Search Engine: Stage 3 Distributed Computing

Luis Perera Pérez, Jorge Lorenzo Lorenzo,
Owen Urdaneta Morales, Carlos Moreno Vega,
Juan López de Hierro and Christian Toffalore

January 14, 2025

Abstract

In this project, we address the development of a scalable and modular search engine using Java and Maven, focusing on implementing a crawler, indexers, and a query engine. TBD

Contents

1 Introduction

Search engines are fundamental tools for retrieving information from vast datasets, especially in the era of big data. Efficient indexing and querying mechanisms are essential to ensure fast and accurate search results. This project focuses on implementing a modular search engine in Java, utilizing Docker and Hazelcast to enable distributed and asynchronous processing for downloading and indexing books within a cluster of computers. By leveraging these technologies, the project aims to optimize the indexing process and enhance scalability.

The primary motivation of this project is to explore how distributed systems can be used to handle large-scale datasets effectively. By utilizing Docker to containerize the application and Hazelcast for distributed computation, the project demonstrates the potential of modern tools to manage the challenges of big data processing efficiently. This approach ensures scalability, fault tolerance, and high performance in indexing and querying operations.

2 Problem Statement

While search engines are well-studied, efficiently handling the demands of big data remains a challenge. Traditional inverted index implementations may not scale adequately with growing datasets. This project aims to address the following key problems:

- Implement Docker and Hazelcast to make the project highly scalable
- Ensuring scalability and performance in a modular search engine design.

The complexity of these tasks lies in balancing performance metrics such as speed, memory usage, and scalability, while ensuring maintainable and extensible code.

3 Methodology

To ensure proper organization, the development of the project has adhered to a modular structure in Java, comprising three main modules. The first module is the **Crawler**, responsible for downloading the books to be processed into a datalake. The second module is the **TSV-based indexer**, and the third is the Query Engine.

Each of these modules follows its own internal organization, with classes separated into **control**, **interfaces**, and **model** directories, all within the IntelliJ environment where the entire project has been developed.

To optimize processing and scalability, the project is designed to operate across distributed nodes. The workload is divided among multiple nodes, with specific nodes assigned to **Crawlers** and others to **Indexers**. This distributed approach ensures that the tasks of downloading and indexing are efficiently parallelized, significantly reducing processing time and improving performance.

Each node operates independently but collaboratively, leveraging Hazelcast to coordinate and balance the workload dynamically, ensuring fault tolerance and effective utilization of resources.

3.1 Distributed Crawler Module

The distributed crawler operates using a three-step synchronization mechanism to ensure consistency across nodes in a Hazelcast cluster.

1. **Local to Distributed Synchronization:** Each node scans its local file system to identify book files. The book content is then uploaded to a distributed map, avoiding duplicates using an existence check. The node signals completion by updating a shared progress map.
2. **Inter-Node Synchronization:** All nodes wait until every cluster member marks its synchronization as complete. This ensures global consistency before moving to the next phase.
3. **Distributed to Local Synchronization:** Each node retrieves books from the distributed map and writes missing files to its local storage. Existing files are skipped to optimize performance.

This mechanism relies on Hazelcast's distributed data structures (e.g., `IMap`) to coordinate tasks, track progress, and maintain data integrity. The design minimizes conflicts and guarantees eventual consistency in the distributed system.

Additionally, the crawler is responsible for keeping information across all nodes accurate and up-to-date to the greatest extent possible.

3.1.1 Distributed Crawler Classes

DistributedBookDownloader The `DistributedBookDownloader` class handles the downloading of books in a distributed environment. Its main functionalities include:

- `void downloadBooks(HazelcastInstance hazelcastInstance) throws Exception`: Coordinates the distributed downloading of books by assigning book IDs to cluster members and processing the results.
- `private static List<Integer> createBookIds(Integer lastID, int size)`: Generates a list of book IDs starting from the last downloaded ID.
- `private static void saveLastBookId(List<Integer> bookIds)`: Saves the last downloaded book ID to a file for continuity.
- `private static Integer readLastBookId()`: Reads the last downloaded book ID from a file to determine where to resume the process.

