this list, a hash table is often used as the data structure for the index. A hash table permits quick access to a term by applying a hash function on the term. If two terms have the same hash entry, a collision resolution algorithm (such as simply using a linked list of collisions) can be employed. The Java JDK hides the details of collisions from the user with the **HashMap** class. Once we find the correct entry for a term, a pointer will exist to the relevant posting list.

The index also may contain other useful information about a term. The size of the posting list indicates the number of documents that contain the term. Typically this is a value stored with the term entry in the index. Other values might include the term type, namely, is the term a phrase, an acronym, a number, etc.

A crucial assumption with many information retrieval systems is that the index fits in memory. If only single term entries are stored in the index then this is easily the case. A 10GB web collection often used for testing information retrieval systems has fewer than one million distinct terms. With an average term length of around six characters, only 6 MB are needed to store the actual text of the index. If we add four bytes for a document frequency, a byte for a type tag, and a fourbyte pointer to the inverted index, we now have 15 bytes for each term requiring a total of 15 MB to store the index. Such storage requirements are easily supported by main memory configurations.

We note that ultimately, in some large diverse collections, it is not possible to guarantee storing the index in memory, and in such cases, if the index exceeds its allotted memory, a b-tree is often used to represent just the term entries in the index. This has the advantage of efficiently finding the entry in $O(\log bN)$ where N is the number of terms time. The b-tree eliminates the need for the index to fit entirely into main memory at a small cost of no longer being able to simply hash on the term and immediately go to the memory address of the corresponding posting list.

Posting List

A posting list indicates, for a given term, which documents contain the term. Typically, a linked list is used to store the entries in a posting list. This is because in most retrieval operations, a user enters a query and all documents that contain the query are obtained. This is done by hashing on the term in the index and finding the associated posting list. Once the posting list is obtained, a simple scan of the linked list yields all the documents that contain the query. A `PostingListNode` object is used in SimpleIR to store the document identifier and the term frequency for each occurrence in a `PostingList`.

Document List

The inverted index only needs a term dictionary, namely the index, and the corresponding posting lists. However, we found it convenient to also include the `documentList` variable in our `InvertedIndex` object. The `documentList` is simply an `ArrayList` of `Document` object. Its purpose is to map a document identifier to a `Document` object. The `Document` object contains information unique to each document, for example, the author, title, date published, text file that contains the document, etc.). Storing the list of document objects with the inverted index makes sense because the only time it is updated is when we update the inverted index.

Index Builder

The index builder drives the indexing process. The constructor reads the configuration file with the `Properties` object to identify what stop word list to use and what text files are used as input. The `build` method calls the parser, and the parser returns a set of document objects. Next, the index builder loops through all the document objects and calls the `add` method associated with the `InvertedIndex` object to add each document to the inverted index. The `IndexBuilder` is designed so that different inverted indexes can be

constructed for different document collections. Also different parsers are easily incorporated for different types of documents. Once all documents have been processed, the *write* method is used to store the *InvertedIndex* object to disk.

Adding a Document

To add a document to the index, SimpleIR has an *add* method associated with the *InvertedIndex*. The *Document* object is created and populated by the parser. The *add* method simply accepts a *Document* object and adds it to the inverted index. The *add* method is given below. The method begins by getting the document identifier for the document to be added. Next, the list of distinct terms is obtained from the *Document*. An iterator is obtained for this list, and now, a loop begins which adds each term to the posting list. For a given term, we first check to see if a posting list already exists for this term. If it does, we simply retrieve the posting list (using the same *containsKey* method used in the *getPostingList* method). If no posting list exists, we instantiate a null *LinkedList* and associate this null list with the *index* *HashMap*. At this point, the term we are adding has a *postingList* associated with it. All that remains is to retrieve the *postingList*, instantiate

It simply contains a *documentID* and a *TermFrequency* object. The *TermFrequency* is a separate object because we wish to handle a case that does not come up too often. A two-byte term frequency can store 32767 occurrences of a term in a given document. Should the term have a higher frequency, we do not want an overflow condition to occur. Instead of checking for this overflow condition throughout all of SimpleIR, we simply build a *TermFrequency* object that has a special *increment* method. The special *increment* method stops incrementing when the term's frequency exceeds 32767 and avoids an overflow condition. While capping the term frequency at 32767 may at first glance seem to skew the calculations, in fact, if a document is long enough to contain a single word 32,767 times, it is probably not the document you are looking for! (Besides, it turns out, as we will see in discussing similarity measures, that very large term frequencies do not proportionately impact the relevance measure).

Retrieving a Posting List

The *getPostingList* method takes a given term as an argument and returns a *LinkedList* that is the posting list associated with the term. If no linked list is available, this means the term does not currently exist in the inverted index and a value of *null* is returned. When a *token* is passed to *getPostingList*, the *HashMap* is checked with the *containsKey* method. This implements a hashing function on the term and returns true if the term exists in the index. When the term exists, we return the corresponding posting list, otherwise, a null is returned.

