CIS555 Final Report

Handy Search: A Cloud-Based, Distributed Search Engine

Harsh Parekh, **A**lex Hirsch, **N**eil Shweky, **D**avid **Y**astremsky

May 2021

Abstract

Handy Search returns high-quality results to user search queries, running on a fault-tolerant, distributed set of nodes on Amazon Web Services. This paper details, analyzes, and assesses Handy Search.

Introduction

Project goals

Handy Search is a search engine aiming to provide high-quality, fast results to search queries. To achieve this, we set a goal for Handy Search to crawl and index 1M+ unique web pages that span a variety of domains and parts of the web. This should provide a sufficiently large corpus to return quality results for most queries. For enhanced performance, our solution will be scalable, distributed, and fault-tolerant.

High-level approach

Search engines provide a simple, quick mechanism for finding information on the web. Behind the scenes, web search for the scale of the web requires a system of components. Based on The Anatomy of a Large-Scale Hypertextual Web Search Engine (Brin & Page, 1998), these components include:

- A crawler to discover sites on the web
- An indexer to process pages and pre-compute which are relevant to potential queries
- PageRank to determine which sites should be prioritized in search results
- A database to store these calculations as well as web content
- A frontend to take in user queries, query the database, and display results.

Handy Search uses the big data streaming service Apache Flink to crawl and index web content, Hadoop MapReduce to compute the PageRank algorithm, and MongoDB as a database and for various computations, including TF-IDF and cosine similarity. Coupling the crawler and indexer in a Flink pipeline allows greater control in colocating compute and setting parallelism for fetching, parsing, and storing web content. TF-IDF and PageRank calculations are triggered after the crawling and indexing are completed. Finally, a node.js servlet is started to accept user queries. At query time, cosine similarity is calculated between the query and crawled pages. Results with the highest cosine similarity and rank are returned.

Division of Labor

We sought to work together by communicating, seeking consensus, and pair programming often, particularly as we prepared for integration and deployment. We were distributed across the globe with varying scheduling restrictions. Collaboration within components was challenging, so we assigned primary responsibility for each component to at most two team members to ensure that component's success. The assignments were as follows:

- Harsh: crawler, deployment, modeling
- Alex: indexer, search algorithm, MongoDB
- Neil: PageRank, UI, search engine servlet
- David: indexer, deployment, documentation

Milestones

Please find the completed milestones in the below chart.

PHASE	MILESTONES	END DATE
Set-Up	• Determined architecture for project components • Started project components	4/12
Components Completed	• Completed crawler, indexer, PageRank, and engine	5/4
Integration	• Achieved high-quality results in under 3 seconds • Completed code documentation for base components	5/5
Deployment	• Ran Flink locally to get crawl results • Deployed PageRank to Amazon Web Services • Deployed search engine to Amazon Web Services	5/9
Finalization	• Debugged functionality and fine-tuned results • Finished code and project documentation	5/10

Figure 1: Project Milestones

Architecture

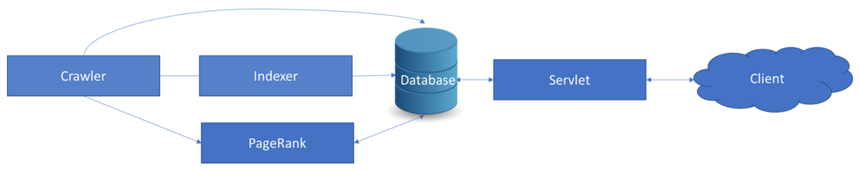

Figure 2: High-Level Architecture

Crawler and Indexer

The crawler and indexer were developed on Apache Flink. They were built to be deployed on AWS's EMR or KDA.

PageRank

PageRank uses Hadoop MapReduce running on AWS EMR.

Database

The database uses MongoDB, specifically MongoDB Atlas to deploy to AWS.

Search Engine Servlet

The search engine uses a Node.js backend to process queries. The UI is written in React. The search engine servlet was deployed on AWS EC2 to provide scalability for concurrent requests.