- `private static List<String> downloadBooksDistributed(HazelcastInstance hazelcastInstance, List<Integer> bookIds) throws Exception`: Distributes the book downloading tasks across cluster members.
- `private static void waitForClusterSize(HazelcastInstance hazelcastInstance, int clusterSize) throws InterruptedException`: Waits until the cluster reaches a specified size.

DistributedBookSynchronizer The `DistributedBookSynchronizer` class ensures synchronization between local and distributed storage of books. Its main functionalities include:

- `static void synchronizeBooks(HazelcastInstance hazelcastInstance)`: Manages the synchronization of books between local storage and a distributed map.
- `private static void waitForAllNodes(IMap<String, Boolean> progressMap, HazelcastInstance hazelcastInstance)`: Waits for all nodes in the cluster to complete their synchronization tasks.

DistributedMetadataSynchronizer The `DistributedMetadataSynchronizer` class is responsible for synchronizing book metadata between local storage and a distributed map. Its main functionalities include:

- `static void synchronizeMetadata(HazelcastInstance hazelcastInstance)`: Synchronizes metadata from a local CSV file to a distributed map and ensures distributed metadata is saved locally.
- `private static int readMetadataFromCSV(IMap<Integer, Map<String, String>> metadataMap, String datamartDir) throws IOException`: Reads metadata from a CSV file and uploads it to the distributed map.
- `private static int syncMetadataToLocal(IMap<Integer, Map<String, String>> metadataMap, String datamartDir)`: Saves metadata from the distributed map to a local CSV file if not already present.
- `private static Set<Integer> getExistingMetadataIds(String datamartDir)`: Retrieves the IDs of metadata records already present in the local CSV file.

GutenbergBookDownloader the class `GutenbergBookDownloader` implements`BookDownloader` and downloads books out of project Guthenberg, looking for a specific ID

- `GutenbergBookDownloader(String saveDir)`: Constructor that initializes the directory where the books will be stored.

- `void downloadBook(int bookId) throws IOException`: Downloads the book with the specified `ID`. Connects to the appropiate Guthenberg page, obtains the link to the txt format book and downloads it as a UTF-8 encoded file to the directory.
- `private static String getTextLink(Document doc)`: Extracts the link to the plain text file out of the books webpage.

GutenbergBookProcessor the class `GutenbergBookProcessor` implements the interface`BookProcessor`, to download and process books with a specific target id

- `void processBook(int bookId)`: Processes a book starting with its `ID`. Extracts metadata, and stores them in a csv; and cleans and organizes the contentss on the book to store it in a processed format.
- `Map<String, String> extractMetadata(int bookId)`: Extracts the metadata through regex, like title, author or language.
- `void writeMetadata(Map<String, String> metadata)`: Writes the metadata into a csv file, preventing duplicate entries and adding a header if the file is new
- `private String extract(Pattern pattern, String text)`: extracts a specific metadata field out of the book thorugh a targeted regex pattern
- `private static void cleanAndSortMetadata()`: Cleans and orders the metadata
- `private static boolean isDuplicate(String id)`: Verifies if a book with the specified `ID` is already registered in the metadata file

3.2 TSV Indexer Module

The Indexer module is responsible for performing reverse indexing to record the locations of words within the lines of processed books stored in the datalake. This module saves the words in the datamart, specifically in the Indexer section, using TSV files. Each file corresponds to a specific word, and its content includes information such as the book ID, the line where the word appears, and the number of occurrences of the word in that line. In this way, the consolidated occurrences of each word across all books are efficiently organized. The main changes it has gone through relate to synchronization and workload distribution across the modules

It has the following components:

DataLakeWatcher The `DataLakeWatcher` class monitors a directory for new files and processes them using a specified book indexer. Its main functionalities include:

- `DataLakeWatcher(String directoryPath, BookIndexer indexer)`: Initializes the watcher for a given directory and indexer, ensuring the directory exists and is valid.
- `void watch()`: Listens for new files in the directory and adds them to a queue for processing.
- `private void initializeQueue()`: Preloads existing files in the directory into the queue.
- `private void processQueue()`: Processes files from the queue by invoking the indexer on each file.
- `private boolean isValidNumericFile(Path filePath)`: Checks if a file name matches a numeric pattern (`+.txt`).

