Search Engine Project - Final Phase Report

Group 52 – CHEN, Jiawei (jchenfq@connect.ust.hk)

1 Overall System Design

The overall search engine system consists of four modules: crawler, indexer, retrieval engine and web user interface.

1.1 Crawler

The crawler is responsible for recursively getting all the page content and link information starting from a root URL provided by a user using BFS algorithm. For this project, we utilized the Jsoup library for establishing HTTP connection to a webpage and manipulating the returned response object that contains all the document information we need. The extracted document information will go through a pre-processing pipeline with stopword removal and stemming, then are stored temporarily in memory, and forwarded to the indexer. The details about BFS Crawling the pre-processing pipeline will be described in the **Algorithms** section.

1.2 Indexer

The indexer is responsible for storing all the pre-processed document information extracted by the crawler into multiple designed database schemes. It utilizes JDBM to support persistence of large object collections in a faster and simple manner. To improve transactional efficiency, instead of indexing one extracted document's information at a time during crawling, the crawling and indexing operations are executed in a sequential manner. After all page information are crawled and stored temporarily in memory, they will then be indexed once for all. The detailed description about the database schemes will be discussed in the **File Structures** section**.** And for the detailed indexing pipeline, see **Algorithm** section.

1.3 Retrieval Engine

The retrieval engine is responsible for calculating the similarity scores between a search query and the indexed documents in database. The documents will be sorted in descending of similarity scores and returned as search results. Specifically, the scoring is based on cosine similarity scores between the query vector and the document vector. These document vectors are constructed at program startup by a Vector Space Model (VSM) which utilizes database information for computing the vectors while the query vector is computed in real

time when the retrieval engine receives a user's search query. The VSM also has a matching mechanism that favours matching in title of documents when encoding the query vector. The retrieval engine supports two different search mode: simple keyword search and phrase search. For the details of VSM and two search modes, see **Algorithms** section.

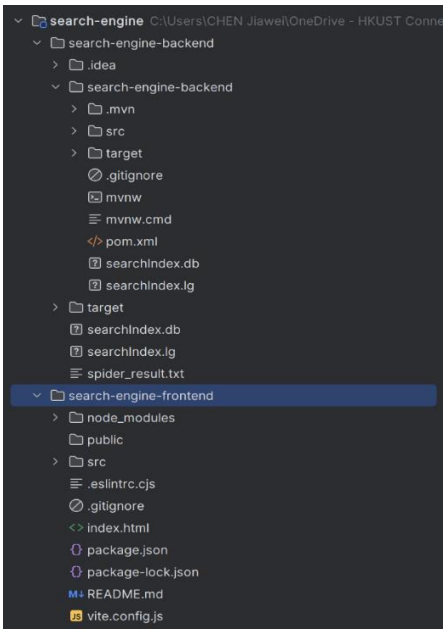
1.4 Web User Interface

The web user interface is responsible for accepting user's search query and send the search request to the backend server. It then takes the response from serve containing all search results and display them. We utilize React for building the web user interface and TanStack Query for handling the communication between frontend and backend server.

2 File Structures

2.1 Project Repository Setup

The project is split into two child directories containing the frontend web user interface and backend server separately. Both have the main codes in the 'src' folder as showed below:



2.2 JDBM Database Schemes

This project has in total six different index and they are all implemented using the HTree library from JDBM. Since the JDBM Database Schemes have the **same design as Phase 1.** The following will only show the design without any explanation. For more information on the detailed design of JDBM Database Schemes, please refer to the Phase 1 JDBM Database Scheme Design documentation.

2.2.1 Page ID Lookup Index

Page ID Lookup Index	
key	URL: String
value	Page ID: Integer

2.2.2 Keyword ID Lookup Index

Keyword ID Lookup Index	
key	Keyword: String
value	Keyword ID: Integer

2.2.3 Keyword Lookup Index

Keyword Lookup Index	
key	Keyword ID: Integer
value	Keyword: String

2.2.4 Forward Index

The Forward Index stores a page's ID as key and corresponding Page object as value. It maps all Page IDs to corresponding Page objects which contain the details of crawled pages.

