Full-Text Indexing for Heritrix

A Writing Project Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment of the

Requirements for the

Degree Master of Computer Science

By

Darshan Karia

Spring 2012

SAN JOSÉ STATE UNIVERSITY

The Undersigned Writing Project Committee Approves the Writing Project Titled

Full-Text Indexing for Heritrix

By

Darshan Karia

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

____ ____________

Dr. Chris Pollett, Department of Computer Science Date

____ ____________

Dr. Mark Stamp, Department of Computer Science Date

____ ____________

Dr. Jeff Smith, Department of Computer Science Date

ACKNOWLEDGEMENTS

I want to thank Dr. Chris Pollett for his guidance and encouragement throughout my project. I would like to thank my committee members Dr. Mark Stamp and Dr. Jeff Smith for their valuable inputs on my project in terms of suggestion to improve my projects. I want to thank my friends here and family members back in my home country for providing me mental support to finish the project.

Abstract

It is useful to create personalized web crawls, and search through them later on to see the archived content and compare it with current content to see the difference and evolution of that portion of web. It is also useful for searching through the portion of web you are interested in an offline mode without need of going online.

To accomplish that, in this project I focus towards indexing of the archive (ARC) files generated by an open source web-crawler named Heritrix. I developed a Java module to perform indexing on these archive files. I used large set of archive files crawled by Heritrix and tested indexing performance of the module. I also benchmarked performance for my indexer and compare these results with various other indexers.

The index alone is not of much use until we can use it to search through archives and get search results. To accomplish that, I developed a JSP module using an interface for reading archive files to provide search results. As a whole, when combined with Heritrix, this project can be used to perform personalized crawls, store archive of the crawl, index the archives, and search through those archives.

Table of Contents

List of Tables

List of Figures

1. Introduction

The Internet Archive [1] project is an open source project intended to store archive crawls of the web. It uses the Heritrix web-crawler [2] to crawl the web and stores web-pages downloaded in archive files. It also tracks the dates when these archives were retrieved from the web and uses that to provide date wise lookup on domains through the "Wayback Machine" [3]. Their mission is to preserve a historical archive of World Wide Web.

This project creates inverted index for archive crawls and facilitates user to search through these archive crawls. It is based on a similar concept established by Wayback Machine, but instead of the domain-based search provided by Wayback Machine, this project provides keywords-based search to a user. A user can perform personalized web crawls limited to domains of interest using Heritrix and store them in compressed archive format. Now a user can use the Indexing Module of this project to index the crawled documents. During indexing, this project calculates a BM25F [4] score for all query words to get more relevant results for user queries while searching. Once indexing is performed and the index is ready, a user can set the Searching Module to point to the newly developed index along with original archive files and search through the crawled documents and to obtain the desired information.

2. Heritrix

In this section, we briefly introduce Heritrix and provide information about where it can be obtained from and how to perform a basic crawl.

Heritrix is an open-source web crawler project used by Internet Archive. It is designed to respect robots.txt exclusion directives and META robots tags. That means, it will not crawl web sites that exclude it using these directives. Heritrix uses an adaptive crawl method to avoid too much traffic on web servers it is crawling.

We now turn to describing how to download and do a basic crawl with Heritrix. Heritrix project is hosted on a sourceforge web-site. It can be downloaded as a binary from the following url:

http://sourceforge.net/projects/archive-crawler/files/

To run Heritrix from the binaries, go to the heritrix-1.14.4/bin directory and type the following command to launch web-interface for heritrix: heritrix --admin=LOGIN:PASSWORD. Here, LOGIN is “admin” and PASSWORD is “letmein”.

If we are running Heritrix from Eclipse as a Java project, we need to set up three parameters in the run configuration before running it. We set the main class of the project to “org.archive.crawler.Heritrix”. We set the program arguments to provide username and password “-a admin:letmein”. We set the VM Arguments “-Dheritrix.development -Xmx512M”.

After finishing the configurations mentioned above we can launch the Heritrix web based UI. To do this, launch any Web Browser and go to the http://127.0.0.1:8080/ web address to access web based user interface for Heritrix [5].

Before we start crawling the web after our first run on Heritrix, we need to configure a few default settings in the crawl profile. We need to provide values for http-headers like “user-agent” by replacing “PROJECT_URL_HERE” with valid project url and replace “from” with valid e-mail address of the person using the crawler. Also, we need to add new user-agent to “user-agents” under the “robots-honoring-policy” section of the profile. Finally, we need to provide seed sites for the crawl under section “Seeds”.

3. Design and Implementation

We are now ready to describe how we designed each component for Heritrix and what was involved in implementing it. To begin Heritrix generates archive files. We will pass these to our Indexing module to generate an inverted index. This inverted index will later be used by our Searching module to search through the archives. Let's briefly look at the roles of the Indexing Module and The Searching module:

1) The Indexing Module

   It needs to iterate through archive files and construct an inverted index for html files inside the archive. It also needs to calculate BM25F scores for relevance of search results based on query terms.

2) The Searching Module

   It provides a Front-end user interface for users to easily search through the index created in first indexing part and get desired results.

Both of above mentioned parts of project are explained in detail in following sections:

3.1 The Indexing Module

The steps to index include but are not limited to iterating through archive files, parsing of html documents, stemming the words, hashing words and docs to create word_id and doc_id, creating a word dictionary, creating a document dictionary, creating the inverted index, and pre-calculating parts of the BM25F score for each word in the word dictionary. Figure 1 shows the activity diagram for Indexing while Figure 3 shows the same workflow in terms of Java classes and shows the execution flow. Figure 2 represents a class diagram of the Indexing Module. All sub-modules of the indexing module are explained individually through Section 3.1.1 - 3.1.7.

