Team MNWD Search Engine Final Report
CIS 455/555: Internet and Web Systems

Divya Somayajula (Search Engine & UI) DIVYAS22@SEAS.UPENN.EDU
Meri Kavtelishvili (Crawler) MERIKAV@SEAS.UPENN.EDU
Nimai Agarwal (PageRank) NIMAI@SEAS.UPENN.EDU
Yixue Wendy Feng (Indexer/TF-IDF) WENDYFYX@SEAS.UPENN.EDU

1. Overview

We built our search engine using React.js for the front-end, Spark Java for the back-end, StormLite framework for the crawler, MapReduce job implemented in Apache Storm for the indexer, Apache Spark for the PageRank algorithm, and AWS S3, SQS, and DynamoDB for the database. All features and implementation details are described in this report. The instructions and necessary dependencies for running this search engine are in the README file.

2. Introduction

2.1. Project Goals and High-Level Approach

The following is a list of goals our team set out to achieve for this project, which we believe we did accomplish, as well as the high-level approach each of us took during the implementation of each component:

1. Crawl from diverse & credible sources to provide the best search results possible
2. Create a search engine that is a good citizen (respects protocols of other domains-robots.txt) and avoid malicious pages, cycles, and traps to accumulate the best quality content
3. Provide the most useful/relevant search results in the fastest time possible by implementing and refining a detailed ranking algorithm that takes into account multiple factors indicative of authoritativeness & proximity to the query
4. Create a database schema that allows for quick queries and retrievals, and minimizes the number of costly updates and scans by using optimizations such as DynamoDB's batch writes and reads
5. Find the optimal parameters (number of threads, nodes, etc) for our search engine, crawler, and indexer for the highest efficiency
6. Have a robust search engine that supports spell check and fuzzy search and enrich the user's search experience with relevant Amazon shopping results

2.2. Rough Milestones & Timeline

2.2.1. Week of April 19th

Crawler: Provide API, Move to DynamoDB and S3, move link queue to database (instead of HashMap), Decide on seed links, + save edges.
Indexer: Extract word level features (frequency) from test docs, create inverted index, design database schema for storing term-document info.
PageRank: Create first pass of page rank algorithm in Spark.
Search Engine & UI: Create sketch/design of search form, plan out route handlers for accessing page ranks/TF-IDFs/document info from database, plan out how to rank documents, use React.js to create search form
Overall: Plan out database schema, Spark routes, create databases

2.2.2. Week of april 26th

Crawler: Make distributed (Storm), fit into Mercator-style, actually crawl pages + testing. +update how we deal with robots.txt.
Indexer: Add support for storing term and document information to DynamoDB; remove stopwords and add lemmatization
PageRank: Deploy PageRank on EMR cluster; read/write from DB.
Search Engine & UI: Implement all route handlers and database functions, implement preliminary algorithm for combining PageRank, TF-IDFs, etc in order to rank documents, connect search engine with database/other components
Overall: Continue to integrate all components together

2.2.3. Week of may 3rd

Crawler: More crawling and testing + quality checking
Indexer: Integrate with crawler to get document content from S3 buckets. Index larger amount of documents on multiple nodes.
PageRank: More testing of ranking system and stress test with thousands of documents and links.
Search Engine & UI: Continue to refine ranking function, improve UI
Overall: Continue to integrate all components together

2.2.4. Week of demo

Crawler: More crawling and testing + quality checking
Indexer: Monitor the indexing process + quality checking
PageRank: Make sure fully integrated with crawler and deployed
Search Engine & UI: Finalize ranking function/UI, add search results from Web Services and support for spell check (as extra credit)
Overall: Finish integration, deploy on EC2

3. Project Architecture

Please see the appendix for a visual overview of the project architecture.

3.1. Database Schemas

1. ***documents*** (DynamoDB)
   The primary partition key of this table is *docId*, which is a hash of the contents of the document. Each *docId* has the following attributes: *docHeader* which is the title/header of the document, *excerpt* which is the first few sentences of the document, *highestFreq* which is the highest frequency of any one term in the document used for normalized TF computation purposes, *lastCrawledTime* which is the the time that the content at this URL was crawled, *numWords* which is the total number of terms, *pageRank* which is the page rank value of this document, *s3Ref* which is the id of the S3 object in which the contents of the document are stored, and url. The purpose of this table is to store metadata about every document crawled. These *docIds* are used as sort keys in the inverted index in order to keep track of the various occurrences of every term. This table is also used by the search engine for display purposes (eg: header, url).

