Search Engine

Álvaro Rodríguez González, Lucía Afonso Medina
Alejandro Alemán Alemán, Farid Sánchez Belmadi
Néstor Ortega Pérez

November 2024

Professionalization of the Project with Java Code

Contents

Abstract

In this project, we designed and implemented a search engine that incorporates three core modules: a crawler, an indexer, and a query engine. The crawler retrieves and filters books from Project Gutenberg, the indexer creates an efficient data structure for storing and retrieving keywords, and the query engine allows users to search for phrases and retrieve relevant metadata. Two indexing approaches were evaluated: storing all words in a single file versus storing each word in its own file. Comprehensive benchmarks were conducted to assess execution time, scalability, and efficiency for both methods, as well as to compare the performance of the system implemented in Python and Java. This work underscores the importance of modular architectures, optimized data structures, and parallelization techniques in building scalable and efficient search engines.

1 Introduction

Search engines play a vital role in efficiently retrieving information from large datasets, making them essential for applications such as research, text analysis, and digital libraries. This project aims to design a scalable and efficient search engine capable of handling large volumes of text data, with a focus on optimizing performance and modularity.

The system has been professionally implemented in Java and consists of three main modules: the crawler, responsible for retrieving books from Project Gutenberg and filtering them by language; the indexer, which processes the books to build an inverted index for efficient retrieval; and the query engine, which enables users to search for phrases and access relevant metadata. Each module has been designed with a strong focus on modularity, ensuring both flexibility and scalability.

To evaluate performance, two indexing methods were compared: storing all words in a single file versus creating a separate file for each word. We compared the execution times and scalability between each data structure in both languages, Java and Python. Benchmarks were conducted on the crawler, both indexing methods and query execution, revealing key trade-offs between modularity, scalability, and performance.

This document provides a detailed overview of the system's architecture, the design decisions made, and the benchmarking results. It concludes with recommendations for optimizing search engine performance and suggestions for future enhancements, including cloud integration and advanced search techniques.

2 Modules and data structures

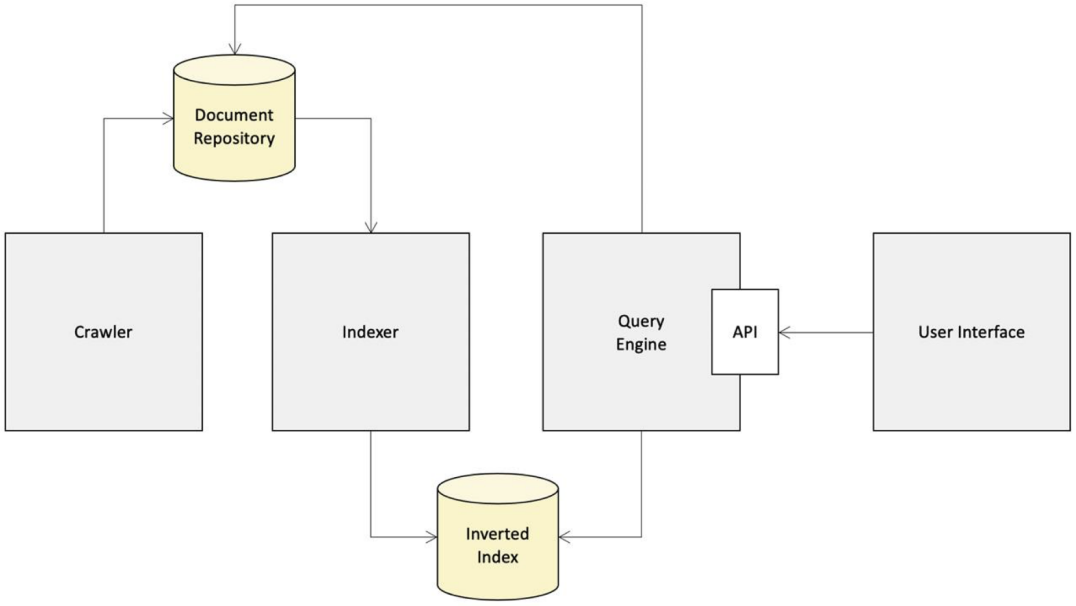

Figure 1: Module and Repository Structure of the Application

2.1 Crawler Module

The purpose of this module is to download a specified number of books from Project Gutenberg, filter them by language, and store the valid books locally. The module selects books randomly by generating unique book IDs and ensures that only books in English, Spanish, or French are retained. If a book fails to meet the language criteria or cannot be downloaded, the module skips it and continues until the required number of books is retrieved.

