Final Project Report – CIS455/CIS55

Jonah Miller, Simmi Mourya, Sadhana Ravoori, Vikas Shankarathota
University of Pennsylvania

I. Introduction

The main goal of this project was to build a scalable web crawler hosted on Amazon AWS complete with a crawler, indexer, pagerank, and a front end.

A. Project Goals

1. To have a functioning, reasonable search engine which retrieved relevant pages.
2. Create meaningful indexes and page rank scores for all the webpages crawled.

B. High Level Approach

The flow of the system goes as follows:

1) The crawler runs and gets a large relevant corpus of crawled links. Uses filtering techniques to reject irrelevant webpages. And stores outputs in S3 and RDS.
2) The indexer runs on the content of the crawled webpages to obtain idf and tf scores of the vocabulary. Outputs are written to S3 and moved to RDS.
3) The page rank runs to convergence and gets a ranking for all the crawled webpages to S3 and moved to RDS.
4) The front end queries the RDS for tf-idf scores and pagerank and delivers the search result.

C. Division of Labour

1) Jonah Miller: DevOps for S3 access control, EC2, Developing and Running the optimized crawler
2) Sadhana Ravoori: PageRank, Utility scripts to move data from S3 to RDS/EMR jobs, DevOps for EMR, Minor Scaling of indexer
3) Simmi Mourya: Running and scaling Indexer. DevOps for Gradle, EMR, Hadoop, EMRFS. Minor Hadoop DevOps for PageRank.
4) Vikas Shankarathota: Front End, RDS management, running/testing crawls

D. Milestones

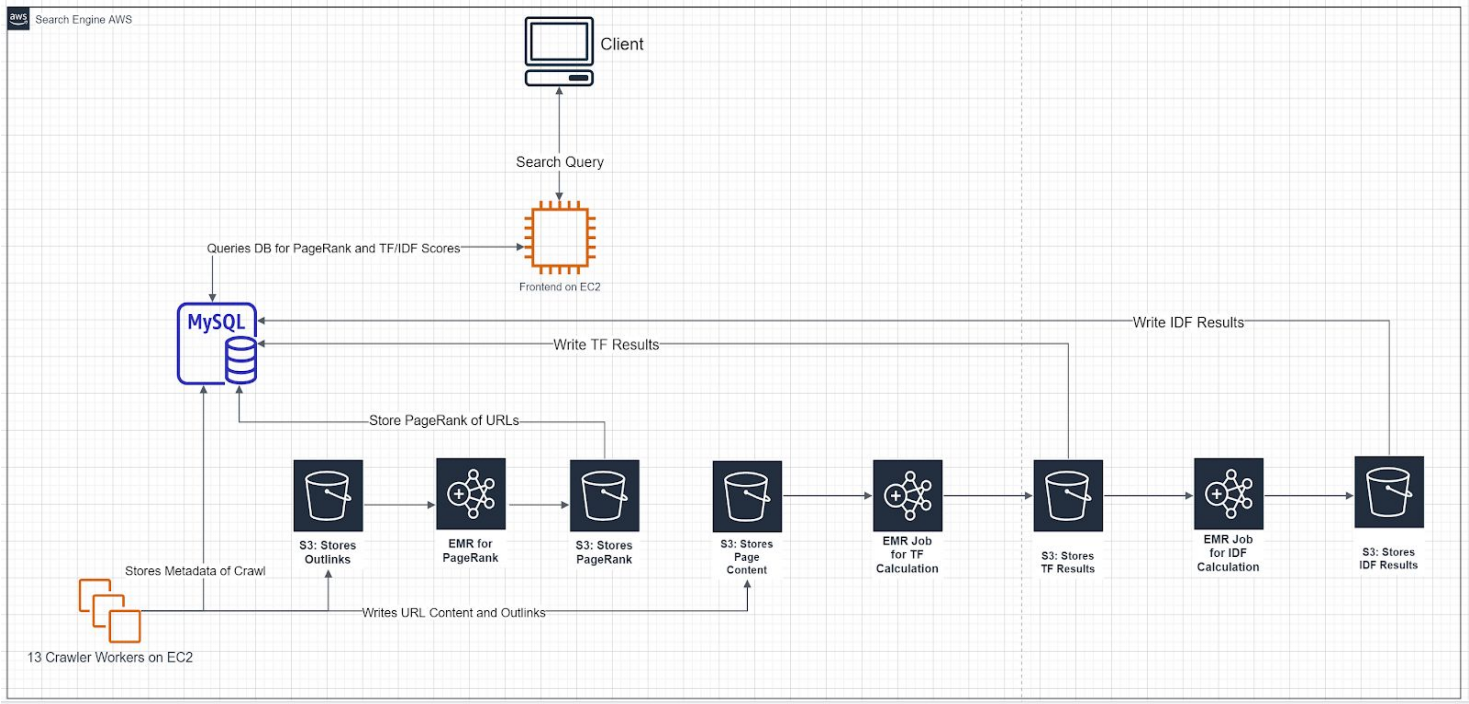
1) Optimizing and scaling crawler
2) Developing access policies for S3
3) For indexer, making performance and scalability evaluation comparisons between HW3 code and EMR jobs
4) Running EMR jobs like PageRank and Indexer on Hadoop locally and making EMR compatible JARs
5) Scaling Indexer and PageRank for huge corpuses.
6) Testing the frontend and scoring
7) Develop corpuses of 300 for testing followed by 1000, 5000, 10000, 20000, 50000, 100000.