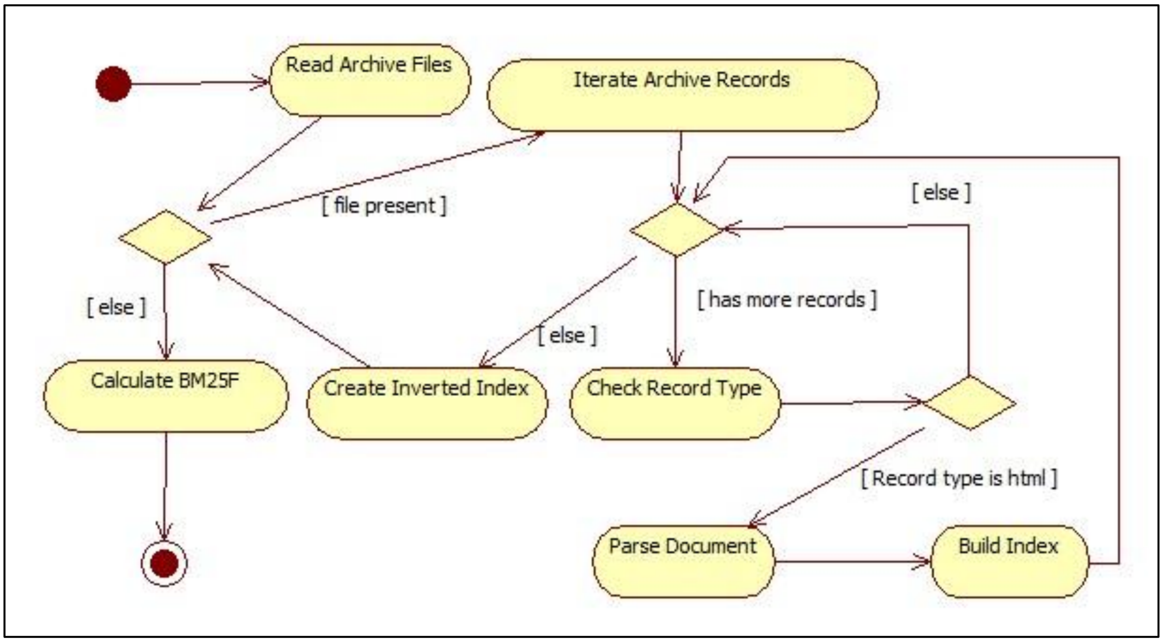


Figure 1: Activity Diagram for Indexing

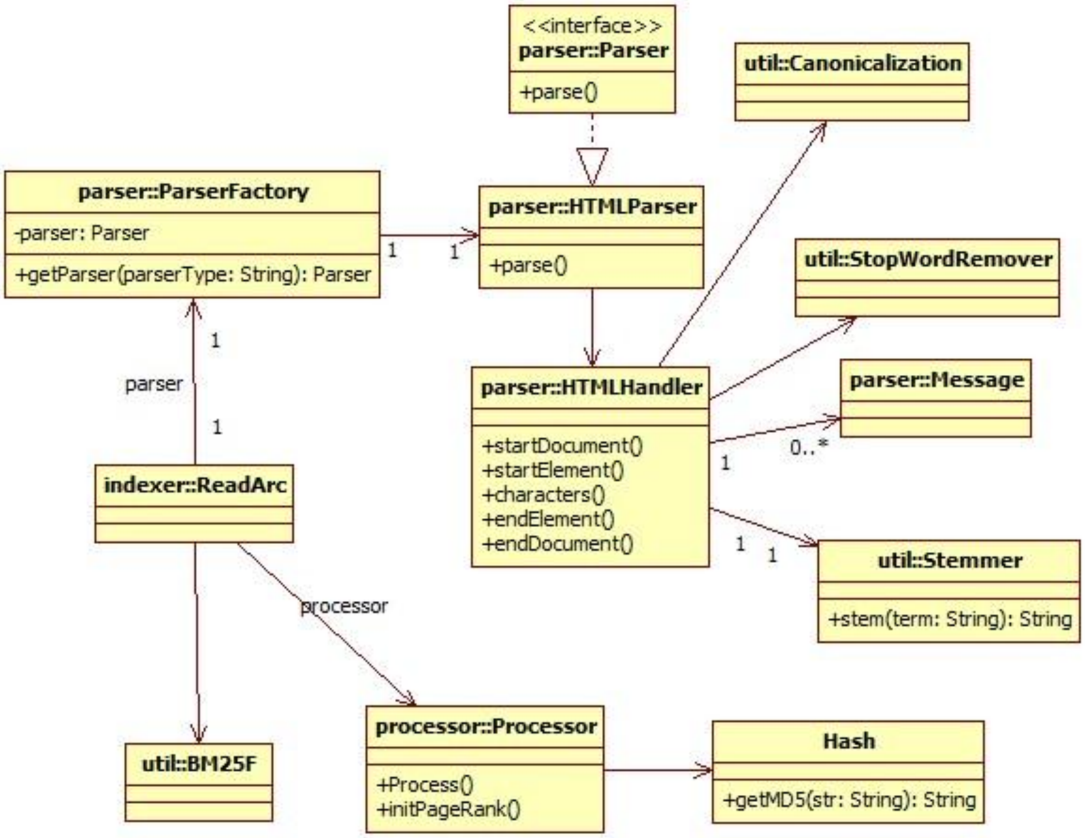


Figure 2: Class Diagram for Indexing

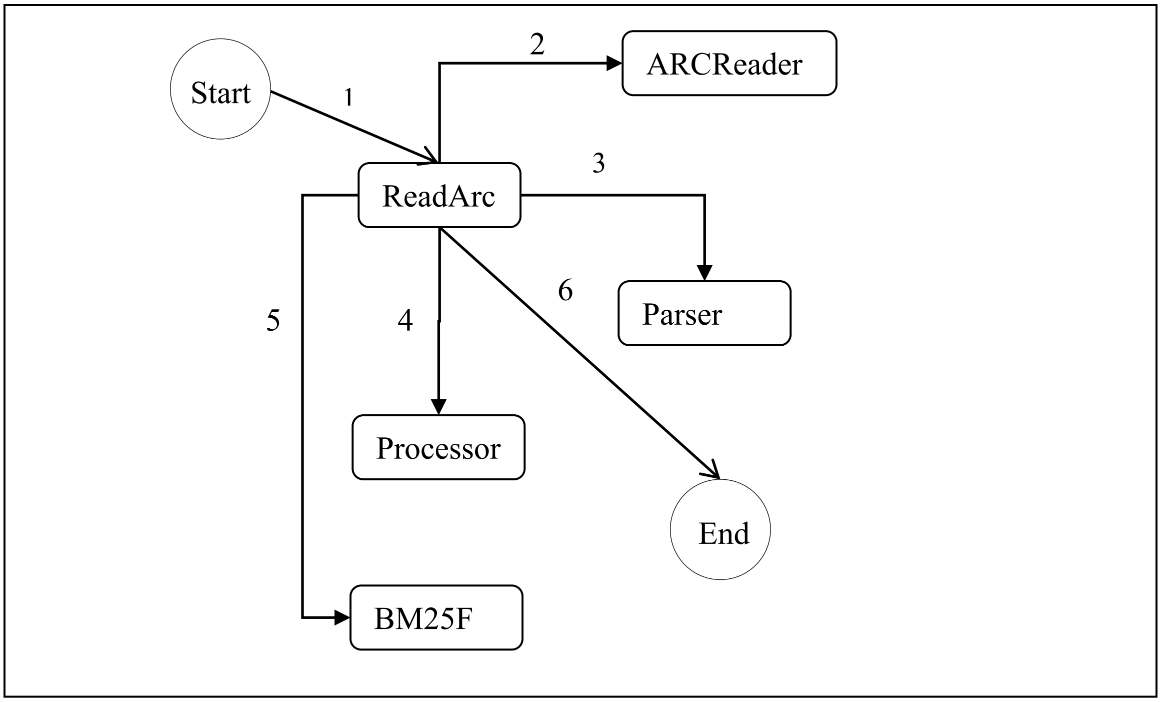


Figure 3: Workflow for Indexing

3.1.1 Iterating through Archive files

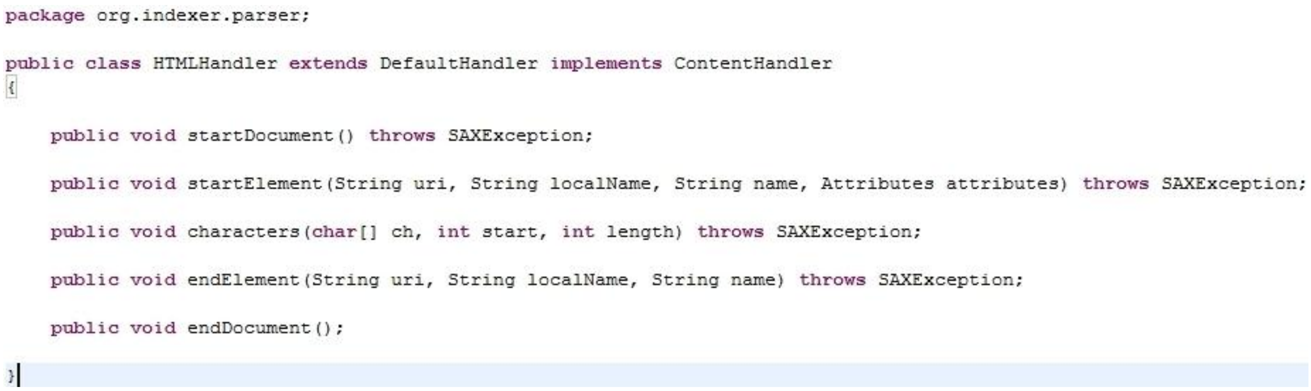
The Archive files created by Heritrix are in a GZip compressed format called ARC [6]. We need to read through GZip files to obtain serialized records from the archive file. Heritrix 1.14.4 has an implementation of org.archive.io.arc.ARCReader class providing us with methods to access records stored in Archive files in random fashion [7]. In order to describe how we iterate through ARC files, it is necessary to describe API of Heritrix [8] a little.

In Heritrix, ARCReaderFactory gives an instance of an ARCReader referring to the provided archive file. One can use its iterator() method to iterate through all records in the file. Each record is of type org.archive.io.ArchiveRecord. An ArchiveRecord contains header information like the offset of that record in the archive file, and the file type. If the file type is HTML, this module dumps the file into a ByteOutputStream and parses the current record using the parser; otherwise it moves to find next record. It does this for all files within the designated arc directory.