2.1.1 The `crawler()` Function

The core functionality of the module is managed by the `crawler()` function. This function automates the process of retrieving, filtering, and saving books, using the following steps:

- **Input:** The function takes two parameters, `num_of_books`, which defines how many books should be downloaded and `datalakepath` which saves the books in the datalake directory.
- **Random Book Selection:** The function generates random numbers between 1 and 99,999 to select books from Project Gutenberg. These numbers correspond to the unique IDs of books in Project Gutenberg's catalog.
- **Book Download:** For each randomly generated book ID, the function constructs a download URL and attempts to retrieve the book's plain text version using the `requests` library.
- **Language Filtering:** Once a book is successfully downloaded, the function verifies its language using a custom `language_filter()` function. This filter checks whether the book is written in English, Spanish, or French, based on the content of the first 50 lines.
- **Saving Valid Books:** If the book passes the language filter, it is saved in the `datalake` directory with a filename corresponding to its book ID (e.g., `12345.txt`). The file is saved in UTF-8 encoding to handle special characters.

- **Skipping Invalid Books:** If the book is not in the desired languages or cannot be downloaded due to errors (e.g., an invalid book ID), the function logs an appropriate error message and skips to the next book.

2.1.2 Enhancements in the Updated Module

Two significant improvements have been incorporated to enhance the performance and maintainability of the module:

- **Refactoring to an Interface-Based Structure:** The module's architecture has been refactored to adopt an interface-based design, promoting modularity and scalability.
  - **Modular Components:**
    - `Downloader Interface`: Defines methods for downloading book content, enabling interchangeable download implementations.
    - `Filter Interface`: Specifies language filtering rules, allowing new languages to be added without changes to the core logic.
    - `CrawlerController Interface`: Manages the overall control flow, ensuring a clear separation of responsibilities.
  - **Advantages:** This design improves flexibility, maintainability, and testability. New components, such as additional language filters or storage mechanisms, can be integrated without altering existing code.

With these enhancements, the module provides improved efficiency through parallel processing and a more robust, maintainable code structure by employing interface-based design.

2.2 Indexer Module

In a search engine, an *indexer* processes the raw data (e.g., books or text) and creates an *inverted index*, which allows for fast information retrieval based on keywords. Instead of searching through all the text for each query, the search engine refers to a pre-built index that lists words and their locations within the text.

2.2.1 Inverted Index

An inverted index is essentially a mapping where each word is associated with the locations (e.g., document IDs, line numbers) where it appears. This enables fast keyword searching, as the search engine can refer to the word's locations directly without needing to scan the entire dataset.

2.3 Query Engine Module

The purpose of this module is to process and analyze words entered by the user, searching for their occurrence in a set of books. Once the words are found, the module extracts metadata from the books, such as the title, author, year, language, and download link. It also reads and presents specific lines from the books where the words appear. All of this is done efficiently through a modular structure and the use of classes, keeping the code clean, reusable, and easy to maintain.

This module is particularly useful for analyzing large volumes of text, searching for keywords, and retrieving detailed information about the books where these words are found. It could be employed in text analysis, research, or applications that need to process and retrieve specific information from large text datasets.

Implemented Improvements: One significant enhancement in the module is the incorporation of parallelization to optimize processing performance, especially when handling large datasets or multiple words in a single query. By leveraging Java streams and their parallel capabilities, the module can process multiple words concurrently. This approach significantly reduces the time required to analyse and search for words

across the book collection.

For instance, the module uses `parallelStream()` to divide the workload across available processor cores, ensuring efficient resource utilization. Each word is processed independently and simultaneously, enabling faster retrieval of results and metadata extraction. This improvement not only boosts the module's scalability but also makes it highly suitable for applications requiring quick responses, such as real-time text analysis or large-scale data processing tasks.