3.3 Page Ranking

Processing a set of Documents

Now that we have a parser and an InvertedIndex method we can show how documents are added to the index.

The *build* method in the IndexBuilder object as given above illustrates a simple index build routine. It starts by instantiating a new InvertedIndex object. The index is cleared and a property TEXT_FILE is read from disk. TEXT_FILE indicates the file that should be indexed. A STOPWORD_FILE enables us to dynamically change the list of stop words used by the search engine. Next, a TRECParser is instantiated for this file with a given list of stop words.

It may seem overly trivial to only index a single file, but in truth, this *build* routine is designed for scalability. Most web search engines use many different processors to parse documents. By separating the parser from the index building, it is possible to launch numerous instances of the parser on numerous machines. Each parser accepts as a parameter the file to parse and the stopwords to use. After the parser is instantiated, the *readDocuments* method is called to actually read and parse the documents. A list of document objects is returned.

3.3 Page Ranking

Description:

PageRank is a link analysis algorithm, named after Larry Page, used by the Google Internet search engine that assigns a numerical weighting to each element of a hyperlinked set of documents, such as the World Wide Web, with the purpose of "measuring" its relative importance within the set. The algorithm may be applied to any collection of entities with reciprocal quotations and references. The numerical weight that it assigns to any given element E is referred to as the PageRank of E and denoted by $PR(E)$.

The name "PageRank" is a trademark of Google, and the PageRank process has been patented (U.S. Patent 6,285,999). However, the patent is assigned to Stanford University and not to Google. Google has exclusive license rights on the patent from Stanford University. The university received 1.8 million shares of Google in exchange for use of the patent; the shares were sold in 2005 for \$336 million.

PageRank reflects our view of the importance of web pages by considering more than 500 million variables and 2 billion terms. Pages that we believe are important pages receive a higher PageRank and are more likely to appear at the top of the search results.

PageRank also considers the importance of each page that casts a vote, as votes from some pages are considered to have greater value, thus giving the linked page greater value. We have always taken a pragmatic approach to help improve search quality and create useful products, and our technology uses the collective intelligence of the web to determine a page's importance

In other words, a PageRank results from a "ballot" among all the other pages on the World Wide Web about how important a page is. A hyperlink to a page counts as a vote of support. The PageRank of a page is defined recursively and depends on the number and PageRank metric of all pages that link to it ("incoming links"). A page that is linked to by many pages with high PageRank receives a high rank itself. If there are no links to a web page there is no support for that page.

Google assigns a numeric weighting from 0-10 (but 0 is used just for penalized or non analyzed-pages) for each webpage on the Internet; this PageRank denotes a site's importance in the eyes of Google. The PageRank is derived from a theoretical probability value on a logarithmic scale like the Richter Scale. The PageRank of a particular page is roughly based upon the quantity of inbound links as well as the PageRank of the pages providing the links. It is known that other factors, e.g. relevance of search words on the page and actual visits to the page reported by the Google toolbar also influence the PageRank. In order to prevent manipulation, spoofing and Spamdexing, Google provides no specific details about how other factors influence PageRank.

Numerous academic papers concerning PageRank have been published since Page and Brin's original paper. In practice, the PageRank concept has proven to be vulnerable to manipulation, and extensive research has been devoted to identifying falsely inflated PageRank and ways to ignore links from documents with falsely inflated PageRank.

Other link-based ranking algorithms for Web pages include the HITS algorithm invented by Jon Kleinberg (used by Teoma and now Ask.com), the IBM CLEVER project, and the TrustRank algorithm.

History:

PageRank was developed at Stanford University by Larry Page (hence the name Page-Rank) and later Sergey Brin as part of a research project about a new kind of search engine. It was co-authored by Rajeew Motwani and Terry Winograd. The first paper about the project, describing PageRank and the initial prototype of the Google search engine, was published in 1998: shortly after, Page and Brin founded Google Inc., the company behind the Google search engine. While just one of many factors which determine the ranking of Google search results, PageRank continues to provide the basis for all of Google's web search tools.

PageRank has been influenced by citation analysis, early developed by Eugene Garfield in the 1950s at the University of Pennsylvania, and by Hyper Search, developed by Massimo Marchiori at the University of Padua. In the same year PageRank was introduced (1998), Jon Kleinberg published his important work on HITS. Google's founders cite Garfield, Marchiori, and Kleinberg in their original paper.

Algorithm:

PageRank is a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page. PageRank can be calculated for collections of documents of any size. It is assumed in several research papers that the distribution is evenly divided among all documents in the collection at the beginning of the computational process. The PageRank computations require several passes, called "iterations", through the collection to adjust approximate PageRank values to more closely reflect the theoretical true value.

A probability is expressed as a numeric value between 0 and 1. A 0.5 probability is commonly expressed as a "50% chance" of something happening. Hence, a PageRank of 0.5 means there is a 50% chance that a person clicking on a random link will be directed to the document with the 0.5 PageRank.

Simplified algorithm:

Assume a small universe of four web pages: A, B, C and D. The initial approximation of PageRank would be evenly divided between these four documents. Hence, each document would begin with an estimated PageRank of 0.25.

