GRS2: A Scalable Distributed Search Engine

Ransford Antwi, Shashank Prasad, Garvit Khandelwal and Shruti Sinha

G13, University of Pennsylvania, CIS555 Professor: Andreas Haeberlen

May 11, 2020

The purpose of this project is to build a fully scalable and distributed search engine. The major components of the search engine include Crawler, Indexer/TF-IDF Retrieval Engine, PageRank, Search Engine and User Interface

1 Introduction

1.1 Approach

We designed and implemented a distributed search engine adopting Mercator style crawler to implement efficient crawling, used Stormlite framework for indexing, Apache Hadoop framework for PageRank and a clean, quick and high quality user interface for the search engine alongwith some interesting extra features.

1.2 Division of Labour

- **Garvit Khandelwal** : Indexer Stormlite(HW3) and EMR
- **Ransford Antwi** : Crawler, Crawler EC
- **Shashank Prasad** : Search Engine, User Interface, EC API-Frontend, EC Cache
- **Shruti Sinha** : PageRank, EC API Backend, EMR Indexer

1.3 Project Timeline

- 04/24 - Basic Interfaces
- 04/29 - Basic Crawler/Indexer/PageRank
- 05/06 - Indexer and PageRank integration with Search Engine
- 05/10 - Search engine rank tuning and Extra credit

2 Architecture

2.1 High level System Overview

The overall system architecture is shown in Figure 1

2.2 Crawler

The design of the crawler borrows heavily from the Mercator design. The crawler was designed for maximum efficiency in terms of throughput, whilst also respecting crawl delays and other admin specified behaviour within the robots.txt of websites. Building a distributed crawler comes with certain challenges that are not present on a single node crawler. To make the crawler as robust as possible, we periodically save the majority of the crawler state on disk so that in the event that it does crash, it can easily resume from where it left off. This feature helped save a lot of time. The URLFrontier was designed to be as polite as possible without sacrificing throughput. We implemented a priority based frontier with 3 different priority levels. The higher the priority of a host name, the more consecutive URLS from that host name can be crawled. The URLFrontier buffers at most a 1000 URLS in memory and the rest are stored on disk. The crawler was built using the StormLite framework which is a miniature version of Apache Storm for stream processing. The architecture for a single worker node is shown in Figure 2. We used BerkeleyDB to handle the database transactions.
The overall crawling process was as follows :

1. Dequeue URL: The URLSpout dequeues a URL from the frontier and emits it to the Crawler Bolt. If the frontier is empty, it refills with a 1000 URLS from disk

2. Crawler Bolt: The crawler bolt first checks the robots.txt associated with the given host. If we are allowed to crawl the given url, we then check the crawl delay and compare with the last time this host was crawled. If the delay has not expired yet, the URL is added back to the queue, else we proceed. Next step is to check whether we already have a document pertaining to this url in our DB. If we do, we send a head request with an If-Modified-Since Header to determine whether the document has changed else we send a regular HEAD request. If the document has not changed we forward it to the document parser bolt, else we send a GET request to download the page and forward the url and document to the document parser bolt.