2. ***contentSeen*** (DynamoDB)
   The primary partition key of this table is *hash*, which is the hashed content of the document that it corresponds to. This table is used to

check if certain document contents (even corresponding to different URLs) have been crawled and stored before, preventing the crawler from going into cycles.

3. ***documentEdges*** (DynamoDB)
   The primary key of this table is *from* and the primary sort key is *to*. A row in this table corresponds to the document with id *from* having an outgoing link to the document with id *to*. For every href link (for document H) in a given document D that a crawler extracts, an entry with primary key D's id and sort key H's id is added to this table. This is directly ingested by the PageRank algorithm for link analysis.

4. ***exploredUrls*** (DynamoDB)
   The primary partition key of this table is *url* and each *url* has an attribute called *docId*. This table represents the urls which have already been crawled and their corresponding entries in the documents table. This helps prevent the crawler from inadvertently crawling content at the same URLs over and over again (if the URL is embedded in many documents, for example) and this helps avoid the crawler going in cycles.

5. ***robotsInfo*** (DynamoDB)
   The primary partition key of this table is *url* and each *url* (where robots.txt of each domain is) has the following attributes: *lastCrawledTime* which is the last time this domain was crawled, *waitTime* which is the required number of seconds between subsequent requests to this domain, and *disallowedUrls* which is a list of relative links in this domain that are prohibited from being crawled. As the crawler explores different domains, it caches the robots.txt politeness information corresponding to that domain in this table, so that on future crawls of the domain, this information is easily accessible and doesn't have to be fetched again.

6. ***terms*** (DynamoDB)
   The primary partition key of this table (our inverted index) is *term* and the primary sort key is *docId*. A row in this table corresponds to the occurrence of term in the document with id *docId*. Other attributes include *normalizedTf* and *tf* for each (*term*, *docId*) combination. We initially had only the *term* partition key, and each term had an array of *docId* it appears in along with the corresponding term frequency. However, more common terms can exist in thousands of documents, making the item size for those terms very large very quickly, exceeding the 400KB per item limit in DynamoDB. We found this issue when we indexed beyond 50k documents, and had to change the terms table schema to include a sort key of *docId*. While this schema dramatically increases the number of items in the table, DynamoDB has no table size limit.

7. ***indexedBuckets*** (DynamoDB)
   The primary partition key of this table is *s3Ref*, referring to the ID of S3 buckets storing crawled document contents. An indexer node downloads one S3 bucket at a time and indexes all documents in the bucket (ranging from 50 to 1000+ documents). To prevent multiple indexer nodes indexing documents from the same bucket, the bucket ID, or *s3Ref* is saved to this table when it's initially downloaded by an indexer so that for each bucket the indexer encounters, it can check if it has been accessed by another node. This table makes it possible for multiple indexer nodes to run concurrently without indexing the same content.
   Failure Handling: Since we only store the bucket ID when it's initially downloaded, if the indexer is interrupted in the processing of indexing documents in a bucket, the leftover documents will not be indexed by another bucket because the bucket ID will already be in this table. This could be improved by keeping track of bucket state if indexing is finished or interrupted. However, because we had a large number of documents crawled, and the indexer rarely ran into issues or failures, we lost at most a couple hundreds of documents which doesn't affect the overall quality of the indexer.

8. ***searchResultsCache*** (DynamoDB)
   The primary partition key of this table is *searchQuery* which is a lowercase, stemmed, and alphabetically sorted version of the search query. Each *searchQuery* has an attribute of type List of Lists called *matches*, which is a sorted, ranked version of the first 150 search results. This table was designed to provide quick access to cached search results by having a standardized version of the query as the primary key. Also, in order to respect the DynamoDB Item size limits, only 150 of the results were cached, as users will most likely only look at up to the first 15 pages.

