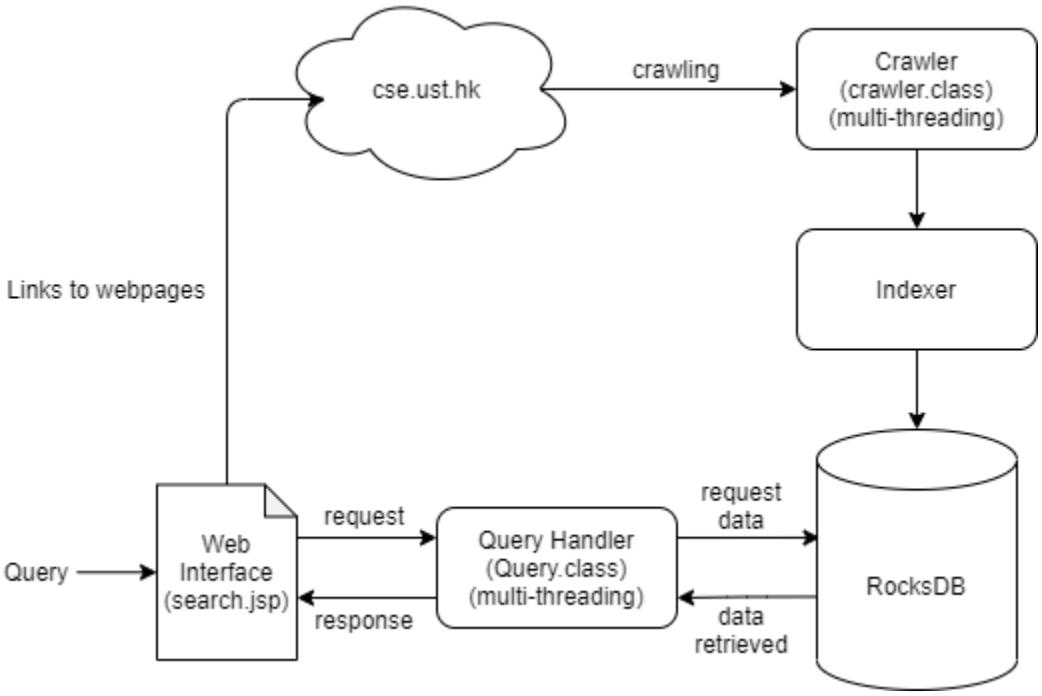
Search Engine for Web and Enterprise Data

Final Phase Documentation

Group 15
Chu Henn Khong (chkhong@connect.ust.hk)
Joszzef Maximillian Adiguna (jmadiguna@connect.ust.hk)
Kelvin Leonardo Hartono (klhartono@connect.ust.hk)

I. Design overview

The search engine that we have created takes inspiration from the multitudes of technologies and algorithms that have been given to us through the span of the course. The search engine itself consists of four different elements, the crawler, the indexer, the query handler and the web interface.



A. Crawling

Crawling is done to get all the links and pages that are in a particular domain, specifically the cse.ust.hk domain. The application itself utilizes multiple functions from the jsoup library, which allows us to finetune exactly how our crawler would work. Crawling takes a lot of time, but mostly the time spent is done when getting a response from a website, which means that the CPU is not fully utilized. Hence, we use threads to speed up the process of the spider when crawling the website. Several algorithms are used to ensure that the link is valid and to handle redirects. Progress is also printed out in order to keep track of websites that have been crawled and to see potential errors.

B. Indexer

The indexer is responsible for inputting the different document information into the specified mappings that we've designed. It utilizes the RocksDB database for high performance key-value insertions and extractions. While crawling, we access the database to input the indexes of the document. To speed up the query time, 9 database files are used at the cost of sacrificing time when doing crawling. But we felt this is a

