INTERNET AND WEB SYSTEMS FINAL PROJECT - SPRING 2017

Bharath Karnati, Ethan Abramson, Kushmitha Unnikumar & Uzval S Dontiboyina

***Abstract*— In this paper we present our methods and approach to creating a scalable web search engine. To accomplish this goal, we utilized a variety of different efficient and scalable computation, storage, and retrieval techniques to handle the vast amount of data being crawled. We also utilize several programing frameworks, libraries, and paradigms to utilize those resources. In this paper we present our approach, our architecture, some implementation details, and an evaluation of our work.**

I. INTRODUCTION

A. Project Goals & High-Level Approach

The primary end-goal of this project is to create a web-based search engine to easily find relevant content for user provided queries. To do this, we designed and built a multi-threaded, stormlite-based, distributed crawler that can crawl a significant portion of the available web and scale so that processing and storage can be done on several different nodes. Utilizing the data obtained from the crawler, we designed and built an indexer that can scale to the size of our datasets and parse the content of each web page. It then creates an inverted index to quickly and efficiently pull up relevant pages given search queries. The indexer, which was run as a MapReduce job, will additionally calculate a TF-IDF ranking to provide relevance measures for matched search terms. Also utilizing the data obtained from the crawler, our PageRank component uses the MapReduce framework for accomplishing a scalable distributed calculation of the ranking for each website. Next, our Front-End component use these stored calculations to look for the search terms in our indexer and find a ranking of relevant pages. Then, it thens looks up the PageRank scores for each website. It then scales and combines these rankings to come up with a final preference ordering. Once that is complete, it send the necessary data to the users computer (responding to an AJAX request) so that their browser can arrange and render the final search results.

B. Division of Labor

1) Uzval: Twitter search, Ebay search, Text to Speech Synthesis, GeoTagging, PDF Handling, Indexer, Scripts, Asynchronous HTTP client

2) Bharath: Crawler, Front-End, Ranking, Request handling Servlet for Front-End.

3) Kushmitha: Indexer, Querying, scripts to convert data across platforms, Dangling pointers & Ranking.

4) Ethan: PageRank, Front-End, & code for integration

C. Actual Timeline

1) Crawler:

- Development Began: April 27th
- First Working Local Tests: May 1st
- First Batch crawl: May 3rd
- Second Batch crawl: May 4rd

2) Indexer:

- Development Began: April 27th
- First Working Local Tests: May 1st
- First Large-Scale Tests on Amazon Elastic MapReduce: May 3rd
- Final Indexer Computation & Integration: May 6th

3) PageRank:

- Development Began: April 27th
- First Working Local Tests: April 30th
- First Large-Scale Tests on Amazon Elastic MapReduce: May 2nd
- Final MapReduce Computation & Integration: May 8th

4) Front-End:

- Development Began: May 3rd
- Completed Most Development and Testing: May 5th
- Final Integration Testing: May 8th

II. PROJECT ARCHITECTURE

The crawler, which was designed to crawl the available web, download the contents of each webpage, compute the hash of the content using Rabin's algorithm and check if it is already present. If not, the URL is stored as key and the hash of the content is stored as the value in a Dynamo DB table. The contents are then stored in a S3 bucket - if it is not already present. The Indexer is then run as a MapReduce job that reads the contents of the bucket containing only the document text and writes the output to another S3 bucket. The Inverted Index is then stored in Berkeley DB for faster access during look up. The PageRank computation is run as a MapReduce job as well that takes an adjacency list representation of the captured hyperlink network (with the dangling links removed) and produces a text file that contains a mapping from URL to PageRank value. This data is then written to DynamoDB for faster access at query-time. The ranking of the relevant documents is done based on the TF-IDF values, PageRank values, percentage of the occurrence of the search query in the document URL, other metadata like the number of times the page has been updated during crawl. While the ranking calculation is being performed, a request is sent to the Twitter and EBay API's to fetch results for the specified search query. On the client-side, all of the HTML, CSS, and Javascript comes pre-loaded with the initial get request for the homepage. When a query is entered, an AJAX request is sent to the server with the user query. When a response is returned, the content gets dynamically injected into the page. The user can then select a particular tab to dynamically load results from Twitter or eBay.

