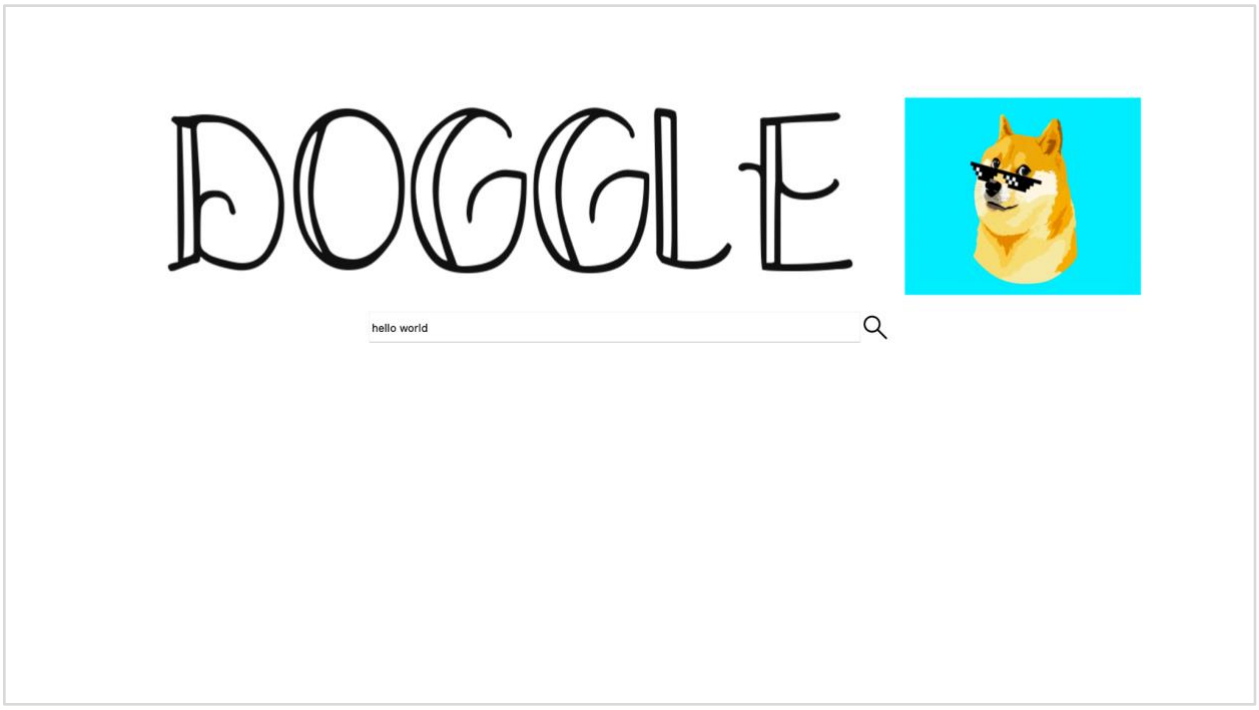
Doggle Search Engine

Hongyou Lin, Jiyu Huang, Zijing Wu, Yunxing Zhang



Introduction

A. Project Goals & High-Level Approach:
    a. Create databases and schemas for fast and accurate query responses
    b. Crawl information (including URL and documents) from a diverse websites and providing information for potential search results
    c. Perform ETL procedures toward data crawled for further processing purpose
    d. Using Indexer and Pagerank to rank the document results, rendering the most relevant results possible
    e. Build an interactive search engine interface for user's search experience; Simple yet cute frontend to intrigue users and robust back end for query processing
    f. Deploy search engine on the cloud for extensive testing/demo

B. Technology Stack Used:
    a. Database & File Storage: AWS DynamoDB, AWS RDS, AWS S3
    b. Crawler: AWS EC2, AWS SQS
    c. Indexer: Apache Hadoop, AWS EMR
    d. PageRank: Apache Spark, AWS EMR
    e. Search Engine & Web Deployment: SparkJava, AWS BeanStalk

C. Division of Labor:

We divided the work into four parts based on the structure of our project architecture:

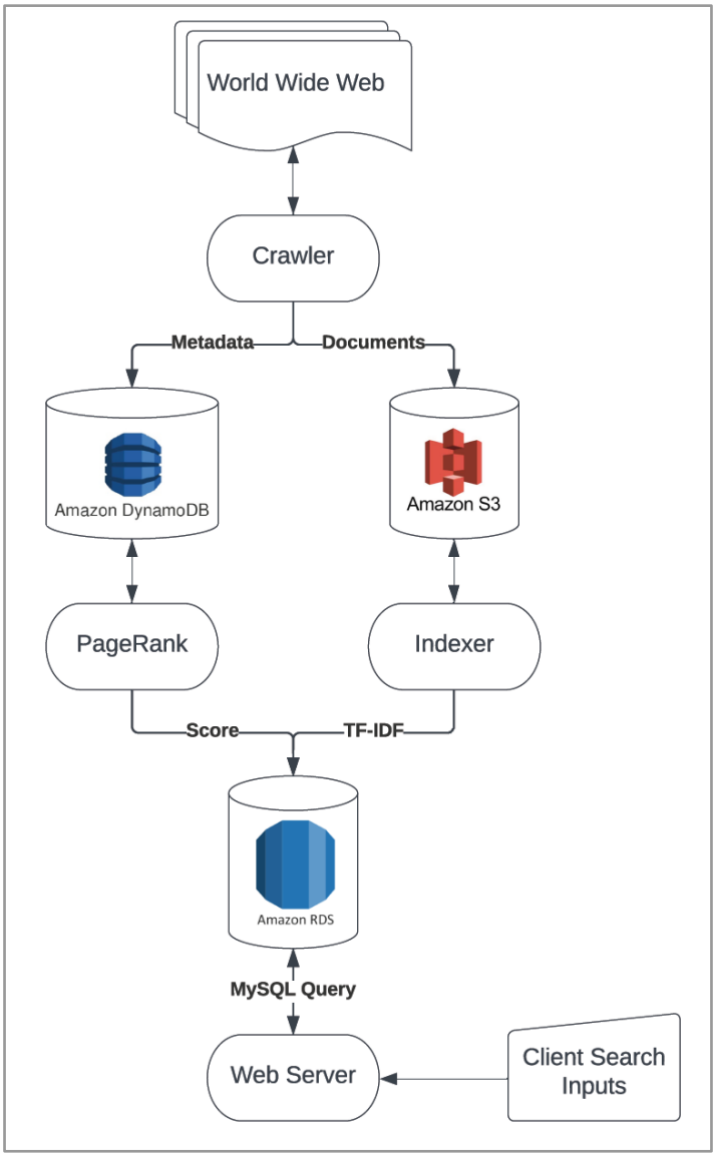
a. Crawler: Hongyou Lin
b. Indexer: Jiyu Huang
c. Page Rank: Zijing Wu
d. Search Engine and UI: Yunxing Zhang

During implementation, we made sure that every member of the team is intimately familiar with every part of the architecture and project interface, and had vital input in the integration, debugging and deployment process.

System Architecture

A. Overall Architecture:

The high-level architecture of DoggleSearch involves 4 major components: the Crawler, the Indexer, the PageRank and Web Server & Search Engine Interface, as shown by Fig-1 below. Each component is functionally dependent on the results produced by its ancestor.