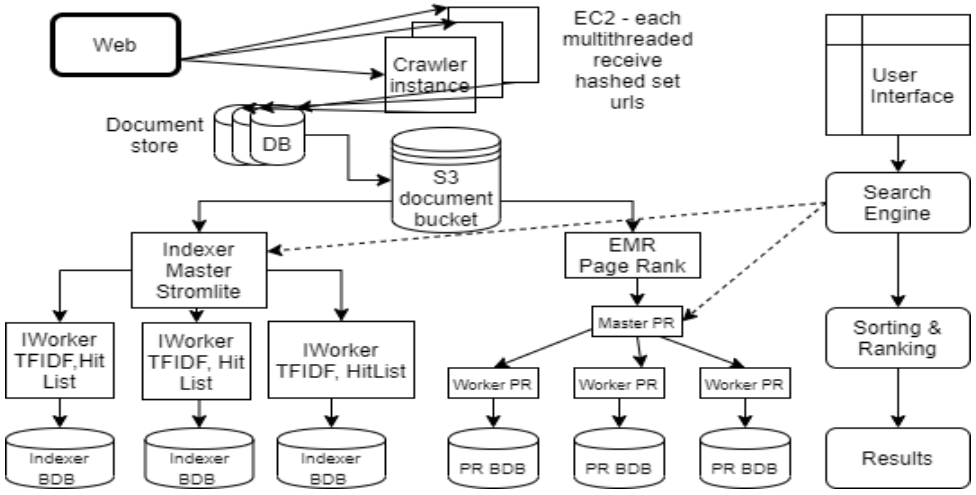

Figure 1: *System Architecture*

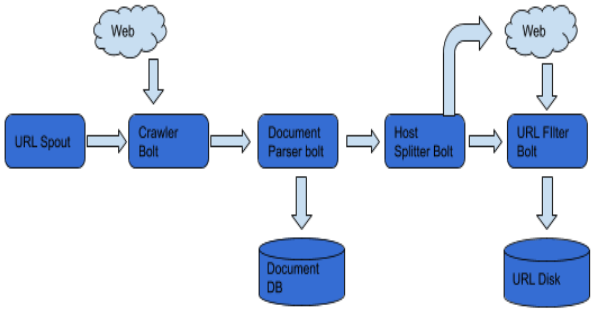

Figure 2: *Crawler Architecture*

3. Document Parser Bolt: This extracts links within each html document as well as performs the content seen test via MD5 hashing of each html page. It the forwards all the links extracted to the host splitter bolt.

4. Host Splitter Bolt: Forwards URLs to the respective worker nodes based on a hash of the hostname modulo the number of workers. It forwards the URLs to the workers' URLFilter Bolt.

4. URL Filter Bolt Bolt: Filters the URLs based on depth, length and other user specified filters such as banned hosts. URLS that pass the filter are then written to the URL disk.

2.2.1 Distributed Crawler

For the distributed component, we implemented a master worker architecture similar to the one in Mercator. The space of host names is partitioned across the worker nodes. The master node is responsible for sending the worker table to each worker node, and the worker node forwards URLs that are meant to be handled by a different node to that worker node's host splitter bolt. The host splitter bolt then forwards URLS to the URLFilter. We also implemented a content seen test by hashing the content of documents passed into the document filter bolt, this is to ensure we don't download the same content multiple times under different URLs. Another extra feature we implemented was the dynamic addition of worker nodes. For a worker node to start, it sends a status update to the master node which then sends a send job signal to the worker node. When the master node receives a worker status update from a new node, it recalculates the worker table and sends the updated worker table to all worker nodes. The workers use the worker table to partition the space of host names and forwards host names to relevant workers via the host splitter bolt. Workers send a status update to the master node every 10 seconds. This is all shown on the master screen, an example is shown in Figure 6

2.3 Indexer

The indexer comprises of Map Reduce jobs which were implemented on **EMR** and on HW3 framework using **Stormlite(Extra Credit)**. For both the implementations, the mapper and reducer had the same job. The input data comes from the crawled data saved in S3 bucket and the indexing was divided in two sequential jobs (Job1 and Job2) to calculate the inverted index, TF/IDF scores and hit list which contain the normalized position of the word in the document.

TF and Hit List Calculation: The key to Job1 is URL and the value is the HTML content of the page. The mapper then parses the content and removes the unwanted tags to just get the raw body and title out of the HTML. The words were then filtered to remove **stop words** for English language and then were stemmed using **Porter Stemmer**. The mapping also accounted for removal of any special characters except characters and numbers. Using the **lexicon** of English words, the **term frequency** of that word(tf) in that document and **hist list**(hit_list) comprising of normalized term location was calculated by dividing the index(i) at

Status Page

Active workers :

