Walmart-Brand Google
CIS 455/555 Final Project Report Spring 2021

Canvas Group Name: walmart-brand google
Repo: https://bitbucket.org/kevc528/final-project/src/master/

Edward Kim (kime022@seas.upenn.edu) | Maxwell Du (maxdu@seas.upenn.edu)
Kevin Chen (kevc528@seas.upenn.edu) | Andrew Zhao (anzhao@seas.upenn.edu)

Introduction

Project Goals

Our main goal for this project is to combine what we have learned throughout the semester and create a very robust distributed search engine composing of the crawler, indexer, ranking algorithms, and user-facing functionality. We want the user to be able to enter any query and receive the most relevant matches.

High-Level Approach

For our project, we plan to use AWS technologies including RDS for the relational database, EMR for running spark jobs, and EC2 for hosting. Additionally, for many components of our project, like the crawler, indexer, and ranking, we plan to take inspiration from research done in the past, like the Mercator crawler design and Google's PageRank. We plan to meet two or more times a week to discuss what we have done so far, see if anyone needs assistance, and conduct basic integration testing.

Milestones

4-20: UI mostly Complete; RDS, EMR, EC2 setup/planned; Crawler: Migrate Berkeley DB store to RDS
4-27: Start Integration; UI complete; Crawler complete; Have most of Indexer complete; Have most of ranking algorithm implemented
5-4: Finish integration, testing, documentation, and hosting; Patch in any final changes; Experimental data analysis

Division of Labor

2.1: Distributed Crawler: Kevin
2.2: Indexer: Maxwell
2.3: PageRank: Eddie
2.4: Search Engine: Andrew, Maxwell
2.4: Web User Interface: Eddie
3: Experimental Analysis and Final Report: Everybody
Hosting/DB Setup: Everybody
Miscellaneous Tasks: Andrew

Project Architecture

Overall Architecture

Before deploying the Search Engine, we first had to run a multi-worker/multi-server web crawler that utilized local Berkeley DB storage and wrote to an AWS RDS Postgres instance. The web crawler stores URL edges within RDS as well as Document Content. After the Crawler completes, the Pagerank algorithm hosted on EC2 is run and it writes the results back to RDS. The Indexer is also run on an EMR cluster, storing TF/IDF scores and term weights into RDS. Afterwards the Search Engine server and Web UI are deployed on an EC2 instance. When the user submits a query via the UI, a GET request is sent to the Search Engine server along with a query string in the headers. The server will parse the query string, removing stop words, then queries the database for

the most relevant documents using the tables the Indexer created. Afterwards, Pagerank and various bonuses are applied to these top documents to determine the order in which they are returned to the user.

Database

Overview

The project relied on a Postgres instance hosted on AWS RDS. Note that we are able to store documents directly in the PostgreSQL database as the text field supports up to 1GB of content, which is much less than the crawl limit of 1MB we set. The crawler content is in a separate table from the URLs because we don't want the large amounts of content to slow down other operations that use only URLs.

Table Schemas

urls(*id*: serial, *url*: character varying(2048))
- *url* NOT NULL
- CHECK CONSTRAINT UNIQUE(*url*)

crawler_docs(*id*: integer, *content*: text, *type*: character varying(2048))
- *id* Foreign Key urls(*id*)
- *content, type* NOT NULL

crawler_preview(*id*: integer, *title*: text, *headers*: text, *preview*: text)
- *id* Foreign Key urls(*id*)
- *title, content* NOT NULL

links_url(*source*: character varying(2048), *dest*: character varying(2048))
- *source* Foreign Key urls(*url*)
- *dest* Foreign Key urls(*url*)

