CIS 455 Final Project

Steve Kreider, Rohan Bopardikar, Adam Cole, Alex Wissmann

December 14, 2015

1 Introduction

We have built a project called CIS-SEARCH: a distributed Web indexer/crawler which will allow users to search queries much like the google search engine. The project was built on several components: namely a web crawler, a TF/IDF indexer, the PageRank algorithm, and an interface for users to search queries and see the results in order of relevance. The relationship and details of these components are discussed the the Project Architecture section of this proposal. Following the dsicussion of the implementation of the project, we have offered an analysis our work and provided this final report of our findings concerning efficiency, accuracy, and other relevant discoveries.

2 Division of Labor

- Rohan implemented the crawler.
- Steven implemented the indexer.
- Alex, due to his familiarity with Apache Spark and how to efficiently integrate its functionality on Amazon webservers, primarily worked on PageRank.
- Adam, due to his DMD background and front-end expertise, focused on designing and implementing the search engine and web interface that accompanies it.

3 Project Architecture

3.1 Crawler

The crawler builds on the solution developed in HW2MS2. In order to increase its robustness for the project, the crawler's design was improved to model the Mercator blueprint.

The crawler was multithreaded and flexible enough to be run on EC2 instances or locally. To adhere to the Mercator standard, a set of queues was used as the URL frontier, with URL's being hashed to the appropriate queue as they were discovered. Each worker was aware of its own "number" and thereby knew what URLs to handle, allowing itself to operate almost independently of the other workers.

The crawler primarily supports HTML content, respecting the robots.txt file and sending the User-Agent: cis455crawler header with every request. It also uses JSoup as a tool to parse HTML files and extract the relevant links for them.

Periodically, data from the crawl was sent to AWS to be used by all parts of the project. The data uploaded included the date a page was visited (to avoid recrawling pages), the actual page itself (for the indexer), what links each page had on it (for PageRank), and the titles of each page (for the search engine). This information was stored on S3 buckets, or, in the case of the titles, on a DynamoDB table, for quick and easy access.

3.2 Indexer

The indexer builds on top of the work of the crawler to create a lexicon, inverted index. It uses MapReduce to analyze the data gathered through the crawler and serve the resulting information across multiple nodes to increase efficiency during lookup. The indexer provides an interface to access this data, by taking in a query and returning weighted answers that make use of the TF/IDF, proximity, and other relevant ranking features.

3.3 PageRank

PageRank builds on top of the work of the web crawler. It uses the data returned from the crawler to perform an implementation of the PageRank algorithm across this data by analyzing the links within documents. This algorithm runs as a Spark job. The PageRank results will then be stored, and we provide an interface to the search engine, which returns the relative rank of pages to assist in creating search results.

3.4 Search Engine and Web-interface

The search engine is an independent Node application which is responsible for taking in user queries and returning ranked, relevant search results. The engine is deployed on AWS's Elastic Beanstalk which handles many aspects of scalability such as load balancing. The engine sits on top of the DynamoDB data tables produced by the crawler, indexer and pagerank. More specifically, when a query is made, the web interface queries the metadata from the crawler's tables, the index information from the indexer and the pagerank data from the pagerank tables, and then uses an algorithm to rank the results.These results are outputted back to the user who is able to browse and select a links to open in his or her browser.

4 Implementation

4.1 Crawler

To follow up on the HW2 solution, the crawler was implemented using the Mercator design detailed in the relevant paper. Just like Mercator, the crawler had the URL frontier, content-seen test, and link extractor components. All of these were designed to operate in a thread safe manner, in order to make the crawler distributed and scalable.

The URL frontier was comprised of multiple FIFO queues, one for each worker. Following the suggestion in the Mercator paper, each worker was responsible for a hash range of URLs, and URLs were hashed to each queue based on the hostname. This way, even if there were multiple workers, only one worker would be requesting pages from a particular host at a time, preventing one host from getting overloaded. In this case, a blocking queue proved to be the best implementation, as the *take*() method allowed a worker with an empty queue to wait for another worker to enqueue an item onto it without needing to exit the thread. Once a URL was removed from the queue, the content-seen test was run.

