Web Search Engine

**Developers:**

Owen Young

Yi Liu

Taylor Atkinson

Blake Lombard

Matthew Clark

**Created For:**

Dr. Michael Galloway

**Table of Contents**

1. Introduction 4

1.1 Purpose 5

1.2 Scope 5

1.3 Definitions, Acronyms, and Abbreviations 5

1.4 References 5

1.5 Overview 5

2. General Description 5

2.1 Product Perspective 5

2.2 Product Functions 5

2.3 User Characteristics 6

2.4 General Constraints 6

2.5 Assumptions and Dependencies 6

3. Specific Requirements 6

3.1 External Interface Requirements 6

3.1.1 User Interfaces 6

3.1.2 Hardware Interfaces 7

3.1.3 Software Interfaces 7

3.1.4 Communications Interfaces 7

3.2 Functional Requirements 8

3.2.1 Website 8

3.2.1.1 Introduction 8

3.2.1.2 Inputs 8

3.2.1.3 Processing 8

3.2.1.4 Outputs 8

3.2.1.5 Error Handling 8

3.2.2 Web Crawler – Parse Page 8

3.2.2.1 Introduction 8

3.2.2.2 Input 8

3.2.2.3 Output 8

3.2.2.4 Error Handling 8

3.2.3 Web Crawler – Recursively Parse a Set Amount of Pages 8

3.2.3.1 Introduction 8

3.2.3.2 Input 8

3.2.3.3 Output 9

3.2.3.4 Error Handling 9

3.2.4 Web Crawler – Scan Until End 9

3.2.4.1 Introduction 9

3.2.4.2 Input 9

3.2.4.3 Output 9

3.2.4.4 Error Handling 9

3.2.5 Web Crawler – Update Scan a Set Amount of Pages 9

3.2.5.1 Introduction 9

3.2.5.2 Input 9

3.2.5.3 Output 9

3.2.5.4 Error Handling 9

3.2.6 Web Crawler – Port Scanning 9

3.2.6.1 Introduction 9

3.2.6.2 Input 10

3.2.6.3 Output 10

3.2.7 Web Crawler – Clear Location from Database 10

3.2.7.1 Introduction 10

3.2.7.2 Input 10

3.2.7.3 Output 10

3.2.7.4 Error Handling 10

3.2.7 Web Crawler – Add Keyword to Database 10

3.2.7.1 Introduction 10

3.2.7.2 Input 10

3.2.7.3 Output 10

3.2.7.4 Error Handling 10

3.3 Use Cases 10

3.3.1 Use Case #1 10

3.3.2 Use Case #2 11

3.4 Classes / Objects 11

3.4.1 Web Crawler: Controller 11

3.4.1.1 Description: 11

3.4.2 Web Crawler: Database Manager 12

3.4.2.1 Description: 12

3.4.2.2 Functions: 12

3.4.3 Web Crawler: Spider 12

3.4.3.1 Description: 12

3.4.3.2 Functions: 12

3.5 Non-Functional Requirements 12

3.5.1 Performance 12

3.5.2 Reliability 12

3.5.3 Availability 12

3.5.4 Security 12

3.5.5 Maintainability 13

3.5.6 Portability 13

3.6 Inverse Requirements 13

3.7 Design Constraints 13

3.8 Logical Database Requirements 13

3.9 Other Requirements 14

4. Analysis Models 14

4.1 Data Flow Diagrams (DFD) 14

5. Planning and Resources 17

6. Risks 18

6.1 System Risks 18

6.2 Feasibility 19

7. Deliverables 19

8. Change Management Process 19

9. Software Performance 19

9.1 Performance Requirements 19

9.1.1 Performance Goals 19

9.1.2 Hardware Configuration 19

9.2 Workload 20

9.2.1 Application Workload 20

9.2.1 System Workload 20

9.3 Component Testing 20

9.3.1 Web Search 20

9.3.2 Port Scanner 20

9.3.3 Web Crawler 20

9.4 Performance Inspection 21

9.4.1 Competitor’s Performance 21

9.4.2 Our Performance 21

9.4.3 Minimum Performance Requirements 21

9.4.4 Customers Expectations 22

10. Design Decisions 22

10.1 Data Storage 22

10.2 Crawler Frequency 22

10.2.1 Web Crawler 22

10.2.2 Port Scanner 22

10.3 Displaying Results 22

11. Conclusion 23

Appendix 24

A.1 Installation Instructions 24

A.1.1 Server Installation 24

A.1.1.1 Required Software 24

A.1.1.2 Server Installation 24

A.1.1.3 Software Installation 24

A.1.2 Software Setup 24

A.1.2.1 Git Setup 24

A.1.2.2 Apache2 Setup 24

A.1.2.3 MySQL Server 25

A.1.2.4 Java 25

A.1.2.5 Website Configuration 25

A.1.2.6 WebCrawler Configuration 25

A.1.3 Running the Software 26

A.1.3.1 Website 26

A.1.3.2 WebCrawler 26

A.1.3.3 Port Scanner 26

A.2 Source Code 26

A.2.1 Database 26

A.2.1.1 locations.sql 26

A.2.1.2 keywords.sql 26

A.2.1.3 siteKeywords.sql 27

A.2.2 Website 27

A.2.2.1 index.htm 27

A.2.2.2 style.css 27

A.2.2.3 /search/index.php 30

A.2.2.4 /search/results.php 30

A.2.3 Web Crawler 33

A.2.3.1 Notes on using the crawler.txt 33

A.2.3.2 Data.java 34

A.2.3.3 DatabaseManager.java 34

A.2.3.4 PageParser.java 40

A.2.3.5 Spider.java 47

A.2.3.6 SpiderThread.java 49

# 1. Introduction

This requirements document illustrates the design and concept of a web search engine for Western Kentucky University hosted domains.

## 1.1 Purpose

This system is being built to offer a new and improved in-house service as opposed to WKU’s current service provided through Google’s custom search. By using an in-house service, we will have more control over style, efficiency, and search relevancy.

## 1.2 Scope

The WKU Search Engine project requires a relatively fast database with equivalently fast database manipulations. This will include table/data additions from the web crawler program as well as user queries from the website. The web crawler will index pages on WKU’s IP range and will include a way to parse web pages, a keyword ranking algorithm, and a way to input website data into the database. The web crawler will parse data in order to look for links to follow in addition to accumulating a list of ranked keywords for that page. The website requires a search box and a way to display the list of websites returned by the database. The search box will require a way to make queries to the database.

## 1.3 Definitions, Acronyms, and Abbreviations

*No definitions, acronyms, abbreviations at the moment*

## 1.4 References

*No references at the moment*

## 1.5 Overview

The rest of this report describes the overall product description, followed by the requirements, analysis models for the project, and change management to the design of the project.

# 2. General Description

## 2.1 Product Perspective

This project will function similarly to Google’s Custom Domain Search, but will be slower due to less resources on the project than Google.

## 2.2 Product Functions

The Project will allow a user to enter queries into a search box, and then return results based on how relevant the keywords were to what is stored on the database. A web crawler will scan IP’s on WKU’s domain in order to find hosting servers to populate the database.

## 2.3 User Characteristics

This project’s users will generally be high school and college students looking for more information on WKU’s website and parent’s looking for information for their children. Due to a wide range of users, the website will need to be easy to navigate as well as be fast to keep users searching for content instead of leaving the site.