In the original form of PageRank initial values were simply 1. This meant that the sum of all pages was the total number of pages on the web. Later versions of PageRank (see the formulas below) would assume a probability distribution between 0 and 1. Here a simple probability distribution will be used- hence the initial value of 0.25.

If pages B, C, and D each only link to A, they would each confer 0.25 PageRank to A. All PageRank $PR()$ in this simplistic system would thus gather to A because all links would be pointing to A.

$$PR(A) = PR(B) + PR(C) + PR(D).$$

This is 0.75.

Suppose that page B has a link to page C as well as to page A, while page D has links to all three pages. The value of the link-votes is divided among all the outbound links on a page. Thus, page B gives a vote worth 0.125 to page A and a vote worth 0.125 to page C. Only one third of D's PageRank is counted for A's PageRank (approximately 0.083).

$$PR(A) = \frac{PR(B)}{2} + \frac{PR(C)}{1} + \frac{PR(D)}{3}.$$

In other words, the PageRank conferred by an outbound link is equal to the document's own PageRank score divided by the normalized number of outbound links $L()$ (it is assumed that links to specific URLs only count once per document).

$$PR(A) = \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)}.$$

In the general case, the PageRank value for any page u can be expressed as:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)},$$

i.e. the PageRank value for a page u is dependent on the PageRank values for each page v out of the set B_u (this set contains all pages linking to page u), divided by the number $L(v)$ of links from page v .

Damping Factor:

The PageRank theory holds that even an imaginary surfer who is randomly clicking on links will eventually stop clicking. The probability, at any step, that the person will continue is a damping factor d . Various studies have tested different damping factors, but it is generally assumed that the damping factor will be set around 0.85.

The damping factor is subtracted from 1 (and in some variations of the algorithm, the result is divided by the number of documents (N) in the collection) and this term is then added to the product of the damping factor and the sum of the incoming PageRank scores. That is,

$$PR(A) = \frac{1-d}{N} + d \left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \dots \right).$$

So any page's PageRank is derived in large part from the PageRanks of other pages. The damping factor adjusts the derived value downward. The original paper, however, gave the following formula, which has led to some confusion:

$$PR(A) = 1 - d + d \left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \dots \right).$$

The difference between them is that the PageRank values in the first formula sum to one, while in the second formula each PageRank gets multiplied by N and the sum becomes N. A statement in Page and Brin's paper that "the sum of all PageRanks is one" and claims by other Google employees support the first variant of the formula above.

To be more specific, in the latter formula, the probability for the random surfer reaching a page is weighted by the total number of web pages. So, in this version PageRank is an expected value for the random surfer visiting a page, when he restarts this procedure as often as the web has pages. If the web had 100 pages and a page had a PageRank value of 2, the random surfer would reach that page in an average twice if he restarts 100 times. Basically, the two formulas do not differ fundamentally from each other. A PageRank which has been calculated by using the former formula has to be multiplied by the total number of web pages to get the according PageRank that would have been calculated by using the latter formula. Even Page and Brin mixed up the two formulas in their most popular paper "The Anatomy of a Large-Scale Hypertextual Web Search Engine", where they claim the latter formula to form a probability distribution over web pages with the sum of all pages' PageRanks being one.

Google recalculates PageRank scores each time it crawls the Web and rebuilds its index. As Google increases the number of documents in its collection, the initial approximation of PageRank decreases for all documents.

The formula uses a model of a random surfer who gets bored after several clicks and switches to a random page. The PageRank value of a page reflects the chance that the random surfer will land on that page by clicking on a link. It can be understood as a Markov chain in which the states are pages, and the transitions are all equally probable and are the links between pages.

If a page has no links to other pages, it becomes a sink and therefore terminates the random surfing process. If the random surfer arrives at a sink page, it picks another URL at random and continues surfing again.

When calculating PageRank, pages with no outbound links are assumed to link out to all other pages in the collection. Their PageRank scores are therefore divided evenly among all other pages. In other words, to be fair with pages that are not sinks, these random transitions are added to all nodes in the Web, with a residual probability of usually $d = 0.85$, estimated from the frequency that an average surfer uses his or her browser's bookmark feature.

So, the equation is as follows:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

where p_1, p_2, \dots, p_N are the pages under consideration, $M(p_i)$ is the set of pages that link to p_i , $L(p_j)$ is the number of outbound links on page p_j , and N is the total number of pages.