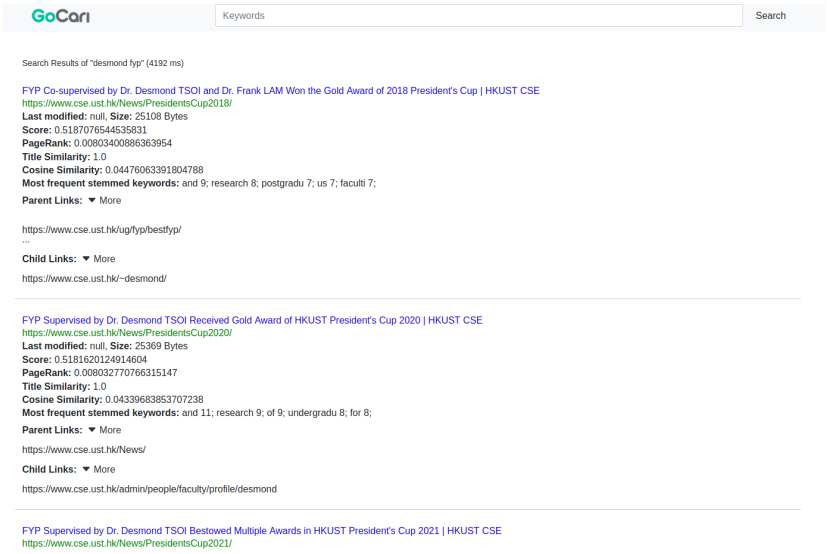
sufficient tradeoff as the focus of our search engine is the speed of the query. The databases are specified further under the file structure section of this document, but as a general rule, it is divided into three different categories, which are scoring, indexing, and mapping.

C. Query handler

The query handler calculates the scoring of each relevant element by computing cosine similarity scores between the title, body text, as well as a secondary pagerank scoring that is taken from the database. The query itself undergoes preprocessing not unlike that of the documents themselves. By taking advantage of all 9 precomputed databases at the stage of crawling, we manage to speed up the query time. Threading is also utilized as a means to fully utilize the CPU of the machine. After some testing on the virtual machine that we got from the CS Department, 2 workers per core is the optimal amount to fully operate the cpu. Furthermore, additional information is also given to the user to facilitate analyzing similarity, such as the scoring that the page has received along with parent links, title matches, etc. Although, at the beginning of accessing the website, the first and second query is significantly slower at around 3-4 seconds, where it decreases to around 2 seconds. We feel that this is the result of caching, in which the first two queries have to load all the needed variables without knowing any spatial or temporal locality.

