CIS 5550 Final Project

Eloic Lacomme Ruiz, Jinze Wang, Yuxiang Ying, Tianmei Liu

Table of Contents

Introduction......2
Architecture......3
Frontend......5
Ranking......5
Evaluation......6
Lessons Learned......8

Introduction

In undertaking this project, the development timeline was segmented into four distinct phases, each spanning approximately one week. This structured approach enabled efficient task allocation but also ensured a cohesive integration of the different components. Below is an outline of the roles and responsibilities, followed by an account of our weekly progress.

Division of Labor

Crawler: Eloic Lacomem Ruiz
Preprocessing: Tianmei Liu
Ranking: Jinze Wang
Frontend: Yuxiang Ying

Timeline

Week 1: Initial Scope

Each team member dedicated this week to setting up and modularizing their respective components. Notably, significant enhancements were made to the crawler, which was restructured for improved resilience. This initial phase also involved testing with data beyond our predefined test cases to enhance fault tolerance.

Week 2: Integration

This week marked the beginning of our integration efforts. The crawler successfully compiled a corpus of five thousand web pages, which were then processed and tested. Concurrently, the EC2 instance was configured to support live deployment tests of the frontend.

Week 3: Scalability

With the preprocessing indexer adeptly managing initial data batches, the scale of operations were increased. Enhancements to the crawler further increased its speed and fault-tolerance, enabling it to process tens of thousands of pages. However, this scale-up introduced performance bottlenecks in the indexer, manifesting as timeouts, which necessitated further optimization.

Week 4: Tuning

The final week was dedicated to refining the system. The crawler's capability was augmented to fetch up to one hundred thousand pages. Improvements to the preprocessing components allowed for efficient handling of increased workloads. Meanwhile, the frontend was fine-tuned to ensure the accurate presentation of search results.

Architecture

Outline of Major Components

The search engine is broken up into four major components: the crawler, the suite of preprocessing tools, the ranker, and the frontend. The crawler searches the web for web pages, which are uploaded and stored on EC2, and sent to the indexer for word processing. The indexer forwards the resulting data into further components, TF-IDF and Page Rank, which compute those two respective metrics. The combined results are sent to the ranker, which the frontend uses to return the results of each query.