The PageRank values are the entries of the dominant eigenvector of the modified adjacency matrix. This makes PageRank a particularly elegant metric: the eigenvector is

$$\mathbf{R} = \begin{bmatrix} PR(p_1) \\ PR(p_2) \\ \vdots \\ PR(p_N) \end{bmatrix}$$

where \mathbf{R} is the solution of the equation

$$\mathbf{R} = \begin{bmatrix} (1-d)/N \\ (1-d)/N \\ \vdots \\ (1-d)/N \end{bmatrix} + d \begin{bmatrix} \ell(p_1, p_1) & \ell(p_1, p_2) & \cdots & \ell(p_1, p_N) \\ \ell(p_2, p_1) & \ddots & & \vdots \\ \vdots & & \ell(p_i, p_j) & \\ \ell(p_N, p_1) & \cdots & & \ell(p_N, p_N) \end{bmatrix} \mathbf{R}$$

where the adjacency function $\ell(p_i, p_j)$ is 0 if page p_j does not link to p_i , and normalized such that, for each j

$$\sum_{i=1}^N \ell(p_i, p_j) = 1$$

i.e. the elements of each column sum up to 1, so the matrix is a stochastic matrix (for more details see the computation section below). Thus this is a variant of the eigenvector centrality measure used commonly in network analysis.

Because of the large eigengap of the modified adjacency matrix above,^[8] the values of the PageRank eigenvector are fast to approximate (only a few iterations are needed).

As a result of Markov theory, it can be shown that the PageRank of a page is the probability of being at that page after lots of clicks. This happens to equal t^{-1} where t is the expectation of the number of clicks (or random jumps) required to get from the page back to itself.

The main disadvantage is that it favors older pages, because a new page, even a very good one, will not have many links unless it is part of an existing site (a site being a densely connected set of pages, such as Wikipedia). The Google Directory (itself a derivative of the Open Directory Project) allows users to see results sorted by PageRank within categories. The Google Directory is the only service offered by Google where PageRank directly determines display order. In Google's other search services (such as its primary Web search) PageRank is used to weight the relevance scores of pages shown in search results.

Several strategies have been proposed to accelerate the computation of PageRank.

Various strategies to manipulate PageRank have been employed in concerted efforts to improve search results rankings and monetize advertising links. These strategies have severely impacted the reliability of the PageRank concept, which seeks to determine which documents are actually highly valued by the Web community.

Google is known to penalize link farms and other schemes designed to artificially inflate PageRank. In December 2007 Google started actively penalizing sites selling paid text links. How Google identifies link farms and other PageRank manipulation tools are among Google's trade secrets.

Computation:

To summarize, PageRank can be either computed iteratively or algebraically. The iterative method can be viewed differently as the power iteration method, or power method. The basic mathematical operations performed in the iterative method and the power method are identical.

Iterative:

In the former case, at $t = 0$, an initial probability distribution is assumed, usually

$$PR(p_i; 0) = \frac{1}{N}.$$

At each time step, the computation, as detailed above, yields

$$PR(p_i; t + 1) = \frac{1 - d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j; t)}{L(p_j)},$$

or in matrix notation

$$\mathbf{R}(t + 1) = d\mathbf{M}\mathbf{R}(t) + \frac{1 - d}{N}\mathbf{1}, \quad (*)$$

where $\mathbf{R}_i(t) = PR(p_i; t)$ and $\mathbf{1}$ is the column vector of length N containing only ones.

The matrix \mathcal{M} is defined as

$$\mathcal{M}_{ij} = \begin{cases} 1/L(p_j), & \text{if } j \text{ links to } i \\ 0, & \text{otherwise} \end{cases}$$

i.e.,

$$\mathcal{M} := (\mathbf{K}^{-1}\mathbf{A})^t,$$

where \mathbf{A} denotes the adjacency matrix of the graph and \mathbf{K} is the diagonal matrix with the outdegrees in the diagonal.

The computation ends when for some small ϵ

$$|\mathbf{R}(t+1) - \mathbf{R}(t)| < \epsilon,$$

i.e., when convergence is assumed.

Algebraic:

In the latter case, for $t \rightarrow \infty$ (i.e., in the steady state), the above equation (*) reads

$$\mathbf{R} = d\mathcal{M}\mathbf{R} + \frac{1-d}{N}\mathbf{1}. \quad (**)$$

The solution is given by

$$\mathbf{R} = (\mathbf{I} - d\mathcal{M})^{-1} \frac{1-d}{N} \mathbf{1},$$

with the identity matrix \mathbf{I} .

The solution exists and is unique for $0 < d < 1$. This can be seen by noting that \mathcal{M} is by construction a stochastic matrix and hence has an eigenvalue equal to one because of the Perron-Frobenius theorem.

Power Method:

If the matrix \mathcal{M} is a transition probability, i.e., column-stochastic with no columns consisting of just zeros and \mathbf{R} is a probability distribution (i.e., $|\mathbf{R}| = 1$, $\mathbf{E}\mathbf{R} = \mathbf{1}$ where \mathbf{E} is matrix of all ones), Eq. (***) is equivalent to

$$\mathbf{R} = \left(d\mathcal{M} + \frac{1-d}{N}\mathbf{E} \right) \mathbf{R} =: \widehat{\mathcal{M}}\mathbf{R} \quad (***)$$