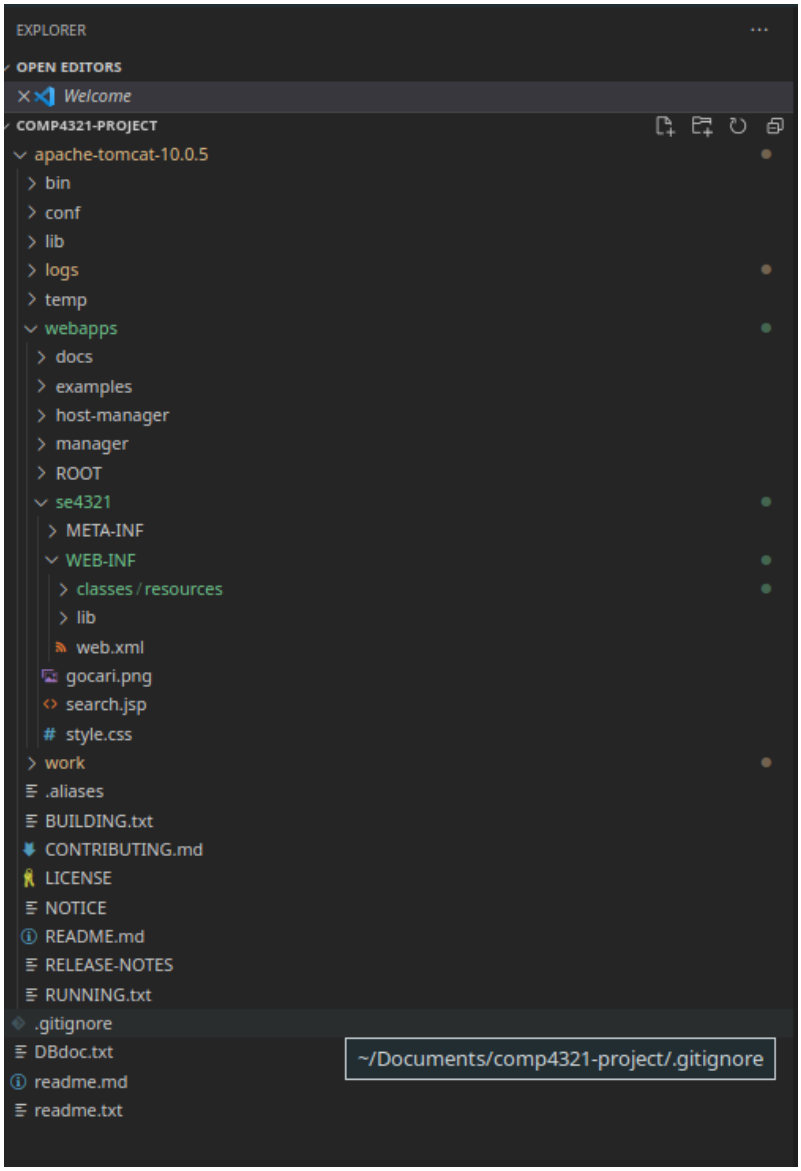
D. Web Interface

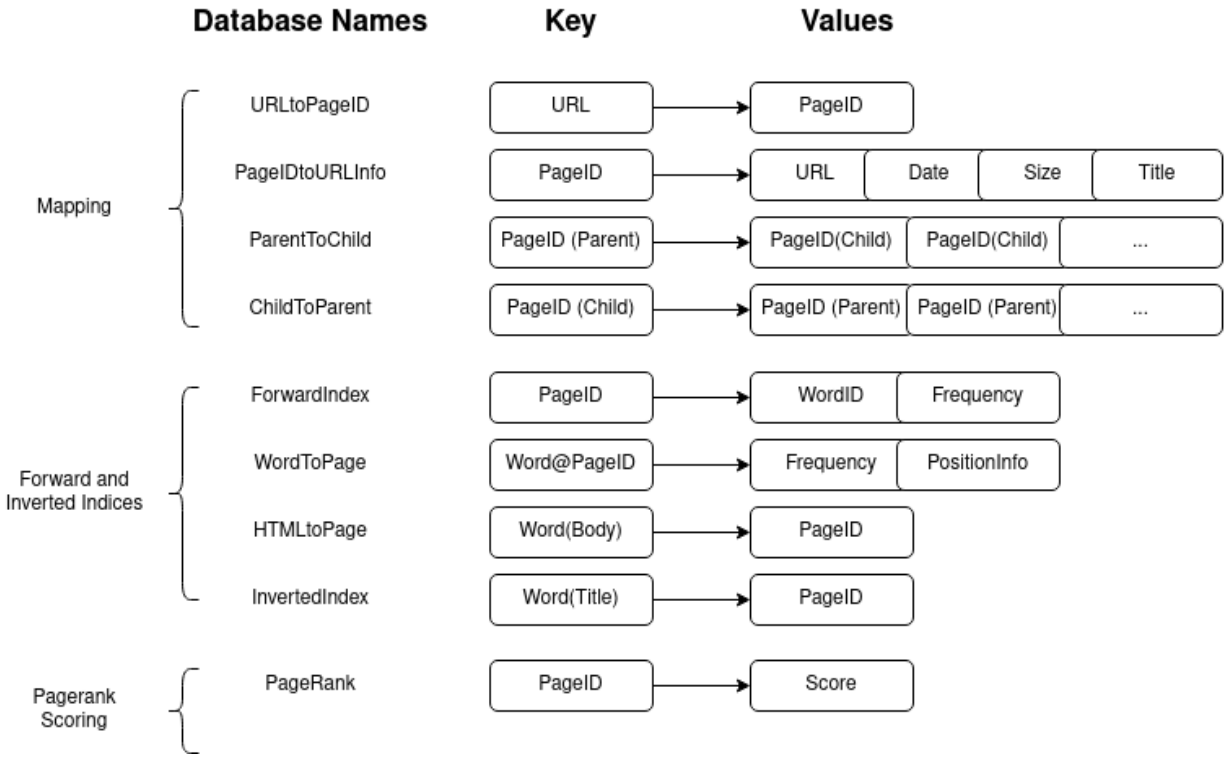
The Web Interface accepts user's query input and sends HTTP requests to the JSP engine in the web server. JSP engine converts and compiles the JSP to a Servlet class and Servlet engine executes it. During execution, the Servlet class produces output in HTML format which will be passed on to the web server as a HTTP response. Finally, the web browser generates the content on the Web Interface from the HTML output inside the HTTP response.



