# Data Processing Workshop II

## Group Project

Group Number: 25

Group Members: 3

Yuhan ZHENG 1930026174

Zhuoheng MA 1930026096

Jiayun XU 1930026140

# Contents

# Introduction

The data processing project aims at retrieving text contents related to the query and ordering them according to relevance. It also carried out a webpage, a 20-therad Spider, an RDD and an elastic search. This report will first provide the whole system design, then explain how to implement it, and compare the result of tasks through using different methods.

In the spirit of excellence, the problem should be analyzed at a higher level. To design a good programming system in a professional view, the following 10 things should be well considered:

1. Achieving the functions
2. Safety
3. Robustness
4. Compatibility and the ability to continuous update
5. Modularity and cohesion (Reducing coupling)
6. Increasing reusability
7. Testability

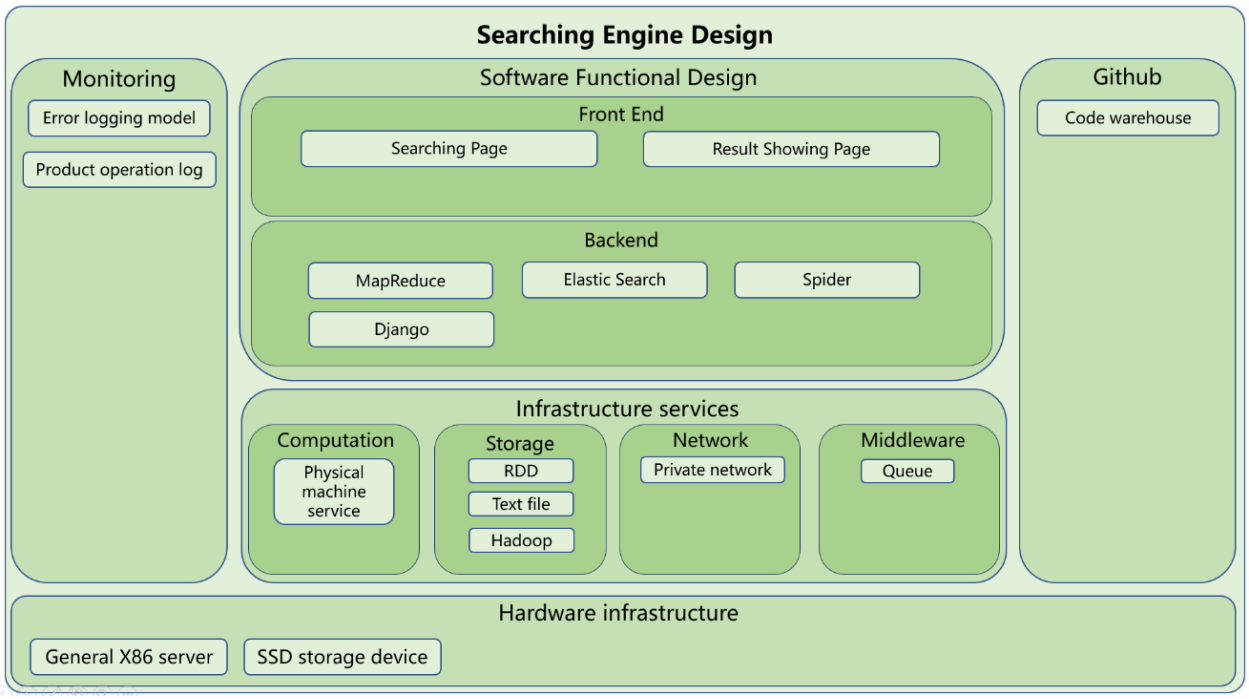


Figure 1 The system architecture

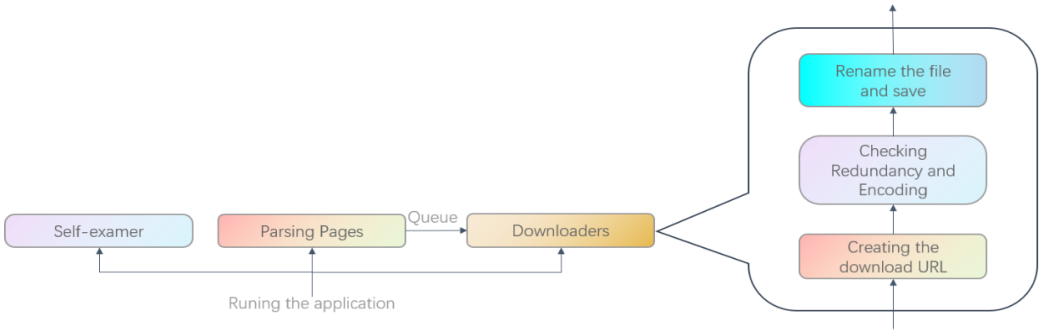


Figure 2 Crawler's architecture

Following the idea, the system is designed as the architecture showing in figure 1. Firstly, it is separated by functions, considering both hardware and software factors, to ensure achieving all the functions, reducing the coupling, and increasing the reusability.

By the way, when implementing the system design, different functions are packed into different python classes or functions, which is exactly increasing the reusability.

## Text collection

The text files are using the spider to collect. And the spider is designed in the architecture in figure 2.

At the very beginning, it starts with different threads for self-examers, parsers, and downloaders. There will be 2 parsers and 20 downloaders – since a page has around 20 books, and some pages have some books it has already crawled before – this architecture can almost give a best performance. The downloaders will always check the queue, and if the queue is not empty, it will begin to download books.

Using this architecture, the spider can reach a dramatic speed, up to 10 Million Bytes per second. And because it used wget in the downloaders, its robustness got a guarantee. And what's more, it can automatically rename the files and check the encoding.

Besides, although it is powerful, it's easy to use. Changing the number of books to download takes only 10 seconds, editing

some parameter in the main function (shown as below).

Running the script, waiting for 3~5 minutes, then all the books are downloads.

