Inverted Index
BIG DATA

Grado en Ciencia e Ingeniería de Datos a

Universidad de Las Palmas de Gran Canaria

Inverted index: Crawler Indexer API

Wojciech Dróżdź	wojciech.dro101@alu.ulpgc.es
Zuzanna Furtak	zuzanna.furtak101@alu.ulpgc.es
Olga Kalisiak	olga.kalisiak101@alu.ulpgc.es
Jakub Król	jakub.krol101@alu.ulpgc.es

Profesors: Hernández Cabrera, José Juan Évora Gómez

Github Repository Link

Contents

Abstract

This project focuses on the design and evaluation of a search engine through various configurations to identify the most efficient approach. The initial step involved creating a web scraper for downloading content from Gutenberg.org, optimized with multithreading to enhance performance. The data was then indexed, storing words alongside metadata like title, author, and location. Two storage formats, SQLite and JSON, were tested, revealing that SQLite provided faster query times despite JSON's quicker storage time. To address performance bottlenecks, the project was re-implemented in Java, incorporating MongoDB and a custom System File Storage (SFS) structure. MongoDB offered organized storage for improved search and distribution capabilities, while SFS utilized a hierarchical folder system for rapid access to data. Docker was integrated to streamline deployment and ensure consistency across environments. Benchmarking using the Java JMH library demonstrated a threefold improvement in scraper speed with Java, although SFS emerged as the optimal storage solution for the query engine due to its superior runtime performance. The results highlight SFS as the preferred approach for future scalability and stability in multi-device environments.

1 Introduction

The goal of this project is to create a search engine and test it in various scenarios. The first task was to create a scraper that would download content from Gutenberg.org. To speed up this process, the scraper was enhanced with the multithreading library, that improved the efficiency multiple times. The next and most important part was the indexer, that used the structures created by scraper (metadata) to make a storage system in which words were stored in simplified order. Each word had his own index, informing the search engine user about the data where the word occurred, e.g. the title of the book, the author, and the location of the word in the book. To test multiple possibilities and select the best option through comparison, we created two data structures allowing to store data. The first was a database in SQLite, with a processing time of 12.5475 seconds, and the second was a JSON structure, with a processing time of 10.0609 seconds. Despite the better time for JSON, the database structure had better search time due to its more organized structure, placing the database structure on better end position.

2 Problem statement

The first issue that needs to be addressed is the excessive runtime of the program. To resolve this, the next step will be to rewrite the code from Python to Java, as it is expected to perform better with large amounts of data that we are testing. Another approach to improving the program will involve revising the indexer, which happened to be inefficient for further development. New methods will include the advanced use of MongoDB and the System File Storage approach suggested during the defense of the previous version. The final part of this task will involve creating and implementing Docker, which will simplify the usage process and give each part of the program his own space (container).

3 Methodology

3.1 Firs step

Firstly, the code will be rewritten from Python to Java to enhance the processes itself. During the rewriting process, there will be added changes to improve the speed of searching for the desired words. The API will be rewritten without significant changes, just to enable word searching.

3.2 Changes in scraper

The main change in the scraper, apart from the language, will be a different way of saving the data. Previously, data was stored in a basic SQLite database. Now, the scraper will use MongoDB to store metadata and the entire content, which should provide better access for the indexer in the next part of the project.

3.3 Changes in indexer

Due to the slow performance of SQLite and JSON structures, they will no longer be used. The new testing structures will be MongoDB and System File Storage.

3.3.1 MongoDB

In this approach, we will store data in JSON format, but in a well-structured way that will enable us to:

- quick searching in big and complexed amount of data
- sorting the data in wanted order enhancing the process of searching data
- distribute data between different severs
- process data and create tf-idf (Term Frequency - Inverse Document Frequency)

3.3.2 SFS System Files Storage

This structure assumes a low-level method of storing data. Each letter of the alphabet will have its own folder named after that letter. Inside each of these folders, there will be additional subfolders named after all possible two-letter combinations that start with that specific letter. These subfolders will contain JSON files, where each file represents a word. The words stored in a given subfolder will start with the two letters corresponding to that folder's name. This approach should accelerate the process of saving and searching all essential data gathered from the downloaded books.