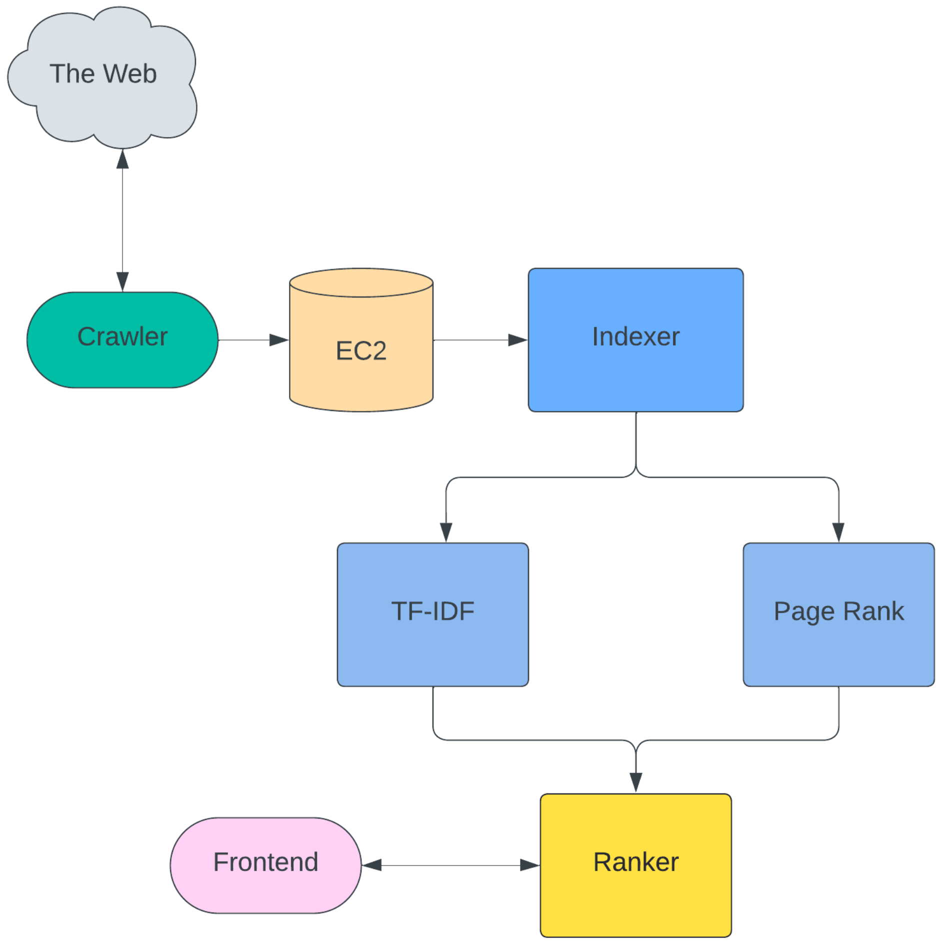

Fig.1 Flowchart detailing the search engine's major components and their interactions.

Detailed Description of Components

Crawler

The Crawler employs a Mercator-style architecture where a queue represents the expanding URL frontier. URLs are distributed among workers based on the hash of the URL. To enhance resilience against slow or offline servers, the crawler implements multiple timeout systems. It also includes several layers of language filtering to exclude non-English pages. Additionally, a filtering system is in place to blacklist certain websites that could trap the crawler or lead to pages with valid responses but no valuable content. To ensure a broad coverage of pages, a limit is set on the number of pages that can be captured from a single domain.

Indexer

The Indexer class processes and organizes web page content to facilitate efficient information retrieval. The words from each of the pages in the pt-crawl table are extracted, filtered, and transformed. The text was filtered through a predefined list of stopwords and cross-referenced against an English words dictionary to ensure only relevant words are processed. Each word is then stemmed to its root form using the Porter stemmer, standardizing variations for more effective indexing. Subsequent steps involve mapping these words along with their URLs and specific positions within documents in parallel using the Flame framework. This data is aggregated and stored back into the KVS under the pt-index table, creating an efficiently accessible index.

TF-IDF

Extending on this indexer class, the tfIdf class calculates the term frequency-inverse document frequency (TF-IDF) scores, enhancing the system's capability to quantify the relevance of words within the document corpus. It begins by loading the previously indexed data from the KVS and uses Flame framework to compute the Inverse Document Frequency (IDF) for each word in parallel, determining its rarity and significance across all documents. Simultaneously, it recalculates each word’s term frequency from the original documents, normalizing these figures against document length to adjust for document size variations. The final TF-IDF score is derived by multiplying the TF and IDF values, with increased weights for words found in sections like titles and headers, emphasizing their importance. These calculated scores are stored in the KVS within the pt-final table for the subsequent ranking. This class underwent several stages of optimization to ensure greater scalability as the number of pages crawled increased. Specifically, intermediate tables were removed, and the total number of computations was reduced to decrease the overall runtime of the process.

PageRank

The PageRank class in the Java package implements the PageRank algorithm, which is central to efficiently ranking the relevance of web pages in a network. The process begins by extracting web pages from the pt-crawl table, where each page's URL is hashed and linked with its initial rank and outgoing links parsed from the page content. This setup initializes the rank values and establishes the connectivity graph essential for the PageRank computation. As the PageRank iterations proceed, each web page distributes its rank across its outgoing links, contributing to the rank of linked pages in the next iteration. This distribution is calculated and managed using Flame's RDD and PairRDD functionalities, allowing for the parallel processing of large data sets across a distributed system. The ranks are periodically updated until convergence criteria are met—defined by a threshold for minimal rank change and a percentage of pages meeting this criterion. Upon convergence, the final ranks are stored back into the KVS under the pt-pageranks table for the subsequent ranking.

Frontend

The primary components of the frontend implementation consist of the User Interface (UI), the API server, and the Key-Value Store (KVS). The process begins when clients submit queries through the frontend, which are then forwarded to the Rank API server operating on port 9001. This server retrieves ranking data from the KVS system and relays it back to the frontend. The frontend organizes the URLs based on their ranks and forwards them to the Title API to fetch corresponding titles. After receiving the titles, the frontend presents the final results to the user. To bolster fault tolerance and handle potential crashes, the frontend server is deployed across multiple EC2 instances connected to a load balancer. This configuration evenly distributes user load across various servers, thereby enhancing both reliability and performance. While we are capable of enabling HTTPS access, we lack sufficient permissions to create an AWS Certificate Manager certificate for the Load Balancer. Hence, we've opted to utilize HTTP for accessing our website.

