CS 242: Information Retrieval & Web Search

TV Series Search Engine

PART A(Crawling and Indexing)

Group 20:

Aishwarya Pagadala

Avinash Reddy Kummeta

BalaSanjana Thumma

Prudhvi Manukonda

Shivaji Reddy Donthi

PROBLEM STATEMENT

In this project we aim to build a TV-Series search engine using Wikipedia pages as the knowledge base. We built a multi-threaded Web crawler using JSOUP that recursively traverses through webpages using a given link and indexes them on the disk. We used Apache Lucene to index the documents iteratively, and fetch results for user queries.

COLLABORATION DETAILS

We have divided the work into 2 parts - crawler Implementation and lucene indexing.

- Crawler Implementation: Avinash Reddy Kummeta, BalaSanjana Thumma and Prudhvi Manukonda
- Lucene Indexing and Search Query Implementation - Aishwarya Pagadala and Shivaji Donthi.

OVERVIEW OF THE CRAWLING SYSTEM

Crawler is multithreaded so for the below architecture we assumed that there is one main crawler thread and 4 child threads.

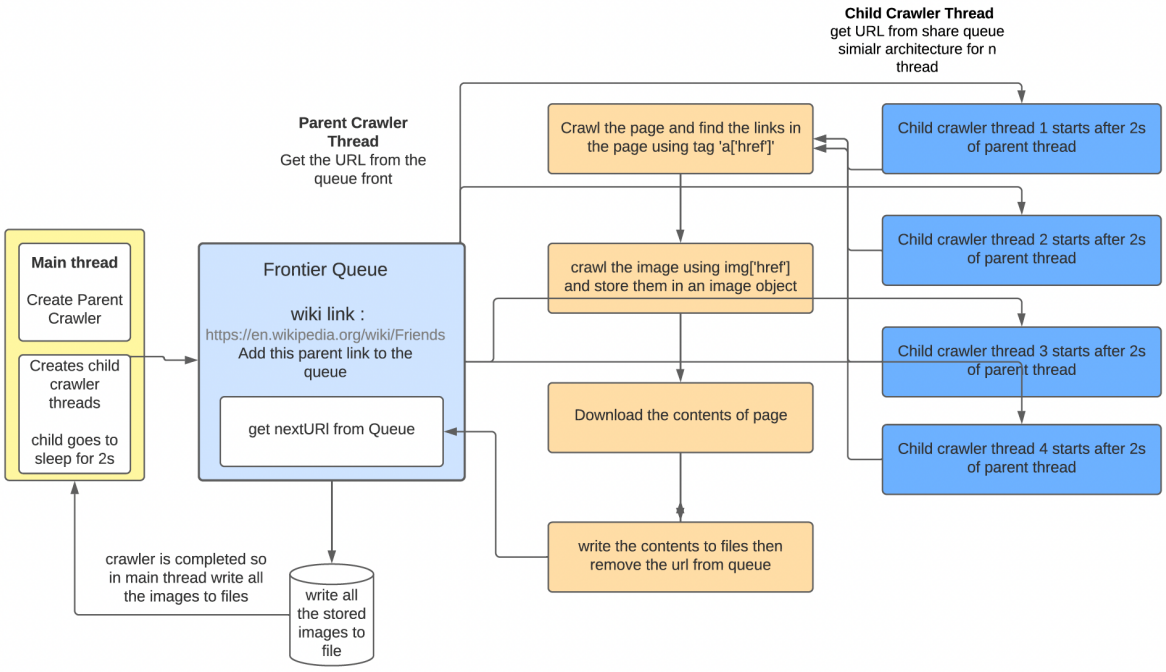
Each crawler thread will parse page data , links and image urls to download data.

In the beginning of the execution parent crawler thread starts first and then child threads start execution after 2-3s so as to give time to parent thread to fill the shared queue.

In the diagram parent thread is represented using orange color and child threads using blue which will follow the process same as that of parent

After crawling is done we have image objects, urls and text stored in a list so in the main thread we write the downloaded images to the file system and then the program completes.