The content-seen test was implemented in a very intuitive manner. First, the URL was checked against a list of all pages visited during the current crawl, as this information could be accessed the fastest. This list, visible to all workers, was maintained in a HashSet, because of its $O(1)$ implementation of the contains() method, making this portion of the test very quick. Subsequently, the URL was compared against the list of all documents crawled from previous runs (and the dates), stored in an Amazon S3 based on their hashed URL name to allow for smooth access. At this point, the crawler then either used the contents of the downloaded document (based on the outcome of If-Modified-Since), or the cached version from S3, to run the link extractor.

The link extractor exmployed JSoup to parse HTML documents and to find all the links. The crawler adhered to RFC guidelines to determine how to interpret a link from an href tag. During this time, the crawler also noted the title of each page as well as a list of all links stemming from that particular document. Each URL was then hashed to the appropriate worker. Because each worker knew the total number of workers as well as its own "number" out of the total, the crawler was completely decentralized and each worker could easily send the URLs it found to the correct queue.

Periodically (the exact amount was tweaked a lot because S3 would sometimes close a handshake if requests were made too quickly in succession), the crawler would upload data in batches to S3, which proved useful to all components of the project. The document contents and dates of each visit were put in S3 buckets based on their hashed URL names. A list of each URL to its hash was also kept in S3 as well, so that the indexer knew what URL it was indexing. For PageRank, a list of all links stemming from each page was stored in S3 too as a simple CSV file. Lastly, the titles of each document were stored in a DynamoDB database to be used by the search engine later.

4.2 Indexer

The indexer is comprised of inputPopulater (which populates the input for the mapreduce job), prelimDriver(the driver for the prelimMapReduce), idfDriver (the driver for idfMapReduce) and TablePopulator (populates the table in DynamoDb with output from the idf-map reduce).

The prelimMapReduce is structured very similarly to a word count job in map reduce. The map step takes in a mapping of URLs to filenames. Each file is donwloaded and parsed in JSoup to retrieve each for in an array. Then for each string in the returned array, the Map first makes sure the word isn't a stop word. Next it maps the word with the URL appended to a value of 1 and the size of the array.

The reduce method of prelimMap reduce adds all of the wordcounts (the 1's from the mapping step) together and preserves the total word count of the document. With this information, we have all we need to calculate the relative frequency of any word in any document.

The next job is idfMapReduce. This job mostly just calculates the count of documents that contain the word. The map class in idfMapReduce doesn't do very much. The only thing it does is make the URL portion of the previous job part of the value rather than the key.

After the mapper runs, for each webpage there is a key for each unique word in that webpage. This means that if we count the number of keys that are the same, we will be able to count the number of pages a word appears on by counting the number of identical keys. The reduce class does just that, and for each URL it appends a string consisting of the URL and the data necessary for counting the relative frequency of the word on that page seperated by " - ".

Finally there is the TablePopulator, this takes in output from idf mapreduce and calculates the tf-idf and adds the data to a table in DynamoDb.

4.3 PageRank

The Initial Edge List: We implemented PageRank as a series of Spark Jobs running on Amazon Elastic MapReduce. The crawler outputted a series of edges in the page graph as comma separated tuples. These were stored in a folder called links-db in our S3 bucket. The first job (now called prepareGraphData) took in all of these edges as a StringRDD, which is then converted to an RDD[(String, String)] by splitting on the comma. However, this data is still raw. It need to be cleaned for dangling links and other edge cases, and converted into a format Spark's GraphX library can use.

Dangling Links: All of the left hand sides of the edge list correspond to pages actually parsed. To get a unique list of vertices, we map to just position 0, and find distinct values. Then, we use a scala provided hashing function to generate Long id values for each vertex. To build the EdgeRDD and VertexRDD GraphX needs, we need to use Long values to identify nodes. So, we map from the RDD[(String)] to RDD[(Long, String)] to get a listing of vertex IDs and URLs. We also map over the edge list, using the same hash function to find corresponding IDs for each URL.

