Google Search - CIS 555 Edition

Jack Goettle JGOETTLE@SEAS.UPENN.EDU

John Wallison JWALLI@SEAS.UPENN.EDU

Max Deng DENGMAX@SEAS.UPENN.EDU

Zimeng Zhou ZHOUZIM@SEAS.UPENN.EDU

1. Introduction

For our CIS 555 term project, we built a Google Search clone. This project leveraged many of the skills we have honed in on and design paradigms we have discussed over the course of the semester - distributed and parallel processing, multithreaded design, stream processing, efficient caching and indexing, and master-worker relationships. Our work culminated in the completion of a lightning fast crawler, carefully crafted indexer and PageRank, and a search engine which combined the previous elements, allowing the end user to interact with our system through an intuitive and easy-to-use web interface.

In this paper, we will discuss how we approached the task of designing an implementing a search engine, how we split up responsibilities among the four of us, the details of our project architecture, and a performance evaluation of our system. Link to repository. Link to live version (will not work if instance isn't running).

1.1. High-level Approach

We began by dividing up work between the four of us, largely according to the four main components of the project. Our first step was to get a working implementation of a crawler so that we could have a sufficient number of documents to test our other components. Then, we began work on the indexer and PageRank while keeping in mind what interface methods we would need to expose to the search engine. The API design between our components required frequent team meetings and communication, and we did not dive too deeply into implementation until we had a solid idea on how everything would fit together.

1.2. Project Goals

We had a few goals for each component of our project. For the web crawler, it had to be fast, robust, and polite. For the indexer and PageRank, they needed to be fast and expose interface methods to our search engine that wouldn't require expensive background computation. For our search engine and web interface, they needed to be fast and designed intuitively for the end user. And most importantly, at a high-level, everything needed to fit together seamlessly in order to avoid code refactorization or expensive computation during querying.

1.3. Milestones

11/29/21

Crawler has working basics complete; indexed 100k docs so that we can test other components. Require determining good seed URLs. Docs were not of the highest quality (some non-english). Documents were stored in AWS. Indexer: rudimentary lexicon and inverted index complete, and can be run on the first docs obtained by the crawler. At this point, the indexer can do a simple calculation based on TF/IDF, but will be extensible to add more features (including our PageRank scores) in the future.PageRank: Simplified PageRank Calculation on small corpus of fixed documents.Simple search interface, able to fetch documents from server via URL.

12/6/21

Crawler tuned to crawl for high-quality documents. Required creating some metrics as to measuring the quality of a website. PageRank: Support for dangling links and Random Surfer Model, basic integration with Indexer component. Indexer: persistent and distributed storage for the inverted index + lexicon. Incorporation of PageRank score into our calculation, as well as any relevant meta-data we crawled.Search interface with open source client side libraries. Designing /search route.

12/12/21

Crawler complete with 1.2M high-quality documents stored. Plenty of testing is complete to ensure documents can be retrieved effectively. Indexer: complete and able to run on the 1.2M documents retrieved. PageRank: Tuned, converges and fully integrated with system. Search interface complete.

12/15/21

Added server-side caching. Tuned interface methods for performance (ex. batch updates and reads). EC: added voice querying through interface. Deployed to EC2 and stress tested the system.

1.4. Division of Labor

We largely divided up labor according to the four main components of the project. Jack handled the crawler and the UI, John handled the indexer, Max handled PageRank, and Zimeng handled the search engine. We worked together closely on integration, testing, and deployment, meeting multiple times a week over the course of the development period.

2. Project Architecture

We modeled our architecture, shown in figure 1, very closely based off of that of Google in their original paper [1]. Since we give an overview of the implementation of each component of the project in the implementation section, we will use this space to give an overview of our database design and how the different components interact.