2.3.1 Two Module Versions

There are two versions of this module, each with a different approach to storing data:

- **One File per Word Version (the main one)**: This version uses a *datamart* that organizes the information into a directory with two folders:
  - **Metadata folder**: This folder stores a file with relevant book information, such as title, author, and so on.
  - **Words folder**: Each processed word is stored in its own file, allowing individual access to each word and its respective occurrences.
- **One File Version**: This version also uses the *datamart*, but instead of separate folders, it stores two files:
  - A file for **metadata**, which is identical to the one in the first version.
  - A single file called **words**, where all processed words are stored together. Unlike the first version, this one does not create a separate file for each word, grouping all the words into a single file, which can be simpler in some scenarios but requires more processing when searching for individual words.

The main difference between the two versions lies in how words are managed and stored, impacting the structure and access to the data depending on the application's needs.

2.4 Data structure: Single File for Words: `words`

In one approach to indexing, all the words and their occurrences are stored in a single file called `words`. This file acts as a comprehensive index, containing information about all the words found in the dataset, as well as their occurrences across different books.

2.4.1 metadatos (Metadata File)

As in both approaches, we use a `metadata` file to store the metadata for each book. The metadata includes important details such as:

- **Book ID**: A unique identifier for each book.
- **Title**: The title of the book.
- **Author**: The author's name.
- **Year**: Year of the release date of the book.
- **Language**: The language of the book.
- **Download Link**: A URL from where the book can be accessed.

An example of an entry in `metadata`:

```
book_id_123
Pride and Prejudice
Jane Austen
2024
English
http://www.gutenberg.org/ebooks/1342
```

2.4.2 `words` (Words File)

The `words` file stores all words in a single structure. Each word is followed by a series of lines preceded by a dash, indicating that the first element is the ID of the book it belongs to, and all the other numbers on the line are the lines where it appears in that book. It follows a structure like that:

```
project
- 45091 25, 78, 102, ...
- 43456 15, 67, 220, ...
banana
- 45331 55, 77, 192, ...
- 47776 52, 47, 55, ...
```

2.4.3 How It Works

When a user searches for a word, the system reads from the `words` file to find the relevant entries. The file contains the occurrences of each word, the books in which they appear, and the specific lines for each book. This approach is straightforward but can lead to performance bottlenecks as the file grows with more words and books.

2.5 Alternative Approach: One File Per Word

In contrast to the single file approach, another method is to store each word's occurrences in a separate file. This structure involves having a dedicated file for each word in a directory, such as `words/`, and for the `metadata` file, it will be on a directory named `metadata/`.

2.5.1 metadata / (Metadata Directory)

As in the previous method, the `metadata` file remains the same, storing important metadata for each book, such as title, author, year, language, and download link. This file structure is identical across both indexing methods.

2.5.2 One File Per Word: `words/` Directory

Instead of storing all words in a single file, this approach stores each word in its own file within the `words/` directory. Each word file contains the same information as the other version, having the same structure each file.

For example, the file `words/apple` would contain:

```
45331 55, 77, 192, ...
47776 52, 47, 55, ...
```

This structure allows the system to retrieve word occurrences by reading only the relevant word file, reducing the amount of data that needs to be loaded for each query.

2.5.3 How It Works

When a user searches for a word, the system navigates to the `words/` directory and locates the file named after the search word (e.g., `apple`). The file contains all the occurrences of the word across different books, along with the specific line numbers. This approach is more scalable than the single file approach, as the file system can handle a large number of smaller files more efficiently than one large file.

2.6 Comparison of Both Approaches

- **Single File (`words`)**: This method is simpler in terms of file management but may face scalability issues as the number of words and books grows. The file becomes larger, and every query requires reading the entire file, which can slow down the search process.
- **One File Per Word (`words/` directory)**: This method reduces the load for each query by limiting the search to a single word file. It is more scalable as the system grows since the search engine only needs to load the file corresponding to the searched word. However, managing a large number of small files can become complex as the dataset expands.

2.7 API

The API serves as the intermediary layer between the user interface and the query engine, facilitating seamless communication and efficient data retrieval while ensuring secure and reliable access to backend resources. Built with Sparks, the API is designed to receive HTTP requests, process them, and return structured responses, utilising the microframework to simplify routing and response handling. This architecture allows users to send search requests and retrieve relevant results in an organized and scalable manner.

