CIS 5550 FINAL REPORT
PennGO

Xiaowei Wu, Minyi Hu, Jinwei Bi, Haochen Gao

Dec 18, 2023

1. Introduction

Our team approached the CIS 5550 project with a focus on creating a robust, scalable cloud-based search engine. Our strategy involved iterative development and integration of various components developed during the course. We emphasized scalability to handle different loads efficiently and designed the system with user experience at the forefront.

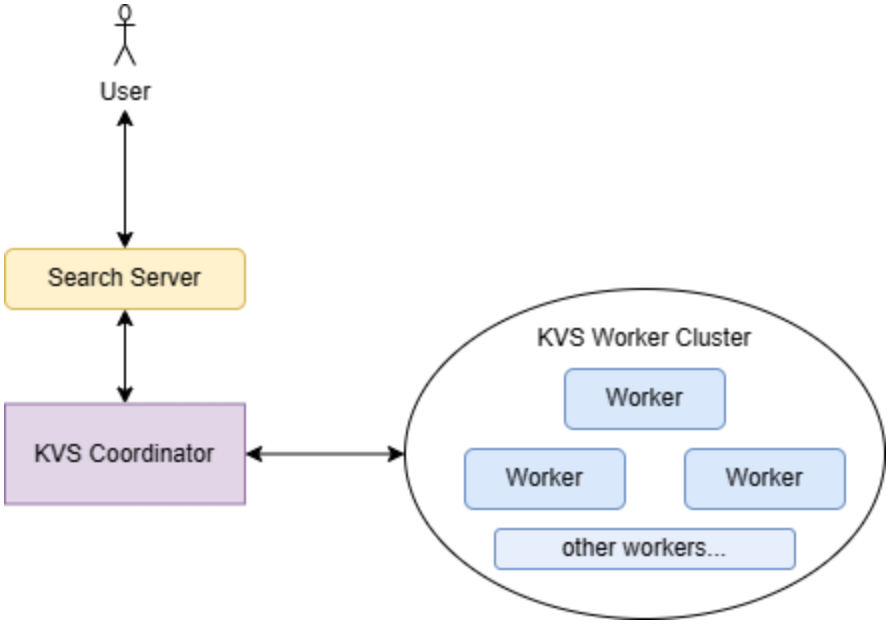
We divided the project into four key components, assigning one to each team member:
1. Web Crawler: Xiaowei Wu was responsible for the development of an efficient web crawler. Her role involved meticulously navigating and crawling websites, with a keen emphasis on complying with web standards and maintaining adherence to politeness policies.
2. Indexer & PageRank: Minyi Hu's work centers on data processing. The indexer retrieves URLs and pages from the pt-crawl table, transforms them into FlameRDDs, and processes content. The PageRank section handles URL normalization, hyperlink extraction, page rank calculation, and state updating for subsequent computations.
3. Ranking: Haochen Gao was in charge of the PageRank algorithm and deployment on EC2, vital for ranking web pages based on relevance and authority.
4. Frontend Development: Jinwei Bi's contributions to the design of a Google-like user interface with a seamless query processing mechanism and user-friendly display of search results, including features like Autocomplete and Predictive Search and spelling check.

Regular team meetings facilitated synchronization and collaboration, ensuring a cohesive and functional search engine with well-integrated components. We held bi-weekly meetings, each lasting approximately two hours. In preparation for the presentation, we dedicated two full days to collaboratively debug and conduct final testing of our project.

