Cloud Quest: Cloud-based Search Engine - Team InfoHunter

Purvansh Jain, Xinyue Liu, Zhijian Wang, Minzheng Zhang,

1.Describe any features you added or enhanced.

For the crawler part, we add persistent RDD to make crawler restartable. Later we found that our crawler might crawl some websites too deep, then we add the depth tacker for every page we crawl. Also, we wrote a url length checker, if the page's url length or depth exceeds some limits, then we will stop crawling on that page. To speed the crawling, we found that if we crawl the same host multiple times, it may trigger the anti-spider mechanism of the host(like block us, also, some robot.txt ask us to wait for a long time). Besides, we decide to add a random extract url to crawl from the crawler queue, because there is a high possibility that the host may just link to the same host links and result in clustering of same-host links and prolonged waiting time. We wrote a new randomFlatMap in FlameWorker, which can put rows from an iterator into a buffer and then randomly select an index from the buffer and then update a new element in that index. Therefore by randomly selecting the url, we can shorten crawlers to visit the same host within a short period of time. Also, if a crawler visits pages in the same host for too many times, we will add the rest links of the host to the next round crawling queue, to let other links get a chance to be crawled.

For the indexing and page rank part, we first create an extractLinks job that helps us to get all graph relationships among pages. The extractLinks persist, so once it is calculated, we don't need to calculate them again even when workers crash. To make the program more efficient, we also wrote another job called HtmlCleaner, which helps us compress the content of the crawl table. We calculate three types of word occurrences for words in pages. They are Priority1, Priority2 and Priority3, which represent word count in title, word count in metadata, word count in page body. In calculating page rank values, we add a buffer inside the loop so only when the buffer is filled, we then make requests to related kvs workers and store them into workers. We also add putQueuedRows in kvs to support multi-thread batch storing.

For the ranking part, we take a lazy-loading pattern. We wrote an indexed links stream, every time the server first sorts related indexed urls and only selects 5000 links and then we calculate their pagerank and tf-idf and do a ranking.

For the frontend part, we implemented infinite scroll and cache. Users can scroll the mouse to get new search results, and can click cache to get our crawling time's view.

2.Describe a few key challenges you encountered – say, broken web content, black-hat SEO, or low-quality rankings for certain queries – and how you dealt with them.

For the Indexer, there are actually two challenges. The first challenge is that there is a memory problem if we use a similar code as what we did in the HW; the crawling file will work pretty well if our crawling file is crawling a 50 Mb file. However, our indexer part will directly crash when our crawl table is 80 Gb. We did not find out this bug until we remembered that when we did indexing in HW 9, many temporary tables were generated. This means that we have to use 3 strategies to fix this problem:

The first strategy is to save the table every time we create a new RDD. However, there is also a disadvantage to this problem. We first try to call persistent, then use “rename_as' ' to transform the table. However, this strategy will not work. Memory will be used up before the renaming process. We need to add the persistent attribute to all the methods in Flamecontext, FlamepairRDD, and FlameRDD. Also, when calling the invoke method, this operation will directly add a key-value pair to a persistent table. Because Liu’s original code had a good structure, it did not take us much time to implement these methods. However, after we finish coding and try deploying it for our real crawling table. We came to the conclusion that this solution would not work. It is too slow. Every time one transaction is done, it will directly write something to the disk. Even though we changed our code later and used a buffer first to push the row to the memory, we still have 5 temp persistent tables to create. This means that we must apply the next strategy to speed up the indexing.

The second strategy is to use the KVS server and client alone. This strategy has one advantage and one disadvantage. The advantage is that we do not need to save 5 temp tables and delete them when programming runs. We actually only need to save the final table. The disadvantage is that we may need to rewrite all our code in the indexer because, right now, the flame structure requires us to do the saving. Also, this type of code may prevent us from doing a distributed system in indexing. We implemented it from scratch for the indexer. To increase the speed of indexing, we did not save our output to the Indexer table until our buffer reached a certain value. Reducing the number of operations of opening and closing the file indexer strongly increases the speed of the indexing part. However, this method has the problem that it cannot do the distributed computation. We also implemented This may lead to a third strategy.

We have not implemented the third strategy, but I will mention some ideas here. The main reason the first method is quite slow is that we need to save 5 temp tables. To prevent saving so many temp tables, we should define a new method in FlamepairRDD such that this method can generate one FlamePairRDD in one transaction rather than 5 transactions.

The second challenge we met was, after we first finished indexing, we found that there were so many useless words generated by certain websites. After finishing the first indexing and observing the result, we finally decided to use jsoup to remove the script section. This code helps our indexing become much more efficient.

Finally, I want to discuss the problem we found when we presented a demo to the professor. Our info hunter cannot search the University of Pennsylvania really well. I think this job can be fixed if we index three words together in our indexer.

For PageRank, if I do this project again in the future, I will spend more time testing the performance of extracting the links, because this is the bottleneck that makes our page rank become super slow sometimes. I will also apply the third strategy I mentioned earlier in the Indexer.

3. Describe the three aspects of the project that you thought were the most difficult.

1. Crawler’s data is unbalanced in distributed nodes. When we do crawling in two machines, we find that it's quite often that one node finishes its work, but waiting for the other one for a long time. We manually assigned workers ids and tried different values,

but it still did not work ideally. Then we rewrote the partition logic and made the partition more balanced.
2. Server response was quite slow at first. This is because it needs to calculate tf-idf and do ranking, we solved this by lazy loading and caching.
3. Crawler always waits for a long time when crawling the same host, and many websites usually do a lot of self-linking, so it's very common we crawl the same host in a short time period multiple times. This is very inefficient because if the host sets a very long time interval to allow the next crawling, we will wait for a long time for consecutive pages from the same host. To solve this problem, we implemented a buffer to randomly select pages to crawl. And as mentioned above, we use some techniques to rebalance the frontier sites so that there won't be a large proportion of links from same host

4.Describe what, if anything,you would do differently if you had to do the project again,and how?

If we had to do the project again, there are several things we would do differently to improve efficiency, performance, and user experience:

1. We want to implement an Apache-Spark-like distributed stream and lazy computation. So it will be easier to do page ranks and other distributed calculations.

2. We will do a search suggestion, based on BK tree data structure, we calculate words' keyboard weighted Levenshtein Distance, and calculate the user's input's distance with our dictionary, returning the top ten words matching.

3. Iteratively refine the ranking algorithm: we will add a click event listener to record which pages they click after search. And then adjust our weight for pagerank and tf-idf.
4. Location based search, we can get users' IP addresses and provide them more specific results, such as city weather.

By implementing these changes, we believe that we could further improve the quality of our search engine, enhance developer and user experience, and streamline the development process.