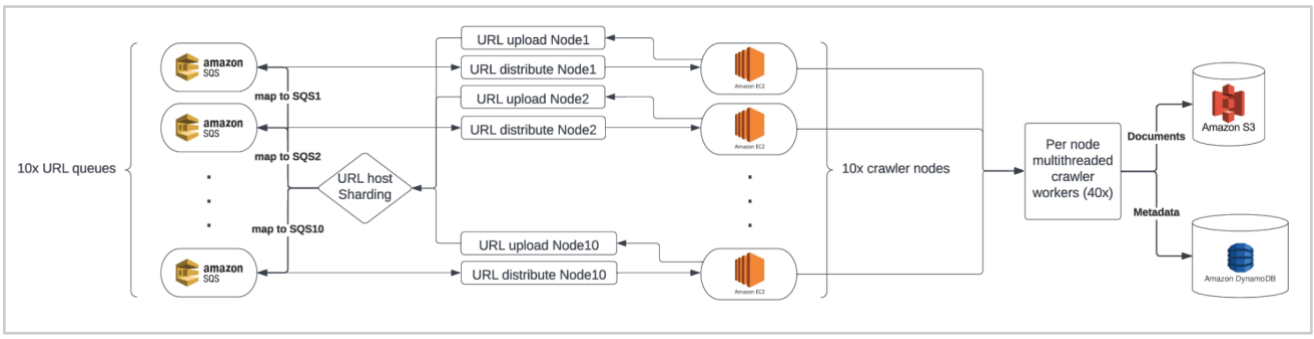


(Fig-1: Overall Architecture Diagram of DoggleSearch)

B. Crawler:

The goal of the crawler is to crawl a diverse set (~1,000,000) of HTML documents from the internet based on a principle similar but not equivalent to the breadth-first search (as mentioned in the Mercator paper, there are modifications of the FIFO queue data structures to implement the politeness constraint), thus it must be distributed and scalable in order to be

efficient. The crawler architecture involves 10 queues running on AWS SQS pairing with 10 EC2 instances, where each EC2 instance is running 40 crawler worker threads, 1 URL distributor thread and 1 document saving thread. The crawled documents & metadata are saved in AWS S3 and AWS DynamoDB respectively, and each HTML document is uniquely identified based on the node ID and its own ID within the batch of documents crawled by that node (docID = Node#-doc#). The architecture diagram for the crawler is shown in Fig-2 below.



(Fig-2: Crawler Architecture Diagram)

Each EC2 instance takes in a batch of URLs each time via the URL distributor thread and populates a global blocking queue of crawling-task queues based on the host of the URL. In other words, each host is uniquely matched to a queue of tasks (URLs & its other information). The 40 worker threads will poll the task queues from this global queue of task queues, perform crawling (run HEAD & GET requests, check crawl-delay, content-seen, robot.txt & language, perform URL normalization, get hrefs, and finally save these documents to DB asynchronously), and append this task queue with one task removed back to the global blocking queue.

Crawling politeness & coordinations among these different nodes of multi-threaded crawlers are achieved by sharding. The goal is to make sure that URLs from a specific host are always gonna be crawled by a uniquely identifiable crawler. To achieve this, each time there are new URLs crawled from a HTML document, it is uploaded to one of URL SQS queues based on its hash value of the host modulo the total number of SQS queues. Thus, in this way, we know that the same URL will always be processed by a specific crawler at all times, preventing repeated savings on documents while achieving scalability.

To ensure the best crawling results, we need to make sure that the crawler is never trapped in some URL loops, which could be caused by several different situations – infinite redirect loops, dynamically generated contents, faulty links, internal self-referential links, etc. In order to avoid these crawler traps, here are some strategies we adopt:

1. We make sure that the initial URL seeds do not produce too many internal self-referential links (i.e. news websites);
2. We limit redirection for 3 times for each URL;
3. We only crawl documents that return a status code of 200 for HEAD or GET requests;
4. We make sure that all hrefs are normalized before populating it back to the url queue;

5. We enforce a limit of 100 documents to be crawled and saved for each host, so that even if a specific crawler worker falls in one of the crawler traps, the whole pool will still be composed by documents from a diverse range of host;

Overall, we use this crawler implemented by the architecture mentioned above to crawl about 900k documents within 2 hours, and the results are saved in AWS DynamoDB and S3 for further processing by Indexer and PageRank.

C. Indexer:

The indexer reads and parses the crawler's output stored in S3, performs a Hadoop MapReduce operation and stores the output as plain files in S3. After running the indexer job, the text output gets loaded into an AWS RDS database table, with word and docId as joint primary key. The implementation details are described as follows:

The mapper of the indexer takes a docId and the content of the document as a key-value pair, both of which are parsed from crawler's output files. It then tokenizes both the document title and the content, and counts the frequencies of each stemmed word. For a token to be considered as a word, we've decided that it needs to be alphanumeric and cannot be one of the stop words. After counting the word frequencies, we calculate each word's doc-TF score using the formula: 0.4+0.6*freq/maxFreq, and emit key-value pairs with the word as key and the docId and TF score as value. In the mapping stage we also keep a record of title hits and emit that as well. The reducer uses the key-value pairs to compute the IDF score for a word, and outputs the TF/IDF score and the title hits for all the documents.

