Search Engine

Ivan Anikin

1. Abstract

This paper details the design and implementation of a Java-based search engine leveraging an inverted index for efficient word querying. The project comprises three distinct modules: a web crawler for downloading and managing books, an indexer that incorporates both expanded and aggregated data structures, and a query engine for processing search requests. To ensure reproducibility and platform independence, the entire system is encapsulated in portable containers using Docker.

The benchmarking results highlight the advantages of the expanded index in terms of faster indexing speeds, while the aggregated index demonstrates slightly superior performance in single-word query scenarios. Ultimately, the expanded structure was selected for its balanced trade-off between speed and scalability, aligning well with the anticipated requirements of future project developments.

2. Introduction

The objective of this project is to create an efficient and robust search engine using an inverted index, a data structure designed to store words and their positions within documents. This approach facilitates rapid and precise searches by not only identifying which books contain specific terms but also pinpointing their exact locations within the text.

Web crawler periodically retrieves books from Gutenberg.org. The crawler saves the downloaded books in a datalake and organizes their metadata in a CSV file for further processing.

The inverted index employed in this project supports efficient word searching by associating terms with their occurrences across documents.

The evaluation of these implementations involved benchmarking tools to measure execution times and determine the most efficient approach for subsequent stages of the project.

To manage distributed storage and enable efficient query execution, the project integrates Hazelcast, a distributed in-memory data grid. Hazelcast provides a scalable platform for managing the metadata, book content, and index data across nodes. Queries are processed through a REST API developed using Spring Boot, allowing the system to return search results to a web browser or other client interfaces seamlessly. This integration ensures high availability and fault tolerance, making the system robust and scalable.

The project also integrates Docker to ensure consistent and reproducible execution environments. By leveraging containerization, the system can be deployed and tested seamlessly across different platforms, enhancing its portability and reliability.

3. Methodology

3.1 Crawler

The Crawler is responsible for downloading books from the Gutenberg website and storing their content and metadata in a distributed Hazelcast Map storage system. This approach replaces the use of local directories and CSV files, ensuring scalability, data redundancy, and faster access in a distributed environment. The process of retrieving books, saving their content, and managing metadata is automated and streamlined with Hazelcast.

3.1.1 Classes and Key Methods

CrawlerCommand:
This class orchestrates the crawling process by coordinating the download of books. It tracks the last downloaded book ID using Hazelcast Maps and retrieves the next set of books. The process is scheduled to run periodically, leveraging Hazelcast's distributed task execution features to ensure consistent updates across nodes.

Crawler:
The core class responsible for downloading books from the Gutenberg website. Instead of saving the content as text files, the book's content is stored as byte arrays in a Hazelcast IMap, ensuring efficient and distributed storage. Metadata, including the book's title and author, is also stored in a separate IMap within Hazelcast. This eliminates the dependency on CSV files and enhances scalability.

The process includes:

- Downloading the book content by its ID and converting it into a byte array.

- Scraping the book's title and author information from the Gutenberg page using data from the <h1> HTML element.
- Storing the metadata in the metadataMap Hazelcast Map, which associates each book ID with its metadata.
- Storing the book content in the bookMap Hazelcast Map, where each book ID corresponds to the byte array representation of the book.

3.2 Indexer

The Indexer is responsible for processing books and creating an inverted index structure to enable fast word retrieval. By leveraging Hazelcast Map storage instead of local directories or hierarchical CSV files, the indexer gains significant advantages in terms of scalability, fault tolerance, and query speed. Two indexing strategies, Expanded Indexer and Aggregated Indexer, were implemented, both utilizing Hazelcast for efficient distributed storage.

3.2.1 Components and Methods

Book:
This class represents each book to be indexed. The getBookId() and getContent() methods retrieve the book's ID and content, respectively. The content can also be modified during preprocessing, ensuring consistency before indexing.

DistributedIndexMap:
Instead of relying on a hierarchical CSV store, both indexing strategies use a Hazelcast IMap to store the index. The keys in the indexMap represent words, and the values are maps where each key is a Book ID and the associated value is a list of word positions within the book. This distributed approach enables real-time updates and fast access across nodes.

HazelcastBookReader:
Replacing the GutenbergBookReader, this component reads books directly from the bookMap in Hazelcast. The read() method retrieves the byte arrays stored in the distributed map, converts them into Book objects, and parses their content for indexing. This eliminates the dependency on local files and ensures that books are accessible across a distributed system.