Any edges that don't point to one of these vertices are dangling links. To remove these dangling links, we join the id column of the vertices to the right hand column of our new edge list. We then just include the original edge data from the results. This ensures that no edges to non-vertices are included. Then, both the list of edges and vertices are mapped to Strings and written to vertices-clean and edges_clean in our S3 bucket.

PageRank: Spark's GraphLoader class allows the user to simply specify a tab separated edge list to construct the graph from. In the last step, we outputted such a list to edges_clean, so now it is loaded into a distributed Graph. Then, PageRank is run with a certain number of iterations. We decided early to run the PageRank algorithm in this way, since the goal was simply to reach the greatest number of iterations we could within out resource contraints. The output of this PageRanks was a Graph of Double values. The vertices of this graph contained the specific PageRank result, along with the vertexID. To generate the final mapping from URL to PageRank score, we load the list of vertices generated earlier, and join on vertexID. We can then easily output a list of URL's and their corresponding PageRank values.

DynamoDB integration: To load the final output data into DynamoDB, we used a hive script, which loaded data from the PageRank output and did a single large table overwrite. This was much faster than using the AWS Java SDK iteratively, since the index could then be built all at once, instead of incrementally. Additionally, this allowed us to easily distribute the copy task on our EMR cluster.

4.4 Search Engine

The search engine is implemented in Node and is responsible for providing an interface for the user to query the data produced by the other 3 parts. This is accomplished by creating a web app with two major routes: the home page ("http://search-engine-dev.elasticbeanstalk.com/") and the results page ("http://search-engine-dev.elasticbeanstalk.com/results"). The app is structured to have independent modules for the AWS database querying, backend data processing and client side views.

The basic flow is as follows: when a user goes to the homepage, they are prompted for a query. The search engine also implements autocomplete, so as the user types in queries, AJAX requests are sent to the '/autocomplete' route in the server which responds with possible words listed by relevance displayed to the user.

Once the user submits a query, issues a request to /results. (for ex."/results?searchQuery=example%20Query"). On the server, the query is split by words and converted to lowercase to ensure matches in database queries. Then the TF-IDF DynamoDB table is queried for the search words. A list of unique URLs that the words are found is generated along with the TF-IDF data for each (word, URL) pair. The list of URLs is passed to the database module, which then returns a list of the corresponding PageRank data as well as metadata

such as titles from the crawler. This entire process is implemented using JavaScript Promises to ensure the entire process is asynchronous and therefore non-blocking.

From here, the TF-IDF data, PageRank numbers, and metadata are passed to the ranking algorithm. First the query and each document is converted into a TF-IDF vector, and we calculate the corresponding cosine similarity between the query and each document. Then we calculate a metadata score which estimates the relevance of the query to the document by analyzing aspects such as if the query terms appear in the document title or URL. Finally, each of these scores is multiplied with the PageRank score generating an array of (URL, rankScore) pairs. This array is sorted by the rankScore and sent back to the client which renders the results as a list of links with corresponding document titles.