Hence PageRank \mathbf{R} is the principal eigenvector of $\widehat{\mathcal{M}}$. A fast and easy way to compute this is using the power method: starting with an arbitrary vector $\mathbf{x}(0)$, the operator $\widehat{\mathcal{M}}$ is applied in succession, i.e.,

$$\mathbf{x}(t+1) = \widehat{\mathcal{M}}\mathbf{x}(t),$$

until

$$|\mathbf{x}(t+1) - \mathbf{x}(t)| < \varepsilon.$$

Note that in Eq. (***) the matrix on the right-hand side in the parenthesis can be interpreted as

$$\frac{1-d}{N}\mathbf{I} = (1-d)\mathbf{P}\mathbf{1}^t,$$

where \mathbf{P} is an initial probability distribution. In the current case

$$\mathbf{P} := \frac{1}{N}\mathbf{1}$$

Finally, if \mathcal{M} has columns with only zero values, they should be replaced with the initial probability vector \mathbf{P} . In other words

$$\mathcal{M}' := \mathcal{M} + \mathcal{D},$$

where the matrix \mathcal{D} is defined as

$$\mathcal{D} := \mathbf{D}\mathbf{P}^t,$$

with

$$D_i = \begin{cases} 1, & \text{if } L(p_i) = 0 \\ 0, & \text{otherwise} \end{cases}$$

In this case, the above two computations using \mathcal{M} only give the same PageRank if their results are normalized:

$$R_{\text{power}} = \frac{R_{\text{iterative}}}{|R_{\text{iterative}}|} = \frac{R_{\text{algebraic}}}{|R_{\text{algebraic}}|}.$$

Efficiency:

Depending on the framework used to perform the computation, the exact implementation of the methods, and the required accuracy of the result, the computation time of these methods can vary greatly. Usually if the computation has to be performed many times (i.e., for growing networks) or the network size is large, the algebraic computation is slower and more memory hungry due to the inversion of the matrix.