Implementation

Stream Processing Engine

Our most significant design decision was selecting a stream processing engine. This would determine the structure of our entire program. We chose Apache Flink due to its low latency, fault tolerance, parallelism, and checkpointing with exactly-once consistency semantics. These qualities were key for providing a performant, distributed, fault-tolerant search engine. Initially, we planned to run our entire program in a Flink topology that included the crawler, indexer, and PageRank running incrementally and continuously, much like any modern search engine. However, we ultimately ran PageRank in Hadoop instead, as PageRank requires significantly greater effort to implement as a stream rather than a batch. Nonetheless, the crawler and indexer operated as a single topology with built-in fault tolerance and checkpointing, attributes that played a key role during a crawling failure described in the evaluation section.

Crawler

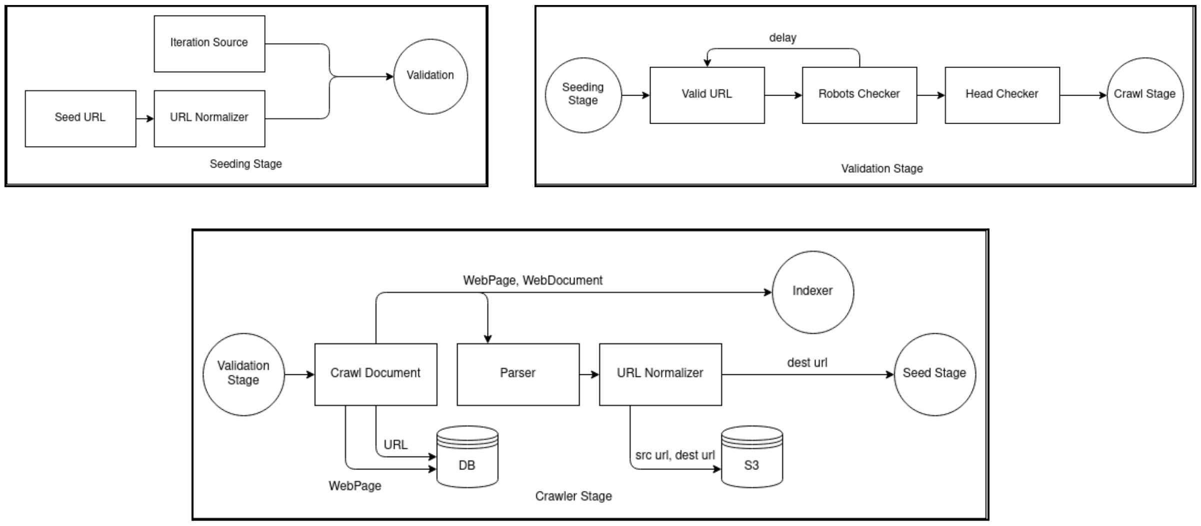

Figure 3: Crawler - Stages of Computation

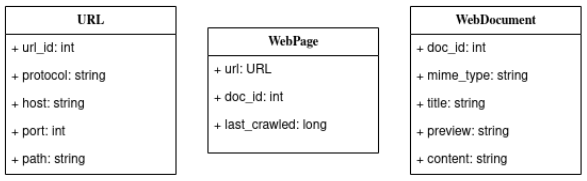

Figure 4: Database Models

The crawler can be broadly split into three stages: seeding, validating, crawling.

1. The seeding stage provides a stream of URLs to the crawler. It is initialized by reading in either a local or an S3 seed file containing URLs to high-quality, diverse web pages. It also takes URLs generated in the crawling stage and feeds them into the validating stage.
2. In the validating stage, URLs are normalized and validated against the URL specification. These normalized URLs are then fed into a robots checker that emits allowed URLs in an order that respects politeness requirements. This step is optimized by distributing URLs to robots checkers based on their hostname and keeping an LRU cache of robots.txt on each worker and a durable cache on the database to avoid repeated fetches from hosts. A HEAD request is then sent. Its response and list of crawled web pages determine if the page needs to be fetched.