## 2.4 General Constraints

Constraints that we have on the project include mainly time and money. There are only approximately four months of development time to complete the project, so we have to keep the scope small and stay on schedule. As for money, the team only has resources for a few computers with no budget for multiple server racks or top of the line mainframe systems.

## 2.5 Assumptions and Dependencies

It is assumed that the team will have access to a desktop computer with the ability to run a Linux server in order to start development.

# 3. Specific Requirements

This section details the specific development requirements to guide the implementation of this web search engine.

## 3.1 External Interface Requirements

### 3.1.1 User Interfaces

The only user interface is the website which allows searching. The site will be broken up into the main site to start searching and the results page that both displays results for the query as well as allows the user to search again. The main page and the results pages are all displayed below.



Figure 1: Main page for website

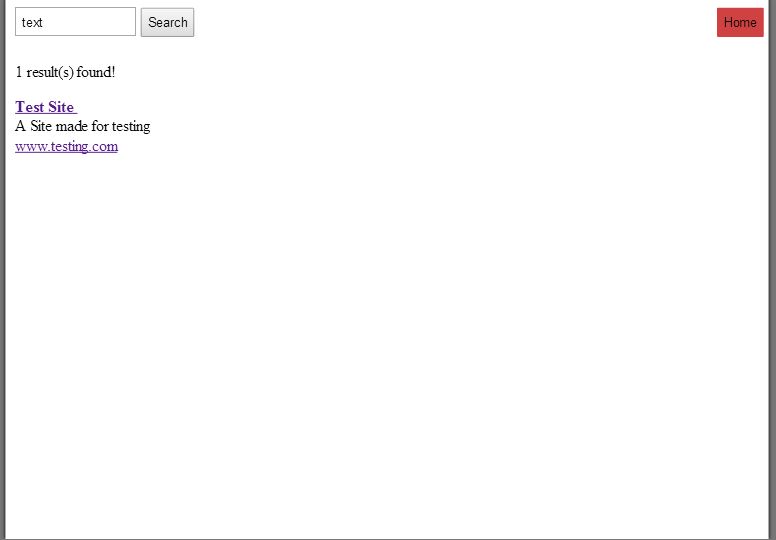


Figure 2: Results page for website; results found



Figure 3: Results page for website; no results found

### 3.1.2 Hardware Interfaces

The hardware interfaces included in this project are a desktop computer with a 500GB hard drive, a dual-core processor, and a broadband modem & Ethernet cord to connect to WKU’s network. In order to increase database and indexing speed, a solid state drive and faster processor would be helpful, but they wouldn’t be required. These details are rough estimates and may change based on users and number of pages/devices needing to be indexed.

### 3.1.3 Software Interfaces

The software interfaces will be broken up into three sections: website, web crawler, and database.

The website will be programmed using HTML, PHP and CSS. HTML will be used for content organization and function. CSS sets the design and look of both the main page as well as the results page. PHP is used to execute MySQL commands as well as organize the returned data from the database to display appropriately on the page. The web server software to host the site will run on an Ubuntu Server machine. The hosting software used is Apache with a PHP server used to process server-side commands.

The web crawler will be implemented in Java and run locally on the main machine. The design on this section consists of a master/slave crawler that gathers data and parses web pages to add to the database. It also includes a port scanner to determine which IPs on WKU’s range of IPs have web servers located on those machines.

The database will be created with MySQL and be stored on the machine where the web server will be hosted.

### 3.1.4 Communications Interfaces

The project will be using MySQL queries to insert and select data from the database. These commands will be executed on the same machine. The web crawler will access servers across the network using HTTP/HTTPS protocols.

## 3.2 Functional Requirements

### 3.2.1 Website

#### 3.2.1.1 Introduction

The website is how the user interacts with the system as a whole. This allows the user to type keywords and query the database for websites.

#### 3.2.1.2 Inputs

The user should be able to input series of words that they wish to find pages about.

#### 3.2.1.3 Processing

The website will parse each keyword the user searched for and query the database for the word. For every word that was found, the results will be organized by the weights of each word combined. When that calculation is finished, the website will then take each webId from the database and match it to the website’s details.

#### 3.2.1.4 Outputs

The websites will be laid in rows with each website name on top, followed by the description, then the URL or IP will be on the bottom.

#### 3.2.1.5 Error Handling

The website will handle no results found by telling the user to use a less specific query. The system will also have limits in place to eliminate buffer overflow. In addition to these, the website will have input validation against SQL injection attacks.

### 3.2.2 Web Crawler – Parse Page

#### 3.2.2.1 Introduction

Gathers the source of a web interface then scans and collects all the text and links on the page.

#### 3.2.2.2 Input

The location of the page (URL IP)

#### 3.2.2.3 Output

Gives the links, text, page metadata, and the keywords and their calculated weights

#### 3.*2.2.4 Error Handling*

The page listener times out: The parser will return null for all output elements.

The page contains no text: With no keywords, the page is not useful to index, so the parser will return null for all output elements.

### 3.2.3 Web Crawler – Recursively Parse a Set Amount of Pages

#### 3.2.3.1 Introduction

The crawler creates a spider that will transverse through a set amount of pages that are not already in the database and gathers the data for storage in the database.

#### 3.2.3.2 Input

A starting URL to search and the number of pages.

#### 3.2.3.3 Output

The pages successfully scanned and parsed.

#### 3.2.3.4 Error Handling

The spider runs out of pages to scan: The spider will stop and return the amount of pages it managed to scan.

### 3.2.4 Web Crawler – Scan Until End

#### 3.2.4.1 Introduction

In order to gather the most results possible, the spider is told to not stop crawling until it runs out of links to follow. The spider will not revisit links it has previously visited in the database*.*

#### 3.2.4.2 Input

A starting URL

#### 3.2.4.3 Output

The pages successfully scanned and parsed.

#### 3.2.4.4 Error Handling

The spider runs out of pages to scan: The spider will stop and return the amount of pages it managed to scan.

### 3.2.5 Web Crawler – Update Scan a Set Amount of Pages

#### 3.2.5.1 Introduction

In order to keep the database up to date on pages that may have changed, the crawler will transverse a set amount of pages, even those already in the database, and detect changes to the page then update the page if necessary.

#### 3.2.5.2 Input

A starting URL

#### 3.2.5.3 Output

The page that have been rescanned.

#### 3.2.5.4 Error Handling

The spider runs out of pages to scan: The spider will stop and return the amount of pages it managed to scan.

### 3.2.6 Web Crawler – Port Scanning

#### 3.2.6.1 Introduction

An alternate way to scan using a port range. The spider will index any page that gives a response and add it to the database. It will parse every page it finds, even ones in the database. If the entry already exists, then it will be erased and re-evaluated.

#### 3.2.6.2 Input

The IP range to scan

#### 3.2.6.3 Output

A list of IPs that has responded with an interface and has been successfully added to the database.

### 3.2.7 Web Crawler – Clear Location from Database

#### 3.2.7.1 Introduction

The crawler creates a spider that will transverse through a set amount of pages that are not already in the database and gathers the data for storage in the database.

#### 3.2.7.2 Input

A starting URL to search and the number of pages.

#### 3.2.7.3 Output

The pages successfully scanned and parsed.

#### 3.2.7.4 Error Handling

The spider runs out of pages to scan: The spider will stop and return the amount of pages it managed to scan.

