Backslash R

"Enter your search"

Lily (Xue) Li
Charlene Tam
YongXue (Cindy) Tao
Wangxing Zhao

Introduction and Project Goals

The search engine was built using React and Node.js as the frontend and Java as the backend. Except for third-party libraries mentioned in the report (all approved by the instruction team) and the Java classes made available to us in the homework assignments, all features were implemented with our own code.

Features Enhanced

KVS: File-based table read/write was used to release heap memory for large tables. Garbage collection was added to reduce the file size further.

Crawler: Crawler added features such as using a blacklist table to avoid crawling some known troublesome patterns in URLs, removing duplicates when finding URLs, keeping counts of crawled results by their domain names, and prioritizing the URL to be crawled next by domain name, and the number of pages already crawled. Crawler also could be recovered from prior urlQueue with logs in the event of crashing.

Indexer: Less useful HTML tags like style or script were filtered out before parsing the page content. The total word count of each page was calculated as well so it's useful for computing the TF-IDF score.

PageRank: Enhanced convergence was added as a new feature so that pagerank calculation could converge sooner when a given percentage of entries meet the criteria.

Ranker: The ranker is responsible for ranking crawled pages that contain at least one search term. Our ranking algorithm ranks web pages in the following order:
1. Pages that contain the exact match of the entire search term (i.e. phrase search)
2. Pages that contain a "near" exact match of the entire search term (a near exact match is defined as phrases that have one less word than the original search term, but otherwise maintain the same word order, e.g. the original search term is "hello world cup", and its near exact matches are "hello world", "hello cup" and "world cup")
3. Pages that contain more unique words in the search term (e.g. all else being equal, a page containing 3 unique words from the search term will be ranked higher than one containing 2 unique words from the search term)
4. Pages with higher ranking scores (which is a combined score based on TF-IDF/cosine similarity and page rank)
5. Pages with a lexicographically smaller URL (e.g. all else being equal, a page at the URL "bar.com" will be ranked higher than one at "foo.com")

Based on the final ranking, the ranker will pick the top 100 web pages to display on the client side via a browser.

Frontend/UI: Different components were created or imported from various libraries such as shards-react to make the website more user-friendly. Some examples include:
1. Menubar to ease navigating between pages
2. Progress/loading bar to display that the result is on its way

Features Added

Infinite scrolling: The infinite scrolling feature allows the webpage to display the first five items on the top of the search results to the user. As the user scrolls down the page, more results are loaded. This feature was implemented using React and InfiniteScroll components. The state was initialized to display five items and a hasMore flag. Inside fetch, it will continue to get more data until the length of this state has surpassed the resulting length, it then sets hasMore flag to false. This is when the page will display "End of Search Results."

Shopping tab: The Rainforest API was used to implement the shopping feature on this webpage. Users may click on the shopping tab and search for the item they are interested in shopping for. The shopping tab was implemented using React where the shopping results are being fetched from the Rainforest API. The top 8 results are displayed at a time with pagination at the bottom of the screen. Users may click next to browse through the shopping list.

Phrase search: The ranker supports phrase search, which will assign the highest ranking to web pages that contain the exact match of the entire search term. In addition, the ranker will assign the next highest ranking to pages containing phrases that have one less word than the original search term but otherwise maintain the same word order. For example, for an original search term "hello world cup", the phrases described above include "world cup", "hello cup" and "hello world".

Images search: In addition to anchor tags, the crawler also extracts image tags using regular expressions, and saves the image URL as well as any alt text to a separate table. The images table was designed to use the parsed alt texts as keywords, and return the search result as a list of image URLs to the front end. Due to time constraints, the search is limited to a single term instead of a phrase search and it must come from the dictionary.

Preview Content: A block of the context surrounding the keyword that the user enters into the query is displayed on the result page. This gives the user a bit of a preview of what the website holds.

Secure Connection (TLS): We used Let's Encrypt to generate a signed certificate for the search website, and updated front-end and back-end code to enable secure connections with additional setups.

Technical Challenges and How They Were Overcome

Crawler:

1. Various connection errors - the proper setting of timeout and catch

The original crawler implementation was naive and when it was first given a real URL, it got a lot of exceptions and terminated itself frequently. After investigating the issue by adding logs and researching, we decided to add timeouts for connecting and read attempts for the connection. The various exceptions were buried in multiple layers of codes, and we were able to find where the root cause was from using the logger.

2. Spider traps - use of domain name queue filtering and blacklist

Initially, the crawler could be easily stuck within a domain, and we had to frequently restart the job with new seed links. We could not manually pick out the useless content and it certainly was time-consuming to monitor the progress closely. To resolve that, we developed a feature that processes and extracts the domain name (rather than hostname) for each URL and used the domain name as the identifier to keep track of the pages we have crawled, to assign more weights for the well-known authority and hubs so that the URL queue could maintain a balanced load in each domain name and prioritize the URL that meets the criteria. Additionally, blacklist patterns were identified and added to the blacklist table, so the crawler is trained to prevent itself from falling into some traps when crawling.