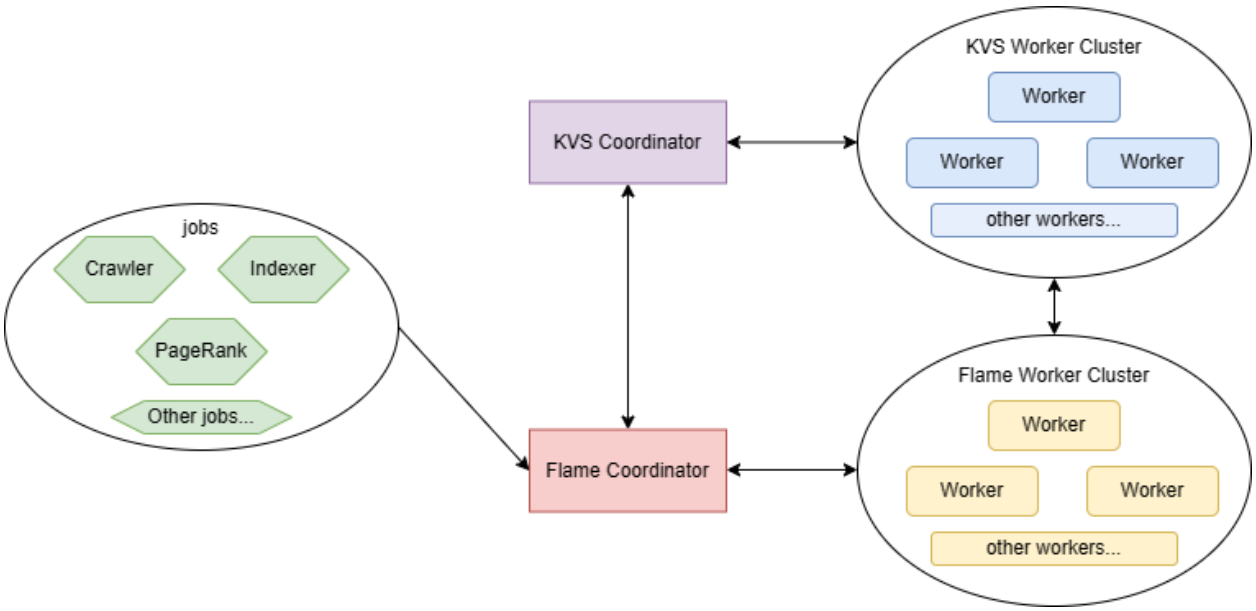
2. Architecture

2.1 Search Service:

We use a storage cluster to store indices, PageRank values, and raw web pages. This cluster is scalable because we can shard the data across different workers. As we add more workers to the cluster, our storage capacity increases. Similarly, we can add more frontend servers to handle a greater volume of incoming requests. We have observed that search queries do not change the state of our backend nodes since they are read-only, making it relatively straightforward to scale the search service. However, the Key-Value Store (KVS) Coordinator becomes a bottleneck at a certain point, indicating that a more advanced architecture is needed.



2.2 Data processing pipeline



In our data processing system, we have a flame coordinator to submit jobs(jar files containing a specific computation purpose). Use all workers in the flame worker cluster to execute them in parallel. Thanks to this feature, our indexing process and crawling can be executed in parallel and has the potential to process big data. Like the search service, this system can scale well, but coordinators will be bottlenecks at a certain stage.

As our KVS workers have a replication management protocol, it can support recovery and fault tolerance. That may be beneficial for future development of some functionalities.

Our data processing starts with crawling - getting raw data, before generating pagerank and indices. Then the intermediate data will be stored in the database of the data processing system. After all processing is done, final data will be stored into the database of search system.

The data structures we used:
URL Frontier Queue: This queue managed the URLs to be crawled. It employed a priority-based system to ensure efficient crawling of relevant and authoritative pages first.
ConcurrentHashMap: This is used in the data storage and access module for providing high concurrency and thread-safe key-value pair storage and retrieval, making it valuable in applications requiring high concurrency and performance, such as caching, data storage, and indexing.
JSON Array: Used for data exchange between frontend and backend, it facilitates data transfer and sharing. The backend sends data in JSON array format upon frontend request, enabling easy processing and display of results.
Flame: FlameRDD, FlamePairRDD, and FlamePair are specialized structures for large-scale and distributed data handling, offering fault tolerance, efficient processing, and support for key-value pair operations like mapping, filtering, aggregation, and parallel processing.