Worker Index	Worker Address	Status	Links Crawled	Links Downloaded	Avg Crawl Rate (links/sec)	Max Crawl Rate (links/sec)	Crawl Duration (minutes)	200 Responses	3xx Responses	404 Responses	Other Http Responses	ShutDown
2	172.31.20.119:8020	Crawling	680	794	13.0	27.2	1.0	191	37	5	447	Shutdown
0	172.31.28.210:8050	Crawling	139	108	1.0	4.0	1.0	138	0	1	0	Shutdown
3	172.31.26.186:8040	Crawling	103	93	1.0	10.2	1.0	102	1	0	0	Shutdown
1	172.31.18.227:8030	Crawling	1366	576	9.0	100.6	1.0	274	41	1	1049	Shutdown

Figure 3: *crawler master node screen*

which the term appears and total number of words in that document(W).The output for Job1 reducer had word, its URL, TF score and the normalized term location.

IDF and Score Calculation: The mapper of Job2 split the word from the intermediate result from Job1 and passed the word as the key and its URL, TF and hit list score to the reducer. The reducer then accumulated all the URLs(n) per word and calculated **inverse document frequency**(idf) using total number of URLs(N). Using the TF(tf) and IDF(idf) scores, the final reducer emitted **inverted index** for each word and the list of URLs it occurred in and its score(tf_idf) and hit list in form of the data structure: <Word, <URL, TF/IDF score, HitList score».

$$hitlist = i/W \tag{1}$$

$$tf = 0.5 + 0.5 * \frac{f_t, d}{max\{f_{t'd} : t' \in d\}} \tag{2}$$

$$idf = log\frac{N}{n} \tag{3}$$

$$score = tf\_idf = tf * idf \tag{4}$$

The results were stored differently for EMR and Stormlite.

- **EMR**: The input to indexer comes from S3 as URL and its content and the final results were also stored in S3 bucket on AWS. The results were then pulled from S3 using a master-workers framework where the master reads the data from S3 and calculates hash for each word in lexicon and spilt it across each worker depending on number of workers attached to master keeping the solution scalable. The workers then store the final results in Berkeley DB and was then made available to search engine.
- **Stormlite**: The Spout in Stormlite framework gets data from S3 bucket and emit one URL and its contents using shuffle framework to mappers and then to reducers bolts. Then the reducers finally use field based grouping to calculate hash based on number of workers in the system and route the final data to final bolt which saves the data in local Berkeley DB. Once the indexer finishes, the data is available for search engine to query.

2.4 PageRank

The pagerank module is invoked after we have a complete crawler corpus in S3. We used the Google paper on PageRank as our base reference. It has been implemented using the Hadoop Mapreduce framework and is run on Amazon's Elastic Map Reduce to reduce execution time and boost efficiency of our program with respect to the HW3 framework. It consists of 6 sequential jobs which is run by a driver class.

1. Extract Link MapReduce: This job reads the corpus from the S3 crawler bucket and creates a mapping for each page to its outlinks which basically created a web graph for that page.

2. Dangling Link MapReduce: For each page it identifies those outlinks that are dangling i.e uncrawled pages without any outlinks of their own.

3. PageRank Initial MapReduce: This assigns an initial rank of 1 for each page.

4. PageRank Algorithm MapReduce: It is an iterative job which performs the actual calculation of pageranks using the formula described in the PageRank paper by Brin and Page which involves a damping factor d of 0.85 The formula used to calculate PageRank of a page A which has pages T1, T2..Tn which point to it is given as follows:

$$PR(A) = (1-d) + d(\frac{PR(T1)}{C(T1)} + ... + \frac{PR(Tn)}{C(Tn)})$$

The results of this algorithm are then fed back to this job and runs iteratively.

5. Normalise MapReduce: In order to account for sinks and dangling links, who cause distortions with respect to the expected ranks, we normalise PageRank values after each iteration . To normalise we multiply each page's rank with a factor, determined by dividing the initial page rank sum before the iteration with the new sum of pagerank values after the iteration. This allows us to redistribute the ranks in the same proportion as before with fewer distortions.