Forward Index	
key	Page ID: Integer
value	Page: Page Object

Page
title: String url: String size: String lastModification: Date parentUrls: List<String> childUrls: List<String> titleFrequencyTable: Map<Integer, Integer> bodyFrequencyTable: Map<Integer, Integer> titleWordPosition: Map<Integer, List<Integer>> bodyWordPosition: Map<Integer, List<Integer>>

2.2.5 Inverted Index

As the project requirement specified, we need to store the stemmed body words and title words into two inverted files:

Body Inverted Index

Body Inverted Index	
key	Body Keyword ID: Integer
value	Page List: Map<Page ID: Integer, Body Word Position: List<Integer>>>

Title Inverted Index

Title Inverted Index
key Title Keyword ID: Integer
value Page List: Map<Page ID: Integer, Title Word Position: List<Integer>>>

3. Algorithms

3.1 Crawler

3.1.1 BFS Crawling

The crawler uses breadth-first strategy (BFS) crawling where it starts with the root URL and fetch all the URLs in the current depth before moving on to fetch child URLs pointed from the current fetched URL. This allows the crawler to fetch the webpages layer by layer. This algorithm is implemented using two array lists, one for storing the unvisited URLs and the other for storing all visited URLs. At the start of crawling, the root URL is added to the unvisited list and a while loop begins. For each iteration of the loop, it extracts the first URL from the unvisited list and checks if this URL is present in the visited list. It will continue and ignore current URL if it is present in the visited list to handle cyclic links. Otherwise, it will extract the needed document information of current URL and store the URL in the visited list, while adding all its child URLs into the unvisited list. The loop stops when there's no more URLs in the unvisited list for crawling.

3.1.2 Pre-processing Pipeline with Stopword Removal and Stemming

During crawling, we get the webpage document from the HTTP response and extract the title keywords and body keywords of the document along with other needed document information such as last-modified date, the parent/child links, size, etc. The extracted keywords (both from title and body) will be processed with Stopword Removal and Stemming. This is done by the StopStem class where a list of pre-defined stopwords stored in a stopwords.txt file is loaded in the memory and used as a lookup table to filter out any matched stopwords from extracted keywords. For stemming, we use the open-sourced Porter's algorithm implementation to stem on the remaining keywords. After these two operations, we will construct a frequency table for the stemmed keywords in title and body separately using HashMap. In addition, a keyword position table will also be constructed for the stemmed keywords in title and body separately using HashMap for later phrase search usage. At the end, all extracted document information will be stored in a Crawled Page object. All Crawled Page objects will be stored in a array list in memory and forwarded to the Indexer for indexing.

3.2 Indexing Pipeline

1. To create or update the indexes after crawling, we need to first find the Page ID through

the **Page ID Lookup Index** by using the URL of crawled page. ***2.*** If the Page ID does not exist, we need to assign a new Page ID to the URL and update the **Page ID Lookup Index** with new <URL - Page ID> entry and **Forward Index** with new <Page ID - Page object> entry. We will also update the **Keyword ID Lookup Index** and **Keyword Lookup Index** with those new keywords inside this new page. ***3.*** If the Page ID already exists, we need to check the last modification date and only update the **Forward Index** with modified Page object if the stored one was outdated. We will check if any child pages becomes unreachable after replacing and update their parent link field accordingly but not delete them from the indexes as clarified by the project specification. The **Keyword ID Lookup Index** and **Keyword Lookup Index** will also be updated with any changes in keywords inside the modified page. ***4.*** After updating the **Page ID Lookup Index** and **Forward Index**, we will update the **Inverted Index**[1] by checking each keyword inside the indexed pages. ***5.*** For each new keyword, a new entry will be added to the **Inverted Index** by mapping the keyword to a list of Page IDs which contains the keyword. Each of the Page IDs will be further mapped to a list of keyword positions in that page. ***6.*** For each keyword that already exists in the **Inverted Index**, we will check if the associated page IDs already exist on the map. ***7.*** For each new page ID, we will simply add the Page ID and keyword position list to the map. ***8.*** For each page ID that already exists on the map, we will replace the indexed keyword position list with the updated one. ***9.*** End database indexing/updating.

3.3 Retrieval Engine

3.3.1 Vector Space Model

The vector space model precomputes all the vector representations of keywords in each document at the program start to enable faster retrieval. The constructed document vectors are stored in a HashMap with their document ids as key. Each document vector is a HashMap of unique keywords in that document as keys and their terms weights as values. To favor title matches, a larger term weight offset (1.5) is used for keywords in title. Thus, the term weight for a keyword in title is thus 1.5 x tf x idf while for keyword in body is simply tf x idf, where here tf is normalized with the maximum term frequency in that document. And the result term weight for a keyword will combine both term weight in title and term weight in body. When calculating the document-query similarity, the score will be normalized with respect to the document length and query length.