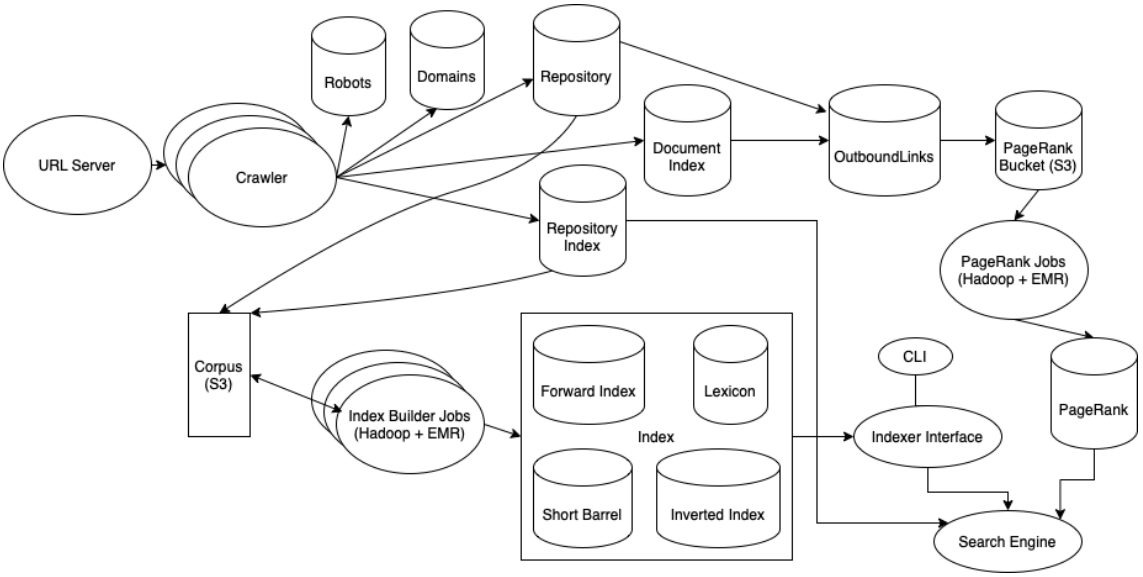

Figure 1. System Architecture

[1] http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/334.pdf

2.1. Crawler

Our crawler uses one URL server and multiple crawler workers. It stores document information in five different databases: Domains, Repository, and Repository index. The **Domains** table contains a mapping of domain names to the number documents indexed from that domain. We use this table the limit how many documents we crawl for each domain. The **Robots** table contains a mapping of domain to robots.txt information, such as the disallowed paths and the crawl delay. The **Repository** table contains a mapping of document ids (MD5 hash) to two values: the compressed document, and the time which the document was most recently crawled. The **Repository Index** table contains a mapping of document IDs to their title, description, and URL. We use this database to display relevant information for each document in the UI as well as to create the short barrel in the index. Lastly, the **Document Index** table contains a mapping of URLs to the following values: a list of document IDs associated with the URL, and the time which the URL was most recently crawled. We used this table to map URLs to documents, as well as to create the **OutBound links** table in S3 which we used for our MapReduce job.

2.2. PageRank

We used Elastic MapReduce to run a PageRank job on **OutBound links** table in S3 which contained a mapping of URLs to all of their outbound links. This job produced a **PageRank** table which contains a mapping of every URL to its PageRank.

2.3. Indexer

The index consists of 4 tables, and a set of static functionality. The **Lexicon** maintains all our word tokens and the number of documents they appeared in (the df). Our complete lexicon contains a little over 4 million words. The **ForwardIndex**, the intermediate step between parsing the documents and building the inverted index, maps document IDs to pairs consisting of words which appeared in the document, followed by that word's score. The **InvertedIndex** and **ShortBarrel**, built by processing the forward index, map words to a ranked list of documents in which they appear. In computing rank, our index distinguishes different types of hits: in addition to having different weights, the short barrel is built only on hits that occur in the title and description for a document, while the full index considers all hits. Finally the **WordMap**, originally designed as a separate data structure, is a set of static functions for identifying stopwords and invalid words, applying normalization, and Porter stemming. We make no distinction between words and word IDs: a given word's key is that same word in its normalized, stemmed form.

2.4. Search Engine

After parsing and normalizing a new query, the search engine will make calls to: our indexer interface to fetch the short or long barrel and compute an IR score, our Pagerank interface to grab PageRanks for the necessary documents, and the Repository Index to get the title and description for the the top search results.