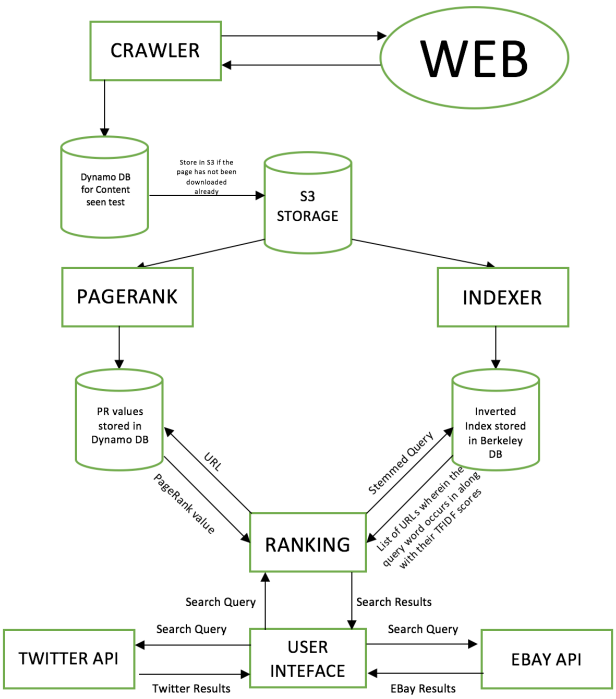

FIGURE: SEARCH ENGINE ARCHITECTURE

Fig. 1. Project Architeture

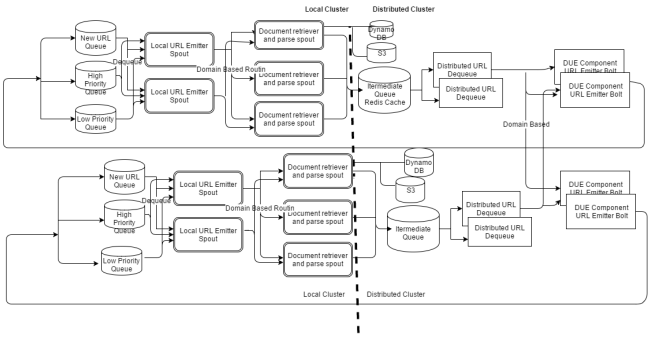

Fig. 2. Crawler Design

III. THE CRAWLER

A. Purpose

The purpose of crawler is to download the content from web politely without violating the robots.txt rules of any given website in a Mercator style based approach. Using the content downloaded from web, we calculate the relevancy of the content for a given search query. An optimized Crawler should download pages at a high frequency, minimize requests made to static content, maximize requests made to frequently updated pages, reduce amount of data being stored by using content seen test, should be distributed and scalable. All these traits were implemented in our crawler.

B. Design

We designed our crawler based on the Mercator design in a scalable, distributed way using the Stormlite framework. We used this framework to distribute the URL frontier among different nodes using

1) Distributed & Scalable: We implemented the distributed nature of the crawler based using the provided stormlite framework. We made this distributed across multiple nodes and scalable with a master-worker architecture by dividing the URL frontier based on domains. In our implementation we have created a functionality where we can add multiple nodes even in the middle of a crawl but it will have to wait only momentarily for the URLs dequeued by the spout to get downloaded so that it will not violate the robots.txt crawl delay.

2) Performance: To download the pages at high rate, as DNS lookup was the major bottle neck in a Crawler, we have built a custom HTTPClient with a asynchronous **DNS resolver** and caching it in case of multiple requests being made. Also we have optimized the performance by putting the thread to sleep without busy waiting.

3) Politeness: We made changes to stormlite framework such that even in case of a multi-threaded implementation, it does not dequeue more than one thread of a particular domain.

4) Priority Queues: We designed our crawler to be continuous with features for updating the content. This was done using using URL frontier queues with multi-level priorities based on the history of the page as perceived by the crawler in its previous crawls. We used in-memory persistent cache for storing the queues which was useful for fast writes and reads.

C. Crawler Architecture

Our Crawler architecture can be divided mainly into two subsections-

1) Local Cluster: Our local cluster was responsible for dividing the distribution of URL frontier among its threads with each Bolt thread responsible for each domain. In local cluster we have made changes to ensure that each thread at any given time is responsible only for one domain. The spout in Local cluster dequeues from the priority queues such that they new URLs are downloaded more often than the old URLs. The bolt is responsible for downloading, parsing and then pushing the out going URLs to an intermediate queue. This bolt also performs content seen test and don't store if that pointer to that content is already present.

2) Distributed Cluster: Our distributed cluster was responsible for the routing of the URL among several nodes and splitting based on the domains. The spout dequeues the URLs from intermediate queue and the emits the URLs across nodes for the bolt to push it to apropriate queue based on the DUE component.