Chapter 4. Sample Codes

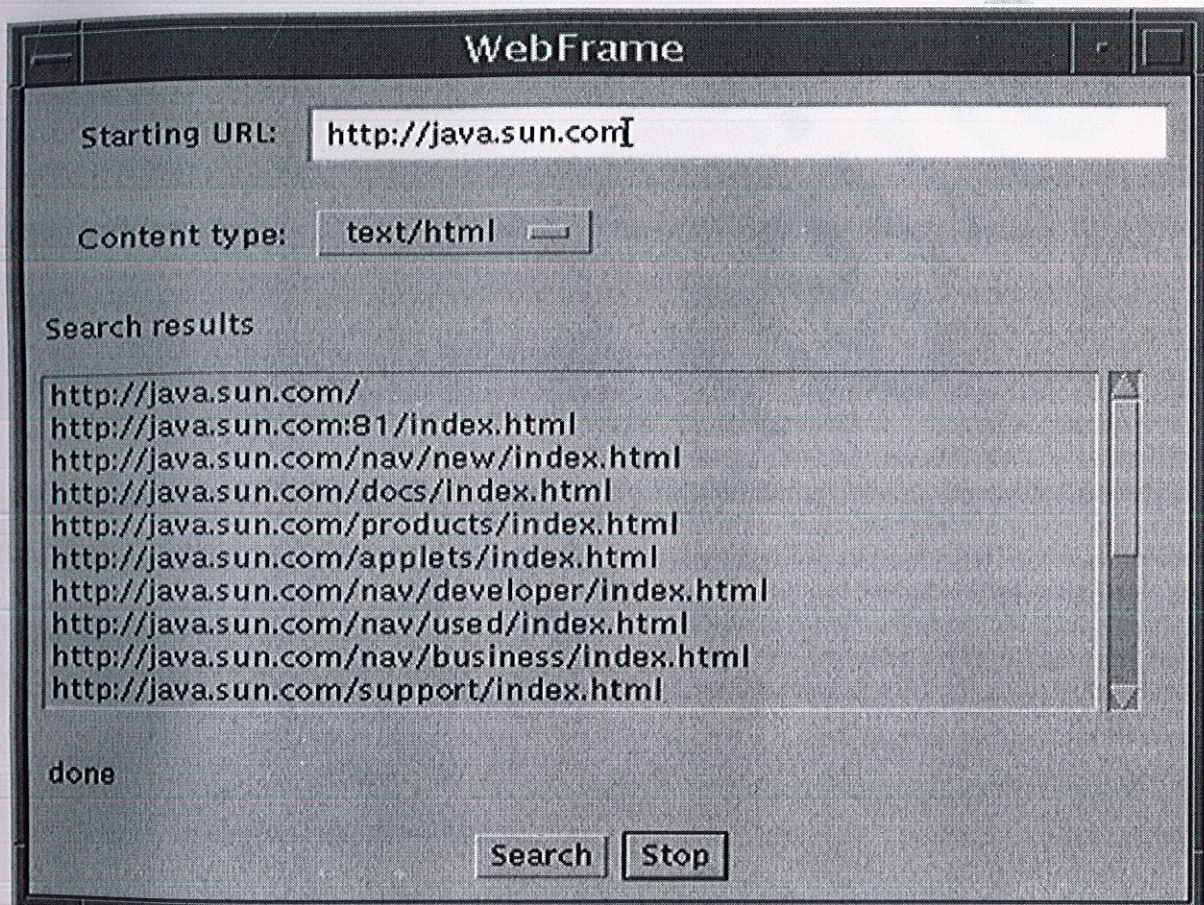
4.1 Crawling

Applet Frame

```
public void init() {  
    panelMain = new Panel();  
    panelMain.setLayout(new BorderLayout(5, 5));  
    Panel panelEntry = new Panel();  
    panelEntry.setLayout(new BorderLayout(5, 5));  
    Panel panelURL = new Panel();  
    panelURL.setLayout(new FlowLayout(FlowLayout.LEFT, 5, 5));  
    Label labelURL = new Label("Starting URL: ", Label.RIGHT);  
    panelURL.add(labelURL);  
    textURL = new TextField("", 40);  
    panelURL.add(textURL);  
    panelEntry.add("North", panelURL);  
    Panel panelType = new Panel();  
    panelType.setLayout(new FlowLayout(FlowLayout.LEFT, 5, 5));  
    Label labelType = new Label("Content type: ", Label.RIGHT);  
    panelType.add(labelType);  
    choiceType = new Choice();  
    choiceType.addItem("text/html");  
    panelType.add(choiceType);  
    panelEntry.add("South", panelType);  
  
    panelMain.add("North", panelEntry);
```

```
Panel panelListButtons = new Panel();
panelListButtons.setLayout(new BorderLayout(5, 5));
Panel panelList = new Panel();
panelList.setLayout(new BorderLayout(5, 5));
Label labelResults = new Label("Search results");
panelList.add("North", labelResults);
Panel panelListCurrent = new Panel();
panelListCurrent.setLayout(new BorderLayout(5, 5));
listMatches = new List(10);
panelListCurrent.add("North", listMatches);
labelStatus = new Label("");
panelListCurrent.add("South", labelStatus);
panelList.add("South", panelListCurrent);
panelListButtons.add("North", panelList);
Panel panelButtons = new Panel();
Button buttonSearch = new Button(SEARCH);
buttonSearch.addActionListener(this);
panelButtons.add(buttonSearch);
Button buttonStop = new Button(STOP);
buttonStop.addActionListener(this);
panelButtons.add(buttonStop);
panelListButtons.add("South", panelButtons);
panelMain.add("South", panelListButtons);
add(panelMain);
setVisible(true);
```

```
repaint();
vectorToSearch = new Vector();
vectorSearched = new Vector();
vectorMatches = new Vector();
URLConnection.setDefaultAllowUserInteraction(false);
```



Robot Exclusion Protocol Check

```
boolean robotSafe(URL url) {
```

```
String strHost = url.getHost();
String strRobot = "http://" + strHost + "/robots.txt";
URL urlRobot;
try {
    urlRobot = new URL(strRobot);
} catch (MalformedURLException e) {
    return false;
}
String strCommands;
try {
    InputStream urlRobotStream = urlRobot.openStream();
    byte b[] = new byte[1000];
    int numRead = urlRobotStream.read(b);
    strCommands = new String(b, 0, numRead);
    while (numRead != -1) {
        if (Thread.currentThread() != searchThread)
            break;
        numRead = urlRobotStream.read(b);
        if (numRead != -1) {
            String newCommands = new String(b, 0, numRead);
            strCommands += newCommands;
        }
    }
    urlRobotStream.close();
} catch (IOException e) {
```



```
return true;
}
String strURL = url.getFile();
int index = 0;
while ((index = strCommands.indexOf(DISALLOW, index)) != -1) {
    index += DISALLOW.length();
    String strPath = strCommands.substring(index);
    StringTokenizer st = new StringTokenizer(strPath);
    if (!st.hasMoreTokens())
        break;
    String strBadPath = st.nextToken();
    if (strURL.indexOf(strBadPath) == 0)
        return false;
} return true;
}
```

4.2 Indexing

Primary Index

** Add a character to the word being stemmed. When you are finished adding characters, you can call `stem(void)` to stem the word.*

```
public void add(char ch)
{ if (i == b.length)
  { char[] new_b = new char[i+INC];
    for (int c = 0; c < i; c++) new_b[c] = b[c];
    b = new_b;
  }
  b[i++] = ch;
}
```

** Adds `wLen` characters to the word being stemmed contained in a portion of a `char[]` array. This is like repeated calls of `add(char ch)`, but faster.*

```
public void add(char[] w, int wLen)
{ if (i+wLen >= b.length)
  { char[] new_b = new char[i+wLen+INC];
    for (int c = 0; c < i; c++) new_b[c] = b[c];
    b = new_b;
  }
  for (int c = 0; c < wLen; c++) b[i++] = w[c];
}
```

** After a word has been stemmed, it can be retrieved by `toString()`, or a reference to the internal buffer can be retrieved by `getResultBuffer` and `getResultLength` (which is generally more efficient.)*

```
public String toString() { return new String(b,0,i_end); }
public int getResultLength() { return i_end; }
public char[] getResultBuffer() { return b; }
private final boolean cons(int i)
{ switch (b[i])
  { case 'a': case 'e': case 'i': case 'o': case 'u': return false;
    case 'y': return (i==0) ? true : !cons(i-1);
    default: return true;
  }
}
```