II. PROJECT ARCHITECTURE

The entire system was hosted on AWS as shown in the figure below. The system comprised of the following components:

a) One EC2 instance that hosts the Java Spark Web App that allows the client to search.
b) One RDS instance which played the role as the data store for the Search Engine
c) S3 Buckets to store input and outputs for PageRank and Indexer
d) 2 EMR clusters for running TF and IDF jobs separately and 1 EMR cluster for running PageRank
e) Set of EC2 instances to run as web crawlers.

The main design choice of the implementation was to have a single database which stores all the outputs required by the Web Application to deliver a search result. As a result the entire system was built around the MySQL RDS instance. Another design choice was to run 2 separate EMR clusters for TF and IDF jobs instead of Chaining 2 jobs in a single EMR job. This helped in developing a stage-wise prototyping for Indexer, especially while running jobs on huge corpuses.



III. IMPLEMENTATION

Crawler

Design Features:

1) Ran 13 crawlers, each on their own extra large Ubuntu EC2 instance
2) Downloaded ~25,000 links per hour (all instances) and a total of 100,000 URLs.
3) Some Seeds were: http://redditlist.com/, https://www.nytimes.com, https://www.tripadvisor.com, https://www.whitehouse.gov, https://www.imdb.com, https://en.wikipedia.org/wiki/Wikipedia:Contents, https://www.geeksforgeeks.org/, https://dmoz-odp.org/,
4) Wrote to S3 buckets and RDS as described above for use in the pagerank, indexer, and frontend query
5) Singleton access to crawler to access frontier, delay map, RDS, crawler shutdown, etc.
6) HashMap used to keep track of most recent crawl time for a host to keep track of crawl delay
7) When crawling a host, store "new Date().getTime() + crawldelay" in RDS. If the time while crawling that host again is less than the time in the map, re enqueue and continue delaying
8) Fast access and occupied little memory
9) Used Apache TIKA for language detection

Topology

a) URLSpout (5 in parallel)

1. Poll frontier (blocking queue) for a URL
2. Send URL to RobotsTxtBolt if it's non-null
3. RobotsTxtBolt (5 in parallel)
4. Parses URL to get host and port
5. Uses host and post to get robots.txt, writes data to RDS
6. Set timeout to 2 seconds to reduce latency
7. Send URL to CrawlerBolt

b) CrawlerBolt (5 in parallel)

1. Send HEAD request to URL using Http[s]URLConnection
2. Set timeout to 5 seconds to reduce latency
3. Reenqueues URLs to frontier if there is a crawl-delay or redirect
4. Only accepts HTML pages less than 5MB
5. Does nothing if an error code (4xx, 5xx) is returned
6. Otherwise passes URL to DownloaderBolt

c) DownloaderBolt (5 in parallel)

1. Fetches document with JSoup
2. Set timeout to 5 seconds to reduce latency
3. Extracts outgoing links using hrefs, does some blacklisting and filtering
4. Saves page text and outgoing links to S3
5. Passes outgoing links to FilterBolt

d) FilterBolt (5 in parallel)

1. Filters out links already seen
2. Databases
3. Wrapper classes are defined to save data to S3 and RDS
4. Extra steps taken to give owner access to S3 objects (ACLs)

