CIS 555 Team Project Final Report

https://bitbucket.org/dyunwen/ldyd-cis555project
Tianxiang Dong: dtianx@seas.upenn.edu
Yifan Li: lyifan@seas.upenn.edu
Hanyu Yang: hanyu@seas.upenn.edu
Yunwen Deng: dyunwen@seas.upenn.edu

I. Introduction:

1. Project goals:

We aim to create a google-style search engine named "Quantum Engine" that consists of four major components: a Web-Crawler, an Indexer, a PageRank Engine, and the final Search Engine and Web-Interface. The overall ideal application should be scalable with distributed storage and perform good searching results.

2. High-Level Approach:

The crawler adopts Mercator style and is designed to be distributed and multithreaded. The indexer indexes monogram, bigram and trigram with extra information like word position in order to help improve searching results. Both indexing and page-ranking use MapReduce but with different implementation frameworks: indexer runs Apache Hadoop MR while page-rank engine uses Apache Spark MR. The search engine also uses Apache Spark to run MapReduce in order to achieve less searching time cost.

3. Milestones check in:

Milestone 1- Nov. 29th Tuesday

1) infrastructure configurations and setup
2) finalized crawler with at least resulting 5,000 web pages for testing

Milestone 2 - Dec. 6th Tuesday

1) Fully crawled raw dataset (> 200,000 quality web pages)
2) Lexicon, inverted index and necessary data structure for indexing
3) Finalized indexer with tested score results of all crawled pages
4) Fully functional Search Engine with a dummy User Interface

Milestone 3 – Dec. 20th

1) Finalized PageRank components integrated with all previous components
2) Improved User Interface

4. Division of Labor:

1) Crawler: Tianxiang Dong
2) Indexer: Hanyu Yang
3) PageRank: Yifan Li
4) Search Engine & Web Interface: Yunwen Deng

II. Implementation and Evaluation

Crawler Implementation and Evaluation

The crawler is designed based on the Mercator-style. The framework of Storm from Homework 3 has been applied. It's a scalable, multithreaded and state-restorable crawler that is written completely in Java. Each crawler instance can be run simultaneously on multiple machines with the only interaction in AWS S3 database. Each instance has their own queue management system (URL Frontier Queue) and the element storing extracted links information in Berkeley DB.

1. Architecture

For the implementation of crawler, we have a master server to supervise the progress of each node, as well as several separate working nodes. The storm framework of homework 2 and the distributed modification of

homework 3 are combined. All working nodes share the same bucket in Amazon S3 for crawled pages' metadata and contents. To support restorability after stopping the crawling, URL frontier Queue and visited URL information are stored separated in local BerkeleyDB. The Field Based streaming of URLs in storm are decided by hostname of urls, in case concurrently requesting more than one document per hostname. We used DigestUtils to hashing a given URL as its key then upload to S3.

2. Evaluation

As our final crawling, we were using 5 crawlers and 1 master in EC2. Each worker has a thread pool of fixed size 50. Our final crawling crawled more than 300,000 pages.

Number of Threads	1	10	20	30	**50**	80	100
Pages Crawled/second	1	2	4	6	**11**	10	11

Number of Workers	1	2	3	4	**5**	7	8
Pages Crawled/second	11	21	31	40	**51**	65	73

Two experiments about the crawler are performed with the results shown above. The experiment of threads is based on a single EC2 m4.xlarge crawler worker instance. The starting root of the crawling is from http://www.cnn.com/. By increasing threads on a single node, the overall performance is increasing until the thread pool has more than 50 fixed capacity.

The experiment of workers is based on each node having 50 fixed threads. The experiment is set up across several EC2 m4.xlarge instances. Apparently, the crawling speed depends on the the number of workers. As shown in the table, the relationship between number of workers/instances and number of pages crawled per second is linear. Therefore, we should choose as many nodes as possible. But since it costs money, we finally decided to created 5 instances as workers for crawling, which turned out to be enough for more than 300,000 pages in 3 hours.