1) ARCHITECTURE :



2) THE CRAWLING STRATEGY

1. For Crawling we are using the wikipedia main page for friends tv show -> https://en.wikipedia.org/wiki/Friends
2. When the main program starts we will check weather the directory provided is clean, if not we will clean it
3. The crawler program is implemented in a multi threaded manner
4. First, the main thread will create a parent crawler (**MultiCrawler**) thread which invokes the constructor of the crawler parent thread.

   The parent crawler initialize the following

   1. static queue for multiple threads to access
   2. Hashmap of visited links to check for duplicates
   3. Counter to keep track of the number of pages to crawl
   4. Max_depth is the total no of pages to crawl

```java
private static String URL;
// Queue of links
static Queue<String> links;
// visited link
static HashMap<String , Integer> visitedLinks;
static ArrayList<ImageClass> imageStore;   // list to store image object
static Integer counter;
static Integer MAX_Depth;
WriteToFile w;
Thread mainThread;

// main constructor for parent thread
public MultiCrawler(String URL , String dirPath , Integer Depth) {
    this.URL = URL;
    links = new LinkedList<>();
    visitedLinks = new HashMap<>();
    counter = 0;
    MAX_Depth = Depth;
    links.add(URL);
    w = new WriteToFile(dirPath);
    imageStore = new ArrayList<>();
    mainThread = new Thread( target: this);
    mainThread.start();
}
```

```java
@Override
public void run() {
    System.out.println("Crawler Started");
    startCrawl();
    //w.imageStoreWrite(imageStore);   // after running the crawler download images
}
```

The main thread constructor will invoke the **run** method and it will start the crawler method.

5. Next main method will create child threads of the **MultiCrawler** thread which invokes the overloaded constructor, which invokes the run method of the thread and child threads will start crawling

```
// overloaded constructor for child threads
public MultiCrawler(String dirPath) throws InterruptedException {
    w = new WriteToFile(dirPath);     //
    mainThread = new Thread( target: this);
    mainThread.sleep( millis: 5000);
    mainThread.start();
}
```

6. For child threads we invoked the sleep method to give time to the parent thread to start adding links to the queue which can be picked by the child threads when they come out of sleep.

7. In crawler we loop until the shared queue is empty and parent and child crawler threads will crawl the links from the queue front (aslo polling and peek from the queue is synchronized to avoid overlapping conditions b/w threads)

```
public void startCrawl() {
    System.out.println("------------- Thread " + this.getThread().getName() + " started crawling ----------------");
    try {
        while(!links.isEmpty() &&  counter < MAX_Depth) {
            // https://stackoverflow.com/questions/24475816/jsoup-404-error - handling error
            String peekLink = getPeekLink();   // synchronised method
            System.out.println("thread :" + getThread().getName() + " -  link: " + peekLink + " counter: " + counter);
            Document doc = getDocument(peekLink);
            if(doc != null && isUrlValid(peekLink)) {
                for(Element link : doc.getElementsByTag( tagName: "a")) {
                    //String innerLink = link.attr("abs:href").trim();
                    String innerLink = formatURL(link.absUrl( attributeKey: "href").trim());
                    String iLink = link.attr( attributeKey: "href");

                    if(!visitedLinks.containsKey(innerLink) && !innerLink.isEmpty() && !peekLink.contains(innerLink) &&
                        links.add(innerLink);
                        visitedLinks.put(innerLink , 1);    // need to stem this for checking duplicates
                    }else {
                       // System.out.println("Visited link : " + innerLink + " skipping the link");
                    }
                }
```

8. Each crawler thread will parse the document and extract the links and store them into a queue.Also crawler will parse the image links and create image objects (image link , alt).

