Search Engine: Stage 2 Development and Benchmarking

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

November 17, 2024

Abstract

In this project, we address the development of a scalable and modular search engine using Java and Maven, focusing on implementing a crawler, two distinct indexers, and a minimal query engine. Each indexer employs a different data structure for the inverted index, enabling a detailed comparison of their performance and scalability under big data workloads.

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, comparing the performance of different data structures for the inverted index.

Previous studies and tools have demonstrated the potential of scalable and efficient search engines. Common approaches involve designing robust crawlers to gather data, constructing optimized indexers to handle large-scale datasets, and implementing query engines capable of handling complex queries. Our contribution builds upon these principles by employing Java, Maven, and JMH for benchmarking, thus combining professional-grade development tools with rigorous performance evaluation.

The main motivation of this project is to explore and optimize data structures for inverted indexes to achieve scalable and efficient search capabilities in a big data context. We aim to provide a detailed analysis and comparison of the selected structures, contributing to the field's knowledge base.

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:

- Selecting and benchmarking two distinct data structures for the inverted index in Java.
- Ensuring scalability and performance in a modular search engine design.
- Integrating and benchmarking components using professional tools like Maven and JMH.

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 **JSON-based indexer**, and the third is the **TSV-based indexer**.

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.

3.1 Crawler Module

The Crawler is the part of the code responsible for downloading the data to be used in the project, as well as preprocessing it. In this case, that data consists of books from the Gutenberg Project.

The workflow is designed to ensure continuity between sessions. The last downloaded book's ID is tracked using a text file named `last_id.txt`. This file is read to determine where the crawler should resume downloading, and it is updated after each successfully downloaded and processed book.

3.1.1 Crawler Interfaces

BookDownloader The `BookDownloader` interface defines the contract for downloading books from the web. It declares the method:

- `void downloadBook(int bookId) throws IOException`: Downloads the book corresponding to the given `bookId`.

BookProcessor The `BookProcessor` interface specifies the operation for processing a downloaded book. It declares the method:

- `void processBook(int bookId)`: Processes the book with the given `bookId` to clean and organize its content.

GutenbergCrawler The `GutenbergCrawler` interface provides the functionality to manage the crawling process. It declares the method:

- `void startCrawling(int numBooks)`: Starts downloading and processing a specified number of books.

LastIdManager The `LastIdManager` interface defines methods for managing the tracking of the last downloaded book ID. It declares:

- `int getLastDownloadedId()`: Retrieves the ID of the last downloaded book by reading the `last_id.txt` file.
- `void updateLastDownloadedId(int lastId)`: Updates the `last_id.txt` file with the new last downloaded book ID.

This interface ensures that the crawler can seamlessly resume downloading from where it left off in the previous session.

MetadataExtractor The `MetadataExtractor` interface specifies methods for extracting metadata from a book. It declares:

- `Map<String, String> extractMetadata(int bookId)`: Extracts and returns metadata for the book with the given `bookId`.

MetadataWriter The `MetadataWriter` interface provides methods for storing metadata in a structured format. It declares:

- `void writeMetadata(Map<String, String> metadata)`: Writes the provided metadata to the storage medium.

3.1.2 Crawler Control Classes

BookManager The `BookManager` class implements the overall book processing workflow and serves as a central coordinator. It implements no direct interface but relies on other components:

- A `BookDownloader` to manage downloading books.
- A `MetadataExtractor` to extract metadata from downloaded books.
- A `BookProcessor` to process the book content.
- A `MetadataWriter` to store extracted metadata.

The `handleBook` method integrates all these components, ensuring proper coordination and error handling during the book processing lifecycle.

CSVMetadataWriter The `CSVMetadataWriter` class implements the `MetadataWriter` interface. It is responsible for recording book metadata into a CSV file. Its key functionalities include:

- Writing a header row when the file is first created.
- Escaping special characters in metadata fields to ensure proper CSV formatting.
- Avoiding duplicate entries by checking if the metadata already exists in the file.

This ensures consistent and clean storage of metadata for all processed books.

FileLastIdManager The `FileLastIdManager` class implements the `LastIdManager` interface. It handles the tracking of the last downloaded book ID using a text file named `last_id.txt`. Its key functionalities include:

- Reading the last ID from `last_id.txt` to determine where to resume downloads.

- Updating the file with the latest ID after each successfully downloaded book.
- Creating the file if it does not exist, initializing it with a default value of 0.