II. File structure

The project repository is set up in the webapps folder of apache-tomcat. This allows easy installation and setup of the project (further specified in the installation instructions section). This allows easier access of all necessary shell scripts and classes, along with allowing immediate deployment of the search engine.





The database consists of several key-value mappings that are stored on-disk. As per course specification, each mapping is stored using RocksDB. As such, each key and value pair are stored in a byte array format. The mappings are grouped into 3 different categories.

A. Mapping

This category contains four different mappings, each of which are important for the purposes of sorting pages and indices as well as scoring and building the index. The first two mappings are to facilitate URL to Page ID translations, as well as storing the metadata of each crawled webpage. The final two mappings are to store children links and parent links for usage in scoring algorithms such as PageRank.

B. Forward and Inverted Indices

This category contains four different mappings that store the forward and inverted indexing used. These mappings are integral for allowing forward lookup as well as backward lookup. Furthermore, position info and frequency are stored to accommodate tf and idf calculations. Title and body mappings are separated to assist in scoring purposes.

C. Pagerank Scoring

This category only has one mapping, which is that of pageID to its pagerank score. This mapping is computed separately after crawling has finished, and is used to augment the ranking system of the returned results.

III. Algorithms used Throughout the Program

A. Document Evaluation methods

To do retrieval of relevant terms matching the query information, we used a combination of scoring algorithms to determine document-query similarity.

1. **Vector space model**

Vector space model is used to store the frequencies of each term in a document and in a title(in a separate file), which is stored inside the database for further scoring. For both the body document and title terms, cosine similarity is used to compare the terms in a document and a query. The cosine similarity of the title has the biggest weight out of all our scoring algorithms, as we felt that most websites put the main topic of the document in the title.

2. **Phrase search**

For phrasal search, we utilize our "WordToPage" database to know the position/location of each word in a document. Terms in a phrase have a fixed distance between each of the terms, and we use this property to get all of the phrases inside a document. As we are doing stop word removal for the documents, for phrases, we also ignore any stop words in the query.

3. **Pagerank**

Pagerank scoring is done after crawling through the Pagerank class. We first build the hashmap containing children counts as well as Pagerank scores. The damping factor that we use is a value of 0.8. Afterwards, we keep track of the total delta between each pagerank score, and breaks if it does not reach the epsilon threshold of 10^{-8} or it reaches the maximum number of iterations of 100. Furthermore, calculation of PageRank scores are done asynchronously, which means that every subsequent calculation uses the updated scores of previous scores if applicable. After each iteration, L1 normalization is done on the pagerank scores. After the maximum iterations has been reached or the hashmap has reached convergence, it is then inserted into the final database of the search engine, along with two other elements that represent both the maximum value as well as the minimum.

B. Stopword removal and stemming

The implementation of stopword detection is done by the StopWord class. Within the StopWord class, it takes an address that points to the text file containing all stopwords, and converts them into a hashmap. Initially, we initialize the stopword list using the list given in the course website. By doing this, we can remove all stopwords by doing a simple lookup of the hashmap. But after several testing, we realized that it removes a lot of important keywords, such as "best" or "worst", which is often used to search for a query such as "best professor in HKUST". So, we changed the list of stopwords to Google's list of stop words that we found

from the internet. As for stemming, we use Porter's algorithm, with its open source implementation in Java.

IV. Features

A. Cosine similarity measures

Cosine similarity measures are used to weight the document similarity with the query given. Weightings are given to accurately differentiate between the value of title similarity with the value of body text similarity. By doing this, we can fine tune just how influential a 'special match' should be in determining the ranking of the pages.

B. Pagerank scoring

Pagerank scoring is done through the aforementioned algorithm mentioned above. Its contribution to the scoring system is through a combined weight that is distributed between cosine similarity scores, title matches, as well as the PageRank scores. The weighting given to PageRank is very small, as it is meant to only differ the pages that are very similar in content.

