Project Report

Team Members: Saurabh Raut, David Binstock, Jay Gala, Heena Nagda

Bitbucket Repo: https://bitbucket.org/jaygala99/searchengine/src/master/

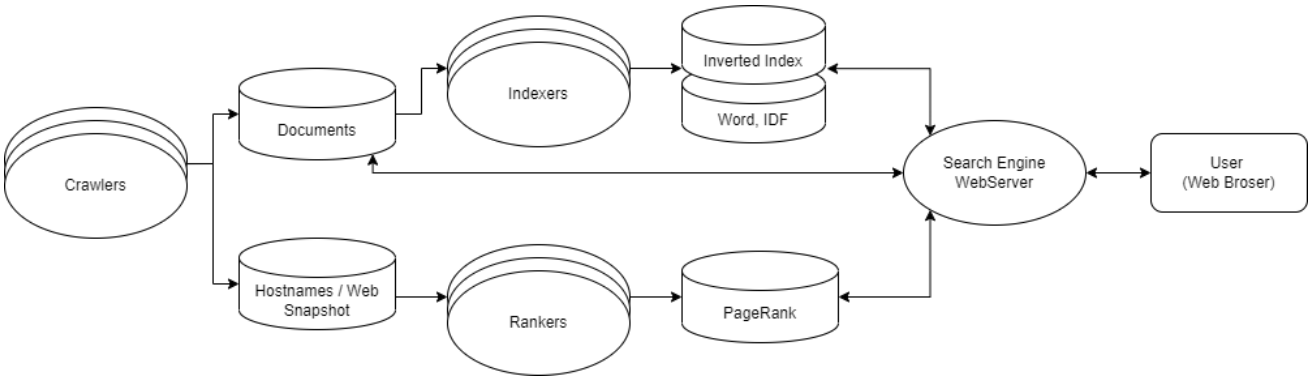
Introduction

The goal of this project was to create a simple search engine, similar to Google, that initially crawls the web for html documents, and then provides a user interface for searching for documents based on a query, or combination of words. The results of the search were to be informed by several different criteria, including pagerank and measures of word and documented frequencies calculated over the corpus. Furthermore, the architecture was intended to be distributed; for this various AWS services were leveraged. We were able to successfully crawl about 500,000 documents and query over this corpus for relevant documents based.

Background

Several different resources and papers were used as inspiration for the design and execution of our project. The google paper discussed in class was helpful in understanding the overall structure of a search engine, though our implementation did not have all of the components indicated in the paper. The various components of our search engine were also all discussed in class, and the relevant slides were used for insight and inspiration, particularly for indexing, pagerank, and final the calculation of a document score and relevance. The mercator paper, which discussed scalable distributed crawling was also referenced, however it should be noted that a slightly different approach was used for our project. Regardless, the paper had lessons and insights that were helpful when thinking through and ultimately implementing our crawler design. Additionally many previous homeworks involved components of this project, including the crawler and map-reduce. These homeworks were used as a starting off point for writing our final code.