This class is crucial for ensuring that the crawler can continue its operations without duplicating previously downloaded books.

GutenbergBookDownloader The `GutenbergBookDownloader` class implements the `BookDownloader` interface. It is responsible for downloading books from the Project Gutenberg website. It:

- Extracts the URL for the plain text file of a given book using the Jsoup library.
- Downloads the content of the book using Apache HTTPClient.
- Saves the book as a text file in the specified directory.

The class handles exceptions and network-related errors to ensure robustness and reliability during the download process.

GutenbergBookProcessor The `GutenbergBookProcessor` class implements the `BookProcessor` interface. It processes and cleans the content of books by:

- Extracting the content between the "START" and "END" markers of Project Gutenberg books.
- Removing unnecessary text and organizing the content into paragraphs.
- Writing the processed content back to the original text file.

GutenbergCrawlerSequential The `GutenbergCrawlerSequential` class implements the `GutenbergCrawler` interface. It manages the sequential crawling and downloading of books. It:

- Starts downloading from the last recorded book ID.
- Retries downloads a configurable number of times in case of errors.
- Updates the last downloaded ID after each successfully processed book.

This class coordinates the flow of book downloads and ensures continuity between sessions.

GutenbergMetadataExtractor The `GutenbergMetadataExtractor` class implements the `MetadataExtractor` interface. It extracts metadata from book files by:

- Searching for patterns in the book text to extract fields such as title, author, and release date.
- Returning a map of metadata fields for further use.
- Handling cases where specific metadata fields are missing or unavailable.

This ensures that all relevant metadata is collected for each book.

Main The `Main` class serves as the entry point for the entire application. It is responsible for initializing and orchestrating all the components required for the crawling and indexing process. Key responsibilities include:

- Initializing the `GutenbergBookDownloader`, which implements the `BookDownloader` interface.
- Setting up the `GutenbergMetadataExtractor`, which implements the `MetadataExtractor` interface.
- Configuring the `GutenbergBookProcessor`, which implements the `BookProcessor` interface.
- Using the `CSVMetadataWriter`, which implements the `MetadataWriter` interface.
- Initializing the `FileLastIdManager`, which implements the `LastIdManager` interface.
- Creating a `BookManager` to coordinate the download, processing, and metadata extraction workflows.
- Setting up the `GutenbergCrawlerSequential`, which implements the `GutenbergCrawler` interface, to sequentially crawl and process the specified number of books (`NUM_BOOKS`).

The `main` method integrates all these components and starts the crawling process by invoking `crawler.startCrawling(NUM_BOOKS)`. This method ensures that the workflow begins with the last downloaded book ID and processes up to the configured number of books. The application handles errors gracefully, allowing retries for failed downloads and ensuring continuity between sessions.

3.2 Json Indexer Module

3.2.1 Json Indexer Interfaces

BookIndexer The `BookIndexer` interface defines the contract for indexing books. It declares the method:

- `void indexBook(int bookId) throws IOException`: Indexes the content of the book corresponding to the given `bookId`.

JsonFileManager The `JsonFileManager` interface provides methods to read and write JSON files. It declares:

- `Word readJson(String filePath) throws IOException`: Reads and deserializes a JSON file into a `Word` object.
- `void writeJson(String filePath, Word word) throws IOException`: Serializes and writes a `Word` object to a JSON file.

LastBookManager The `LastBookManager` interface handles operations related to tracking the last processed book ID. It declares:

- `int readLastProcessedBookId(String filePath) throws IOException`: Reads the ID of the last processed book from the given file.
- `void updateLastProcessedBookId(String filePath, int lastBookId) throws IOException`: Updates the file with the new last processed book ID.

WordCleaner The `WordCleaner` interface defines a method for cleaning individual words. It declares:

- `String cleanWord(String word)`: Cleans a word by removing unwanted characters and normalizing it.

WordLemmatizer The `WordLemmatizer` interface provides a method for lemmatizing words. It declares:

- `String lemmatize(String word)`: Converts a word to its base form (lemma).

3.2.2 Json Indexer Model Classes

This part shows the structure that is perform to hold the words data. The primary purpose of the `Word` class, as well as the other two, is to store all the necessary information about a word in an organized structure. This structure allows for seamless serialization and deserialization into JSON format, facilitating efficient data storage and retrieval for further processing or analysis.

