G16-CodeCrafters Report

Qimei Lai(maylaiqm@seas.upenn.edu), Ruxue Yan(yanruxue@seas.upenn.edu)
Tang Gao(tanggao@seas.upenn.edu), Xijie Jiao(xijiej@seas.upenn.edu)
05.05.2024

Introduction

Our goal is to develop a search engine that can efficiently retrieve and index web pages from the Internet and deliver relevant search results in accordance with user queries. We utilized a combination of techniques, such as efficient crawling, text preprocessing and stemming, TF-IDF scoring, PageRank, and result grouping, to provide accurate and well-ranked search results.

Responsibility & Timeline

Qimei Lai: Indexer, Integration, Data Processing, EC2 deployment
Ruxue Yan: Fine-tuned ranking, Searcher, Frontend
Tang Gao: Crawler, Pagerank, Data Processing, EC2 deployment
Xijie Jiao: Pagerank, EC2 deployment

	Week 1	Week 2	Week 3	Week 4
Crawling	small scale crawling tests and debugging based on hw	larger scale crawling and relaunchable crawling	page clean at crawling, blacklist filtering and large scale crawling	continued crawling and EC2 crawler deployment
Data Processing	initial data processing at indexer	further data processing at crawler and indexer	migrate majority of data processing to crawler	
Indexing	indexer testing and debugging on small scale data	indexer testing on large scale data	indexer refinement and indexing of large scale data	final indexing of 1m+ crawled pages and indexer EC2 deployment
Pagerank			Pagerank Debugging	Pagerank on large scale data
Searcher		TF-IDF ranking	integration with pt-index and pt-pageranks	descending in # query words in titles & contents
Frontend	frontend with mocked data		frontend & searcher integration	
Code Integration & EC2 Deployment	webserver and kvs integration	frame integration	crawler and indexer integration	pagerank, searcher and frontend integration & EC2 deployment

Architecture and Implementation Details

Crawler

During the implementation of crawler, we found that we need to find a sweet spot for filtering aggressively while keeping the useful pages you have tried to request. For the exponentially expanding queue, we keep each page a certain max number of child urls and conduct sampling to keep the size reasonably large but with fixed length. The initial blacklist is stored in an in-memory table in the kvs store but as the pattern increases, we encounter deadlock situation and performance issue when calling the expensive scan operation so that we avoided the in-memory table but switch to blacklist some url or host by placing columns in the host and pt-crawl table, fasten the crawl process. For maintaining the polite etiquette of crawling with speed, the crawl-delay and allow attribute in the robots file is cached and the redundancy to parse the robot file is prevented, thus faster speed. In terms of the crawling logic, we try to explore more hostnames provided the limited size of the corpus at 120000 instead of just crawling thousands of pages under a single host. Spider trap alerts are implemented where two pages crawled within a short time's content is compared and we find an interesting website will take in your random url and create a repetitive pattern in the child url where all these urls are leading to the same page content, which is an example of a spider trap. We also do a heavy filtering on the url being parsed, spider-trap-like urls, or urls containing other languages, or account, script, session, ordering related urls are all filtered out. With this optimization, the crawler can crawl up to 20 pages per second with a random hopping to outer space of the internet. But it comes with a cost where each parent link jump too far away and

with a fixed size queue and heavy sampling rate, only a few of the child urls left for the next crawl round, which results in a radial and acyclic topology and a near decay factor pagerank value for nearly all page, because there is no cycles.

Indexer

The indexer first reads the rows from the pt-crawl table into FlameRDD with the fromTable method of FlameContext. Then, it uses the mapToPair method of FlameRDD to convert the data of each row into FlamePairRDD where the key is the url and the value is a string of the url's page content. Next, the indexer uses the flapMap method of FlamePairRDD to produce the inverted index with the word as the key and a string of the list of urls where the word appears in its content as the value.

We initially implemented the majority of data cleaning and processing (including extraction of title, header and metadata, removal of tags and non-main-body page content, etc.) in indexer and it was feasible for 5-6k crawled pages. After experimenting with 10k+ crawled pages, we realized that the memory required for large scale batch indexing is too large and we will not be able to scale up at the indexing step if we don't reduce stored page content size at crawling stage. Therefore, we decided to migrate the majority of data processing into crawler.