Indexer

a) Design Choices:

1) For indexer, made performance and scalability evaluation comparisons between HW3 code and EMR MapReduce jobs
2) Decision between Apache Spark and Hadoop MapReduce on EMR
3) Ran a basic version of Term Frequency (TF) MapReduce job on Hadoop locally
4) Wrote Gradle scripts to package Hadoop programs into JARs to run as Custom JAR step job on EMR clusters
5) Learned EMR and, EMRFS (an HDFS-compliant file system to access objects in S3) interfacing.
6) Design choice between running TF and IDF jobs in a chain as a single EMR job or a series of jobs
7) **Scaled the bottleneck Mapper of basic TF** to support downloading individual content files directly inside Mapper. The former method reads lines directly from the documents. This does not scale well. To solve this, the key idea was to instead provide a huge text file with the paths of all the documents (collected web pages) present in another S3 bucket. This way, the TF map reads a line which essentially points to a document in another S3 bucket and can be directly downloaded during the Map phase. Now, Pattern Matching for preprocessing can be done on a file in one go instead of doing it line by line (first approach). This scales extremely well, the first approach took **~2 hours** for ~8k documents while the second approach takes **11 minutes** for the same 8k documents. The concept of "accessing multiple small files is costlier than accessing a small number of big documents" comes into play here.

b) EMR Configuration: Master:1 m5.xlarge, Core: 2 m5.xlarge (all nodes with two 32GB EBS volumes)

c) TF MapReduce Job:

Map:
Connect to EMRFS by using the following command:
fs=FileSystem.get(uri,context.getConfiguration());

Read individual file inputs from S3 content bucket by connecting to hadoop's FSDataInputStream and reading the contents of the file via BufferedReader to a String.
Remove all special, accented characters and digits. Tokenize into words and after lowecasing and removing stop words, emit each resulting word (delimited with corresponding Document ID: SHA 256 Hash of URL) with frequency 1.

Reduce:
Sum up the same word occurrences per document and emit the Term frequency sum for each word-document combination.
context.write(delimitedToken,newDoubleWritable(sum));
This TF score is non-normalized and gets normalized later while making query and document vectors during search time. The outputs of reduce are written to an intermediate S3 bucket which is used by IDF MapReduce EMR job.

d) IDF MapReduce Job:

Map: Emit the lines of files written by TF job. Minor operation of separating token from document id.
context.write(new Text(token), new Text(hashURL+"=="+freq)); where freq is TF score and hashURL is document ID.

Reduce: A local hashmap is maintained to keep a counter of document frequency (DF) for each token. For every occurence of the token the hashmap is the corresponding hashURL+"=="+freq for that token is added to the hashmap and a DF counter is incremented. This DF alongwith total number of documents is used to calculate unique IDF weights for each unique token using the following formula.
IDF = Math.log10(1 + (TotalDocs/DF));

This IDF score is then emitted along with the token. The output of the reduce phase was written to an S3 bucket.

e) Running jobs:
The TF and IDF jobs were run one after the other as two separate EMR MapReduce jobs. The outputs of both the reduce phases were written to RDS using Batch execute scripts for faster updates to the DB.

f) Scoring:
The IDF scores were later used for calculating an IDF weighted Query vector. The TF scores were used to calculate an Euclidean normalized document vector. Since query can be thought of as a small document, this vector is compared with all the document vectors corresponding to the documents in which query words occur.

PageRank

1) We considered two different approaches for the implementation of the PageRank algorithm as a mapreduce job.
2) The first implementation involved running the PageRank job on the HW3 map reduce framework.
3) Since the pagerank algorithm has to be run iteratively, we tweaked the code to run the job iteratively. We calculated the number of EOS needed to determine when the job was completed in the PrintBolt of the topology.