3. In the crawling stage, the crawler fetches the document at the URL and parses the HTML, RSS, or XML documents to extract all the links. These links are fed back into the seeding step. The source-destination pairs of links are stored in an S3 file which is used by PageRank. The URL and document are encapsulated in a WebPage model and a WebDocument model that contains the data and metadata for the URL and document respectively and are sent to the indexer. Additionally, the URL that was just crawled is added to an S3 file to help jumpstart any future crawls.

Determining the split of computation was a critical design decision. In particular, we ensured steps passing large amounts of data such as the WebDocument object are co-located on the same worker node. This minimizes network communication amongst the workers. However, higher co-location comes at the expense of independent scaling of individual components, so we came up with a colocation strategy that maximizes the number of cuts (individually scalable steps) while keeping network communication low.

Indexer

The indexer receives the WebPage and WebDocument constructs from a given webpage. It parses the document into a DOM tree using Jsoup. It saves the title and first 300 characters as a preview in the WebDocument, allowing for efficient retrieval for search queries. Then, as it walks the DOM tree, it strips special characters, omits stopwords, lemmatizes words, and counts the frequency of words. In addition, it notes certain special tags (e.g. “title”) around words that should be weighted more heavily. The indexer emits tuples with the document ID and word to a sink, where a MongoDB MapReduce job turns them into the inverted index. Using the weightings, it emits term frequencies to a side stream. It sends the updated WebDocuments to another side stream. The side streams cache the data until the target batch size is reached or Flink is terminated, bulk writing them to the database. Once crawling is complete, the indexer triggers a TF-IDF calculation using MongoDB’s aggregation pipeline, outputting a collection of tuples with the document ID, word, and TF-IDF.

PageRank

PageRank uses Hadoop MapReduce to iteratively run the PageRank algorithm. The algorithm has 3 stages:

1. Initiation: Take in source-destination pairs of URL hashes and create an adjacency list for each node. Set the initial PageRank to 1.
2. Iteration Stage: Complete 20 iterations of the PageRank algorithm.
3. Finishing Stage: Record the maximum difference between the final two iterations. Output the URL-rank pairs that were computed.
4.

The final results are outputted to S3 and then imported into MongoDB using a bash script and mongoimport. These ranks are combined with TF-IDF results for each URL.

Engine

The search engine has a React frontend and a Node.js backend. When a user inputs a query, it strips the query of special characters, splits the query into terms, and lemmatizes the terms using the same algorithm as is used in the indexer. The engine then uses MongoDB’s aggregation pipeline to calculate the query’s combined TF-IDF cosine similarity with each document’s, and multiplies that by the document’s PageRank. The results are sorted and the top thirty are returned to the user.

Evaluation

Crawling

While we were not able to deploy to Amazon Web Service's EMR, KDA, or EC2 due to permissions issues, we were able to run the local crawler. The most important metric we have is the reason we used Apache Flink: fault tolerance and failure recovery. The computer running Flink lost internet for two hours, yet the crawler persisted through the network outage and recovered from failure to continue crawling. This is due to the fact that our crawler was configured to create distributed restore checkpoints every minute. If a failure occurred, the crawler would restart automatically from the last checkpoint after a set period, ensuring exactly-once semantics.

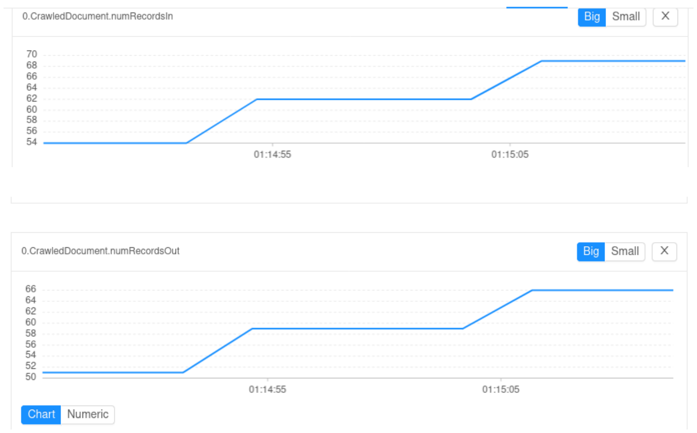

Figure 5: Documents Crawled Vs. Time

In running locally with 6 threads (1 per core), we found that the bottleneck was the network I/O during crawling. What further amplified the bottleneck was a 500Kbi/s average internet bandwidth on the machine. We were able to reduce the bottleneck by using asynchronous network I/O, but it traded off the exactly-once semantic guarantee. Therefore, we kept using synchronous I/O. That resulted in the bulk of crawling time to be spent submitting requests to web servers, since that blocking time took longer than computational tasks. This bottleneck is evident in Figure 5, which shows that the crawler is blocked for about a minute processing URLs during which it cannot accept any new ones, then repeating this behavior with the next URL set.