## Indexing

The methodology of implementing indexing and ranking is TF-IDF, also called term frequency-inverse document frequency. Initially, in order to derive term frequency, for each document, get each word in this document by splitting the raw text file by space and total word number of this document. Search input may contain punctuation or special characters, thus, they are also kept in term frequency. The term frequency of a specific word is the result of the number of times the word appears in a text divided by the total word number of this text. Secondly, the IDF of a certain word is calculated by dividing the number of files that a certain word appears from the total file number. All the raw text files are stored in HDFS.

For a single computer, in order to get tf-idf scores of each word effectively, pyspark and pyspark SQL are applied in the code. MapReduce is also used to accelerate the speed of computation. Before it is converted to data frame structure, data is stored in RDD structure. Finally, a file named *tfidf-index* saved all the tf-idf scores of all the words that appear in corpus. For the small file set, the code is run under Jupyter notebook. However, it is too large for the notebook to run a large

| Environment | Small file set (100MB) | Large file set(1.2GB) |
|---|---|---|
| Single Computer | 1.1 minutes | 14 minutes |

Table 1

text file, thus, it is running under terminal by specifying the driver memory and executor memory to 4G and 8G respectively. For a single computer, the time it takes to process indexing jobs for a small file set and a large file set is given in the table (Table 1).

For cluster, the code is almost the same but specify the master at the beginning of the code.

Three nodes with 4 cores are connected to a master. The times it takes to process indexing jobs for small and large file sets is given in the table (Table 2).

| Environment | Small file set (100MB) | Large file set (1.2GB) |
|---|---|---|
| Cluster | 1.3 minutes | 16 minutes |

Table 2

In order to accelerate the indexing speed, Elasticsearch is applied. When loading the raw file in Elasticsearch, term index is inserted into the data to speed up the search process. In the *create_index* function, the hyperparameters *analyzer* and *search_analyzer* are the indicator that the way the input raw text and input is treated. In this group project, they are defined as *whitespace*, since the user may input some keywords with punctuation. It creates indexes of all the word when loading the file as well, which takes about two minutes for a 1GB file. It is one seventh time than a singular computer processing speed. While for the small file, it just takes 25 seconds.

## Ranking

It first loads the parquet file that stored tf-idf scores and then searches for the match keywords one by one. By calling the *top* function, ranking is performed when backstage returns the top N result, where N is the parameter that with the highest tf-idf scores. Normally, it takes 25 seconds to return the result no matter whether the keyword is a single word or a sentence.

