Inverted Amigos Stage 3

Żukowska, Radosława Sajdokova, Anna Oczowiński, Tymoteusz

Perdomo de Vega, José Luis López Fortes, Eduardo

January 14, 2025

Abstract

This project focuses on building a scalable, distributed system for managing an inverted index and handling large-scale queries. Using Hazelcast for distributed data processing and Docker for containerized deployment, the system efficiently crawls, indexes, and process data from books downloaded from the Gutenberg website. Multiple instances of crawlers, indexers, and search engines are deployed across machines to ensure scalability, while NGINX load balancing optimizes query dispatch among these instances. The system includes an API and search engine, with performance validated through load testing to identify and resolve bottlenecks.

1 Introduction

In the previous stage we implemented multiple data structures implementing inverted index in Java. That solution proved to be much more efficient than the Python one and also new data structures proved to be faster both in term of indexing and searching. However they did have a limit for their efficiency as all these solution are only to be implemented on one computer and are not suitable to make use of scaling the computer resources by introducing more machines.

Often, especially for managing a large workload of data and queries, useful is introducing a distribute system that connects multiple devices/machines. Such systems can process the data much faster and more efficient than a single device as the solution makes use of the resources of all the computers in the network.

This stage of the project implements the Hazelcast for distributed data processing and Docker to enable easy deployment across multiple computers in a lab environment. The distributed system aims to achieve higher scalability, improved performance, and efficient resource utilization for processing and querying large datasets.

2 Problem Statement

Biggest problem met in this stage is managing and querying large-scale datasets which is a big challenge when comes to a single-machine implementation. Such systems lack the scalability needed to process huge amounts of data efficiently and handle high query loads. To solve this problems, the project aims to improve the existing inverted index by enabling distributed processing and modular deployment.

By deploying multiple crawlers, indexers, and search engines across different machines, the system can scale effectively to meet growing requirements and process huge number of data at one time. Additionally, NGINX will be utilized to ensure efficient query distribution across instances, improving system performance and resource utilization.

We can underline main problems solved in this stage:

1. Enhance the inverted index to support distributed processing.
2. Enable modular deployment to allow scaling by deploying multiple crawlers, indexers, and engines across multiple machines.
3. Ensure query distribution and scalability using NGINX for load balancing.
4. Develop an API and search engine for querying the inverted index.
5. Load testing to simulate user queries and identify performance bottlenecks.
6. Deploy the system using Docker, with a cluster composed of multiple search engine containers and a load balancer for efficient resource utilization.

The project also includes developing an API and search engine for querying the distributed inverted index. The entire setup will be containerized using Docker, with the cluster architecture comprising multiple search engine containers and a load balancer, enabling deployment and scalability. This approach provides a practical solution for handling really big data.

3 Solution

The code of this project can be found at our Git Hub Repository

The Docker images for the modules of this project can be found at the following link: Docker Hub Repository

The distributed system consists of multiple nodes working together to crawl and index data. The developed distributed system contains:

- Docker: Containerizes the application for consistent deployment across all laboratory computers.
- Hazelcast: A distributed computing framework used for task distribution, coordination, and data sharing between nodes.
- NGINX: Acts as a load balancer to distribute user queries among the nodes running the query engine
- Crawler and Indexer: Algorithms in Java that fetch books from the Gutenberg website, tokenize the content, and create a file-per-word inverted index.

3.1 Implementation of the Distributed Architecture

The distributed architecture for this project was designed to ensure scalability and efficient handling of large-scale datasets. The implementation uses the Hazelcast, a Java-based distributed computing framework and containerization with Docker and load balancing with NGINX.

1. Distributed Framework with Hazelcast

   Hazelcast is the key component in our distributed system, which allows easy distribution of tasks and data across multiple nodes in the network. For distributing the workload we used fallowing features:

   - Cluster Management: Hazelcast creates a cluster of nodes, each running as a separate Java Virtual Machine (JVM). These nodes communicate and share data in a distributed manner, ensuring scalability.
   - Distributed Data Structures: We used Hazelcast's distributed maps and queues to manage and distribute tasks, such as crawling and indexing, across multiple nodes. This ensured that no single node was overwhelmed with work.
   - Task Distribution: Tasks for crawling books, indexing words, and handling queries were partitioned and distributed among nodes. Hazelcast automatically balanced the workload by dividing tasks evenly across the cluster.