The easiest fix would be to increase parallelism for crawling. While this was not possible with a single worker, it is trivial to implement on a multi-node cluster on AWS. Deploying to AWS would also provide greater network bandwidth, further reducing the impact of this bottleneck.

In the end, we crawled **35,000 unique** documents, lower than we expected. However, this was run locally on one machine in one night to provide results for searching.

PageRank

PageRank ran quite fast, running on 5 worker nodes (and one master) in 16 minutes to rank 134,461 web pages. The PageRank ran for 20 iterations and the max change in the last iteration was 0.001.

Searching

Handy Search was tested for its ability to handle multiple word search queries using Apache Bench, as shown in Figure 6. The data from testing up to five-word search queries indicate response time does not increase with additional query terms. We attribute this to MongoDB's aggregation pipeline and sharded collections that provide parallel computation over multiple queries.

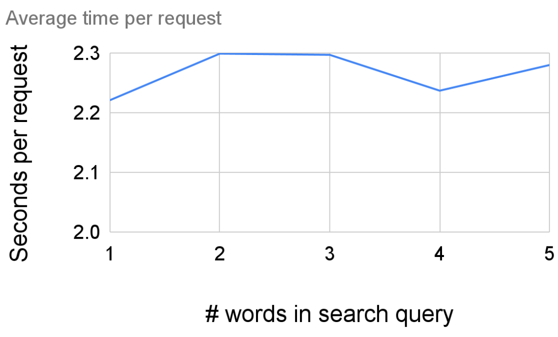

Figure 6: Search Term Length

Lessons Learned

One lesson was that remote collaboration makes it easy for components to get isolated. While our aim was to collaborate amply using software like Code With Me, it became more efficient for us to assign responsibility for components, then have periodic coordination calls. Hence, we were reliant on individuals for each part, increasing the risk of failure if one person was unable to get their part done. Fortunately, we stayed productive, but the risk was there.

In a similar vein, we learned that getting APIs and base functionality done early was key. While we created base APIs and data requirements, we realized that blocks in a certain component go downstream. For example, Flink makes it challenging to add data to an input source, like the crawler needs. This blocked the crawler from running, blocking downstream components from running with real data. We also changed data requirements during integration, resulting in late-stage code overhauls.

Perhaps most importantly, we learned the importance of building minimum viable products early. This is challenging to do in a streaming system like Flink, in which components are interconnected. Because we did not make this a focus, we did not meet self-imposed deadlines and were left scrambling during deployment. In addition, it meant we did not have a corpus of data for the indexer, PageRank, and search engine to use for fine-tuning. If we had deployed a simple Flink program to AWS early on, we would have discovered that we either needed greater permissions to do so or that we should switch the data processing engine. We would also have discovered that we needed to use Java 8 rather than 11 earlier and avoided a last-second need to port our code between versions. In the end, after attempting a variety of deployments to EC2, EMR, and KDA on AWS, we found we had inadequate permissions to run Flink. This is a discovery we could have made weeks earlier. As it happened, there was too little time to switch to another stream engine, leaving us running it locally.

Conclusions

Overall, this project was one of the most challenging and rewarding software projects we have ever worked on. Understanding the building blocks of Google and the web—plus how to use modern tools like AWS, Flink, and Kubernetes—will be invaluable for our future careers and projects. Scouring enterprise software documentation, navigating last-second failures, and integrating a large codebase were key experiences that we will be able to carry into our post-graduation roles. We are incredibly proud of the work we did in bringing this search engine to life and grateful for the opportunity to do so.