Description of Architecture



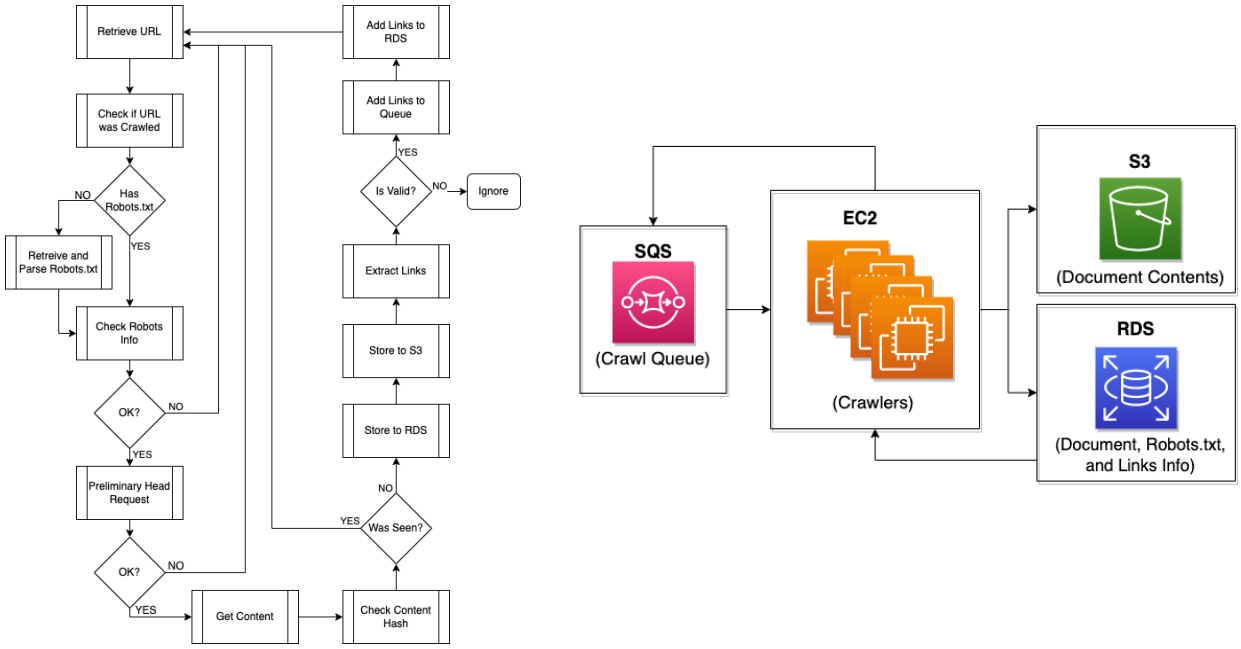
Broadly speaking, the architecture includes four main components which operated relatively independently: a crawler, indexer, ranker, and the search engine web server. The crawler was responsible for crawling urls and storing documents, as well as keeping track of links between sites. The indexer was responsible for parsing document contents by word and generating the indices and

other statistics needed for the search engine. The ranker used the links between sites generated by the crawler to compute the PageRank for each page that was crawled. Finally the search engine server was responsible for interfacing with the user and displaying results based on the user query and the various parameters computed by the indexer and ranker. The crawler, indexer and ranker are all distributed, operating on multiple machines.

First, URLs were crawled using a crawler similar to what we built in HW2, however the crawler was modified to be distributed and to be able to handle a large volume of documents. This data was stored in a distributed manner using several AWS products such as RDS for relational database storage and S3 for distributed file storage. The indexer was then used to generate the inverted indices (term frequency and document frequency) and the Page Ranker was used to generate the page rank of each document; these were both done as MapReduce jobs using Hadoop. Finally our search engine addresses the queries from users in an efficient way.

Implementation

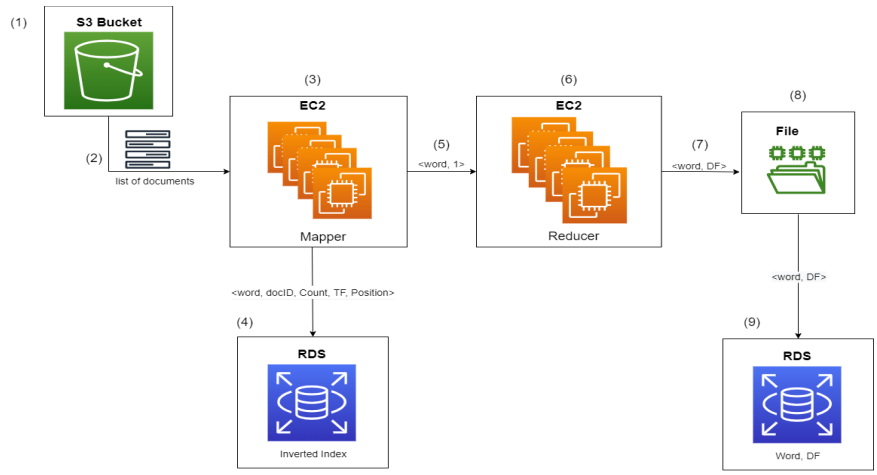
Crawler:



First, URLs were crawled using a crawler similar to what we built in HW2, however the crawler was modified to be distributed and to be able to handle a large volume of documents. At the recommendation of TA's, and in the interest of mitigating memory performance issues, Amazon's SQS (Simple Queue Service) service was used as a distributed queue, rather than each crawler thread having its own queue, as discussed in the mercator paper. The crawl queue was therefore implemented using AWS's SQS as a distributed queue accessible to all crawler instances. The data was stored in a distributed manner using several AWS products; document information, as well as links and robots.txt information were stored in RDS (AWS's relational database) and the document contents were stored in S3 (AWS's distributed file system). Each crawler utilized multiple threads, and there were six different crawlers each running on an EC2 instance.