- Dynamic Scalability: The Hazelcast cluster can dynamically scale by adding or removing nodes without disrupting the ongoing processes, making it ideal for a system that needs to adapt to varying workloads.

We consider that is important to discuss why we defined the hazelcast cluster as we did. We are initializing the cluster adding the IP of the members, like this:

```
config.getNetworkConfig().getJoin().getTcpIpConfig()
                .setEnabled(true)
                .addMember("10.26.14.200")
                .addMember("10.26.14.201")
                .addMember("10.26.14.202")
                ...
```

Where the IPs are the ones of the machines that are going to be part of the cluster (in this case, the laboratory machines). However, if we run the application inside a docker container, since it creates a new network, the IPs of the machines are not the same as the ones of the host machine. We can fix that by ading the following line to the code:

```
config.getNetworkConfig().setPublicAddress(args[0]+":5701");
```

Where 'args[0]' is the IP of the host machine and it is passed as an argument to the application when running the application. This way, the hazelcast cluster is going to be created with the correct IPs. It may seem that the intialization of the cluster will only work for the laboratory machines, but it is not true. The cluster can be created in any machine, as long as the IPs are correctly set. So, when we define the members of the claster as we did, we are not limiting the cluster to the laboratory machines, but we are making sure that the cluster is going to be created with the correct IPs, and that for a computer to be part of the cluster, it must have the IP of the laboratory machines. In conclusion, rather than limiting which machines can be part of the cluster, we are in some way, limiting the number of machines that can be part of the cluster.

Taking that into account, we could have written any IP's in that piece on code ant it would have worked as long as the IP's passed as arguments were the correct ones. But for the sake of simplicity, we decided to write the IP's of the laboratory machines.

For the deployment the datamart into the hazelcast cluster, we followed a different approach from the one we used in the previous stages. Instead of having this java data structure into hazelcast "HashMap(String, ResponseList)" (where the key is the word and the value - a ResponseList object - the book id and the appearances) we have two IMultiMap (a data structure native from hazelcast) of the type (String, Integer). The first one maps a word to the bookId in which that word appears, and the second one maps an artificial key (word—bookID) to the appearances of that word within that book. This way, we can have a more efficient way of storing the data and we can have a better performance when querying the datamart.

2. Containerization with Docker

   To enable easy deployment and ensure consistency across multiple devices, the application was containerized using Docker. Before creating the Docker image, we first built the application's artifact. Additionally, we needed to include the stopwords file so that it could be accessible by the application. To achieve this, the stopwords file was moved to the src/resources folder of the project.

   Once these steps were completed, we proceeded to create the Docker image. The key steps in this process were:

   - Creating Docker Image: Docker image was created for the main application. The image contains the necessary Java runtime, dependencies, and the application code.
   - Networking: Hazelcast nodes running inside containers discover each other automatically through the shared network that the system could communicate efficiently.

   The Docker image was created using the following Dockerfile:

```
# Usa una imagen base de OpenJDK 17
FROM openjdk:17-jdk-slim

# Crea un directorio dentro del contenedor para tu aplicaci n
WORKDIR /app

# Copia tu archivo JAR al directorio de trabajo en el contenedor
COPY out/artifacts/Inverted_Amigos_Stage3_jar/Inverted_Amigos_Stage3.jar /app/
    Inverted_Amigos_Stage3.jar

# Define el comando que se ejecutar  cuando se inicie el contenedor
ENTRYPOINT ["java", "-Xms4g", "-Xmx8g", "-jar", "Inverted_Amigos_Stage3.jar"]
```

When we were testing the API with the 2000 documents, we noticed that the time of response was very high. We realized that all the nodes had little heap memory free, so we decided to allocate more heap memory to the nodes. We did that by adding the options '-Xms4g' and '-Xmx8g' to the entrypoint of the docker image. This way, the nodes will start with 4GB of heap memory and can use up to 8GB of heap memory.