2.7.1 Key Features

- **Endpoint Management:**
  - The primary endpoint `/search` allows users to submit phrases as query parameters. The API processes these phrases and returns the results in JSON format. This endpoint supports `GET` requests and handles errors gracefully:
    - **400 Bad Request:** Returned when the required `phrase` parameter is missing or empty.
    - **500 Internal Server Error:** Captures and reports unexpected server-side issues.
  - A custom **404 Not Found** handler ensures that all invalid routes return a consistent JSON response.
- **Port Configuration:**
  - The API listens on port `8080`, which is specified during initialization. This provides a consistent entry point for clients.
- **Cross-Origin Resource Sharing (CORS):**
  - CORS middleware is implemented to allow requests from external origins. This enables seamless integration with frontend applications hosted on different domains, ensuring compatibility with modern web development standards.
- **Extensibility:**
  - The API is designed to be modular and easily adaptable. New endpoints or features can be added with minimal impact on existing functionality.
- **Scalability:**
  - By leveraging a lightweight framework and efficient data processing techniques, the API can handle a growing number of requests without significant performance degradation.

This server provides a robust foundation for building interactive applications that require efficient query processing and structured data exchange between clients and backend systems. Its focus on modularity, simplicity, and error handling ensures reliability and ease of use for both developers and end-users.

2.8 User interface

The user interface module allows users to interact with the book search engine through a simple and clear interface.

2.8.1 Structure and design

An HTML file defines a web page where users can search for words or phrases within a book collection. The interface includes a text input field and a search button that sends queries to the backend and dynamically displays the results in a dedicated results area.

The interface uses a clean, professional design implemented in the ‘style‘ block. Soft colors and a centred layout make the application visually appealing, while ‘box-shadow‘ and ‘border-radius‘ add a modern touch. The flexible structure allows the interface to adapt to different screen sizes.

The file is designed to detect when the user presses ”Enter” in the search field (‘onkeydown=”handleKeyPress(event)”‘). This detail contributes to an intuitive interaction, as users can initiate searches either via the keyboard or the search button.

2.8.2 How it communicates with query engine

The HTML uses JavaScript to manage interactions with the API. The ‘async‘ and ‘await‘ functions are used to perform asynchronous HTTP requests, which ensures a smooth user experience by preventing page reloads when displaying search results.

When a user enters a word or phrase in the search field and clicks the button or presses the ”Enter” key, the ‘searchWords()‘ JavaScript function sends a ‘GET‘ request to the API endpoint, built with Spring Boot. The request URL includes the search term as a query parameter (‘phrase‘).

The API responds with JSON data, including the search results corresponding to the entered phrase. JavaScript then uses ‘await response.json()‘ to process the response data and display it in the results area.

2.8.3 Error Handling

The ‘searchWords()‘ function includes error handling through a ‘try...catch‘ block, ensuring that any issue with the request or response (e.g., if the API is down or returns an error) is displayed to the user as a friendly message. This improves usability by keeping the user informed if something goes wrong.

3 Benchmarks

In this section, we will present a comparison between the two versions of the query engine based on their performance. By executing a series of benchmarks, we aim to measure the efficiency of each version.

The benchmarks will help us understand the trade-offs between the modular approach, where each word is stored in an individual file, and the version that consolidates all words into a single file. The we will take into count is the searching time, it will provide valuable insights into which version is more suitable for different use cases, particularly when dealing with large datasets.

3.1 Benchmark of Execution Times of Two Versions of the Indexer Module

We evaluated and compared two versions of the Indexer module (referred to as *Indexer Words File* and *Indexer One File Per Word*) in terms of execution time for processing an increasing number of books. The results show a clear performance improvement of Indexer one file per word over Indexer one unique file.