* $m()$ measures the number of consonant sequences between 0 and j . if c is a consonant sequence and v a vowel sequence, and $\langle . \rangle$ indicates arbitrary presence,

$\langle c \rangle \langle v \rangle$ gives 0
 $\langle c \rangle v c \langle v \rangle$ gives 1
 $\langle c \rangle v c v c \langle v \rangle$ gives 2
 $\langle c \rangle v c v c v c \langle v \rangle$ gives 3


```
private final int m()
{ int n = 0;
  int i = 0;
  while(true)
  { if (i > j) return n;
    if (! cons(i)) break; i++;
  }
  i++;
  while(true)
  { while(true)
    { if (i > j) return n;
      if (cons(i)) break;
      i++;
    }
    i++;
    n++;
    while(true)
    { if (i > j) return n;
      if (! cons(i)) break;
      i++;
    }
    i++;
  }
}
```

* $vowelinstem()$ is true $\Leftrightarrow 0 \dots j$ contains a vowel

```
private final boolean vowelinstem()
{ int i; for (i = 0; i <= j; i++) if (! cons(i)) return true;
  return false;
}
```

* $doublec(j)$ is true $\Leftrightarrow j, (j-1)$ contain a double consonant.

```
private final boolean doublec(int j)
{ if (j < 1) return false;
  if (b[j] != b[j-1]) return false;
```

```
        return cons(j);
    }
```

** cvc(i) is true \Leftrightarrow i-2,i-1,i has the form consonant - vowel - consonant and also if the second c is not w,x or y. this is used when trying to restore an e at the end of a short word. e.g. cav(e), lov(e), hop(e), crim(e), but snow, box, tray.*

```
private final boolean cvc(int i)
{ if (i < 2 || !cons(i) || cons(i-1) || !cons(i-2)) return false;
  { int ch = b[i];
    if (ch == 'w' || ch == 'x' || ch == 'y') return false;
  }
  return true;
}
```

```
private final boolean ends(String s)
{ int l = s.length();
  int o = k-l+1;
  if (o < 0) return false;
  for (int i = 0; i < l; i++) if (b[o+i] != s.charAt(i)) return false;
  j = k-l;
  return true;
}
```

```
private final void setto(String s)
{ int l = s.length();
  int o = j+1;
  for (int i = 0; i < l; i++) b[o+i] = s.charAt(i);
  k = j+l;
}
```

Merge

```
class merge
```

```
{
```

```
    public static Pattern pattern;  
    public static BufferedWriter bufferedWriter;
```

```
    public static String mergeIndex(String split1, String split2)
```

```
    {
```

```
        Matcher matcher1 = pattern.matcher(split1), matcher2 = pattern.matcher(split2);
```

```
        StringBuffer stringBuffer = new StringBuffer();
```

```
        Double double1, double2;
```

```
        boolean check1 = matcher1.find(), check2 = matcher2.find();
```

```
        while(check1 == true && check2 == true)
```

```
        {
```

```
            double1 = Double.parseDouble(matcher1.group(2));
```

```
            double2 = Double.parseDouble(matcher2.group(2));
```

```
            if(double1 < double2)
```

```
            {
```

```
                stringBuffer.append(matcher2.group(0) + " ");
```

```
                check2 = matcher2.find();
```

```
            }
```

```
            else if(double1 > double2)
```

```
            {
```

```
                stringBuffer.append(matcher1.group(0) + " ");
```

```
                check1 = matcher1.find();
```

```
            }
```

```
            else
```

```
            {
```

```
                stringBuffer.append(matcher1.group(0) + " " + matcher2.group(0) + " ");
```

```
                check1 = matcher1.find();
```

```
                check2 = matcher2.find();
```

```
            }
```

```
        }
```

```
        while(check1 == true)
```

```
        {
```

```
            stringBuffer.append(matcher1.group(0) + " ");
```

```
            check1 = matcher1.find();
```

```
        }
```

```
        while(check2 == true)
```

```

    {
        stringBuffer.append(matcher2.group(0) + " ");
        check2 = matcher2.find();
    }

    return stringBuffer.toString();
}

public static void main(String args[])
{
    pattern = Pattern.compile("\\(([,^,]+),([^\^]+)\)");

    try
    {
        BufferedReader bufferedReader1 = new BufferedReader(new FileReader(args[0]));
        bufferedReader2 = new BufferedReader(new FileReader(args[1]));
        bufferedWriter = new BufferedWriter(new FileWriter(args[2]));

        String s1 = bufferedReader1.readLine(), s2 = bufferedReader2.readLine();

        while(s1 != null && s2 != null)
        {
            String[] split1 = s1.split(":"), split2 = s2.split(":");

            if(split1[0].equals(split2[0]))
            {
                s1 = mergeIndex(split1[1], split2[1]);
                bufferedWriter.write(split1[0] + ": " + s1 + "\n");
                s1 = bufferedReader1.readLine();

                s2 = bufferedReader2.readLine();
            }
            else if(split1[0].compareTo(split2[0]) < 0)
            {
                bufferedWriter.write(s1 + "\n");
                s1 = bufferedReader1.readLine();
            }
            else
            {
                bufferedWriter.write(s2 + "\n");
                s2 = bufferedReader2.readLine();
            }
        }
    }
}