3.1.2 Parsing HTML Documents

HTML documents can be parsed in two ways, using DOM Parser or SAX Parser. SAX refers to "Simple API for XML" [9]. In SAX, XML documents are parsed in sequential way and it uses event-based model for parsing. The SAX parser processes XML documents state-dependently. A DOM parser needs to build tree structure in memory based on whole document. A SAX parser is generally more memory efficient compared to DOM parser. A SAX parser, being event-based, only needs to store data related to current event and unclosed tags for reference of close tags. For this project, I have used the SAX Parser implemented in Java [10].

A SAX Parser needs a handler to be defined indicating what to do on certain events like the start of document, the start of element, the end of element, etc. Java has a DefaultHandler class containing all these mentioned methods. I have extended the handler and created my own HTMLHandler class and overridden the methods which I required in order to store the terms in the documents and information about them. The class skeleton looks like the following:



```
package org.indexer.parser;

public class HTMLHandler extends DefaultHandler implements ContentHandler
{

    public void startDocument() throws SAXException;

    public void startElement(String uri, String localName, String name, Attributes attributes) throws SAXException;

    public void characters(char[] ch, int start, int length) throws SAXException;

    public void endElement(String uri, String localName, String name) throws SAXException;

    public void endDocument();

}
```

Figure 4: HTMLHandler Methods

In the preceding figure, the methods are functioning as the names suggest. The explanation of each method is given in Table 1.

Table 1: HTMLHandler Methods

Method	Use
startDocument	Called initially when document parsing is started
startElement	Called each time a start tag is found in document
characters	Called whenever Character Data is encountered in the document
endElement	Called each time an end tag is found in document
endDocument	Called at the end when parser has finished scanning through document

3.1.3 Stemmer

Stemming is a process of reducing the derived words to their root forms [11]. E.g. ‘stemmer’, ‘stemmed’, ‘stemming’, ‘stem’, all refers to same root ‘stem’. This helps in consideration of words with same root in search result.

In this project, while parsing html web-pages, stemming is performed on the words before they are passed to a hashing module and stored into the index. For stemming, the open-source Porter Stemmer developed in a language named Snowball was used.

Along with stemming, stop-word removal is also performed before I store the words in the word dictionary. **Stop-words** are the words which have little or no lexical meaning and are used for grammar. A **Posting list** is a mapping from a word to all the documents in which it appears. For more information on these concepts refer to Section 3.1.6.2. A Posting list for stop-words can grow very large based on how common it is in the documents. Very large posting lists can slow down index performance and removing them does not affect search results too much. A stop-word remover will remove all such words to improve indexing performance.

3.1.4 Hashing

MD5 hashing is performed on words and document urls to obtain strings of the same length for word_id and doc_id. A class was developed which would perform MD5 hashing based on whether a flag is set or not. If the flag is set, it will perform the hashing of the given string and generate an MD5 hash of that to be used as Id. The first 8 characters from the generated MD5 hash served as an id for the given word or document for its identification. After getting the hashed value of the given string, they are stored them in an object of Message class. One object of Message class is used for each document and a list of message objects to process them together after all the HTML documents from current archive file have been parsed.

3.1.5 Word Dictionary

A Word Dictionary uses the word and the first 8 characters of MD5 hashing of the word as described in Section 3.1.4. It consists of a mapping from the word_id to the string word. The word dictionary and inverted index, both are stored using an on-disk Binary Tree data structure implemented in JDBM Project [12]. A Word Dictionary looks like following:

word_id -> Word

3.1.6 Inverted Index

An index refers to a mapping of the document to a set of words appearing in the document. In contrast to that, an inverted index refers to a mapping of the word to set of documents where the word appears. The Inverted Index [13] consists of two basic components, Document Dictionary and Posting List. They are explained in following sections:

3.1.6.1 Document Dictionary

The Document Dictionary contains details for each document in the collection and has a mapping from doc_id to the details of the document. Table 2 below gives the summary of the fields, gives brief information about each of them, and significance of them in this project:

Table 2: Document Dictionary Structure

Field	Info	Used for

url	Actual Link to the document	Providing live link in search result
file	Archive file where it is stored	Locating archive file in order to load cached result from archive file
offset	Offset at which it is stored in archive file	Locating record referring to the document within archive file
length	Length of the record in archive file	Determining length of the document
title	Title of the Document if present	Title of the search result
desc	Description found in meta tag of the page to generate snippet of the result	Snippet of the search result

3.1.6.2 Posting List

A Posting List contains information about occurrence of words within collection of documents. It has a mapping from the word to the collection of documents where it appears. This is an important index structure referred to while processing a search query. In addition to a posting list, detailed information about each document appearing in result is retrieved from the Document Dictionary discussed in Section 3.1.6.1. Then for each document corresponding to search term, details are retrieved from Document Dictionary. Details regarding Posting List are explained in Table 3 below:

Table 3: Posting List Structure

Field	Info
word_id	MD5 hashed word to uniquely identify word

doc_id	MD5 hashed document link to uniquely identify document
element	Html element where the word occurred in document
count	Frequency of the word in the given document
length	Length of the html element where the word appeared in the document

3.1.7 BM25F Score

BM25F [14] is an extension to original Okapi BM25 [4] ranking function. Okapi BM25 is used to rank plain text documents, while BM25F is used to rank more structured documents like an XML or an HTML file. The BM25F scoring function gives a relevance score for a document for the query terms based on the importance of the element in which the terms appear.

In order to reduce the amount of calculations required to be done in the Searching Module, the Relevance Scores for the collection of documents based on word-dictionary is pre-calculated in Indexing Module itself. So, the Searching Module just needs to add the scores of the document for all query terms and get total BM25F score.

The following sub-sections provide the steps for calculating BM25F scores for a document based on given query terms:

3.1.7.1 Weight

The weight of the term ‘t’ in the document ‘d’ can be calculated as shown in the following equation:

$$weight(t,d) = \sum_{c\ in\ d} \frac{occurs_{t,c}^{d}\ .boost_c}{\left((1-b_c)+b_c\cdot\frac{l_c}{avl_c}\right)}$$

The terms appearing in above equation are explained in Table 4,

Table 4: Explanation for terms in BM25F Weight function

Term	Explanation
$occurs_{t,c}^{d}$	Term frequency of term ‘t’ in document ‘d’ in field ‘c’
$boost_c$	Boost factor applied to field ‘c’
b_c	Constant related to the field length
l_c	Field length
avl_c	Average length for the field ‘c’

3.1.7.2 Inverse Document Frequency

Inverse Document Frequency weight of term ‘t’ can be derived using following function:

$$idf(t) = \log\frac{N-df(t)+0.5}{df(t)+0.5}$$

The terms appearing in above equation are explained in Table 5,

Table 5: Explanation of terms in BM25F idf function

Term	Explanation
N	Number of documents in the collection
df(t)	Number of documents where term 't' appears (Document Frequency)

3.1.7.3 Final Scoring

Finally, we use the Weight and Inverse Document Frequency calculated for each query term from Section 3.1.7.1 and Section **Error! Reference source not found.** respectively and use those values in the equation given below to calculate final BM25F score of the document for given query:

$$R(q,d) = \sum_{t\ in\ q} \frac{weight(t,d)}{k_1 + weight(t,d)} \, . \, idf(t)$$