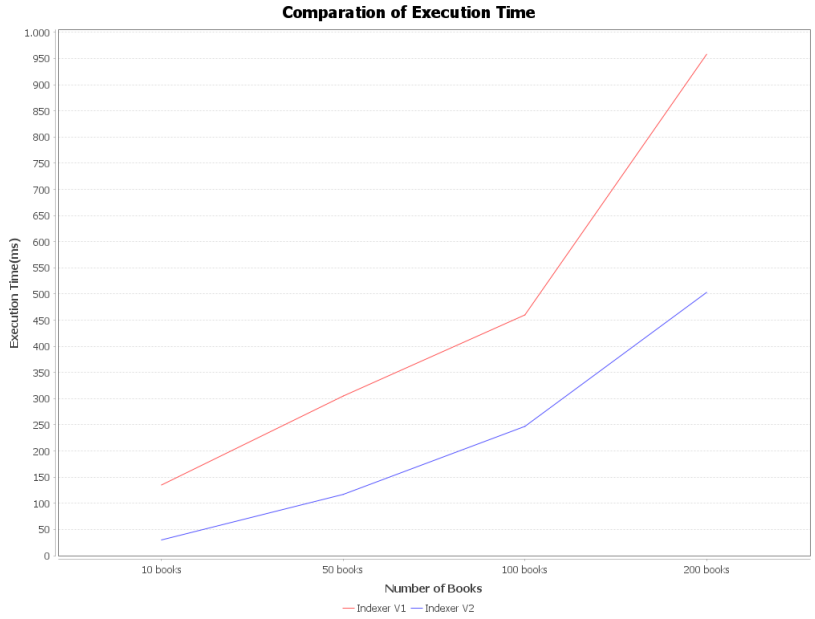

Figure 2: Execution Time Comparison for Indexer file per word vs Indexer one unique file

- **Efficiency of Indexer One File per Word:** The second version of the indexer proved to be significantly faster across all tests. As the number of books increases, the execution time of *Indexer One File per Word* grows more gradually compared to *Indexer One single File*. This suggests that Indexer One File per Word includes optimizations that better handle the scaling of data.
- **Scalability of Implementations:** For 10 books, *Indexer One File per Word* took 135 ms, while Indexer One Single File required only 30 ms. As the number of books increases to 200, Indexer File per Word reaches an execution time of 958 ms, while Indexer One Single File takes 503 ms. This difference in speed suggests that Indexer one unique file is capable of scaling more efficiently, which is essential for applications that need to process large volumes of data.
- **Linear Relationship in Scalability:** The graph shows that Indexer File per Word exhibits a more pronounced, nearly quadratic increase in execution time as the number of processed books grows, whereas Indexer one unique file follows a more linear trend. This implies that Indexer one unique file may have implemented data structure or search algorithm optimizations that avoid unnecessary duplications in processing.

In conclusion, the results support the superiority of Indexer one unique file in terms of performance and scalability compared to Indexer file per word. For data-intensive processing systems, such as indexers or search engines, these improvements can translate into significant efficiency when handling large volumes of information.

3.2 Benchmarking of Indexers in Java and Python

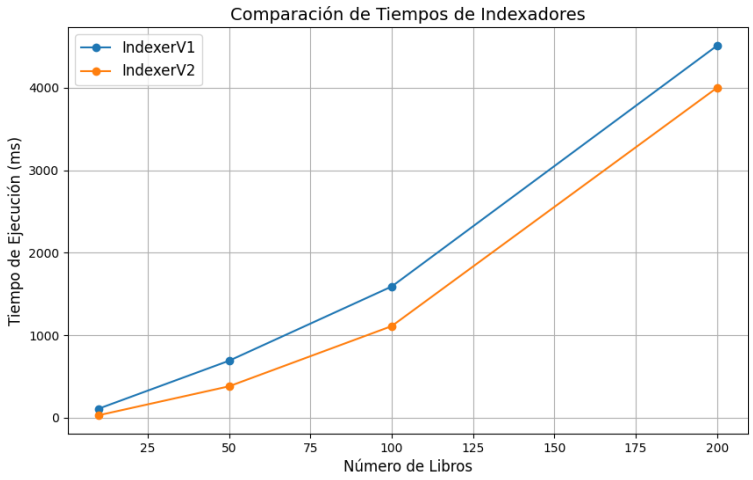

Figure 3: Execution times of Indexers in Python.

3.3 Key Observations

- **Scalability:** Both in Java and Python, **IndexerV2** demonstrated better scalability compared to **IndexerV1**, reducing execution time significantly for larger datasets.
- **Python Results:** Python's **IndexerV1** showed consistently higher execution times than **IndexerV2**.
- **Java Results:** Java implementations outperformed Python for larger datasets, highlighting its efficiency in handling computationally intensive tasks. **IndexerV2** showed significantly better performance than **IndexerV1**, particularly for datasets of 200 books.