BookAllocation The `BookAllocation` class stores information about a word's occurrences in a specific book. It includes:

- `int times`: The number of times the word appears in the book.
- `List<Position> positions`: A list of positions where the word appears in the book, represented as `Position` objects.

Position The `Position` class represents the location of a word in a book. It includes:

- `int line`: The line number where the word appears.
- `int wordIndex`: The index of the word within the line.

Word The `Word` class represents a word and its associated information. It includes:

- `String word`: The word itself.
- `Map<String, BookAllocation> allocations`: A map where the key is the book ID and the value is the allocation information (`BookAllocation`) for that book.
- `int total`: The total number of occurrences of the word across all books.

3.2.3 Json Indexer Control Classes

BookManagerControl The `BookManagerControl` class implements the `LastBookManager` interface. It manages:

- Reading the ID of the last processed book from a file.
- Updating the file with the latest processed book ID.
- Creating the file if it does not exist and initializing it with 0.

JsonFileManagerControl The `JsonFileManagerControl` class implements the `JsonFileManager` interface. It provides methods for:

- Reading `Word` objects from JSON files.
- Writing `Word` objects to JSON files.

It uses the Google Gson library for serialization and deserialization.

WordCleanerControl The `WordCleanerControl` class implements the `WordCleaner` interface. It cleans words by:

- Removing special characters and punctuation.
- Normalizing the word to remove non-ASCII characters.
- Returning a trimmed and cleaned version of the word.

WordLemmatizerControl The `WordLemmatizerControl` class implements the `WordLemmatizer` interface. It uses the Stanford CoreNLP library to lemmatize words by:

- Annotating the word with linguistic information.
- Extracting and returning the lemma of the word.

IndexerControl The `IndexerControl` class implements the `BookIndexer` interface. It is responsible for:

- Reading the content of books and breaking it into words.
- Cleaning and lemmatizing each word using the `WordCleaner` and `WordLemmatizer`.
- Storing the occurrences of each word, along with its position, in a JSON file.
- Merging new data with existing JSON files if the word already exists.

The `executeIndexing` method ensures that only unprocessed books are indexed by reading the last processed book ID and iterating through available files in the `datalake/books` directory.

Main The `Main` class initializes and executes the indexing process. It:

- Creates an instance of `IndexerControl`.
- Calls the `executeIndexing` method to index all unprocessed books.
- Handles exceptions and provides feedback on the indexing process.

3.3 TSV Indexer Module

The Indexer 2 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 2 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.

3.3.1 TSV Indexer Interfaces

FileHandler The `FileHandler` interface manages operations related to handling books from the datalake and saving files in the datamart within the Indexer 2 module. It provides the following methods:

- `List<String> readLines(String filePath)`: Reads lines from a file located at the specified `filePath`. Returns a list of strings where each element corresponds to a line in the file.
- `List<String> loadBooks()`: Loads and returns a list of identifiers or paths for books stored in the datalake, preparing them for processing and storage in the datamart.

Indexer The `Indexer` interface serves as the main interface for the Indexer 2 module, integrating and utilizing all other interfaces necessary for its execution. It provides the following method:

- `void execute()`: Initiates the overall indexing process by coordinating the operations of various interfaces, managing the indexing of words, and organizing data from the datalake into the datamart.

WordDataHandler The `WordDataHandler` interface is responsible for handling the lemmatization, cleaning, and splitting of words within the Indexer 2 module. It provides the following methods:

- `String lemmAdd(String word)`: Finds and returns the lemma of the given word, simplifying it to its base form.
- `String cleanWord(String word)`: Cleans the word by removing special characters, numbers, and unwanted symbols, leaving only relevant text.
- `List<String> cleanAndSplit(String paragraph)`: Cleans and splits a paragraph into a list of individual words, preparing the text for indexing.

3.3.2 TSV Indexer Control Classes

BookIndexer The `BookIndexer` class orchestrates the indexing process for books stored in the `datalake` and generates reverse indexes saved in the `datamart`. It implements the `Indexer` interface and depends on other components for its operations:

- **FileHandler**: Handles the reading of book files from the datalake and prepares them for processing.
- **WordDataHandler**: Cleans, splits, and lemmatizes words from the book paragraphs.
- **TsvFileHandler**: Manages the creation and updating of TSV files where the reverse index is stored.