9. ***document-contents*** (S3 Bucket)
   Document contents are stored in S3 bucket. Each object in the bucket contains on average 70mb of data, which corresponds to about 300-400 different document contents. While crawling the web-pages, the content is stored in Java main memory in a hashmap. In the hashmap, the key is the document id, and the value is the content of the corresponding document. Once enough documents are buffered in the memory (about 60mb of data), then the hashMap is serialized into a JSON object using gson library, and then the serialized object is written into a file and uploaded to S3 as a new object. This way, we can maintain a reference from each docId to its contents, and from S3 object to docIDs.

10. ***SQS Crawler Queues***
    There are two main queues that the crawler uses. Wikipedia queue for Wikipedia domain, and another one for every other domain. For the Wikipedia queue, the seed url was Wikipedia portals page, and since Wikipedia is mostly self-contained (each document mostly links to other Wikipedia pages), the crawler that was working on this queue ignored all the other domains. The other queue was populated using odp, dmoz, and botw web directories. Crawlers working on this queue ignore all Wikipedia pages.

4. Implementation Details, Issues & Difficulties, and Optimizations

4.1. Crawler

Implementation: Crawler was implemented using StormLite in Java. The topology architecture is the following: It has a **urlSpout** that retrieves urls from the database, and then forwards these urls to the **filterBolt**. For each url, the filterBolt decides what to do with the url, either emit, disregard, or put it back in the queue. In addition to this, it also makes an edge (to be used by pageRank) and puts it in the database. **ContentFetcher** bolt then gets the content of the received url, checks if the content already is seen, and if not, sends the new content to the **linkExtractor** and the **docSaver**. LinkExtractor extracts all the links from the content and saves them to the SQS queue. The docSaver saves the document with all the metadata, and also saves the document contents.

Issues & Difficulties: Initial issue was the speed of the crawler. It would crawl less than one document in every 5-7 seconds. This meant that we would not have 200k documents crawled in time. In addition to this issue, we also had to monitor content quality and diversity of the sources. No matter what seed url we started from, if one of them was Wikipedia, Wikipedia was turning out to be the majority of the crawled documents.

Optimizations: To fix the diversity of the resources issue, we created two separate queues, one holding links from Wikipedia and the other holding links from every other domain. This as well sped up the crawler and made sure that we would have resources not only from Wikipedia.

There were several optimizations done to increase the speed of the crawler. In the end, about 600k documents were crawled at a rate of three docu-

ments per second.
1) Initially, there were only three bolts/spouts. One was for fetching urls, other for contents, and the last one saved the docs and also extracted the urls. Extracting the urls from the documents was taking a lot of time and was the bottleneck. Separating work for extracting the links and saving the documents considerably sped up the crawler.
2) To minimize the need to retrieve data from the databases, each read also returned some extra information. (e.g asking the database if some object with specified id exists, also returns the object if it exists). In addition to this, some information about robots for the domain was stored in the main memory instead of the database. Batch read/write/delete was used to speed up the crawler. In the UrlSpout, each thread read and deleted ten urls at a time. LinkExtractor saved ten urls to the frontier at a time instead of saving one at a time. Number ten was chosen because it is the maximum number SQS supports for batch processing. Documents and Edges were also buffered up in the main memory and written into the dynamoDB database 25 at a time (25 is the highest number dynamoDB supports for batch processing). In addition to the documents, document contents were also buffered up in the memory in a hashmap (the way it was described above), and then written into S3 file on average 60-70mb at a time.
3) I also experimented with the number of threads to see what worked the best. In the end, I left 10 urlExtractors, 15 filers, 20 docFetchers, 20 savors, and 30 linkExtractors.
4) Initially, grouping strategy was group by field, based on the domain. This worked well for the crawler that was working on many domains at once. However, I changed this grouping strategy to shuffle grouping for the crawler that was working on the Wikipedia pages. The reason is that since there was only one domain, not every executor was being utilized, and every url was going to one executor.
5) In addition to other optimizations, scaling horizontally also helped speed up the crawler. Instead of running only one EC2 instance, we ran 2-3 instances at a time, which doubled the crawling time.

