Project Proposal for Group 0xFF

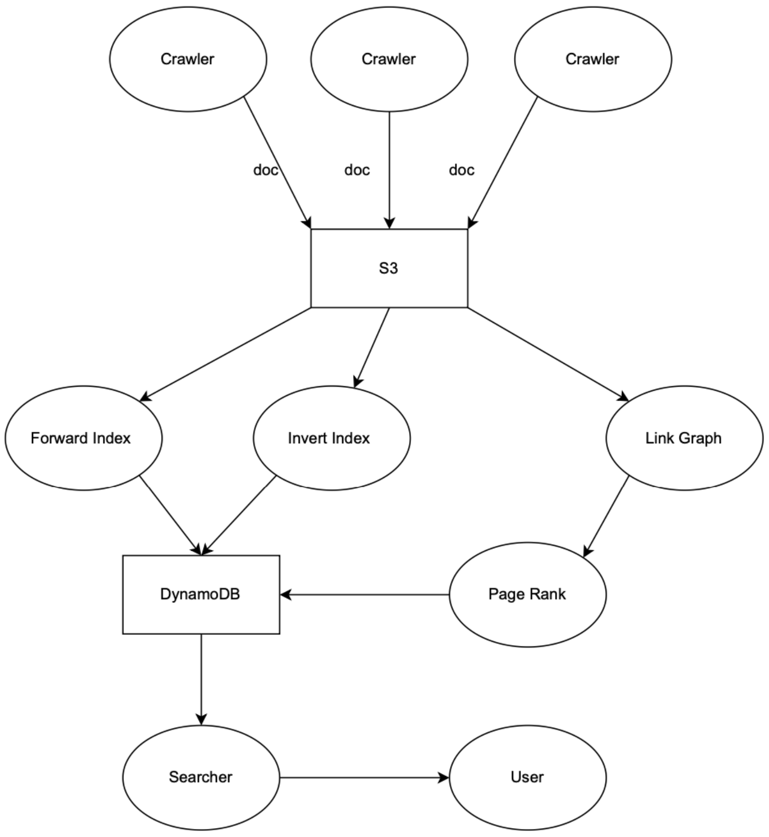
Github Repo: https://github.com/yx1215/CIS555-SearchEngine

Group members:
Yixiang Xiao, yx1215@seas.upenn.edu
Hanye Wang, hanyew@seas.upenn.edu
Shuo Sun, sunshuo@seas.upenn.edu
Yupei Sun, ysun09@seas.upenn.edu

1. Introduction

For this project, we built a web search engine based on the distributed crawler, indexer and pagerank. The crawler crawled web pages and stored the documents on Cloud, then the indexer and pagerank were created by MapReduce using the information fetched by the crawler. Also, a front-end user interface was built in React to allow users to retrieve relevant information and interact with our system. The interactions between components are supported by REST-style messages and Java Spark framework.

2. Architecture



The above figure shows the architecture of our project.

First part of the project is the distributed crawler. The crawler fetched documents from the internet, and saved them to S3, an AWS cloud storage system. Then both index and pagerank make use of the documents stored on S3. For the index part, the program takes the saved content from S3 and creates both forward and inverted indices. For the page rank part, the program creates a link graph which represents the transition between different links, and then applies the pagerank algorithm to the graph, calculating the pagerank for each document. Forward indices, inverted indices and pagerank are all stored in dynamoDB. After finishing both indexing and page ranking, the searcher can start to work. When a user enters a query, the searcher will find documents relevant to the query and give a score to each of these documents, and return the documents to the user with them sorted by the score calculated. The implementation details about how we crawl documents, how forward indices, inverted indices and page ranks are created, and how the document final scores are calculated will be discussed in the next section.

3. Implementation Details

a. Crawler

For the crawler, we use stormlite to build a concurrent, distributed crawler and we have crawled over 300,000 web pages using 9 EC instances. The main AWS tools that are used in crawlers are EC2, Simple Queue Service (SQS), DynamoDB, and S3.

The crawler is built based on the Mercator model. It is a multi-threaded program, so there are crawler master and crawler workers in our crawler, the master is responsible for creating and managing the workers, calculating the number of crawled web pages, and terminating the whole program under the specific situations, and the worker is responsible for crawling data from the website and processing and storing the data to the database. We implement a shared queue as a URL frontier to store URLs that need to be connected and crawled. Each worker will get and remove one URL from the queue as the first step. Based on the scheme of the URL, the worker will choose the right HTTP protocol module to connect to the website, and before crawling the data from the website, we will use 'cis455cralwer' identification to check robots.txt to make sure our crawling activity is polite and legal. Then the worker will crawl the data and check if the data is stored in the database already or not. If it is stored, the worker is supposed to stop the processing and storing of data and begin a new crawling loop. If it is not stored, the worker will store the hashcode of the content as checksum and MD5 hashes of the URL as visitedURL to avoid duplication, and store the content as a document to the database, which will be used by the indexer and PageRank. The third step is getting more links. When we parse the content, the worker will extract all links in the content using JSoup, and select the valid and unique links from them and add them to the shared queue.