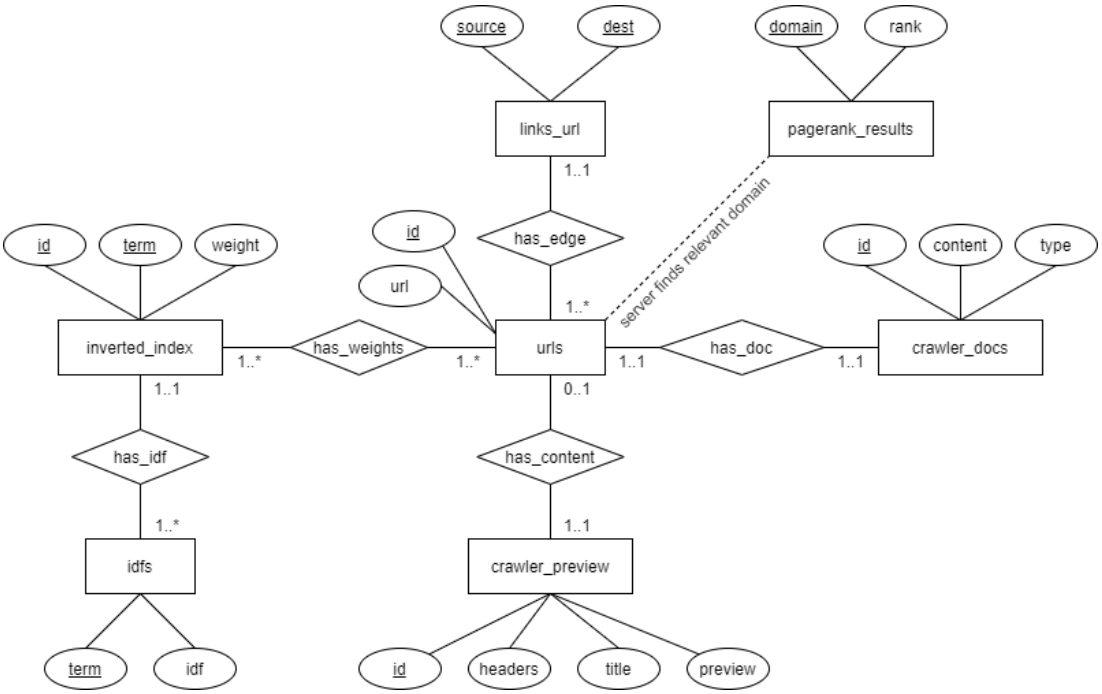
domains(*source*: character varying(2048), *dest*: character varying(2048))

pagerank_results(*domain*: character varying(2048), *rank*: double precision)
- *rank* NOT NULL

idfs(*term*: character varying, *idf*: double precision)
- *idf* NOT NULL

inverted_index(*term*: character varying, *id*: integer, *weight:* double precision)
- *term* Foreign Key idfs(*term*)
- *id* Foreign Key urls(*id*)
- *weight* NOT NULL
-

ER Diagram



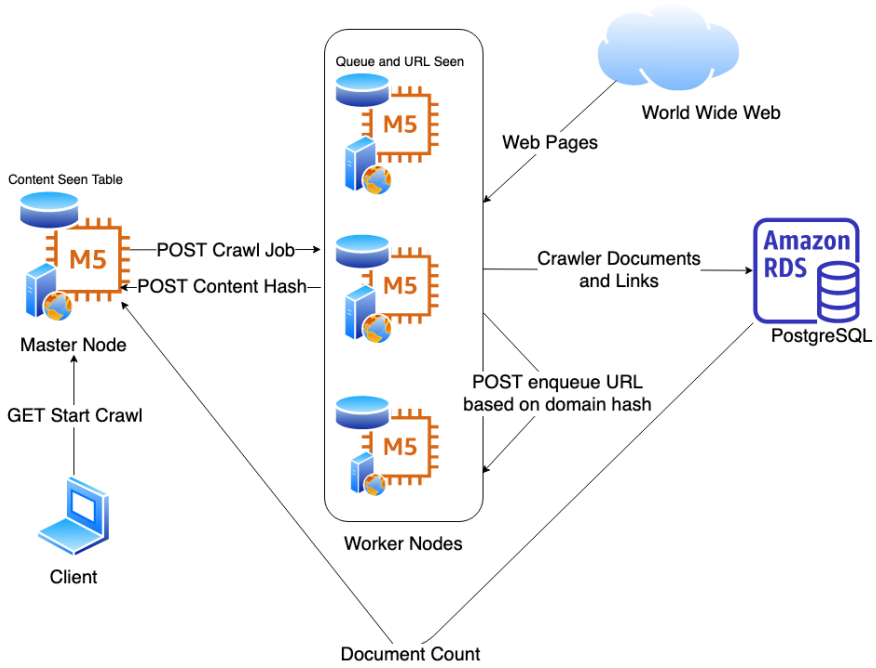
Although “domains” is a table, it is treated more as an intermediate table in order to retrieve the “pagerank_results” table and thus will not be included in the ER diagram (more on domains table in Pagerank section).