4.2. Indexer & TF-IDF Retrieval Engine

Implementation: As described above, the main indexer logic is a MapReduce job implemented in Apache Storm. The job runs on a local cluster, and saves inverted index (term-document pair) information to a cloud database in DynamoDB. Each component is described below:

DocSpout reads crawled document content from S3 from **FileGenerator** object, clean content (remove HTML tags using JSoup and filter documents), and save document title and excerpt to documents table in DynamoDB. **FileGenerator** gets document content from S3, implement hasNext() and getNextDoc() similar to Java iterators. The IDs of S3 buckets being/already indexed are saved to *indexedBuckets* table to allow the indexer to run from multiple nodes without indexing the same content twice. **MapBolt** gets processed document content from **DocSpout**, and emits term-document pairs (can support multiple executors). The document text is processed into terms using the Stanford CoreNLP pipeline, which performs tokenization. In the case that we run into large documents, we process only the first 30k tokens in each document. We then remove stopwords using a predefined list containing 127 stopwords from NLTK, and filter terms that only contain letters, numbers and dash (-), and are between 2 and 25 characters. After filtering, each term is lemmatized using Porter Stemmer, and emitted with the document ID. When it finishes emitting term-document pairs (reach the end of the document or exceed 30k tokens), it emits a EOS signal to **ReduceBolt**. **ReduceBolt** processes each term-document pair as it comes from MapBolt and saves in a singleton **Documents** object. The inputs are grouped by document ID, so it was guaranteed that all terms in each document would be processed by the same executor. Once it receives EOS from a document, it emits the inverted index for the current document (each unique term and its term frequency in the current document) to **TermBolt**. Similar to the forward index in the Google paper, **Documents** is a singleton object that keeps track of document metadata (highest frequency of any word, and total number of words in the document), and term (TF, first position, etc) state before the reduce stage. After receiving EOS from a document, **ReduceBolt** calls updateDb() to save document metadata to the documents table and the document state is removed from memory. **TermBolt** receives input from ReduceBolt and batch write to the terms table in DynamoDB (can support multiple executors). Each TermBolt executor has its own **TermDocInvIndex** object, which is a temporary object for storing the inverted index in memory before batch writing to DB.

Issues & Difficulties: One of the biggest challenges is that DocSpout emits documents too fast for the subsequent bolts to process and save to DB. Unlike StormLite, we can't put a limit on how big the task queue is, and if the spout emits content faster than bolts will process then, it's guaranteed to explode and run out of memory at some point, especially if we have to index a large amount of documents. To deal with this, we first limit the spout to only have one executor, and have the spout sleep for some time (1 sec) after emitting each document. However the time it should sleep takes a bit of time to tune for different machines or nodes, and time for sending request/receiving response to DB is also unpredictable. To deal with this, we added the **TermDocInvIndex** object, so that each **TermBolt** executor can save a couple thousands term-document pairs in memory before it flushes them and writes to DB. In addition, we created a singleton **UpdateState** object to keep track of how many TermBolt executors is currently writing to DB. **DocSpout** polls for the state through this object, and if at least one TermBolt executor is updating DB, it stops emitting new documents. One limitation is that if any TermBolt executor is writing to DB, the whole process stops. However, batch write in DynamoDB is fairly fast, and with only 4 TermBolt executors, this was not a huge bottleneck in indexer speed.
Another challenge is having the indexer run on multiple instances (local + EC2) and being able to index large amounts of documents. Luckily, once a single indexer pipeline is set up and writing to a remote cloud DB service such as DynamoDB, the only thing left to make concurrency happen was controlling the input of **DocSpout**. We implemented **FileGenerator** to read from S3 buckets and output one document at a time with functions hasNext() and getNextDoc() similar to a Java iterator. While these two functions emit documents, we added extra logic to deal with retrieving new buckets when all documents from the current bucket are emitted, downloading/parsing an S3 bucket into documents, checking if a bucket has already been indexed, etc. When running the indexer, we specify how many buckets we should process and index using **maxBuckets** in **FileGenerator**, instead of the max number of documents, and once exceeded, the program can terminate. The main indexer program polls for the **FileGenerator** state to check if it should terminate. Upon termination, the main indexer program sends signals for all TermBolt executors to flush term-document pairs in memory to DB, shutdown all DynamoDB and S3 clients, and shutdown the local cluster.