6. Finalise MapReduce: The last job puts the final clean ranks as url vs rank from the last iteration into S3.

2.4.1 Design Decisions

Our pagerank web graph would consist of urls i.e per page as an individual vertex as suggested in the Google paper instead of having a domain per vertex. In addition, the pagerank module is protocol independent i.e two pages having the same url but different protocols for eg. http://abc.com and https://abc.com would have a single entry/vertex in our pagerank web graph. We also chose not to get rid of self links, since we wanted to avoid any changes to the original web graph and therefore used normalization instead.

We used the HW3 framework with SHA-1 hashing to distribute the pageranks for the corpus stored in S3 to multiple worker servers local Berkeley DB instances. We decided to use local Berkeley DB instances to save pageranks and indexed data over RDS and DynamoDB because it's pros outweighed the others. DynamoDB could have proved to be expensive and would not have been easy to query while RDS would not have been scalable and would have required distribution from our end.

2.5 Search Engine

- **Back-End** The Search engine back-end is responsible for communicating with both the indexer and page rank and use their outputs to compute the top URLs relevant to the query. In our architecture, the indexer and page rank expose a spark java server with routes registered to query them for data related to the query. Once the user enters the search query, it is filtered to remove all the stop words and the filtered words are sent to the indexer. The indexer sends a back a structure with a collection of urls for each word along with their TF/IDF and position scores. From the indexer payload, we extract all the urls and sent them to the pagerank to get the page rank score for each URL. Once the respective payloads are obtained a local cluster with a url spout, a scorer bolt and enqueue bolt is started. The url spout emits urls to the scorer bolt which computes the score for each url and emits the (url,score) value pair to the enqueue bolt. The enqueue bolt enqueue to a priority queue based on the score. The priority queue is used to get the results to the user interface.
- **Front-End** The front-end exposes a spark java server for browsers to supply the query string. Once the query string is received it issues this to the back-end and uses the priority queue generated to generate construct a HTML page back to the user

3 Ranking

The ranking function requires the following inputs for a search query

- TF/IDF scores for word in the query string for each url
- Position score/ Hit List for word in query string for each url
- Page Rank Score for each url
- Word frequency in the query string

Algorithm 1: Ranking algorithm

```
Result: Computed scores for Top 100 URLs
Get TF/IDF scores from the indexer
Get HitList scores from the indexer
Get PageRank scores from the index
Calculate the word frequency in query String
for url in URLs returned from indexer do
    scoreUrl = 0.0;
    for word in query String do
        scoreUrl+= (TF/IDF score for word in
         url)*(frequency of word in
         queryString);
        scoreUrl+= 0.5*(Hit List score for word
         in url);
    end
    scoreUrl *= (pageRank for url);
    Add scoreUrl to results priority Queue
end
```

4 Extra-Credit

- We integrated search results for businesses using Yelp API and also included the current weather details for different locations using the openweather API. These results were triggered based on the search query entered by the user.
- We included pagination to our search results in the UI along with a cache feature in the search engine which allowed us to retrieve popular query results immediately if the query was already searched.
- Content Seen Test within the crawler such that it does not download the same content multiple times.
- Dynamic Addition of Worker nodes in the Crawler
- Incremental Crawling supported
- Indexer was implemented using HW3 framework as well as EMR.

5 Evaluation

5.1 Crawler

The final production ran was deployed on a total of 7 amazon EC2 instances. Six of which were worker

nodes and one master node. The worker nodes were of type "xlarge" which consisted of 16GB of memory. Each worker node had an additional EBS volume attached of 100GB each. The master was of type "medium" which consisted of 4GB of memory and had no extra storage attached. Despite the worker nodes being of the same type, the throughput varied across the nodes. The best performing nodes had an average throughput of **19 pages crawled per second** and reached a max throughput of **114 pages per second**. The worst performing node had an average throughput of **4 pages per second** and recorded a max throughput of **24** pages per second. The average throughput was calculated as a moving average for the entire duration of the crawl and the max throughput was the highest recorded throughput within every status update. (Status updates occurred every **10** seconds). Total average throughput versus number of nodes is plotted in Figure 4.