```

Secondary Index

class secondaryIndex

public static void main

```
while(s1 != null)
{
    bufferedWriter.write(s1 + "\n");
    s1 = bufferedReader1.readLine();
}
```

```
while(s2 != null)
{
    bufferedWriter.write(s2 + "\n");
    s2 = bufferedReader2.readLine();
}
```

```
bufferedReader1.close();
bufferedReader2.close();
bufferedWriter.close();
```

```
}
catch (IOException e)
```

```
{
    e.printStackTrace();
}
```

```
}
```

Search Engine

Secondary Index

```
class secondaryIndex
{
    public static void main(String[] args)
    {
        try
        {
            BufferedReader bufferedReader = new BufferedReader(new FileReader(args[0]));
            BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(args[1]));

            String string;

            long offset = 0, c = 0;

            while(true)
            {
                string = bufferedReader.readLine();

                if(string == null)
                    break;

                c++;

                if(c == 1000)
                {
                    c = 0;

                    bufferedWriter.write(string.split(":")[0] + ":" + offset + "\n");
                }

                offset = offset + string.length() + 1;
            }

            bufferedReader.close();
            bufferedWriter.close();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```


4.3 Retrieval

```
String[] queryWordSplit = queryWord.split(":");
queryWordSplit[1] = (new retrieval()).Stemmer(queryWordSplit[1]);
if(queryWordSplit[0].equals("content"))
{
//      System.out.println("content:" + queryWordSplit[1]);
int index = Arrays.binarySearch(secondaryGlobalIndexKeySet, queryWordSplit[1]);

        long offset;

        if(index >= 0)
        {
            offset = (Long)secondaryGlobalIndex.get
            (queryWordSplit[1]);
        }
        else
        {
            if(-index - 2 < 0)
            {
                offset = 0;
            }
            else
            {
                offset = (Long)secondaryGlobalIndex.get
                (secondaryGlobalIndexKeySet[-index - 2]);
            }
        }

        globalIndexFile.seek(offset);

        String string = "", string1 = "", documentID = "";
        boolean c = false;
        int count = 0;
        double termFrequency;

        //                                while(count < 1000)
        while(true)
        {
            string = globalIndexFile.readLine();

            if(string == null)
                break;
        }
    }
}
```

```
count++;

if(queryWordSplit[1].equals(string.split(":")[0]))
{
    c = true;
    break;
}
}

if(c)
{
    string1 = string.split(":")[1];
    Matcher matcher = pattern.matcher(string1);

    while(matcher.find())
    {
        documentID = matcher.group(1);
        termFrequency = Double.parseDouble(matcher.group(2));

        if(documentsRetrieved.containsKey(documentID))
        {
            documentsRetrieved.put(documentID,
            (Double)documentsRetrieved.get(documentID) +
            termFrequency);
        }
        else
        {
            documentsRetrieved.put(documentID, termFrequency);
        }
    }
}
```

Result And Conclusion

On the successful completion of our final year project there are many things that we have learnt. We have achieved to complete successfully most of our objectives in a very time and phase oriented manner. At the end of the day we are able to draw many conclusions from our effort. The crawler is able to harvest data from particular web page and store it in the page dump. The data from the web pages stored in the page dump has been successfully parsed and stored in indexes which makes the searching for a particular word very easy. When a query is put in the search engine it searches for the word in the stored index and looks for the particular webpage where the word can be found. After this it provides a list of pages to which a particular query may be connected to. The program is able to produce results though not optimized but in a rough fashion. It is not possible to point which is the correct answer for the query made by the user because it is not an intelligent search engine. So its upto the user to decide which result is of worth to him and which is not. Though many different ranking techniques are available for usage but none of them is perfect. Each technique has got its own set of limitations and advantages.

Literature and references

- **Sergey Brin and Lawrence Page [1]** - The Anatomy of a Large-Scale Hyper textual web search engine
*Computer Science Department, Stanford University,
Stanford, CA 94305, USA (August 7,2000)*
- **Jungoo cho [2]** – Crawling the web.
department of computer science and the committee
on graduate studies of Stanford university
(November 2001)
- **Thom blum [3]** – web crawling with help of java. (January 1998)
- **Gautam Pant, Padmini Srinivasan, and Filippo Menczer [4]** – *crawling the web*
Department of Management Sciences , School of Library and Information Science ,The
University of Iowa, Iowa City IA 52242, US
- **Andrew mcallum [5]** - inverted index and Efficient Clustering of High Dimensional
Data Sets with Application to Reference Matching. *zWhizBang! Labs Research 4616*
Henry Street Pittsburgh, PA USA
- **Web indexing tools [6]**- Kevin broccoli (September- 1999)