3.3.2 Simple Keyword Search

This is the default search mode for the retrieval engine if the search query is not enclosed by double quotes. Before calculating the document-query similarity, the search query will go

[1] Refer to both Title and Body Inverted Index, since the processing pipeline for these two are the same, in this section we call it Inverted Index for short.

through the same pre-processing pipeline with Stopword Removal and Stemming during crawling and transform into a query vector. Then, for each keyword in the query vector, we will find the set of document ids that contains this keyword by utilizing the inverted index. Then, the union of the matched/found document ids will be treated as the retrieved documents and proceed to calculate their document-query similarities. This helps to narrow down the scope of search from the entire document collection to a selected subset that contains any one of the query keywords.

3.3.3 Phrase Search

As described in the **File Structures** section, our inverted index includes extra information about the keywords' positions in each document. This allows us to detect if a phrase with specific keyword order appears in a document. To enable phrase search mode, just enclosing the search query with double quotes. Before calculating the document-query similarity, just like the **Simple Keyword Search**, the search query will also go through the same pre-processing pipeline with Stopword Removal and Stemming during crawling and transform into a query vector. To find **exact matches**, we will first find the set of document ids that contains all the query keywords, this can be done by set intersection operation and utilizing the inverted index again. Then, check if the order of keywords in these documents matches the order of query phrase. To do that, we apply different positional offsets to the keywords in documents based on the keyword order of query phrase. For example, if a search query phrase is "keyword_0 keyword_1 keyword_2", it will first be transformed into query vector [StopStemKeyword_0, StopStemKeyword_1, StopStemKeyword_2]. Now suppose we have a document containing these three keywords and we obtain their positions from the inverted index where StopStemKeyword_0 = [34], StopStemKeyword_1 = [35, 77] and StopStemKeyword_2 = [36, 45, 89]. The positional offset for StopStemKeyword_0 would be 0 since its index in the query vector is 0, and the offset for StopStemKeyword_1 and StopStemKeyword_2 would thus be 1 and 2 respectively. The offset adjusted positions of these three keywords in document would become StopStemKeyword_0 = [34], StopStemKeyword_1 = [34, 76] and StopStemKeyword_2 = [34, 43, 87]. We can observe that they share a common position 34 and this indicates that the query phrase does appear in this document. The implementation of the above observation would be using set intersection on the position arrays and check if the result set is empty or not. To find non-exact matches, we essentially perform **Simple Keyword Search** on the query. The exact and non-exact matched page ids will be stored separately and retrieved documents, then forward to calculate their document-query similarities. The result exact and non-exact matched pages will be returned to frontend separately where exact matches will display above the non-exact matches.

3.3.4 Mixed Search

The mixed Search can accept a query containing exactly one phrase and several other keywords. It returns a list of exact matched results and non-exact matched results. For exact matched results, it means that the returned pages must contain the phrase and also contain at least one of the extra keywords. While the non-exact matched results means that the returned pages must contain the phrase, but they do not contain any of the extra keywords. For example, if we search '"fatal beauty" women', the **movie index page** would be exact match since it has the phrase "fatal beauty" and contains the keyword 'women' while the **fatal beauty page** would be non-exact match since it only contains the phrase "fatal beauty" but does not contain the keyword 'women'.

4.1 Relevance Feedback (Get Similar Pages)

To help users to find pages relevant to a specific page, the relevance feedback function is implemented. For each search result page, there is a 'Get Similar Pages' button associated. Upon clicking the button, the top 5 frequent stemmed keywords of the target page will be automatically extracted, concatenated, and put into the search input box. Then a user could click the search button send the query to backend search engine. The new set of retrieved pages would thus have higher similarity scores to the target page.

4.2 Stemmed Keywords List for Query Formulation

To facilitate query formulation, a list of available stemmed keywords is provided for user to examine. Once a stemmed keyword button is clicked, it will be appended to the search input bar and user could still type input manually to expand the search query. This helps to form a more accurate and relevant query compared to a manually created one. An additional parameter 'raw' is included for each search query. If any stemmed keywords are selected, this parameter would be set to false, and the backend retrieval engine will not perform stopword removal and stemming on the query. However, we did not implement detection and processing for separating stemmed and unstemmed keywords in a search query, so users can use either stemmed or unstemmed keywords but not mix them up in a search query.

