Search Engine, Stage 2

Elisa Breeze
Nicolás Trujillo
Luna Hernández
Carlos Suárez
Ivan Anikin

November 2024

GitHub link

Abstract

This paper shows the development of a search engine in Java using an inverted index for efficient word querying. It has three modules: a crawler for downloading books, an indexer with expanded and aggregated structures, and a query engine. All this is is made reproducible in portable containers by using Docker.

The Benchmarking results revealed that the expanded index offers faster indexing, while the aggregated index is slightly faster in single-word queries. We chose the expanded structure for its overall balance of speed and scalability, especially taking into account future project expansions.

1 Introduction

The idea of this work is to create a search engine with an inverted index, that stores words and their positions in documents, allowing efficient searches. The index we created, does not only store which book contains a specific words, but also the location where it appears, allowing a more precise outcome.

By using an inverted index, we can implement fast and effective word searching when a term in queried. In this project we implemented two types of indexers, tested them and chose the best one to continue working with in the next Stages of the Project. The two implementations differ on how the positions of the words are saved in relation to the Book ID of the book they appear in. We tested them using Benchmark tools, to measure the execution times of the search engines of each version, to see which implementation is the most efficient.

Apart from the Indexer, we also created a Crawler, to download the books periodically from Gutenberg.org and save them in a Datalake, and it also saves the metadata of the books in a csv.

And lastly, so that we could test our search engine, we created query enginea common query engine for both data structures, that takes the word or words to be searched for, looks for it and gives back information about the book the word is in, the books name, author, URL, the occurrences and the paragraph it's in. With that, we also included the Docker, which is used to make sure we have consistent and reproducible environments for running the program basically in a portable container.

2 Methodology

2.1 Crawler

The Crawler is responsible for downloading the books from the Gutenberg website and saving them in a the directory. The process of retrieving books using their identifiers, saving their content as text files, and storing the metadata about each book is automatic. Here are the important parts of the crawler:

2.1.1 Classes and Key Methods

- **CrawlerCommand**: It controls the crawling process by coordinating the downloading of the books by checking the last downloaded book ID and downloading the next three books. It also schedules periodic downloads by invoking the process every interval.
- **Crawler**: Performs the actual task of downloading the books from the Gutenberg website. It downloads the book by the ID, and saves the book in the directory as a text file, retrieving in the process its title and author

by scraping that data from the Gutenberg book page using data from the h1 HTML element. After that, it saves the metadata for each book in a CSV file called metadata.csv.

2.2 Indexer

The Indexer is responsible for processing the books and creating an index structure that allows fast retrieval of words. We implemented 2 different indexing strategies, which we will explain below: Expanded Indexer and Aggregated Indexer.

2.2.1 Important parts of the code:

- **Book**: Represents each book to be indexed and with its methods getBookId() and getContent() it is possible to retrieve the ID of the Book and its content, and it is possible to modificate the content aswell.
- **HierarchicalCsvStore**: The directory structure is based on the first few characters of the word, and the word is used as the file name. There are two indexer strategies that use this directory structure: expanded and aggregated indexers.
- **GutenbergBookReader**: Reads books from the directory, creating a Book object for each one. The read() method then retrieves the text files, parses the file name to get the Book ID, and reads the content for each book to initialize the instances of the Book object.

2.2.2 Indexer Strategies

Here are the two Strategies explained:

1. **Expanded Indexer**:
    - Each occurrence of the word in a book is indexed in an individual CSV file.
    - Each CSV file contains occurrences with the BookID and the position of the word in the book.
2. **Aggregated Indexer**:
    - All positions for a given word across different books are aggregated into a single CSV file.
    - Each word has its own CSV file, where the book ID and the list of positions are saved as a single entry for each word.
    - The positions for each book are stored as a list, each position separated by a semicolon.

Both of the indexing strategies allow flexibility when handling word occurrences. The expanded strategy provides a more detailed, individual entry for each occurrence, while the aggregated strategy groups word positions into a single file per word, making it more efficient for bigger datasets.

2.3 Query Engines

The Query Engine is one of the most important roles in the search engine architecture, as it serves as the component that interprets inputs, locates the indexed terms, and makes it possible to have a readable output for the user, giving the information about the word looked for.

2.3.1 Query Engine for Both Indexing Types

Here are the most important parts of The Query Engine:

- **Main:** Initializes the Search Engine by setting up instances of the input, output, and search engine command classes, and managed the user input and query process.
- **SearchEngineCommand:** This class links the input and output of the search engine, making communication easier between the query and indexing components. First it initializes the paths for the book folder, index folder and metadata file. Then it takes the user input through the Search Class, queries the terms (using CommonQueryEngine Class), and gives back the search results for the term looked for using the SearchOutput Class.
- **CommonQueryEngine:** This class is basically responsible for the query process. First it loads metadata of the Books form the metadata CSV, saving it in a map where each entry has the book ID, title, author and URL. It also loads the word's index from its CSV file and reads the file for occurrences and positions of the words inside the book. Then, it loads the word indices, checks if there are common books between various search terms, and find the paragraphs where the terms appear, highlighting the word looked for before saving it. Lastly, it saves it and gives back the information of the book metadata combined with the paragraphs. Then there is also the Paragraph Extractor class, which reads through each book file and finds the paragraphs where the queried word is found, highlighting the word, counting the occurrences and saving the paragraphs.