Another structural decision we made to ensure scalability and speed is to limit the length of url list to each word. Given some common words can appear in hundreds of thousands of urls when the number of crawled pages grow above 1 million, the memory size required for the url lists of these common words and the url duplication check in these url lists become a bottleneck. In our experiment of unlimited size url list, the indexing speed starts with 10+ page per worker per second to ~1 page per worker per second as the processed pages grow above 3k. Based on experience that people usually do not go beyond the first few pages of search results and experiments with url list size of around 500, 1000, 1500, 10000 and 15000 to test the indexing speed, we decided to limit the length of url list string to 100,000 characters (around 1000-1500 urls) given the usual url length is 40-100 characters to ensure efficient memory use, fast indexing speed and relevant query results. When the url list string length is exceeded, there are two approaches to how the size will be contained at around 1000-1500 urls. The first approach is to choose the one with larger word count between the last url in the existing list and the new url. The second is to sort all urls in the list plus the new url based on word count and retain the top 1000 of them in the list. We assign 95% for the first approach and 5% chance for the second approach and use a pseudo-random number generator to decide which approach to go with.

We also incorporated a list of 170 stop words and implemented word stemming in the indexer to enhance memory efficiency and query result breath and relevance.

PageRank

The computation of PageRank is simplified since the crawling results were sent to PageRank with only two main columns: main URL and child URL. We follow HW9 to do the PageRank with a damping factor set to 0.85. The results of most URLs are very close to 0.15. The reason for this is that to keep the queue size within a limit; we set the sampling rate low so that most child URLs will be dropped during the iteration. This will lead to the computing graph differing significantly from the true web topology. These dead nodes will lead to low PageRank values being updated and will also take away PageRank value from the cycle along with their process of being dropped.

Searcher

The Searcher is responsible for processing user queries, retrieving relevant documents from the index, and ranking the search results. The code defines a set of stopwords, which are common words that are filtered out during text processing. It initializes an IDF cache and a list of suggestions by scanning the pt-index table in the Key-Value Store (KVS) during the initialization phase. When a user submits a search query through the /search endpoint, the query string is preprocessed by removing HTML tags, special characters, converting to lowercase, and removing stopwords. The remaining query words are stemmed using the Porter Stemmer algorithm, which helps in matching words with different grammatical forms and improving the recall of the search engine. The Searcher then computes the TF-IDF scores for each word-URL combination. For each stemmed query word, the code retrieves the corresponding row from the pt-index table in the KVS, which contains a comma-separated list of URLs and their term frequencies. The Normalized TF is calculated as log10(1+term_frequency), which addresses the issue of very high term frequencies and makes the TF value less sensitive to high frequencies. The IDF for each word is retrieved from the idfCache using the formula log10(1+1500/document_frequency), where 1500 is the limitation we set manually on the number of URLs per word for faster indexing. The TF-IDF score for each word-URL combination is calculated as TF * IDF, and simply added together. A threshold of 0.4 is applied, and only word-URL combinations with TF-IDF scores above the threshold are kept. The TF-IDF and PageRank scores are combined using a weighted sum: final_score = 0.15 * PageRank + 0.85 * TF-IDF.

The search results are stored and sorted in a priority queue based on their combined scores. The Searcher iterates over the sorted results, retrieves the corresponding row from the pt-crawl table in the KVS, and extracts the URL, title, and page content. The search results are then grouped based on the number of matching words in the title and content using a nested map structure. The Searcher sorts the search results based on the number of matching words in the title in descending order. Within each title group, the results are further sorted based on the number of matching words in the content in descending order. If an exact match between the query and the document title is found, that result is prioritized and placed at the top of the ranking. It ensures that more relevant results are displayed first, even if they have slightly higher TF-IDF scores. Finally, the sorted search results are added to a response map, which is converted to a JSON string and returned as the response to the client with appropriate headers.
The Searcher also handles requests for suggestions through the /words endpoint. The suggestions list, which was populated during initialization, is converted to a JSON array and returned as the response.

Scalability & Fault Tolerance

Our architecture is designed to be scalable in several ways, the use of a distributed KVS allows for horizontal scaling by adding more nodes to the cluster, increasing the total storage and processing capacity by putting in different kvs workers. Evenly distributed keys ensure that both kvs workers and flame workers can execute the jobs in parallel. The indexing and search processes can be parallelized by dividing the workload across multiple nodes or threads. The frontend can be scaled horizontally by deploying multiple instances behind a load balancer to handle increased user traffic. While our current implementation does not have explicit fault tolerance mechanisms, the use of a distributed KVS provides some resilience to node failures. If a node fails, its data can be recovered from replicas on other nodes, ensuring data availability and consistency.