4.3 SPA User Interface

Instead of using JSP to build up the frontend of this project. We utilized React to build up an intuitive and friendly SPA (Single Page Application) which allows dynamic loading of components according to the application state and the request endpoint. This eliminates the necessity to reload the whole page and gives a fast and reactive experience to the user. To bridge between the backend search engine and the frontend UI, a REST API is developed to

accept and return data in JSON format. For loading the content, such as the query result, a HTTP request call will be sent to the JSON API for getting the data using Tanstack Query. Then, the returned data will be parsed and displayed to the user accordingly. We also using Material UI for styling of components to give a more consistent and elegant display of the webpage. The search results are displayed in a neat and well-organized manner so that users can easily identify relevant pages. The components are also responsive to window sizes and applied with different animations to give a native user experience. A separate crawler interface is also implemented for the user to crawl different webpages. This also allows user to perform crawl before search to update the database manually.

4.4 Customization on Target Searching Section

We have implemented a UI for selecting the desired section of searching during simple keyword search or phrase search. A user can choose to search query terms appear in document title only, body only or both (appear in either of them). This helps users to narrow down the search range and perform a more accurate search. Note that is customization is applicable to all search modes (Simple Keywords Search, Phrase Search, Relevance Feedback Search and Stemmed Keywords Search).

4.5 Special Implementation Techniques

4.5.1 Fast Retrieval

To enable fast page retrieval, the term weights of all pages are precomputed and store in memory at the server start-up by using different index in the database. The range of similarity calculations is also narrowed down by ignoring documents that does not contain any of the stemmed query terms or phrase. This eliminates the overhead to calculate similarities for all documents in database and the size of data to send back to the frontend for display.

4.5.2 Index and VSM Updating Mechanism

The crawler is implemented in a way such that during recrawling, if a child page is modified but the parent page is not, it would ignore the parent page and update the index with modified child. The crawling will not end until all child pages are checked or the maximum number of pages to crawl (check) is reached. If a page is modified, not only the index will be updated, the Vector Space Model (VSM) holding all document vectors will also be updated dynamically. This ensures the index and VSM are up to date with all modifications detected and handled during crawling. With the updating mechanism and crawler user interface, user can dynamically update the database in the middle of searching during sever running.

5. Testing

Testing is primarily done through unit testing and functional testing. Each class has been tested during creation to ensure non-trivial functions are implemented correctly. Loggings are enabled in the backend server to record the operation process. Exceptions are properly handled and logged to server output through try-catch block for to ensure that the program does not exit ungracefully. Manual testing has been done on both crawler with indexer and retrieval engine. For crawler with indexer, we examine the results of indexed database after crawling on a small number of webpages (5 page) and manually calculate the theoretical word frequency, position, id count, etc. we then cross validate if the indexed database matches our theoretical calculations. For retrieval engine, we query it using different queries and validate on all intermediate calculation results (tf, idf, query vector, document vector, similarity scores, etc.) as well as the final displayed content in the frontend UI.

```
requestCrawl:Handling
Start Crawling root url: https://www.cse.ust.hk/~kwtleung/COMP4321/testpage.htm ...
Crawling Done
Start Indexing ForwardIndex Table, Url PageId Table, Keyword Tables...
Indexing ForwardIndex Table, Url PageId Table, Keyword Tables, Done
Start Indexing BodyInvertedIndex Table...
Indexing BodyInvertedIndex Table Done
Start Indexing TitleInvertedIndex Table...
Indexing TitleInvertedIndex Table Done
Updating Vector Space Model
Updating Vector Space Model Done
Start Writing Pages to spider_result.txt File...
Writing Pages to spider_result.txt Done
requestCrawl:Handled Successfully
```