Ranking

The ranking process primarily uses two metrics: PageRank and TF-IDF. PageRank assesses the relative importance or quality of each webpage, while TF-IDF measures how relevant a webpage is to a given keyword.

When a user's query is received, the Ranker performs similar preprocessing steps as the rest of the system: splitting the query into words, removing stopwords, and stemming. It then extracts keywords to identify relevant webpages. For each web page containing a keyword, we calculate the product of its PageRank and the TF-IDF score of the keyword on that page. This product forms the basis of the ranking score for that page regarding that keyword.

The ranking scores for each keyword are summed across all web pages. This cumulative score determines the final ranking, with pages featuring multiple relevant keywords achieving higher scores. The results are then sorted by these scores to determine the order in which web pages are presented to the user.

To optimize performance and scalability, the Ranker is designed as a server-like worker. It binds to a port and listens for incoming requests from the Frontend. Utilizing a Key-Value Store (KVS) model, keywords are used as row keys in a table, with URLs as column keys and rank-values as column values. This setup allows for rapid retrieval and updating of rank values.
Pre-computation of rankings minimizes runtime calculations, enhancing response times. For scalability, additional Ranker instances can be deployed alongside a coordinator, similar to other distributed systems like KVS and Flame, to handle increased load.

Evaluation

Crawler Testing

The crawler was evaluated for its performance over time using a varying number of nodes. **Fig.1** illustrates four ten-minute trials of the crawler, each of which is using a greater number of nodes. The figure shows that the greater the number of nodes used, the more pages that were crawled. For instance, at the 10 minute mark the crawler with one node found 1300 pages, with two nodes found 1858 pages, 2411 pages with three nodes, and 3045 pages with four nodes. This upward relationship demonstrates that increasing the number of nodes distributes the workload between more workers, leading to better performance.