   noOfVotesNeeded =
   noOfReduceThreads * noOfWorkers
4) The PrintBolt received this number of end-of-stream keys, it would issue a GET request to the MasterApp with the details of the same job.
5) We kept track of the number of iterations in a text file to which the MasterApp wrote everytime an iteration was started.
6) This initially worked well with a small set of webpages but there were a few issues we ran into with this implementation when we tried to increase the number of threads in the threadPool and hence decided against using this implementation.
7) The second implementation was the hadoop job which ran on an EMR cluster. The map phase of the PageRank job accepts a string containing the URL, PageRank score and the outgoing links and computes the contribution of the URL to the outgoing links. Score assigned to each of the outgoing link = (pagerank score)/(number of outlinks). The mapper phase emits two types of result. The first one is the outgoing link as the key and the score assigned to it as the value, the second one is the current url as the key and all its outgoing links as the value.
8) During the reducing phase, we first determined the type of result from the mapper and then summed up all the individual contributions of the different URLs.
9) The reducer emits the results in the same form that the mapper accepts. The reasoning behind this is that the PageRank algorithm runs iteratively, and the output of the previous iteration becomes the input to the next iteration. Thus the reducer output becomes the input to the mapper in the next iteration.
10) We used a damping factor of 0.85 to combat page sinks and get the final PageRank score of the link. PageRankScore = 0.85*PageRankScore + 0.25.
11) We computed the pagerank scores for the dangling links in the same way.
12) We run multiple iterations of the algorithm until a fair degree of convergence. We determined that the number of iterations taken on a corpus of size 100,000 to be around 30.

Front-End

The front end of the search engine was hosted on an EC2 instance t2x.large which was visible to the world on a public port. The content was hosted by using the Java Spark server designed in HW1MS2 during the duration of this course. Velocity was used to display the dynamically obtained search results. The results and main search page were constructed with a combination of CSS, HTML, and Velocity (.vm) files.

The MySQL RDS was hosted on a 2x.large instance and was the main data storage for the crawl meta-data as well as the outputs of the indexer and pagerank. The data that was stored in our RDS instance consisted of: TF scores as postings, IDF scores, pageranks, allow rules, disallow rules, crawl delays, and urls crawled.

The web server would query the MySQL RDS for the values it required to calculate the TF-IDF scores and the PageRank of the returned documents. Upon a search query,

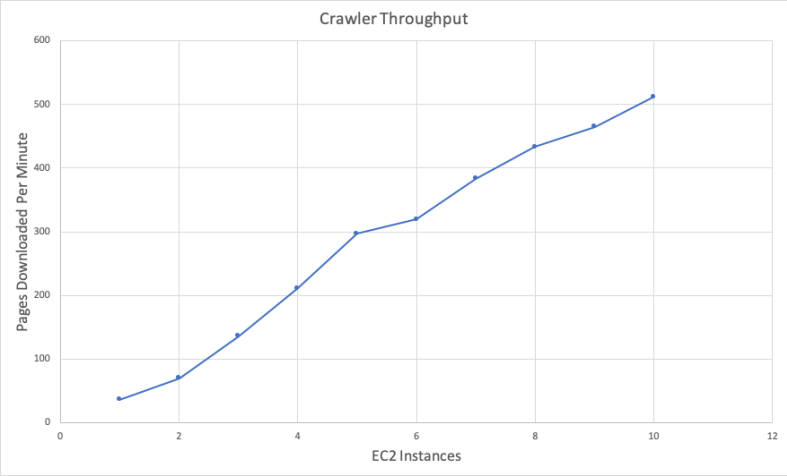
1) Stopwords were removed and the term frequency of the words in the query were calculated
2) Common IDFs were pre-cached in memory while any other IDFs had to be retrieved from the database. These common words were selected to be cached as they are often queried for.
3) The $W_{t,q}$ vector was found for the search query.
4) The TF score and corresponding document was retrieved from the database if the word occured at least 3 times.
5) Euclidean Normalization was carried out on the document vector of the tf scores corresponding to search terms that appeared in a document.This vector was multiplied with the $W_{t,q}$ vector to find the tf-idf score of the document.
6) The pagerank scores of the top 50 tf-idf scorer documents was queried from the database.
7) The final rank of the page was calculated by multiplying the two values.
8) Out of these top 25 pages were listed on the results page.

Due to the large size of the database, the queries take some time and hence we had to clean out some of the garbage values in order to try to speed up the execution. Some of the things we did was; remove non English words being indexed, remove words greater than a length of 15, remove words lesser than a length of 3, return hits if the word appears on the page

at least 3 times, removal of stopwords, lower-casing the characters, reducing the number of redundant queries etc. One of the main reasons for this extremely large size was due to the fact that our Indexer did not use stemming which would reduce the size, but in turn would yield more inaccurate search results.