In order to distribute the program, we use stormlite to separate the work into several spout and bolts, the difference between the stormlite topology that we used in HW2 and this project is that we remove the bolt, which is designed for parsing the DOM of the data and building channels, but we still using one spout for sending URLs to each worker and one bolt for crawling data, one bolt for processing, store data and extracting links from the data, and one bolt for check the validation of the extracted URLs and add the qualified ones to the shared queue.

Considering that the task is to crawl a large number of web pages, we use 16 threads and implement our crawler on the AWS to improve the performance and store the data. The shared queue is implemented using the Simple Queue Service(SQS), which will send a message with a URL to each request from the worker. And the workers are implemented in the ECs, we start 9 instances at the same time and each of them is linked to its own Simple Queue so that we can use 9 different seed URLs and broaden our crawling range, preventing crawling pages only in a specific theme or under limited domains. Instead of using local storage like Berkeley DB as HW2, we use S3 and DynamoDB to store different data. In S3, we store the content of each crawled web page and the name is the assigned doc ids. In DynamoDB, we create 5 tables to store different key-value pairs. The checksum table stored the hashed content of web pages, which is used to check whether the content is stored already or not. The visitedURL table stores the visited URLs in order to avoid URL duplication. The docid_to_url and url_to_docid tables stores the mapping relation between doc ids and URLs, and another fileToDoc table stores the mapping relation between S3 file ids and doc ids.

In order to speed up the process and decrease the times of accessing S3 and DynamoDB, we decide to store the content of websites in a local temporary txt file and append other websites' data to the same temporary file after that. And we mark the beginning of each website's data using "***DOCID: xxxxx", so that the indexer and page rank can find the data efficiently and correctly. Only when the size of the file is equal to or greater than 50MB, we will store the temporary txt file to the S3 storage and store the mapping relation of doc ids and file ids to the DynamoDB. When the program wants to find the data of one specific URL, it will look up the docid in the DynamoDB and then find the related file id and use the file id to find the data in S3.

b. Indexer

When creating the index, we made use of the Apache Hadoop framework and Elastic Map Reduce (EMR) clusters of Amazon Web Services for processing the data, and Amazon DynamoDB for storage. There are three main parts of the indexer implementation, all of which are implemented as a Hadoop Map Reduce job so that we can make use of the EMR clusters.

The first part is the data preprocessing job. It takes the crawled documents and corresponding document id as input. During the map phase, we first use the Jsoup html parser to extract the plain text from each document, getting rid of the html tags and other javascript texts, as these texts usually contain very little information, so we decide not to index them. The extracted plain texts are then split and lemmatized so that the same word in different forms can be normalized and better indexed. For each of the normalized and split words, we emit a tuple where the key is the document id and the word, and the value is the position the word appears in the document. Then during the reduce phase, the reducer simply gathers together emitted key-value pairs from the mapper with the same word and document id, and outputs the docid, the word, along with a list of positions representing the appearance of the word in the document id. For example, if the word "hello" appears in doc with id 1 at position 1,3 and 7, one output line of the reducer will be "(1, hello) [1, 3, 7]".

The second is the forward index job. It takes the output from the data preprocessing as input. During the map phase we simply split the input line and emit the docid as the key, the word and occurrence position as the value. During the reduce phase, for each docid the reducer collects all the words occurring in the document as well as the corresponding occurrence position list. Then the reducer creates two hashmaps. The first one is for the number of occurrences for each word, where the key is each word and the value is the corresponding number of occurrences. The second one is for the occurrences of each word, where the key is each word and the value is the first five occurrences of the word in the document. Besides the two hashmaps, we also use a variable to keep track of the sum of squares of each word's occurrences in the document, as it's needed for calculating the term frequency, and with it calculated in advance, we don't need to calculate it again every time when we need the term frequency, which can speed up search queries. These two hashmaps and the variable are saved together with the document id as an item into the forward index DynamoDB table, where the document id is the primary key. For the occurrences, we didn't save the full position list into the DynamoDB, because there is a 400 KB size limit for each item saved into any DynamoDB table, and doing so can easily exceed this limit for some long documents. Another reason is that the position lists are usually used for phrase checking. The first few occurrences of the word can already be enough for us to decide whether a phrase occurs in the document, because in most cases if two words occur as a phrase in the document, they will start to occur together in the document, so the first few occurrences will be very close.