PageRank Algorithm Implementation and Evaluation

1. Architecture

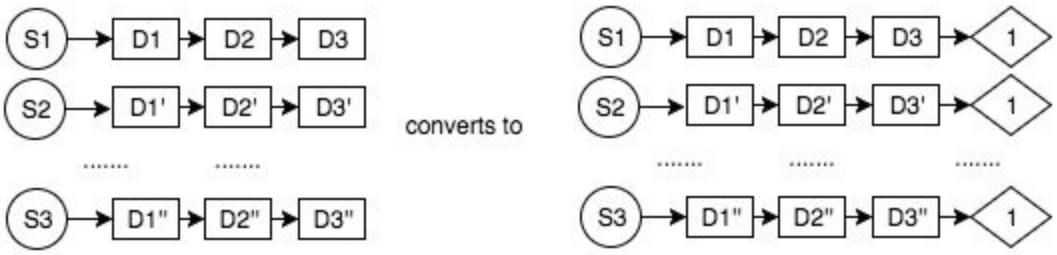
For our implementation, the PageRank of a given URL is determined by the several factors, including a pre-defined decay factor of the entire system, the total number of nodes in the "graph of the internet", and the links between each pair of URLs in the given system. At the end of each iteration, the total rank values that go to all sinks and dangling links will be collected and re-injected back into the system upon the start of next iteration. Finally, the **Apache Spark** Map-Reduce Framework is used to reduce the execution time and boost efficiency of the program.

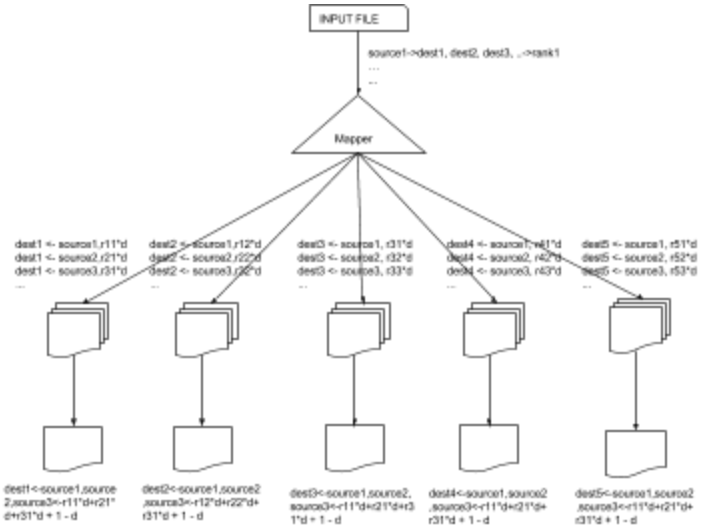
2. Framework Selection

The first attempt to implement the algorithm was made with Stormlite. However, it turns out the Stormlite package might have come with some inherent problems that cannot be fixed soon enough. The other possible approach with Hadoop fails due to extreme time consumption when dataset is scaled up. For instance, a later single-iteration test-run on a large data-set with 45k entries took about 1 hour to finish. So we finally adopts **Apache Spark** to finish page-rank job. It only took Spark only about 20 minutes to build the entire graph and then 10 minutes each to run every iteration.

3. I/O & The Algorithm

The PageRank program consists of two steps. The first step is to build a "graph of the internet" from raw links data. Basically, the map-reduce job takes a file of the form:





Where the circles represent source nodes, rectangles represent destination nodes, and diamonds represent the rank of each corresponding source node.

The Mapper reads each line and breaks it into multiple K-V pairs. For each Si, the mapper emits S1 -> [D1, D2, …, Dn]. Also, for each of D1, D2, …, Dn, the mapper emits Di -> []. The purpose of doing this is that in the reducing phase when all the NODE -> LIST pairs are reduced by key, the nodes with zero out-degrees will be reduced to having to no outbound links and thus can be easily recognized in the graph. Later we will use this information to compute the sum of ranks of all the sink/dangling links in the graph. The file in Dig2 will used as the starting point of iterations for the PageRank algorithm.

A single iteration of the PageRank MapReduce Job can be best illustrated by the graph on the left. The final results after all the reducer jobs are completed will the be uploaded to S3 buckets and local BDB stores on each worker server.

4. Evaluation

To reach convergence, we run 15 iterations for pagerank calculations. Some sample high score pages like http://www.worldnow.com (78.77), http://www.language-archives.org/(254.52), http://archives.cnn.com/TRANSCRIPTS(164.16), https://wordpress.com/(87.99) and https://twitter.com/share (75.9056504488218).

Indexer Implementation

Overview:

Indexer engine is implemented in multithreaded Hadoop MapReduce framework, processing data source from S3 and metadata from crawler machine.

1. Data source:

- URL:

After crawler finished, 5 crawler machine will generate a crawled hashed Url files for each, which is input stream file for indexer engine. Totally, we processed 300,000+ pages from crawler, generated 1.7M words from that, about 6.2 G raw data.

- S3 file content:

The web contents are stored in S3 bucket, using SHA-1 hashed URL as key. So by reading hashed URL from crawled file, we can approximately process 20 urls per second per machine for mapper step.