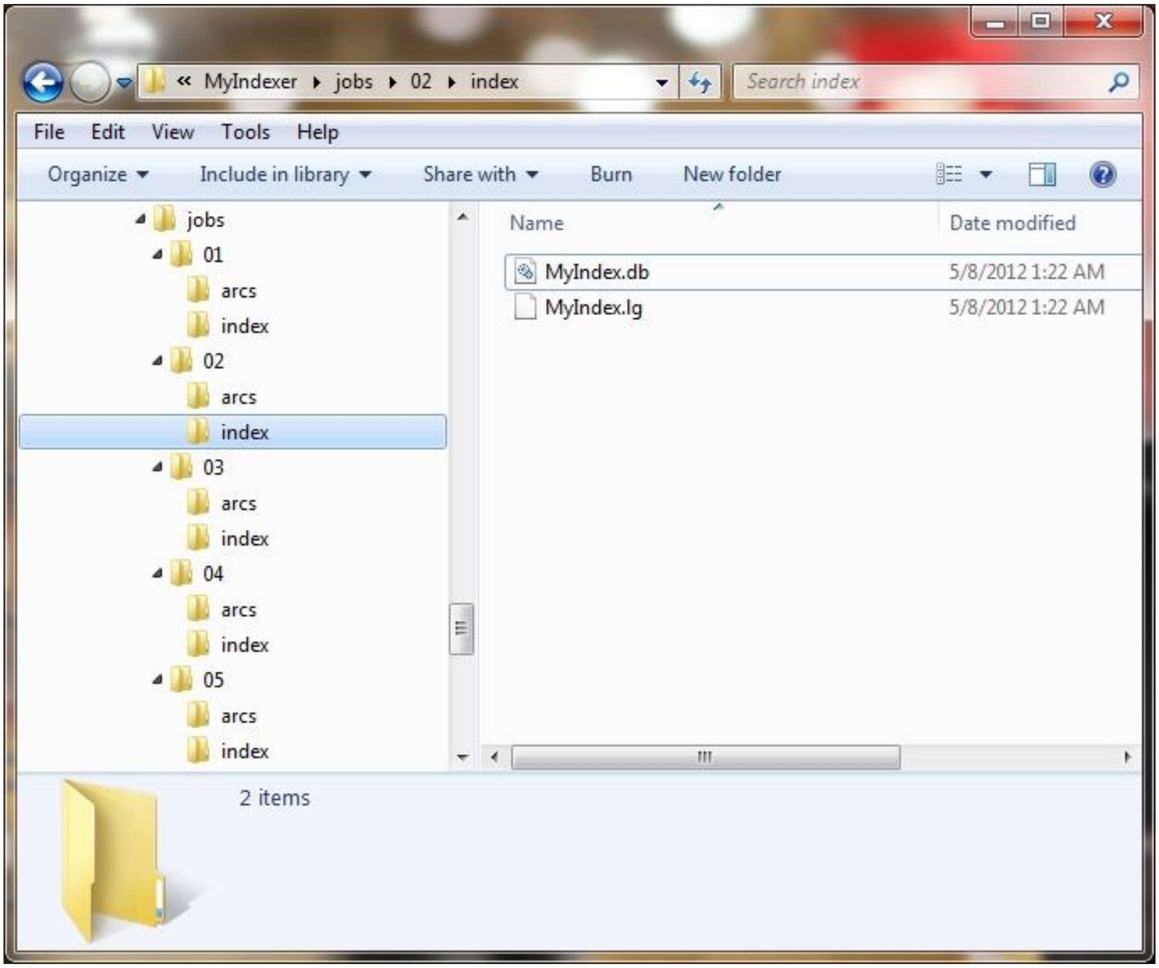
The terms appearing in above equation are explained in Table 6,

Table 6: Explanation of terms in BM25F Rank function

Term	Explanation
weight(t,d)	Weight of term 't' over all fields for document 'd'
k_1	Free parameter
idf(t)	Inverse Document Frequency term 't' over collection of documents

3.1.8 Index Folder Structure

The default directory structure used by the indexing module to organize indexing jobs containing archive files and index files is shown in Figure 5.



- jobs
  - 01
    - arcs
    - index
  - 02
    - arcs
    - index
  - 03
    - arcs
    - index
  - 04
    - arcs
    - index
  - 05
    - arcs
    - index

Name	Date modified
MyIndex.db	5/8/2012 1:22 AM
MyIndex.lg	5/8/2012 1:22 AM

Figure 5: Directory Structure for organizing Indexing Jobs

In Figure 5, the “01”, “02”, “03”, “04”, and “05” directories under “jobs” directory refer to job numbers. The “arcs” directory under each job directory contains the archive files for the particular indexing job. The“index” directory under job directory contains the generated index files for corresponding archive files for the job.

3.2 The Searching Module

An index alone is not useful unless there is a way to use this index for searching through archives for documents. To achieve that, the Searching Module was developed. It is a Java-based Web-application. Along with different modules of Searching Module, it uses some support classes from the indexing module to read index format and archive files. In the Searching Module, there is one search page in which you can enter your query and retrieve results for the same from your archive collection using its index.

The different Use case scenarios for the Searching Module are as shown in Figure 6.

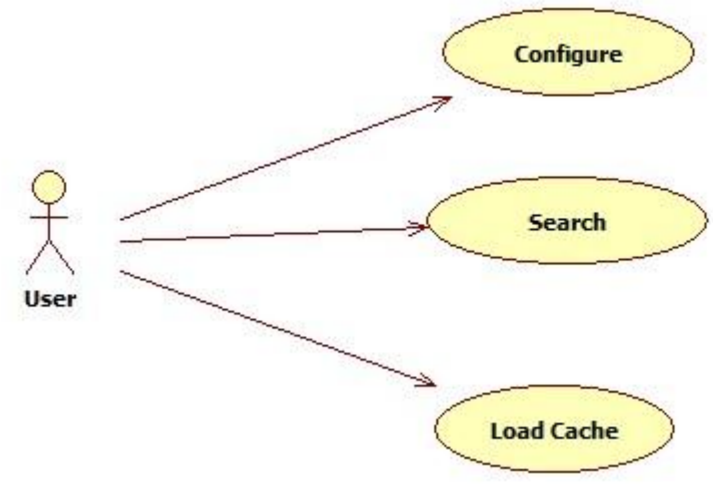


Figure 6: User Case Diagram for Searching

The Searching Module is divided in mainly three JSP modules:

1. The Query Module

   This module is used for entering search query and displaying results based on that.

2. The LoadCache Module

   This module is used for loading cached version of the document from archive file.

3. The Settings Module

   This module provides various settings related to searching.

All three modules are explained in detail below in Sections 3.2.1 - 3.2.3:

3.2.1 The Query Module

This is the main searching module. It consists of index.jsp, IndexReader class, Record class and index itself. Main function of this module is to provide results based on user entered search query.

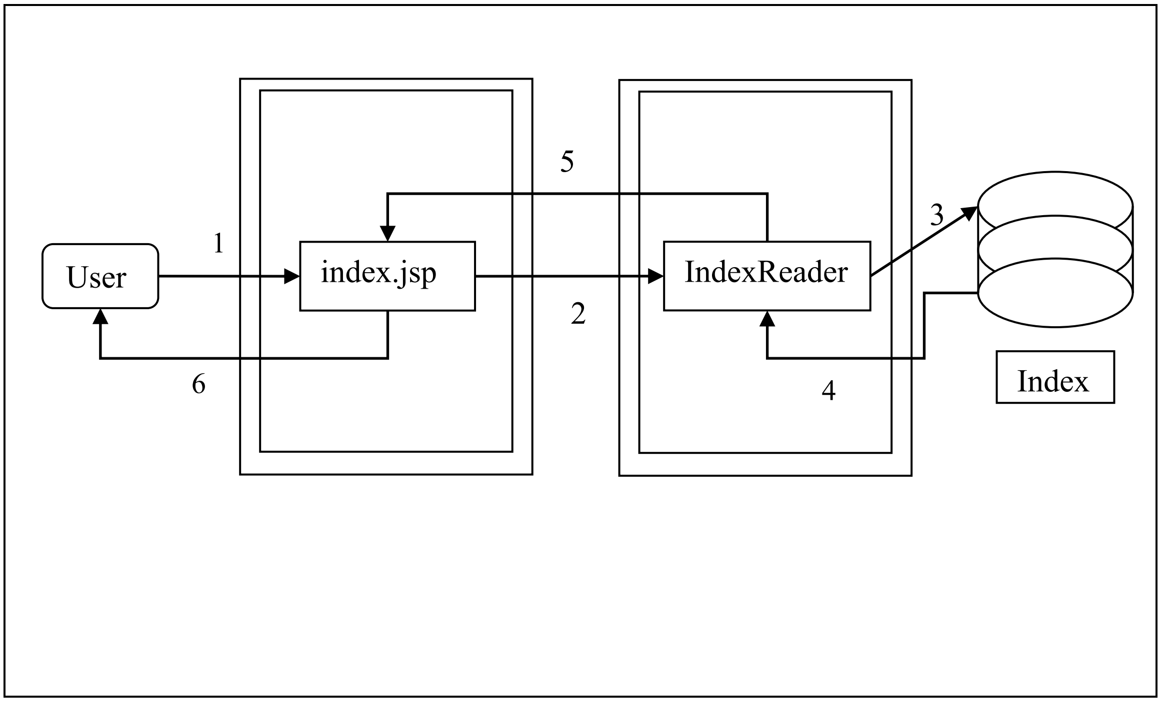


Figure 7: The Query Module Internals

The workflow of the Query Module is shown in the Figure 7. In the figure, numbers from 1 to 6 represent steps of workflow and they are briefly explained below:

1) The user inputs the query in search box of index.jsp and submits the request.