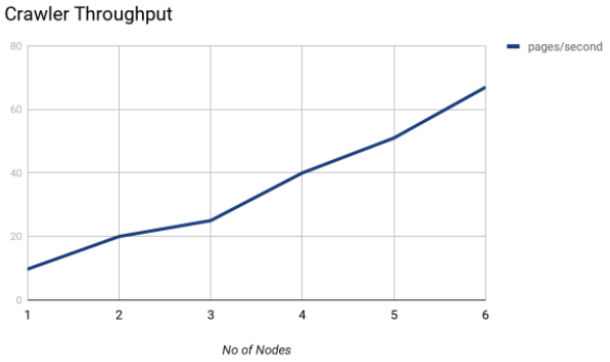

Figure 4: *Throughput vs number of nodes*

The final run downloaded a total of **1.07 million** webpages over a period of 620 minutes. Each of the 6 nodes was set to download a maximum of 300 000 pages, however, one node crashed immediately leaving us with 5 nodes and another ran into a crawler trap so downloaded URLs at a slow rate. Luckily, because of implementing incremental crawling, we were able to restart some of the nodes without losing much progress. A summary of the HTTP responses encountered is shown in figure 5.

5.2 Indexer

The performance evaluation for indexer was carried out in two different implementations.

Corpus Size	Framework	Time
50k	EMR	30 mins
50k	Stormlite	5 hours 30 mins

Table 1: *Time comparison between EMR and Stormlite*

Based on the table above, we decided to switch to EMR since the time taken to index using HW3 was too long. While using EMR on a large corpus size, we

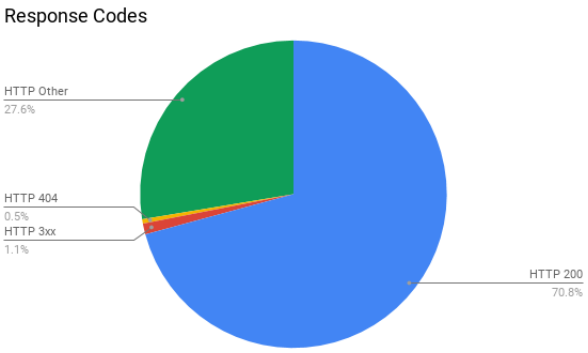

Figure 5: *HTTP Response Codes*

encountered memory issues due to the intermediate results being stored on local EC2 instances deployed in EMR. In order to resolve this issue and tune our map reduce jobs, we increased the size of our JVM memory for map and reduce of our instances from -Xmx2458m, -Xmx4916m to -Xmx2867m and -Xmx5734m respectively. Additional EBS volumes of 100gb was also added to the 64gb default volume for each instance. We also found out that Porter Stemmer was computational very expensive to run and that became the bottleneck of the system in map phase. The above changes greatly improved our performance with a runtime of 25 mins on a corpus of 70k without stemming which resulted in 500,000 different tokens.

Stormlite: There were several factors to experiment while running the Indexer on Stormlite framework. The variation in corpus size, thread pool size in ExecutorService, number of Map, Reduce Bolts and number of Workers were observed. The time observed are as follows:

Corpus Size	Thread Pool	M/R Bolts	Workers	Time
200	1	5	2	5 mins
2k	1	10	4	40 mins
2k	10	10	5	20 mins
20k	10	10	5	3 hours
50k	20	10	10	5 hr 30 mins

Table 2: *Time comparison in Stormlite*

The time taken for Stormite was more because of couple fof things. A lot of packets containing each word in each URL with intermediate scores was sent over the network to each worker. Porter Stemmer was implemented and storing, retrieving so many results from BDB turned out to be bottleneck of the system.

5.3 PageRank

We observed that with changing corpus size, the time taken to the run the entire module differed for the number of instances used at a time.