2. Parsing:

- Parsing approach:

For parsing step, after download file content from S3, we use stanford NLP library to get normalized word from file content. Instead of using snowball stemmer engine, we use stanford NLP lemmatizer API to get better search performance, which generated more words. For instance, “am”, “are”, “is” are converted to “be”, “car”, “cars”, “car's”, “cars'” ar converted to “car”.

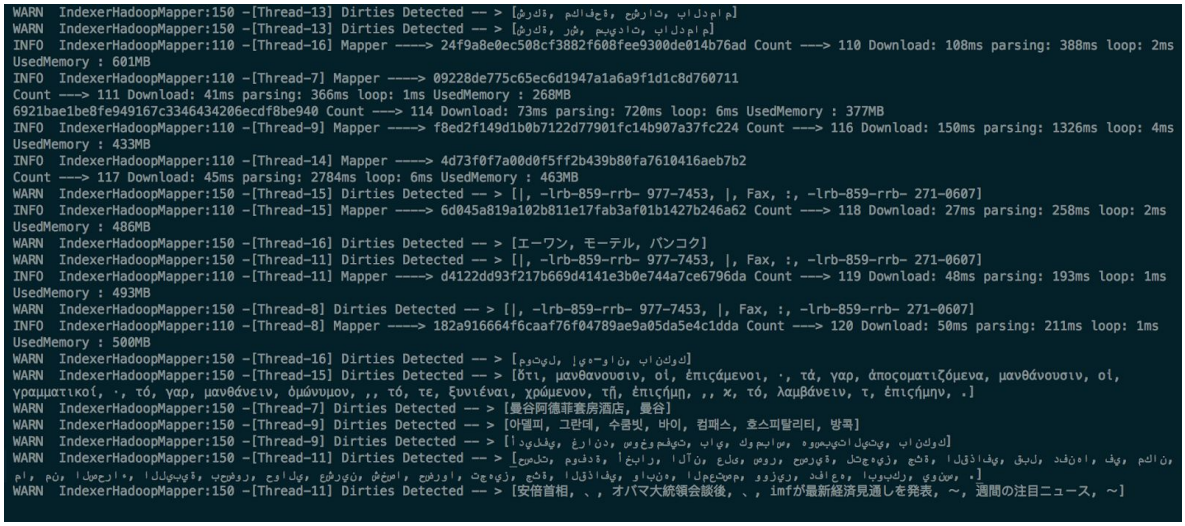
- Parsing rules

We use regex to extract the word. Here is the rule we use to determine if we find a word. For bigram and trigram part, we use in memory dictionary to store some high frequency words.

1. Any word must start from letter or number.
2. Any word can contains letter, number, &, -, @
3. Any word must end with letter or number.
4. Any word is not in stop words list, we use stop words from http://courses.cs.washington.edu/courses/cse573/12sp/A2/stopword.list
5. Bigram and trigram words are used in http://www.ngrams.info/download_coca.asp.

- Problem:

Stanford NLP library is quite efficient in processing English file content, but when we use that to lemmatize word from other language, it will take 100x time we used in normal file. To improve performance, we design our own detection mechanism, which examines the pages whose parsing time is over 50ms, extract words from them, if more than half of words are not satisfied with our word pattern, we marked them as alien files. If they are marked more than third time, we jump that files. By doing this, we improve parsing efficiency. The log below shows the alien files we detected.



3. MapReduce:

- Mapper

We start mapper process for each file, use a frequency map to store every word’s frequency in that file. And after parsing step, we get a total count in that map, so that we can calculate TF value for each word. In this step, we don’t multiply TF by IDF because we want to store more information for each word, and emit as (word, url, tf) tuple.

- Reducer

In reducer process, we simply reduce by word. The hadoop framework handled the IO output for us, so in the final step, we get results file approximately 1.2G for each machine.