2) The index.jsp divides the query into list of terms and submits it to IndexReader class.

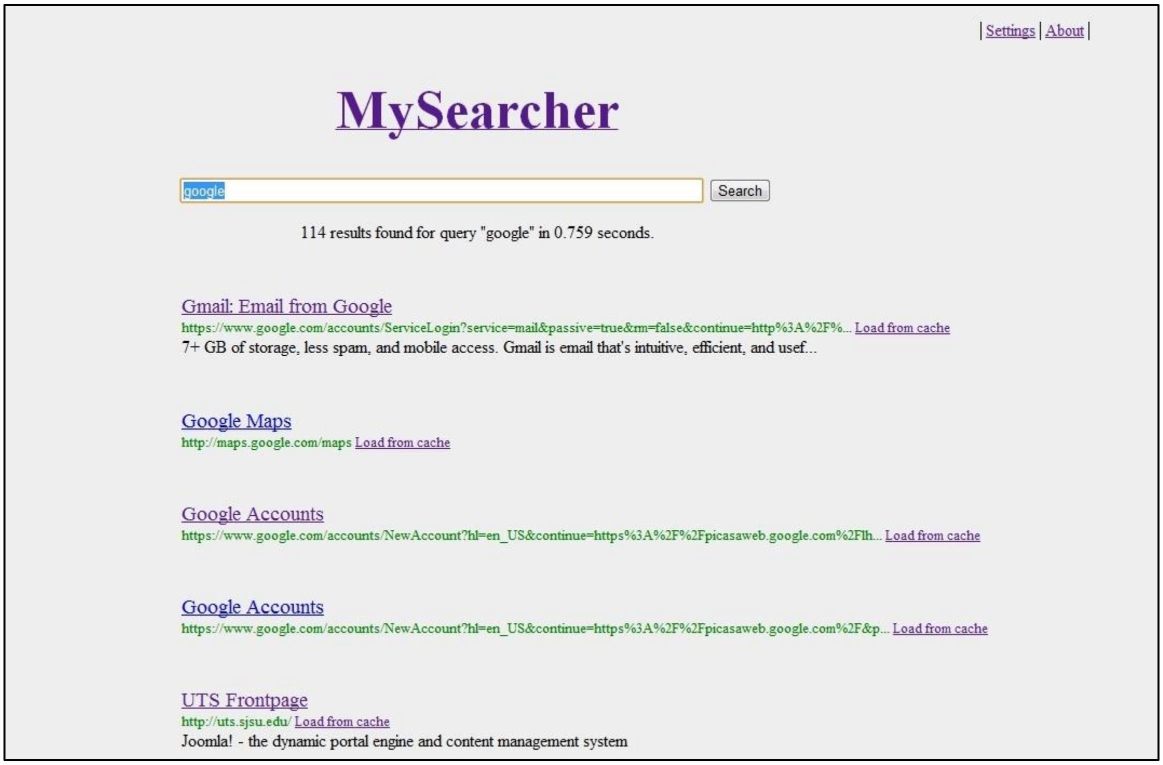
3) The IndexReader class goes to the currently pointed index and requests for list of documents matching the terms in the query from Posting List. It also retrieves detail for all such document from Document Dictionary.

4) Index provides the requested results back to IndexReader.

5) The IndexReader class builds the list of Record objects by using the details obtained from index and combining BM25F scores for each document where search query term appears. At the end, it sorts the List of Record objects in descending order of BM25F score. This List of Record objects is sent to index.jsp file for user to see.

   Here, every Record object contains important information for search result like Title, the link to the original document, the information needed to load cached version of document and snippet of the document content when available.

6) The index.jsp file displays the formatted result for the user query on the page. Sample of search results for query “google” showing records can be seen in Figure 8.



114 results found for query "google" in 0.759 seconds.

Gmail: Email from Google
https://www.google.com/accounts/ServiceLogin?service=mail&passive=true&rm=false&continue=http%3A%2F%... Load from cache
7+ GB of storage, less spam, and mobile access. Gmail is email that's intuitive, efficient, and usef...

Google Maps
http://maps.google.com/maps Load from cache

Google Accounts
https://www.google.com/accounts/NewAccount?hl=en_US&continue=https%3A%2F%2Fpicasaweb.google.com%2Flh... Load from cache

Google Accounts
https://www.google.com/accounts/NewAccount?hl=en_US&continue=https%3A%2F%2Fpicasaweb.google.com%2F&p... Load from cache

UTS Frontpage
http://uts.sjsu.edu/ Load from cache
Joomla! - the dynamic portal engine and content management system

Figure 8: Search Results

3.2.2 The Load Cache Module

This module takes request from the Query Module to retrieve the archived version of document based on file name of the archive file containing the document and offset for that result within that archive file. It shows the html document on screen to user. It might lack certain pictures or formatting of the document if the original pictures and cascaded styles sheets linked to the document are no longer available.

Workflow of the Load Cache Module is shown in Figure 9 and it is explained below according to the numbers given in the figure.

1) User selects the document to load from archive by using “load from cache” link for corresponding document.

2) The required details to locate and load the selected document from the archive file are passed as parameters to loadCache.jsp. These details include name of the archive file where it is stored and offset to the record containing document.

3) loadCache.jsp locates the archive file based on the information provided and location of all archive files related to the index provided by config.properties file. Using ARCReader class, loadCache.jsp requests index for the document at in provided archive file, at given offset.

4) Archive file returns the record containing the requested document to loadCache.jsp

5) loadCache.jsp reads that record and returns it to the user.

