CIS555 Final Project - Search Monster

May 2022

- Zhonghao Lian; zhlian; Email: `zhlian@seas.upenn.edu`
- Ning Wan; ningwan; Email: `ningwan@seas.upenn.edu`
- Xincheng Zhu; kyriezxc; Email: `kyriezxc@seas.upenn.edu`
- Haoran Liu; haoranl; Email: `haoran@seas.upenn.edu`

Repository: https://bitbucket.org/ningningwa/search-engine/src/master/

Abstract

Our project aims to create a robust distributed search system based on the content we learned this semester. We use AWS technologies, including RDS for data storage, EMR to run spark jobs, and EC2 for hosting. Finally, we achieved a search engine with high efficiency and accuracy.

1 Introduction

In this work, we present the design and implementation of our Web Search Engine. The search engine consists of five basic components: Distributed Crawler (Xinchen Zhu), Indexer (Haoran Liu, Ning Wan), PageRank Engine (Ning Wan), Search Engine (Zhonghao Lian), and Web Interface (Zhonghao Lian). Important design decisions, implementation challenges, and detailed performance evaluations are included in this report.

2 Project Architecture

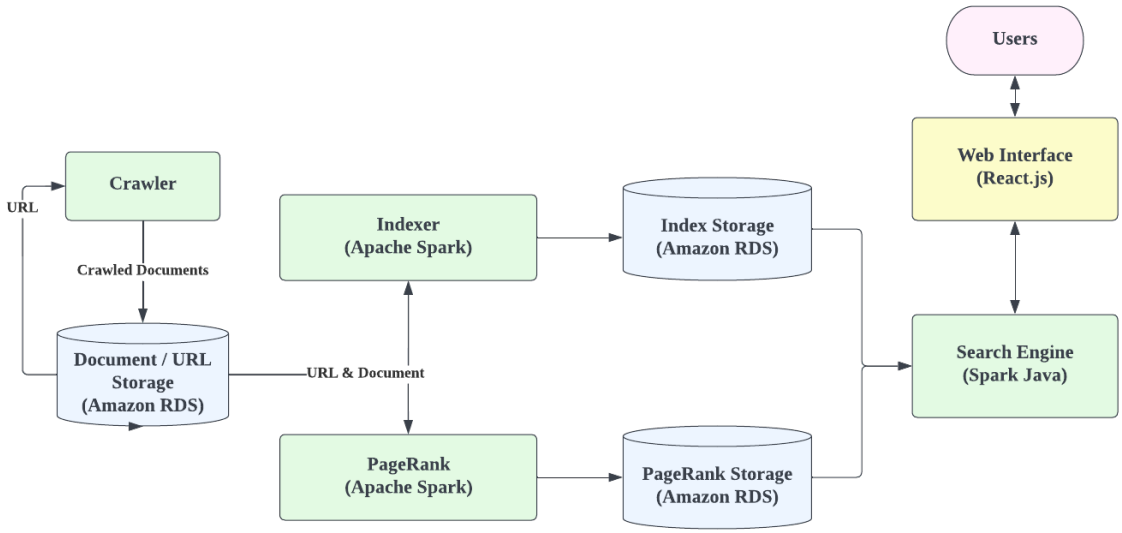

Figure 1: System Architecture

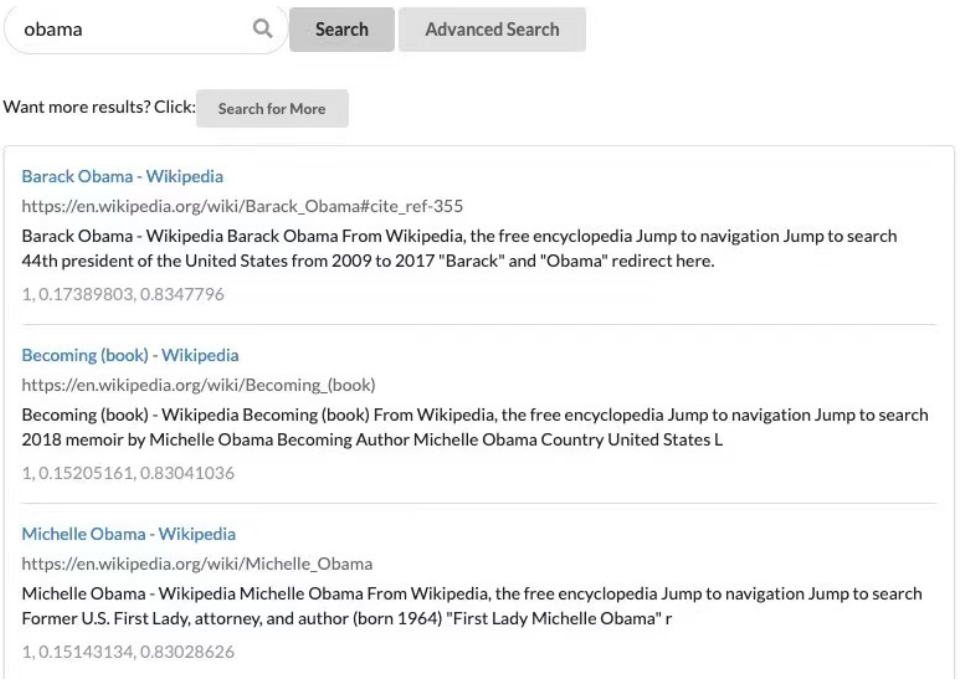

Figure 2: Sample search result page.

The workflow starts at the Distributed Crawler component. The crawler retrieves documents from the Internet and stores the documents in Amazon RDS. While the document contents are recorded, the crawler also records URL relationships: URL a has a link to URL b. The URL relationships construct a network graph, which makes it easier for the PageRank Engine to run the computation.

The Indexer component, which is implemented with the Apache Spark framework, builds inverted index and computes the IR score with the retrieved documents. For each document, the Indexer first parses it, and then stores each $< word, URL >$ pair as an item in the inverted index. When all terms in a document have been indexed, the corresponding word frequency is computed for each item, which can later be processed into tf/idf score. The inverted index is also stored in Amazon RDS.

Meanwhile, the PageRank Engine leverages the network graph that the Crawler built, to compute the PageRank score for each URL. When the Indexer and PageRank components finish their work, both IR and the PageRank score are ready to be used by the Search Engine.