Table Sizes

urls: 357,130 tuples
crawler_docs: 357,130 tuples
crawler_preview: 356,804 tuples
links_url: 76,226,552 tuples
domains: 170,580 tuples
pagerank_results: 11,543 tuples
idfs: 5,581,280 tuples
inverted_index: 172,062,620 tuples

Implementation

Crawler



Overview

Our crawler takes the HW2 StormLite implementation and extends it by distributing the work among multiple nodes. For the architecture of the crawler, we took inspiration from the Mercator paper. This crawler makes use of both local storage on nodes through BerkeleyDB and cloud storage on AWS Relational Database Service (RDS). Overall, the architecture involves a master node as a “controller” and interface for the client, and multiple worker nodes that do the bulk of the work for the crawl. These nodes communicate with each other through HTTP via Spark Java servers. The crawler is hosted on three EC2 instances, with one instance running a master node and workers, and the other two instances each running only workers.

The Master

The master node is responsible for controlling the crawl: when it starts, when it ends, and keeping track of its progress. It provides an interface for the client to start crawls and terminate crawls through HTTP requests. The

master is responsible for sending crawl configurations and termination requests to the workers that have established a connection with it. Additionally, there is a background thread running every few minutes to check whether we have reached our target corpus size, in which the master will send the termination request to all workers.

Additionally, the master node is responsible for ensuring that crawled content is not duplicated. To do this, there is an on disk BerkeleyDB store of hashes for document contents. Before inserting a document into the RDS database, each worker must check with the master node to ensure that their document contents are not duplicates of another document.

The Workers

The worker nodes are responsible for doing the actual "crawling" of the web. The worker follows much of the StormLite implementation of the crawl task from HW2. There are three main parts of the storm topology: the queue spout, document fetch bolt, and link extractor bolt.

First there is a queue spout, which pulls a URL from the URL frontier for processing. Details of the queue will be presented in the next section, but we can assume that URLs we pull from the queue follow the robots exclusion standard.

Next, there is a document fetch bolt. This bolt will take the URL returned by the spout and first make a HEAD request. After this request, it will check the content type and content length headers to ensure that it is an HTML or XML document with a size of less than the specified maximum size from the configuration. Next, this bolt will check the Last-Modified header and ensure that if the URL has been seen and hasn't been modified since the last crawl, that it will pull the locally stored version instead of making a GET request for the content. After these checks have passed, the bolt makes a GET request for the document contents. Before inserting this document in the database, it will send the hashed contents to the master node to ensure that the document hasn't been seen before under another URL. Finally, the document is inserted into the RDS database and passed on to the link extractor bolt.

The link extractor bolt is responsible for extracting hyperlinks from HTML documents and adding them to the queue to be crawled next. This bolt uses JSoup to parse the HTML and extract the links. After extracting the links, the bolt will send these extracted links to the URL queue and also add them into the links table in RDS for pagerank.

The Queue

The queue, or URL frontier, is actually a collection of queues, where each worker has their own queue. To determine what queue a URL goes to, the crawler considers the hash of the domain. This sharded queue makes it easier to enforce the robots exclusion standard, including disallowed URLs and crawl delays. In this scheme, each worker will only have to manage the domains that hash to it instead of every domain. To make routing easier, each worker will utilize the WorkerRouter class that will hash the domain of the URL and send a request to the corresponding worker. Note, that in the case of inter-domain redirects, the document fetch bolt will route the redirected URL to the correct worker queue.

As for the actual mechanics of the queue, each workers' queue is stored on disk through BerkeleyDB and each queue has its own domain manager. The BerkeleyDB implementation stores the URL with an "earliest possible request date." This date is the earliest possible date to send the next request based on a domain's robots.txt crawl delay. Then as we take them from the queue, we take URLs from earliest date to latest date. This way we are getting a BFS-like traversal using a FIFO-like queue, where earlier URLs inserted in the queue will be extracted first as long as the crawl delay (if exists) has passed.

The domain manager is responsible for fetching the robots.txt for each domain that is present in the URL frontier and enforcing crawl delays and disallowed paths. It plays an important part for filtering which URLs can be added to the queue and what the "earliest possible request date" is. This information is stored in memory since we can assume that given our crawl size of less than a million, we have enough storage in memory to disallowed paths. Additionally, the domain manager keeps track of the last request to that domain so we can enforce crawl delays. If we extract a URL and the crawl delay for its domain hasn't elapsed yet, we insert it back into the queue with a new "earliest possible request date" based on the remaining crawl delay.

