CS242 Project – Part A submission

LOTR INFORMATION RETRIEVAL

JORGE MERCADO, hOVANES KESEYAN, Janakiram Kuppa

2018

Contents

[1. Objective 2](#_Toc505936418)

[2. Collaboration details: 2](#_Toc505936419)

[2.1. Jorge Mercado 2](#_Toc505936420)

[2.2. Hovanes Keseyan: 2](#_Toc505936421)

[2.3. Janakiram Kuppa: 3](#_Toc505936422)

[3. Crawling System Overview: 4](#_Toc505936423)

[3.1. Crawling Architecture 4](#_Toc505936424)

[3.2. Crawling Strategy 5](#_Toc505936425)

[4. Lucene Indexing Strategy Overview 7](#_Toc505936426)

[4.1. Indexing Architecture 7](#_Toc505936427)

[4.2. Text Analyzer choice 9](#_Toc505936428)

[4.3. High-level Data Model of Potential features from LOTR IR 9](#_Toc505936429)

[4.4. Index creation process run time results 10](#_Toc505936430)

[5. System Limitations 10](#_Toc505936431)

[6. Obstacles and solutions 10](#_Toc505936432)

[7. How to deploy the crawler and how to build the Lucene Index 11](#_Toc505936433)

# Objective

Objective of this project is to a create web crawler to crawl websites that contain data about characters, their details, locations in the world, and other information gathered from plot points and stories told in the world created in legendary author J.R.R. Tolkien’s works: The Lord of the Rings, The Hobbit, and The Silmarillion. Intent is to crawl a large and popular online Wiki located at http://lotr.wikia.com, known as “The One Wiki to Rule Them All”. This wiki is populated by fans with data from all the literature behind this world, including not only the original books but other canonical media such as the popular and successful movie series from the 2000s. Each character has a wiki page which describes any known information about their timeline, city/town of origin, actors that have portrayed them in media, relationships to other characters, and important plot points among the immense library of information that is available. The data collected from the crawler will be indexed using Lucene first, and later using Hadoop. We will compare the performance of these alternatives.

# Collaboration details:

## Jorge Mercado

Jorge helped develop the entire concept in java by coding a significant portion of Crawler code. Jorge provided the overall direction to the team on how to implement the crawler and provided valuable inputs on best ways to achieve the results. Jorge helped integrate the Indexing code into the base crawler code. Jorge also helped set up the code so that it can operate in either “crawler” mode with applicable parameters and “indexer” mode that takes in the indexer related parameters. The output from crawler mode will be xml files in a specific directory that will be used by the indexer mode.

Crawler Code built by Jorge has the following concepts embedded within it:

1. Get all URLs in page and post to queue
2. Do not crawl a page if it leaves the site
3. Parse Robots file to ensure that code follows the crawling ethics
4. Skip a website if it has already been crawled
5. Check for duplicate website and handle the logic related to this
6. Retrieve the features needed for setting up the search engine in the next phase of the project

## Hovanes Keseyan:

* Hovanes researched various topics that were discussed in the team meetings and came up with the proposal that is interesting, unique and one that meets the requirements of the project. So, the consensus was to crawl and build a search engine based on the world created in legendary author J.R.R. Tolkien’s works: The Lord of the Rings, The Hobbit, and The Silmarillion.
* Hovanes helped write up the summary of the intent of our project that was submitted to Prof Hristidis and Nhat Le.
* Hovanes also helped put together a data model for the potential features that we plan on collecting from the webpages.
* Hovanes implemented portions of web crawler code to identify and retrieve relevant text content of the webpage without erroneous web page data. Hovanes also implemented code for the crawler to extract personal information items from the right sidebar of each page to be indexed as a subset of the total page content.