With Elasticsearch, there are several hyperparameters that need to be defined when searching. In *Search_data* function, a hyperparameter *size* can be defined to indicate how many results Elasticsearch returns. It takes about 0.1 seconds to return the search result in the search engine, which achieves a state-of-art performance.

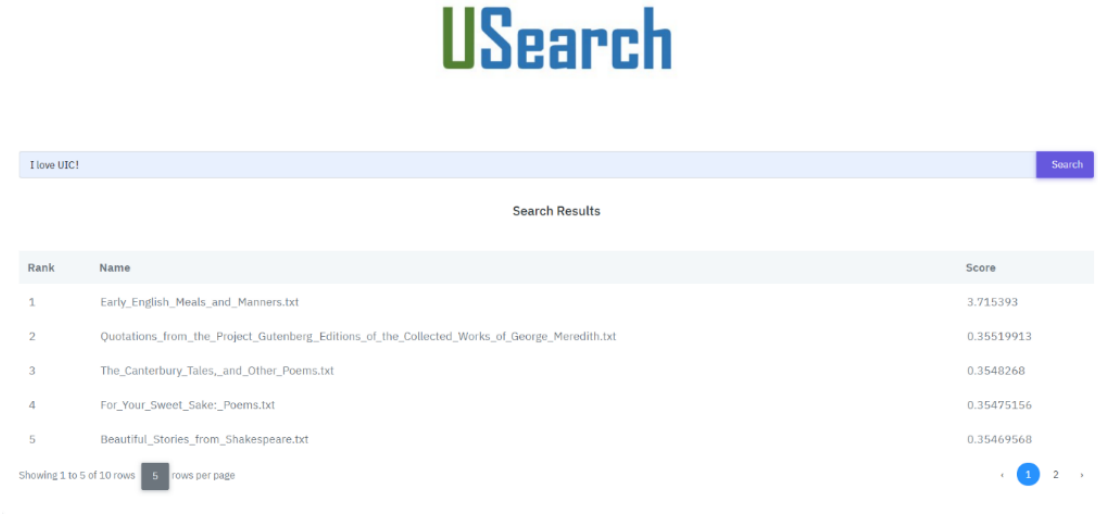


Figure 3 The webpage interface

## Webserver & Interface

Two web pages are created in the project, showing the results returned with and without third-party library searches. On each webpage, HTML, CSS, and JS tools are used to write the front-end interface, with bootstrap as the development frame. Then, functions inside Django accept and process requests. After that, the webpage is posted to the server and returns the result of the TF-IDF search.

The web page is called USearch and with a self-designed label displayed at the top of the page (Figure 3). There are mainly two elements in the body. One is the search box, where for searching text by entering the keyword, and the other one is a table, where the result displays.

The table shows the rank, name, score of the results, and follows the order of score from high to low. Besides, it can be customized how many search results are displayed on the page and turned the page to see the subsequent results (Figure 4).

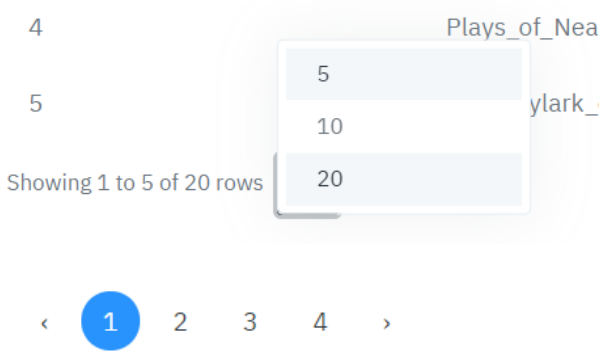

Figure 4 The searching result

## Conclusion

This project can achieve crawling documents, sorting by relevance and web search functions, has a certain practicality. However, in real search engines, each keyword will be given different weight according to the order of search keywords, which can use Page Rank to improve the performance in the future work. Besides, the system doesn't use database as the middleware, which limits its real-time capability. The above problems mentioned will be studied in the future work.

## Future work

To get the information in real-time, the system should better be designed as a streaming work system – considering the needs of high-speed reading and searching and the application scenarios of writing once and reading many times, the system can use Redis as the database and check the data updated on the website, if the website's data is updated, then running the spider to crawl the new data and update the index. And since the amount of data is really huge, each time updating the index will take a long time if it updates will the traditional way – it is better to use Page Rank to do the updating.