D. Extra Credit

1) Asynchronous DNS Resolver: As DNS lookup is the bottleneck, we have created a custom HTTPClient with DNS cache.

2) Content Seen: We have calculated the hash of the content and stored it in dynamodb along with content in S3 such that multiple URLs with same content are not stored as seperate copies.

IV. THE INDEXER

A. Purpose

The purpose of building an inverted index for a search engine is to optimize speed and performance in finding relevant documents for a search query. Without an index, the search engine has to scan every document in the set of crawled documents, which would require considerable time and computing power. For example, while an index of 100,000 documents can be queried within milliseconds, a sequential scan of every word in 100,000 large documents could take hours. The additional computer storage required to store the index are traded off for the time saved during information retrieval.

B. Preprocessing

A preprocessing script was included since the crawled documents is stored in HTML format in S3. The script reads the crawled documents, strips the newline feed and extracts the text from them, combines a thousand documents into one, where each line starts with URL separated from its contents with a special string and writes into another bucket. This step reduces the time computation for the Indexer since a single read would fetch the entire contents of a document and makes it easier to keep track of the document a particular word occurs in.

C. Implementation Description

Building an Inverted Index can be considered as a MapReduce job. The input for the MR job is the S3 bucket from the preprocessing step above and the output is another folder in a S3 bucket. Amazon's Elastic MapReduce was used to run the job on around 200000 documents and it took 8 minutes to complete the job with 1 master and 3 core nodes, all of type m3.xlarge. The steps in the mapper and reducer phase are described below.

1) MAPPER PHASE: Each mapper gets one line from a file in the S3 bucket. The mapper then splits the line based on the special string used during the preprocessing step to obtain the URL and its contents. The contents are then tokenized and stemmed using the Stanford CoreNLP library. The location of each word in the document is stored in an arraylist and the normalized term frequency of each word is calculated using the formula,

$$a + (1 - a)\frac{freq.\ of\ the\ word}{max\ freq.\ of\ a\ word\ in\ the\ document}$$

where a is set as 0.5. Thus, each mapper emits the word along with an object of type custom class that implements the Apache's Writable interface. The custom class has attributes as the document URL, arraylist to store the positions a word occurs in that particular document, normalized term frequency and TF-IDF score.

2) REDUCER PHASE: Each reducer gets a word and a list of iterable custom class objects. The number of objects in the list is determined to calculate the inverse of document frequencies. The list is iterated to calculate the TF-IDF values, stored in the custom class object and is emitted as a JSON string that gets written into the output folder of a S3 bucket.

D. Extra Credit - Implementations were done but could not be integrated for the final search

Metadata from each of the crawled documents were retrieved by using Jsoup parser and stored. A inverted index was built with the corpus as the metadata. The idea was to include this table as well for final ranking and give extra weighting as compared to the normal inverted index. Provisions were made to keep track of the positions a particular word occurs in a document so that this could be used for phrase based search queries and for displaying the matching content at the front-end.

V. PAGERANK

A. Purpose

When categorizing the information on the web it is important to be able to find relevant information of the highest quality in the vast amount of information available. Due to the connected nature of the web, hyperlinks encode a significant amount of information about the importance of webpages. The PageRank algorithm utilizes this hyperlink structure of the web to infer the absolute, query-independent, structural quality of a webpage within the larger network.

B. Algorithm Description

Algorithm 1 MapReduce PageRank Algorithm

Iterate Until Convergence

procedure MAP($node, currentPR$)
  $deg \leftarrow |\{n2 \in Nodes\ s.t.\ (node, n2) \in Edges\}|$
  for $n \in Nodes$ s.t. $(node, n) \in Edges$ **do**
    $emit(n, \frac{currentPR}{deg})$
  end for
end procedure
procedure REDUCE($node, values$)
  $newPR \leftarrow \sum values$
  $emit(n, newPR)$
end procedure

One small detail in the above algorithm is that the adjacency list for each node must also be passed back-and-forth between the map and reduce functions so that the input format properly matches the output format. In practice, I tried to use a counter in hadoop to keep track of the sum of errors, but I was worried about over-specifying convergence on EMR and having the job cost a lot of money. So, instead, I ran it in batches of 25 iterations each. Since the output format exactly matches the input, it was easy to chain the jobs one after another if needed.

C. Implementation Description

Our PageRank implementation utilizes the MapReduce framework to calculate the ranking in a computationally scalable manner. After our final crawl we utilized Amazon's Elastic MapReduce so that we could easily scale to the many machines needed to compute efficiently with the amount of data being used.

