CIS555 Final Project Report

Feng Xiang, Yezheng Li, Xinyu Ma, and Shenqi Hu
{fxiang, yezheng, xinyuma, hshenqi}@seas.upenn.edu

I. INTRODUCTION

A. Project Goals

In this project, we aim to build a functional web search engine that incorporates what we have learned in CIS555. To achieve this aim, we have the following goals in mind: a) Our system is able to crawl a large corpus of web documents. b) Our system can process large amounts of data efficiently. c) Our system can return accurate and meaningful results based on user search query. d) Our system is robust and can be deployed on the cloud as a real product.

B. High-Level Approach

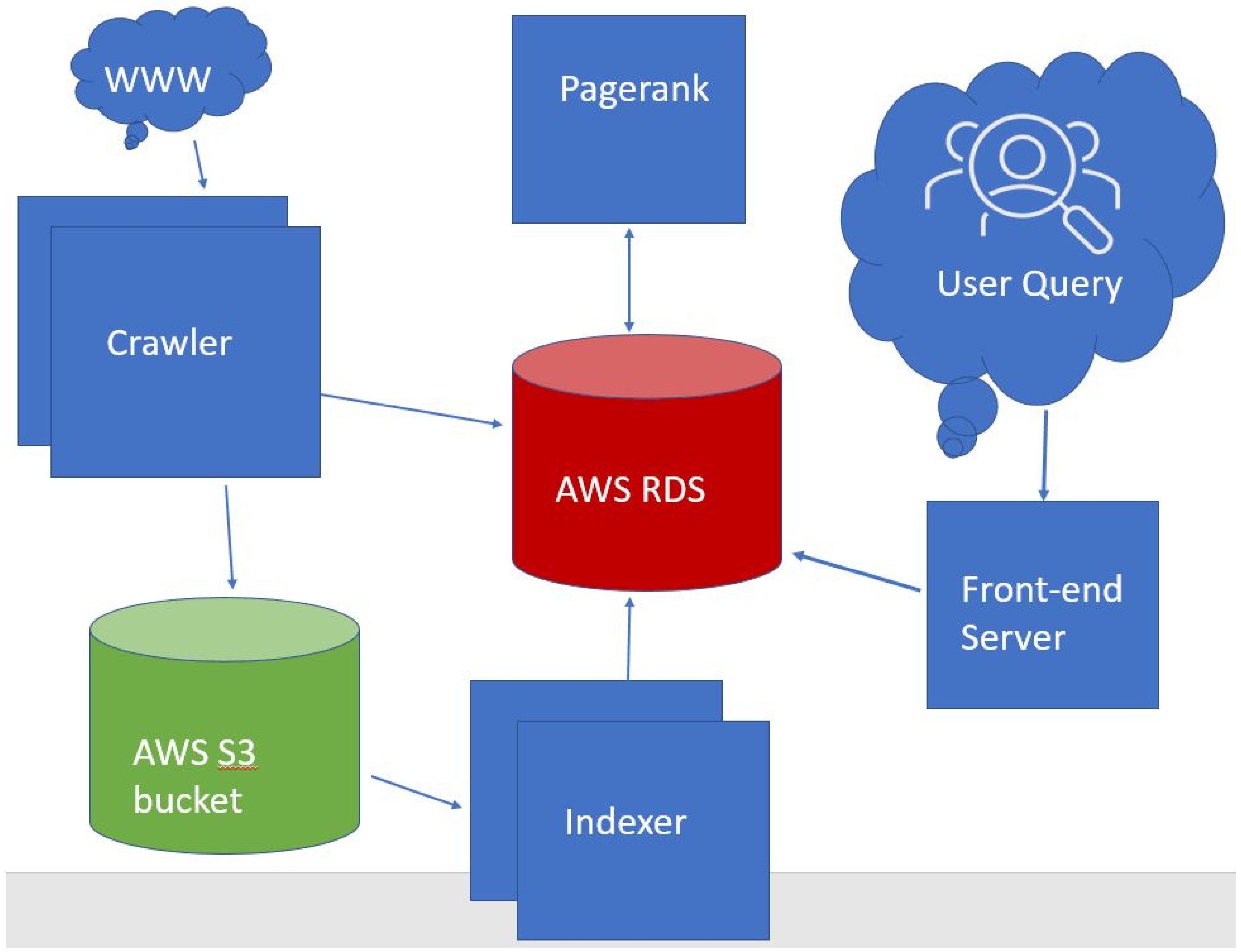
Our system consists of three stages. The first stage involves crawling web pages from the web, indexing these web pages, and extracting web links used for page rank, and storing results into databases. The second stage involves creating tf-idf values and pagerank values. The third stage involves building frontend and user interfaces, creating ranking functions and displaying search query results based on user search query.