ProcessedBooksManager The `ProcessedBooksManager` class manages a set of processed books to ensure no duplicates are indexed. Its main functionalities include:

- `ProcessedBooksManager() throws IOException`: Initializes the manager, creating necessary resources and loading previously processed books.
- `private void initializeIndexerResources() throws IOException`: Ensures the directory and file for processed books exist.
- `private void loadProcessedBooks() throws IOException`: Loads previously processed books from the file into memory.
- `boolean isProcessed(String bookId)`: Checks if a book ID has already been processed.
- `void markAsProcessed(String bookId) throws IOException`: Marks a book as processed and updates the resource file.

TSVFileManagerControl The `TSVFileManagerControl` class handles reading and writing operations for TSV files. Its main functionalities include:

- `List<String> readLines(String bookFilePath)`: Reads all lines from a given file into a list.
- `void saveWordsToFile(String word, String bookId, int paragraphIndex, int count)`: Saves word data (book ID, paragraph index, and count) into a structured TSV file within a hierarchical directory structure.

TSVIndexer The `TSVIndexer` class indexes books by processing their content and saving word data into TSV files. Its main functionalities include:

- `TSVIndexer(ProcessedBooksManager processedBooksManager)`: Initializes the indexer with a processed books manager and other utilities.
- `void indexBook(String filePath)`: Checks if a book has already been processed, processes its content, and marks it as processed.
- `private void processContent(String bookId, String bookContent) throws IOException`: Processes the content of a book, cleaning and lemmatizing words, and saves them into TSV files.
- `private String extractBookId(String filePath)`: Extracts the book ID from the file name.

WordCleanerControl The `WordCleanerControl` class cleans and normalizes words. Its main functionality includes:

- `String cleanWord(String word)`: Removes special characters, normalizes to ASCII, and retains only alphabetic characters in a word.

WordLemmatizerControl The `WordLemmatizerControl` class is responsible for lemmatizing words using the Stanford CoreNLP library. Its main functionalities include:

- `private void initializePipeline()`: Initializes the Stanford CoreNLP pipeline if it hasn't been initialized yet. It sets up the necessary annotators like tokenization, sentence splitting, part-of-speech tagging, and lemmatization.
- `public String lemmatize(String word)`: Takes a word as input, converts it to lowercase, and uses the pipeline to annotate the word. It then retrieves and returns the lemmatized form of the word from the tokenized document.

3.3 Query Engine Description

The Query Engine is composed of several key classes that collectively handle the loading of data, processing of user queries, and serving results via HTTP. Below is a detailed description of its components:

3.3.1 Distributed Query Classes

Lemmatizer The `Lemmatizer` class utilizes the `StanfordCoreNLP` library for lemmatizing words. Its main functionalities include:

- `Lemmatizer()`: Initializes the `StanfordCoreNLP` pipeline with properties for tokenization, POS tagging, and lemmatization.
- `String lemmatize(String word)`: Takes a word as input and returns its lemma (base form). If no lemma is found, it returns the original word.

MetadataLoader The `MetadataLoader` class loads book metadata from a CSV file. Its main functionalities include:

- `Map<String, Map<String, String>> loadMetadata(String filePath)`: Reads a CSV file and organizes the metadata into a map, where keys are book IDs and values are maps of metadata attributes (e.g., title, author, language).
- `String[] parseCSVLine(String line)`: Parses a CSV line to handle fields enclosed in quotes or containing commas.

QueryHandler The `QueryHandler` class handles HTTP requests for processing queries. Its main functionalities include:

- `void handle(HttpExchange exchange)`: Processes incoming HTTP `GET` requests, extracts query parameters, and returns search results to the client.

QueryProcessor The `QueryProcessor` class processes search queries using lemmatization and an inverted index. Its main functionalities include:

- `Map<String, Object> processQuery(String query)`: Lemmatizes the query word and searches for results in the index. Returns a map of book details and occurrences.
- `String sanitizeWord(String word)`: Cleans unnecessary characters (e.g., quotes) from a word before processing it.

