CIS5550 Project Report - HTTP200

Group members:
Changqi Xiao, chthon@seas.upenn.edu
Rongkai Liu, rongkai@seas.upenn.edu
Xuanbiao Zhu, zhuxb@seas.upenn.edu
Yuxin Hu, yuxinhu@seas.upenn.edu

1. Features added and enhanced

1.1 Index compression and stop words remove

Basically, to tokenize words, we use the simple and efficient unigram method. However, simply applying unigram tokenizer and extract all words will generate lots of words and thus overwhelm the KVS. A typical page without word compression normally contains around 600 words. To optimize the indexing and reduce the influence of word reduction, we use following methods:

1) Stop words dictionary: Stop words are common words that are frequently used in natural language but are generally considered to have little or no meaning on their own, it takes up roughly 30% - 50% of the total words in a page and acts like the "noise" in the page and influences the true index in "Normalized Term Frequency". We built a stop word list to remove them from the text content. This helps us reduce the amount of data to be processed, save a certain amount of storage for the resulting TF, IDF tables, and focus more on the meaningful portion of content we crawled.
2) In addition, stemming is adopted to reduce words to their base or root forms. The reference stemming algorithm we utilized is https://tartarus.org/martin/PorterStemmer/. Introducing stemming helps us reduce duplicated rows (roughly 33% of total words) led by the word with the same meaning but in different forms. Thus, the accuracy and efficiency of text processing phase is largely improved.

1.2 FlatMapToPair and MapToPair optimization

In our homework Flame worker design, the *FlatMapToPair* operation of *PairRDD* and *MapToPair* operation of *FlameRDD* first scans all the *FlamePairs* within a given key range in KVS, applies a lambda function to each pair, iterates through the resulting collection of pairs returned from the lambda, and then puts those pairs back into KVS during the iteration. Although this approach is theoretically sound, it is extremely inefficient due to the fact that each KVS put operation triggers a disk write of the entire KVS row to the log file. This becomes particularly problematic for certain operations, such as indexing and computing the transfer table in a single iteration of page rank, which can generate a large number of *FlamePairs* with the same key. This commonly results in .table files of exaggerated size.

To address this issue, on the Flame side, we tried "caching" the returned *FlamePairs* from the lambda as a map of Rows in memory within the Flame worker and then write them back to KVS using *putRow* until a predefined caching map size threshold is reached. Using this method, we can ensure that each key will not be written many times to the log file. The later computation we conducted showed the efficiency indeed improved to a certain extent.

1.3 Spellcheck and Infinite scroll

Based on the project handout, we implemented the spellcheck and infinite scroll pagination in our frontend. The spellcheck feature is built on the Bing search API (please refer to https://www.microsoft.com/en-us/bing/apis/bing-spell-check-api). When the misspelled word typed by the user is a short edit distance from a dictionary word. The suggested word will be displayed below the search bar for reference. Also, the search results will be dynamically retrieved when the user scrolls to the bottom of the page, instead of clicking on explicit pagination.

2. Key challenges encountered

2.1 Crawler

In a bid to enhance the efficiency of our web crawling process, we implemented a distributed crawler with eight dedicated workers operating on individual servers, and a separate server assigned for the master. The core challenge during the crawling process is managing a frontier of URLs for the next iteration of crawling, particularly because we often need to pause the crawler to evaluate its performance.

Initially, we tried to persist the frontier in a KVS table. The process entailed extracting URLs from a page and subsequently adding them to a “to-crawl” table. Upon restarting the crawler, we would cross-reference the “to-crawl” table with the existing tables of previously crawled URLs, and then proceed with the uncrawled URLs. However, this method was proved inefficient as the “to-crawl” table became excessively large, thereby requiring substantial time for exchanging copious amounts of URLs between servers. Moreover, the size of the “to-crawl” table also significantly increased the time taken to write to and read from the table.

As an alternative approach, we tried saving only the first few URLs from a page, which would serve as seed URLs for a restart. Nevertheless, this adjustment did not yield the desired improvement, with the crawler still processing at a rate of a few pages per second per worker. To mitigate data exchange between workers as much as possible, we realized that it was unnecessary to persist the frontier. Rather, it was sufficient to just have some seed URLs to initiate the crawling process.

At the end, I leveraged a selection of APIs capable of generating unique URLs, including:

1) https://www.reddit.com/r/random
2) https://en.wikipedia.org/wiki/Special:Random/Category
3) https://www.isimonbrown.co.uk/dicestack/
4) https://newsapi.org/v2/everything

These websites consistently provide random URLs each time they are accessed, which ensures the crawler always has a set of unique seed URLs not present in the previously crawled table. In addition to these major modifications, we have also implemented some minor enhancements to further improve the crawler’s efficiency. These include:

1) Each worker now saves a copy of robots.txt and the access time in a map.
2) We have instituted a separate daemon program to monitor the workers’ status and restart them if necessary.

2.2 Slow processing caused by large crawl tables

The tables containing crawled content are typically pretty large, which are often several gigabytes in size and distributed on different workers. To avoid the risk of memory overflow, the persistent tables have to be utilized throughout our implementation. But the disk I/O of some extremely large log files still led to the inefficiency issue during processing, even though the Flame optimization mentioned in the previous section was adopted.