D. PageRank:

Hadoop Spark in Java and Elastic Mapreduce on AWS were used for PageRank algorithm realization. The algorithm was built with specifications taught in class: 0.15 for alpha as decay factor and 0.85 as beta. For convergence, we set a maximum of 15 runs. For pagerank workflow, The data for in-degree link and out-degree link was fetched from the crawler's table from DynamoDB. The result of the ETL process was stored in Spark dataframe with two columns: From(URL: String) and To(URL: String). After RDD manipulations, we uploaded our data to both S3 and DynamoDB. The results were later loaded from S3 to Aurora RDS for search engine usage.

E. Web Server & Search Engine Interface:

The search engine and UI are built on Spark framework. The main class *SearchEngine* defines the port and routes and starts multi workers. The front-end consists of *MainPage* and *SearchPage*. In *MainPage*, the server will initialize AWS service and create connection to AWS RDS for later use. The interface consists of a picture of our doggle search and a text box where users can input their search queries and then click the search button. The *SearchPage* will

display the results from the server side, and show the number of results and time of searching progress. There will be 10 documents in each page, and the maximum number of results we will display is 100. Users can choose the page they want to visit at the bottom of the search page.

The bulk of the search engine operation happens on the server side. A *WebMaster* class contains a connection *conn*, a blocking queue *queryQueue*, three maps *queryMap*, *timeMap* and *testMap* record all information we will use later. The *getIds()* function in *WebGetResults* is the main function we retrieve desired results depending on index and pagerank database. It will first normalize the input query (remove punctuation and multi space, filtered stop words, transfer into lowercase) in order to stay consistent with Indexer. Each query is split into a list of multiple terms. Then search each term in the Indexer table, retrieve the top 100 results ordered by their index score. The index score is consists of two parts: TF*IDF score and isTitle score. If a term is shown in the title, it is probably a key word in the document. The score is TFIDF+0.5*isTitle. In this step, we also collect corresponding docId, and their pagerank score by joining tables together. Put indexer score in a result map first, the map's key is docId and value is index score. If a repeated docId appears, accumulate its score. Put pagerank score in another result map, key is docId and value is pagerank score. Now each map should have the same number of entries. Before the next sum up step, find the maximum values in each set so that we can normalize all values. For each docId, calculate its final score by

$$final = 0.8 * TFIDF + 0.2 * pagerank.$$

If a pagerank score is NULL, we treat it as 0.15, since in our pagerank score calculation the minimum value it could be is 0.15. At last, this function will sort the final scores, return an ordered list of docId.

The *WebWorker* is the thread of a single worker. Taking a query from *queryQueue*, it will call *WebGetResults.getId()* and get the list of results *ids*. Then retrieve documents from dynamoDB, using *ids* and call *DynamoDB.getDocs()*. Now we have a Map<docId, Doc> ordered by correlation. Add those Doc objects in a result list, and put the searching query and its corresponding result in *queryMap*. In *searchPage*, when a web request including query parameter comes in, it will put the query in *queryQueue* first, then a WebWorker will handle it. *SearchPage* then takes the result from *queryMap* and displays all information we described above.