C. Division of Labor

Feng Xiang:	Crawler, website backend/query and frontend design
Yezheng Li:	Indexer
Xinyu Ma:	Pagerank
Shenqi Hu:	Website frontend design

II. ARCHITECTURE

A. Database / Data Flow



Our project architecture can be summarized as by the diagram above.

B. Crawler

We started from the crawler implementation from HW2 as a baseline model, and on top of that we made several major modifications as follows. The final implementation runs on EC2 nodes in a fully distributed way by RPC-style communication and can crawl pages stably with a rate of 200,000 pages per hour.

Crawled pages are stored as objects in AWS S3, metadata (URLs, headers, page adjacency) is stored into RDS, runtime data (frontier queue and crawling path) is stored in local Berkeley Database. Worker deployment, start, and shutdown is fully controlled remotely by a master interface and EC2 console.

- **Stability**. For unusual header fields and robots.txt description, we converted hashmap to treemap and used lowercase contains to parse them. Catch all possible exceptions during a single URL run. Store current frontier queue to local store for every 100 URLs. Once the cluster crashes, master can send the worker list saved before to recover the identical stream routers topology which makes sure a stable field grouping policy. Task queue and thread pool are set to finite size and reject further execution requests to avoid an overwhelmed thread pool which frequently causes starvation (too many threads are accessing a concurrent resource).
- **Scalability**. All pages crawled go to S3 once got and retry until a successful put, all metadata goes to RDS, all runtime (crawling path, queue) goes to local store periodically which is synced to S3, and the task, thread, frontier queues are limited in size and are blocked once full (block spouts, filters from generating/adding new tasks/URLs). All local space needed is contained in size, and all AWS resources are configured in an auto-scaling group.
- **Efficiency**. Every worker tries to run at 100% CPU usage. URLs are evenly distributed to each worker thread for full parallelization. Frontier queues and filters are kept in local workers. When adding URLs, each of them is routed by the host field making sure that a single URL always goes to the same machine, in which way machines keep all URLs of a given host locally and can decide if a URL is seen independently. Pushing data into S3, RDS, BDB, and adding URLs to concurrent frontier queues is performed batch by batch.
- **Usability**: Created a frontend page that keeps monitoring crawling rate, number of requested and kept pages. Utilized S3 interfaces to build a simple lookup path to check the actual cached contents in S3 of any URL.
- **Performance**: Added features for crawler to read chunked data in http packet which is very commonly used. Added support for wildcard usage in robots.txt description. Any URL request that is found not allowed to send at this time is requeued. For better quality, filters are in place to discard bad URLs or pages. In our implementation, we discarded any URL that contains a special character or is longer than 100 chars, and any page whose <p> nodes content is less than 3000 chars in total or less than 300 chars in average.