Storage, Scalability, and Crash Recovery

The crawler storage involves both BerkeleyDB and an AWS RDS cloud database. The distinction between what is stored in BerkeleyDB and RDS involves whether other parts of the search engine require access to information and the size of what we are storing. It is well known that cloud databases are slower than on disk storage because of the connections involved to connect with a cloud database and execute queries.

In my implementation, the reason for storing the URL queue, URLs seen, and content seen on disk is two fold: to minimize read and write speeds, especially with individual reads and writes, and also to provide support for crash recovery.

The URL queue is fairly small, at most a few million URLs, only used for the crawler, and requires frequent reads and writes, so storing this in BerkeleyDB on disk is more optimal. For similar reasons, we chose to store content seen on disk in the master node. We also store URLs seen on disk because each worker manages a subset of URLs, making it easier and faster to only check URLs that the worker manages. The alternative is checking all URLs crawled in RDS, which involves looking through a much larger set of URLs. We know that if there are two identical URLs, they must also have the same domain and hash to the same worker so checking the URLs seen table in the worker will suffice.

As for crash recovery, we opt to store crawler state, like the queue, URLs seen, and content seen on disk instead of in memory because if our crawler crashes, we can easily pick up from where we left off and continue crawling. And if instances crash, we can use EBS volumes to preserve state and continue our crawl. Note that we don't store domain manager information on disk because we can easily refetch robots.txt files as we process URLs from different domains after a crash. This is because it is a state that remains fairly constant throughout our crawl. To track worker status, there is a GET "/alive" route that returns the last time the worker added documents to the database. Additionally, if any worker StormLite cluster dies, a GET request to "/restart" can be used to restart the StormLite cluster and pick up from the state that has been preserved on disk.

Whenever we do have to write to AWS RDS, the crawler makes sure to use batch writes. This is because the initial connection to the database is expensive and doing this for every single write would be inefficient. Instead we write documents and links in batches and use the same connection. And since database writes can take some time and we don't want to block the main execution of the crawler, we utilize a thread pool to allocate threads for RDS writes.

Indexer

The indexer, implemented in Java, parses all the documents stored in the "crawler_docs" table by the crawler. By running Apache Spark on an Elastic MapReduce cluster, the indexer performs two jobs: First, it creates an record-level inverted index that maps terms to the document ID they belong to. While doing so, it counts the frequency of each term within the document to compute the term-frequency (TF) score for each (term, document ID) pair, along with the number of documents containing each term (DF). The second allows the indexer to compute the IDF of each term, which it stores in a second table named "idfs." Using the IDF scores for each term, the indexer then calculates the TF-IDF for each term within a specific document, and this final weight is stored in the "inverted_index" table. From the 360,000 total documents within our corpus, the "inverted_index" table contains 170,000,000 tuples of form (term, id, weight), while the "idfs" table contains 5,500,000 tuples of form (term, idf).

At a lower level, the indexer uses Spark SQL and JDBC to connect to our Postgres database and read/write data. For each (term, document ID) = (i, j) pair, the TF score is calculated using tf(i, j) = a + (1 - a) * freq(i, j) / max_freq(j), where a = 0.5, and max_freq(j) is the frequency of the term that occurs the most within the document. For each distinct term, IDF is calculated using idf(i) = log(N / n_i + 1), where N is the total number of documents in the corpus and n_i is the number of documents the term appears in. Using these numbers, the final weight for each term within a document is calculated using tf(i, j) * idf(i).

The terms themselves are obtained by splitting the document by whitespace and punctuation. They are then regularized to lower-case, trimmed, and stemmed using a Snowball Stemmer, which is a slightly improved version of the Porter Stemmer. All non-ASCII characters are also removed. If the term does not belong to a

600-word list of stopwords, such as “almost” or “those,” and the term is less than 50 characters, we include it into our tables and proceed with the above computations.

Pagerank

The Pagerank algorithm follows the Random Surfer Model and is implemented in Python in a Jupyter Notebook and utilizes PySpark SQL in order to query and process the data. Two notebooks had to be created in order for Pagerank to be run, the first cleans the URL edges and the second actually runs the iterative Pagerank algorithm. We deployed both notebooks on a single AWS EC2 instance.

