# Search Engine

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

January 2025

*Deployment in a cluster*

# Contents

## Abstract

In this project, we designed and implemented a distributed search engine built around three core modules: a crawler, an indexer, and a query engine. The system was developed in Java and uses Hazelcast to enable cluster-based scalability. Docker was employed to containerise the system, ensuring portability and simplifying deployment across distributed environments.

The crawler retrieves and filters books from Project Gutenberg, the indexer constructs efficient distributed data structures for keyword storage and retrieval, and the query engine processes user queries by distributing tasks across the cluster. To ensure high availability and balanced load distribution, we tried to employ NGINX as a load balancer.

This work highlights the potential of distributed systems and containerisation to build scalable and efficient search engines.

## 1 Introduction

Search engines are essential tools for efficiently retrieving information from large datasets, making them indispensable for applications such as research, text analysis, and digital libraries. This project focuses on designing a scalable and efficient distributed search engine capable of handling large volumes of text data while optimising performance and resource utilisation.

The system has been developed in Java and comprises three primary modules: the crawler, which retrieves books from Project Gutenberg and filters them based on specific criteria; the indexer, which builds distributed data structures to facilitate efficient text retrieval; and the query engine, which processes user queries to deliver relevant results. The system leverages Hazelcast as a distributed computing framework, enabling tasks to be executed across a cluster of nodes, and runs in Docker containers to ensure portability and simplify deployment.

To enhance scalability and ensure high availability, we attempted to integrate NGINX as a load balancer, aiming to manage request distribution and balance loads across the cluster. These design choices highlight the potential of containerised distributed systems to address challenges associated with large-scale data processing, such as bottlenecks and resource limitations.

This document provides an in-depth exploration of the system's distributed architecture, the technologies and methodologies employed, and the results of benchmarking experiments. It concludes with insights into the system's performance and suggestions for further advancements, including integration with additional distributed frameworks and enhanced query capabilities.

## 2 Modules and data structures

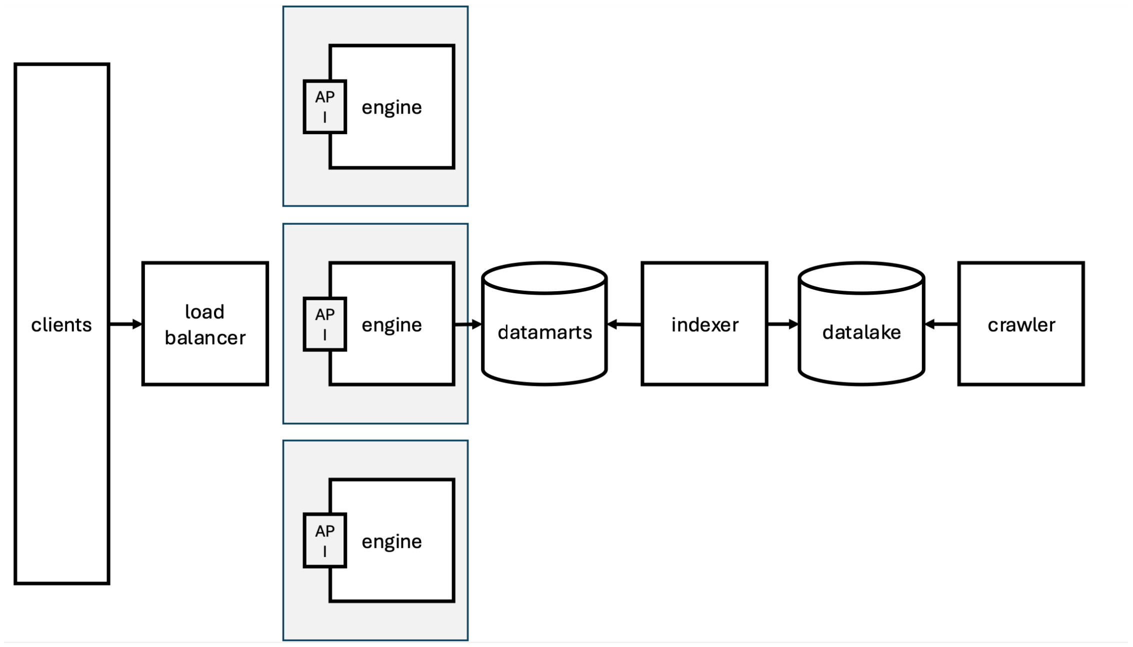


Figure 1: Module and Repository Structure of the Application

In this architecture, both the crawler and indexer modules are designed as clients of the query engine. This means that while the crawler and indexer handle data collection and organization, they rely on the query engine to interpret search queries, ensuring a consistent and centralized interface for accessing information.