Optimizations: The major bottleneck for the indexer is writing large amounts of term-document pairs to DynamoDB. After updating the terms table schema to use document ID as sort key, we were able to utilize batch write to speed up the indexer instead of performing only update operations. We increased the number of executors for **MapBolt**, **ReduceBolt** and **TermBolt** all to 4, and ran the indexer on two EC2 instances in addition to the local instance. Since the EC2 instance had more memory, we increased the number of entries that **TermBolt** could keep in memory before writing to DB to reduce the time the topology remained idle (when DocSpout is not emitting new documents).

4.3. PageRank

Implementation: The page rank algorithm was implemented in Spark in Java and runs on an EMR cluster. The algorithm followed the specifications given in class, with ranks propagated along edges at each iteration and aggregated with a decay factor of 0.15 and running for at most 25 iterations or until convergence. After the page ranks are all computed, each value was written to the documents table.

Issues & Difficulties: There were bugs with the EMR DynamoDB connector which I used to load our DynamoDB edge graph into a Spark

RDD. I ended up having to scrap the plugin and simply dump the dynamo table to an S3 file, and load that in and parse it in Spark manually. This approach ended up fixing some frustrating bugs. There were also many issues with deploying the Spark job to EMR. At times, the Spark job would randomly terminate without errors, even though it would run to completion locally. I had to simplify some of the logic to make it work.

Optimizations: Spark allowed the computation to run faster due to the parallelism. Because of the massive size of the dataset (the edge graph had far over a million edges) it was necessary to distribute using Spark and EMR or the computations would have taken too long. I was careful to write the Spark operations so they would run efficiently.

4.4. Search Engine and User Interface

Implementation: The search engine and UI consist of a React.js client-side application and a server built on the Spark Java framework. The React.js application maintains a few States such as *page* and *results*, so a change in any one of these will refresh and re-render the page. The interface consists of a text box where users can input their search queries and then click the Search button. Upon this, the function *onSearchHandler*() is called and a request is made to the route on the server side called */getDocuments* with the parameters of the search query and default page number of 0. How this route works will be described later on, but once the client-side receives the String response, the Javascript JSON parser is used to create an array of objects, each of which represents a search result, and the state *results* is set to this, so the page is re-rendered, and the user can view their search results. If the user wishes to see the previous 10 or next 10 results for their query, the handlers *prevPage*() and *nextPage*() are called, respectively, and they work just like *onSearchHandler*() does.

The bulk of the search engine operation happens on the server side. The Route *DocumentRetrieval*, corresponding to */getDocuments*, contains a *handle*() function which first parses out the *query* and *page* parameters from the Request object. Each query is split into a list of multiple terms. In order to stay consistent with the indexer, each term is made lowercase and stemmed according to the Porter Stemmer algorithm and all stop words are filtered out. Then, the terms are sorted in alphabetical order and concatenated into a string, and then the *searchResultsCache* table is checked. If the result R from the cache is not empty, this means this query has been made before and we can simply return those cached results to the user. R is indexed into depending on the requested page number and that segment of 10 results is translated into a JSON-translatable string and returned. Otherwise, for each query word, a hit list (list of document ids in which this term occurs) and the corresponding normalized TF scores are retrieved from the database, and the IDF scores are computed based on this. Then, the intersection of all hit lists for all terms is found and this is the set of candidate documents that need to be ranked. Additionally, the query weight score (TF * IDF) for each term is found. Then, an ExecutorService thread pool with 20 threads is created, the candidate documents list is split into segments, and each segment is passed off to each thread. The Java Callable task submitted to each of these threads is an instance of *ScoreCalculator*. In *ScoreCalculator*, if the segment of documents passed in exceeds the limit of 1500, only the first 1500 will be taken for processing and score computation. Then, a set of BatchRead requests (for every document) each of size 100 are created and made to DynamoDB, and upon receiving these, the thread will extract the page ranks and store them in an in-memory hash map. Then, for each document, its cosine similarity with the query vector is calculated by summing up the products of query term weight, normalized TF, and IDF for each term. Then, a weighted average of this cosine similarity score and the page rank for the document is computed and added to a list. Once all scores have been computed for all assigned documents, the thread returns this list. Back to the main handler in *DocumentRetrieval*, a custom Comparator is used to aggregate the results from each thread together and sort in descending order. The first 150 are inserted into the cache table, and then based on the requested page number, these results are indexed into and converted to a JSON-translatable string and returned.