3.3 Query Engine

The Query Engine plays a critical role in the search engine architecture by interpreting user inputs, locating indexed terms, and delivering humanreadable results. By replacing CSV-based storage with Hazelcast Map storage, the Query Engine achieves faster data access, seamless scalability, and enhanced fault tolerance. This enables efficient query processing, even in a distributed environment.

3.3.1 Key Components and Methods

Main:

Initializes the Search Engine by setting up instances of input, output, and query engine components. It manages user input and coordinates the query process, ensuring smooth interaction between various system modules.

SearchEngineCommand:

Acts as a mediator between the query and indexing components.

This class:

- Initializes Hazelcast Maps for storing book metadata, indices, and book content.
- Processes user input through the Search class.
- Executes queries using the CommonQueryEngine class.
- Outputs search results through the SearchOutput class, retrieving data directly from Hazelcast for high-speed operations.

CommonQueryEngine:

Responsible for executing queries, this class leverages Hazelcast Maps to store and retrieve data dynamically

Metadata Storage:

Book metadata (ID, title, author, and URL) is stored in the metadataMap Hazelcast Map, removing the need for CSV files. Each metadata entry is accessible via the book ID.

- **Index Storage:**
  Word indices are stored in the indexMap Hazelcast Map. The keys represent words, while the values are maps associating Book IDs with occurrences and positions of the words in the text.

- **Query Processing:**
  For each search term, the engine retrieves the associated data from indexMap, identifies common books containing all terms, and finds the specific paragraphs where the terms appear.

- **Paragraph Extraction:**
  The ParagraphExtractor reads book content directly from the bookMap Hazelcast Map. It identifies and highlights occurrences of the search terms, counts them, and extracts relevant paragraphs for display.

4 Docker Integration

To ensure consistent and reproducible environments for running the search engine, Docker was used to containerize the application. By encapsulating the application and its dependencies into portable containers, Docker provides a robust solution for deployment, testing, and scaling.

Modular Containerization

Each module of the search engine (Crawler, Indexer, and Query Engine) is containerized individually to enhance modularity and maintainability. This approach allows for independent development, testing, and deployment of each component, minimizing dependencies and potential conflicts.

Dockerfile Design

For each module, a dedicated Dockerfile is created, leveraging a multi-stage build process to optimize the container size and runtime performance:

- **Multi-Stage Build Process:**
  1. **Build Stage:**
     - A Maven-based build environment is set up, where the working directory is defined, and the pom.xml and source files are copied into the container.
     - The mvn package command is executed to compile the module and produce a JAR file, with tests skipped to accelerate the build process.
  2. **Runtime Stage:**
     - A clean, minimal base image is used for the runtime environment.

        - The compiled JAR file from the build stage is copied into this runtime image, ensuring that only the necessary artifacts are included in the final container.

Container Management

Docker Desktop is utilized to manage the containerized environment, providing tools for building, running, and monitoring containers. By maintaining separate containers for each module, the system achieves:

1. **Isolation:** Dependencies are isolated within their respective containers, preventing conflicts between modules.
2. **Modularity:** Each module can be updated or replaced independently without affecting other components.
3. **Optimization:** The minimal runtime image reduces resource usage and improves startup times.

Deployment and Scalability

Using Docker Compose or Kubernetes, the containers can be orchestrated to run in a distributed setup. This enables horizontal scaling of individual modules, such as deploying multiple instances of the Query Engine to handle increased query loads.

Benefits of Docker Integration

1. **Consistency:** Docker ensures that the application behaves identically across different environments, eliminating issues caused by varying system configurations.
2. **Portability:** The containerized application can be deployed across any system that supports Docker, enhancing flexibility.
3. **Efficiency:** The multi-stage build process reduces image sizes and minimizes resource overhead during runtime.
4. **Reproducibility:** Developers and testers can recreate the production environment locally, enabling efficient debugging and testing.

By leveraging Docker, the search engine project achieves a modern and scalable deployment architecture, ensuring reliability and efficiency in diverse environments.

5 Execution steps

mvn clean install

docker network create shared-network

docker build -f crawler/Dockerfile -t crawler-image .

docker run -d --name crawler-container --network shared-network crawler-image

docker run -d --name crawler-container2 --network shared-network crawler-image

docker run -d --name crawler-container3 --network shared-network crawler-image

docker build -f indexer/Dockerfile -t expanded-indexer-image .

docker run -d --name expanded-indexer-container --network shared-network expanded-indexer-image

docker build -f query-engine/Dockerfile -t query-engine-image .

docker run -it --name query-engine-container -p 8080:8080 --network shared-network query-engine-image

http://localhost:8080/api/query-engine/search?query=test