Inverted Index

BIG DATA

Grado en Ciencia e Ingeniería de Datos
Universidad de Las Palmas de Gran Canaria

Wojciech Dróżdż
`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`

January 15, 2025

Profesors

José Juan Hernández Cabrera, José Évora Gómez

Contents

Abstract

This report presents the development and implementation of an inverted index system designed for Big Data applications. The system comprises a crawler for data acquisition, an indexer for efficient storage and retrieval, and a query engine for user interaction. Leveraging *Docker* containers, *RabbitMQ* as a message broker, and *Nginx* as a load balancer, the architecture is scalable and distributed. Key experiments compare different data structures for the index and demonstrate the system's ability to handle concurrent queries. The results showcase the practicality of this approach in creating a robust and high-performing search engine.

1 Introduction

The exponential growth of digital data has posed significant challenges for information retrieval systems. Search engines, one of the most impactful innovations in the digital era, rely heavily on efficient indexing method to provide fast and accurate results. An inverted index is a fundamental data structure used to map terms to their occurrences in a dataset, enabling quick lookups for keyword searches.

The aim of this project is to develop an inverted index-based search engine capable of handling large datasets while maintaining scalability and efficiency. By leveraging modern technologies such as *Docker* for containerization, *RabbitMQ* for inter-process communication, and *Nginx* for load balancing, we designed a distributed system that can scale horizontally to meet the demands of Big Data.

This report details the methodology, implementation, and experimental validation of the proposed system. Emphasis is placed on comparing data structures, optimizing performance, and ensuring the system's reliability under heavy loads.

2 Problem Statement

Traditional search engines often struggle to maintain performance as the size of their datasets grows. This is particularly problematic in Big Data environments, where the volume, variety, and velocity of data far exceed the capabilities of conventional systems. The challenges include:

- Efficiently indexing large datasets while minimizing memory usage.
- Enabling fast and accurate search queries even under heavy loads.
- Designing a scalable architecture that can distribute workloads across multiple machines.
- Ensuring reliability and fault tolerance in a distributed environment.

This project addresses these challenges by implementing an inverted index system that combines modularity, scalability, and performance optimization. The system is designed to support:

- High-performance communication with *RabbitMQ*.
- Load balancing with *Nginx* to handle concurrent queries.

- Optimized data structures for indexing and retrieval.
- Easy deployment and development with *Docker* containers.

3 Methodology

3.1 System Architecture

The main system is divided between three main modules. The following sections describe the core components of the system:

3.1.1 Crawler

The crawler is responsible for fetching data from external sources, such as the Gutenberg Project. It extracts metadata from books, including critical information like the author, title, and other descriptive attributes. This metadata forms the foundation for building an efficient and comprehensive index. The document data is forwarded to one of the indexers from the cluster, which processes it into an inverted index.

3.1.2 Indexer

Indexer processes the metadata generated by the crawler and constructs an index that maps words to their occurrences in the dataset. For each word, the index stores:

- The frequency of occurrences.
- The title of the book in which the word appears.
- The author of the book.
- Additional metadata, such as publication year and genre, if available.

After indexing a batch of documents an index update notification is sent to all of the query engines in the cluster, which append new data into their index.

3.1.3 Query Engine

The query engine stores and updates the inverted index and provides a RESTful API, which can be used to retrieve the data by clients.

The inverted index is stored in a MongoDB database. A *RabbitMQ* queue is used to receive information about the updates to the inverted index.

Query engine supports advanced search functionality for the index, including keyword searches and filtering based on metadata attributes. The engine exposes a RESTful API built with Apache Spark, enabling programmatic access to the search functionality.

3.2 Module communication and Load Balancing

3.2.1 Message Broker

RabbitMQ was used to establish a connection between the crawlers, indexers and query engines. It is used to notify indexers about new crawled documents, query engines about new inverted index updates, as well as for transferring the data. This approach enhances the scalability of the system, as multiple instances of each module can communicate without direct dependencies.

3.2.2 Load Balancer

Nginx acts as the entry point to the system, functioning as a reverse proxy and load balancer. It maintains a dynamic registry of all connected devices by tracking their IP addresses. This enables efficient distribution of incoming client requests across multiple instances of the query engine using the *round-robin* method, ensuring high availability and optimized performance under heavy loads.

3.3 Client Container

The client container provides a simple React app, which can be used for testing and demonstration purposes. A screenshot of the web interface is presented in Section 4 to illustrate its usability and design.

3.4 Dockerized Architecture

Each component is encapsulated in a *Docker* container, ensuring consistent and reproducible execution across different environments. This containerized approach allows for a faster deployment and development.

4 Experiments

To evaluate the performance and scalability of the inverted index system, we conducted a series of experiments focusing on the following aspects:

4.1 System Setup

The experimental environment consisted of multiple *Docker* containers deployed across a distributed cluster of machines. The cluster included:

- Several instances of the crawler, indexer, and query engine.
- A central *RabbitMQ* message broker for communication.
- *Nginx* as a load balancer to distribute client requests evenly.

The containers were orchestrated using `docker compose`, simplifying deployment and management. The system was configured to index 2,000 documents from the Gutenberg Project.

4.2 Scalability Testing

To assess the system's scalability, we replicated the query engine and deployed multiple instances across the cluster. Using *Nginx* as a load balancer, we simulated a large number of concurrent client requests. The system's ability to handle increasing client loads was measured by tracking:

- Response times for search queries.
- The number of concurrent clients supported without degradation in performance.