### 3.2.7 Web Crawler – Add Keyword to Database

#### 3.2.7.1 Introduction

Deletes a location from the database and clears all keywords associated with that page.

#### 3.2.7.2 Input

The page location

#### 3.2.7.3 Output

The removal of the input location

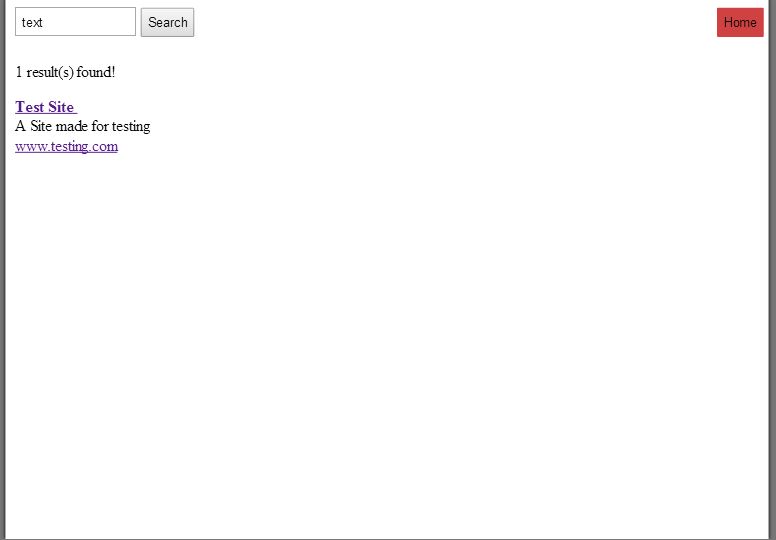
#### 3.2.7.4 Error Handling

The location was not found in database: Exits the function

## 3.3 Use Cases

### 3.3.1 Use Case #1

The user searches for the word “text”, which is sample data.



### 

### 3.3.2 Use Case #2

A user searches for the word “nothing”, which is not contained in our database.



## 3.4 Classes / Objects

### 3.4.1 Web Crawler: Controller

#### 3.4.1.1 Description:

Has control over every process in the crawler. It is used to create the spiders and initialize the Database Manager.

### 3.4.2 Web Crawler: Database Manager

#### 3.4.2.1 Description:

A static class that holds the connection and interactions with the database.

#### 3.4.2.2 Functions:

Clear location from database, Add keyword to database

### 3.4.3 Web Crawler: Spider

#### 3.4.3.1 Description:

Is responsible for crawling and parsing the web pages.

#### 3.4.3.2 Functions:

Parse Page, Recursively parse a set amount of pages, Scan until end, Update Scan, Port Scanning

## 3.5 Non-Functional Requirements

### 3.5.1 Performance

The system must be able to provide a high level of accuracy, so the project must provide 90% relevancy to search queries in the first 10 results. This will be determined from unit tests with predetermined results matching search results based on keyword weight values. Search queries must also return results within 5 seconds (not dependent on user’s internet speed).

### 3.5.2 Reliability

The team will limit the downtime of the system to 1 minute per day or 2 hours per month, depending on maintenance required. At the end of the system’s creation, the software should run reliably by itself, but hardware failure or bugs may still cause some unavoidable downtime.

### 3.5.3 Availability

The system will be available from anywhere that broadband internet is available and the website hasn’t been blocked.

### 3.5.4 Security

The system will only have necessary system ports as well as port 80 open to allow the web server to deliver data. The web crawler will use either port 80 and 443 to gather data on WKU’s websites. Prepared SQL queries and input validation are also used to prevent SQL injection.

### 3.5.5 Maintainability

The system is located on one computer, so hardware replacement costs are low and time for the replacements are quick. As for software maintainability, the code will be commented well with function descriptions included for every function.

### 3.5.6 Portability

As long as any future system has the ability to run Ubuntu, has a large enough hard drive for the database, and a processor fast enough for the web crawler to scan and index, the system will be able to be transferred to another machine.

## 3.6 Inverse Requirements

The system must not inhibit the normal network in any significant way. This includes proper security standards applies to the system to stop malicious attacks to the network. Also, the web crawler cannot maliciously scan or store data affecting the privacy of personal or encrypted devices.

## 3.7 Design Constraints

Design constraints at the moment are limited to lack of resources (time, money, manpower) to be able to scale up the design. If development continues at a reasonable pace, time may open at the end of the project to increase efficiency of the search process. This includes scaling the server to multiple devices with a load balancer, as well as general software touch ups to decrease search time.

## 3.8 Logical Database Requirements

A Database will be used for this project. It will be a database run by MySQL on an Ubuntu Server. The data that will be stored will be keywords, keyword weight, the website url, website name, description, and the full text of the website. There is currently planned to be three database tables. The tables are as followed:

* keywords(keyId, word) - keyId is the primary key. word is a 15 character varchar. Basically a string with a max length of 15 characters.
* locations(webId, name, description, url, siteFullText) - webId is the primary key. name is a 15 character varchar for the name of the site. description is a 300 character varchar for like the first line on the page or something like that. url is a 1500 varchar to store the URL/IP of the webpages. The field siteFullText is a string field that will hold the full text of the website. We can use this field to see if a search phrase appears inside of a page.
* siteKeywords(webId, keyId, weight) webId and keyId are both the primary key for a compound key. weight is an integer for the weight of the keyword.

Note: The size of the fields might change as the project progresses.

We currently have the storage capacity to store around 500GBs of data. The data will be kept in the database until it is updated by the web crawler. In the event that we run out of space we will look into adding more storage capacity or reducing the amount of data we are storing for each website.

## 3.9 Other Requirements

No other requirements at the moment

# 4. Analysis Models

## 4.1 Data Flow Diagrams (DFD)

**Web Crawler**

Query database for links

* List of keywords + weights
* URL/IP (location)
* Description
* Links on page

**Master**

Parsed Data

Location Data

Location Data

Parsed Data

Parsed Data

Location Data

**Slave**

**Slave**

**Slave**

Insert into database table

[WEIGHT] **|** [KEYWORD] **→** [LOCATION]

[LOCATION] **|** [DESCRIPTION]

This diagram shows the transfer of data from the slaves of the web crawler to the master. It also shows the sequence for which it populates the database and queries for more locations to parse.

**Database**

|  |  |
| --- | --- |
| keyId | keyword |

keyId: ID to link the keyword to its weight

keyword: word stored from website

**keywords**

|  |  |  |
| --- | --- | --- |
| webId | keyId | weight |

webId: ID to link to page details

keyId: ID to link the keyword to its weight

weight: relevancy of keyword to page

**siteKeywords**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| webId | name | description | url |  | siteFullText |

webId: ID to link keywords to page

name: title of web page

description: short description of web page

url: URL or IP of web page

determine if page has changed

siteFullText: full text of content area of web page

**locations**

**Website/PHP**

OUTPUT: next location for slave to parse

INPUT: list of links on page

INPUT: List of keywords + weights,

location, description

**Web Crawler Master**

**Master**

OUTPUT: Site Name, location, description

INPUT: User keywords

This diagram shows how the database is populated and queried by the other parts of the system.

**Website**

**Homepage**

List of all webpages, description of each, location

Query database with keywords

**Results**

LIST: Location title listing with short description

SEARCHBOX: User-entered keywords

**PHP**

This diagram shows how the website processes user queries to the server, the database, then returns the results to the user.