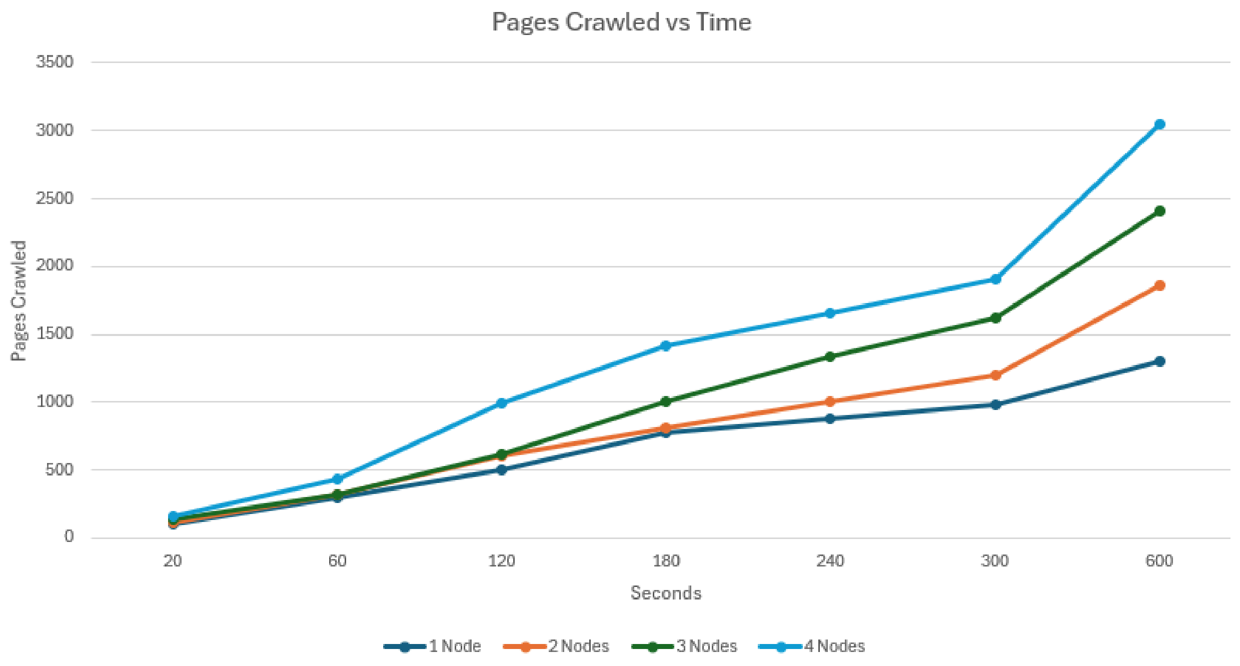

Fig.1 Graph showing pages crawled against time with varying number of nodes

We additionally noticed that as time went on, the rate at which the crawler stored pages slowed down. Taking a look at the number of pages crawled per second in **Fig.2**, the crawler reaches its highest point at the start of its cycle, between 5 and 7.7, and slowly decreases as time elapses. This throttling is likely caused by the increasing size of the queue, which leads to greater time to add and remove urls as the url frontier keeps increasing. Through manual observations, the crawler was able to collect around 20,000 pages in about two hours, and about 50,000 pages in six hours.

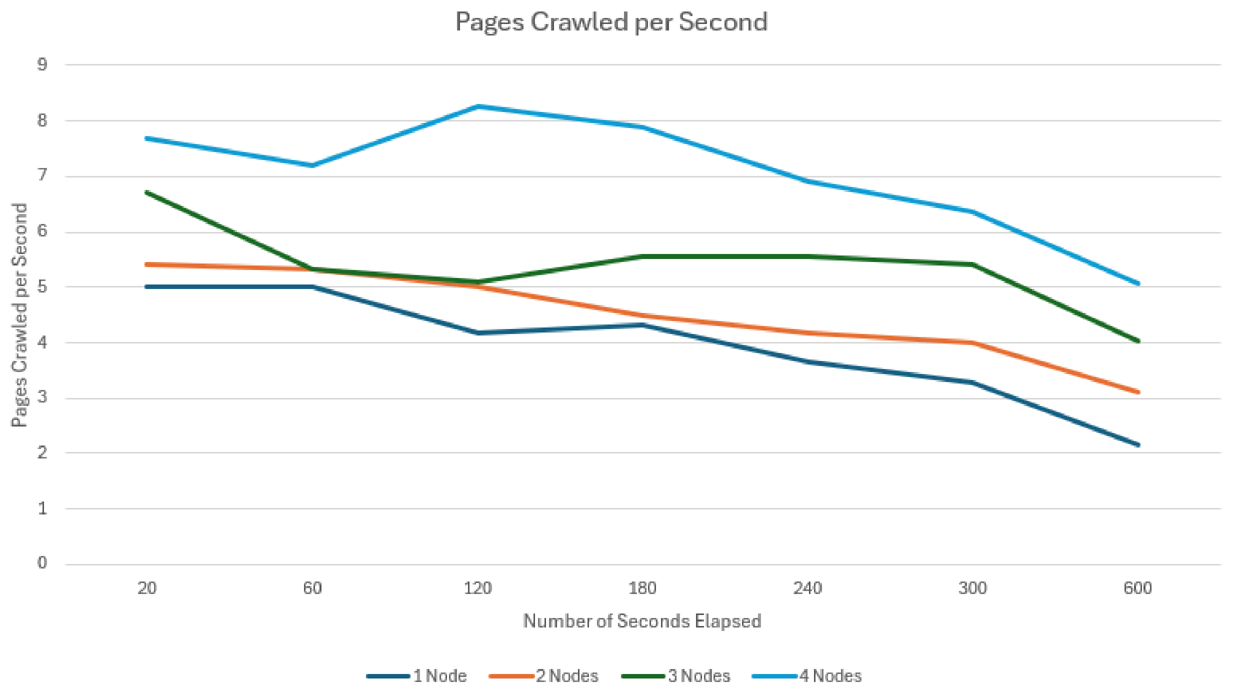

Fig.2 Graph showing the number of pages crawled per second as time elapses for varying numbers of nodes

Preprocessing Testing

The runtimes of both the indexer and TF-IDF were tested against various corpus sizes, and the results are shown in **Fig.3**. When using smaller corpuses, tf-idf tends to run slower than the indexer by less than one minute, which highlights how at this scale both classes perform about equally. However, when testing against larger corpus sizes, such as above 100k, the runtime for the indexer largely outweighs the runtime for tf-idf, which became a bottleneck in the system. This discrepancy in runtimes was partly due to optimizations being performed on TF-IDF early on when it was detected to be the limiting factor with larger corpus sizes. These optimizations led to TF-IDF running at 1.5 times the speed it originally ran at. However, since indexer issues were not detected until a larger scale, this class was not optimized as much as the others, leading to these results.