The output of the crawler were RDS tables and S3 files that would then be used by the other components of the system. Overall, about 500,000 documents were crawled. The crawling itself only took about a day or two, once the program had been optimized. Part of what made the crawler relatively effective was that very little internal state was kept in the crawler instances. Instead all major data structures were saved within AWS products. The queue was saved in SQS, the document and robot information was stored in RDS, and the document content was stored in S3. Because of this, program memory did not need to store much information and heap size never got out of control. Additionally, this distributed architecture made it simple to start and stop crawlers as all state was stored externally rather than in memory.

Indexer:



1) An S3 bucket stores all the crawled pages in multiple 100 MB documents.
2) We send the list of all the documents to the Mapper
3) The Mapper runs the map job (in a multi-threaded setting). We have 4 EC2 instances running multithreaded Hadoop map-reduce jobs
4) The mapper function sends each word, docId and its corresponding count, TF, and positions array to the RDS database (in batches)
5) It then sends the word, with the value of 1 to the reducer
6) The Reducer gets a word and a list of 1s as input. Each 1 corresponds to the word occurring in a document. The Reducer counts the document frequency of each word
7) The Reducer outputs word and document frequency (DF)
8) This information is stored in files by the reducers
9) Then finally, the data from the files are sent to the RDS database (in large batches)

We made use of 4 EC2 (m4.xlarge) instances running together to perform the map-reduce job.

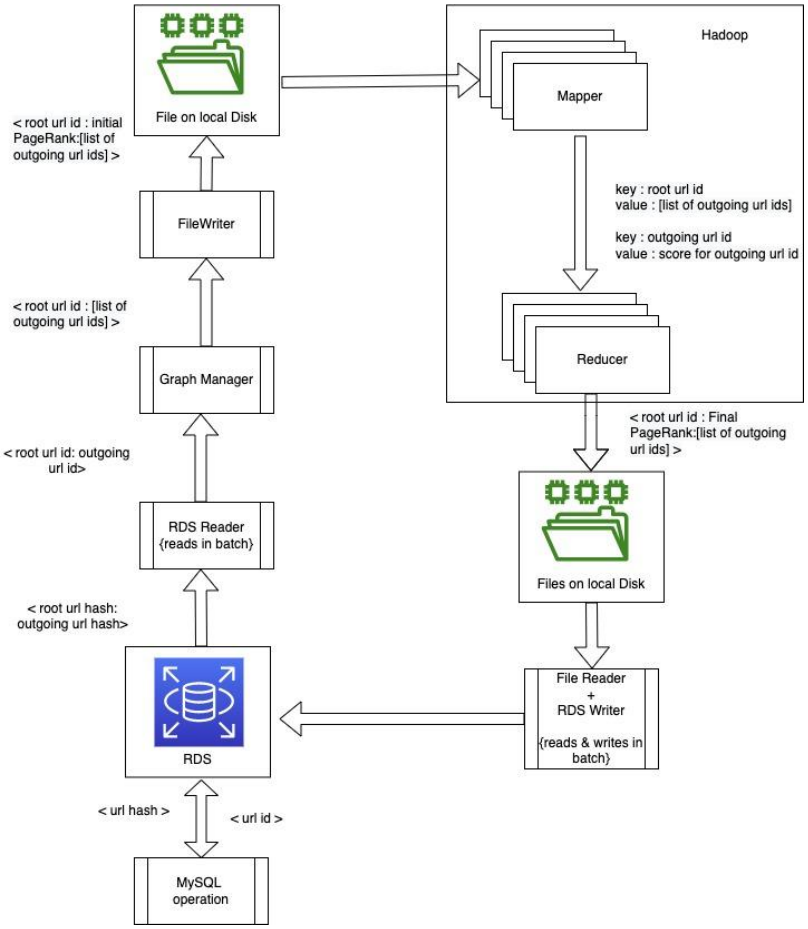
Description of Mapper:

We used a multithreaded Mapper (4 threads). The input to the Mapper was the name of the document in the S3 bucket. The Mapper would then retrieve the contents of that document from the S3 bucket. The documents themselves contained multiple crawled HTML pages (about 500 to 700) which were separated by a delimiter and then a line indicating the document ID followed by the actual HTML content. The Mapper then split each document into the HTML pages, and for each HTML page, it extracted the document ID and the content. Then, the mapper used Jsoup to parse the HTML page and extract only the text. Then, it used Stanford's CoreNLP library to tokenize and lemmatize the extracted text, followed by removal of stop words and some other sanity checks and cleaning operations. After

which, the word was added to a map and the count of the word was noted (i.e. how many times was the word present in the document) and the position of the word was also noted (we only store a particular word's first 10 positions). After going through all words of a particular HTML page, this pages' word information was sent to the RDS database in a batch. This information included the word, document ID, the count, and its term frequency or TF (calculated using TF with norm and value of a = 0.4). The process was then repeated with the next page in the document. All of this was multithreaded and used 4 threads for each mapper. Each of the 4 EC2 instances was used as a Mapper, and then a Reducer later.

Description of Reducer:
We used 8 Reducers per EC2 instance. The input to the reducer was the word as key and an iterable of integer 1. The total length of the iterable indicated the number of documents that contained the particular word. So, in the Reducer, we calculate the number of 1s in the value, which corresponds to the document frequency and store this in a file (which is done by the Reducer implicitly using context.write()). After all the Reducers were completed, each file was read and all its words and corresponding document frequency were sent to the RDS database as a large batch to reduce on the network latency and write delays on the RDS. Moreover, since all this was being run on multiple EC2 instances and also in segments, we were storing document frequency instead of IDF and if the word already existed in the RDS database, then we just updated its DF count. This was easily done using an SQL query.

PageRank:



PageRank was implemented using Hadoop Map-Reduce. The program took four inputs from the command line. (1) Number of iterations for the PageRank, (2) PageRank local disk directory, (3) Task to be done, (4) flag if batching is to be done or not. The whole process was split into 3 isolated tasks:

Task-1:
First the “root url id to outgoing url id” table was read from the RDS and used to create an in-memory adjacency matrix hash map with key as root url id and value as list of outgoing url ids. Once the in-memory matrix was formed, this data was stored into a file on local disk (named hadoop_in.txt) in the format {root url id | initial Page Rank | outgoing url id 1 | outgoing url id 2 | outgoing url id 3 | …}.
Task-2:
Next, the Hadoop map-reduce topology was created, and the hadoop_in.txt file created in part 1 was used as an input to the Hadoop map job. The number of reducers was set to 20. Once started, the map-reduce job took about six (6) seconds to complete the 15 iterations. The number of iterations were used instead of checking convergence to improve the time complexity. At the end of the last iteration of map-reduce the final score was stored in the files on the local disk. The Mapper read line by line from hadoop_in.txt and emitted <outgoing url id> as key and <root url page rank / number of outgoing links from root url id> to the reducers. The reducer then aggregated the incoming page rank score for each url id, and then added the damping factor to remove sinks and hogs.
Task-3:
Finally the process read the data (final PageRank) from reducer output files in chunks and sent them to the RDS in batches. Writing in batches reduced the writing time significantly.

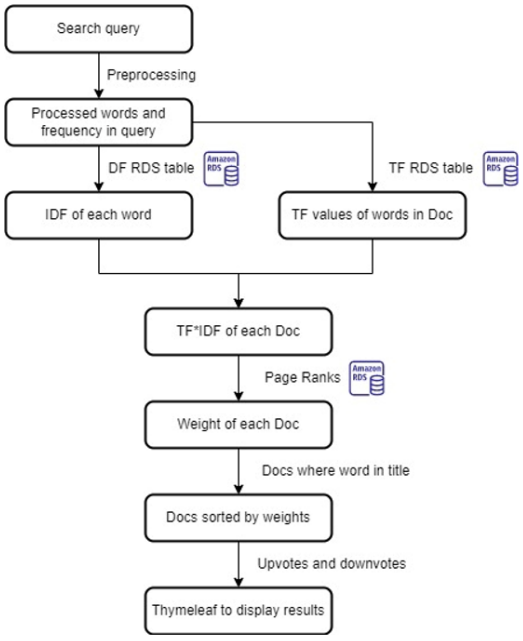
Issues and Resolution:
When we tested the pagerank solution with the small corpus (~40,000 crawled documents) performance appeared to be reasonable. However, switching to the larger corpus (~500,000 crawled documents) led to various performance issues related to time and space.