The Search Engine component is implemented as a web server, listening to a specific port. The Search Engine uses the Spark Java framework. It defined several routes to be called by the front-end. When the Search Engine received a query, it pre-processes the query to eliminate stopwords and punctuation. Then, relevant database helper functions are called to fetch inverted index and PageRank from RDS. The ranking for each indexed URL is computed based on a combination of IR and the PageRank score. When the ranking is done, the search results are marshaled into a JSON string and are sent to the front-end via HTTP response. The front-end then parses the results and shows them to the users.

3 Components

3.1 Crawler

3.1.1 Workflow

The crawler contains two main distributed parts: crawler and webcrawler. Crawler is used to crawl from base urls user input in the main function defined in the crawlertopology class, parse the HTML to get sub-urls (follow the instructions in robots.txt file of the website by checking the "allow"/"disallow") and store them in the Redis local database. Upon the data stored in the Redis reaching some thresholds

(depending on the RAM of the PC) or the crawler finished the work, the webcrawler will connect to the AWS RDS database to build the relations and write insert queries to store the urls, htmls and the relations between them (shown as parent and child id) into RDS table.

3.1.2 Crawler and Redis

Redis is an open source (BSD licensed), in-memory structure store. We decided to implement it in our crawler in order to increase the running speed of the crawler part. In the first edition of the crawler code, the crawler and the webcrawler work at the same time (crawl the urls and htmls, parse and store into RDS database at the same time). We found out the crawler died frequently. The reason for this is in several parts: the ram of the PC has been used up; the webcrawler lose the connection to RDS database which result in crawler stop working. We then decided to make crawler and webcrawler work separately by running crawler first and storing the html files in the HDD and urls/id in ram. When webcrawler tried to store the htmls into RDS database, it will loop through the local dictionaries and delete the files it stored. After two days of work, we realized this method is quite inefficient. We only store around 10,000 records in the RDS database. Finally, we decided to use Redis to store all the files we need (urls, generated ids, htmls, relations).