4.3 Deployment Validation

To ensure reliability, we deployed the system using *Docker* containers on different machines. The deployment process involved:

1. Initializing the *RabbitMQ* broker and *Nginx* load balancer.
2. Starting multiple crawler and indexer containers to ingest data.
3. Launching query engine instances and verifying their connectivity through *RabbitMQ*.

A live demonstration was conducted, showcasing the query engine's ability to handle requests in a distributed environment. The load testing results and deployment success confirm the system's robustness and scalability.

4.4 Endpoint Demonstration and Practical Filtering

The system supports various endpoints to facilitate user interaction. Screenshots were captured to illustrate these functionalities:

- Clean Terminal: shows the initial state of the API with no input. It presents a clean interface, ready for user interactions.
- Filtering by Word, Author, and Word Positions: Figure 2 demonstrates the filtering capabilities of the system. A query was made with the word "run," authored by Lewis Carroll, and constrained to word positions between 6,000 and 19,000. The results showcase the precise filtering achieved through these parameters.
- Word Count Functionality: presents the output of the word counter endpoint. For a dataset of 16 books, the indexer processed a total of 388,224 words, demonstrating the system's efficiency in handling large-scale data.

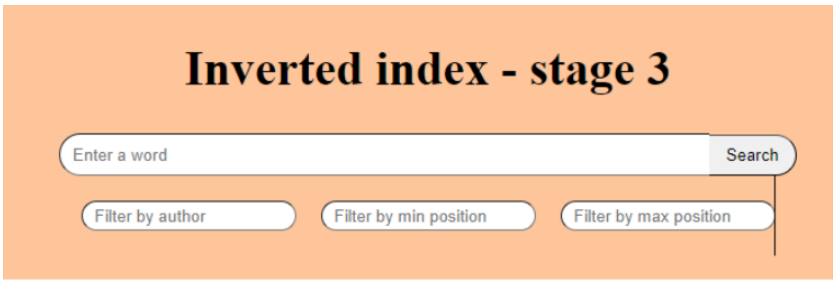
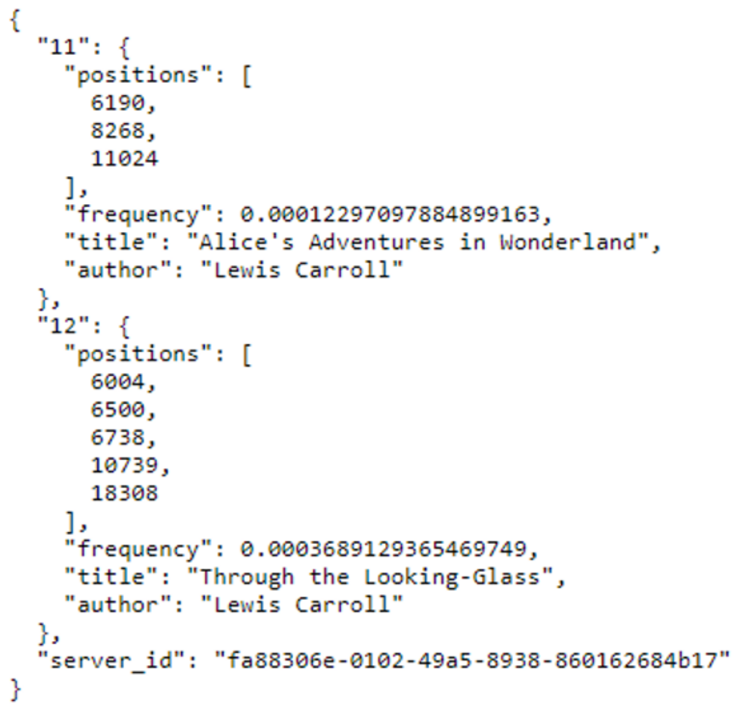


Figure 1: Clean Terminal: The API interface at startup, displaying a ready state without any inputs.



```
{
  "11": {
    "positions": [
      6190,
      8268,
      11024
    ],
    "frequency": 0.00012297097884899163,
    "title": "Alice's Adventures in Wonderland",
    "author": "Lewis Carroll"
  },
  "12": {
    "positions": [
      6004,
      6500,
      6738,
      10739,
      18308
    ],
    "frequency": 0.0003689129365469749,
    "title": "Through the Looking-Glass",
    "author": "Lewis Carroll"
  },
  "server_id": "fa88306e-0102-49a5-8938-860162684b17"
}
```

Figure 2: Filtering by Word, Author, and Word Positions: Query with the word "run," author Lewis Carroll, and positions between 6,000 and 19,000. The results illustrate accurate and refined filtering.

4.5 User Interaction Testing

A dedicated client container was used to evaluate the user experience. Users can submit queries and view the results in real time. The interface's responsiveness and ease of use were assessed.

The experiments demonstrated that the system performs efficiently under realistic workloads, with the ability to scale horizontally by adding more containers as needed.

5 Conclusions

The development and evaluation of the inverted index system demonstrate its efficacy in handling large-scale data and high query loads. By leveraging *Docker* containers, *RabbitMQ*, and *Nginx*, the system achieves modularity, scalability, and reliability. The experiments validate the performance improvements gained through optimized data structures and distributed deployment.

Key findings include:

- The system can handle a significant number of concurrent clients when deployed with multiple query engine instances and a load balancer.
- Docker containers simplify deployment across machines, enabling rapid scaling.

Future work could focus on further optimizing the indexer for real-time updates, exploring alternative data structures, and extending the system to support additional query types and analytics.