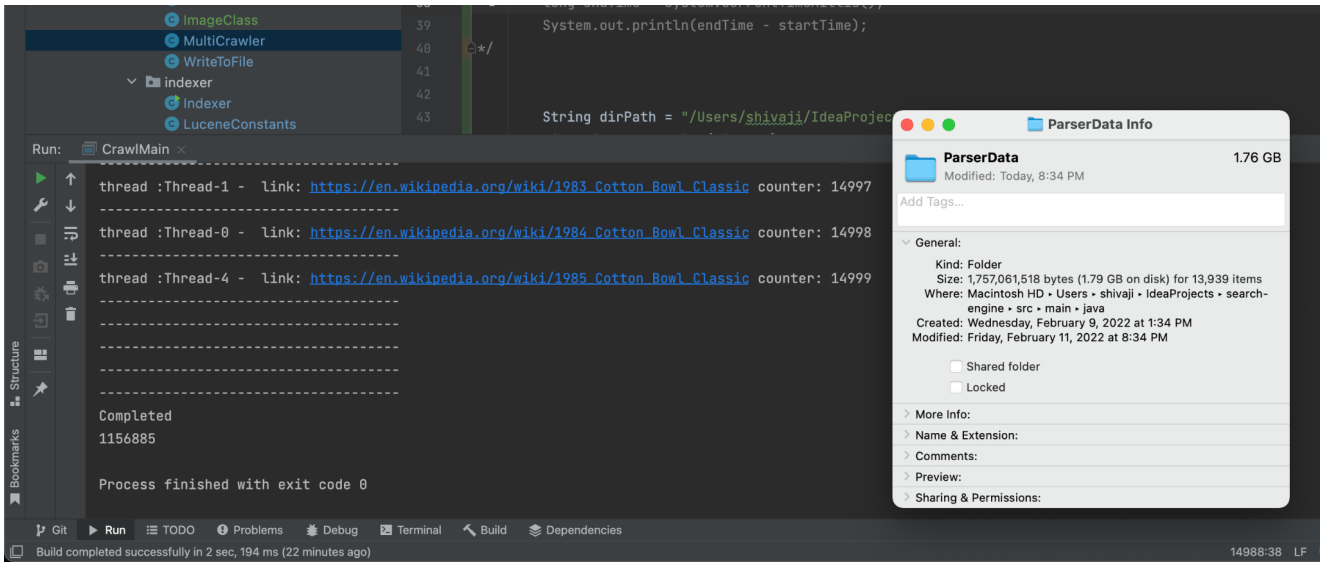
9. Then the crawler will download the body of the page and store them into documents , which is similar for images.

3) CRAWLING RUNTIME

To collect 1.67GB of data, the crawler took about 20 minutes (1156885 milliseconds).

To ensure that the crawler doesn't jump into other websites as the we go deeper levels, we used a filter on the link that verifies whether the link url contains the string wikipedia

This filter innerLink.toLowerCase().contains("wikipedia")ensures that the knowledge base is built strictly on wikipedia pages.



The crawler is run on 4 threads and took 20 minutes to scrape 13,939 documents with a total size of 1.5GB.

OVERVIEW OF THE LUCENE INDEXING STRATEGY

1) FIELDS USED IN THE LUCENE INDEXING

There are a lot of elements in a scraped wikipedia page but the majority of the useful information is contained within the title and paragraph elements. Apart from paragraphs, we have also indexed the input document filepath so that we can display this link in the query results as the resultant page.

The following are the Indexed Fields :

[1] Titles - “title”

We extracted all the title tags content under each anchor tag and used a stringBuilder to append all the content and stored it as a value against the “title” index.