C. User interface

The user interface of GoCari is designed to be as intuitive and friendly for the average user. It displays the results in a neat and structured manner so that users can easily identify relevant pages. A set of attributes corresponding to the page, as well as parent links are given to facilitate deeper searches on the topic. The results are also returned with extra information regarding the whole collection, as well as the time it takes to collect the documents in said collection.

D. Special Implementation Techniques

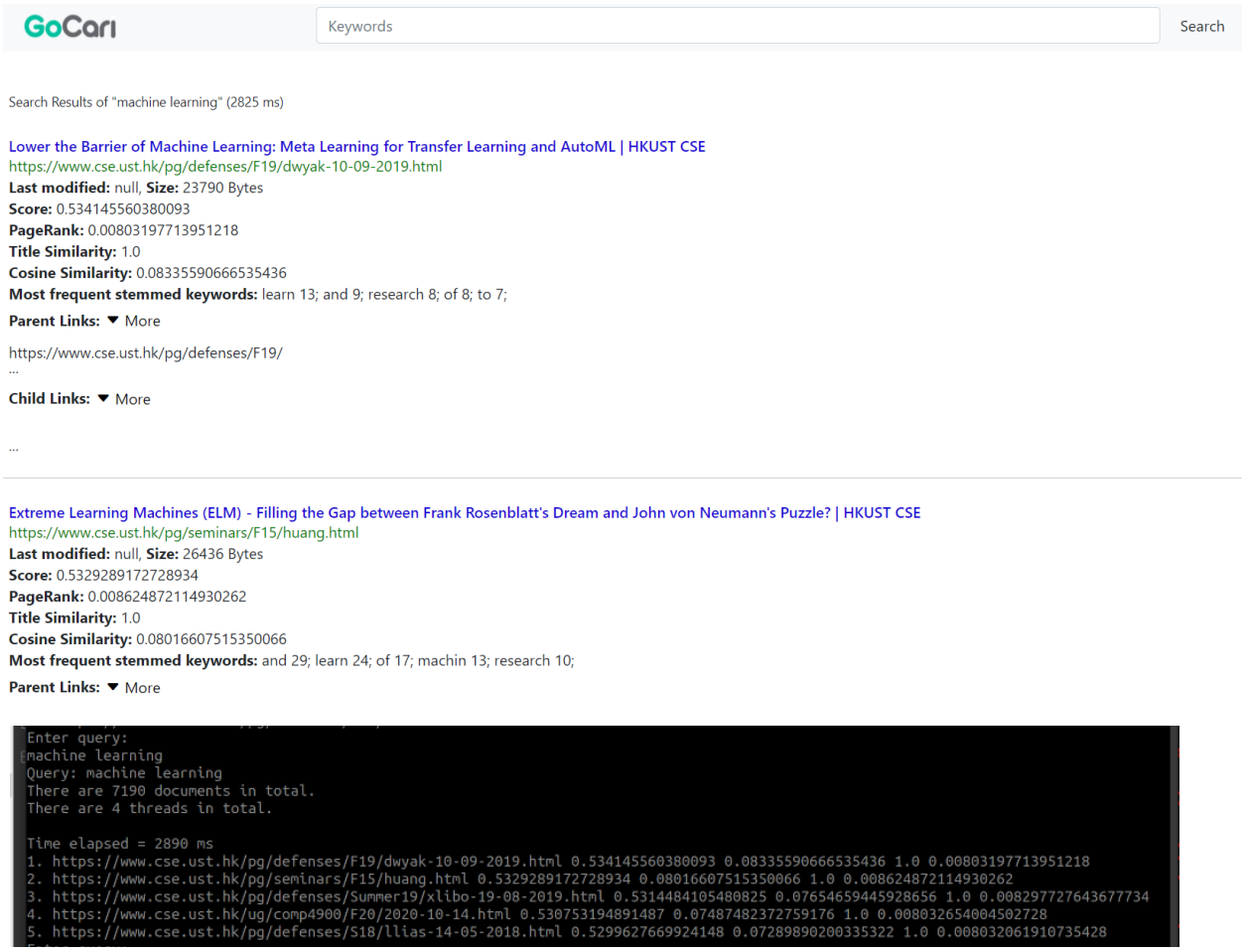
To aid in speeding up the whole crawling and querying process, we have utilized multithreading using the standard java library. This is to fully utilize the processing power of the virtual machine, which would then be allowed to execute concurrently instead of a single threaded execution. The idea is to divide the collection of documents to be divided amongst the different threads, where the different threads would then execute a smaller version of the indexing/querying functions.