The following steps were taken to improve the time complexity issues:
(1) Increase the number of reducers from 5 to 20.
(2) Writing final pagerank from local disk file to RDS in batching instead of writing each entry individually.
(3) Converted “root url hash to outgoing url hash” table into integer id table using SQL query before starting the pagerank instead of converting them at the time of pagerank internally.

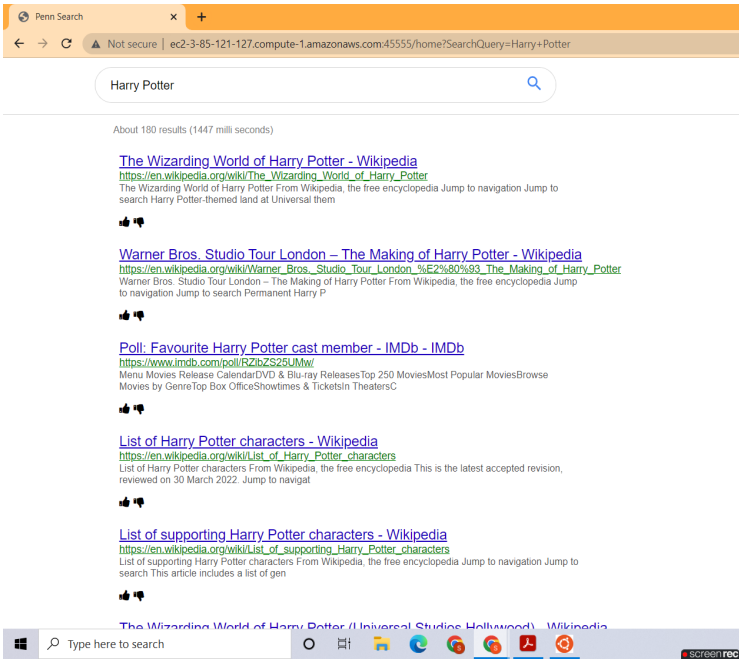
The following steps were taken to improve the space complexity issues:
(1) Instead of reading the large outgoing link table at once in the memory, we started reading the table in chunks. This reduced the memory requirement significantly.
(2) Previously We were using the `Long` data type to store the url ids. However, the ids are just the incremental count that can maximum reach up to ~500,000. So using `Integer` for the ids is a reasonable choice and reduces the memory requirements.
(3) Increased the Java heap memory before starting the pagerank process.

SearchEngine & UI:

Search Engine



Search Results UI



The search engine logic worked as follows:

(1) Preprocess the entire query using Stanford NLP library
(2) Compute weight of each word by querying DF table of indexer
(3) Generate tables in SQL of documents for each query word. This is limited to 200 and sorted in descending order of Counts. Join these tables (server-side join) to get documents that contain all the query words. Limit these entries to 200 again sorted in descending order of counts. Note that if only very few documents (less than 10) contain all words then we take words from the search query in sequence of IDF values and take at most 200 entries.
(4) For these at most 200 entries, we have Document-Word-TF and Document-Word-Count map.
(5) Compute TF*IDF values according to the Document Vector model. Note that if the Word-Count of some word is very high then we add 1 to its TF value to give it more weightage.
(6) Compute PageRank for these documents.
(7) Some pages had very high pagerank so first normalize the page ranks.
(8) Compute document score for these at the most 200 documents by adding TF*IDF and PageRank values.
(9) Query the crawler table for documents that have all query words in their title and assign these pages high document weight.
(10) Merge these special documents with the documents from Step 8.
(11) If the document is upvoted or downvoted then increase or decrease that document's weight depending on the count.
(12) Pass all these documents to Thymeleaf to render the UI using HTML and CSS in paginated fashion.

Extra credit features in Search Engine:

(1) Implemented the advanced part of extra credit feature of upvote and downvote mentioned in Search Engine and Web User Interface part of Project PDF. On each upvote or downvote by the user, an AJAX call is made that increases or decreases the ranking of that document.