As the crawler runs, it adds URL “edges” into a table in our Postgres RDS instance called “links_url”. The table contains two columns, which together create the primary key: “source” and “dest”. The “source” attribute is the URL of a crawled page and the “dest” attribute is an outgoing URL from the page. The “links_url” table is essentially an edge list representing a graph of a tiny subset of the web.

The cleaning script takes the “links_url” table and stores it into a Spark Dataframe. From there we created a user defined PySpark SQL function that extracts the domain information from the URL from both “source” and “dest”. Here, for increased specificity, the extracted domain differentiates between “cis.upenn.edu” vs “upenn.edu.” Afterwards we only keep the unique edges from domain to domain and store the results in RDS in the “domains” table. Our initial plan was to run Pagerank on just the “links_url” table, so URL edges, however, we pivoted to run Pagerank on domain edges instead. The crawler was already running when we decided on this, so the cleaning script was necessary. To handle hogs, we removed all self loops (same domain to domain). To handle sinks, for every sink s with incoming edges E, for each edge (u,v) in E, we added an edge (v,u). Additionally, we followed the PageRank algorithm from class by adding a decay factor $\alpha = 0.85$ and $\beta = 0.15$.

The Pagerank notebook first queries the “domains” table and then loads the results into a spark dataframe. Afterwards we calculate the amount of rank each node will give to each neighboring node and store it as a tempview. Finally, we run the actual iterative Pagerank algorithm and JOIN between the edge list, weights, and previous iteration’s rank results in order to get the pagerank results of the current iteration. The script will terminate at 100 iterations or if the maximum difference of ranks between consecutive iterations is less than 0.1. We found that the algorithm ran for about 29 iterations.

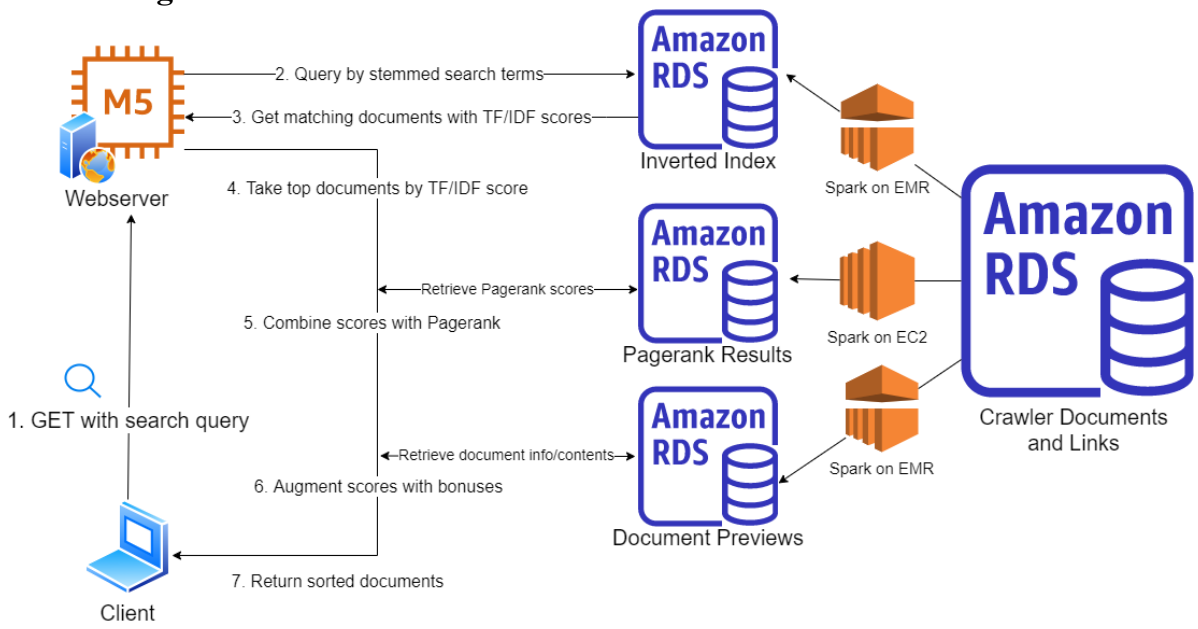
Before writing back to RDS into the “pagerank_results” table, we normalized the ranks, because we saw the results would range anywhere from a number less than 1/100 to a number greater than 500. Having the ranks range from [0,1] inclusive also helped with weighting during actual queries.