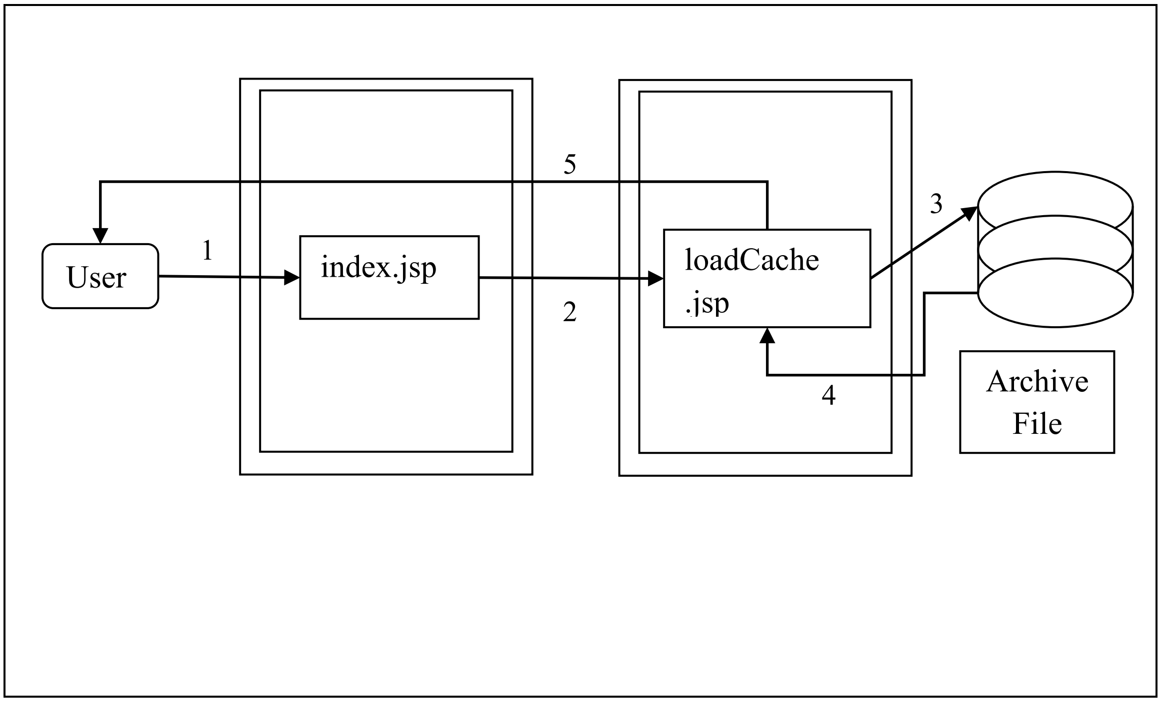
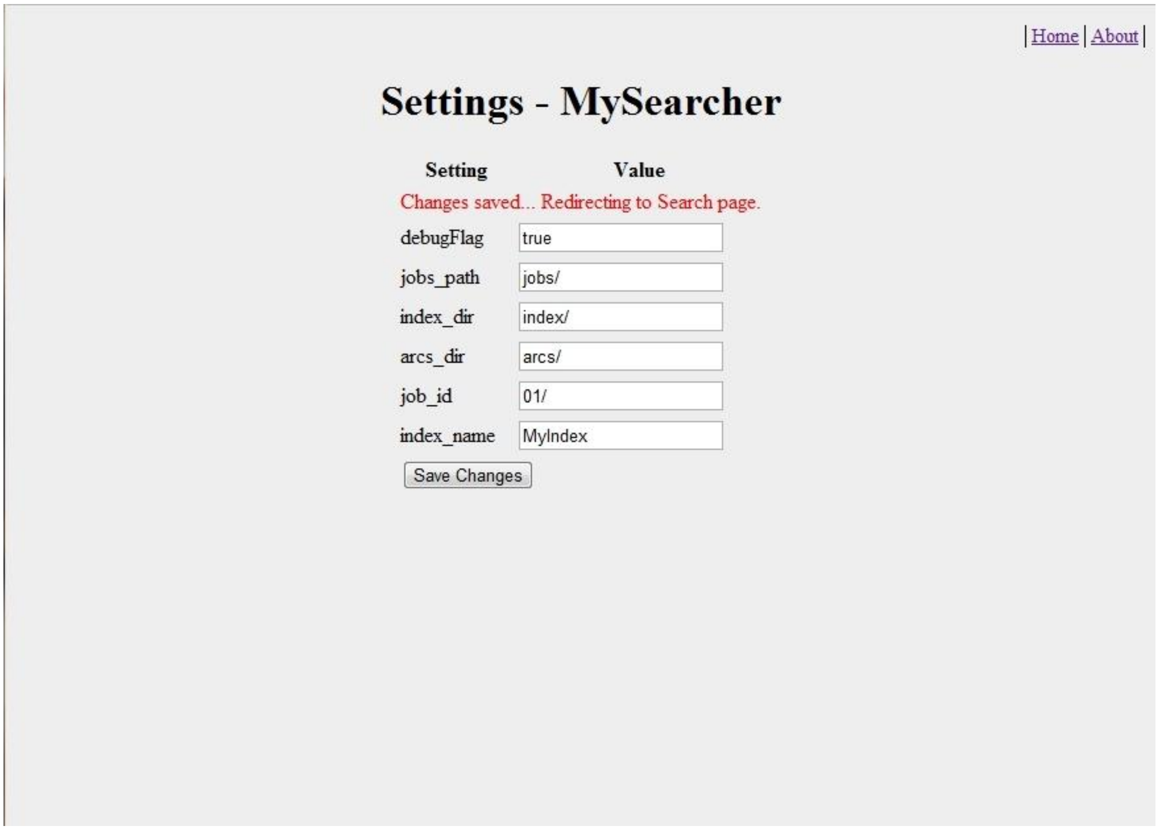


Figure 9: Load Cache Module Working Diagram

3.2.3 The Settings Module

This module allows the user to change various parameters related to the index and archive files. The Query Module and the Load Cache Module refers to the settings saved through this module. This module consists of the settings.jsp and the config.properties file.

In the settings.jsp, the user is provided an option to retrieve or change settings related to indexing and archive files.



Settings - MySearcher

Changes saved... Redirecting to Search page.

Setting	Value
debugFlag	true
jobs_path	jobs/
index_dir	index/
arcs_dir	arcs/
job_id	01/
index_name	MyIndex

Figure 10: Settings Module

Figure 10 shows how the settings on web-page look like. Those settings are explained in Table 7 with their default value. These settings will determine behavior of entire web-application.

Table 7: Settings of MySearcher

Setting	Option	Default Value
jobs_path	Path to jobs folder where archive jobs are stored	Jobs/
job_id	The job id in the jobs_path to refer	01/

arcs_dir	Path to directory containing archive files relative to jobs_path.	Arcs/
index_dir	Path to directory containing the index file relative to jobs_path	Index/
index_name	Name of the index file	MyIndex
debug_flag	Prints debugging information on console	false

Figure 11 shows the workflow of how Settings Module saves configuration file. The steps from the figure are explained below based on the numbers assigned:

1) User opens the settings.jsp file. The settings.jsp file retrieves the key-value pairs from configuration file as label-input pairs on generated html page.
2) The settings.jsp updates the values in the configuration file upon user request by clicking the save button.
3) Updated values are reloaded on the settings.jsp page.
4) A JavaScript function redirects user to index.jsp in 3 seconds after update is performed. This time period gives user chance to see the changes.
5) index.jsp is displayed to the user for him to start searching based on the newly updated configuration.

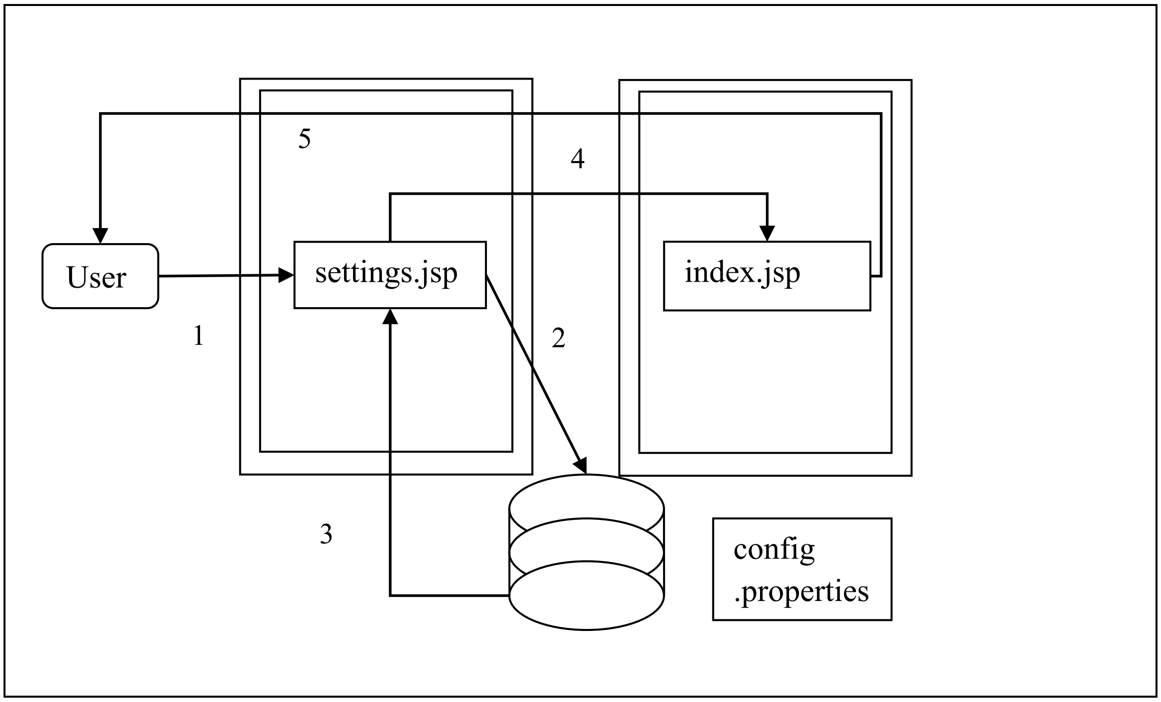


Figure 11: Settings Module Working Diagram (Save file)