(2) Added in memory caching of search results document's hashes to avoid repeated search procedure for same query.

Note 1: All the tables that are queried to get search results already have SQL index based on the query's required column.
Note 2: CSS for Search Results was taken from: https://codepen.io/MuminjonGuru/pen/XQBOge

Evaluation of Results

Overall, about 500,000 documents were crawled, and crawling took about two days, with around 12 hours of total crawling. We could have easily crawled more documents, but stopped earlier in the interest of leaving time for performing the indexing and page rank calculations, testing out the search engine and user interface, and fine tuning the overall project. The Indexer identified around 4 million unique words and performed the calculation in a couple of days.

The PageRank calculation was very efficient: step 1, reading the raw data (root url id : outgoing url id) (for 500,000 documents) from RDS, creating an adjacency matrix and storing into the disk file, took 139159 ms. Step 2, running hadoop mapreduce over the data with 20 reducers, took 3444365 ms. And step 3, reading the final page rank (for 500,000 documents) from the local disk and writing to the RDS (writing in batches), took 6396 ms.

Simple search queries appear to produce generally relevant documents. Some searches return better results than others, but on an average, one word search queries take about 1200 milliseconds. For each additional query word the result time increases by about 200 milliseconds. The quality of search results seemed to be pretty good after all the fine tuning and after additional factors consideration.

Areas for improvement

Crawler:

There are a number of areas for improvement. First of all, given more time, we could have easily crawled for more documents. Unfortunately the dependence of the Indexer and PageRanker on final crawled results coupled with the time needed to complete these steps required us to cut short our crawling in the interest of leaving time for final debugging and tuning.

Another improvement would be additional measures to ensure a more broad set of domains crawled. A large quantity of pages crawled were from wikipedia, which works well in terms of topic breadth, however results might be enhanced with a broader range of web domains. In an effort to reduce the memory used by each crawler, distributed systems were used rather than each crawler thread having its own queue and using a storm cluster to group field group urls. However there are additional measures that could be taken to establish a more even crawl of urls, for example keeping a global count of pages crawled per domain (on RDS) or perhaps creating several different SQS queues which have domains sent to them via a hash and alternating between these. Alternatively each thread could have its own internal queue limited to a small number of urls in addition to the global SQS queue. Given more team, the team would investigate these approaches to see if they could yield better results

Indexer:

We could have further improved the speed of the mapper by having even larger batches across multiple mapper threads and making the entire process thread friendly. We could have also found a substitute to

StanfordCoreNLP which took some time to annotate the document. Moreover, we could have used a better lemmatizer than StanfordCoreNLP which did not work very well for some words.

Additionally, we could have done more investigation into the various AWS resources and their different configurations, analyzing each of their pros and cons in order to determine which ones suited our specific use case. For example, we could have explored the option of using EMR to perform the map-reduce job for the indexer.

PageRank:
Intra domain loops are not taken care of in the current page rank implementation. For example there are many pages that crawled with the domain “imdb”. In this case every single page that is of “imdb” domain has a back edge to the “imdb” home page, and the home page of “imdb” therefore has an unreasonably high page rank. However, the algorithm should emphasize interdomain links more instead of intra domain pages. To improve this algorithm should omit the intradomain back edges.

Search Engine:
A few thousand top page ranked pages could have been cached in a small table along with their TF scores of its words. Now, on getting a search query we could start by checking if all query words are there in these documents. If yes, then display these documents first and then do the remaining procedure. If we do this then we will get a few good results very quickly because we are querying only one table and then remaining results can be displayed after these. Furthermore, calculation of the final document score could be fine tuned even more to produce a better result by looking at query phrases and relative positions of words in documents.

Conclusion

Ultimately the project was successful; all the independent components work and produce their desired result. We were successfully able to crawl 100s of thousands of documents, formulate inverted indices, calculate their TF and IDF scores, calculate their page rank, and aggregate this all into a final analysis that could generate on the fly results to a given query. While there is certainly improvement that could be made, from the quantity and quality of crawled data, to the calculation of document statistics, to the overall query algorithms and final returned results, the functionality is all there and working. Our search engine is able to successfully take in user queries and return relevant documents from a corpus of pages that we crawled, indexed, and ranked ourselves.