Document Previews

In addition to the “crawler_docs” table, which contains document IDs and their corresponding full HTML content, we have an additional table called “crawler_preview” which stores document IDs to a document’s **title, headers,** and **preview**. The title column is simply the webpage’s title, and the headers column contains all text within <h1> and <h2> tags. The preview column contains the text within the first few <p> tags of the document. An Apache Spark job on EMR constructs the table, and uses JSoup to parse the necessary information of each document.

The primary goal of this table is to allow the search engine to efficiently retrieve a document’s information to display on the frontend, without having to use JSoup to parse the entire document content each time a query is performed. Another equally important reason for the table is to allow our search engine to “bonus” a document’s ranking by key components within its HTML, similar to how Google rewarded text with larger font sizes in their early search engine. We’ll talk about this more in the next section.

Search Engine & Web User Interface



The search engine is a Spark Java application which consists of an API with a single command: /search. The values to be searched are represented as a query parameter in the URL. Once received, compiling the results takes 3 major steps. The first step is to obtain the relevant documents by querying the TF/IDF tables, and then perform a preliminary ranking using cosine similarity. To do this, the server first normalizes the search terms in the identical way that we normalized terms to store into our inverted index. It then sequentially assigns an index to these search terms, which is used for creating vectors for the query and documents.

Using Spark SQL (in Java), we query the "inverted_index" table by these search terms to obtain a dataframe of (term, document ID, weight) tuples, which we convert into a JavaPairRDD of (document IDs, vector) pairs using a series of map and groupby operations. Each of these vectors, which are represented as arrays, contain the document weight of each term at the index we assigned the term previously. The query is also converted into a vector, which similarly contains the weight of each search term (calculated using the frequency of the term within the query and the IDF). As a result, we now have a vector for the query and vectors for all the matching documents, where each vector contains all the non-zero weights necessary for cosine similarity. Now, the documents are ranked by their vectors' cosine similarity scores with the query vector, which forms a preliminary ranking of the documents based solely on TF-IDF.

From here, our server continues with a pre-defined number (currently, 2000) of documents with the highest TF-IDF scores. To combine these scores with pagerank, we obtain the domain of its URL and query the "pagerank_results" table to retrieve their pageranks. For each document, we take the logarithmic of its pagerank and combine it with its TF-IDF score using a weighted sum, where the weights were determined through manual testing of the search results.

Finally, the TF-IDF and pagerank score may be further incremented by several "bonuses" that reward a webpage for being similar to the query at significant parts of its HTML content. This was inspired by the early Google model, which stored font information within documents to refine its search results. Using the "crawler_preview" table, we retrieve the title, headers, and preview of the documents. If the terms within a document's title or headers overlap with the terms of the query, a small bonus is added to the document's score (based on the number of overlapping terms). For each overlapping term, the bonus is scaled by the IDF of the term in order to further reward documents that share "rarer" terms with the query. A bonus is also applied if the entire search query is contained within the title or headers. This bonus is scaled by the length of the query, which follows the idea of "phrase search," where longer sequential word matches likely mean that a document is directly related to the query. To further support phrase search, but ensuring that this bonus is still applied if the query differs

slightly from phrases within the document, we also use an Apache Commons library to compute the **longest common subsequence** between the search query and document title/headers. If the longest common subsequence has a length that is at least 90% of the full query length, which means that the query is nearly identical to a phrase in the document, we apply an additional bonus. Finally, another bonus is applied if the document's preview contains the query. This is based on the reasoning that the first few sentences of a document likely contain key information about what the document is about.

We spent a few days tuning the search engine's parameters and determining the weights of TF-IDF, pagerank, and our custom bonuses to the overall score. Originally, we programmed pagerank as being too impactful, and webpages on popular domains such as Wikipedia would always overwhelm webpages with far more relevant content. For this reason, we limited the maximum pagerank of a website to be 0.1 to decrease the impact of the few highest-ranking domains, used the logarithmic of the pagerank instead, and lowered its weight factor until pagerank was a helpful, but not too overwhelming, component of our search results. Through much trial and error, we similarly tuned the weight of our additional bonuses until the top retrieved documents were generally relevant, but documents on unpopular domains and with low TF-IDF scores did not show up first due to simply having a title that matches the query. Once we struck a solid balance, we found that our bonuses significantly increased the performance of our search engine compared to using only TF-IDF and pagerank.