QueryServlet The `QueryServlet` class is a `HttpServlet` responsible for handling HTTP `GET` requests related to search queries. Its main functionalities include:

- `QueryServlet(QueryProcessor queryProcessor)`: Constructor that initializes the servlet with a `QueryProcessor` instance to handle query processing logic.
- `void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException`: Handles HTTP `GET` requests. It extracts the query parameter `"word"`, processes it using the `QueryProcessor`, and returns the results in HTML format. If the query is missing or empty, it returns an error message.

TSVIndexLoader The `TSVIndexLoader` class loads an inverted index from TSV files in a specified directory. Its main functionalities include:

- `Map<String, Map<String, List<String>>> loadIndex(String baseDir)`: Reads TSV files from the given directory, parses the data, and builds an inverted index mapping words to books and lines.

Main The `Main` class is the entry point for the application, which sets up the HTTP server, loads data, and initializes necessary components. Its main functionalities include:

- `static void main(String[] args) throws IOException`: Starts the application by:
  - Waiting for necessary files and directories to be available.
  - Loading the inverted index using `TSVIndexLoader`.
  - Loading metadata using `MetadataLoader`.
  - Creating a `QueryProcessor` instance with the loaded data.
  - Registering the `QueryHandler` to handle API requests.
  - Starting the HTTP server on port 8080.
- `private static void waitForFiles(String path)`: Checks if the specified file or directory exists, retrying a fixed number of times with delays between attempts. If the file or directory is not found within the retries, it throws a runtime exception.

Using NGINX as a Load Balancer for ApiQuery

NGINX can be utilized as a load balancer to distribute incoming API requests to multiple backend instances of the ApiQuery application. Below is a brief explanation of the process:

1. **Configuration of Upstream Servers:** The backend application (ApiQuery) is deployed on multiple instances running on different ports or servers. NGINX is configured to include an `upstream` block, where the addresses of these instances are specified:

```
upstream backend_api {
    server 127.0.0.1:8081;
    server 127.0.0.1:8082;
    server 127.0.0.1:8083;
}
```

2. **Load Balancing Strategy:** NGINX supports various load balancing strategies such as `round-robin` (default), `least-connected`, or `IP-hash`. These strategies determine how requests are distributed among the backend instances.

3. **Proxying Requests:** Incoming requests to the NGINX server are proxied to one of the backend instances. For example:

```
location /api/query {
    proxy_pass http://backend_api;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
}
```

4. **Fault Tolerance:** NGINX can monitor the health of backend servers and avoid sending requests to servers that are down, ensuring high availability.

5. **Scalability:** By adding more backend instances to the `upstream` block, the system can handle increased load without modifying the application code.

This setup enables efficient request handling, ensuring reliability and scalability for the ApiQuery application.

4 Experiments

For our experiments, we set out to test the performance and scalability of our inverted index implementation. The experiment involved downloading books, processing their content, and indexing the words using a multithreaded approach.

We conducted the experiment in two configurations:

1. On a single machine, where all threads operated locally to manage downloads and indexing.

2. In a simulated distributed environment, using Docker Compose and Hazelcast to create a cluster of nodes across multiple machines.

The setup involved configuring a Hazelcast cluster with Docker Compose, allowing us to simulate and later test the system under a distributed architecture. Both configurations were used to evaluate the potential scalability and behavior of the inverted index in different environments.

The experiment focused on exploring the feasibility of integrating multithreading with a distributed system for large-scale indexing tasks.

4.1 Experiments with Multi-Machine Setup

As part of the experiments conducted in class, we explored the deployment of our system using DockerHub and Docker in a multi-machine setup. The goal was to test the functionality of the Crawler, Indexer, and QueryEngine components across different physical machines, leveraging Docker for containerized

execution and NGINX for load balancing.

The process involved the following steps:

- Images for the `Crawler`, `Indexer`, and `QueryEngine` were uploaded to DockerHub to enable seamless distribution across machines.
- On each machine, the respective Docker image was pulled and executed using Docker commands. This allowed us to deploy the components independently on three different machines.
- The `docker-compose.yml` file was modified to adjust the dynamics of the system. Instead of generating nodes manually, the setup was streamlined to directly execute the `Crawler`, `Indexer`, and `QueryEngine` services on their respective machines.
- The `NGINX.conf` file was updated to specify the IP addresses of the machines hosting the components. This configuration enabled NGINX to perform load balancing across the machines, distributing requests among them.