# 5. Planning and Resources

Our team consists of 5 people: Owen Young, Yi Liu, Taylor Atkinson, Blake Lombard, and Matthew Clark. We’ve split the team into three groups based on everybody’s interests and skills. Owen and Blake are working on the web crawlers and master program which will be written in Java. Yi and Matthew will be working on the user interface and front-end of the search engine using HTML, CSS, JavaScript, and PHP. Taylor will be working with the team organizing and optimizing the database as well as instructing the team on the database queries/functions we need to keep the system running smoothly. Everybody is willing to help out where needed since we all know Java and web design, so if there are problems with the workflow, we can easily compensate for these situations.

By the second milestone we will have a basic search engine that is able to index at least one location. This includes parsing the page, weighing keywords, adding data to the database, and letting a user search for the page. By the third milestone we should have a search engine that is able to index most of the IP range as well as be able to let the user search for pages, although the speed & relevancy might not be the best. By the final milestone speed should be relatively quick (<5 seconds) and word weight will be tuned to display the most relevant results.

After classes on Monday and Wednesday the team will meet for around 10+ minutes to discuss any problems or new ideas that we have for the project. We will use Fridays to manually make changes to the system as a team as well as individually if needed. The team has a Facebook group set up if we need to ask any questions or post comments, and we also have a GitHub repository to do version control as well as project backup and collaboration.

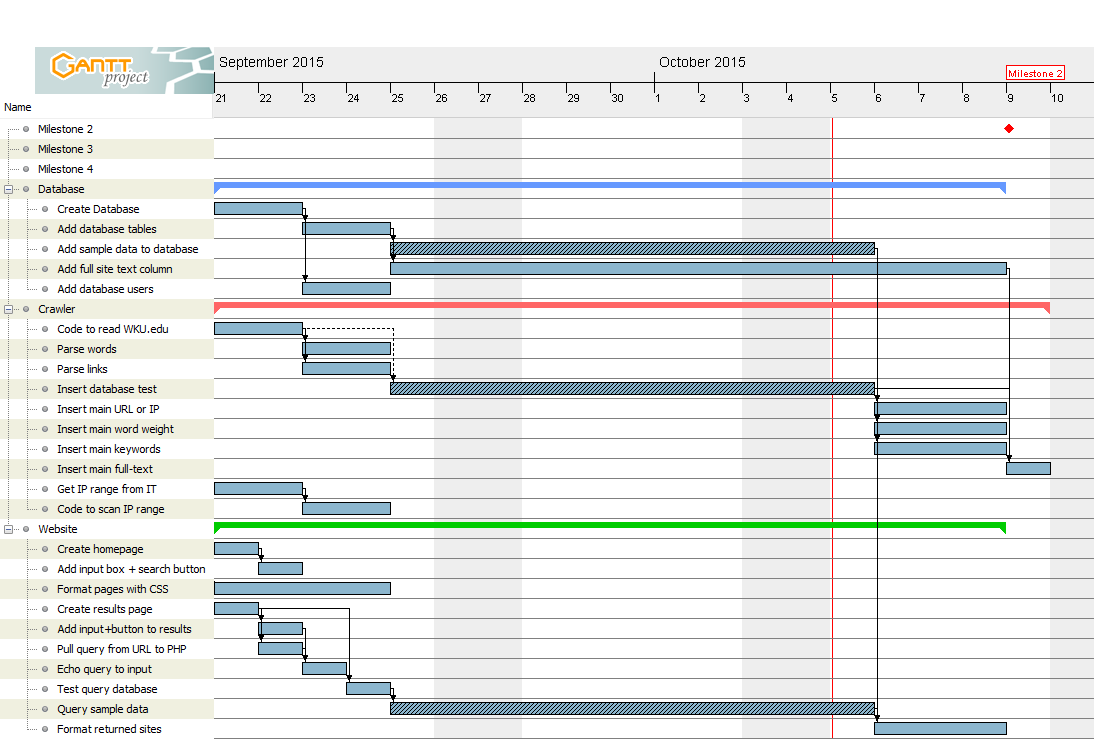


Figure 4: A Gantt chart displaying project scheduling until milestone 2

# 6. Risks

## 6.1 System Risks

To minimize data loss, we are using GitHub for version control. We also have to plan for malicious search queries, denial-of-service attacks, and general large amounts of traffic. If any of these occur, our system could go down leaving users without a search engine. In order to keep these risks low, we will perform input validation as well as usage tests to make sure the system can handle relatively heavy loads. Another risk we could face is defective hardware, but we can handle that risk by using temporary machines to continue programming and work on other parts of the project while the new hardware is shipped. Along with defective hardware, we also run the risk of hardware being too slow for the database queries or web crawler parsing to be efficient. To minimize, we will test the system early and often so we have time to order better hardware by the project completion. Since our team is well rounded, we can also handle the risk of moderate project requirement changes as well as team changes dictated by time crunches and sickness.

## 6.2 Feasibility

The number of users that visits www.WKU.edu is approximately 162,454 unique visitors per month. Using this data, we can make a rough estimate that probably 1/20 users use the search engine. This brings the number of users down to around 8122 unique people that use the search engine per month, and with about 3 searches per user, this brings the number of searches to around 24,369 / month. Without administrative data, that’s the best estimate that we can make on the number of search queries.

Using the current Google custom search with a wildcard character, we were able to estimate about 101,000 different pages on the WKU website. There are about 300 words on each page, but multiple are repeated. From this, we can say that we need at least 101,000 entries in our table with keys linked to a keyword table.

# 7. Deliverables

The system will be given to the client, which includes the software (web crawler, port scanner, website code, and database) and hardware (computer system that the software runs on).

# 8. Change Management Process

Every week the progress of the project will be discussed with the client and if changes need to be made to the requirements, the team will decide who needs to change the documents. These decisions will be based upon which sub-team’s requirements have changed.

# 9. Software Performance

## 9.1 Performance Requirements

### 9.1.1 Performance Goals

We have set a few goals for the software that we would like to reach.

1. Web Searches should be less than 1 second
2. The Port Scanner should be able to finish within 12 hours.
3. The Web Crawler should be able to Index every page within 24 hours.

### 9.1.2 Hardware Configuration

|  |  |  |  |
| --- | --- | --- | --- |
| Hardware Item | Name | Speed | Other Information |
| CPU | Intel Celeron J1900 | 1.99GHz | Quad-Core |
| System Memory |  | 1333MHz | 4GiB Capacity |
| Solid State Drive | ADATA SP900 | Up to 91K IOPS | 128 GB Capacity |
| Hard Drive | Hitachi E7K500 | 7200 RPM | 500GB Capacity |
| Network | Gigabit Ethernet Controller | Up to 1GBit/s | 46Mbit down 58Mbit up |

## 9.2 Workload

### 9.2.1 Application Workload

The following are estimates on how much time a user will be using functions on our site.