When faced with design choices, we prioritized scalability and fault tolerance. For instance, choosing a priority queue for the URL frontier over a simple FIFO queue was driven by the need for intelligent prioritization of URLs, which is crucial for large-scale web crawling.
Our architecture's scalability is evident in its distributed nature. By deploying multiple workers on different nodes, the workload gets evenly distributed, allowing the crawler to efficiently handle larger portions of the web. This design also facilitates easy scaling up or down based on the crawling demands.
We implemented robust error handling to manage server errors, timeouts, and invalid URLs, preventing crashes and ensuring continuous operation. By saving the state of the URL frontier, our crawler can resume from where it left off in case of a shutdown or crash, rather than restarting from scratch. The use of concurrent data structures and thread-safe operations minimizes the risks associated with multi-threading, such as race conditions, enhancing the stability of the crawler. For extra features, our system supports Autocomplete and Predictive Search as well as spelling check.

3. Ranking

To determine the ranking, we use features such as TF-IDF and PageRank, as well as the match between the search query and the document's title, which provides an additional score for the final ranking.

The final ranking score is:

$$\text{Score} = \text{TF-IDF} * \text{pagerank} + \text{title match score}$$

To compute the final ranking based on various scores, sort the scores in descending order.

4. Evaluation

Single Node (k=1) with Different Threads: 10 Threads: Achieved around 2-3 pages per second. This reflects efficient utilization of resources in a local, single-node setup. 20 Threads: Slightly higher rates were observed, averaging around 4-5 pages per second. The increased thread count allowed for more parallel processing but also led to higher resource contention. 30 Threads: The performance plateaued around 6-7 pages per second. Beyond this point, the benefits of additional threads diminished due to the limits of the single-node's processing capabilities.
Multiple Nodes (k=2, 3, ...) with 10 Threads Each: 2 Nodes: Doubling the nodes led to a near-linear increase in performance, achieving around 4-6 pages per second. This showcased the scalability of the crawler in a distributed environment. 3 Nodes: With three nodes, the crawler could process approximately 6-8 pages per second, maintaining the trend of linear scalability.
More Nodes: As we increased the number of nodes further, we continued to observe a proportional increase in the page crawling rate, underscoring the crawler's efficiency in a distributed setup. It's important to note that we primarily crawled pages from Wikipedia. When crawling more complex websites like CNN, the speed decreased, indicating the crawler's varying efficiency depending on the website's complexity and structure.

Search result: The evaluation of our search results indicates that all returned items are relevant, with the most pertinent results appearing at the forefront. The service consistently returns results within approximately three seconds and maintains uninterrupted operation without crashes.

5. Lessons Learned

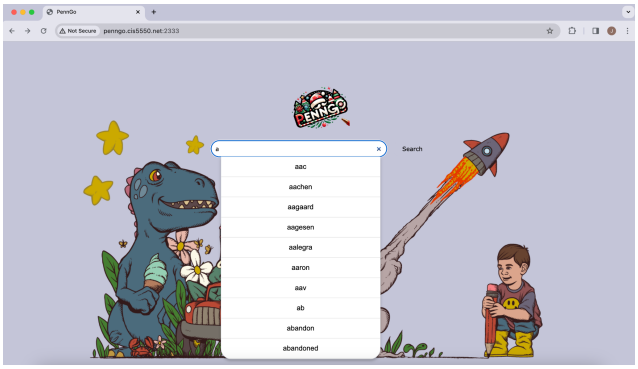
The crawler effectively navigated and indexed a significant number of web pages, adhering to web standards and politeness policies. It managed to crawl a diverse range of content from Wikipedia, BBC, and CNN, demonstrating its capability to handle different web architectures and content types. The crawler excelled in efficiently directing its efforts towards relevant content, thanks to its effective URL prioritization and filtering systems. Parallel processing through concurrency and multithreading, coupled with defined connection and read timeouts, optimized resource use. Its ability to restart using a persistent URL frontier and the deployment of multiple workers on EC2 notably improved its scale and speed. Additionally, enhanced URL normalization significantly boosted its accuracy. However, the crawler struggled with JavaScript-heavy sites and memory management under heavy loads. Its error handling for unexpected server behaviors and timeouts also required further refinement to improve overall robustness and operational continuity.