The key methods and their roles are:

- `void execute()`: The primary method to execute the indexing process. It identifies and skips already indexed books, reads new books, processes their content, and updates the reverse index.
- `private void processBook(String bookId, List<String> paragraphs)`: Processes the paragraphs of a given book by:
  - Cleaning and lemmatizing words.
  - Counting word occurrences in each paragraph.
  - Saving the results to a TSV file.
- `private Set<String> loadIndexedBooks()`: Loads the list of already indexed books from a file stored in the `resources` directory.

- `private void saveIndexedBooks()`: Saves the last indexed book's ID into the `lastBookId.txt` file to ensure continuity in future indexing.
- `private String getLastIndexedBookId()`: Reads the ID of the last indexed book from `resources/lastBookId_indexer2.txt`.

This class ensures efficient coordination between its components, allowing for a structured and reliable reverse indexing process while avoiding redundant operations.

TsvFileHandler The `TsvFileHandler` class implements the `FileHandler` interface and is responsible for managing all file operations related to reading books from the `datalake` and saving reverse index data into `TSV` files in the `datamart`. The primary functionality includes handling books and organizing their content into well-structured TSV files.

- **loadBooks()**: Reads all book files from the `datalake/books/` directory and returns a list of file paths for processing.
- **readLines(String bookFilePath)**: Reads and returns all lines of text from a specified book file located in the `datalake`.
- **saveWordsToFile(String word, String bookId, int paragraphIndex, int count)**: Handles the creation and update of reverse index files for individual words. Key features:
  - Extracts the book ID from the file path.
  - Organizes words into directories based on the first two letters of the word.
  - Writes data in TSV format with the following fields: `Book_ID`, `Line`, and `Occurrences`.
  - Adds a header to the TSV file if it is newly created.

This class provides robust exception handling for file operations, ensuring smooth integration with the indexing process while maintaining a well-organized structure for the reverse index in the `datamart`.

WordDataHandlerImpl The `WordDataHandlerImpl` class is responsible for handling word lemmatization and cleaning operations. This class implements the `WordDataHandler` interface and utilizes the `StanfordCoreNLP` pipeline for lemmatization. The class is designed to process words by cleaning them of unnecessary characters and converting them into their base form (lemma) for more effective indexing.

- **Lemmatization Pipeline**: The `StanfordCoreNLP` pipeline is initialized once in a static block to ensure efficiency. It uses the `tokenize`, `ssplit`, `pos`, and `lemma` annotators for English language processing.

Key methods of this class include:

- `String lemmAdd(String word)`: This method takes a word, converts it to lowercase, and uses the `StanfordCoreNLP` pipeline to find its lemma. It returns the lemmatized form of the word. If no lemma is found, it returns the original word.
- `String cleanWord(String word)`: This method cleans the word by:
  - Removing any special characters such as apostrophes.
  - Filtering out words that start with an underscore or contain numbers.
  - Keeping only alphabetical characters and removing all non-ASCII characters.

  It returns the cleaned word in lowercase.
- `List<String> cleanAndSplit(String paragraph)`: This method cleans and splits a paragraph into individual words. It removes any special characters, converts the text to lowercase, and returns a list of words.

This class ensures that words are properly cleaned and lemmatized, which enhances the accuracy of the indexing and search processes by standardizing word forms and removing unnecessary characters.

3.4 Query Engine Module

The Query Engine module is designed to load, process, and search data from the indexed books. It supports querying words across multiple books, providing metadata and occurrence details for each search result.

3.4.1 Query Engine Interfaces

IndexLoader The `IndexLoader` interface provides functionality to load indexed data from a specific storage format (e.g., JSON or TSV). It declares:

- `Map<String, Map<Integer, WordData>> loadIndex(String basePath) throws Exception`: Loads the index structure from the given `basePath` and returns it as a nested map, where each word maps to a set of books containing metadata on occurrences and positions.

MetadataLoader The `MetadataLoader` interface facilitates the loading of book metadata. It declares:

- `Map<Integer, Metadata> loadMetadata(String filePath) throws Exception`: Reads metadata from the specified `filePath` and returns a map of book IDs to their respective metadata.

QueryProcessor The `QueryProcessor` interface defines methods for executing word queries on the loaded index. It declares:

- `void run()`: Launches an interactive query session where users can input search terms.
- `Set<Integer> processQuery(String query)`: Processes the given query and returns a set of book IDs where the word appears.