3. Load Balancing with NGINX

   NGINX was configured as the load balancer to distribute user queries across multiple instances of the query engine. Its implementation includes:

   - Reverse Proxy Configuration: NGINX acts as a reverse proxy, forwarding incoming requests from clients to the appropriate query engine instance.
   - Load Balancing Algorithms: A round-robin algorithm which is choosing all elements in a group equally in some rational order, usually from the top to the bottom of a list and then starting again at the top of the list and so on was used to evenly distribute the query load among all available query engine instances. This ensures optimal resource utilization and prevents bottlenecks.
   - Fault Tolerance: If a query engine instance becomes unavailable, NGINX automatically redirects queries to other healthy instances, maintaining system reliability.

4. Application Implementation in Java

   The application was developed entirely in Java, following a modular design. Key components include:

   - Crawler Module: The crawler fetches books from the Gutenberg website and stores the data in the distributed system. It is designed to handle multiple threads, enabling parallel crawling of large datasets.
   - Indexer Module: The indexer processes the crawled data and builds a file-per-word inverted index it tokenizes the downloaded content using the `GutenbergTokenizer` and creates an inverted index for the words using the `FilePerWordInvertedIndexHazelcast` implementation. The data is stored in a file-per-word format to enable efficient querying in a distributed environment. This module uses Hazelcast's distributed maps to store index data across the cluster.
   - Query Engine: The query engine is implemented as a Java-based REST API. It processes user queries by searching the distributed inverted index and returning results. It also includes a CLI edition for command-line interaction.

5. Workflow of the Crawler and Indexer

   - Initialization:

     Hazelcast initializes the `pagesMap` with page numbers and their processing statuses. The application ensures that the map is initialized only once, even in a distributed setup.
   - Task Execution:

     Each node fetches the next unprocessed page from the `pagesMap`. The crawler downloads books corresponding to the assigned page. The indexer processes the downloaded content and updates the inverted index.

- Metadata distribution

  Creating metadata for the books that are downloaded and indexed is done locally. This metadata includes crucial details such as book IDs, titles, authors, and other relevant attributes, serving as a structured representation of the dataset. The metadata generation process ensures that each book is accurately documented and ready for efficient querying and retrieval.

  Once the metadata is created, it is uploaded to Hazelcast, which acts as a distributed in-memory data grid. By using Hazelcast, the metadata is made available across all nodes in the distributed system. This ensures that every instance of the crawlers, indexers, and query engines can access and utilize the same set of metadata, enabling coordination and efficient distribution of tasks. This approach data integration and provides distributed system to manage the metadata increasing the system's scalability and performance. Every one more node(computer) in cluster will increase the performance of the whole distributed system.

6. Scalability and Performance

   The distributed architecture allows the system to scale by adding more nodes to the Hazelcast cluster. Load testing with simulated user queries demonstrated that the system can handle increasing workloads efficiently, distributing tasks and queries.

By combining Hazelcast's distributed computing capabilities with Docker's containerization and NGINX's load balancing, the project achieved scalable architecture suitable for processing and querying large datasets in a distributed environment.

4 API Specification

The solution provides a simple API for retrieving the information. The API was implemented using Java and the Spring Boot framework. It works as an interface for querying the indexed data generated by the distributed crawler and indexer system.

- API endpoint GET /stats/:type

  This endpoint provides statistical information about the system based on the requested type. The following types of stats are supported:

  `totaldocuments`: Returns the total number of documents indexed by the system.

  `uniquewords`: Returns the total number of unique words indexed.

  `processingtime`: Returns the average processing time for indexing a document.

  `nodesstatus`: Returns the status of all nodes in the Hazelcast cluster.

- API endpoint GET /documents/:words?from=...to=...author=...language=...title=...

  This endpoint retrieves documents containing the specified words with additional filtering options, including language and title.

  `words:` A list of words separated by + (e.g., word1+word2+word3).

  Query Parameters:

  `from`: Start date for filtering documents (optional, format: YYYY).

  `to`: End date for filtering documents (optional, format: YYYY).

  `author`: Filter documents by the author's name (optional).

  `language`: Filter documents by language (optional).

  `title`: Filter documents by title (optional).