V. Testing

Testing is primarily done through functional testing and unit testing. Whilst not included in the final repository, each class is thoroughly tested during creation, so as to assure that non-trivial functions are implemented correctly. The server pages are turned on, and manual testing is done through the query as a user might use it. The results are then taken

and compared with automated tests of the Query object, as to assure there is no data corruption during the transfer.

In addition to functional testing, we also implement the catching of exceptions, to ensure that the program does not exit ungracefully. These try and catch clauses are useful to narrow down the problems present in the code during debugging as well as during runtime, and allows for closer inspection of errors in the code. As shown in the figures below, we can then crosscheck the results of each query document.



Finally, to ensure correct distribution of weights, we do several trials with the same query of ‘Hong Kong’ to be able to tune the weights of the score appropriately. Afterwards, we settled on the weight for title similarity as being 0.5, 0.4 for matches in the body, and 0.1 for the pagerank scoring.

VI. Installation procedure and version control

Installation is made to be as easy as possible, with prerequisites being that the machine is linux based and has java installed on its PATH. The first step is to clone the repository

containing this project. After cloning the repository, a set of commands are given, and can be used in linux machines by running the command ‘source .aliases’. Afterwards, the commands compile_all as well as on_server sequentially will compile the java classes, as well as run the server on your localhost using Apache Tomcat. After running, simply navigate to the specified url and GoCari is ready to use. Several other commands are also provided to run certain functions separately. These functions are crawl and calculate_pagerank, which are used to crawl, index, and populate the databases as well as calculate the pagerank scoring. Another command for database control is purge_db, which clears the whole db folder to make space for crawl and calculate_pagerank. Finally, there is test_query, which creates a singleton query object to test out similarity checking without the need of the web interface.

We utilized Github primarily to do version control of the development of this search engine. Branches are utilized to fully isolate the different features of the development cycle, as well as provide backups of particular files that are important in the development cycle, for example Query classes. Furthermore, we use several libraries during the development of GoCari, namely the jsoup and rocksdb libraries, specifically version 1.13.1 of jsoup, as well as RocksDB version 6.15.5 along with the latest version of Apache Tomcat.

VII. Conclusion

As we were designing our algorithms and database, we found that there are some imperfections in our execution and also room for improvements, which are:

A. Strengths

1. GoCari’s user interface is highly intuitive, providing a similar form and function to most commercial search engines. What makes it different to commercial search engines, are that they provide further information regarding the parent and child links, giving the user a snapshot of the underlying network graph, which may be useful in determining further sources to research. Furthermore, the addition of publicly shown scores, should allow easier determination of page quality.
2. GoCari’s crawler and indexer is robust, and creates separated database files so that they are independent in the case of failure of a database. Utilizing multithreading, we can also significantly improve the time needed to crawl through web pages, as well as do preprocessing on the links themselves. The choice of data structures, as well as database design, also provides needed optimization, as they reduce the overhead time when accessing the database as well as doing calculations.

B. Weaknesses

1. We are not used to the legal type of links on the website, and as there are a lot of types, we were overwhelmed and needed to handle this one-by one. This may cause us to miss some important link issues. For example, until now, we are confused whether "http://www.example.com" and "https://www.example.com" should be treated differently. Furthermore, there are links with the same content, which we have not yet figured out how to handle.
2. Our database is stored as a string representation of an object that is converted to bytes. As such, when retrieving it, we need to parse the string, and do some operations for it to be an object, which leads to more time consumed. Instead, we could have just serialized an object into bytes, which would have saved a lot more time.
3. The crawler takes so much time for the reason of handling redirections, where we need to do a get request for every child links in the server to ensure the link is correct, which takes a lot of time.

C. Interesting features

1. Scheduled crawler should be added to ensure that our database is updated
2. Further coverage using multiple testing techniques, such as unit testing as well as black-box testing could have assisted in increasing coverage and speeding up the development of our search engine.
3. On top of stemming, or to replace it, we could have added lemmatization. This is to handle other words that are related which is not in the document.