## Janakiram Kuppa:

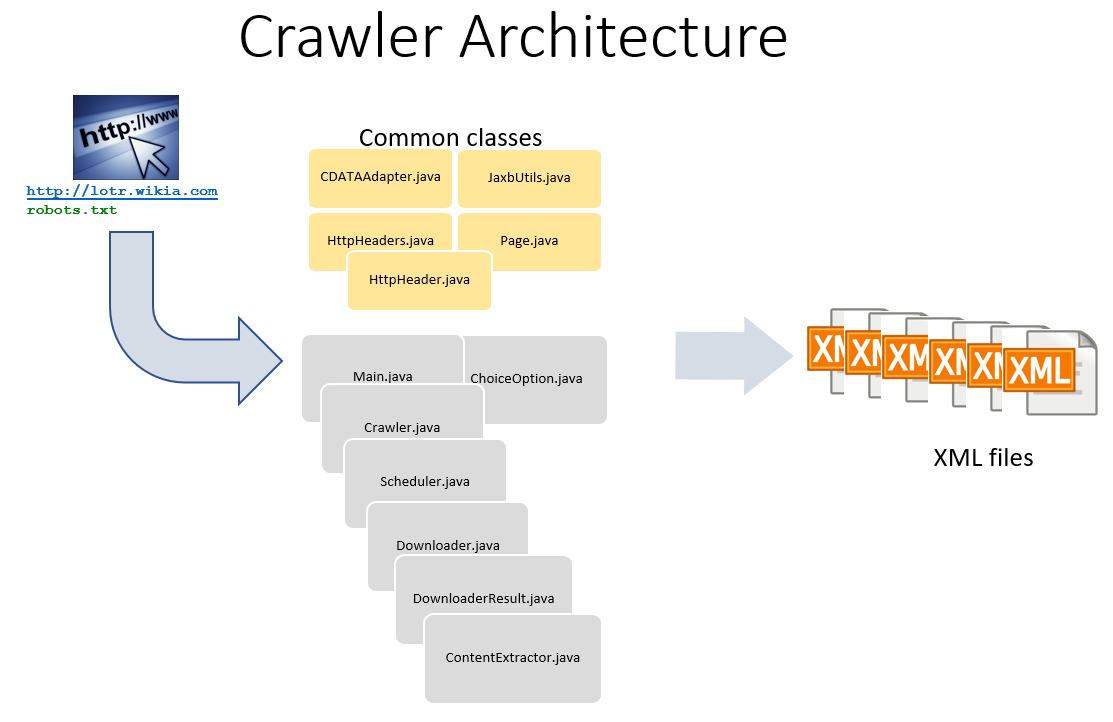
* Janakiram handled the entire coordination with the team depending on the needs and availability of the team.
* Brought various ideas on what we can do for the project. All such ideas were discussed and considered before the team decided to go with Hovanes’s recommendation of crawling the web for Lord of the Rings world.
* Helped provide suggestions on creating code so that it can be easily scaled when we do Part B of the project and the code is written so that it has decent performance. He proposed that we could adopt techniques so that the data we get from the crawler goes into a memory buffer and commit the entire buffer to a file when the buffer gets full.
* Coded the java classes that do the entire core Indexing logic using Lucene libraries. Also coded the java classes that create the csv file with process timings for the Indexer.
* Created the Crawler and Indexing process architecture diagrams.
* Performed various levels of testing on the code as it was developed, fixed some defects and provided comments on what to address to the team for some issues. Played the key role in doing performance testing and identified issues that resulted in some revamping of crawler code.
* Helped put together most of the Project report for the project

# Crawling System Overview:

## Crawling Architecture

Crawler Architecture has been designed so that it puts out XML files as it processes the lotr.wikia.com website and all the associated links from that webpage. Every XML file contains a document comprising of several data elements. These XML files will go in as Input to the Indexer. Figure 1 below illustrates the architecture of the crawler.