3.1.3 Details in Crawler

In this part, we want to show what base urls we choose for the projects and processing details. We crawl from Penn website, all news platform home page, wikepedia daily news and some hot words (like covid). When getting the urls from the web, we will generate an id automatically and store the relations of the urls as a set of parent id. We define parent id as the id of the "root" url is the parent id of the urls we parse from it.

3.1.4 Webcrawler

As mentioned in the above part, we separately the crawler part out in order to guarantee the crawler work successfully and continuously. In webcrawler part, we use Java Persistence API (JPA) framework to store relations into RDS database and spring framework to simplify the code. We define two tables in the RDS database. The structure of them shows as table 1 2:

Key	Type
id	String(32)
html	mediumtext
updateDate	String(64)
accessUrl	String(2048)
accessTimes	Integer(32)

Table 1: t document structure.

Key	Type
id	String(32)
cid	String(32)

Table 2: t doc rel structure.

3.2 Indexer

3.2.1 Workflow

To index the documents, we crawled from the website, we tried a few different strategies in Java. We used two frameworks in this process, one is multithread, and the other is Spark MapReduce. We use the multithread method firstly because what we crawled down is not files but URLs and HTML. In parsing, we first use Jsoup to extract the content from HTML and then parse the content and we wanted to use multithread to test the speed of parsing web pages and store the results in local Berkeley DB. After testing, if we only parse the title, we found that multithread can process about 1500 records

in RDS in one minute. However, if we want to parse the content in the entire HTML, multithread can only process dozens of URLs a minute. And the time of writing to the RDS could be even longer than the process of the parse. The specific amount of processing is related to the content contained in this URL. But this speed is far from meeting our requirements. Therefore, we expand and use Spark's API for reading RDS to use MapReduce and run it on EMR.

3.2.2 Implementation

The running speed has been dramatically improved after running the indexer with spark MapReduce. Therefore, we choose to process each URL in two parts. The first part is to extract and parse only the title and excerpt in HTML. Parse the header is very helpful for us to locate the title information more accurately. The second part is extracting and processing the content in only one HTML. Parse the body can help us determine the relevance between the scope of this article and the problem in the subsequent query.

We first divide the content into words according to the spaces in the parsing process. After that, we deleted the invalid part, such as punctuation in the word. Finally, we use the englishStem API to stem words. In addition, we set up a stop word list, which does not count the high-frequency words that have no practical significance, such as the, a, an, and so on. In addition, We define the first 25 words in each HTML as the exact of the HTML.

For title, the weight is calculated by:

$$weight^{(i,j)}(x) = 0.5 + 0.5 * \frac{freq(i,j)}{num\_of\_unique\_words} \tag{1}$$

For content, the weight is calculated by:

$$weight^{(i,j)}(x) = 0.5 + 0.5 * \frac{freq(i,j)}{max\_freq(j)} \tag{2}$$

We use different formulas to calculate the tf of each word in title and content. Where num_of_unique_words represents the total number of unique words in this title. Max_freq represents the frequency of the term with the highest frequency in the URL. The reason why we adopt different formulas is that the title is generally short. If Max_freq is applied, most tf will be 1 or 0. This situation is unfavorable for us to locate the weight of words.

3.2.3 Meta Data Used for Improved Ranking (Extra Credit)

As we mentioned, we have tested the indexer with two content versions. The first is with title and excerpt, the second is with the whole body. The version with title and excerpt shows better performance, which has better ranking performance, such as better ranking speed and more reasonable output, for the front end. We also include our excerpt in the database so that the front end could show some short descriptions to users for short reviews.

3.2.4 Result Format

In the process of parsing and writing the converted index results into RDS, we also include the location of each word for the first time, the URL corresponding to the word, the weight of the word, and the exact of the URL. The format of the data sheet is as follows. For the results of title and body, we use the same data structure.

excerpt	id	loc	term	title	url	weight
TEXT	INT	INT	TEXT	TEXT	TEXT	DOUBLE

Table 3: Indexer Result Table

In the process of parsing and writing the converted index results into RDS, we also include the location of each word for the first time, the URL corresponding to the word, the weight of the word, and the exact of the URL. The format of the data sheet is as follows. For the results of title and body, we use the same data structure.