### 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.

The core functionality of the module is managed by `WebCrawlerController`. This class 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.

### 2.1.1 Enhancements in the Updated Module

In the updated implementation, the crawler module has been enhanced to integrate with Hazelcast, a distributed in-memory data grid. This improvement allows the module to not only save the filtered books on the local disk but also upload them to a distributed map in Hazelcast. This distributed map, named `datalake`, provides efficient and scalable access to the books for other modules, such as the indexer and query engine.

The integration with Hazelcast allows multiple crawlers on different machines to run concurrently, all contributing filtered books to the same `datalake` map. This distributed approach accelerates the mass download process by sharing the workload, making the system faster and more scalable for large-scale applications.

## 2.2 Indexer Module

In a search engine, an *indexer* processes the raw data 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.

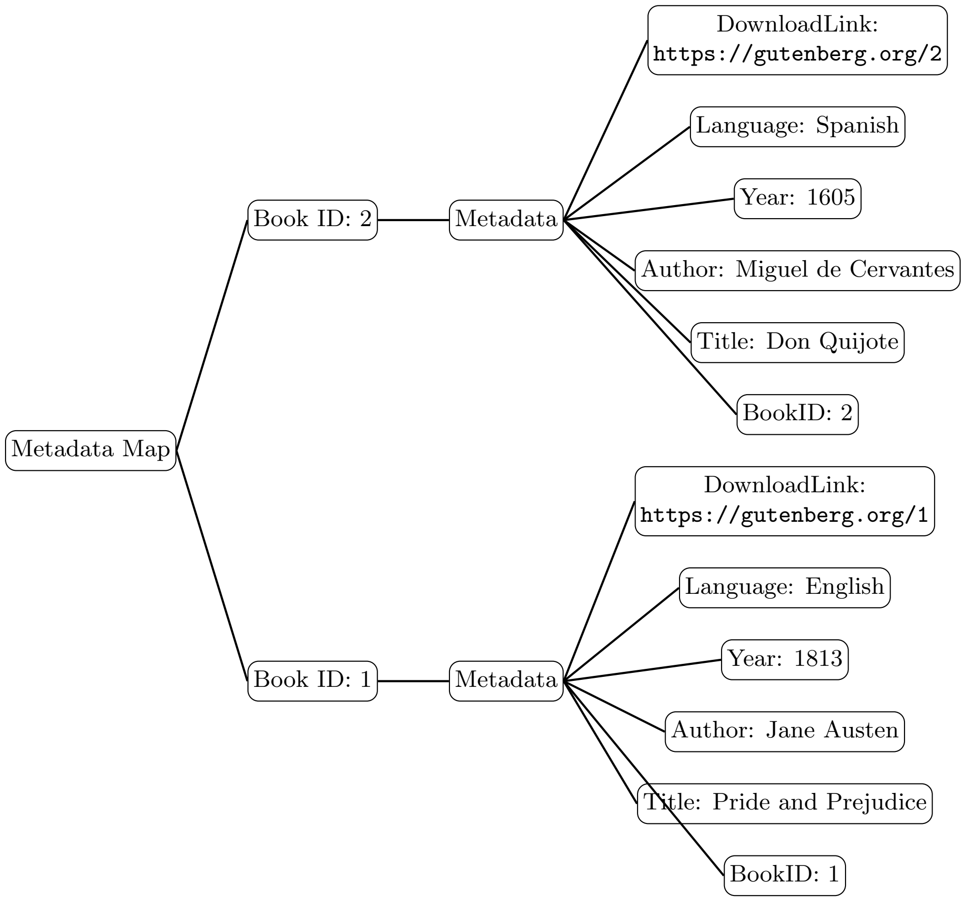
### 2.2.1 Inverted Index

An inverted index is essentially a mapping where each word is associated with the locations (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.