5 Evaluation

5.1 Crawler

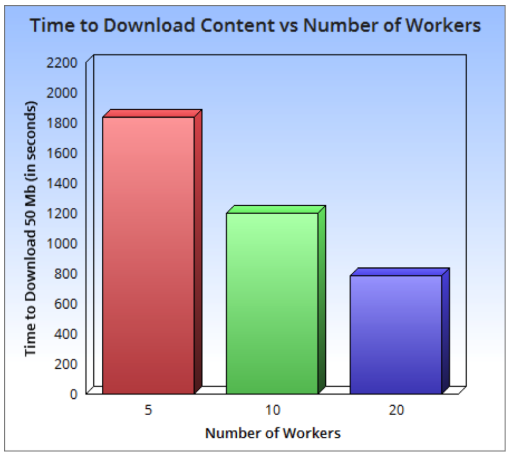

Figure 1: Time to Download 50 mb vs Number of Workers

As one can see from the above graph, the time it took to download 50 megabytes of content (empirically equating to approximately 2500 documents) reduced significantly as the number of workers was scaled up. However, as the graph indicates, there was a diminishing return to just mindlessly increasing the number of nodes, as HTTP requests could only be made so fast. Additionally, increasing the number of workers from 5 to 10 allowed URLs to be more evenly distributed among the queues (as there were more of them), but going from 10 to 20 did not allow for the crawler to realize as much of this distribution advantage, as the URLs were not as badly distributed in the 10 node case as they were in the 5 node case. As an aside, more than 20 workers did not seem to provide enough of a performance difference in this test to justify including more data in the graph.

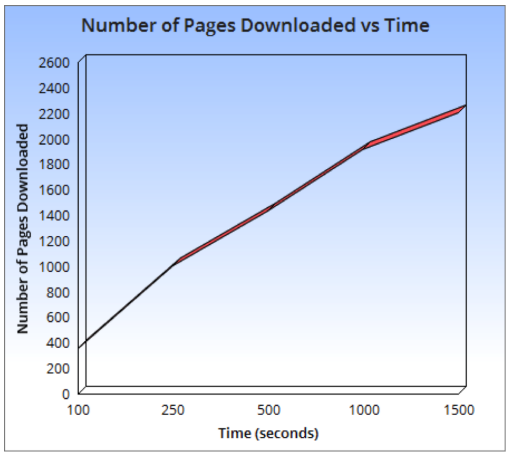

Figure 2: Number of Pages Downloaded vs Time (for 20 workers)

The above figure shows that as one would intuitively expect, as the crawler ran for more time, the number

of pages downloaded increased. However, we once again see diminishing returns coming into effect, as the rate at which pages were downloaded slowed down over time. Experimenting with the number of nodes did not seem to alter this phenomenon too significantly. The throttling in crawl rate, however, was likely due to the increase in the number of pages in each queue. As more time went on, each queue grew at a faster than linear rate (because one page generally has more than one link). Thus, the time it took to place new URLs into the queue increased as a result, creating a small rate slowdown. Although the graph does not show data for even larger periods of time, our empirical evidence confirmed that the slowdown occurred at a greater rate as even more time went on.

Ultimately, as the crawler was run in multiple iterations, the number of workers used varied between 10 and 30, depending on how urgently we needed pages for the other components of the project. However, as long as resources were not an issue, as the graphs show, despite some diminishing returns, more workers proved to provide better performance overall.

5.2 Indexer

The indexer runs relatively fast. The bottleneck with the indexer is definitely the map function of preliminMapReduce. This is most likely because every iteration of the map function has to download a file from S3.

The search times on the index were initially bad because the database had URL as a sort key for the table. The search times were sped up drastically when we removed the sort key and changed the value to be a list of mappings of URLs and TF-IDF scores. The upload for the indexer ran really fast. We mostly attribute this to being able to run a bunch of instances of the TablePopulator at the same time without concurrency checks because of the way we split the input file.

5.3 PageRank

Evaluation of PageRank results was fairly simple, since we were using the Spark provided implementation. Using small sample sets, however, we tested the elimination of dangling links, and ensured that the URL corresponding to each ID was replaced appropriately in the final results. We separated each the preparation, main computation, and final output into separate Spark jobs.

To reduce memory usage, we increased levels of parallelism for the joins in the first and last steps. With parallelism at 100, memory usage per task was low enough to handle our final dataset. Memory usage was the only real constraint on these tasks, CPU usage and runtime were both minimal.

The primary bottleneck of this data pipeline was the PageRank computation itself. Memory usage in each node increased with each iteration, filling up the Java heap space and throwing exceptions. Even after isolating the PageRank algorithm as the only memory user, the errors persisted. To resolve this issue, we decreased the fraction of storage allocated to memory. However, this had the tradeoff of increased disk reads/writes. The figure below shows the maximum number of iterations we could run on the final dataset for each amount of caching.