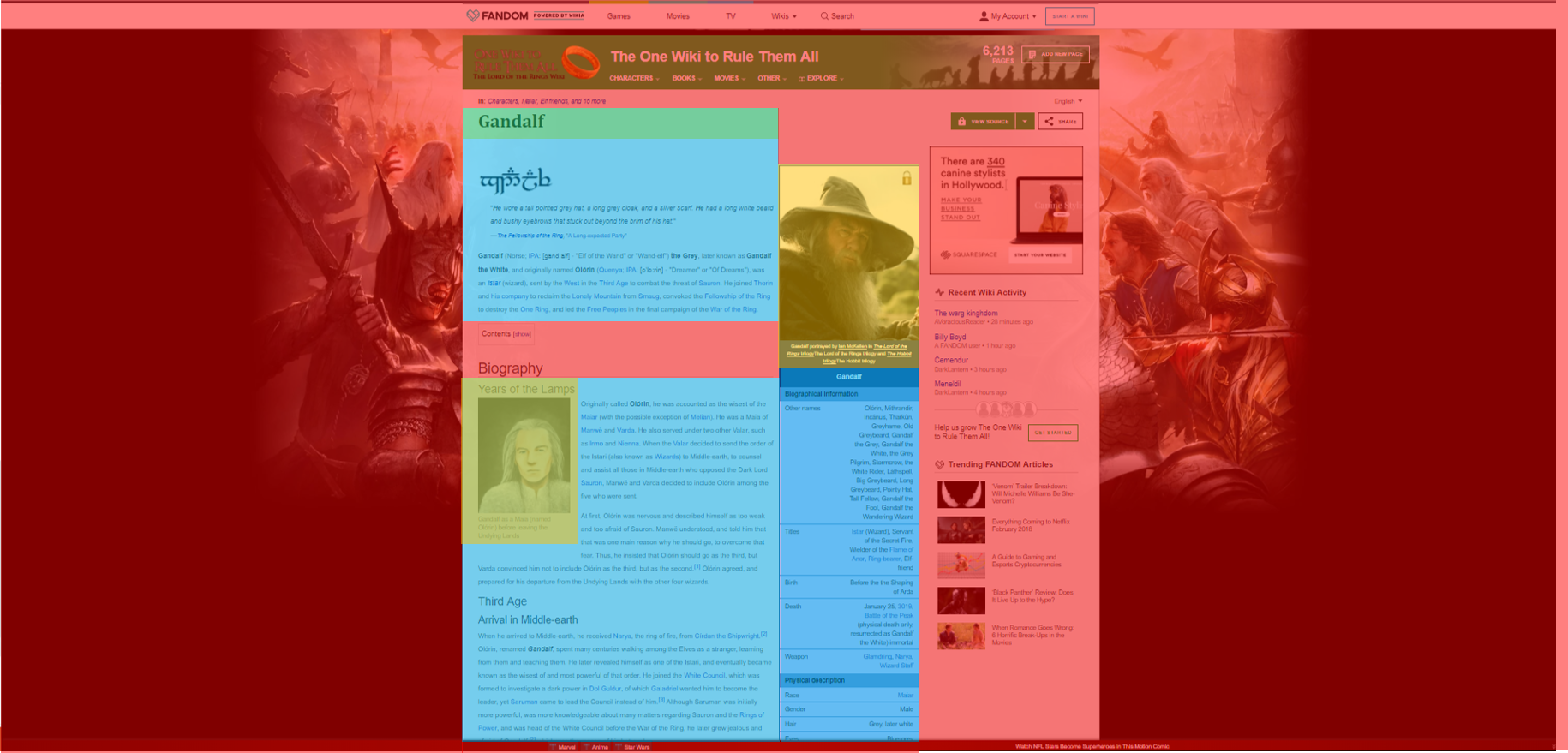
**Figure 1: Crawler Architecture**



## Crawling Strategy

Crawler code starts with the seed page of <http://lotr.wikia.com>. Then it reads in the Robots.txt file. The code follows the crawling ethics set in the Robots.txt file by checking if crawling is disallowed or allowed. If crawling is disallowed, it skips over the page and if it is allowed, it invokes the logic to download the content of the page. Some of the core elements that are captured include the Title, URL, the main block of text content on the page, and the image metadata. Figure 2 below illustrates the elements that would be collected and indexed on an example page. Green indicates the title of the article, blue indicates the relevant content text, yellow indicates image data, and red indicates erroneous content that is not indexed.

**Figure 2: Indexed Content**



Crawler code checks for duplicate pages and skips over the web pages that have already been processed once.