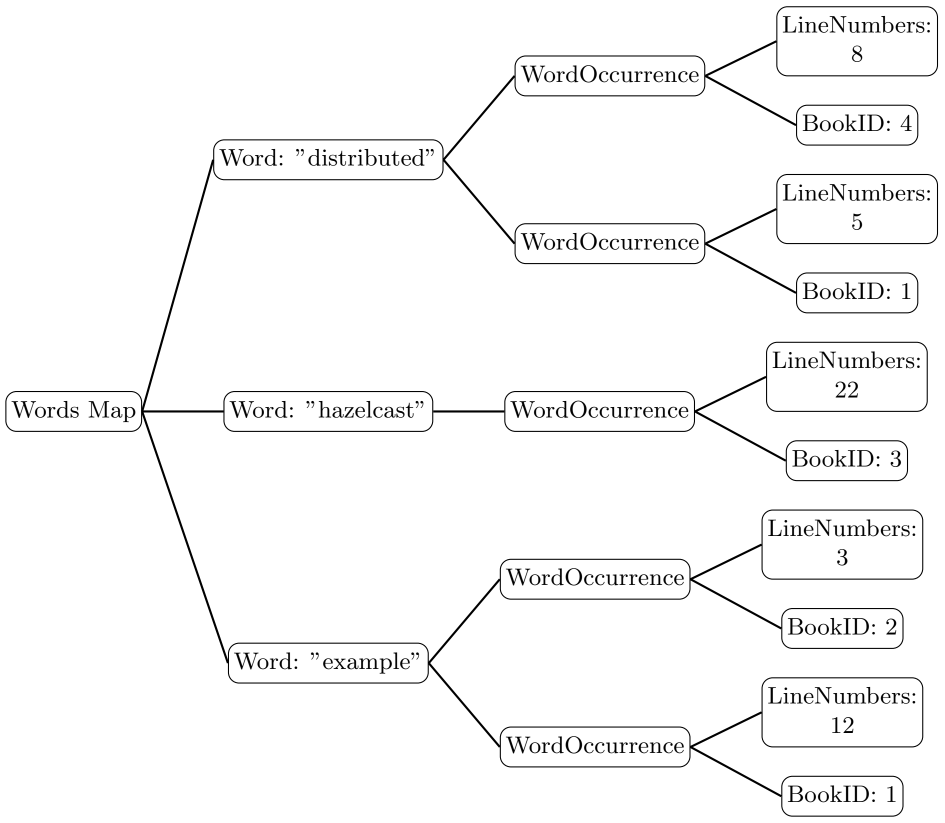
In this implementation, the indexer processes the books stored in the Hazelcast `datalake` map. By accessing the distributed map, the indexer retrieves the filtered books uploaded by the crawler module and processes them to build two additional distributed maps.

### 2.2.2 Data structures

- **`metadata`:** This map contains metadata for each processed book. The key is the book's unique ID, and the value is an object of the `Metadata` class, which holds relevant information about the book, such as its title, author, language.



- **words:** This map is the core of the inverted index. The key is a string representing a word, and the value is a list of WordOccurrence objects. Each WordOccurrence object contains the book ID where the word appears and the specific lines in the book where the word is found.



This distributed architecture, using Hazelcast, ensures that the indexer can efficiently process and store large volumes of data. The metadata and words maps provide structured, scalable, and query-ready data for the search engine's query engine module.

## 2.3 Query Engine Module

The purpose of this module is to process and analyse 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 analysing large volumes of text, searching for keywords, and retrieving detailed information about the books where these words are found. It could be used 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 metadata filtering in queries. This allows users to filter results based on specific metadata attributes, such as language, author, or publication year, making the search results as personalised and relevant as possible. For example, a user searching for the word "revolution" can now filter the results to only include books written in French or published before 1900, narrowing down the search to their specific needs.

Additionally, we have improved the way queries are processed. When a query includes multiple words, the system now prioritises finding books that contain all the specified words, rather than simply listing books that match each word individually. For instance, a query for "love tragedy" will first return books that contain both words, ensuring more relevant matches, rather than books that only mention "love" or "tragedy" separately. This refinement guarantees more accurate and meaningful search results for the user.

## 2.4 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 back-end resources. Built with Sparks, the API is designed to receive HTTP requests, process them, and return structured responses, using the micro-framework to simplify routing and response handling. This architecture allows users to send search requests and retrieve relevant results in an organized and scalable manner.