Conclusion

While Python offers competitive execution times for small-scale tasks, Java proves to be more efficient for handling larger datasets due to its better runtime optimizations and thread management capabilities. These results emphasize the importance of choosing the right programming language and algorithm for the specific scale of the indexing workload.

3.4 Benchmarking Crawler: Java vs Python Execution Times

The chart compares the execution times of a web crawler implemented in Java versus Python for different numbers of books. The X-axis represents the number of books, while the Y-axis represents execution time in milliseconds. Java is depicted by the blue line, and Python by the red line.

3.5 Analysis

- **Performance Differences**: For all tested quantities of books, Java consistently outperforms Python. This trend is more evident as the number of books increases, suggesting Java's efficiency in handling larger data sets in this specific context.

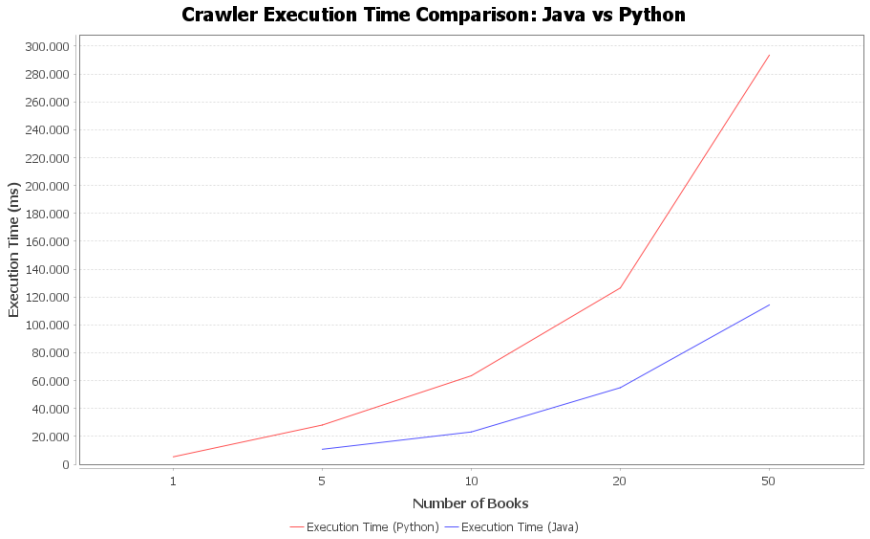

Figure 4: Execution Time Comparison for Java vs Python

- **Scalability**: The execution time for both Java and Python increases with the number of books, but Python's growth rate is noticeably steeper. This indicates that while both languages experience performance degradation with increasing workload, Python's execution time grows more rapidly than Java's.
- **Potential Causes**: Java's lower execution time can be attributed to its compiled nature, allowing for faster runtime performance compared to Python, which is interpreted. Additionally, Java's memory management and optimized threading may play a significant role in its advantage.

3.5.1 Analysis

Based on the benchmark results, we can conclude that the first version of the module, which uses one file per word, is significantly more efficient and scalable compared to the second version, which stores all words in a single file. Although the second version may show a slight improvement in terms of storage space by consolidating the words, this structure introduces several issues. First, the search time in the second version increases considerably because all words are contained in a single file, forcing the system to process and search through a larger amount of data for each query. This is clearly reflected in the higher search times and the larger standard deviation in the results.

Moreover, the first version allows for greater modularity, as each word is stored independently, making it easier to manage parallel or distributed queries. The second version does not adapt well to scenarios where individual words need to be modified, deleted, or added, as any change requires rewriting the entire file. Finally, scalability becomes a challenge in the second version, as the number of words grows, handling a single file becomes slower and less efficient, impacting both performance and resource usage. Therefore, the first version is preferable in terms of efficiency, scalability, and long-term maintenance.