* 40% searching
* 60% selecting item from results

### 9.2.1 System Workload

The following are processes that can be expected to be running while the our software is running

* Apache2
* Web Crawler
* Port Scanner

## 9.3 Component Testing

### 9.3.1 Web Search

|  |  |  |  |
| --- | --- | --- | --- |
| Search Item | First Search | Second Search | Third Search |
| Wku – item in database | 0.017 seconds | 0.017 seconds | 0.017 second |
| Lkjsdflks – item not in database | 0.001 seconds | 0.001 seconds | 0.001 seconds |
| No query was inputted | 0.001 seconds | 0.001 seconds | 0.001 seconds |

### 9.3.2 Port Scanner

|  |  |  |  |
| --- | --- | --- | --- |
| Port to Scan | 1st Test | 2nd Test | 3rd Test |
| IP with Open Port | 1.007 seconds | 1.008 seconds | 3.013 seconds |
| IP with Closed Port | 15.033 seconds | 15.038 seconds | 15.045 seconds |

### 9.3.3 Web Crawler

|  |  |  |  |
| --- | --- | --- | --- |
| Item Tested | 1st Test | 2nd Test | 3rd Test |
| Website not in database |  |  |  |
| Website in database |  |  |  |
| Website in database with changes |  |  |  |

## 9.4 Performance Inspection

### 9.4.1 Competitor’s Performance

This is the performance of our main competitor Google.

|  |  |  |  |
| --- | --- | --- | --- |
| Search Item | First Search | Second Search | Third Search |
| Wku | 0.51 seconds | 0.52 seconds | 0.51 seconds |
| Lkjsdflks | 0.19 seconds | 0.19 seconds | 0.64 seconds |
| No Query was inputted | Doesn’t Search | Doesn’t Search | Doesn’t Search |

### 9.4.2 Our Performance

This is our current performance. It is likely that this will slow down as we crawl more sites though.

|  |  |  |  |
| --- | --- | --- | --- |
| Search Item | First Search | Second Search | Third Search |
| Wku – item in database | 0.017 seconds | 0.017 seconds | 0.017 second |
| Lkjsdflks – item not in database | 0.001 seconds | 0.001 seconds | 0.001 seconds |
| No query was inputted | 0.001 seconds | 0.001 seconds | 0.001 seconds |

### 9.4.3 Minimum Performance Requirements

The following are Performance items that will allow us to be competitive.

|  |  |
| --- | --- |
| Search Item | Max Search Time |
| Item in Database | 0.5 seconds |
| Item not in Database | 0.5 seconds |
| No query was inputted | 0.005 seconds |

We are currently able to reach and surpass these requirements which will make us be very competitive but our search time will increase with the amount of pages we crawl.

### 9.4.4 Customers Expectations

Our users will be using this search engine as a replacement for Google, Bing, Yahoo, and other web search engines for searching Western related materials. To keep our users happy we are going to need to provide near instantaneous search results that are both relevant and up to date.

# 10. Design Decisions

## 10.1 Data Storage

The data in this project is being stored in a database hosted by MySQL Server. The reason we picked this database is because it is easily integrated into a PHP site and it is also easy to work it into a java program.

## 10.2 Crawler Frequency

|  |  |
| --- | --- |
| Crawler | How often it runs |
| Web Crawler | Once a week |
| Port Scanner | Once a week |

These are subject to change depending on a few factors:

* How long they take to run
* How often the data they find changes

### 10.2.1 Web Crawler

The Web Crawler will take longer to run than the Port Scanner but the data it retrieves will also be updated a lot more frequently than the Port Scanner. We are starting out with it running once a week and will decide if we need to increase its frequency.

### 10.2.2 Port Scanner

The Port Scanner will take a lot less time to run than the Web Crawler but the data it will be collecting will also be updated significantly less than the Web Crawler. We are going to run the Scanner once a week and we will probably not increase this frequency.

## 10.3 Displaying Results

In the beginning, we had another idea for how to order the results. The way we would have displayed the results is that we would have done an SQL search for them and then display the results based solely on the order they were found in the database. We later decided that we were going to change that a bit. We decided that we would give each keyword a weight and have the results be ordered by the weight of the keywords.

# 11. Conclusion

The project described herein displays how the system runs and the inner workings of the system as a whole. The goal of this system is to replace the Custom Google Search that is currently being used by Western Kentucky University to search their site. We believe that switching over to our own solution will provide us with a more customizable environment and potentially a more up to date search engine. The project includes a website to search on, a web crawler to get the page data, and a port scanner to locate any servers running on the WKU Network.

# Appendix

## A.1 Installation Instructions

The following are instructions on how to install and set up our software package.

### A.1.1 Server Installation

Before we can install our software we are going to need something to install it on.

#### A.1.1.1 Required Software

The following are the software the software was tested on. Other versions will probably work but they have not been tested and are not supported.

* Ubuntu Server 14.0.4.3 LTS
* Apache 2.4.7
* MySQL 5.5.46-0ubuntu-.14.04.2
* Java 1.7.0\_79
* Git 1.9.1

#### A.1.1.2 Server Installation

Installing Ubuntu Server is outside of the scope of this document. You can use the instructions from the website provided. <http://www.ubuntu.com/download/server/install-ubuntu-server>

#### A.1.1.3 Software Installation

You will need to install the programs listed up in A.1.1.1. This can be done by running the following command**:**

sudo apt-get install apache2 mysql-server openjdk-7-jdk git

This command will install all of the software.

### A.1.2 Software Setup

#### A.1.2.1 Git Setup

To get the code for the project you are going to need to use Git. You are going to need to clone the repository.

1. You will need to go to where you would like to store the code
2. Then you will run the command git clone <https://github.com/Zortrox/CS360WebSearch.git>
3. You will be prompted for a Username. Enter the Github username provided.
4. You will then be prompted for a Password. Enter the Github password provided.

You will now have all the code for the project.

#### A.1.2.2 Apache2 Setup

Now we need to take the files we have gotten from Git and put them where the web server can show them to the world. We can do this by going to the CS360WebSearch folder and running the following command:

Cp site /var/www

This command will copy the site folder and rename it to the /var/www which is where the web server defaults too. There are other ways to do this but this is the easiest way to do it.

#### A.1.2.3 MySQL Server

Whenever MySQL is installed it will ask you to set a root password for the server. This is **NOT** the root account on the server. Then you will have to follow the following steps:

1. Log into the root account using the command “mysql –u root –p”
2. Create a new database by running the command “CREATE DATABASE webSearchEngine;”
3. Exit MySQL by typing “Exit”
4. Now go to the database folder in the CS360WebSearch folder
5. Now run “mysql –u root –p webSearchEngine < locations.sql;”
6. Now run “mysql –u root –p webSearchEngine < keywords.sql;”
7. Now run “mysql –u root –p webSearchEngine < siteKeywords.sql;”
8. Now run “mysql –u root –p webSearchEngine < webServers.sql;”
9. Now reenter MySQL by running “mysql –u root –p”
10. Now create a “web” user by running “CREATE USER ‘web’@’localhost’ INDENTIFIED BY ‘PASSWORD’;” – with password being whatever you want it to be
11. Then create a “webCrawl” user by running “CREATE USER ‘webCrawl@’localhost’ INDENTIFIED BY ‘PASSWORD’;” – with password being whatever you want it to be
12. Now we are going to allow the “web” user to view the tables. “GRANT SELECT ON webSearchEngine.\* TO ‘web’@’localhost’;
13. Then we give the ability to read and write to the “webCrawl” user. “GRANT SELECT INSERT UPDATE ON webSearchEngine.\* TO ‘webCrawl’@’localhost’;”

#### A.1.2.4 Java

Java should be good to go after you run the install command listed in A.1.1.3

#### A.1.2.5 Website Configuration

Now that we have the files where the website can read them and the database all set up and ready, we now need to update the website configuration so that the website can talk to the database. This can be done in the following steps:

1. From the CS360WebSearch directory, go to the site/search directory.
2. Open up results.php in a text editor. (Nano, Vim, etc)
3. Edit the $pass value on the 9th line to be the password of your “web” user.
4. Save the file.