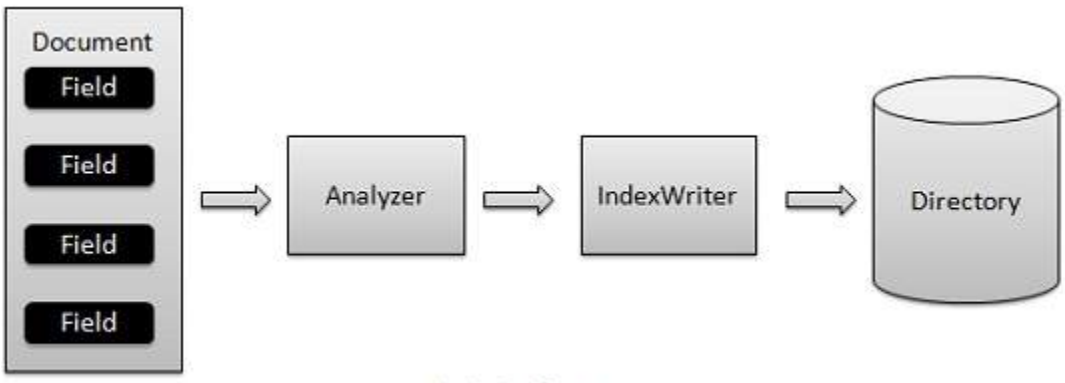
[2] Paragraphs - “para”

Since there are multiple “p” tags in a webpage, we combined all the paragraph content in a string and used it as the value for “para” index.

[3] File Link- “path”

This is the input file path of which we extracted the content for indexing. Although we did not use this index field in the query search criteria, We added this field as an index to our document so that we can retrieve the link to the original txt file for the highest score.

Since all the raw html data was stored as a text file, We used JSOUP to parse the text inside document as DOM and retrieved tag content of the html by tag names.



Indexing Process

2) TEXT ANALYZER CHOICE

Although we did experiment with a Custom Analyzer, we used Lucene StandardAnalyzer() like a due to its better and simpler fit for the job of indexing.

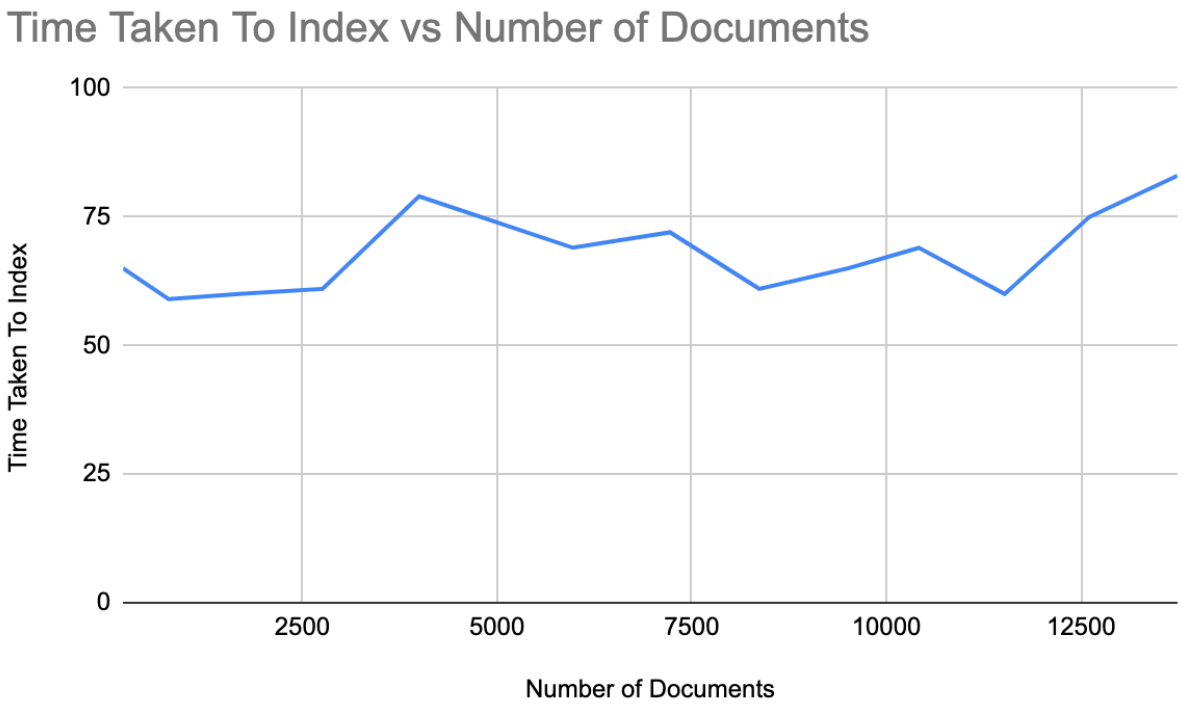
Standard Analyzer in Lucene converts each token to lowercase, removes common words and punctuations, if any. This helps in obtaining better hits for the search query.