The times observed were as follows:

Corpus	Number of instances (type: m5x.large)	Logged time
2.5k	3	5min
30k	3	12min
140k	2	43min
140k	3	24min
140k	4	19min
220k	4	30min
300k	4	42min
1million	4	2hr 53min

Figure 6: *pagerank runtimes*

It was observed that with increasing corpus size and having a constant number of instance nodes, the time taken to run the pagerank emr was increased almost linearly. Moreover, with increasing instance types from 2-4, the time taken also decreased as shown in the table above. We could not run on more than 4 instances (m5.xlarge) due to restrictions/limitations of the AWS educate account. In addition, for a smaller corpus like 2k, convergence of pagerank values was seen in 4-5 iterations, however for a larger corpus for 140k or more, we observed convergence around 15-18 iterations and therefore decided to run a total of 18 iterations to get final converged pagerank results for 1 million.

5.3.1 Experiments

We optimized our query time from BerkeleyDB from each of the workers by restructuring our logic to get pagerank results. Earlier, the master on receiving a request for pagerank values for a list of urls from the search engine, would iterate through the list and ask the worker(based on hashing) responsible for its value and combine these results and send a map back to the search engine. However we optimized it, such that the master on receiving a request, identifies and sends an object per worker, which consists of the urls that a certain worker would hold based on hashing. Each worker responds with an object consisting of pagerank results for the url list, which is then combined and sorted by the master. The master finally sends the top 100 urls back to the search engine. This decreased the number of connections/requests made per worker which was sequential earlier and was slowing down our response time for the query results in the search engine.

5.4 Search Engine

The average search time obtained for a query is **1.8 seconds** (averaged over 25 queries)

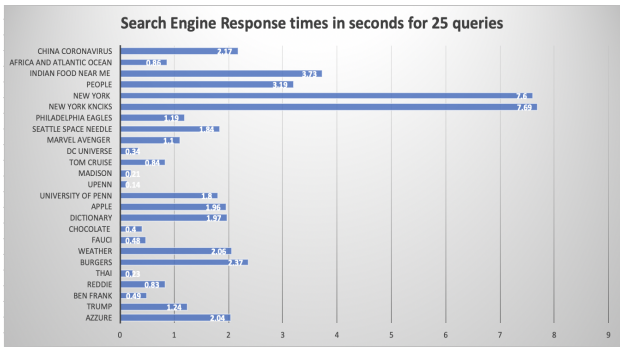

Figure 7: *Search times for 25 queries*

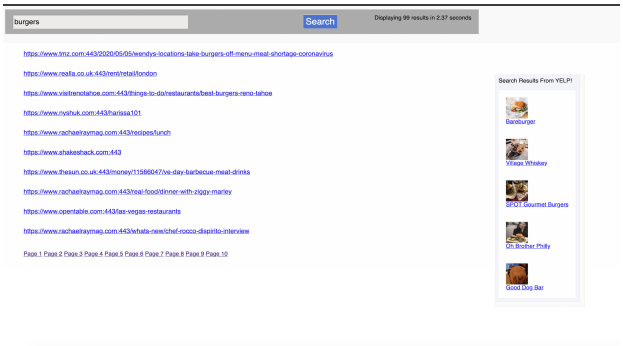

Figure 8: *Search results with integrated results yelp business API*

6 Conclusion and Lessons Learnt

Through this project, we implemented the basic moving parts of a full distributed search engine. We observed that fully distributing each component increases the throughput of data traffic / data crawled and also reduces the time required to answer the search query to the user. The results were satisfactory and there is a lot of room for improvement. Some of the areas for improvement are:

- Index more data
- Employ search engine optimization techniques to increase the quality of results
- Research techniques to understand the context of the query and provide relevant results

This was an awesome project and challenged us in many ways but we're grateful for the opportunity to have been able to carry it out.

7 References

NAJORK, M. AND HEYDON, A. 2001. On high-performance Web crawling. SR, Tech A68 Compaq Research Center, Palo Alto, CA.

S. Brin and L. Page. "The Anatomy of a LargeScale Hypertextual Search Engine". Stanford University. Computer Science Department, 1998