Issues & Difficulties: Initially, the search engine used just a single thread to retrieve all the data for each candidate document and perform all the score computations. However, as the number of documents indexed grew larger and more "general" queries were inputted by the user, the performance of this single-threaded engine significantly slowed down as it had to handle a much larger set of documents, and this in turn worsened the user experience. So, a multi-threaded approach was taken and a thread pool was created, where each thread was responsible for computations for a certain subset of the documents. This noticeably helped the search times as many of the computation heavy operations were happening in parallel. One challenge to this was finding the optimal number of threads in order to achieve the balance between full parallelism and thrashing with diminishing returns. The thread pool started out with 5 threads and was incremented up till 20, however it was found that a thread pool greater than 20 did not help with performance at all. Another issue that was encountered, even with multiple threads, was that each thread was responsible for processing too many documents. In the Google paper, it was mentioned that in order to put a limit on response time, only a certain number of documents, specifically 40,000, of the total set of candidate documents were even considered. This optimization was emulated in our search engine, and a cap of 1500 documents was established for each thread, for a total limit of 1500*20 = 30,000. Now, going back to the cosine similarity mentioned above, the traditional formula for this is as follows:

$$sim(d_j, q) = cos\theta = \frac{\sum_{i=1}^{t} w_{i,j} \quad w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} \quad \sqrt{\sum_{i=1}^{t} w_{i,q}^2}}$$

However it is clear that the calculation of the normalization factor in the denominator is extremely expensive as it requires a full scan of the inverted index for every document in question. In this paper, multiple approximations to this formula were presented, and Method 3 was chosen: $sim(d_j, q) = cos\theta = \sum_{i=1}^{t} w_{i,j} \quad w_{i,q}$. Dropping the normalization factor was found to result in the fastest query times with not much sacrifice of quality rankings for our search engine. The last difficulty is regarding the set of candidate documents. Originally this search engine simply used the *union* of hit lists, resulting in a much larger set of potential documents, many of which matched on only one or two of the query terms, having no significance in relation to the full query. So, like said above, the *intersection* of hit lists of each term was used to be the candidate document list, and this provided a much richer set of relevant documents and did *not* result in too few matches, as initially expected.

More Optimizations: Stop words don't have any critical meaning to the query and simply increase the number of documents that need to be handled and sorted, significantly slowing down search times, therefore stop words were filtered out before any processing was done to minimize such extraneous calculations. Finally, keeping in mind scalability and the potential for this system to be used by multiple, concurrent users, a search cache table was created in DynamoDB. As shown in the evaluation section below, this significantly speeds up future queries by avoiding performing the same computations and sorting all over again. Given that users most likely search up many of the same popular queries (for example, the president of the United States), it is important to take advantage of the first time this heavy lifting is done for any particular query and store it for the future.

4.5. Extra Credit Features

4.5.1. Amazon Shopping Search Results

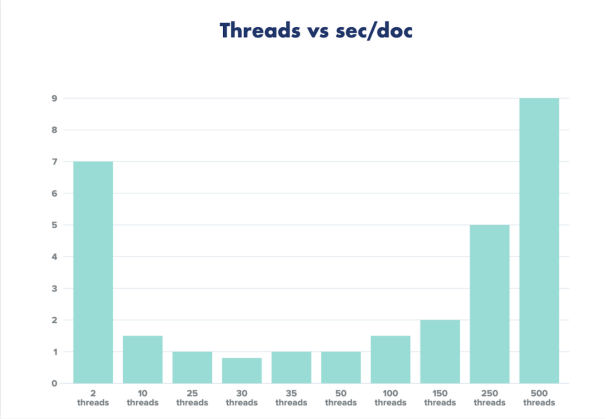
Rainforest, a real-time product data API, is used in order to retrieve shopping search results from Amazon for every query inputted by the user. Specifically, to the "/request" path, the API key and search term are passed in. The JSON response is parsed and the top 12 results, along with their titles, urls, images, and prices, are retrieved and then displayed to the user in the right-hand panel of the interface.