Crawler code has been enhanced to do multi-threading as well to ensure that it operates on a multiple threads thereby increasing the performance of the crawler.

Time taken to process the content download on each page is recorded by getting the difference between the start and end time before and after invocation of the java code that invokes the downloader. As the crawler code is running, user will be able to witness the URL of every page being processed, depth of the page, disposition of the page to indicate if it has been skipped or not, the name of the thread being processed, and the download time taken by the crawler.

Get and Set methods have been coded for the core elements that need to be captured by the crawler.

Output of the Crawler code is a series of XML files. Currently, the design is kept at the most granular level so that each XML contains only 1 document. This can be modified so that multiple documents fit into a single XML if the need arises in the future. Each document that is downloaded from the crawler will contain all the data elements that are needed for indexer so that keyword search functionality can function well during the search engine phase of the project.

# Lucene Indexing Strategy Overview

## Indexing Architecture

Indexing Architecture has been designed so that it takes the XML files put out by the Crawler as input. Indexing logic will create index files using Lucene framework (open source java based indexing and search library). In addition, Indexing logic will put out a CSV file containing the time taken by the indexing process on each document. Figure 3 below outlines this architecture.

Core of the indexing logic revolves around the usage of Lucene framework which does the bulk of the work to create the index files. The appropriate lucene java classes (analyzer, document etc.) had to be used along with the applicable methods within those classes (getdocument, adddocument etc.) by invoking the IndexWriter method.

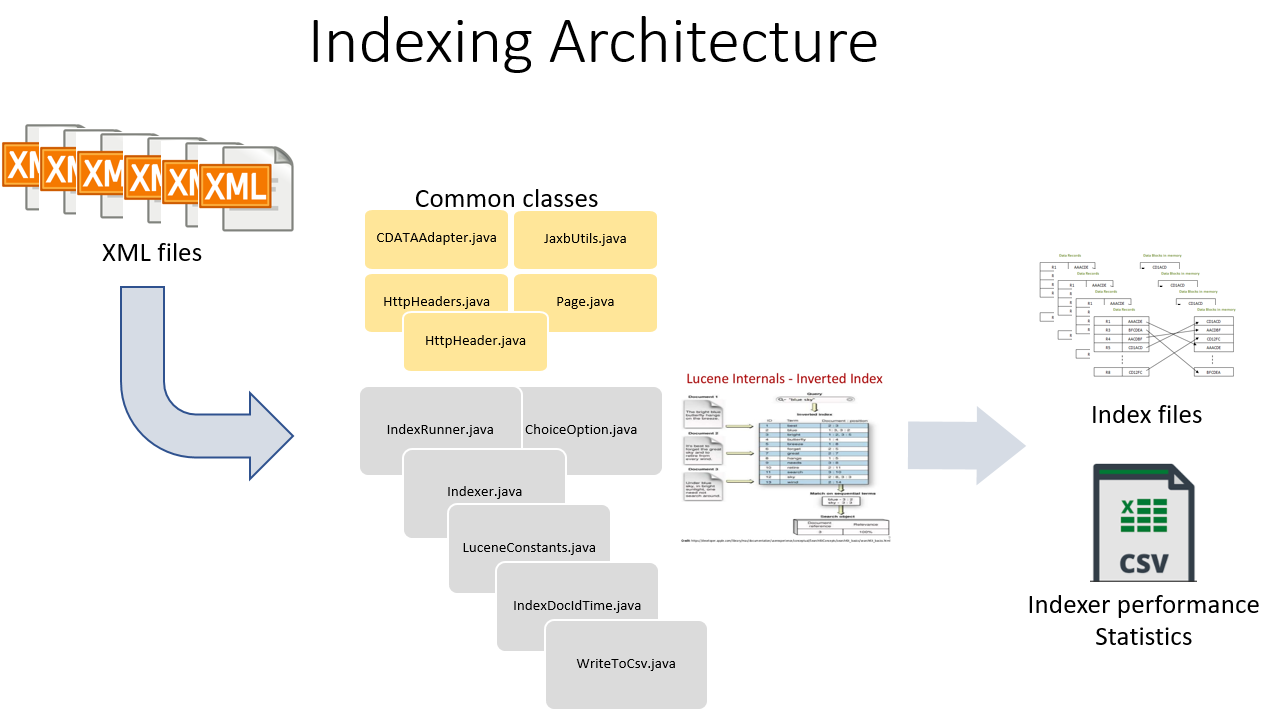
Following fields are indexed:

1. url
2. normalizedUrl
3. title
4. depth
5. content
6. headers
7. images

Indexing process has the logic to capture the processing time taken to index a document. Times are captured at the most granular individual document level. The team felt that this gives the maximum flexibility in terms of how to graphically report the run time numbers for the indexer against the number of documents processed. In addition, it was felt that this strategy would come in handy in the future phase of the project when comparison needs to be made between the performance of the indexer with lucene in a regular single processor environment against the performance in a distributed environment like Hadoop.

Code has been optimized so that the Indexer logic will write to a List in Memory to record the document name and the time taken to process that document. Once the Indexing process is complete, it dumps the list to a CSV file. This has been done so that the code doesn’t have to open, write and close the csv file multiple times.

**Figure 3: Lucene Based Indexing Architecture**



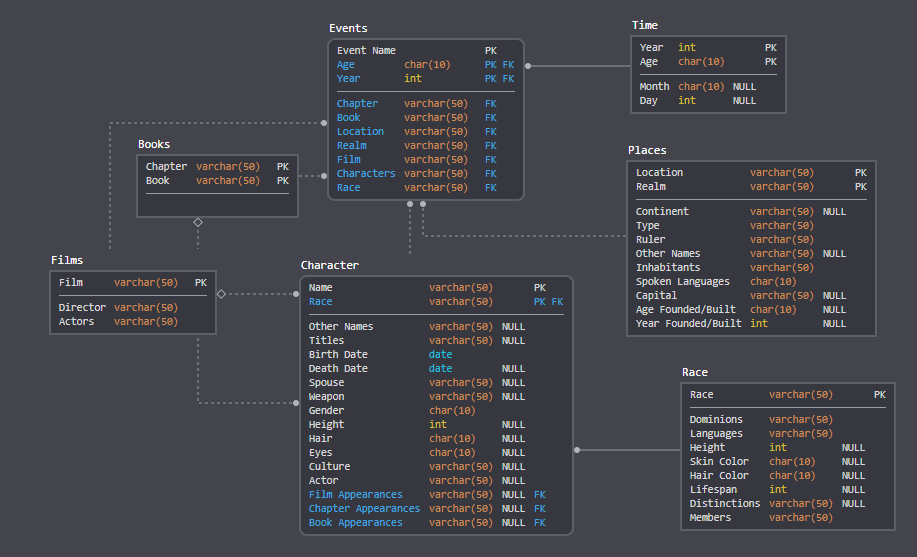
## Text Analyzer choice

Stop Analyzer was invoked and used within the Indexing code to ensure that all the unwanted keywords (example: and, the, or etc.) that won’t add value to the search engine in the later phase of the project are dropped from flowing through the indexing logic and don’t get indexed.

## High-level Data Model of Potential features from LOTR IR

Figure 4 below outlines the high-level data model for the proposed features that will be captured by the crawler of the LOTR wiki link.