3.6 Benchmarking Query Engine: Java vs Python Execution Times

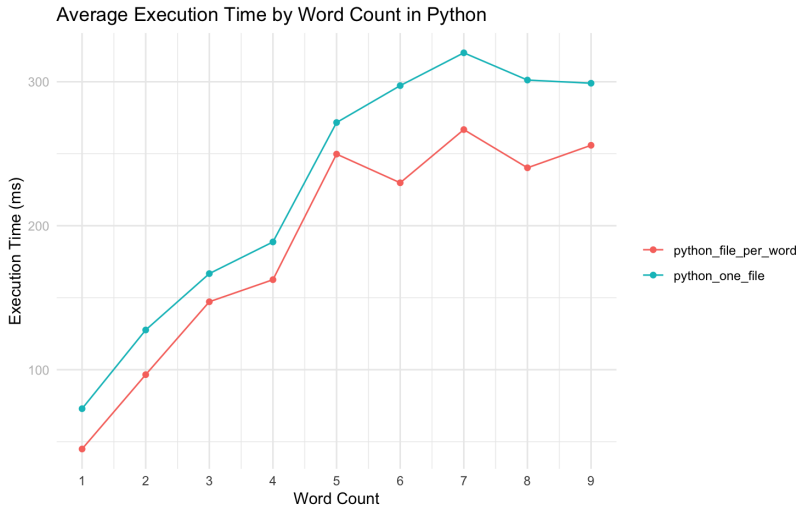

Figure 5: Execution time of different queries tested in the two different data structures in Python

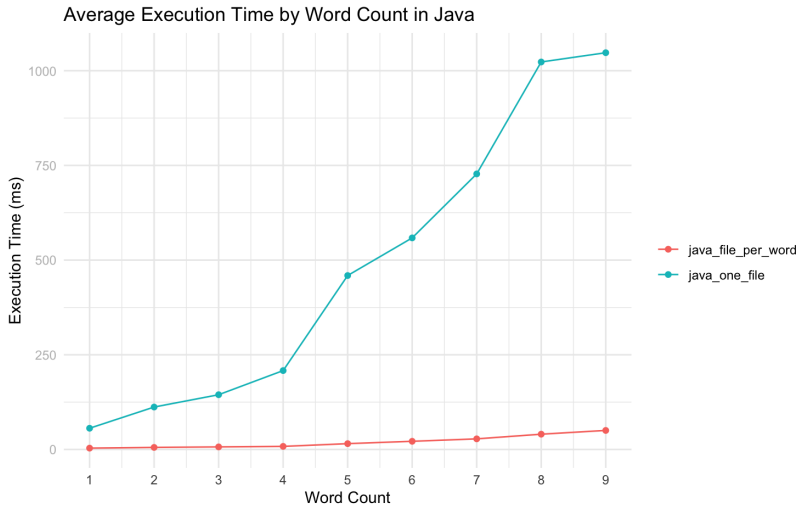

Figure 6: Execution time of different queries tested in the two different data structures in Java

3.6.1 Analysis

For this analysis, we tested the same books and the same phrases for each word count in both Python and Java to ensure consistency and comparability. It is important to note that execution time can vary depending on the number of books in which a phrase appears and, more significantly, on the specific lines where the phrase occurs. If a phrase appears lower in the content of a book, the execution time might be longer due to the need to process more lines before finding the target phrase. These factors should be taken into account when interpreting the results.

The graph compares the execution times of two Python methods, *One Unique File structure* and *File Per Word*, as word counts increase from 1 to 9. Initially, *File Per Word* is faster, but both methods show rising execution times with increasing word counts. Beyond three words, *One Unique File structure* becomes slower, peaking significantly at six words, while *File Per Word* maintains a more stable and consistent performance. This suggests that *File Per Word* is better suited for handling larger word counts due to its

linear scalability, whereas *One Unique File structure* may be more efficient for smaller inputs but struggles with higher complexity as the word count grows.

On the other hand, the second graph compares the execution times of two Java methods, *One Unique File structure* and *File Per Word*, based on varying word counts (1 to 9). The execution time for *File Per Word* remains relatively flat across all word counts, demonstrating high scalability and minimal impact from increasing input size. In contrast, *One Unique File structure* shows a steep increase in execution time as the word count rises, indicating poor scalability. While *One Unique File structure* performs moderately well for smaller word counts, its execution time grows exponentially beyond 4 words, exceeding 1000 ms at 9 words. This highlights *File Per Word* as the more efficient and scalable approach for larger datasets, whereas *One Unique File structure* is suitable only for small-scale inputs.