Search Engine and User Interface Implementation and Evaluation

1. Architecture outlook:

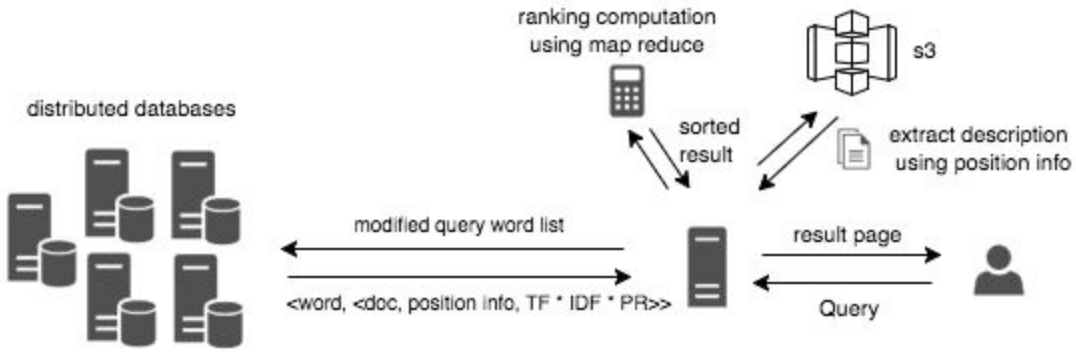
The search engine server is built over Jetty. All distributed storage worker servers, which communicate directly to the search engine server, are supported by Spark Framework. In order to achieve fast searching performance, we also apply Apache Spark to run mapreduce job for constructing proper and sorted search result list. All information exchange, like user query content, hit document lists, index and page-rank scores, is handled through HTTP requests.

This particular design allows scale-up, fast and reliable information retrieval and less search time consumption. More importantly, it realizes fault tolerance and still functions normally if one or two distributed storage machines are down due to possible power outage, poor TCP/IP communication, etc.

2. Interface usage details

The user interface adopts basic web search application services. We also enable **autocompletion** to improve typing convenience. The **autocompletion** is implemented by constructing trie using 10000 most commonly-used english words using Javascript.

3. Implementation details



Searching process illustration

The search server handles all user traffic for querying and returning search results. Inside search engine server, an extra crucial thread created in query servlet is in charge of all communication with distributed storage worker servers. Each worker server, located in different machine, has its own running server supported by Spark Framework and each has Berkeley Database with partitioned index and page-rank storage locally. Once a request arrives, the search engine first parses and reconstructs query words after lemmatization. Then, it broadcasts the query to all available worker servers using HTTP POST. Each worker server decides whether a query word is within its storage, proceed or not to fetch the word document list stored in its local Berkeley Database and then respond to the search engine.

After all responses from worker servers to the are collected, the search engine conducts a mapreduce job using Apache Spark to find out the proper intersection among all documents, and compute scores composed by index and page-rank scores, then sort based on number of words that one document contains and also the ranking scores. Then the result servlet fetches documents from S3, constructs description, then finally generates the result page to the user.

The ranking function we developed entails server factors below:

- original indexing and page-rank scores;

- domain name length of the url and the length of the url
- Position info: increment weight when a query word appears in the document's title

In summary, the function

factor * (∑ TF * IDF) * PageRankScore / (length of url or domain name)

factor: a compound factor that takes consideration of term appearance in url, metadata, and title

The performance of the above function is acceptable and satisfying. Detailed search result evaluation will be discussed in next step.

Fault-Tolerance database design

In order to improve robustness and in case some of nodes crashing down, our distributed nodes do not only include its own data but also the node right ahead of it, like the node 3 would include both data in node 2 and 3, and node 1 would have data from its own and node 5. Every time the search engine wants to retrieve data from nodes, it sends the living nodes list along with the query. Once one node detects that its former node is offline, it will take responsibility to return the data of that dead node.

In our system, we can support at most two of total five nodes offline and the search engine just works fine, as long as these two nodes are not continuous. In other words, only one single node offline will have no impact on our search results. The only difference is that since one node may retrieve more data than it used to do, the searching time might be a little bit longer than usual.

4. Evaluation

The search engine performance is examined in 3 aspects: results, time and query input.

a. *Results*: the overall search results are good. According to the crawled content and common domain ranking provided by Google, our returned results and ranking are consistent and acceptable.
b. *Search time consumption*: the average searching time takes less than 2 seconds. The Spark mapreduce handles large scale computation very well. For instance, word "university" has around 50,000 and "pennsylvania" has 20,000 entries. The query for "university of pennsylvania" only takes 1.5 seconds to complete.
c. *Query*: the length of user query doesn't affect performance that much in our implementation. Typical time cost examples are listed below:

	monogram	bigram	trigram	Words > 4
time cost /seconds	< 0.5	< 1	< 1.5	< 1.5
# of entries retrieved	~16,000	~50,000	~50,000	~250,000

III. Conclusion

The final implementation successfully achieved most of our initial design for the system. Every component is scalable and distributed: the design of stored data and use of distributed Berkeley DB contributes to overall scalability and shortens data retrieval cost, efficient MapReduce procedures of indexer and page-rank engine assure fast and correct intermediate data processing. However, even though we crawled 300,000 pages in a short time, the data is still insufficient with respect to searching results. Also, we can still improve searching performance by introducing more useful document information and adjusting ranking function.