3. Implementation

3.1. Crawler

We designed a fast and distributed crawling system using one URL server and a number of crawler workers, usually somewhere between 3 to 10. The URL server served lists of URLs to each crawler, and each crawler was built on top of our multithreaded StormLite framework. Our crawler consisted of one queue spout and multiple (generally = the number of threads) document fetcher, link extractor, and document store bolts, as shown in figure 1. We iterated heavily on this design to optimize performance, which will be discussed later in the paper.

Early on, we ran into the issue of getting stuck in a domain space and executing more of a DFS than a BFS on the web. This is bad because we were risking getting stuck in a spider trap on an evil website, and we also didn't want our entire document repository to consist of Wikipedia. To circumvent this issue, we wrote a function that would retrieve the domain name from a URL and we set a maximum on the number of documents we would crawl from any domain. We stored this information in a "Domains" database, which mapped domain name to a count of websites with that domain. This maximum number, however, varied according to the domain - trivially, crawling wikipedia.org should not be constrained to the same number of documents as jackgoettle.com. To overcome this, we collected data on the 200 most popular domains on the

internet [2] and subject them to higher maximum document counts, ranging from 20k to 60k, while any websites not in the top 200 were subject to a maximum of 5k documents.

To understand how our system worked, it is helpful to trace the "journey of a URL" through our system.

THE JOURNEY OF A URL

The journey of our URL begins in our URL server - either it is the seed URL, or it was enqueued previously by one of the worker nodes. Our URL will wait patiently in the URL server until it is the first element in the queue, and once this occurs, it will wait for a dequeue request from the queue spout of a worker node.

The queue spout will emit the URL to the document fetcher bolt if two conditions are true: the URL is not misformed, and the domain count for the URL does not exceed its maximum. The document fetcher will take a URL and do the following (in order): check if we have seen the URL on the current crawl (exit if yes), check if we have seen a robots.txt file for the domain and download it if not, satisfy a crawl delay if necessary, make a head request to the URL, retrieve the document from storage if it has not been modified, else download the document from the URL, update the "last crawled" times for the document and robots.txt file, and emit the document and some meta data.

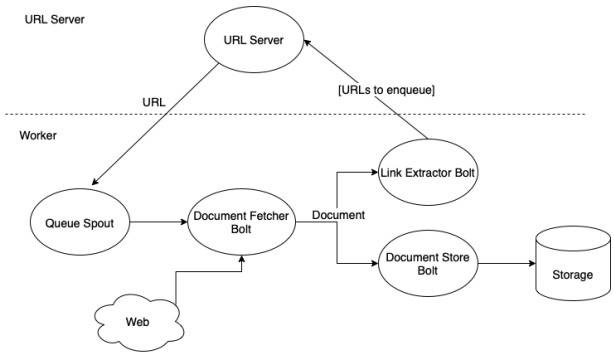

Figure 2. Crawler Design

The link extractor bolt will extract all outbound links from the document and send them to the URL server to enqueue. The document store bolt will store the document in our database. We found that separating these tasks into two bolts allowed us to speed up our system drastically.

3.2. Indexer

The indexer, at a high level, consists of three major components: the individual MapReduce jobs, which process documents, create the index, and store it; the database, which maintain the index; and the interface, which allows the other components to query the index. The various steps of the build process are accomplished by specialized MapReduce jobs.

BUILDING THE INDEX

The first indexing step iterates through each item in the corpus, extracting the title, description, and HTML body. Each item is decompressed, tokenized, and normalized by the WordMap. This first pass outputs lines $d_j - w_i - s_{i,j}$, where $s_{i,j}$ is the score of a particular word w_i in document d_j. For the full index, the score was computed by $s_{i,j} = 4n_t + 2n_d + n_b$, where n_t, n_d, and n_b are the number of title, description, and body hits, respectively. Short index scores are computed by $s_{i,j} = 4n_t + n_d$. The next steps, making the Lexicon and the Forward Index, both take their inputs from the initial forward pass. The MakeLexicon step simply outputs a pair $w_i - df_i$, yielding the document frequency of every word in our corpus. The MakeForwardIndex step groups by doc ID, and outputs $d_j - max_j - w_1 : s_{1,j} - w_2 : s_{2,j}...$ for each document. Here, max_j is the maximum score of any word in d_j, which is necessary to compute a normalized tf. Both a full and short forward index is built, while the Lexicon is only built once on the full corpus. Finally, the InvertedIndex is built by inverting each item of the forward index: each word is mapped to a list of documents in which it occurs, sorted in descended order of normalized term frequency ($s_{i,j}/max_j$), yielding lists of the form $w_i - d_1 - d_2 - d_3...$, which constitute our InvertedIndex and ShortBarrel.