C. Indexer

We parse the body for each document/url from Amazon S3 bucket in Crawler part. We only extract <title><p> and concatenate them together then do the WordCount style calculation. Results are saved on mysql RDS with following major tables:

Id2url (807914 rows)

id	hostname	port No	filepath	date_created	rawUrl	max_count
INT NOT NULL PRIMARY KEY (id)	VARCHAR(255) NOT NULL	INT	TEXT	TIMESTAMP DEFAULT now()	TEXT	INT (max_count of each WordCount appeared in that document/ url/ page)
175948	news.delta.com	80	/coronavirus-update-aircraft-fogging-b-roll	2020-05-08 06:18:36	https://news.delta.com/coronavirus-update-aircraft-fogging-b-roll	8

invertedIndexStemmed (175896214 rows)

word	urlid	count	tf	weight
VARCHAR(255) NOT NULL	INT NOT NULL	INT	FLOAT(8)	FLOAT(8)
coronavirus	175948	2	0.55	6.48338

corpus (1859126 rows)

wordStemmed	numDoc	idf_log
VARCHAR(255) NOT NULL PRIMARY KEY	INT	FLOAT(8)
coronavirus	1336	11.788
coronaviru	36966	8.46765

as well as two auxiliary tables helping especially for the case my stemmer is not good at the beginning: **invertedIndexSmall**(word, urlid, count) -- similar to **invertedIndexStemmed** but has 198618083 rows with many dirty words (only indexed till 40,000 urls then we decided to switch to **invertedIndexStemmed** and depreciate **invertedIndexSmall**); **stemmer** (raw -- PRIMARY KEY, target) with 2246656 rows**.**

- **Scalability**.We have an indexer running on EMR but only use it before indexing 10^4 urls/documents/pages. After that we use 17 indexers running 17 EC2s since running on EC2s are (1) easier to debug and control, especially when RDS get stuck; (2) sending query for multiple urls (in batch, typically, I send one query for 20-100 urls' (word,id) rows, that is about 2^15-2^17 rows).
  we wish we could debug a EMR version with (1,2) as well but we do not have enough time to improve the EMR version.
- **Efficiency**. My maximum speed is 2,000 urls per min. Running majorly with r5.xlarge, 4 CPUs runs 20-50% it power (and of course, freeable memory decreases all the way unless I reboot the database).

- **Usability**:
  Idf_log, weight (as well as tf) makes frontend easier to use output tables from Indexer;
  I calculate idf_log in **corpus**, tf, weight(product of idf_log and tf) in **invertedIndexStemmed.** These attributes (numDoc, idf_log, tf, weight are calculated via mysql query correspondingly), only one time after all (word,id, count) pairs in **InvertedIndexStemmed** are indexed.

- **Performance**: Since **InvertedIndexStemmed** is better stemmed (partially by using and being standardized table **stemmer**), **InvertedIndexStemmed** of course performs much better than **InvertedIndexSmall**. (in one point of view, we can see the performance when combining with frontend).

D. Pagerank

We implement the Pagerank algorithm, using Apache Spark computing framework. We chose Apache Spark for its speed and ease of use.
The implementation follows closely to Google did with some modification as described below.

- We model the link graph using an edge graph, where each row of the data is a (source, target) tuple. Suppose there is an edge (A, B), then it means that web page A references web page B.
- In Spark, all data are stored in resilient distributed datasets (RDD).
- We first create an adjacency list representation from the edge graph such that each row (source, [targets]) records all the target pages that a source page references to. We call these RDD links.
- We then assign unit rank to all the source web pages. We call this RDD rank.

- In each iteration, we iteratively re-compute the rank of a source page by summing over the ranks of all the other pages that point to it. This can be done quite handy in Apache Spark because of its concise syntax and function compositions.
- We test convergence after each iteration and terminate once the rank vector does not change.

Below are implementation details of the algorithm -- duplicate edges, self-loops, sink nodes/dangling links, and determination of convergence.

- Duplicate edges: we exclude any duplicate edges.
- Self-loops: Only 0.004% of our data have self-loops, which are eliminated before computing pagerank.
- Sink nodes/Dangling links: We employ the random surfer model by implying a dampening factor of 0.85 when computing the page rank. More formally, page rank is computed as

$$P(i) = (1-d) + d \sum_{(i,j)\in E} \frac{P(j)}{O(j)}$$

  d is the damping factor and *O(j)* is the number of neighbors that web page j has. This model has taken care of both issues.
- Encoding input: Spark has made it relatively easy for output to be used as input in the next iteration. Because the states are remembered across iterations.
- Testing for convergence: The algorithm converges if the component-wise difference of pagerank values for 99.5% of nodes in two iterations is less than 0.001.