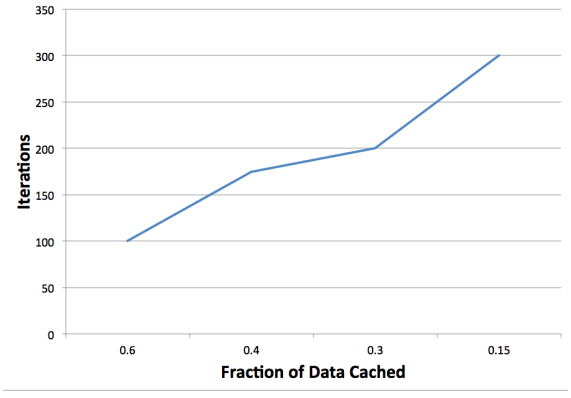

Figure 3: Number of iterations allowed with given spark.storage.memoryFraction setting

If no memory is dedicated to caching, then there is no bound on the number of iterations. However, with no caching, the CPU and I/O demands of the job were too high to finish in a weekend with our 6 node EMR cluster. Thus, we settled on 300 iterations, which runs in a much more reasonable 21 minutes with a memory fraction of 15%.

The performance of the Hive script copying from the S3 output to the DynamoDB table was directly related to the Write Capacity of the DynamoDB table. However, with this set to a reasonable level (about 100) for the duration of the copy, this step was not a bottleneck.

5.4 Search Engine

The search engine is deployed on AWS' Elastic BeanStalk which ensures availability and scalability. This is because Elastic Beanstalk automatically scales the application to grow or shrink depending on the applications needs. It also provides a load balancer to ensure good response time regardless of the number of concurrent requests made.

Therefore, the bigger bottleneck of the search engine was querying the DynamoDB tables and computing the rankings list. Each additional query word meant querying more words, urls, and metadata and therefore, becomes a larger computation job. However, the Elastic Beanstalk servers have enough power to make these queries and computations in a reasonably fast time. Below I have provided a study of the response time vs. the number of query terms. As you can see, response time becomes longer for longer queries but is not unreasonable.

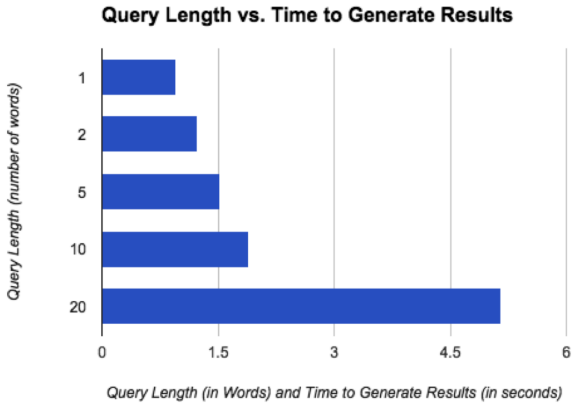

Figure 4: Query Length vs Time to Generate Search Results

While the quality of the rankings are somewhat subjective, I did a qualitative study to ensure that ranked pages had the relevant query terms and that pages with higher PageRank, cosine similarity and relevant metadata ranked higher than less related pages.

6 Conclusion

In conclusion, we were able to create a scalable and distributed search engine that operated independently on Amazon AWS servers and shared data between its individual components in order to produce queries in a relatively short amount of time. Based on our evaluations for each part, we found ways to optimize our implementations such that the latency for each component was minimal given the massive amount of data we obtained. Similarly, through our indexing, PageRank, and overall ranking system, we felt that given the pages crawled, the search engine delivered relevant results without a significant delay for the user. In the future, with more time and resources, we would hope to increase the quantity and quality of pages crawled in order to further test the efficacy of our search results. Additionally, we would hope to employ more components, such as a page's title or URI content, into our indexing to better reflect the key points of a document into our rankings.