Figure 1 Loggings during request handling for checking correctness of operation flow

```
requestSearch:Handling
query:test page
mode: Keyword Search
section:title
query vector:{12=1.0, 3=1.0}
page for 12={0=[1], 4=[2]}
page for 3={0=[0]}
MatchedPageIds:[0, 4]
final matched page IDs:[0, 4]
requestSearch:Handled Successfully
```

```
requestSearch:Handling
query:computer science
mode: Phrase Search
section:body
query word ids:[740, 734]
page for 740={161=[395], 5=[13, 27, 29, 50, 57, 83, 99, 134], 293=[127], 6=[3, 11, 17, 24, 29, 39, 40, 61, 79, 119, 221, 224, 236], 200=[239], 73=[129], 79=[309]}
after intersection:[161, 5, 293, 6, 200, 73, 79]
page for 734={226=[93], 258=[357], 228=[432], 5=[14, 16, 28, 58, 84, 100, 135], 133=[449], 6=[4, 18, 25, 30, 62, 120, 225], 200=[491], 296=[198], 94=[404, 407, 421], 239=[369]}
after intersection:[5, 6, 200]
MatchedPageIds:[5, 6, 200]
pageWordPos:{5={740=[50, 83, 99, 134, 57, 27, 13, 29], 734=[83, 99, 134, 57, 27, 13, 15]}, 6={740=[224, 3, 39, 40, 11, 236, 79, 17, 119, 24, 29, 61, 221], 734=[224, 17, 3,
 119, 24, 29, 61]}, 200={740=[239], 734=[498]}}
firstWordPos:[50, 83, 99, 134, 57, 27, 13, 29]
firstWordPos after:[83, 99, 134, 57, 27, 13]
firstWordPos:[224, 3, 39, 40, 11, 236, 79, 17, 119, 24, 29, 61, 221]
firstWordPos after:[224, 3, 17, 119, 24, 29, 61]
firstWordPos:[239]
firstWordPos after:[]
MatchedPageIds:[5, 6]
section:body
query vector:{740=1.0, 734=1.0}
page for 740={161=[395], 5=[13, 27, 29, 50, 57, 83, 99, 134], 293=[127], 6=[3, 11, 17, 24, 29, 39, 40, 61, 79, 119, 221, 224, 236], 200=[239], 73=[129], 79=[309]}
page for 734={226=[93], 258=[357], 228=[432], 5=[14, 16, 28, 58, 84, 100, 135], 133=[449], 6=[4, 18, 25, 30, 62, 120, 225], 200=[491], 296=[198], 94=[404, 407, 421], 239=[369]}
MatchedPageIds:[161, 226, 258, 228, 5, 293, 133, 6, 200, 296, 73, 79, 239, 94]
final exact matched page IDs:[5, 6]
final non-exact matched page IDs:[161, 226, 258, 228, 293, 133, 200, 296, 73, 79, 239, 94]
requestSearch:Handled Successfully
```

Figure 2 Loggings during request handling for checking correctness of intermediate and final results

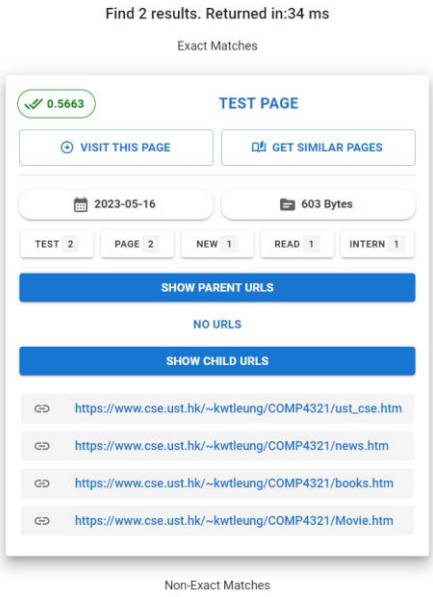


Figure 3 Validation on correctness of displayed page content

6. Installation Procedure

Please refer to the readme.md in the same zip folder for details.

7. Conclusion

7.1 Strengths and Weaknesses

The user interface of this project is highly intuitive and provides a similar user experience compared to commercial search engines. In addition, the crawler and indexer are robust and can dynamically update the index and VSM. With precomputed document vectors, the search performance is further optimized to allow fast retrieval. However, our database does not separate the index into different database files. This design has the risk of failure of entire database if only one of the index encounters failures. Moreover, the crawler takes a long time to crawl because it handles webpages one by one, which does not fully utilize the computing resources. At last, the retrieval engine only considers document-query similarity as ranking criteria and does not employ link-based ranking algorithm, thus the retrieved pages might have low quality and limited relevance to the user's intention.

7.2 Re-implementation Strategies

PageRank could be implemented for ranking the search results. An iterative method could be used to compute the PageRank until the values converges. Since the PageRank scores are query independent, they can be precomputed to avoid computation overhead during each search operation. To speed up the crawling and searching process, multithreading could be

introduced in the pipeline to fully utilizing the computing resources. This allows the program to execute tasks concurrently instead of a single threaded execution. For crawling, the multiple requests can be handled in parallel, and the order of crawling can be properly handled at the end of each thread. For indexing/querying, the pages could be divided into small subsets and assigned to different threads where they all execute a same but smaller version of the indexing/querying functions.

7.3 Interesting Features

For instance, scheduled crawling and indexing could be added to update the database automatically and periodically to avoid manual effort on updating database. Another interesting yet useful feature could be query suggestions which helps users to form queries more easily and accurately.

8. Contribution

CHEN, Jiawei (20763842) 100% (This is a group of one people)