3) RUNTIME OF THE LUCENE INDEX CREATION PROCESS

There were a total of 13939 files in our Parsed Directory after crawling.

To analyze how much Lucene Indexing takes as we increase the number of documents, we increased the number of documents of to be indexed gradually from 200 to 13939 by iteratively increasing the number of files by about 1000.

Number of Documents	Time Taken To Index
200	65
786	59
1692	60
2760	61
4000	79
5975	69
7224	72
8370	61
9507	65
10418	69
11518	60
12607	75
13737	83



As it can be seen above the time taken not strictly linear with respect to the number of documents but as the number of documents to increase the Index creation time does seem to increase.

4. Storage of Lucene Indices:

We used FSDirectory to write the created lucene indices to an output folder. This folder directory is then used as an input to the DirectoryReader and IndexSearcher, using which we find the document scores for a given query with the help of Query Parser.

SEARCH QUERY RESULTS

Here, we take a search query string from the user, using which we find the top 100 documents in their increasing order of score. Since we are indexing a particular document using more than one field, i.e title, paragraph and file path, we used MultiQueryparser to retrieve the results for the search query with respect to all title and paragraph indexed fields. We did not use file path as a parameter to the MultiQueryParser search field. We used it to retrieve the original document link path, using which we can go to the document which has the high scores.

Below are the search results for a sample search query string “Lisa Kudrow”. We get the top 100 scored documents - document ID, score and the path to the original text file using which the index was created. We have also displayed a preview of file content in the results.

For detailed information, we used the explain function of the query parser, which gives step the details on calculating the score for each resultant document.

```
Time taken for Indexing: 54milliseconds

Rank-[1]  doc=201 score=2.4954057 shardIndex=0
Document Path: /Users/shivaji/IdeaProjects/search-engine/src/main/java/ParserDatatemp/document_57.txt
Page Content Preview:  Phoebe Buffay is a character portrayed by Lisa Kudrow, one of the six main characters from the American sitcom Friends, created by
-----------------------------------------------------------------------------------------------------------------------------------------------------------
Rank-[2]  doc=68 score=2.460887 shardIndex=0
Document Path: /Users/shivaji/IdeaProjects/search-engine/src/main/java/ParserDatatemp/document_11.txt
Page Content Preview:  Lisa Valerie Kudrow[1] (/'ku:droʊ/; born July 30, 1963)[2] is an American actress, comedian, writer, and producer.
-----------------------------------------------------------------------------------------------------------------------------------------------------------
Rank-[3]  doc=72 score=2.3202615 shardIndex=0
Document Path: /Users/shivaji/IdeaProjects/search-engine/src/main/java/ParserDatatemp/document_103.txt
Page Content Preview: "The One with Ross's Wedding" is the two-part fourth-season finale of the American television sitcom Friends, comprising the 96th a
-----------------------------------------------------------------------------------------------------------------------------------------------------------
Rank-[4]  doc=69 score=2.2302406 shardIndex=0
Document Path: /Users/shivaji/IdeaProjects/search-engine/src/main/java/ParserDatatemp/document_10.txt
Page Content Preview:  Courteney Bass Cox (previously Courteney Cox-Arquette; born June 15, 1964)[5][6] is an American actress, director, and producer.
-----------------------------------------------------------------------------------------------------------------------------------------------------------
```

```
Rank-[1]  doc=201 score=2.5001264 shardIndex=0
Document Path: /Users/shivaji/IdeaProjects/search-engine/src/main/java/ParserDatatemp/document_57.txt
Page Content Preview:  Phoebe Buffay is a character portrayed by Lisa Kudrow, one of the six main characters from the American sitcom Fr
Query Result Explanation:
2.5001264 = sum of:
  1.0787765 = weight(title:lisa in 201) [BM25Similarity], result of:
    1.0787765 = score(freq=4.0), product of:
      1.2462212 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
        120 = n, number of documents containing term
        418 = N, total number of documents with field
      0.8656381 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
        4.0 = freq, occurrences of term within document
        1.2 = k1, term saturation parameter
        0.75 = b, length normalization parameter
        792.0 = dl, length of field (approximate)
        2221.469 = avgdl, average length of field
  1.4213499 = weight(para:lisa in 201) [BM25Similarity], result of:
    1.4213499 = score(freq=5.0), product of:
      1.6250726 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
        82 = n, number of documents containing term
        418 = N, total number of documents with field
      0.8746378 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
        5.0 = freq, occurrences of term within document
        1.2 = k1, term saturation parameter
        0.75 = b, length normalization parameter
        2072.0 = dl, length of field (approximate)
        4475.675 = avgdl, average length of field
```