DATABASE MAINTENANCE AND INTERFACE

Saving the tables to the cloud is accomplished by 4 parallel MapReduce jobs, which store items from S3 to their respective DynamoDB table. Once the index tables are saved, the indexer can be queried by the IndexerInterface, which provides

[2] https://moz.com/top500

specialized API intended for the search engine including: using the WordMap to parse user input, querying the Lexicon to build the query vector, querying the ForwardIndex to build document vectors, and querying the ShortBarrel and InvertedIndex to get matches. In addition, an indexer command line interface is provided for testing and debugging.

3.3. PageRank

Our PageRank implementation was written with Hadoop MapReduce. After crawling the documents we made a script to write link and outgoing links to DynamoDB. In this pre-processing step we added logic to remove self-loops and dangling links from the table as suggested by the PageRank paper. The PageRank implementation is split into four distinct jobs. The first job counts the number of lines in the partitioned file set. Then we run the second PageRank job which splits outgoing rank among outgoing links. In the mapper we also send back edges to handle sink nodes and emit rank from sink nodes. We run the PageRank job between two alternating output files until we reach a convergence of $1e-4$ and check this convergence by running a third difference checking job which outputs the absolute difference for each node which sends to the fourth maximum difference job every 10 iterations of PageRank.

3.4. Search Engine and Web Interface

The search engine was also closely modelled from Google's original paper. We first parse the query, which involves splitting the words, stripping the punctuation, converting to lowercase, removing stop words, and applying porter stemming. We then create a query vector, where each element is the inverted document frequency of a word in the query. We initially search through our short barrel to find the top 300 document ids for each word in the query. If we manage to retrieve at least 100 documents from the short barrel, we continue. These document ids will form the basis from which we do IR and page ranking. From all the document ids we collect from the short barrel, we create a document vector based on the normalized term frequencies between the document and query words. We calculate the normalized term frequency through the equation $a + (1-a) \times \frac{freq(i,j)}{max(freq(l,j))}$. We then rank each document based on the cosine similarity between the document vector and the query vector. We select the top 100 documents, and then find the corresponding PageRank values. After tinkering with the ranking function, we found the equation $IRScore \times (\frac{(pageRankScore - 8.269156e-07)}{7.688274e-06} - 0.1)^{2.5}$ to produce decent search results. In the fraction part of the equation we are normalizing the pageRankScore, by subtracting the mean PageRank score by the overall PageRank standard deviation. After ranking the top 100 documents by PageRank and IR score (cosine similarity), we fetch the document title, url, and description to send to the frontend. If we do not manage to query 100 documents from the short barrel, we do a search in the long barrel to fill up the documents queried by the short barrel until the total queried documents is at least 100. We then continue with the procedure as outlined above. Finally, after each query, we save the top 100 documents that were ranked in a DynamoDB table, so that for future queries that are the same we do not have to calculate the rankings again.

We built the frontend using create-react-app, where it makes the GET request "/search?q=bidenpage=1", where we have the query parameter "query" and "page", which specifies which page to retrieve. Each page contains 15 search results.

4. Evaluation

4.1. Crawler

We employed an extensive logging mechanism to evaluate the running time of every part of our system, with high granularity, so that we could spot bottlenecks and work to improve our system. We logged, across all instances, average time spent: idle, fetching a URL in the spout, checking document storage, checking robots.txt storage, downloading robots.txt, satisfying a crawl delay, making a head request, making a get request, retrieving a document from storage, downloading a document from the web, putting a document in storage, and parsing a document for outbound links. We stored information about each crawler run, such as the number of documents indexed, average crawl time, number of workers, and number of threads in DynamoDB so we could benchmark our performance.