E. Search Engine Ranking / User Interface

The frontend of the search engine is built using the HW2 code as a framework on top of Apache Spark, using Jsoup to extract URL titles and page text to generate a paragraph (140 word) description. There is pagination and a search bar at the top of the search page which displays the current query. The HTML files for the pages are stored locally and read upon start of the program. HTML for the search result page is generated upon receiving a query, but the CSS is static and read from the local file.

We have tried different ways to create the final ranking function. The eventual one used in deployment is sum of tf-idf score + 0.5*pagerank score. (we also try cosine similarity instead of sum of tf-idf score but may be due to our bad implmentation, cosine similarity does not provide good search results.) We give page rank score a less weight because some very important websites have many subdomains (wikipedia, for example), such that the weight for a given relevant page is diluted a bit.

IV. Experiment Analysis

A. Crawler

For the performance metric to describe a crawler implementation, we use the number of HTML pages (within 1024KiB) that can be crawled per hour. Our HW2 implementation will be used as a baseline model. Modifications and improvements upon performance we've added during our revision are mainly focused on parallelization, thread scheduling, and worker cluster, which can be described as number of active threads T, size of blocking queue of Java thread pool Q, and number of EC2 xlarge worker nodes N. We tested different combinations of above parameters and measured their crawling rates, and drawed a table to show their performances compared with each other and the baseline model.

Performances under Different Threading Conditions

Active Threads	Blocking Queue	Number of Workers	Rate / Hour
1	INFINITY	1	5,000
4	INFINITY	1	21,000
10	INFINITY	1	40,000
4	25	1	20,000
10	60	1	38,000
4	INFINITY	6	57,000
10	INFINITY	6	121,000
4	25	6	105,000
10	60	6	19,000

According to the above graph, we can tell that the number of workers and active threads greatly accelerates the crawling speed which also demonstrates that crawling is a suitable task for parallelization. On the other hand, it's clear that if we don't limit the maximum size of thread queue, there will be starvation of threads because there are too many threads who want to access a concurrent resource and what's worse new requests are adding in continuously without the remaining ones resolved, and these conditions will lead to a crash eventually. So limiting the size of the blocking queue is necessary.

B. Indexer

We have various experiments on improving performance (speed and quality) of Indexer.
(1) Speed: With r5.xlarge mysql RDS, speaking of number of urls to INSERT INTO (in each query):

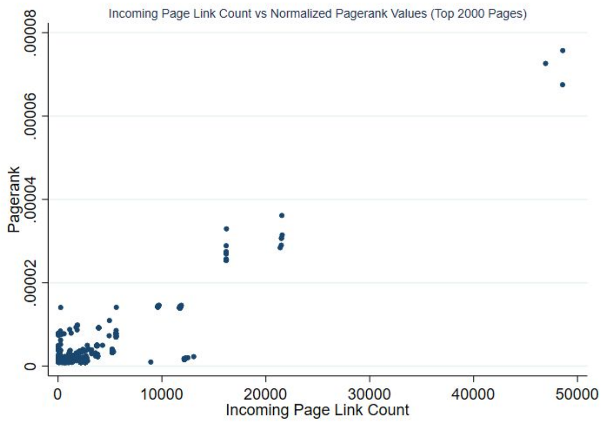
Number of urls to INSERT (for each query)	Speed (roughly number of urls per min)	reason
8	1000	RDS get stuck by too frequent insertion (remember I have 17 EC2's at most)
16	1700	
32	2000	
64	1600	
128	1000	Executing INSERT query at low frequency