The third part is the inverted index job. It also takes the output from the data preprocessing as input. During the mapping process, we still split the line, but this time we emit the word as the key, and the document id plus the position list as the value. During the reducing process, for each word the reducer will gather the documents in which the word occurs and create a hashmap where the key is the document id and the value is the number of occurrences of the word in the corresponding document. We don't keep the position list in the inverted index because we already have it in the forward index. We also use a separate variable(nDoc) to keep track of the total number of documents that the word occurs in. Then we save the word, the hashmap and the variable nDoc together as an item to the inverted index table in DynamoDB, where the word is the primary key. Same as forward index, the limit on each item will also force us to reduce the size of each item. When the item for a word exceeds the limit, we will repeatedly drop the key-value pair in the hashmap with the lowest occurrences until the total size of the item is lower than 400KB. We can eliminate the documents with a small number of the word occurrences because fewer word occurrences usually mean that the document is less relevant to the word, thus less likely to be considered as a valid searching result.

With the above described indexer structure, we only need to preprocess the data one time and also be able to run the forward index and inverted index in parallel to save time. If we make the forward index and inverted index job take the raw crawled documents as input, they can also run in parallel, but that way we would need to do the preprocessing twice, which is more time consuming.

c. **Pagerank**

Similar to the indexer, the pagerank also makes use of Hadoop framework, EMR and DynamoDB. The pagerank consists of two parts, both of which are implemented as Hadoop Mapreduce jobs.

The first job is used to create the linking graph between different documents. It takes the raw crawled documents as input. We use DynamoDBMapper to get the URL of a document and parse it with Jsoup to extract the outlinks from the current document, then map them into document id's using DocidToUrlItem from the edu.upenn.cis.utils package. In this step, the mapper has document id as key and document body as value to process the document links, and the reducer takes the current document id and the document ids of its outlinks and concatenates these document ids into a string. This step will reduce the length of data stored in memory since the length of the document id can be much shorter than the URL. If the document does not have outlinks, we add a link to itself to prevent rank sinks and dangling links.

The second job is the application of the pagerank algorithm to the graph. In calculating pagerank scores, we create a PageRankItem object in the DynamoDB table, where the primary key is the document id and the other attribute is pagerank score. Then we use mapreduce to calculate pagerank score for each document based on the number of document ids of its outlinks (outdocument ids) and the damping factor. In the mapper, we first initialize the pagerank score to be 1 for each document if it does not already have one in the DynamoDB table. Then, for each of the output documents we emit a tuple with key being the out-docid and value being the pagerank score of the original document divided by the total number of outdocuments. Then the reducer will gather all the scores sent to each document and calculate the final pagerank score using the dumping factor.

d. Search Engine and Front-End

The front-end is based on React using Ant Design css library. We implemented a Google-like search page and result page. The search bar utilizes NLP techniques to do auto-complete. The result description will be chosen according to the query phrases.

The search engine backend server consists of two parts, the server for REST API and the server for NLP. We run the two servers on the same EC2 instance locally so that the frontend can directly call localhost without Internet delay.

The servers have only two user APIs which are for getting query results (/api/result?q=query&p=1) on the SparkJava server and for getting choices for search auto-complete (/api/choice?q=query) on the Flask server. We choose to make up result JSON for frontend on the SparkJava server side, so we use Jsoup to parse the result urls and find the title and the sentence where the query phrases appear most frequently in the html file to be the description. Here we come up with a O(m*n) algorithm to find the sentence mainly using HashMap.The Flask server utilizes the BERT NLP model to do a word prediction task so that the search auto-complete function is totally lexicon-based.

The whole search engine server is deployed by NGINX on a m5-large EC2 instance and only supports HTTP connection because of the lack of SSL certificate.

Whenever a user gives a search and presses the search button, our search engine will first go through the documents where the words in the query occur and calculate a score for each document based on the tf-idf and the pagerank. Suppose V_d is the document vector, V_q is the query vector, p is the pagerank, and d is the average distance of between query words in the document, score for the document is calculated as $p(V_d \cdot V_q + 5/d)$. Note that smaller the d, higher the score. So the d here allows our search engine to process phrases. After calculating the score for the relevant documents, it will give out a ranked list of urls to our search engine server. And then, the search engine server will go through the urls and make up a json array containing the url, title and description of the webpage. At last, the client should see the page with results.

4. **Evaluation**

a. **Crawler**

Before optimation, the speed of crawling is unsatisfying, one instance can crawl one web page every 3 seconds, and we use higher performance machines and using more threads, we crawled over 300,000 web pages in two days and the average time of 5000 web pages for each instance is about 30 mins.

The speed of different instances is various, and one main cause of this phenomenon is the content of the web page, crawling a news website is about 3 times faster than crawling Wikipedia pages.