Now the website can communicate with the database

#### A.1.2.6 WebCrawler Configuration

Now we need to allow the web crawler to be able to access the database so that the website can use the data that the crawler finds. We can do this by doing the following:

1. From the CS360WebSearch directory, go to the robots/WKUSpider/src directory
2. Open DatabaseManager.java and edit the String pass on line 14 to be your password
3. Save the file

Now the web crawler can communicate with the database.

### A.1.3 Running the Software

#### A.1.3.1 Website

The website will automatically be running whenever apache starts. This will be at system startup by default.

#### A.1.3.2 WebCrawler

The Web Crawler will have to be run manually. We recommend making a jar file and running it whenever it is needing to be run.

#### A.1.3.3 Port Scanner

The Port Scanner will also need to be ran manually. You can also make a jar file for this and run it whenever it needs to be ran.

## A.2 Source Code

### A.2.1 Database

#### A.2.1.1 locations.sql



#### A.2.1.2 keywords.sql



#### A.2.1.3 siteKeywords.sql



### A.2.2 Website

#### A.2.2.1 index.htm



#### A.2.2.2 style.css

html {

height: 100%;

}

body {

background-color: #777;

width: 100%;

height: 100%;

margin: 0px;

}

#wrapper {

background-color: #FFF;

width: 80%;

min-height: 100%;

margin-left: auto;

margin-right: auto;

box-shadow: 0px 0px 4px #000;

overflow: auto;

}

#logo {

position: relative;

top: 25%;

display: block;

margin-left: auto;

margin-right: auto;

width: 20%;

height: auto;

}

#form-main {

position: relative;

top: 25%;

margin-left: auto;

margin-right: auto;

width: 90%;

text-align: center;

}

#form-results {

padding-left: 10px;

padding-top: 10px;

float: left;

box-sizing: border-box;

width: 50%;

}

input {

padding: 5px;

width: 55px;

}

#input-query {

width: calc(100% - 60px);

box-sizing: border-box;

}

#results-list {

padding: 10px;

}

#results-top-bar {

width: 100%;

overflow: auto;

}

#home-button div {

font: 13.3333px Arial;

float: right;

padding: 7px;

border-radius: 1px;

background-color: #D04242;

margin-top: 10px;

margin-right: 5px;

}

#home-button,

#home-button:visited,

#home-button:active {

color: black;

}

#### A.2.2.3 /search/index.php



#### A.2.2.4 /search/results.php

<?php

//timing setup

$startTime = microtime(true);

$endTime = null;

//setup user and connect to database

$user = "web";

$pass = "webSearch!";

$database = "webSearchEngine";

$mysqli = new mysqli("localhost", $user, $pass, $database);

//check connection

if (mysqli\_connect\_errno()) {

printf("Connect failed: %s\n", mysqli\_connect\_error());

exit();

}

//create array of strings in query \*\*IN PROGRESS\*\*

$moreStrings = false;

$queryPart = $query;

while ($moreStrings) {

$stringPos1 = strpos($queryPart, "\"");

$queryPart = substr($queryPart, 0);

}

//creates a partial query that finds values in columns

function createConstruct($wordArray, $column, $useKey = false) {

$construct = "";

$firstRow = true;

foreach( $wordArray as $wordKey => $wordValue ) {

if ($firstRow == true) {

if ($useKey)

$construct .= "$column LIKE '$wordKey'";

else

$construct .= "$column LIKE '$wordValue'";

$firstRow = false;

} else {

if ($useKey)

$construct .= " OR $column LIKE '$wordKey'";

else

$construct .= " OR $column LIKE '$wordValue'";

}

}

return $construct;

}

//remove all string searches and put them into an array

$stringSearch = array();

preg\_match\_all("/([\"'])(?:(?=(\\\?))\\2.)\*\\1/", $query, $stringSearch);

$query = preg\_replace("/([\"'])(?:(?=(\\\?))\\2.)\*?\\1/", "", $query);

//create array based on user-inputted words

//get all keyword rows from database based on user-inputted words

//$query variable is the user's search

$querySplit = preg\_split('/\s+/', trim($query));

$keywordQuery = "SELECT \* FROM keywords WHERE " . createConstruct($querySplit, "word");

$keywordRows = $mysqli->query($keywordQuery);

$keysFound = $keywordRows->num\_rows;

if ($querySplit[0] == "") { //if no query tell user

$endTime = microtime(true);

$totalTime = round($endTime - $startTime, 3);

echo "You must input a query. Time taken: $totalTime seconds.";

}

else if ($keysFound == 0) { //if query returns no results, inform user

$endTime = microtime(true);

$totalTime = round($endTime - $startTime, 3);

echo "Sorry, there are no matching result for <b> $query </b>. Time taken: $totalTime seconds.";

}

else {

//get all keyIds of the user-inputted keywords

$keyArray = array();

while ($resultsRow = $keywordRows->fetch\_row()) {

array\_push($keyArray, $resultsRow[0]);

}

//gather all webIds of websites based on keywords found

//sort in descending order based on cumulative word weights

$webArray = array();

$webIDQuery = "SELECT webId FROM siteKeywords WHERE " . createConstruct($keyArray, "keyId");

$webIDResults = $mysqli->query($webIDQuery);

while ($siteKeywordsRow = $webIDResults->fetch\_row()) {

$webID = $siteKeywordsRow[0];

$wordWeight = $siteKeywordsRow[2];

if (in\_array($webArray, $webID)) {

$webArray[$webID] += $wordWeight;

} else {

$webArray[$webID] = $wordWeight;

}

}

arsort($webArray);

//get all location data based on webIds found

$websiteRowQuery = "SELECT \* FROM locations WHERE " . createConstruct($webArray, "webId", true);

$websiteRows = $mysqli->query($websiteRowQuery);

//display number of results found in how much time

$sitesFound = count($webArray);

$plural = $sitesFound > 1 ? "s" : "";

echo "<p>$sitesFound result$plural found in ";

$endTime = microtime(true);

$totalTime = round($endTime - $startTime, 3);

echo "$totalTime seconds</p>";

while ($website = $websiteRows->fetch\_assoc()) {

$title = $website['name'];

if ($title == "") $title = $website['url'];

$desc = $website['description'];

$url = $website['url'];

if (substr($url, 0, 4) != "http") {

$url = "http://" . $url;

}

echo "<p><a href='$url'> <b> $title </b> </a> <br> $desc <br> <a href='$url'> $url </a></p>";

}

}

?>

### A.2.3 Web Crawler

#### A.2.3.1 Notes on using the crawler.txt



#### A.2.3.2 Data.java



#### A.2.3.3 DatabaseManager.java

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.ArrayList;

public class DatabaseManager {

static Connection connection;

private static final String url = "jdbc:mysql://127.0.0.1:3306/webSearchEngine";

private static final String user = "crawl";

private static final String pass = "webCrawl!";

private static ArrayList<String> visited = new ArrayList<String>();

/\*\*

\* sets up the driver and readies the functions needed

\*/

public static boolean Initialize() {

try {

Class.forName("com.mysql.jdbc.Driver").newInstance();

System.out.println("Creating connection...");

connection = DriverManager.getConnection(url,user,pass);

System.out.println("Connection established!");

return true;