Reflecting on our crawler development, we see clear avenues for improvement. Advanced URL filtering and prioritization algorithms focusing on content relevance and domain authority would optimize link-following decisions. Enhancing content parsing abilities for diverse web content types, and incorporating machine learning for pattern recognition, could significantly improve link-following strategies. Crucially, scalability enhancements are vital for distributed system operation, accommodating web growth and complexity. This involves distributed crawling for balanced workloads, dynamic resource allocation, efficient queue management, scalable data storage

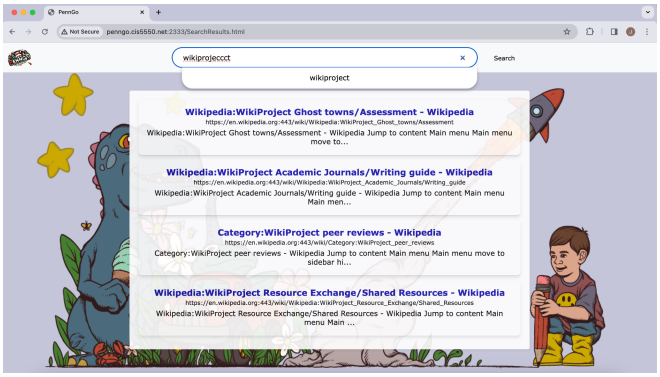
solutions, and adaptive crawling rate control. These collective improvements would greatly boost the crawler's performance and adaptability in a dynamic web environment.
In our continuous efforts to innovate and enhance the user experience, we plan to introduce a series of advanced features into the frontend design of our web browser. Among the most crucial is the integration of Voice Search, a cutting-edge technology that will allow users to conduct searches using voice commands, offering a hands-free, efficient, and accessible search experience. Additionally, we aim to implement Location-based Results, ensuring that search outcomes are customized according to the user's geographical location, providing more relevant and localized content.

6. Screenshots for some basic and extra features

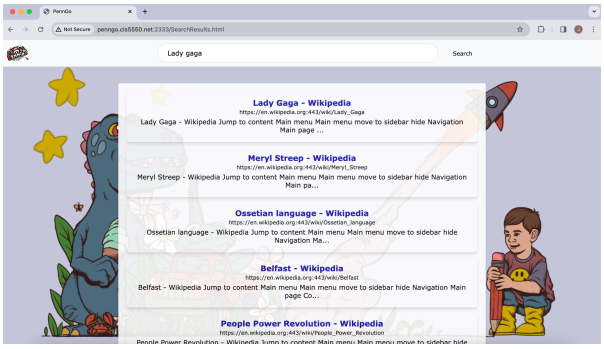
Main Page



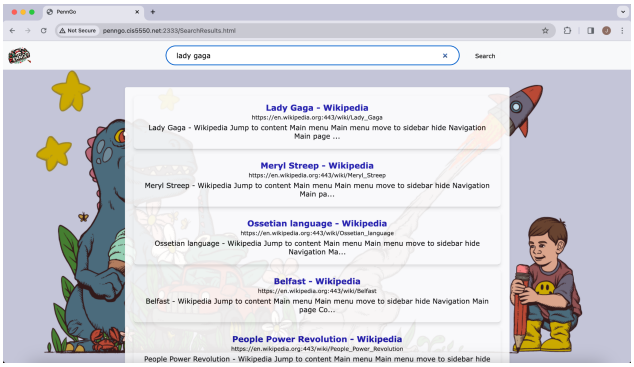
Autocomplete and Predictive Search



Search Results and spell check



Search Results



Phrase Search