LIMITATIONS OF THE SYSTEM

1. Since crawler is multi threaded we cannot really decide on how many threads we need to invoke which can optimally parse the urls, so ideally when thread count is b/w 3-7 our crawler is good but beyond that there is some issue while incrementing the counter which is used to keep track of Max depth.
2. We are using a counter variable to assign filename to each document and the counter is incremented using synchronized method.But there is a situation where another thread is writing the data to file while another thread increments the counter in the meantime so because of this we see in output document names some counts from the document file name will be missing.
3. While writing downloaded images to files using the image links some of the images cannot be opened (not sure of the problem we tried to debug it but since it is a stream it was difficult for us to debug the problem).

OBSTACLES AND SOLUTIONS

1. When a crawler extracts links from the page it will check if the page is already crawled using the visited Links stored in hashmap.
2. Also we did some checks to the url link before parsing
    a. We check if a connection can be established using jsoup to the page if the response status code is other than 200 we will not parse the page and remove it from the queue.
    b. Also we will format the link and remove the navigation links (starts with "#" #/link) -> which on crawling redirects to the same page (which avoids crawling the page multiple times) and

       query parameter ("?/") -> same pages with different query parameters have almost similar content in wikipedia. By removing this duplicates can be eliminated.

3. We encountered some exceptions like a malformed **exception** and **socket timeout exception** which will stop the application so we did some checks to the url using URL package which checks whether the url is valid and can be parsed.
4. In crawler we tried to filter the links using links but we have seen some links which are redirecting to the same page even though the absolute URLS are different (an Example related to this is mentioned below)

    1. https://en.wikipedia.org/wiki/List_of_Friends_and_Joey_characters

    2. https://en.wikipedia.org/wiki/Emily_Waltham

    3. https://en.wikipedia.org/wiki/Mike_Hannigan

Links 2&3 will direct to the link 1 even though both links are different

5. While displaying the query results, Lucene only returns the document ID and its path where hit had occurred. But because Wikipedia pages are so long and there is a chance that a particular search query has occurred in multiple places that

INSTRUCTIONS OF DEPLOYING CRAWLER AND LUCENE INDEXING

1. To run the crawler run the crawler script file ./crawler.sh located in src/cralwer directory

$:./cralwer.sh <wikipedia-url> <output-directory-path> <depth - number of documents to crawl> <number of threads>

Arguments:

<wikipedia-url> - eg: https://en.wikipedia.org/wiki/Friends

2. To run the Lucene Indexer run the indexer script file ./indexer.sh located in src/indexer directory

$:./indexer.sh <input-directory> <outputdirectory> <query string>

Arguments:

#java indexer <input directory> <outputdirectory> <query string>

#<input directory> - location where crawler stored documents

#<output directory> - location where Indexer should store indexed files of FSDirectory()

#query string - query string

java indexer $1 $2 $3