  The response contains a list of matching documents, with each document's metadata and matching criteria.

5 Experiments

We went to the laboratory several times to test the application in the following scenarios:

- 13 nodes crawling, indexing and responding to queries with 2000 documents; 1 load balancer and 7 clients sending queries (using the code provided in the virtual campus). The indexing time was about 5 minutes which is not incredibly fast, but it is acceptable. We think the reason is that, even though we were using the putAllAsync method, they were still 13 nodes trying to acquire the lock of the map. The response time was between 4-6 seconds, which is not bad, but it is not great either. We think the reason for this is that the datamart was disributed between many nodes, and that could be affecting the time of retrieval. However, the system only threw one timeout exception in approximately 7000 queries, which is a good result.
- 7 nodes crawling, indexing and responding to queries with 2000 documents; 1 load balancer and 7 clients sending queries. The indexing time was slightly worse than the previous test, but the response time was better, between 2-4 seconds. However, when the number of queries increased, the response time increased as well and the system threw more timeout exceptions and the number of queries per round decreased because of that. We were right in our hyptothesis that the number of nodes could be affecting the response time but in long term, it could be affecting the system's performance.

This is an example of a trade-off that we had to make. We could have less nodes to distribute the data and have a better response time, but that could be affecting the system's performance in the long term. We could have less nodes to have a better performance, but that could be affecting the response time. We decided to have more nodes because we think that the response time is acceptable and that the system's performance is not going to be affected in the long term.

6 Conclusion

The distributed architecture implemented in this project underline efficient solution for working on big data and large volumes of data from the Gutenberg repository. The combination of useful technologies such as Hazelcast for distributed computing, Docker for containerization, and Spring Boot for API development has resulted in a high-performance system capable of handling complex data retrieval and processing tasks. Fallowing implementations were done during this stage.

- Distributed Crawling and Indexing

  Distributed crawling and indexing tasks across multiple nodes using Hazelcast. Each node independently processes portions of the dataset, ensuring parallel execution and faster completion of tasks.

  The use of a `pagesMap` ensures no duplication in processing, as nodes coordinate via Hazelcast locks to track completed tasks. The distributed approach significantly reduces the time required to index large volumes of data, making the system scalable and suitable for large-scale datasets.

- Scalability and Load Balancing with NGINX

  Deployment of the API on multiple nodes using Docker containers. Load balancing implemented via NGINX to distribute user queries across all nodes efficiently.
  This setup enables equal distribution of computational load, preventing bottlenecks and reliability. The system can handle increased user traffic and queries without lowering in performance.

- Querying and Search Functionality

  Development of a Spring Boot API with endpoints for:

  Retrieving specific text fragments based on word position and document ID (GET /text).

  Searching for documents containing specific words with optional filters for author, date range, title, and language (GET /documents/:words).

Integration of the HazelQueryEngine to query distributed data efficiently supporting for queries with multiple words and criteria for data retrieval. Users have access to an API that allows to produce queries.

- Containerization and Deployment

  Key aspect was easy deployment of the application across multiple machines. We used containerization of the application using Docker. Deployment of containers on multiple laboratory computers to form a distributed network. Coordination of crawlers, indexers, and queries using Hazelcast and NGINX. The system can be deployed easily on any environment, reducing setup complexity.

- Distributed Query Processing

  We used distribution to optimize query performance in large datasets across a distributed system. Hazelcast's distributed data structures allow parallel processing of queries on multiple nodes. Queries are distributed and executed simultaneously across nodes, using the distributed file-per-word index. The system provides fast query results, even for huge datasets.

- KEY ASPECTS OF THE PROJECT

  Distributed Systems: The project gave us the experience in designing, implementing, and deploying distributed systems.

  Containerization: We learned how to use Docker technology for containerization which allows simplified deployment and scalability of the application.

  API Development: Development of RESTful APIs provide understanding of query handling and user interaction with the system.

  Hazelcast Integration: Using Hazelcast we could learn how to work with distributed systems and how the integrate and synchronize them.

  Scalable Architecture: The project underlined the importance of scalable and fault-tolerant architectures for large-scale applications especialy in field of Big Data.