} catch (Exception e) {

e.printStackTrace();

}

return false;

}

public static void Exit(){

try {

connection.close();

System.out.println("Connection closed");

} catch (SQLException e) {

e.printStackTrace();

}

}

public static int getLocation(String url){

try {

PreparedStatement pst = connection.prepareStatement("SELECT webId FROM locations WHERE url = \""+url+"\"");

ResultSet rs = pst.executeQuery();

int index = -1;

while(rs.next())

index = rs.getInt(1);

return index;

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return -1;

}

//(webId, name, description, url, hash)

/\*\*

\* Adds a new location to the database

\* @param url - the URL of the page

\* @param name - the Title of the page

\* @param description - A description of the page

\* @param fulltext - all the text on the page

\* @return the index of the location

\*/

public static int addLocation(String url, String name, String description, String fulltext){

try {

PreparedStatement pst = connection.prepareStatement("SELECT webId FROM locations WHERE url = \""+url+"\"");

ResultSet rs = pst.executeQuery();

int index = -1;

while(rs.next())

index = rs.getInt(1);

if (index != -1) {

pst = connection.prepareStatement("UPDATE locations SET siteFullText = ?, description = ? WHERE webId = ?");

pst.setString(1, fulltext);

pst.setString(2, description);

pst.setInt(3, index);

pst.execute();

return index;

} else {

PreparedStatement pst2 = connection.prepareStatement("INSERT INTO locations (name, description, url, siteFullText)"

+ " values (?, ?, ?, ?)", Statement.RETURN\_GENERATED\_KEYS);

pst2.setString(1, name);

pst2.setString(2, description);

pst2.setString(3, url);

pst2.setString(4, fulltext);

pst2.executeUpdate();

ResultSet rs2 = pst2.getGeneratedKeys();

rs2.next();

return rs2.getInt(1);

}

} catch (SQLException e) {

e.printStackTrace();

}

return -1;

}

public static int addKeyword(String keyword){

try {

PreparedStatement pst = connection.prepareStatement("SELECT keyId FROM keywords WHERE word = \""+keyword+"\"");

ResultSet rs = pst.executeQuery();

int index = -1;

while(rs.next())

index = rs.getInt(1);

if(index != -1)

return index;

else

{

if (keyword.length() >= 15) {

System.out.println(keyword + " too long for keyword column");

return -1;

}

pst = connection.prepareStatement("INSERT INTO keywords (word) " + "values (?)",Statement.RETURN\_GENERATED\_KEYS);

pst.setString(1, keyword.toLowerCase());

pst.executeUpdate();

ResultSet rs2 = pst.getGeneratedKeys();

int ni = -1;

while(rs2.next())

ni = rs2.getInt(1);

return ni;

}

} catch (SQLException e) {

e.printStackTrace();

}

return -1;

}

/\*\*

\* Adds a Data object to the database for a page.

\* @param data - An array list of Data objects

\* @param pageID - The id of the page index. This is returned from the addLocation() function

\*/

public static void addData(ArrayList<Data> data, int pageID){

loop:

for(Data d : data){

int keyID = addKeyword(d.word);

// skips the word if it wasn't in the database

if(keyID == -1)

continue;

try {

PreparedStatement pst = connection.prepareStatement("SELECT \* FROM siteKeywords WHERE webID = " + pageID + " AND keyId = " + keyID);

ResultSet rs = pst.executeQuery();

while (rs.next())

{

pst = connection

.prepareStatement("UPDATE siteKeywords SET weight = ? WHERE webId = ? AND keyId = ?");

pst.setInt(1, d.weight);

pst.setInt(2, pageID);

pst.setInt(3, keyID);

pst.execute();

continue loop;

}

PreparedStatement pst2 = connection.prepareStatement("INSERT INTO siteKeywords (webId, keyId, weight)"

+ " values (?, ?, ?)");

pst2.setInt(1,pageID);

pst2.setInt(2,keyID);

pst2.setInt(3, d.weight);

pst2.execute();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

public static void visit(String u){

visited.add(u);

}

public static boolean visited(String u){

return visited.contains(u);

}

public static void printLocationDatabase(){

try {

PreparedStatement pst = connection.prepareStatement("SELECT \* FROM locations");

ResultSet rs = pst.executeQuery();

while (rs.next()) {

System.out.println();

System.out.print(rs.getInt(1) + " : ");

System.out.print(rs.getString(2) + " : ");

System.out.print(rs.getString(3) + " : ");

System.out.print(rs.getString(4) + " : ");

System.out.print(rs.getString(5) + " : ");

}

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

public static void printKeywordDatabase(){

try {

PreparedStatement pst = connection.prepareStatement("SELECT \* FROM keywords");

ResultSet rs = pst.executeQuery();

while (rs.next()) {

System.out.println();

System.out.print(rs.getInt(1) + " : ");

System.out.print(rs.getString(2) + " : ");

}

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

public static void printDataDatabase(){

try {

PreparedStatement pst = connection.prepareStatement("SELECT \* FROM siteKeywords");

ResultSet rs = pst.executeQuery();

while (rs.next()) {

System.out.println();

System.out.print(rs.getInt(1) + " : ");

System.out.print(rs.getInt(2) + " : ");

System.out.print(rs.getInt(3) + " : ");

}

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

#### A.2.3.4 PageParser.java

import java.io.BufferedReader;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.io.InputStreamReader;

import java.net.URL;

import java.net.URLConnection;

import java.util.ArrayList;

import java.util.Comparator;

/\*\*

\* A tool for parsing the webpage data to collect information useful for building a search engine.

\*/

public class PageParser {

String url, title;

ArrayList<String> links;

ArrayList<Data> dataNodes;

String preview = "";

String[] lowWeight = {"of","a","the","and","is","in","to","all","in"};

String text = "";

boolean isEmpty;

// we are also gonna need a dictionary of words and a way to determine weights

//161.6.0.0 - 161.6.255.255

/\*

\* Notes:

\* We can actually parse all the info in one go,

\* but for now I have it separate so we can

\* catch area-specific bugs easier.

\*/

/\*\*

\* This object will hold all of the page info until it's stored in a database.

\* @param url of page

\*/

public PageParser(String url){

// store the url with this object for later storage in the database

this.url = url;

// the source code of the page

String src = null;

try {

src = getSource(url);

} catch (FileNotFoundException e) {

System.out.println("Above link 404?");

return;

}

if(src==null)

isEmpty = true;

links = new ArrayList<String>();

dataNodes = new ArrayList<Data>();

// The title text is probably among the most important things to consider

title = getTitle(src);

// System.out.println(title + "\n");

// for parsing

// for now just gets links

gatherLinks(src);

String text = gatherText(src);

text = text.replaceAll("\\s+"," ");

this.text = text;

// System.out.println(text);

preview = getDescription(src);

if(preview.equals(""))

preview = text.substring(0,text.length() >= 295 ? 295 : text.length())+"...";

// remove punctuation for now

text = text.replaceAll("\\.", "");

text = text.replaceAll("\\:", "");

text = text.replaceAll(",", "");

text = text.replaceAll("&nbsp", "");

text = text.replaceAll(";", "");

String[] keywords = getKeywords(src);

if(keywords != null)

for(int i = 0 ; i < keywords.length ; i++)

addData(keywords[i], 100);

String[] titleWords = title.split(" ");

for(int i = 0 ; i < titleWords.length ; i++){

addData(titleWords[i], 80);

}

String[] words = text.split(" ");

for(int w = 0 ; w < words.length ; w++)

addData(words[w], (int)((0.0 + words.length - w) / words.length \* 100.0));

//---------------------------------------------------------------------------------------------------Remove eventually

// Collections.sort(dataNodes,new OrderNode());

}

// This is just for display purposes. Can be deleted later...

class OrderNode implements Comparator<Data>{

@Override

public int compare(Data o1, Data o2) {

if(o1.weight > o2.weight)

return -1;

else if (o1.weight < o2.weight)

return 1;

return 0;

}

}

// DON'T FORGET THE HEADINGS!!!!