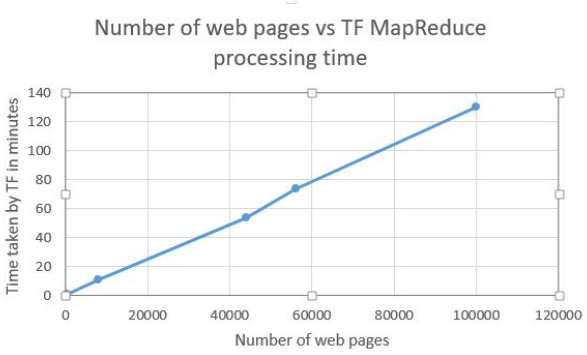
IV. EVALUATION

Crawler



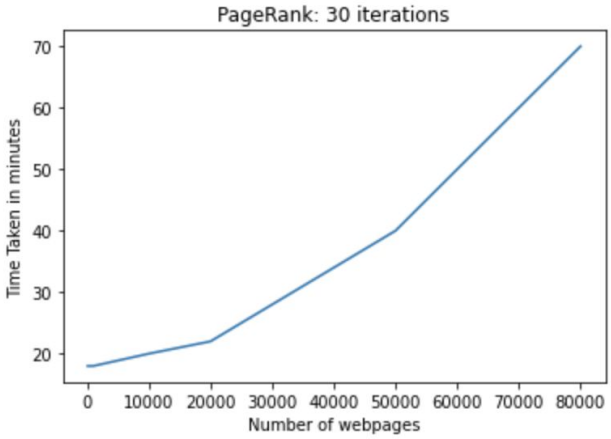
The seed had some effect on the number of links downloaded per minute by an instance, but overall growth in throughput for the crawler was linear. This is what I expected; all of the EC2 instances were the same (XL instances containing 4 vCPUs of 16 GB RAM each), so I knew the crawler would run at more or less the same speed on each instance. The latency primarily depended on a) the quality of the server that the crawler was requesting (bad servers caused slow downloads) and b) crawl delay. Both of these components depend on the seed.

Indexer



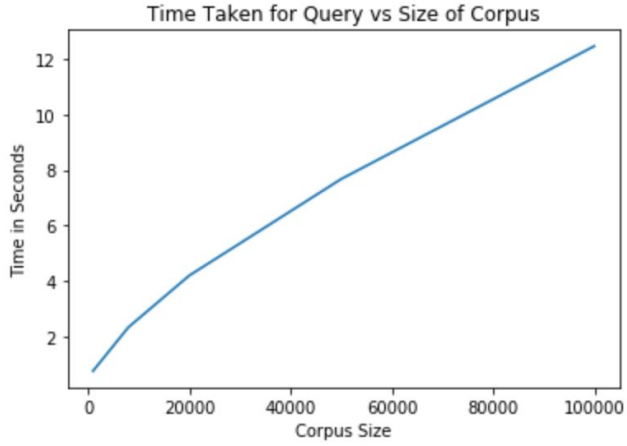
The Indexer performance increased somewhat linearly with increasing number of documents. The optimized TF MapReduce cluster (Master:1 m5.xlarge, Core: 2 m5.xlarge) finally takes somewhere around ~2 hours to produce TF scores on 100,000 documents. ~98% time is consumed by the Map phase. The IDF MapReduce job for 100,000 documents runs in 6 minutes.