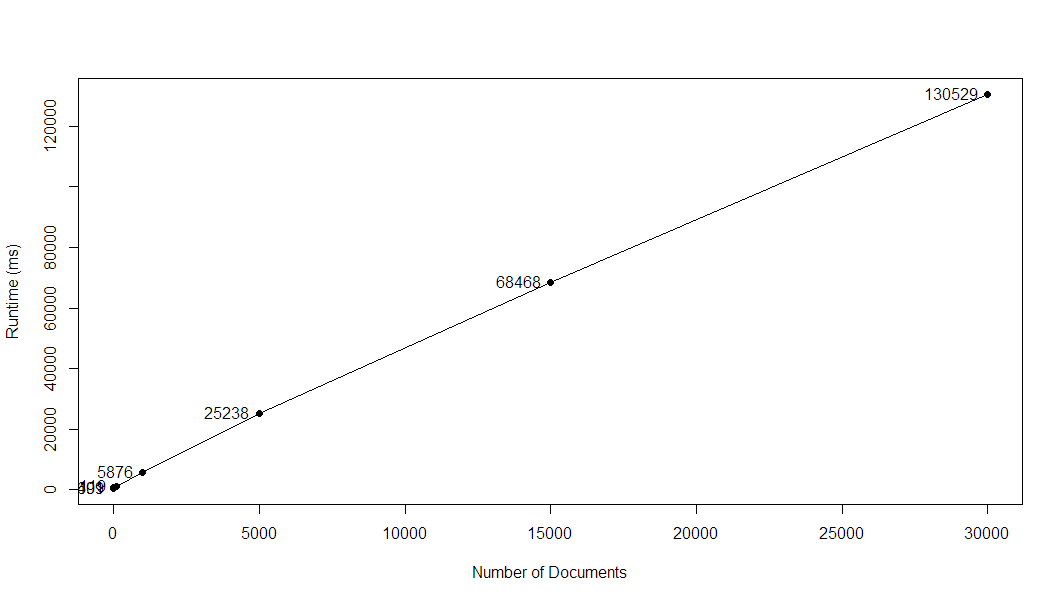
**Figure 4: Proposed Data Schema**



## Index creation process run time results

The runtime of the indexer was recorded during the indexing process up to the indexing of 32,563 files. The results of the indexer performance are recorded in the graph in Figure 5 below, and the table of selected values below it. Indexer performance appeared to be pretty consistent even at an increasing number of documents indexed. The time per document in the steady state seems to converge at around 4.5 ms per document.

**Figure 5: Indexer Performance vs Number of Documents**



|  |  |
| --- | --- |
| Number of Documents | Runtime(ms) |
| 1 | 491 |
| 10 | 609 |
| 100 | 1119 |
| 1000 | 5876 |
| 5000 | 25238 |
| 15000 | 68468 |
| 30000 | 130529 |

# System Limitations

Since the code runs only on a single machine and not in a distributed environment, the performance of the crawler and indexing process to complete dumping the 5GB worth of data takes a few hours.

For this release of the System, java 8 is the version that needs to be used. It was found that java 9 doesn’t work with certain code components.

# Obstacles and solutions

All the team members being new to the collaborative team environment for an Online course had to come up with innovative ways to communicate effectively and achieve the common goal of getting the project executed successfully.

Team decided that it is best to use an Integrated Development Environment for this project. So, it was decided to use open source IntelliJ IDE which proved to be immensely useful.

In addition, a few initial challenges on how to share the code with each other was overcome by the decision to use github which helped immensely in coordinating various releases of the code through the execution of the project.

Capturing each of the proposed data features (~40 fields), upon deeper thought into the indexing logic behind it, proved to be too high risk for an unnecessary reward. Treating all important content on the page as a block of text should be sufficient, and it should also help with compute time. Data from these fields may be separated later upon visualization of the data set as much can be done with this wealth of data.

Some of the team members used Mac to create the java code that wasn’t fully functional in a windows environment. So, code had to be adjusted to make sure it is functional in both environments. The issues were related to the following:

1. Gzip xml input files to the indexer were not getting processed successfully in the windows environment with default parameters. Code had to be adjusted so that UTF-8 sequence error doesn’t occur while indexer tries to process the gzip files.
2. Folder structures referenced in linux or mac environment use “/” and windows environment expects this to be a “\”. Code had to be adjusted to account for this.

When we did performance testing, it was found that the crawler code can be tuned for better performance. Significant effort was put in to revamp the crawler code to perform better. The issue was related to the fact that only 1 page per second was being downloaded despite the number of threads. This was tuned so that crawler could download n pages where n = number of threads.

# How to deploy the crawler and how to build the Lucene Index

To download the code for the project please run

Git clone <https://github.com/jorgemariomercado/cs242-crawler.git>

Attached make file has been created with details on how to compile and run the crawler and indexer.