VI. SEARCH ENGINE & WEB INTERFACE

A. The Front-End

Our front-end is designed to support a few different search functions. By default, you will receive standard google-style results for your search query. We also provide buttons that will pull in data from eBay or Twitter. All of this is supported using a myriad of front-end tools such as bootstrap, javascript, jQuery, and AJAX to asynchronously request content from our web server and dynamically load that content into the DOM on the client's machine. This allows our pages to be easily cached in the browser and also allows for the quickest possible load times when requesting varied content. We also utilize bootstrap and jQuery from CDNs that allow for easy caching in the client's browser.

B. Preprocessing for Querying

The output from the Indexer was written into a S3 bucket. This was moved to Berkeley DB, with the word as the primary key and the JSON string containing the list of document URLs, positions in the document, normalized term frequency and the TF=IDF scores as the value, for faster look up. We tried out moving this data to Dynamo DB but the write latency was high and there was limit of around 400KB per record, which we thought would cause problems since we supported the search for stop words as well in our implementation. Thus, we chose Berkeley DB as our database.
In addition, the PageRank algorithm wrote the computed PR values in a S3 bucket, which again had to be moved to a database for querying. Amazon's Dynamo DB was used for this purpose with the URL as the partition key.

VII. RANKING MECHANISM

The search query is passed onto the ranking function as the parameter. The query is tokenized and stemmed using StanfordCoreNLP. We used three parameters to rank the search results - The TF-IDF value of a word in the document, the PageRank value of the document and the percentage overlap of the search query with the URL of the document. These three numerical values are computed and combined. The weighting given to each of these factors were determined based on the results returned during our testing. The top ten results, based on decreasing numerical value so obtained are returned to the front end for displaying to the user. For search queries with multiple words, only those documents are considered that have both words occurring, i.e, the intersection of the URLs obtained from the inverted index stored in Berkeley DB.

VIII. EXTRA CREDIT

1) Twitter Search Results: This would enable user to see the search query results based on the tweets. In this section, we have customized the twitter results based on the geolocation, popularity of the tweet, and timestamp on the tweet. We also have included an custom written obscene content filter and language filter inorder to scrape out unnecessary data. We overcame the limitation which twitter4j has, for the number of search results which is 100. Instead, each search query of ours would retrieve 18,000 tweets(Twitter blocking limit). Instead of using the traditional way of retrieving the results by direct API calls, we iterated through the IDs in order to retrieve the tweets and applied filters in order to get good results.

2) Ebay Search Results: This extra credit consists of connecting to the ebay through ebay's Finding api. We retrieved the products based on search query. In case if there are products that doesn't match a full query pattern, we iteratively changed the phrases of the result in order to accoommodate results that provides a partial match.

3) Speech Synthesis: *In order to present user with a voice feedback of the search results, we have tried using freetts library to output a voice based on the search results. We tried to leverage Voice synthesizer's text Synthesis capability in order to perform this action. The voice is pretrained based on the pronunciation which it has received. The instance of the voice was used to output the final result in terms of speech by taking the results input. In the demo, we were succesful in showing the functionality, but couldn't interface it with program on EC-2 instance.

4) Geo Tagging: In order to present user with the best results possible. We tried tracking the user's location using their request. We downloaded an opensource database of IPs GeoLiteCity.dat. Based on the IP address, we queried the database and retreived the location from which the HTTP request came from based on the IP address. We used this result to provide a better result to the user request.

5) PDF Pages Handling: *We used pdfbox in order to handle the pdf type file formats, downloads during the crawling phase. This program would convert the content of a pdf into a string format which would be readily saved into the S3 database along with it's corresponding hashvalue. Since, the corpus was already ready before we added the functionality, we didn't include the results in our final ranking.

6) Asynchronous HTTP Client using Netty: *We developed an asynchronous http client using netty. In our multi threaded approach, we tried to make asynchronous http requests with call back functions invoking the thread once the response is ready. During the waiting time, the crawling thread would be put to sleep, so that it doesnt use any resources. We replaced this with our homework implementation while doing the final crawl.

7) Removing the Dangling Links before Pagerank calculation: We are careful to remove 'dangling links' which can come from a number of sources (pages yet to be crawled, improperly formatted hyper-links on web pages, pages with incompatible robot.txt policies, etc). We are also careful to

reduce the impact of 'rank-sinks' that result from pages with no outgoing links (or only self-loops) by adopting the random surfer model and introducing a damping factor into the PageRank calculation.