3.3 PageRank

3.3.1 Workflow

The PageRank is developed with Java, Apache Spark and deployed in AWS EMR. We use Spark here to simplify our work in constructing the structure of MapReduce and experience the built-in feature of the Spark framework. Besides, Spark stores all the info in RAM, so the data would have better computed time compared to Hadoop MapReduce method. This component is to generate ranking scores based on the relationships between different urls. We grab the relationship data from the remote RDS Mysql database, and then run spark on after deduplicating and groupby each node. The final result is stored in AWS RDS.

3.3.2 Algorithm Implementation

The algorithm follows the basic logic of PageRank. In addition to that, we have successfully handled some issues.

For the dangling links, we make the surfer choose a random outgoing link at probability α=1.5. Hence, we can solve the relationship matrix with equation

$$PageRank^i = 0.15 * M * PageRank^{i-1} + 0.85 \tag{3}$$

or in non-matrix form

$$PageRank^{(i)}(x) = 0.15 \sum_{j \in B(x)} \frac{1}{N_j} PageRank^{(i-1)}(j) + 0.85 \tag{4}$$

Here we use a non-metric form in order to put it as a Spark MapReduce format. In order to check convergence, we set the threshold to be 0.00001. That means, after each iteration, if all PageRank scores calculated have the diff value within that threshold, we can conclude our calculation has reached the convergence and then write the result to the remote database.

3.3.3 Result Format

The PageRank Algorithm is calculated with the above steps. Then, the result is written in the database in the following format.

Key	Type
url	String(32)
score	Integer(32)

Table 4: PageRank Result Table

We keep the raw data here so that the front end could make use of the potential of the PageRank score as it needs.

3.4 Search Engine

3.4.1 Workflow

The Search Engine components are implemented in the Spark Java framework. The Search Engine is basically a web server running on a specific port, say 8001. When the Web Interface components (details in the next section) send HTTP request to one of its routes, the Search Engine will react accordingly, query data from the database, and sort it, before finally returning the data in JSON format to the Web Interface.

Since the indexing and PageRank results are stored in AWS RDS, we implemented a Database Helper, which serves as a middle layer to retrieve data (using MySQL queries) from RDS. When a route receives a query, it first pre-processes the query by eliminating punctuation and stop words. In order to have query terms in line with the indexing results, the routes will stem each word with englishStemmer, before trying to retrieve indexing items from the Database.

Our implementation of Search Engine adopts a tiered search strategy: we have two buckets of index: one from URLs that have large PageRank scores (about 100,000), another from all URLs (about 500,000). When a user submits a search, we first retrieve the inverted index from the smaller bucket. If the number of returned results is too small, we will provide users with a button to search for more results, where users can submit searches into the large bucket. In this way, we strike a balance between search efficiency and results quantity, and also provide users with more flexibility.

3.4.2 Ranking

The Database Helper will return a list of indexing items to the route. By now, we have the IR score (i.e. tf/idf) for each indexing item (i.e. each term x url combination). We then retrieve the PageRank score for each indexing item. In order to achieve maximum efficiency, the whole PageRank table is pre-loaded into the memory of the Search Engine as a HashMap. Thus, for each indexing item, we only have to retrieve the corresponding PageRank score from the in-memory table, using url as key. We compared the search efficiency by querying an individual PageRank score from RDS and found that by storing the PageRank table in memory, we can increase search efficiency by over 10X.

Approach	Average Search Time
Store PageRank in-memory	2
Retrieve each PageRank from RDS	25

Table 5: Search efficiency increase by 10X by storing PageRank in-memory.

After we have both IR and PageRank score retrieved, we can compute the document ranking. We employ Document Vector Model for ranking. Specifically, we use the Cosine Similarity between document vector and query vector as the core metric for ranking. The metric is computed as follows:

$$sim(d_j, q) = \frac{\sum_{i=1}^{t} w_{i,j} \cdot w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} \cdot \sqrt{\sum_{i=1}^{t} w_{i,q}^2}} \tag{5}$$

Where $w_{i,j}$ represents weight for term i in document j. Each element of document weight is a combination of IR and PageRank score, which is computed as follows:

$$w_{i,j} = \alpha \cdot IR(i,j) + (1-\alpha) \cdot PageRank(j) \tag{6}$$

Where α is a weighting factor to balance the results of IR and PageRank. Meanwhile, $w_{i,q}$ represents weight for term i in query q. Each element of query weight is computed by tf/idf:

$$w_{i,q} = TF(i,q) \cdot IDF(i) \tag{7}$$

We have done an extensive search for factor α, before finally setting α to 0.8.