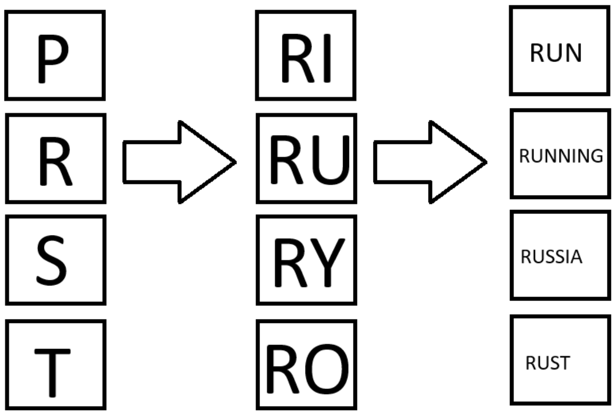

Figure 1: Visualization of System Files Storage approach

3.4 Docker

Docker is going to be a platform that will simplify the running process by using containers. Containers are lightweight, portable units that package an application along with its dependencies and environment, ensuring consistency in process of developing and testing on multiple devices.

3.4.1 Key Features of Docker:

- **Isolation**: Each container runs independently with its own container, avoiding conflicts between each part of the end product.
- **Portability**: Containers can run on any system with Docker installed, regardless of the underlying OS.
- **Efficiency**: Containers use fewer resources compared to virtual machines, as they share the host OS kernel.

4 Experiments

In the experiments section we are going to give each part of the previously mentioned search engine code into benchmark testing. The tests will be conducted using a special Java library, JMH, which is dedicated to this type of experiment. The main parameter considered will be the running time of the code.

4.1 Scraper speed

Scraper acquired speed of 1000 books downloaded in 21.980 seconds. In comparison with previous version score, which was 1000 books per minute, rewritten code made three times better score making noticeable difference.

```
Benchmark                               Mode  Cnt   Score   Error  Units
CrawlerBenchmark.benchmarkDownloadBatch avgt       21,980           s/op
```

4.2 Indexer

4.2.1 File indexer

Indexing processes in the newly tested approach, as demonstrated during benchmarking, showed a result of 80.799 seconds.

```
Benchmark                               Mode  Cnt   Score   Error  Units
FileIndexerBenchmark.benchmarkIndexing  avgt       80,799          s/op
```

4.2.2 MongoDB Indexer

In this case results of benchmarking processes gave a time of 38.525 seconds.

```
Benchmark                                  Mode  Cnt   Score   Error  Units
MongoIndexerBenchmark.benchmarkIndexBooks  avgt       38,525          s/op
```

4.3 Query Engine

To make clean comparison we decided to make identical test of query engine as before. We compared the speed wile 100 books were indexed and picked identical 100 common English words to test runtime of the code.

4.3.1 Mongo Query Engine

Achieved time for MongoDB Query is 3.459 seconds.

```
Benchmark                                               Mode  Cnt  Score   Error  Units
MongoQueryEngineBenchmark.benchmarkSearchCommonWords  avgt       3,459          s/op
```

4.3.2 File Query Engine

Achieved time for File Indexer Query is 0.059 seconds.

```
Benchmark                                              Mode  Cnt  Score   Error  Units
FileQueryEngineBenchmark.benchmarkSearchCommonWords  avgt       0,059          s/op
```

4.4 Comparison

End results of the benchmarking processes showing times in seconds for Python and Java in every tested approach.

	Time [s]								
	Scraper	Indexer				Query engine: time for 100 books indexed			
Language/Module	Time per 1000 books	JSON	SQLite	MongoDB	SFS	JSON	SQLite	MongoDB	SFS
Python	60	10.06	12.55	X	X	3.43	2.48	X	X
Java	21,98	X	X	38.52	80.79	X	X	3,45	0.059

Table 1: Benchmark comparison

5 Conclusions

Final result of this experiment has shown that despite of predictions Python turned out to be faster in some aspects. Scraper score have shown that Java is almost three times faster than code run in Python which is significant improvement. Two new approaches for the used structure have shown way longer required time to make inverted indexed structure but in Query engine tests advised System File Storage turned out to be the fastest tested structure. To decide which approach is better we have to think that in the future our Search Engine is going to be tested on multiple devices which are mainly using the Query Engine. In terms of that, using System File Storage should be better for stability of future server.

6 Future work

In the next part of our project, we are going to create a complete and easy-to-use API. This will be crucial for testing how the search engine we have developed will perform on multiple devices. There will be a local host with two servers that will allow users to search from many devices simultaneously, enabling us to test our project under challenging conditions.