8) AJAX in Front end: We are using AJAX to dynamically load content into a single page within the clients' browser.

IX. EVALUATION

A. Indexer

The initial test on the Indexer was run without compiling thousand documents into one. Hence, it took around 40 minutes to index 30000 documents. The post on piazza that Hadoop works better on huge files rather than many small files gave us the idea to combine thousand documents into one. This processing step made a huge difference in terms of time computation and the indexing on 200000 documents took 8 minutes to complete. Another observation that we made was Hadoop is inefficient for small datasets in terms of time taken to complete since provisioning took the most time as compared to actual computation. However, with increase in data size, EMR works favorable.

B. PageRank

When running our PageRank implementation on a single machine we are able to process relatively small networks quite quickly. A network with a few dozen nodes will iterate to acceptable convergence within a few dozen seconds up to a minute at most at nearly free computational costs (assuming hardware is available). On the other hand, such a dataset running on Amazon Elastic MapReduce (with two small compute nodes) would take almost 10 minutes to complete and cost a total of almost $2. Clearly, for small datasets the economics are not in favor of the utility computing model. However, if we consider the ease and simplicity of moving from a small dataset on EMR with a few machines up to a large dataset and many compute machines, then we are able to see the desired trade-offs offered by the utility compute model. Without purchasing, maintaining, and configuring a compute cluster all to ourselves we are able to focus on the new services that can be built on top of these pay-for-use computing tools.

We tested two identical runs through EMR varying only the number of compute nodes utilized. With a relatively small dataset (1 GB) and 5 m4.large compute nodes the computation took 25 minutes total. When running the same dataset for the same number of iterations on 10 machines, the total took 23 minutes to process. I would assume that most of the computation time was in overhead moving between map and reduce phases, so increasing the number of machines didn't yield a much larger boost in computation time. Overall these computations cost a grand total of $2 combined with data storage costs.

X. CONCLUSIONS

Crawling, indexing, ranking, and finding relevant information from the World Wide Web is a challenging endeavor. In this project and this paper we present our methods for tacking this challenge in a manner that is extensible and scalable to much larger collections and computations.

XI. TAKEAWAYS FROM THE PROJECT

- An understanding of the functioning of Amazon Web Services
- Understanding of how tough the real scalable systems are and getting the first hand experience in dealing with their challenges.
- The utility computing model can allow us to quickly build scalable web systems
- Integrate early and often!

XII. OUR SEARCH RESULTS

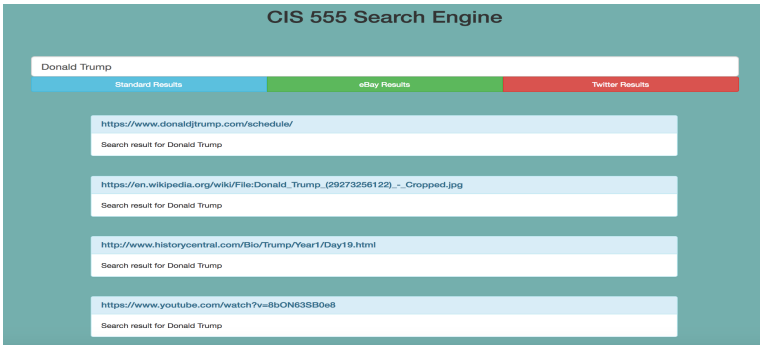

Fig. 3. Search Results

Fig. 4. EBay Results

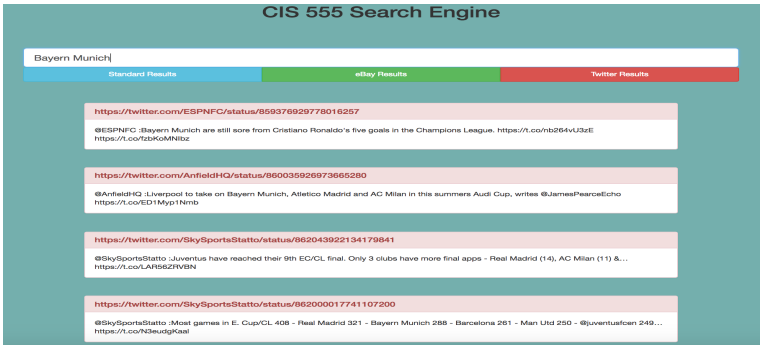

Fig. 5. Twitter Results