From research, we learnt that when the writing operations involves opening and closing the file, the overhead of these operations can add up. So, the key ideas we came up with to address the problem are 1) using large SSD storage instead of HDD; 2) partitioning the crawled content into several chunks with reasonable size. Regarding the second method, for example, if we want to process 80,000 URLs, it can be done in four Flame tasks instead of one single submission. In each time, we distributed 20,000 URLs and their content on our KVS workers. This ensures that the transfer table, though keeps accumulating during the processing, would not increase to an incredibly large size to make the KVS operations unacceptably slow.

2.3 KVS shard unevenly

Our KVS partitions all row instances by the row key. Typically, we use Hashing to encode the URL for each page as the row key and put a row into the KVS System. However, we noticed that the distribution of the crawl table was extremely imbalanced among all eight KVS workers. To deal with this problem, we modify the encoding process to add a constant number of random characters before the previous encoded URL, since the KVS concerns more on the prefix of the encoded URLs.

3. Most difficult aspects of the project

3.1 Relatively slow response when fetched results are large

One of the most difficult issues we encountered is to make the query that matches a large number of results get responded in an acceptable time interval. The slow query could be caused by multiple reasons, and we think they are mainly

1) The KVS implemented lacks an indexing mechanism to help speed up the data retrieval process by allowing the database to quickly locate the relevant data based on the query. After desired URLs are found, extra operations consuming more time are needed to load page content to generate the title and the brief description of each search result.
2) Each query will be split into words, all URLs containing the expected word should be fetched and sorted to generate the response. Though pagination is introduced to reduce the response time, the time-consuming sorting is inevitable when the fetched rows are large.
3) Caching is absent. We didn’t have enough time to refine the implemented KVS for caching of searched and hot queries. Also, reasonably choosing what query results to cache is also a hard topic.

4) A better deployment structure could be adopted. The final architecture we adopted is to have an EC2 instance running the webserver (also working as a gateway node) responsible for frontend and backend, while the other nine instances form a KVS cluster including a master and eight workers. Even though they are put in the same region and availability zone, the network latency is still not negligible. A possible solution is using AWS VPC to setup a private virtual cloud and better configure the network traffic to speed up the response.

3.2 Performance issue is hard to resolve

During the project implementation, especially the data processing phase, our system performance is highly restricted by the existing implementation. The difficulties we met are

1) Existing implementations are highly coupled. KVS is built on the webserver, and Flame is built on the KVS. When errors occurred, it's hard to locate the real origin of them. The modification of a single component, even though may solve the current problem we are working on, cannot guarantee new bugs will not be introduced and the other functionality will not be negatively affected.
2) The discrepancy between EC2 instances and our local development environment is also a contributing factor. The different hardware and running environment make it difficult to estimate if a solution behaves well on own computer will also work as expected on the cloud. For example, the page rank computation Flame task works well on our own computers with all KVS workers running locally. But when deployed and distributed to several EC2 instances, the processing speed is much lower, which might be caused by a series of reasons involving the hardware performance, network latency when communicating with KVS nodes, disk I/O overhead, etc.

3.3 Blowup of Flame tables

Another hard-to-solve difficulty is the massive .table files generated during the Flame jobs. Initially, we attempted to compute page rank, TF, and IDF using AWS EC2 instances. However, due to the limited storage space on EC2 (even though extra EBS storage was mounted), these large table files still caused several crashes. This issue was particularly pronounced when using *FlameRDD* or *FlamePairRDD*, which generate persistent transfer tables consuming significant disk space. To address this problem, we ultimately switched to local platforms with better CPU and SSD providing faster read/write and large enough storage space. Despite running all index and page rank jobs locally in several chunks, the process still took over 24 hours and required more than 500GB of disk storage. This problem was primarily due to the implementation of the KVS, which appended row instances rather than overwritin g existing rows in the table file. Although we recognized this issue, we were constrained by time and unable to modify the underlying logic, which would have required a complete rewrite of the KVS.

4. Thoughts and Conclusions

If we are able to do the project again, we think the following things should be done differently

1) Re-implement the components used throughout the project. We think a great change is to decouple them from each other to ensure that they can function independently without any complications. In addition, we could also consider utilizing off-the-shelf frameworks to enhance the performance of our design. For instance, examining the source code of DynamoDB and MongoDB could aid in creating a superior version of our KVS, while studying the source code of Apache Spark could allow us to optimize our analytics engine.
2) A caching component should be added to our web system. This will enhance the search engine's performance by reducing the time taken to retrieve frequently accessed data. The caching component can store the results of previous search queries in a cache, making them readily available for future requests. When a user submits a query, the search engine first checks if the data is available in the cache. If so, the search engine retrieves the data from the cache, rather than executing the entire query again. In the deployment phase, an in-memory caching node, with functionality close to Redis could be added to our system architecture to significantly improve the system performance.
3) More experiments should be conducted on fine-tuning our ranking algorithm. During the past project, we spent our time primarily on improving the search engine's performance. However, with access to better frameworks in a new iteration, we can explore additional factors that may affect the search results ranking. By investing time and resources into refining our ranking algorithm, we can achieve this objective and provide a superior search result to our users.