2.4 Docker Integration

To make sure we have consistent and reproducible environments for running the search engine, we used Docker to containerize the application. This allows for the packaging of the application, together with all its dependencies, into a

portable container.

The Docker setup for this project includes a Dockerfile for each module, and the use of Docker Desktop for managing the containerized environment. We containerized each module individually, to ensure modularity and maintainability, and for each module we created a specific Dockerfile, which uses a multi-stage build process to optimize the container size and to make sure that only the necessary artifacts are included. It makes use of maven and defines the working directory, copying the pom and source files to the container. Then is executes the maven command to build the module while is skips the tests to speed up the process of building. Once that is done, it starts with a clean image to keep the runtime minimal, and copies the compiles .jar file from the build stage into the runtime image.

By following this structure, each module can be built and deployed independently and the dependencies are isolates, which reduces conflicts across the modules.

2.4.1 How to Execute Docker

Step 1: Build the Project
Run the following Maven command to clean and build the project:

```
mvn clean install
```

Step 2: Create the Volumes
Use the following commands to create the required Docker volumes:

```
docker volume create datalake
docker volume create metadata
docker volume create datamart
```

Step 3: Build and Run the Containers
To create containers, first build the images, then run the containers. You can stop and start them using the following commands:

```
docker stop <container_name>
docker start <container_name>
```

Below are the build and run commands for the different modules:
Crawler:

```
docker build -f crawler/Dockerfile -t crawler-image .

docker run -d --name crawler-container -v datalake:/app/data
    ↪ /datalake -v metadata:/app/data/metadata crawler-image
```

Expanded Indexer:

```
docker build -f indexer/Dockerfile -t expanded-indexer-image
    ↪  .

docker run -d --name expanded-indexer-container -v datalake
    ↪ :/app/data/datalake -v datamart:/app/data/datamart
    ↪ expanded-indexer-image
```

Aggregated Indexer:

```
docker build -f indexer/Dockerfile --build-arg PROFILE=
    ↪ aggregatedIndexer -t aggregated-indexer-image .

docker run -d --name aggregated-indexer-container -v
    ↪ datalake:/app/data/datalake -v datamart:/app/data/
    ↪ datamart aggregated-indexer-image
```

Query Engine:

```
docker build -f query-engine/Dockerfile -t query-engine-
    ↪ image .

docker run -it --name query-engine-container -v datalake:/
    ↪ app/data/datalake -v metadata:/app/data/metadata -v
    ↪ datamart:/app/data/datamart query-engine-image
```

3 Experiments and Results

To analyze the performance of the different modules, we executed benchmarks to analyze the execution times, so that we could see which option has a better performance.

To do so, we included a benchmark function in a separate file, to do the testing and measure the execution time of each one to find out for example which one is the best Data Structure to work with.

Here you can see the results we obtained:

3.1 Crawler Benchmark results

Index Type	Execution Time (ms)
Crawler	6068.557

Table 1: Benchmark test results

3.2 Indexers Benchmark results

Index Type	Execution Time (ms)
Aggregated	193934.472
Expanded	28327.979

Table 2: Benchmark test results

3.3 Query Engine Benchmark results

Index Type	Execution Time (ms) "man"	Execution Time (ms) "immediate imminent"
Aggregated	70.004	0.007
Expanded	80.360	0.007

Table 3: Benchmark test results

4 Result Discussion

The crawlers execution time is 6068.557 ms, which is a baseline for the performance of our system, which we are going to be able to analyze further with the other results, which are more interesting taking into account we have 2 data

structures to compare and decide which is the better option.

When analyzing the indexers execution time we can see a big difference between the two different types: the aggregated Indexer takes 193934.472 3.23 minutes), while the expanded indexer takes 28327.979 ms (about 28.33 seconds), which makes us see that it is a lot more efficient.

Lastly the Query Engine benchmark results where very similar: For searching "man" it took 70.004 ms for the aggregated structure, and 80.360 ms for the expanded structure. On the other hand, when searching more than 1 word (in this case we tested searching for "immediate imminent", the execution time for both structures was exactly the same: 0.007 ms.

These results shows us that in the query process the Aggregated Indexing structure has a slightly better performance for single-word queries compared to the expanded structure, but the multi-word searches have the same results. In our opinion the best structure to stick with is the expanded one, especially taking into account the indexing time, and although the searching time for single words is a little bit longer, on the long run, and if intended to expand, in general it's better to go with the expanded structure.

5 Conclusion

In conclusion, these results show clear differences between the indexing complexity of both structure types and little differences in the query performance. Depending on the priorities you could choose on or the other, but we think that it's better to stick with the expanded structure, as the indexing is much faster, and the word queries are very similar to the Aggregated Index searches.