For DynamoDB, the time for scanning the content hash value in the checksum table is about 22ms, and writing an item to the checksum table is about 4 ms. Scanning visitedURL table and writing the URL to this table is 3ms and 1ms. The time for scanning and querying for url_to_docid table and docis_to_url table is similar because the data stored is similar, just exchange the key and value, it is about 20ms for scanning and less than 10 ms for querying. As for the fileToDoc table, scanning, writing, and querying cost 6 ms, 4ms, and 3ms respectively. Obviously, scanning the whole table to check duplication costs more time than just writing new items to the table.

The S3 storage we use about 73.9GB to store all data of web pages in 2k document, the average size of each file is about 37MB, considering that some documents are not uploaded automatically by the crawler because the crawler is terminated under some condition but the temporary txt file is less than 50 MB, we upload this kind of txt files using another program. Therefore, the average size is reasonable.

In the Simple Queue Service, we implement 9 queues for each instance. After crawling 300k web pages, the longest queue remains 4M URLs in its line, and the shortest queue has 156k URLs.

As for the EC2, for all instances, the CPU utilization is about 99%, and all of the CPU is used fully considering that we have 16 threads for each crawler program.

b. **Indexer & Pagerank**

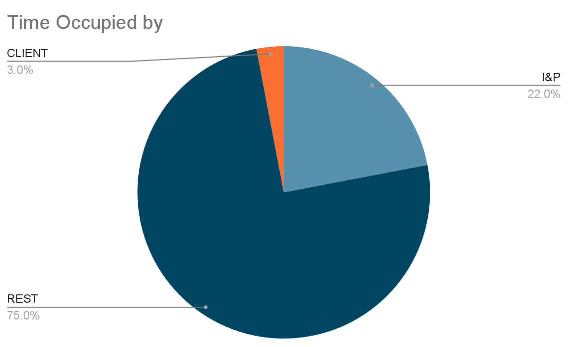
For indexer and pagerank, since they both made use of the EMR, we tried to use clusters with 2 workers and 5 workers to finish the jobs to test the performance improvements. The following is the table showing the running time for indexer and pagerank. For indexers, both forward index and backward index are tested using 300000 documents. For pagerank, the time is the one used to run 10 iterations of the pagerank algorithm.

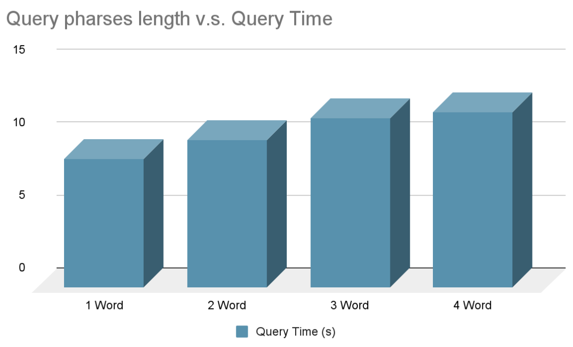
Num of cluster workers	Forward Index Job	Inverted Index Job	Pagerank Job
2	9h 43min	8h 17min	10h 36min
5	3h 51min	3h 28min	4h 42min

We can clearly see from these results that increasing the number of cluster workers will improve the performance of the map-reduce jobs.

c. Overall Performance

For simplicity, we call the indexer and pagerank part the I&P part, search engine part the REST part and frontend the CLIENT part. We are trying to find bottlenecks in these parts and have done some statistical analysis. What should be noticed is that the CLIENT part is decided by different hardware.





As we can see from the left graph, the REST part shows an unusually large span of time occupation. After testing on each module in the REST part, we find that the time consumed mostly comes from retrieving files from S3. That's where we can make an optimization. We know that S3 is only optimized for large files, so we ensemble html files together to make a large file and then store it on S3. However, as the graph shows, the performance is not satisfying as well. Another approach we came up with is when we get the sorted urls, we can crawl the document contents again to substitute the retrieving process.

If the above approach works, we should expect a great acceleration on the response speed since if we simply remove the code where retrieving files is involved, the REST part will decrease to 5% in the left graph above and average search time is in 3-4 seconds. Our current performance is shown in the right graph above which is a bit slow because of the retrieving process taking lots of time.

5. Conclusion

To conclude, we've successfully built all four parts of the search engine, including the crawler, indexer, pagerank, and front-end search engine. The overall performance of the search engine is good, giving reasonable results for most of the given search queries. One issue with our system is that it takes a relatively long time to process each queries, especially those with multiple words. Most of the time is used to fetch items from DynamoDB. This might be the place that future improvements can be made to. Another issue could be that 300000 documents is still a relatively small document set, which can lead to some keywords still missing in our lexicon. So by crawling more we can always improve our search results.