After applying the bonuses, we sort the pages by their final rankings, attach each document's URL, title, and preview, and send the results as a JSON to the Web UI. The Web UI itself consists of a simple search bar that looks similar to Google. When a user types in a search and sends the request, we will send the request to the search engine. Upon receiving the response, the urls will display from best matched to worst matched web page in a paginated fashion. The client side Web UI was created with React and utilizes the react-bulma-components library.

Deployment

There are two parts to our deployment: running the web server and running the crawler, indexer and pagerank tasks. For setting up the web server, we decided to use a single EC2 node to both host the query server and the web UI. In the EC2 node, we have the Spark java search engine running on port 45555, while the React web UI server runs on port 80. To communicate with the search engine, the React UI sends an AJAX request to localhost on 45555 to obtain the results. In regards to scalability, the amount of requests the server can process is dependent on how powerful the EC2 node is: if needed, we can scale up the EC2 to ensure smoother request processing.

Our approach to running the tasks (crawler, indexer) depends on the tasks on hand. We used EMR to run the indexer and the content creator as both programs were written using Apache Spark in Java. On the other hand, the crawler and the pagerank were run using EC2 instances. Scalability for these tasks largely depended on how powerful the instances were. For example, smaller EMR instances would often fail because there was not enough memory in the executors. In that case, we increased the memory usage to handle the additional data and tuned other Spark parameters, such as the number of partitions in our Dataframes.

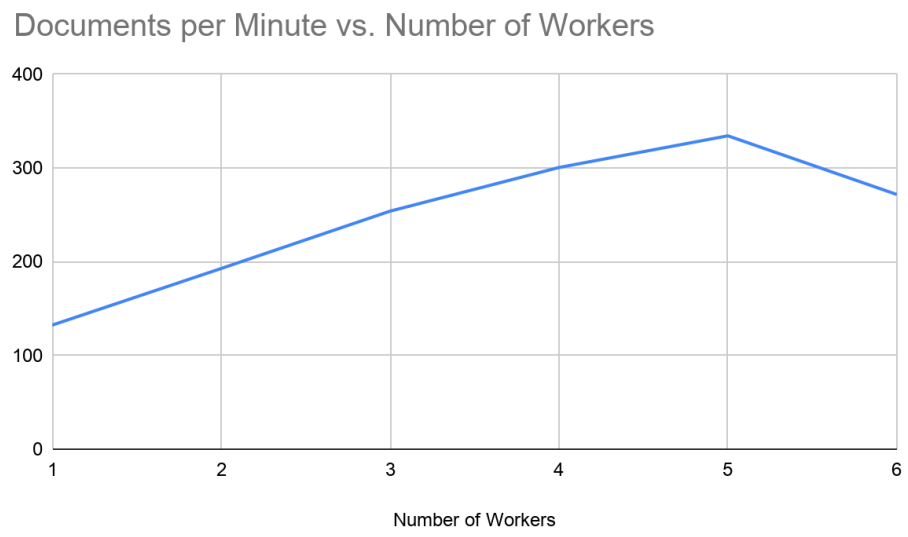
Evaluation

Crawler

For the crawler, we evaluated performance by running it on one EC2 instance, with different amounts of workers per node. This evaluation would be used to pick the optimal number of workers per EC2 instance.

To track performance, we ran a controlled experiment to track how many documents the crawler processes in a five minute window. We made sure to start at the same seed URLs and clear the database each run. Below are the results we found.

Number of Workers	Documents Crawled in 5 Minutes	Documents per Minute
1	662	132.4
2	964	192.8
3	1270	254
4	1503	300.6
5	1672	334.4
6	1359	271.8



From these figures, we observe that as we increase the number of workers, the rate we crawl increases until a certain point. Our peak is at 5 workers per instance. After this, we see that the crawl rate starts to decrease. This is likely due to the number of threads that each worker uses on the machine. With more and more workers requiring more and more threads, thrashing can occur. Thrashing is when processing time increases because the operating system keeps switching threads and it takes a long time until we switch back and make progress for each thread. In our crawler, not only does StormLite use threads, but Spark Java and database I/O also use thread pools. We can see that past 5 workers, the effects of thrashing can outweigh the benefits of using more workers. This experiment was really useful for helping us pick the correct number of workers to run on each EC2 instance to maximize crawler throughput.

Indexer

The indexer performed the heavy task of parsing all the documents in our corpus and writing over 170 million tuples into the database. Since we ran the indexer multiple times as we fixed bugs or as the crawler added documents, we were able to benchmark the performance of the indexer across a key parameter of Spark jobs: the number of partitions. Upon reading all documents in our corpus into a Spark Dataframe, which is the parent RDD of almost all the tasks in the Spark job, we repartitioned the Dataframe in attempts to improve the indexer's performance.