4. Testing and Results

4.1 Results for Indexing Test

I tested my Indexing performance by indexing 38 archive files of size 3.49GB. These archive files were retrieved from a Heritrix crawl ran on April 11, 2011 on sjsu.edu domain. Below is part of the result also demonstrating profiling of the indexer:

```
Path: jobs/03/arcs/

Following files are found:


*IAH-20110411213501-00000-130.65.104.126-8080.arc.gz
Phase - I of iterating arc files and parsing html docs started... 0 Seconds
Phase - I of iterating arc files and parsing html docs finished... 15 Seconds and 15
Seconds from start to end
Phase - II of processing messages and creating index started...15 Seconds so far
Phase - II of processing messages and creating index finished...7 Seconds and 22
Seconds from start to end

*IAH-20110411213510-00001-130.65.104.126-8080.arc.gz
Phase - I of iterating arc files and parsing html docs started... 24 Seconds
Phase - I of iterating arc files and parsing html docs finished... 19 Seconds and 44
Seconds from start to end
Phase - II of processing messages and creating index started...44 Seconds so far
Phase - II of processing messages and creating index finished...31 Seconds and 75
Seconds from start to end
                                        .
                                        .
                                        .
*IAH-20110411221922-00037-130.65.104.126-8080.arc.gz
Phase - I of iterating arc files and parsing html docs started... 12912 Seconds
Phase - I of iterating arc files and parsing html docs finished... 0 Seconds and
12912 Seconds from start to end
```

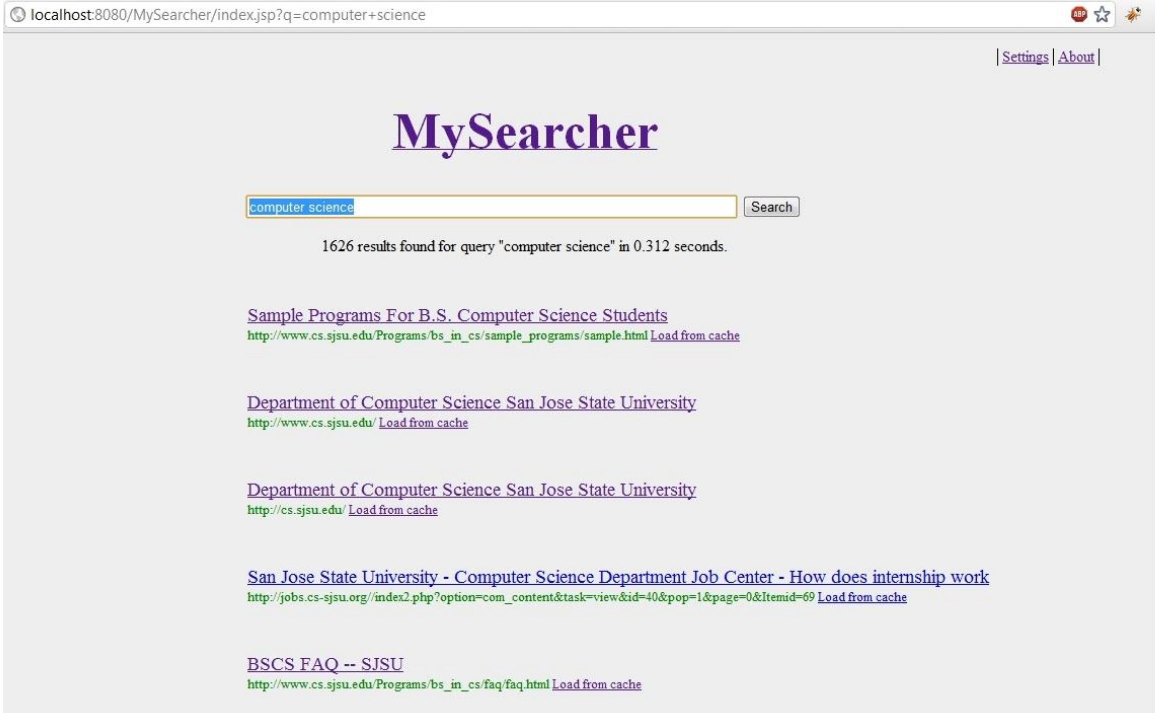
```
Phase - II of processing messages and creating index started...12912 Seconds so far
Phase - II of processing messages and creating index finished...533 Seconds and
13446 Seconds from start to end

Phase - III of pre-calculating BM25F score started...13446 Seconds so far...
Current Working Directory: C:\Users\Darshan\workspace\MyIndexer
invertedIndexTree exists and has 56900 members
docDictionaryTree already exists and has 22479 members
miscTree already exists and has 5 members
Phase - III of pre-calculating BM25F score finished...243 Seconds for this phase
and 13887 Seconds from start to end
24109 html files processed...
```

As you can see above, it took around 13887 seconds to create index for 38 archive files of total size 3.49GB. It retrieved 24109 html documents out of those archive files and created index based on them. Size of created index is roughly ~700MB.

4.2 Results for Searching Test

To test the performance of the Searching Module, I used the index created for the test performed in Section 4.1. It has approximately 25,000 documents indexed on sjsu.edu domain from April 11, 2011. The original archive size is approximately 3.50GB. I used The Settings Module of The Searching Module to use this desired index and set of archives for The Query Module.



localhost:8080/MySearcher/index.jsp?q=computer+science

computer science

1626 results found for query "computer science" in 0.312 seconds.

Sample Programs For B.S. Computer Science Students
http://www.cs.sjsu.edu/Programs/bs_in_cs/sample_programs/sample.html Load from cache

Department of Computer Science San Jose State University
http://www.cs.sjsu.edu/ Load from cache

Department of Computer Science San Jose State University
http://cs.sjsu.edu/ Load from cache

San Jose State University - Computer Science Department Job Center - How does internship work
http://jobs.cs-sjsu.org//index2.php?option=com_content&task=view&id=40&pop=1&page=0&Itemid=69 Load from cache

BSCS FAQ -- SJSU
http://www.cs.sjsu.edu/Programs/bs_in_cs/faq/faq.html Load from cache

Figure 12: Search Results Test

Speed Test

Now, as shown in Figure 12 performing search for “computer Science” query in the Query Module, it retrieves 1626 results for the query “computer science” in 0.312 seconds.

Table 8 represents more search results benchmarking obtained using the same methodology explained above. This test helps to get an idea of average performance of Searching Module for various queries.

Table 8: Speed Test Results

Query	No. of Results	Time Taken

google	114	0.232
mechanical and aerospace engineering	667	0.32
department of computer science	2782	0.7
clark hall	238	0.424
Search	760	0.236
san jose state university	7561	1.135

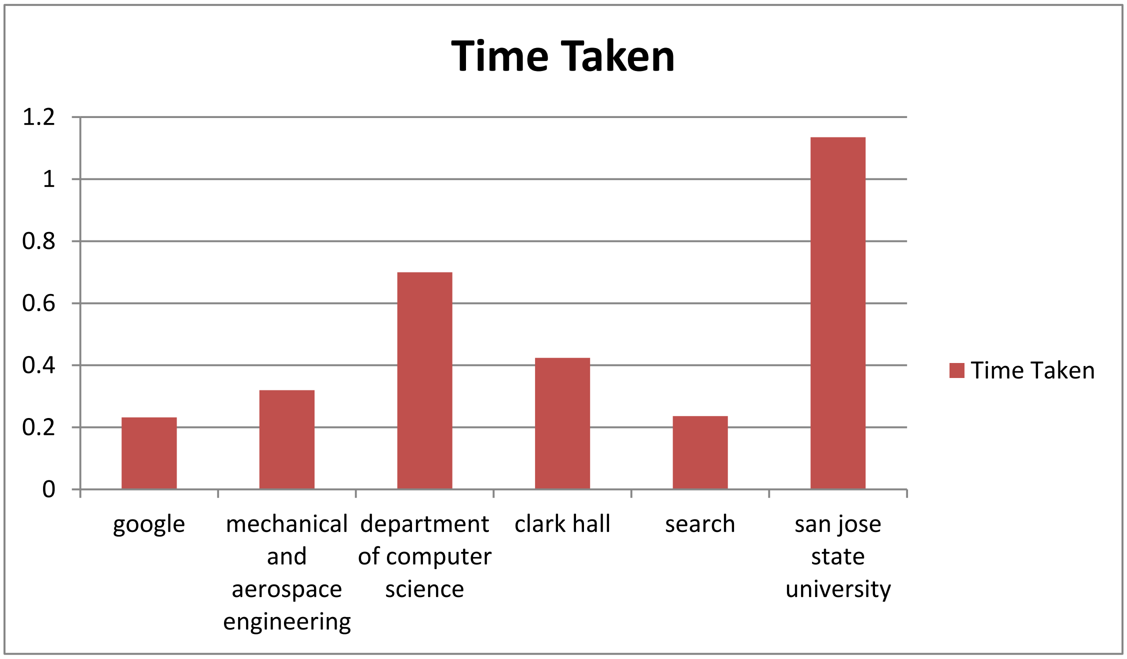


Figure 13: Graph for Time Taken for Query

Figure 13 represents the Query vs. Time Taken graph. Figure 14 represents the Query vs. No. of Results graph. Both of these graphs use data from Table 8.