/\*\*

\* This function should be used when the object is created in order to gather all the links

\* on the page.

\* @param src - sourse code of site

\*/

private void gatherLinks(String src){

// gets the first link location (every page should have at least one)

String code = src.substring(src.indexOf("href=\"")+6);

// infinite loop so we can go through the entire string

// we could use while it contains a link, but this actually takes less processes

while (true) {

// gets the link are; ends with a "

String link = code.substring(0, code.indexOf('"'));

// adds the wku site to incomplete links

if (link.charAt(0) == '/')

link = "https://www.wku.edu" + link;

// make sure it's a WKU page and not a file

if (link.contains("wku.")

&& (link.contains("php") || link.contains("htm")

|| link.charAt(link.length() - 1) == '/')) {

// System.out.println(link);

links.add(link);

}

// move to the next href if it exists and exit the loop otherwise

int nextIndex = code.indexOf("href=\"");

if(nextIndex == -1)

break;

else

code = code.substring(nextIndex+6);

}

// System.out.println("\nDone - found " + links.size() + " WKU links");

}

/\*\*

\* Returns the title of the page

\* @param src of page

\* @return The title of the page

\*/

private String getTitle(String src){

if (src.contains("<title>")) {

String code = src.substring(src.indexOf("<title>") + 7);

code = code.substring(0, code.indexOf("</title>"));

return code;

}

else

return "";

}

private String getDescription(String src){

// name="description" content="

if(src.contains("name=\"description\" content=\"")){

String code = src.substring(src

.indexOf("name=\"description\" content=\"") + 28);

code = code.substring(0, code.indexOf("\">"));

return code;

}

else

return "";

}

private String[] getKeywords(String src){

// name="description" content="

if(src.contains("name=\"keywords\" content=\"")){

String code = src.substring(src

.indexOf("name=\"keywords\" content=\"") + 25);

code = code.substring(0, code.indexOf("\">"));

code.replaceAll(" ", "");

return code.split(",");

}

else

return null;

}

private String gatherText(String src){

String text = "";

int index = src.indexOf("<p>");

if(index == -1)

return "";

String code = src.substring(index+3);

while (true) {

if(code.contains("</p>"))

text += " " + clearTags(code.substring(0,code.indexOf("</p>")));

index = code.indexOf("<p>");

if(index == -1)

break;

else

code = code.substring(index+3);

}

return text;

}

private void addData(String in, int weight){

Data d = new Data(in);

d.weight += weight;

if(!dataNodes.contains(d)){

if(title.toLowerCase().contains(in.toLowerCase()))

d.weight += 100;

for(int i = 0 ; i < lowWeight.length ; i ++)

if(lowWeight[i].equals(in))

d.weight /= 4;

dataNodes.add(d);

}

else{

d = dataNodes.get(dataNodes.indexOf(d));

d.weight += 5;

for(int i = 0 ; i < lowWeight.length ; i ++)

if(lowWeight[i].equals(in))

d.weight /= 4;

}

}

/\*\*

\* Removes the html tags and their content "<" to ">" from the string

\* @param input string

\* @return the string without html tags

\*/

private static String clearTags(String input){

String edited = "";

for(int i = 0; i < input.length() ; i++){

if(input.charAt(i) == '<')

while(input.charAt(i) != '>') i++;

else

edited += input.charAt(i);

}

return edited;

}

public ArrayList<String> getLinks(){

return links;

}

public ArrayList<Data> getData(){

return dataNodes;

}

/\*\*

\* Returns the source code from an weblink as a string.

\* @param link of the webpage

\* @return Source Code as String

\* @throws IOException

\*/

private static String getSource(String link) throws FileNotFoundException{

// Gonna go ahead and catch the IO Expection here

try {

// opens a url and buffer

URL url = new URL(link);

URLConnection con = url.openConnection();

BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream(), "UTF-8"));

// reads the buffer into string

String inputLine;

StringBuilder str = new StringBuilder();

while ((inputLine = in.readLine()) != null)

str.append(inputLine);

in.close();

// return the completed string

return str.toString();

} catch (IOException e) {

e.printStackTrace();

}

return null;

}

}

#### A.2.3.5 Spider.java

import java.util.Queue;

public class Spider extends Thread{

int id;

int numOfSearches = 0;

SpiderThread controller;

Queue<String> links;

public Spider(int id, int numOfSearches, SpiderThread con) {

this.numOfSearches = numOfSearches;

this.id = id;

this.controller = con;

links = con.links;

}

/\*\*

\* Starts the crawling on a new thread.

\* If the links queue is currently empty, then wait a few seconds for another spider to populate it

\*/

public void run(){

if(links.isEmpty())

try {

System.out.println("Thread " + (id == -1 ? "Initial" : id) + " is waiting for links");

Thread.sleep(3000\*id);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

if(!links.isEmpty()){

System.out.println("Thread " + (id == -1 ? "Initial" : id) + " is starting");

run(links.remove());

try {

Thread.sleep(4000);

if(!links.isEmpty()) run(links.remove());

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Thread " + (id == -1 ? "Initial" : id) + " is finished");

} else{

System.out.println("Thread " + (id == -1 ? "Initial" : id) + " found no links!");

}

}

/\*\*

\* Recursive method that will recursively parse pages

\* @param url

\*/

private void run(String url){

System.out.println("Spider " + id + " is crawling... " + url);

controller.numOfPages++;

if(numOfSearches == 0)

return;

PageParser page = new PageParser(url);

if (!page.isEmpty) {

int pageID = DatabaseManager.addLocation(page.url, page.title, page.preview, page.text);

if (pageID != -1)

DatabaseManager.addData(page.getData(), pageID);

DatabaseManager.visit(page.url);

// fix this-------------------------------------------------------------------------------------------

for (String u : page.getLinks())

if (!DatabaseManager.visited(u))

links.add(u);

}

if(links.isEmpty())

return;

numOfSearches--;

run(links.remove());

}

}

#### A.2.3.6 SpiderThread.java

import java.util.LinkedList;

import java.util.Queue;

public class SpiderThread {

int numOfThreads, maxThreads, limit;

String startingPoint;

int numOfPages = 0;

Queue<String> links = new LinkedList<String>();

public SpiderThread(int maxThreads, String start, int amt) {

this.maxThreads = maxThreads;

startingPoint = start;

limit = amt;

startThreads();

}

public synchronized void startThreads(){

links.add(startingPoint);

Thread[] threads = new Thread[maxThreads];

Spider spider = new Spider(-1,1,this);

spider.run();

for (int n = 0; n < maxThreads; n++) {

Thread thread = new Spider(numOfThreads++,limit,this);

threads[n] = thread;

thread.start();

}

for (Thread t : threads)

try {

if(t!=null)

t.join();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}