4 Program Execution

1. **Indexer**: The first step is to execute the corresponding version of the indexer module, depending on the chosen data structure. The processed books are read from the datalake and stored in the datamart for future queries. In this step, it is important to specify the paths for both the datalake and the datamart.

2. **Crawler**: Next, the crawler is executed to download the specified number of books from external sources and store them as text files in the datalake. As with the indexer, it is crucial to define the datalake path.

3. **Query Engine**: Finally, the query engine allows searches to be performed once all books have been processed by the indexer. The query engine accesses the indexed data in the datamart and provides the relevant results. In this step, both the datalake and datamart paths must be specified.

Steps to Run the Project with Docker

The project requires loading pre-generated Docker images and running the appropriate `docker-compose.yml` file, depending on the desired data structure and project setup. Follow these steps to start the system:

1. **Load the Docker images:** Ensure you have the `.tar` files for all the required Docker images. Load each image using the following command:

   ```
   docker load -i <image_name.tar>
   ```

   Repeat this step for all provided images.

2. **Run the appropriate `docker-compose.yml` file:** Depending on the structure and modules you want to execute, select the corresponding `docker-compose.yml` file and run it. Use the following command:

   ```
   docker-compose -f <docker-compose-file.yml> up --build
   ```

3. **Access the user interface:** If everything is correctly configured and the Nginx service is running inside the container, you can access the user interface in your browser at:

   ```
   http://localhost:3000
   ```

   Nginx will serve the user interface and redirect any necessary requests to the backend services.

4. **Switching between configurations:** To test different data structures or setups, stop the current containers using:

```
docker-compose -f <docker-compose-file.yml> down
```

Then, execute the corresponding `docker-compose.yml` file for the desired configuration.

5 Conclusions

In this project, we successfully implemented a search engine that includes three core modules: a crawler, an indexer, and a query engine. Through rigorous testing and benchmarking, we evaluated the performance of different design choices and data structures, enabling us to draw the following conclusions:

1. **Efficiency of Modular Design**
   The adoption of modular architectures across all modules significantly improved code maintainability, scalability, and flexibility. Particularly in the crawler and indexer modules, the interface-based design enabled seamless integration of new components, such as additional language filters or indexing techniques.

2. **Inverted Index Approaches**
   Comparing the two indexing methods—single file for all words versus one file per word—revealed the advantages of the modular "one file per word" approach. This method demonstrated superior scalability, efficiency, and ease of query execution, especially when handling large datasets. It also provided better support for parallel processing and dynamic updates.

3. **Language Comparisons**
   Benchmarking the crawler in Java and Python highlighted the performance superiority of Java, particularly as the number of books increased. Java's compiled nature, efficient threading, and memory management allowed it to scale more effectively than Python, making it the preferred choice for high-performance applications.

4. **Scalability and Optimization**
   The benchmarks underscore the importance of choosing scalable data structures and leveraging optimizations such as parallel processing. These techniques are essential for handling the exponential growth of data in modern applications, ensuring both speed and efficiency.

5. **User Experience and Accessibility**
   The query engine and user interface provided a seamless search experience, enabling fast and intuitive access to large datasets. The use of parallelization in the query engine further enhanced its responsiveness, making it suitable for real-time or large-scale applications.

6 Future Work

1. **Enhanced Parallelization**
   Implementing advanced parallelization techniques or distributed computing frameworks like Apache Spark could further improve performance, especially for large datasets.

2. **Additional Languages**
   Expanding the language filter to support more languages would increase the system's versatility and applicability.

3. **Dynamic Index Updates**
   Incorporating dynamic indexing capabilities would allow real-time updates to the index, enabling the system to adapt to changes without requiring full re-indexing.

4. **Vectorized Search Algorithms**
   Exploring vectorized search techniques or incorporating machine learning models could enhance search accuracy and relevance.

5. **Cloud Integration**
   Migrating the system to a cloud-based infrastructure could enable better scalability, higher availability, and integration with other big data tools and services.

By addressing these areas, the system can evolve into a more robust, scalable, and efficient platform for large-scale information retrieval, aligning with the needs of modern data-intensive applications.

A Link to GitHub repository

Search Engine Repository