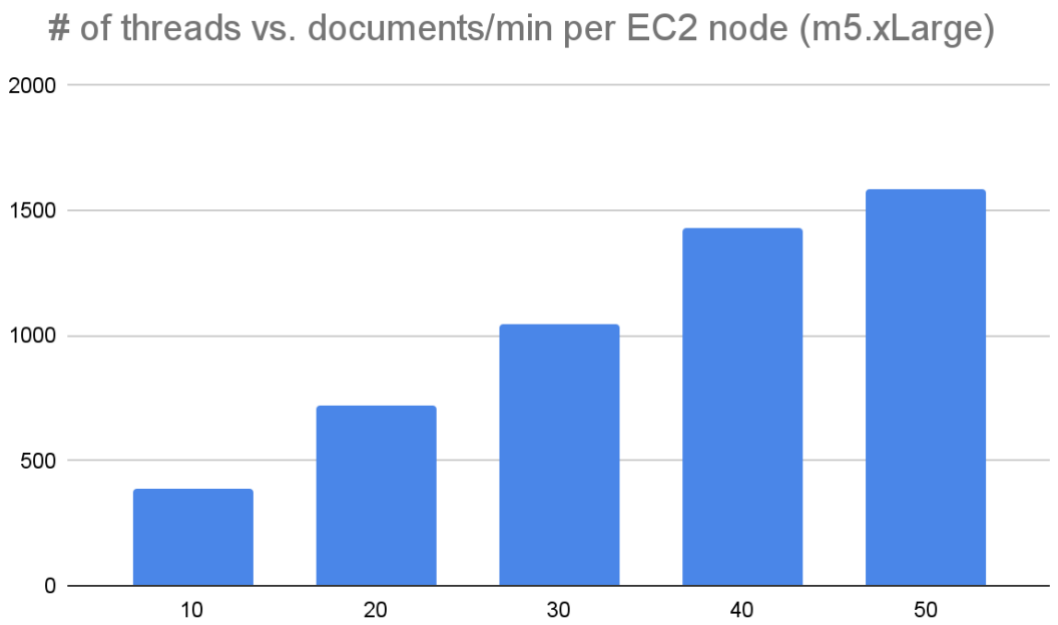
Evaluation

A. Crawler:

As mentioned above, the crawler was operated on 10 separate EC2 instances (m5.xLarge) matching with 10 SQS queues, and each EC2 node is running 40 crawler worker threads. This configuration is chosen by experimentation. We have tried modifying the number of threads for each EC2 instance and modifying the number of instance-queue pairs (EC2 & SQS).

The result for modifying the number of EC2 instance & SQS queue pairs is relatively straightforward – there seems to be a linear relationship between the number of nodes and the amount of documents crawled for a given period of time. This theoretically makes sense because these nodes are operating independently from each other, because the only coordination among them is the URL host sharding computation for populating URLs to the desired SQS queues. Thus, we could easily scale up the performance by increasing the number of nodes running at the same time, as long as the queues do not become empty. This is true for our experiments with 1, 5, and 10 EC2 nodes with each node crawling for 10,000 documents – we simply got a linearly increased amount of documents (10,000, 50,000, 100,000) for the same period of time. Maybe with an increasing number of queues and nodes, the sharding computation will be more costly, but it should be negligible in general.

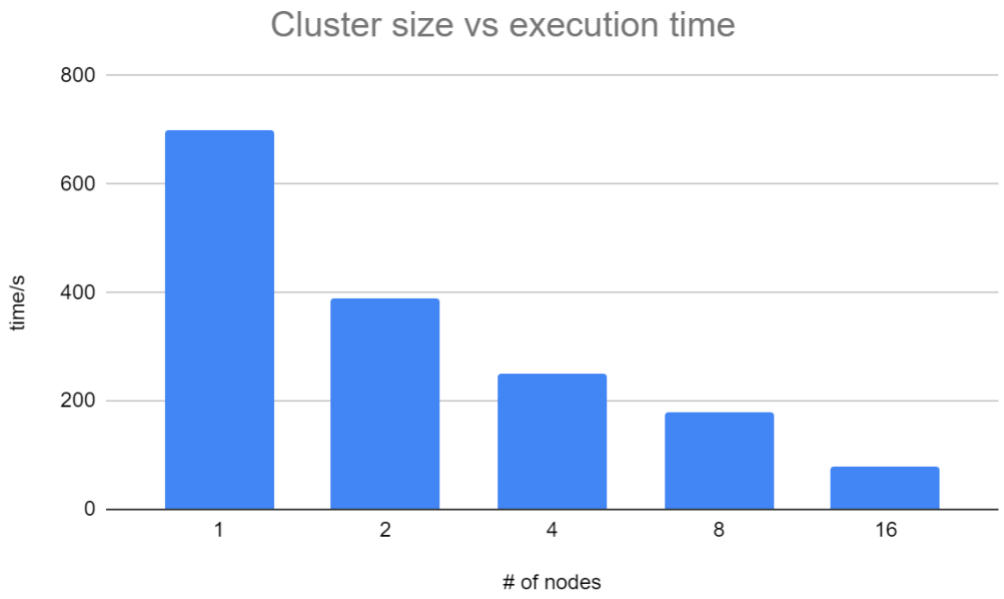
The result for modifying the number of threads for each EC2 instance is different. Since multithreading and concurrency requires more system throughput for each node, there is an upper limit beyond which the performance of the crawler node will decrease. Indeed, the crawler node will crash because of memory use if there are too many threads running. The experimental results for the number of documents per minute crawled by a EC2 node (m5.xLarge) with 10, 20, 30, 40, and 50 threads are shown below:



As we can see, there is roughly a linear relationship between the number of threads vs. the documents crawled per minute for the given EC2 instance, but there is clearly a diminishing return from 40 threads to 50 threads. Indeed, the node with 50 threads crashed several times in experiments because of the Java memory issue, and very rarely crashed with 40 threads. Thus, we decided to choose 40 threads per EC2 instance in order to balance performance and stability.

B. Indexer:

The indexer was run on the AWS Elastic MapReduce platform. We experimented with the size of the cluster by indexing 5000 documents using 1, 2, 4, 8, and 16 nodes respectively, running on m5.xlarge instances. The result can be seen from the following graph:



Through these experiments, we've found a near linear increase in speed as we increase the number of nodes. This means that our indexer MapReduce job exhibits parallelizability and scalability. This makes sense because the parallelism of the mapper is limited by the number of input files and the parallelism of the reducer is limited by the number of words, both of which are really large numbers.

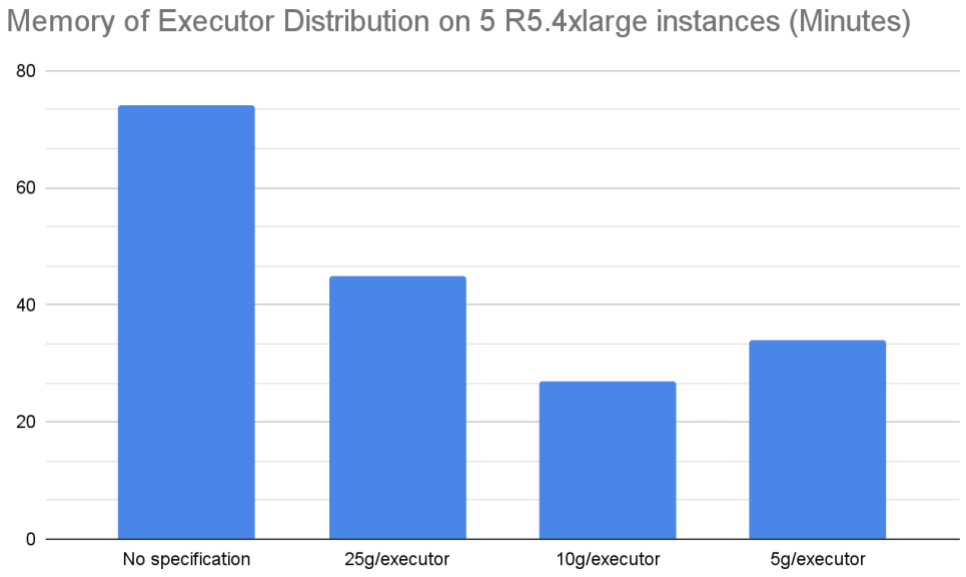
To index the 900,000 documents crawled from the web, we've decided to use 10 core nodes as a compromise between speed and cost. The job was done in 2 hours and 37 minutes.

C. PageRank:

We encountered numerous difficulties for EMR setup as well as database connections. When we were testing the algorithm locally, We couldn't connect to Aurora RDS or DynamoDB due to IAM Firewall rules. For EMR Deployment, we have tried numerous settings after job failures and realized that in order to run a step within EMR, we need to specify the main class and spark context within spark job submission.