4.5.2. Google-Style Spell Checker

If the results for a search query are low (less than 10), the spell checker route is invoked. The spell checker computes the *Levenshtein* distance between the source word and words in our corpus of valid English words until it finds the closest match. The corpus is a Google corpus of 10,000 most frequently used words in the English language, available here. Some of the words in that corpus are nonsensical unfortunately which can rarely cause the spellchecker to return an invalid result. However, in general the spellchecker returns accurate results and has an advantage of sorting its corpus by frequency of use, letting me return the most common (most frequently used) matching English word.

5. Evaluation

5.1. Crawler



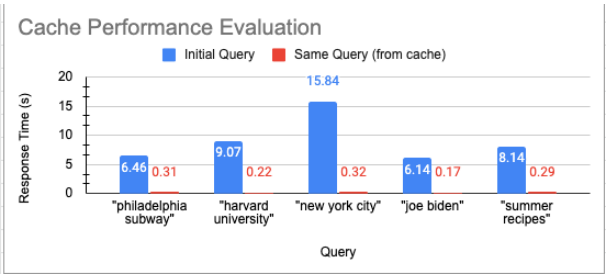
On average, we ran three separate EC2 nodes for the crawler. Which approximately tripled the crawler speed. In the end, we crawled 600k documents at a rate of 2-3 documents/second.

5.2. Indexer

We ran 2 EC2 instances for the indexer, each indexed 1 document/second (note that the spout sleeps for 1 sec after emitting each document to prevent the storm queue from filling up too quickly, see the previous section for detailed explanation). Running two instances for 14 hours a day, it was able to index 100k documents per day. We indexed around 200k documents total. In each indexer instance, there's one document spout (*DocSpout*) executor, and 4 executors for each subsequent bolts. We found that increasing the number of bolt executors didn't not significantly boost indexer speed, but increased memory load.

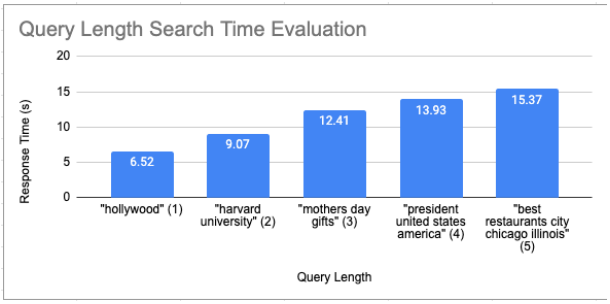
5.3. Search Engine

5.3.1. Cache Performance Evaluation



This analysis of speedups due to the cache is modeled after the 5.3 Search Performance section in the Google paper found here.

5.3.2. Query Length Search Time Evaluation



Intuitively, longer queries match on a larger set of candidate documents, which subsequently increases the load on each thread (for a fixed number of threads) increasing the overall latency of the system and its response time to the user.

6. Conclusions

In this project, we successfully designed and developed a distributed Web indexer & crawler, as well as performed link analysis using the PageRank algorithm. All three of these components were used as the basis of a Google-style search engine, which allows users to input queries and view ranked & relevant results in a matter of seconds. Building this system taught us a great deal about scalability and efficiency, as well as how to create a robust solution that can handle the complex, and often messy, structure of the Web. Additionally, this project certainly emphasized the importance of defining clear and consistent interfaces between each component, such as database schemas, before implementation begins as this avoids many integration issues later on.

