CIS 5550 Project Final Report

Bingqing Fan, Geoffrey Brandt, Michael Franco, and Michelle Naps

1. Extra/Enhanced Features

- **Crawler Optimization:** We had to enhance our crawler in order to account for the scope of our project. For example, instead of crawling the entire body of a page, we restricted it so that we only crawled for elements in the body that contained useful words for the indexer, or anchor tags that contained href attributes to other URLs. We also noticed that some of the optimization/filtering we were doing in our indexer could be moved to the crawler, like filtering out URLs that contained a ".." within them. This allowed the crawler to provide better results for the other jobs, and increased their performance
- **Infinite Scrolling:** The frontend server handles user search queries by initially fetching the top 10 ranking URLs and generating a result page. To enable infinite scrolling, the server attaches a scroll event listener to the window. When the user reaches the end of the page, triggering the scroll event, an AJAX request is sent to the "/scroll" endpoint on the backend. This request includes the current page number, which is used to calculate the ranking offset for retrieving additional results.
- **Search Suggestions:** When the user types in a search query but the last word in the query remains unfinished, the user will be given a list of suggested words to finish the query. Our algorithm takes the user query, picks all words in the index that begin with the unfinished word, calculates the tf-idf value for each suggested word, and returns the top ten best words to complete the user's query. This was done in the frontend due to issues with communication between the frontend and the workers.

2. Key Challenges

One key challenge was making the data processing compatible with both the crawler and the frontend. Importing our own homework implementations posed too many challenges and we spent a large amount of time attempting to bug fix those issues. We ended up utilizing the professor's jar files to at least get the crawler and the data processing going, so that we could generate the necessary data for the frontend to use. However, for the frontend we needed to make GET requests to the backend's workers in order to properly send and receive data, so we could no longer use the professor's jar files. Therefore we went through numerous rounds of bug fixing in order to make our components compatible with each other, and eventually we added the endpoints for the GET requests so that the frontend could communicate with the backend after it had generated the required index and page rank tables. Eventually, we realized the best way to do this was pass the KVS Master's address to the frontend and use the KVSClient to access our pre-built tables.

Additionally, our group had difficulties in making the frontend quick and responsive for the user. For example, we first originally implemented the autofill suggestions to call the /suggestions endpoint anytime the user typed in a value into the input field. However, since the requests

could not be processed quick enough to keep up, this flooded the backend with too many requests and oftentimes completely crashed it. To get around this, we implemented a timeout feature so that the /suggestions route is only called when the user hasn't typed for more than one second. This prevented the backend from experiencing a flood of requests and allowed it to return responses consistently.

In testing on a local laptop machine, the web crawler would process 125-150 pages/min of content. The crawler was then transitioned to an AWS EC2 Windows environment, and it turned out that a c4.8xlarge instance was needed for comparable performance on a single instance. In initial testing of a c4.large instance, the AWS instance would quickly freeze and need to be restarted.

Initially, the testing in the AWS EC2 Windows distributed environment with just two instances was complicated by communication difficulties between the instances. The instances were both in a shared security group that was open to all inbound and outbound traffic with all protocols and on all ports between the instances, and the Windows Defender firewall was disabled on both instances. The netstat tool was used to identify java.exe processes on the relevant ports that needed to be terminated. Then, PowerShell Test-NetworkConnection (tnc) commands were used to verify bi-directional communication on the relevant ports. Eventually, substitution of IP address for "localhost" in the command line resolved the issue and allowed the web crawler to startup in the distributed environment, albeit with slow performance that lasted only a few minutes until one of the workers would terminate with a Java socket exception (connection reset). We remained hopeful that the web crawler would be able to achieve the goal of distributed crawls amongst four workers in the AWS Ubuntu environment when one of the team members was able to do so in the AWS student lab environment on c4.large instances. However, the performance in the distributed crawls never reached close to that of the single instance crawls, leading our team to defer on the use of distributed crawls to provide crawl tables for the project.

In its final version, the web crawler stored content from 90K-plus web pages, with each crawl capped at 10K pages for testing purposes. During this round of testing, seven web crawls reached the final 10K pages mark, three needed to be manually terminated due to getting mired in cyclic crawl paths despite the reputable nature of the host domains, and four web crawls terminated in various application error states. The crawl tables from these web crawls were useful in validation of the indexer, pageranker, and front end. However, without a custom aggregator to process these files, our final search engine was only supported by 10K pages.

3. Difficult Aspects of the Project

One of the most difficult aspects of this project for us was working with AWS. Most of the other aspects of the project were essentially based off of homework assignments, but since we did not have too much practice with distributed systems over AWS this posed a challenge for our group. There were a number of issues that we had not experienced during HW3, such as setting up multiple instances so that they were able to communicate with each other, managing storage

(especially in terms of the free version limit), and also managing the speed of our jobs since they differed from our local machines.

Another difficult aspect of this project was modifying our solutions to real-world websites. Our solutions operated great on the simple and advanced tests provided for homeworks 8 and 9, but once we began using real websites from the internet we encountered issues that we hadn't seen before. For example, the websites we crawled sometimes contained broken web-content (such as faulty/unpredictable HTML), which would result in our indexer accidentally generating "undesired" words. To fix this, we implemented more filtering both in the crawler and the indexer to only utilize proper query data that a user would submit. In fact, it seemed that for every new iteration of the crawl table, there was a new corner case that our algorithms did not take into account. So, it was a tedious process of continuous debugging throughout the entirety of the project.

Finally, the speed of the jobs on our EC2 instances was untenably slow at first, so optimizing our jobs was one of the larger challenges of this project. On our local machines, the crawler, indexer, and page ranking algorithms would operate at a satisfactory level, but on the EC2 instances their performance dropped to a very low level – even with multiple workers. Constantly needing to re-evaluate our algorithms for any bottlenecks or opportunities for optimization was very tedious, frustrating, and time-consuming.

4. For the Future

If we were to do this project again, we would definitely start earlier. Even though we began working on this as soon as the groups were finalized, if we had gotten deeper into the project sooner we would have had more time to address difficulties and errors. Especially since the bulk of this work was done around the week of the exam and finals period, having more time to tackle the issues we ran into would have made the process more bearable.

Also, going into the project with more familiarity with AWS would have been greatly beneficial. We spent a lot of time on our project learning how to use the EC2 instances properly for the scope of our project. HW3 did provide us with some background knowledge on deployment, but we ran into issues with implementing distributed systems, storage, security groups, and the speed of the virtual machine that we had not encountered in HW3.

Besides, as the size of index and page rank tables grow bigger and bigger, generating search results for queries gets dramatically slow, especially for queries with multiple words. One thing that we could do is to minimize the number of calls made to the KVS by fetching necessary data in bulk. Currently, multiple calls are made to retrieve data on disk for each word and URL, which can be time-consuming. Instead, we can try to fetch relevant data in larger batches to reduce network overhead. Moreover, the IDF value for each word can be pre-calculated and stored in memory, avoiding redundant calculations in each iteration.