For optimization, we recognized several bottlenecks of spark performance: EMR total memory allocated, number of executors, number of partitions, and jdbc driver usage. The initial reason that the EMR job failed was that there was not enough RAM memory allocated to store results of computation. After choosing bigger instances, the speed of computation was limited by the number of executors and number of partitions - spark didn't utilize the full potential of each instance created. We increased the number of partitions and added more instances for parallel computing. We also simplified the workflow of the algorithm and solved jdbc driver issues.

Chart below shows the speed-up for memory of executor distribution. After several heuristic tries, we landed on the conclusion that 10g per executor across nodes would be good. Our speculation toward a performance decrease of 5g/executor would be that overhead for each executor limited the performance.



D. Web Server:

The main factors we calculate final scores are the value of TF-IDF, isTitle and pagerank score. Although we normalize them by dividing maximum value, so the numerical values of index score and pagerank score are fair, the way we sum them up is still just choosing different weights and adding up. That causes two potential problems. First, it ignores the order of words in the input query. For example, when we search "all i want for christmas is you" and "you is christmas for want i all", they will receive the same results, since in our index table there is no detailed position information. Second, still using the song title as an example. According to our algorithm, it will directly drop stopwords. So in searching progress, such a song name including many common words and stop words will only remain "christmas" after normalization. Thus, after we search the song, the final results are actually results related to "christmas" but not the song title. In general, when we search a single word query, celebrities or famous companies(not

too long), the results are good. However, when we search a long query with specific meaning, the results may not be what we actually desired.

Processing speed is another challenge we must face. To achieve faster searching, we changed our searching strategy a few times. At first, we retrieve index scores from RDS, pagerank scores and Doc from dynamoDB. But obviously S3 is faster than dynamoDB, and the whole process always takes around 10 seconds. Next we tried to select docId from the index table first, then select docId's corresponding pagerank score from the pagerank table. But it performed worse: sometimes it might be faster, but when given a long input query, it may take over 15 seconds. Then we realize that we should not do so many queries. At last, we directly join the index table and pagerank table together in the term querying step, and the processing time becomes much faster. Usually it only takes less than a second, and for longer query it will not take over 3 seconds.

The choice of weight factors of index score and pagerank score is a thing we need to adjust. At first we simply add them up. But the result did not make sense in most instances. When users search for something, what they desire most are related results, but not necessarily an authoritative website. To make things heuristic, we created a list of input query including 30 words, tried different weight factors, and obtained which proportion is the best. Finally, we decided to give the index score weight 0.8 and pagerank score weight 0.2. This proportion return results make most sense.

Conclusion

In this project, we accomplished both the user interface of a search engine and its complications under the hood as a distributed system with scalability and efficiency by hosting it on AWS cloud, which allows users to do web search using keywords and view ranked results. From crawler and indexer for information retrieval to pagerank algorithm for link analysis, we've acquired knowledge toward distributed system's scalability, efficiency, and trade off among different services provided by AWS. Although the modern version of Web 2.0 is already deeply regulated, we can still find ourselves in a complicated web world where specifications and clearance are urgently needed.We also learned that for a distributed system to work smoothly, we need clear APIs from each component to be standardized.

For future reference, we could fine tune our model for better search results and add more features into the system. For the crawler, it would be better if I could implement some routes for checking the overall crawler status (i.e. documents saved, completion percentage per node, etc.), as well as a script for automating deployment and running of all crawler nodes. Another improvement we could think of is that instead of heuristically assigning weight to TF-IDF score and pagerank score, we can build a quantitative metric to measure the outcome. For the indexer, we could expand on our current hit list structure and include fancy hits and anchor hits as described in the Google paper. We could also index bigrams in addition to words so that more contextual information can be maintained. For the front-end, an improvement that can be made

on UI is that we could build our frontend on Javascript framework(e.g. React). Now we are using simple html/text, only providing information of html title and url. By using React, we can make the UI more interactive, and display more features from our crawled results.