No repartition (default of 4 partitions): Never finished
200 partitions: 46 minutes
400 partitions: 69 minutes

800 partitions: Never finished (Not enough memory: too much memory overhead required in our driver program to manage the partitions)

Given that our EMR cluster could have been upgraded much further, such as by using more than 3 EC2 instances, these results signal that the indexer can be scalable with an increased number of documents as long as we find the right number of partitions. We also observed that the Spark job spends over half of its time writing to the RDS database, which tells us that it can likely finish in a much shorter time if we upgraded to an RDS database with greater performance or higher throughput.

Pagerank

The cleaning notebook had to process 76,226,552 tuples (URL to URL) and extract the domains of each url. After cleaning and taking care of hogs and sinks, there were about 185,989 Domain to Domain edges with 11,543 unique domains.

Number of Iterations Performed to Convergence: 29
Cleaning Notebook: 12 minutes average runtime
Pagerank Notebook: 5-8 minutes average runtime, 10-17 seconds per iteration

Search Engine

Here is a table with query results for chicken (one word only):

Concurrent Requests	Total Requests	Time (seconds)	Time individual request
1	1	6.8	6.8
5	5	13.708	2.742
5	10	22.493	2.249
10	10	22.55	2.255
10	20	37.554	1.877
15	15	22.395	1.493
15	30	52.922	1.764
20	20	37.878	1.893
20	40	68.383	1.709

Here is another table for: funny cat videos, a multi-word query,

Concurrent Requests	Total Requests	Time (seconds)	Time for individual request
1	1	10.831	10.831
5	5	26.235	5.246
5	10	41.619	4.161
10	10	40.175	4.017
10	20	66.421	3.321
15	15	66.021	4.401
15	30	98.44	3.281

20	20	66.135	3.306
20	40	121.176	3.029

All tests are done on an m2.small EC2 machine. We tested concurrent requests in multiples of 5, where the total number of requests is the same as the number of concurrent requests. The tests were done through running Apache Bench. There are a few details to note: multi-word requests take longer due to the additional processing requirements. Furthermore, additional concurrent requests make the average request time smaller, largely due to the server's thread pool and RDS caching abilities.

Conclusions

Given the duration of the project, we were really proud of what we were able to accomplish and what we were able to learn. We were able to take advantage of the knowledge we learned in this class and apply it to a project. Additionally, we worked really well together as a team and helped each other out profoundly when certain issues arose.

Overall, we are happy with our search engine, along with the performance of our crawler and appearance of our Web UI. We found that our search engine returns fair and relevant results for a variety of search queries across sports, news, politics, and more. However, if this was a longer-term project, or if we had a chance to redo the project based on what we've learned throughout the process, there are a few areas where we could have improved the search engine further.

What We Could Have Done Better

Crawler

Given more time, the crawler could have benefitted from better monitoring, better document filtering, a change in how links are written in the database, and support for more content types.

First of all, better monitoring could help with determining the specific causes to errors or slowdowns in the crawler. Currently, the only monitoring in the crawler is a route that returns when a worker last wrote to a database. This could be improved by providing a list of what URL each executor on a node is running and what step it is on (fetching, extracting links, etc).

Another improvement that could be made to how domain links are written to the database. Currently, the crawler inserts the full URLs into the links table, but it was later determined that pagerank works best on URL domains. However, this was after the crawler had inserted many documents into the database and there wasn't time to recrawl. Instead, this was handled through a cleaning script. But, it is clear that the best way to handle this is to have the crawler insert the CNAME as endpoints of edges that we use for pagerank.

Indexer

Although we mapped document urls to IDs, we made the choice to not do so for our inverted index, where we stored the terms themselves as primary keys. Given that we limit the length of the terms in the inverted index, and that many of these terms are just a few characters, we initially made this choice since we predicted that the difference in query speed would be negligible compared to querying by integers, and that it may even be faster since it would not require an additional query into a separate table to obtain the terms' IDs. However, this may have been a small mistake, and using IDs instead could have improved our search engine's efficiency. Although our search currently does not take too long (a couple of seconds), we are very curious about the potential performance benefits of reconstructing our logic to map terms to distinct IDs (in a lexicon) and querying the inverted index by these IDs instead.