3.4.3 Product Search (Extra Credit)

We implemented Amazon product search as an EC feature. In order to get structured search results from Amazon, we used Rainforest API, which is a real-time product API that returns Amazon search results. When the user clicks the 'Advanced Search' button (see details in next section), a route that's responsible for product search will receive the corresponding query. Afterward, the route will fetch product items via Rainforest API and organize the results. The users are then able to navigate both regular search results (from retrieved documents from RDS) as well as Amazon Product results, which we interpret as 'the products you may like.

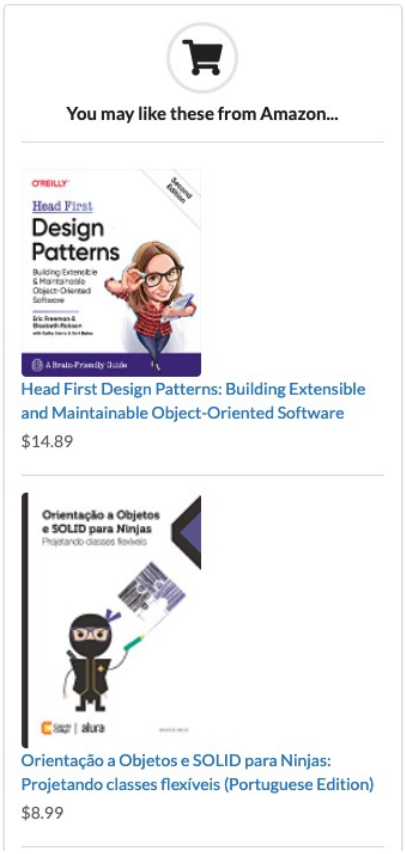

Figure 3: Sample Amazon product search.

3.5 Web Interface

The Web Interface aims at providing the users with a clear and concise interface where they can search for keywords or phrases. In our context, we also want the Web Interface to support debug functionalities. Thus, we create two modes for the Web Interface: normal mode and debug node. In the normal mode, search results are presented similar to the Google pattern. While in the debug mode, tf/idf, PageRank, and the weighted score are presented alongside each search result. That way, Web Interface is used as an extension for our IDE and terminal, with which we can tune the weighting parameters easily.

The Web Interface is implemented in the React.js framework. Since our Search Engine is implemented with Spark Java and will be communicating with the front-end via RESTful API, we can basically choose any front-end framework. We decided to use React.js among several front-end frameworks because its modular structure creates flexibility, which makes it easier to present several individual search components. Apart from React.js, we also use Semantic UI as the theming framework to provide the users with a more aesthetic search interface.

The search interface consists of three main components: search bar, search results, and relevant products. When a user types in a query and presses the search button, a fetch request with the corresponding query is sent to the Search Engine. The request will block and wait until the Search Engine finishes and sends back the results. The search results component will parse received JSON and populate each individual search result item. A similar procedure is done with the relevant products component.

For the ease of testing and presenting, we provide the users with two search buttons: search and advanced search. For search, only results retrieved from our crawled documents will be shown. For advanced search, both retrieved documents and the items from the Amazon product search will be shown.

4 Evaluation

4.1 Crawler

For the crawler part, we already improved the performance by implementing Redis local database as talked about above. With Redis, the crawler can store 5000 urls/htmls and 5000 relations within 5 minutes. On average, the speed of crawling can reach 2000 records/sec. If the base url used is relative simple, for example, wikipedia page, the speed can reach up to around 3500 records/sec.

For webcrawler part, after using Redis, the performance is relatively good. Some web contains large htmls, like some news platform page. It will take around 7 hours to store 6000 to 7000 urls/htmls into doc table, including 7000*500 (assuming 500 as the relations number) into rel table. In short, the speed is 51430 records/hours (actually higher since each url has more connections than the assumption). The performance in some simple url will increase by around 30 percent.

Finally, there are around 500,000 urls crawled into AWS RDS.