3. KVS memory limitation - file-based offset with the append-only approach

With the crawler running on an EC2 instance and accumulating pages, the contents from the crawled page can no longer be saved in memory as it caused an out-of-heap memory exception. It also leads to concurrency issues if there are multiple get/put calls within a short time. As a result, we switched to file-based KVS storage and wrote contents to disk. The conversion freed the memory and we never experienced any memory issues or race conditions.

Indexer: The basic logic of Indexer was simple: we iterate through each row in the content table to parse saved web pages and save the (word, URL: position1 position2, position3) tuples into the Index table. However, how to scale this job up so that we can finish indexing over 300k pages within a reasonable time, without taking up all the disk space turned out to be a challenging task. With the basic implementation from homework, indexing a sample of 20k pages took over 2 hours and the intermediate tables generated during the process took over 50 GB. Therefore, we experimented with several different implementations, and ended up with a 2-step approach: 1) run the Indexer to produce an index_imm file with all words to URL information, but with duplicate words; 2) Run the Consolidator to save rows with the same key in the final Index table. Below are some obstacles we had along the way:

1. The biggest bottleneck in the original implementation was the gigantic intermediate tables created through operations like flatMapToPair. To avoid this, we have each Flame worker process pages in memory and send the output directly to the KVS store.
2. The second bottleneck was that put and get requests to KVS were slow and we were sending put requests for every word on every page, which took a ton of time. So instead of putting a single row in KVS, we batched many rows together and then sent via one put call.
3. To keep only the valid English words and filter out invalid words like />@lcomputer'}, we initially tried storing a static dictionary of about 400k valid English words in a FlameRDD. However, as looking up on a FlameRDD was slow, we instead stored the dictionary at each FlameWorker level so that lookups are local and faster.

<u>PageRank</u>: The main challenge of page rank was the rounds the program had to take before it converged. The way this issue was resolved was by adding a percentage convergent number. Instead of having page rank fully converged, the program stopped when most of the nodes converged.

<u>Ranker</u>: The main challenge with the ranker was how to support phrase search by verifying that a page contains the exact match of the search term. Given the implementation of the enhanced index table, we were able to use the list of indices at which a particular word appears in a particular URL. Specifically, we first picked all the URLs containing all words in the search term. Then for any such URL, we grabbed the list of indices corresponding to each word in the search term. The problem then became checking if there exists a sequence of contiguous indices, in which each index comes from one of the lists of indices corresponding to the search term. This can be solved by looping through the outer list and comparing the indices of each pair of adjacent nested lists at a time.

Most Difficult Aspects

<u>Getting "good" crawl results</u>: One difficult aspect of this project is getting relevant search results from the crawler. If we just let the crawler run without watching it, even though it may crawl many pages, a lot of them are not relevant to the user's search query. In doing this project, we had to "babysit" the crawler and watch the kind of websites that it is crawling down.

<u>Scaling things up is hard</u>: Our first self-implemented version of the server, KVS, Flame, crawler, indexer, and pageRank worked great when tested in the simple and advanced sandboxes. However, when dealing with over 10k pages, all parts of the system slowed down dramatically and resources like memory and disk space were drained. Given the limited time and computing resources we had, we had to rewrite our KVS and indexer so that they were able to finish the job within time and without exploding RAM or disk space.

<u>Improving search speed</u>: The last difficulty we ran into is to deliver the search results to the user within a reasonable time (~20s). Initially, the search for a single term could take up to 5 minutes to show up and a search for a phrase could take up to 20 minutes. Locating the source of the latency was tricky, but with logging and tools like Postman, we found that most of the latency came from sending a large number of queries (2000+ in the case of simple words like "banana") sequentially to get the page rank and word count of all relevant URLs. We then solved the problem by making the Ranker multi-threaded and capping the number of URLs we fetch.

Lessons Learned & Conclusion

In this project, we put together the crawler, indexer, ranker, and front end to create our search engine. Our goal initially was to crawl around 1 million pages but we soon realized that scaling things up is hard. Had we known this, we would have spent less time perfecting our crawler to achieve the 1 million page goal and instead would have tested the indexer and pageRank jobs with larger datasets earlier. So the lesson we learned here is to always keep scale in mind when designing a system with huge datasets. Given more time, we would like to crawl more pages from different seed URLs and watch them more closely to get more links from authorities and hubs. Currently, our indexer only includes words from a given dictionary file so it filtered out results including special words like a person's last name. In the future, to improve search results we would add more terms to the dictionary or find a way to distinguish English words when seeing one. With these improvements, our search engine will become more accurate and robust.