As we can see in Figure 13, the time taken to retrieve the result generally increases as we introduce more terms in the search query. If we combine this finding with results shown in Figure 14, we can derive that Time taken to retrieve result increases as we introduce more terms in search query or if there are more results to process and sort.

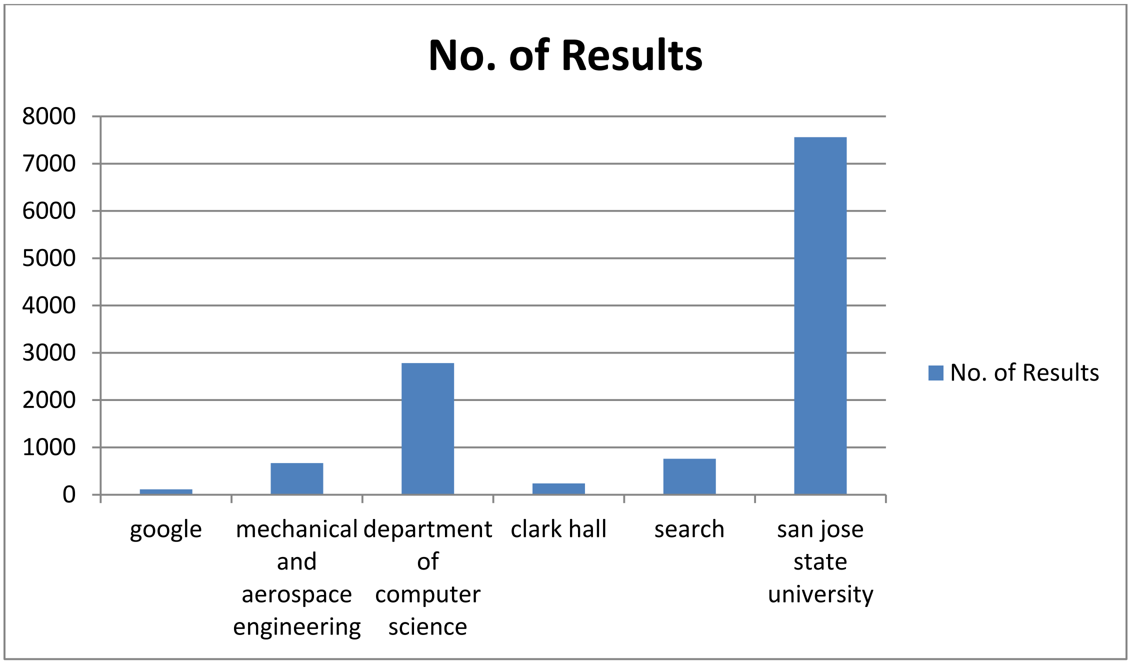


Figure 14: Graph of No. of Results for Different Queries

5. Technical Difficulties & Limitation

“In a day when you don't come across any problems — you can be sure that you are traveling in the wrong path.” – Swami Vivekananda [15]

As the quote suggests, one cannot finish a task without facing any challenges or difficulties. This project was no exception. Some of them are predicted during research work, while others are only to be seen during implementation.

This section contains one such Technical Difficulty and its workaround:

5.1 GZIPInputStream bug

This project processes GZIP files for processing of Archive Records. GZIP file format [16] specifies that single GZIP file can accommodate multiple members provided each members have their own headers specified.

Java’s implementation of GZIPOutputStream allows producing such files, while GZIPInputStream fails to read more than one members from the GZIP file. There is a known bug for this reported to developers of JDK [17]. They tried to fix this in JDK 1.6.0_23 (update 23), but that still has bug in its implementation and old work around by Heritrix Developers to original bug no longer works due to the code change within the method for reading stream through GZIP file.

In order to use ARCReader class provided by Heritrix 1.14.4 for archive reading, the version of Java Runtime Environment has to be 1.6.0_22 or below.

6. Conclusion

Full-text indexing for Heritrix, in combination with Heritrix, provides everything one may need to create searchable archives. To make the archives generated by Heritrix searchable, this project adds the Indexing Module to generate inverted index for the archive files and the Searching Module to provide a web-based search interface facilitating users to perform search through these archives.

Heritrix crawls the web and stores documents in archive file. Our Indexing Module of this project uses these archive files to create Inverted Index for them. The Searching Module provides the web interface for users to search through the archived documents using the inverted index along with archive files. The returned results have links to view archived versions of documents as well as current versions of the documents if they are currently available on web.

This project serves three purposes:

1) It provides keyword based searching compared to Internet Archive project, which provides only domain based searching.

2) It can be used for archiving different versions of the web referring to different times and see the evolution of the web by pointing the search engine to different indexes and comparing the cached documents.

3) It can be used for archiving of important documents in case either they are no longer available online due to certain reason or you do not have internet connection and want to search through those documents.

8. References

[1] "Internet Archive," [Online]. Available: http://archive.org/.

[2] P. Jack, "Heritrix Web Crawler," Internet Archive, 2012. [Online]. Available: http://crawler.archive.org/.

[3] "Internet Archive: Wayback Machine," [Online]. Available: http://archive.org/web/web.php.

[4] "Okapi BM25," [Online]. Available: http://en.wikipedia.org/wiki/Okapi_BM25.

[5] K. Sigurđsson, M. Stack and I. Ranitovic, "Heritrix User Manual," [Online]. Available: http://crawler.archive.org/articles/user_manual/index.html.

[6] M. Burner and B. Kahle, "ARC File Format," 15 September 1996. [Online]. Available: http://archive.org/web/researcher/ArcFileFormat.php.

[7] J. E. Halse, G. Mohr, K. Sigurđsson, M. Stack and P. Jack, "Heritrix Developer Documentation," [Online]. Available: http://crawler.archive.org/articles/developer_manual/index.html.

[8] "Heritrix 1.14.4 - Java Docs," [Online]. Available: http://builds.archive.org:8080/javadoc/heritrix-1.14.4/.

[9] "Simple API for XML - Wikipedia, the free encyclopedia," [Online]. Available: http://en.wikipedia.org/wiki/Simple_API_for_XML.

[10] "SAXParser (Java 2 Platform SE 5.0)," [Online]. Available: http://docs.oracle.com/javase/1.5.0/docs/api/javax/xml/parsers/SAXParser.html.

[11] P. Martin, R. Boulton and M. Andrew, "Snowball Stemmer," [Online]. Available: http://snowball.tartarus.org.

[12] "JDBM Project," [Online]. Available: http://jdbm.sourceforge.net/.

[13] S. Büttcher, C. L. A. Clarke and G. V. Cormack, Information Retrieval: Implementing and Evaluating Search Engines, MIT Press, 2010.

[14] J. P. Iglesias, "Integrating the Probabilistic Model BM25/BM25F into Lucene," [Online]. Available: http://nlp.uned.es/~jperezi/Lucene-BM25/.

[15] "Swami Vivekananda," [Online]. Available: http://en.wikiquote.org/wiki/Swami_Vivekananda.

[16] "GZIP file format specification version 4.3," [Online]. Available: http://www.gzip.org/zlib/rfc-gzip.html#file-format.

[17] "Bug ID: 4691425," 24 May 2002. [Online]. Available: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4691425.

[18] J. He, H. Yan and T. Suel, "Compact full-text indexing of versioned document collections," *Proceedings of the 18th ACM conference on Information and knowledge management,* pp. 415--424, 2009.