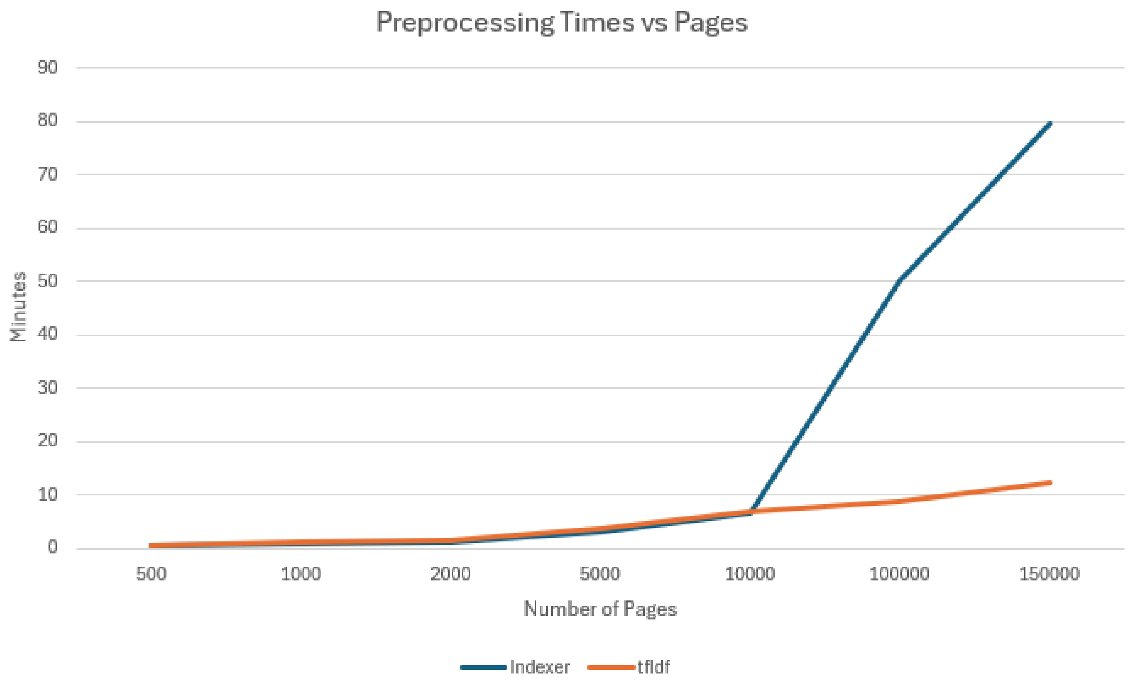

Fig.3 Graph of time taken to run indexer and tf-idf at several corpus sizes.

Search Testing

To test the efficiency of the frontend to display queries, ten different searches were chosen, each of which were performed five times, and the results were averaged out. Across the trials, the frontend was able to display results at an average of 1.85 seconds, finding between 80 and 100 results for each phrase. The speed of the frontend is partly due to the batching approach taken where results are loaded using infinite scroll in fixed sized batches, which allows the frontend to delay displaying all results until necessary.

Lessons Learned

Our search engine project has demonstrated notable capabilities and effectiveness in handling basic queries. The crawler proved to be exceptionally robust, efficiently indexing tens of thousands of relevant pages within a few hours and seamlessly forwarding them to the indexer. The preprocessing components effectively extracted and computed relevant information from the data, with results promptly displayed by the frontend.

Despite these successes, the project also highlighted areas for improvement. One major takeaway is the importance of early and frequent integration. This approach would allow us to identify and address scalability issues more effectively as the project progresses. Early integration testing would have likely mitigated some of the unexpected setbacks we encountered, leading to a more refined and robust final product.