We deployed our crawler to EC2, with each instance having one URL server and a varying number of worker nodes. **At peak performance, we were able to crawl roughly 1500 documents per minute using one URL server and 10 worker nodes, each with 5 threads.** We found that this number scaled linearly with the number of EC2 instances we were running - with 4 EC2 instances, we were able to crawl 6000 documents/min. We did not try parallelizing any further due to cost concerns.

Initially, the bottleneck in our system was the cost of storing documents in the database. We solved this issue by increasing read and write capacity in DynamoDB. The bottleneck then became the process of extracting links and sending them to the URL server to enqueue. We fixed by doing two things: separating putting the document in the database and parsing the document into two bolts so they could run in parallel, and making the enqueue request to the master a batch request, rather than sending over the URLs individually. These weren't the only optimizations we made while crawling the web - the design was iterative, and thanks to our extensive logging mechanism, we were able to make actionable observations and drastically improve our system.

4.2. Indexer

The MapReduce jobs to generate the index were run on Amazon EMR. With 2-4 nodes, the initial pass through the 68GB corpus completed in 1h16m, and from there building the Lexicon, ForwardIndex, and InvertedIndex took under 15m each. The ShortBarrel, with its much smaller size, was built locally on Hadoop in under an hour, then pushed to DynamoDB.

4.3. PageRank

In total we processed 1,209,728 nodes which took up more than 15gb split across three output files. On the full set of data with a convergence threshold of $1e-4$ it took 22 hours to fully converge with autoscaling enabled on Amazon EMR.

4.4. Overall System

We iteratively tuned our search engine for speed and quality of results, making a few modifications to our APIs to allow for batch reads and updates. Furthermore, the number of documents initially fetched to be ranked (300 per query word), as well as the number of documents which cosine similarity was calculated (top 100 of selected docIds) were hyperparameters we manually tuned so that without losing too much search quality we were able to speed up the query. We were able to pinpoint the bottlenecks by adding debugging to our server logs, which log out how many milliseconds/seconds each step takes.

We used Apache Benchmark to stress test our overall system. We utilized EC2 autoscaling groups to test performance across multiple t2.xlarge instances. Our system was able to handle **1000 concurrent requesters each making 1000 queries with an average response time of 1.3 seconds**. This is largely due to our auto-scale enabled system, as well as query result caching.

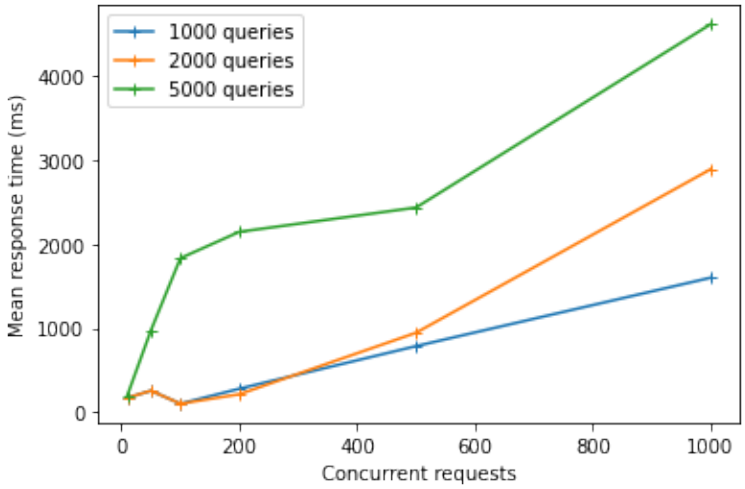

Figure 3. Stress tests

5. Conclusion

This project was challenging for a handful of reasons, specifically related to scalability, speed, and interaction between different components. We are proud of our final product - it returns good search results in around 1s on average (much faster if cached). We believe we had success in completing this project because we communicated early and often, defined interfaces for interaction between components before implementing, and prioritized speed in our implementations.