Extra Features

Cached pages: The crawled pages' key content, such as the title, h1 to h5 tags, and main content, are abstracted and shown as a preview of the pages being crawled or crawled.
Infinite Scrolling: The frontend supports infinite scrolling, where additional 10 search results are loaded as the user reaches the bottom of the page.
Spellcheck: The spell check function uses the levenshteinDistance function to calculate the edit distance between the input and each word in the dictionary. The function returns the word from the dictionary with the minimum edit distance from the input to suggest the closest matching word from the dictionary
Search suggestions: It calls the spell check function with the user's input and an array of suggestions. It creates autocomplete suggestion elements for up to 10 matching suggestions, highlighting the portion of the suggestion that matches the user's input.

Ranking

TF-IDF: This feature measures the relevance of a document to a given query based on the frequency of query terms in the document and the rarity of those terms across the entire document collection. The TF-IDF score for a word-document pair is calculated as: (1 + log(term_frequency)) * log(1500 / document_frequency). The term frequency is the number of occurrences of the word in the document, and the document frequency is the number of documents containing the word.
PageRank: The PageRank score represents the importance and authority of a web page based on its link structure. Pages with higher PageRank scores are considered more authoritative and ranked more. The PageRank scores were calculated using the same algorithm in HW9, an iterative algorithm based on the link structure of the web. The PageRank results are saved for each url.
Final Score: The TF-IDF scores for each query word and document URL combination are calculated and summed. The summed TF-IDF score is combined with the PageRank score using a weighted sum: final_score = 0.15 * PageRank + 0.85 * TF-IDF. The search results are sorted in descending order of their final scores.
Matching Words in Title and Content: Within groups of results with the same number of matching words between the query and the document title, the results are further sorted in descending order of the number of matching words in the content. If an exact match between the query and the document title is found, that result is prioritized and placed at the top of the ranking, which avoids focusing on documents containing partly query words with high term frequencies.

Evaluation

Crawling Performance: The initial version run on a single worker yields a crawl speed of around 3 pages per second at first with an exponentially grown frontier queue. After executing a fixed size queue with sampling and running the crawler on 8 workers, the final speed for our corpus containing around 1220000 is downloaded within 20 hrs, etc. around 17 pages per second.
Indexing and Crawling Performance: Our experiments showed that with a single node and a single thread, we could index approximately 50 pages per second. With 10 nodes and 20 threads, the throughput increased to around 400 pages per second, demonstrating good scalability.
PageRank Computation Time: Computed PageRank scores over 1000 URLs (test for speed) with a single node took around 1.5 seconds, with convergence typically achieved within 3 iterations. Introducing additional nodes showed a near-linear reduction in computation time, indicating good scalability. For the final pagerank on the whole corpus of 1220000 pages, the runtime for each PageRank update iteration is 8 minutes, for a total of 4 iterations of 30 minutes.
Search Query Response Time: We measured the average time taken to complete a range of different search queries, with and without certain expensive features enabled. The average response time for simple search queries without expensive features was around 50ms. When PageRank integration and result grouping were enabled, the average response time increased to around 200ms. However, disabling result grouping brought the response time down to around 100ms, suggesting that result grouping is a relatively expensive operation.

The indexing and crawling scaled effectively with more nodes and threads. PageRank computation also improved with additional nodes, making it feasible for large web graphs. Although search query response times were generally good, enabling certain ranking features impacted performance. Optimization opportunities exist based on query complexity. Overall, our search engine showed strong performance and scalability, delivering relevant results.

Lesson Learned

We consider this project to be a success overall. We were able to develop a functional search engine with core features to return suggestions and results. The search results were generally relevant and well-ranked. The modular design and the use of key data structures like the inverted index and TF-IDF scores facilitated efficient implementation and scalability. Implementing the ranking algorithm and result grouping logic was relatively complex and required careful consideration of performance implications. The suggestion feature could be further improved by incorporating more advanced techniques like prefix-based matching or machine learning models for better quality. While our system demonstrated good scalability, we did not implement explicit fault tolerance mechanisms, which could be a potential area for improvement.