PageRank



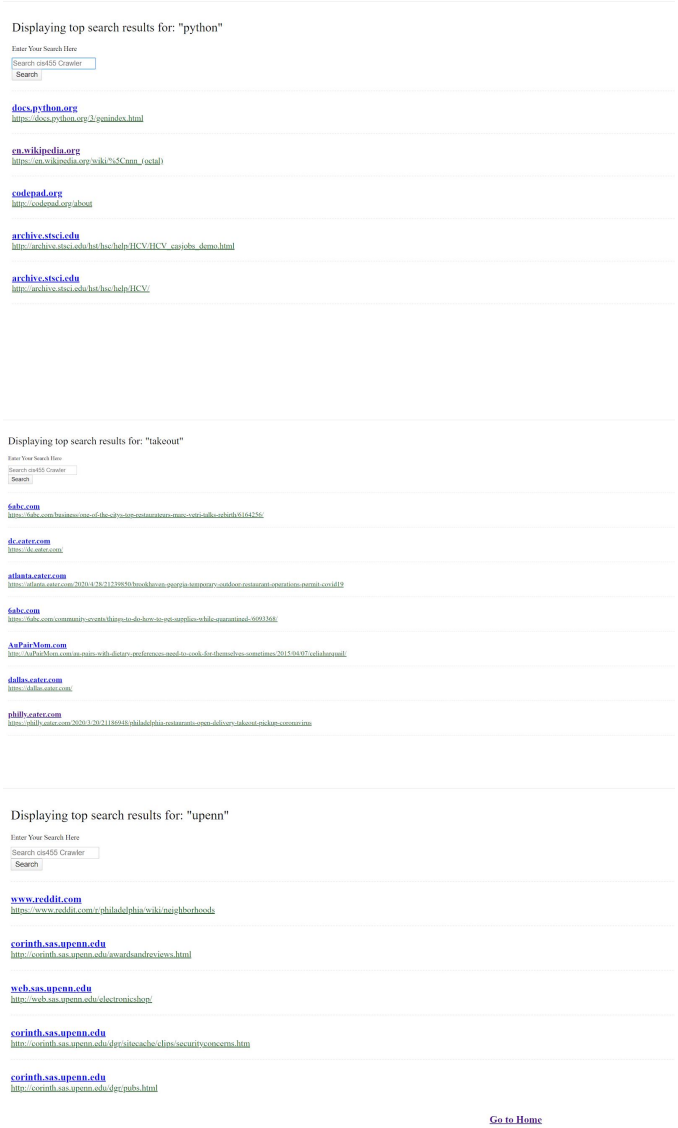
We observed that as the number of webpages increased it, the time taken to run the mapreduce job increased almost linearly. Initially from 100 to 20000 webpages, we noticed that the amount of time taken remained almost constant (18-20 minutes). We ran this job on a cluster with 1 master (m5.xlarge) and 2 workers (m5.xlarge).

Fetching Search Results



The search result query time was a bottleneck in our approach. Initially we assumed that having a SQL database would make searching faster, but it turned out to be a very expensive process. That along with the tradeoff of not using stemmed indexes in turn for better search results made our fetch time slightly on the slower side despite best methods to optimize it.

V. SAMPLE OUTPUTS



VI. CHALLENGES FACED

We faced a myriad of different problems when we were doing this project. Some of the most notable ones were:

1) Not having access to AWS CLI and IAM to allow each other's AWS account to access common resources
2) Difficulty in dealing with a 3 hour session window for session-based S3 credentials. Eventually solved by using anonymous S3 credentials and making bucket policies public
3) Figuring out how to set S3 object permissions to read objects placed in bucket by anonymous credential write, ended up being solved with the Java SDK CannedAccessControlList class
4) Figuring out the policy for EMR Default Role insufficient permission EC2 permissions by updating the role to have full S3 access. Policy attached: AmazonS3FullAccess
5) Handling edge cases of badly formed URLs while crawling (e.g. links containing "." or "..", non-UTF-8 characters, self links)
6) Size of database to query to generate search result output which was partially solved with database optimization and caching.
7) Coming up with a scalable way to move the data from the S3 bucket to RDS.

VII. CONCLUSION AND FUTURE SCOPE

- You should never ever assume that any content received from the Internet is well formed (regarding both pages and URLs)
- The crawler would download a lot of good, relevant, popular pages, but would download even more garbage. This was expected, but still unnerving to see it download so many links from an irrelevant site.
- RDS was a great way to format the data (both metadata for use by the crawler and indexer), but made for very slow queries from the frontend, which were difficult to optimize.
- It's important to have a well-defined structure of a) how all of the components will communicate with one another and b) how the components will store/retrieve big data sets efficiently
- A corpus of documents is best formed when it was accumulated in a single crawl. Amalgamating more than one corpus can create a disjoint corpus and interfere with pagerank/indexer
- It's important that everyone is on the same page about absolutely everything regarding the component configurations, particularly because the crawler, the indexer/pagerank, and the frontend execute in series. Misunderstandings are inconvenient at the least and catastrophic at worst.

We are happy that at the end of the day we were able to build a functional basic search engine given the resources. Some possible advancements we would have loved to have explored would be handling SEO, crawling for additional file types or page metadata, and exploring different data storages for faster querying.

ACKNOWLEDGMENTS

We would like to thank Professor Andreas Haeberlen for the opportunity to work on this project as a part of the course. Also, we would like to thank Rishab Jaggi for his inputs and advice during the project.