4.2 Indexer

In our testing phase, we used multithread for index. However, due to the limitations of multithread, the processing speed is not very fast. The average processing speed of the title is about 1500 per minute, and the processing speed of the body is about 30 per minute.

After realizing the short performance, we have improved our algorithm inspired by the Spark MapReduce implemented in our PageRank. For here, the process is more complicated than PageRank because storing and parsing each url in EMR requires not only more advanced hardware support like enough EC2 RAM but also adequate input parameters such as the Spark partition value and input url batch size. We set the hardware to be m5-xlarge (1 master and 2 scores) to make our budget as expected as well as increase memory space. Then, we fine-tuned the partition value to be 200, 400, 800, with 100,000 input urls but all failed due to lack of memory. After successfully tuning those attributes, we decided each data input to be 50,000 and npartions to be 200. This can help us extract more data from RDS at one time through the above operations, and the writing speed to RDS will also be improved.

The result is as expected. It only takes fewer than three hours to parse and write the body of every 50000 URL data, while it only takes eight minutes to parse and write the title and excerpt of every 50000 URL data. The "title and excerpt" version is what we finally adopted. And it takes fewer than two hours to finish the indexer algorithm for all 500,000 urls.

4.3 PageRank

We have run over 100,000 urls (around 6 million relationship tuples) to generate corresponded PageRank scores on m5-xlarge EMR (1 master and 2 cores). Eventually, the running converges and ends with a 90-round PageRank calculation. It only takes 29 minutes to finish the algorithm running. The performance is good enough that no further performance testing is scheduled for PageRank. More money could be saved to tune indexer parameters.

4.4 Search Engine

We evaluated the search efficiency of our Search Engine components. The evaluation was done by submitting multiple queries to the Search Engine and recording the time needed for each search. The results are presented in table 2-4.

Term	Number of Results	Search Time (s)
Pennsylvania	4494	1.74
Book	4334	1.69
Computer	948	1.53
UPenn	1210	1.56
Microsoft	1602	1.90

Table 6: Search time for single-word queries.

Term	Number of Results	Search Time (s)
Machine Learning	2120	4.19
Distributed System	1632	4.31
App Store	6918	4.28
Public Policy	5503	4.30
Fast Food	1252	4.54

Table 7: Search time for double-word queries.

Term	Number of Results	Search Time (s)
Deep Reinforcement Learning	2039	5.53
Special Interest Group	2966	5.63

Table 8: Search time for triple-word queries.

We can observe from the tables that for a single-term query, the average search time is around 1.6; for a double-term query, the average search time is around 4.3; for a triple-term query, the average search time is around 5.5.

In our implementation, several approaches are adopted in order to boost the efficiency:

- Load the PageRank table fully in-memory to enable efficient PageRank score search.
- Partition inverted index tables (in our case, 10 partitions) to enable multi-threaded data retrieval.
- Tiered search strategy to impose limitations on corpus and result size (details in 3.4.1).
- Use multi-threaded execution for each query word to enable parallel search and processing.

5 Acknowledgment

We would like to thank Professor Zachary Ives and all TAs for their guidance during this semester.

6 Conclusion

In the process of this project, we have better applied the knowledge learned this semester. We divided a search engine into five parts, and each team member worked together. We not only had a deep understanding of the part we were responsible for but also had a comprehensive understanding of the overall architecture of a search engine. Although we have faced many problems in designing the project, for example, our initial framework assumption is not efficient in our actual operation. However, we finally overcame these problems through unremitting efforts and had a deeper understanding of the project.

Overall, our search engine performed well. Although as the length of search statements increases, the time to return results will become longer. But relatively speaking, it has been an excellent search speed. In addition, we have also tried to search for keywords on various topics, including sports, art, humanities, etc. our search engines can return results related to the amount of more than 200 billion silly girls. This result shows that our data storage capacity is large enough to deal with various topics of queries.

In this process, we also practiced how to use AWS technologies. And have a deeper understanding of distributed computing. Moreover, if we have the opportunity, we want to continue to improve our project. First of all, because the amount of data in the inverted index results is too large, which slows down the speed of search engine results, we plan to shard the inverted index table. Divide into multiple tables according to the initial classification to improve the rate of searching each word from the database. This approach is a help to improve the running speed of our search engine.