### 2.4.1 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.
  - The `/test` endpoint was designed to test the performance of the system having some clients doing queries simultaneously. This endpoint sends queries to the system just with words, without using the filter of the metadata.
    This endpoints 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 front-end 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:**
  - Using 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 back-end systems. Its focus on modularity, simplicity, and error handling ensures reliability and ease of use for both developers and end users.

# 3 Program Execution

1. **Query Engine**: The first step is to start the query engine, which acts as the server to handle search requests. The query engine accesses the indexed data in the datamart to provide relevant results.
2. **Indexer**: Once the query engine is running, the indexer module is executed to process the books stored in the datalake.
3. **Crawler**: Lastly, the crawler is executed to download the specified number of books from external sources and store them as text files in the datalake. This step ensures that new data is added to the system for indexing and querying. It is crucial to define the datalake path during this step.

## 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. **Create a Custom Network:** Create the custom network using the following command:

   ```
   docker network create --driver bridge --subnet=10.26.14.0/24 hazelcast-network
   ```

   This creates an internal network for container communication.

3. **Run the query engine container:** Execute the query engine container using the following command:

   ```
   docker run --network hazelcast-network --name queryengine queryengine_image
   ```

4. **Run the indexer container:** Start the indexer container and map the required port:

   ```
   docker run --network hazelcast-network --name indexer -p 5701:5701 indexer-image
   ```

5. **Run the crawler container:** Run the crawler container with the appropriate volume mapping:

   ```
       docker run --network hazelcast-network --name crawler -v

   "/path/to/datalake:/app/mounted-folder" crawler_image java -jar /app/app.jar /app/mounted-folder
   ```

   Replace `/path/to/datalake` with the actual path on your system. Ensure the image name `crawler_image` matches the one loaded from the `.tar` file.

## 4 Problems during the Development

During the development of this third stage, we faced multiple challenges that prevented us from fully meeting the expected objectives. Despite our best efforts, certain issues arose that were beyond our control and limited the successful completion of specific aspects of the project.

One major issue was related to Docker, specifically with redirecting ports in the containers. This problem impeded the proper implementation of Nginx, which was intended to act as a load balancer to distribute queries across multiple servers. Without a correctly configured Nginx, we were unable to achieve the desired level of scalability and efficiency in handling search requests.

While we explored various solutions and devoted significant time to troubleshooting, the combination of technical limitations and time constraints made it impossible to resolve these challenges within the scope of this stage. Nevertheless, the work completed provides a strong foundation for addressing these issues in future iterations.

## 5 Conclusions

In this project, we implemented a search engine in a cluster environment, comprising three core modules: a crawler, an indexer, and a query engine. In the following, we summarise the key conclusions drawn from the development and testing of this stage.

1. **Cluster Implementation with Hazelcast**
   In this stage, we successfully adapted the system to run on a distributed cluster using Hazelcast. This enhancement significantly improved the scalability and performance of the search engine, enabling seamless collaboration between multiple modules and efficient management of large datasets.

2. **Enhanced Query Engine Capabilities**
   The query engine was upgraded to include advanced features such as metadata filtering. Users can now refine their searches based on attributes like language, author, or publication year, making the search results more relevant and personalised. Additionally, the query engine now prioritises books that contain all specified keywords when multiple words are included in a query, improving the accuracy and usefulness of results.

3. **New API Endpoint for Automated Queries**
   A new endpoint, `/test`, was implemented in the API to support automated query testing. This feature allows the system to generate and execute queries automatically, facilitating performance evaluation and stress testing of the query engine under different conditions.

4. **Modular and Scalable Architecture**
   The modular architecture, combined with the integration of Hazelcast, enabled smooth scaling of the system across multiple nodes. This design also simplified the addition of new features and optimisations, such as the metadata filter and enhanced query logic.

5. **Improved User Experience**
   The refinements in the query engine and the addition of metadata filtering provide users with a more intuitive and efficient search experience. The system's ability to handle distributed queries and return precise results makes it well-suited for large-scale, real-time applications.

These enhancements reflect significant progress in the system's development, addressing scalability, accuracy, and usability challenges to deliver a robust and feature-rich search engine.

## 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*