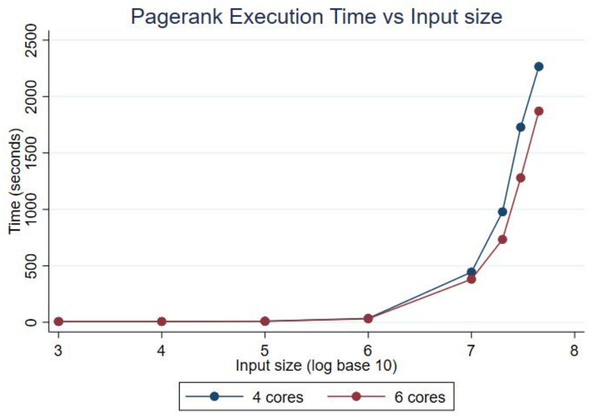
(2) Comparing with **InvertedIndexSmall**, **InvertedIndexStemmed** definitely provide better results for frontend since words are taken care of more carefully. Following queries are the ones we take care of most:
[one word]: zoom skype; noodle rice noodle; wikipedia; weather ->hurricane; University
[two words]:Donald Trump -> White House-> Joe Biden -> Bernie Sanders -> presidential election
Street drug -> remdesivir gilead coronavirus

Delta airline -> warren buffett
Saudi Arabia -> Saudi Aramco -> oil price

C. Pagerank

Overall, our pagerank algorithm implemented in Apache Spark is quite fast. In our benchmark, it converges in roughly 30 minutes and within 15 iterations. This seems to be expected, as in Google's original paper, it takes roughly 50 iterations for 300 million web pages to converge. We present below some descriptives of the pagerank result.
The figure on the left plots pagerank value against the raw incoming link counts for the web pages with top 1000 highest page rank values. There is a strong and positive correlation, suggesting that our implementation works as expected.





To understand the performances of the algorithm under different system configurations and input sizes, we do the following experiments.

- We select the input size of the link graph from {10^3, 10^4, 10^5, 10^6, 10^7, 2*10^7, 3*10^7, 4.5*10^7}.
- We select the number of processing cores from {4, 6}
- The evaluation metrics used are convergence time (second) and iterations

We don't change the memory setting for the experiment, because Spark is an in-memory data processing framework. Without enough memory, the program will crash. The machine used for the experiment is Intel i9-9900T CPU @2.10GHz with 8 cores..
For the number of iterations for convergence, smaller input size converges faster (~5 iterations) than larger ones (12~ iterations). But the convergence is very fast, confirming the exponential convergence theory. Iterations for convergence does not depend on the number of cores used, as expected.
For the time it takes to run the program, the figure above plots two lines corresponding to using 4 cores and 6 cores. As expected, 6-core has higher throughput and thus runs faster. The difference is more significant when input size grows. In our full dataset, using 6 cores takes 31.18 minutes while using 4 cores takes 37.76 minutes. The improvement isn't very large for our use case. We hypothesize that Spark is an in-memory framework, so when data can be loaded in the memory, CPU may not be very important.

D. Search Engine

Average Time to Complete a Query and Show Results (second)

# of keywords	# of results per page	# of thread fetching	completion time
1	5	1	2.98s
2	5	1	3.32s

1	10	1	6.53s
2	10	1	7.12s
1	10	8	2.31s
2	10	8	2.87s

Under other parameters fixed (80 results to be fetched from sql query, cache in session enabled and fetch page abstraction from S3 bucket enabled), we can see that the number of threads out for fetching has a large impact on the overall completion time, which means we are spending quite a lot time getting pages abstraction and titles. We may deploy our server inside the same region where our S3 bucket resides to improve this. On the other hand, the number of keywords makes little influence on the query time, so we believe our RDS is responding very fast and the way we do the query that only using SQL statements helps a lot.

V. Conclusions

In conclusion, this final project is one of the most challenging and fruitful products that we have worked on in our studies. We enjoy how this final project pieces together different components that we have learned throughout CIS555. We also think that this final project is a success. This project also teaches us how to collaborate and work in a team environment. The lessons we learned in this project and good practices that we formed will certainly help us in our future careers.

The hardest part about this project is to design scalable and robust applications that can handle the massive data load required for the project as well as learning about different APIs and how to use AWS.

If we could do the project over again, we would like to spend more time on writing design documents first to design a better schema. More concretely, we would like to try out different methods to compute pagerank values (using all links vs aggregating subdomains by domains), better ways to detect language (all pages vs only English language pages), and alternative ways to construct and tune search functions