This setup allowed us to experiment with the deployment and coordination of the system components in a distributed environment, providing insights into the behavior of the system under a multi-machine configuration.

4.2 Challenges Encountered

Several challenges were encountered during these experiments:

- One major difficulty was modifying the IP routes to establish proper connections between the machines. The load balancer did not function as expected, preventing seamless communication.
- The system code was not fully adapted to the laboratory machines. This mismatch caused additional issues, as the code performed differently compared to our local tests.
- Debugging and resolving these connection issues required significant time and effort, highlighting the complexities of deploying distributed systems in heterogeneous environments.

Despite these challenges, the experiments provided valuable experience in managing multi-machine setups and troubleshooting distributed systems.

5 Project Conclusion

Our project successfully established an efficient and distributed system for downloading, processing, and indexing books by leveraging modern technologies such

as Docker and Hazelcast. The architecture is designed to ensure speed, scalability, and robustness. The main workflow starts with a crawler responsible for downloading books from various sources and storing them in a structured format. Subsequently, the distributed indexer, operating on a Docker and Hazelcast-based cluster, processes the books and organizes words into indexes, using the TSV format. This approach guarantees optimal performance and high system availability, even with increasing data volumes.

The modular separation of the system, combined with the use of distributed clusters, not only enhances scalability but also simplifies collaboration between components. This structure allows for adding new features or integrating additional data sources without disrupting the workflow.

Challenges and Lessons Learned

During development, we encountered technical and logistical challenges that enriched our experience. Some of the key difficulties included:

1. **Synchronization in a distributed environment**: While working with Hazelcast, we faced synchronization issues among nodes, particularly when implementing partitions and replications to ensure data consistency. We learned the importance of properly configuring failover and persistence policies to prevent data loss.

2. **Error and failure handling**: Ensuring the system continued functioning despite network failures, download interruptions, or processing errors was a constant challenge. We implemented retry strategies and automatic recovery mechanisms, significantly strengthening the workflow's robustness.

3. **Interoperability between technologies**: Integrating multiple technologies, such as Docker, Hazelcast, and text-processing tools, required time to standardize communication protocols and data formats. This reinforced our ability to design interoperable and modular systems.

In summary, this project not only allowed us to develop a robust and efficient solution for distributed word indexing but also provided valuable lessons on designing distributed systems, handling errors, and teamwork in high-complexity environments.

6 Future Work

For future work, we aim to address several areas of improvement to enhance the system:

- **Efficient Indexing Structure:** Develop a more efficient indexing structure than the one presented in this project. This new structure should be

capable of handling large volumes of data without compromising performance, ensuring better scalability for extensive book collections.

- **Multithreading in the Crawler Module:** Implement multithreading to allow simultaneous book downloading and parallel processing. This enhancement would significantly accelerate these operations, improving the overall speed and optimizing the use of available resources.
- **Task Automation in Docker:** Incorporate a `Task` class to enable periodic and automated execution of tasks. This improvement would simplify the implementation of repetitive workflows, such as indexing new books or updating existing data, ensuring smoother system integration.
- **Icountdownlatch:** implementing Icountdownlatch, similar to java's CountDownLatch, it would allow for a more efficient management of tasks across the different computers
- **Better Indexation:** Implementing more advanced indexing algorithms could significantly speed up the process, and reduce workload
- **Custom File Format:** Implementing a custom file format could significantly reduce space usage and speed up API request answers
- **Dashboard:** Implementing a real time dashboard tool to check the performance and benchamark the network would be an essential tool to prevent problems if the system is scaled up

These proposed improvements aim to increase the overall efficiency of the system while preparing it for future challenges, such as handling large datasets or implementing more robust automation workflows.

7 GitHub Repository

The code for this project can be found in the following GitHub repository:

`https://github.com/LuisPereraPerez/BigData_project`