For the future, there are definitely more enhancements we would like to make to our search engine system to improve its performance and output even higher quality results. One such improvement is instead of indexing only single terms, our indexer could also index phrases of 2-3 adjacent terms. For example, if a query was "new york city", storing document matches in the inverted index which contain these three terms contiguously, rather than separately, would provide more relevant search results since this would filter out documents that simply contain each of the individual terms separately and have very little semantic proximity to the query. Another enhancement would be to integrate factors like font, capitalization, positions, types of text (anchor, title), etc into the ranking function, like mentioned in the Google paper. Instead of simply treating every term's contribution to a given document's context meaning equally, the search results would be significantly enriched if there were some metrics to measure this by. For example, terms that are in bolded font and towards the beginning of a document are most likely a better indicator of a document's semantics rather than small text at the bottom of a document. Finally, the user experience with this search engine could be improved by having some sort of a feedback system where users judge how good or bad search results are for a particular query. This feedback could be stored and then used to re-rank the same results in the future. So, instead of relying on solely fixed features of documents and their terms, having a subjective aspect of user opinions incorporated into the ranking functionality could further improve this system.

7. Acknowledgements

We would like to thank Professor Zachary Ives, as well as the teaching assistants, for all of their guidance and instruction throughout this semester.

8. Appendix

The code repository for this project can be found here.

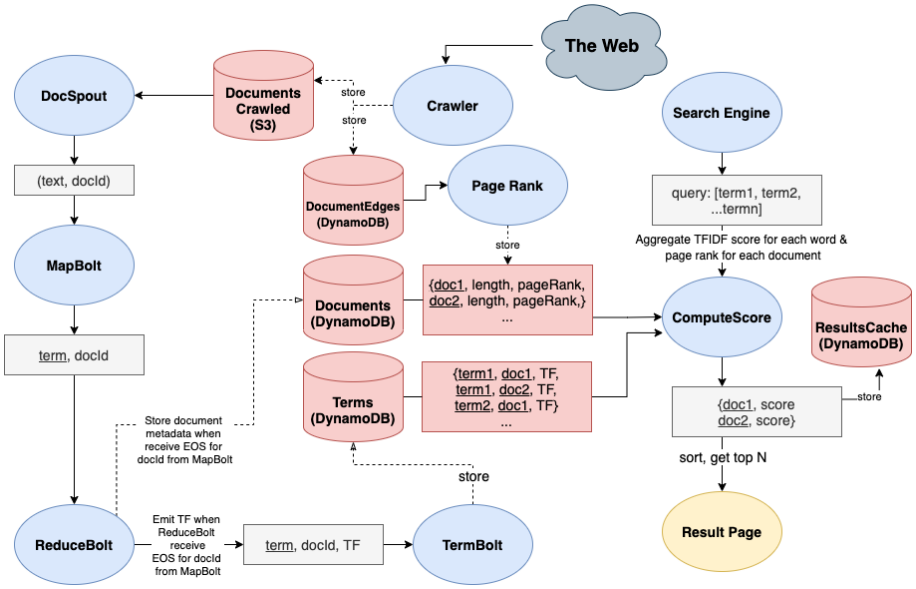


Figure 1. Project Architecture Diagram

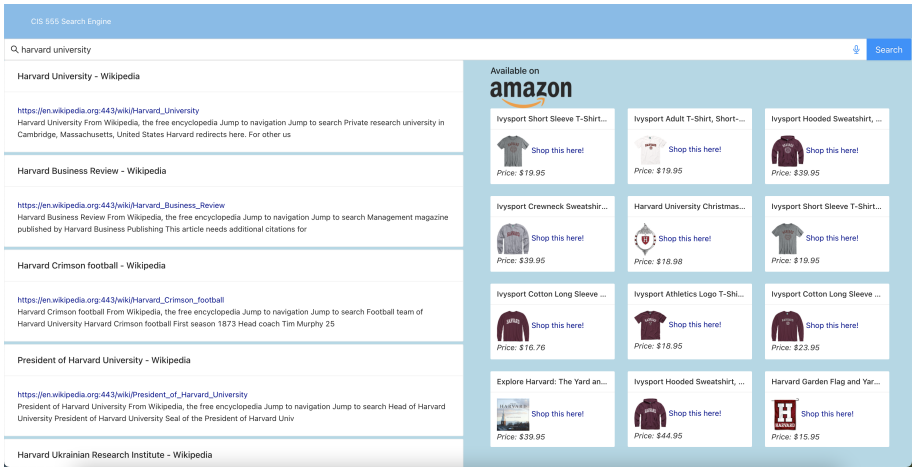


Figure 2. Screenshot of Search Engine in Action