3.4.2 Model Classes

Metadata The `Metadata` class stores essential information about a book, including:

- `id`: The unique identifier of the book.
- `title`: The title of the book.
- `author`: The author of the book.

WordPosition The `WordPosition` class represents the occurrence of a word within a book, storing:

- `line`: The line number where the word appears.
- `occurrences`: The number of times the word occurs in the specified line.

WordData The `WordData` class stores all occurrences of a word within a single book. It maintains:

- `totalOccurrences`: The total count of the word across the book.
- `positions`: A list of `WordPosition` objects representing where the word appears in the book.

3.4.3 Control Classes

CSVMetadataLoader The `CSVMetadataLoader` class implements the `MetadataLoader` interface. It:

- Reads book metadata from a CSV file.
- Parses metadata fields like `title`, `author`, and `id`.
- Returns a map of book IDs to their respective `Metadata` objects.

JSONIndexLoader The `JSONIndexLoader` class implements the `IndexLoader` interface. It:

- Loads index data from a JSON-based directory structure.
- Organizes words into nested maps, mapping each word to the books in which it appears.
- Handles exceptions for missing or invalid files to ensure robust processing.

TSVIndexLoader The `TSVIndexLoader` class also implements the `IndexLoader` interface. It:

- Reads index data stored in TSV files.
- Supports a flat storage structure, avoiding complex directory hierarchies.
- Parses word data, including book occurrences and line-level details, into nested maps.

SimpleQueryProcessor The `SimpleQueryProcessor` class implements the `QueryProcessor` interface. It:

- Accepts user queries to search for words in the indexed data.
- Retrieves and displays metadata, such as the book title and author, alongside word occurrences.
- Displays line-specific positions for the queried word within each book.

Main The `Main` class serves as the entry point for the Query Engine module. It:

- Prompts the user to select between JSON and TSV-based index formats.
- Loads metadata using the `CSVMetadataLoader`.
- Initializes the chosen `IndexLoader` to load the word index.
- Creates an instance of `SimpleQueryProcessor` to handle queries interactively.

The `Main` class ensures a streamlined setup and execution process, allowing users to query indexed data efficiently.

4 Experiments

4.1 Crawler

To evaluate the performance of the `Crawler`, we conducted benchmarking tests to measure the time it takes to download and process books. The tests were performed using varying numbers of books: 5, 10, 50, and 100 books. The results were recorded in milliseconds per operation (ms/op), and a graph was created to visualize the progression of download times.

Results:

- **5 books:** 6653.942 ms/op
- **10 books:** 6892.691 ms/op
- **50 books:** 78253.531 ms/op
- **100 books:** 162529.243 ms/op

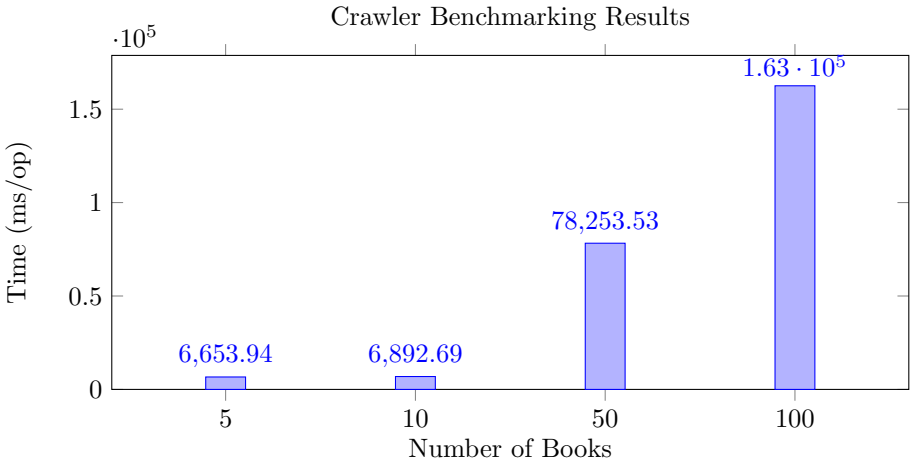

Figure 1: Time taken by the Crawler to download and process books.

4.2 Json Indexer

To benchmark `Indexer I`, we measured the time required to index books across varying workloads: 5, 10, and 50 books. This test demonstrates the system's scalability and efficiency for small- to medium-sized datasets.

Results:

- **5 books:** 11490.596 ms/op

- **10 books:** 55785.952 ms/op
- **50 books:** 328787.845 ms/op

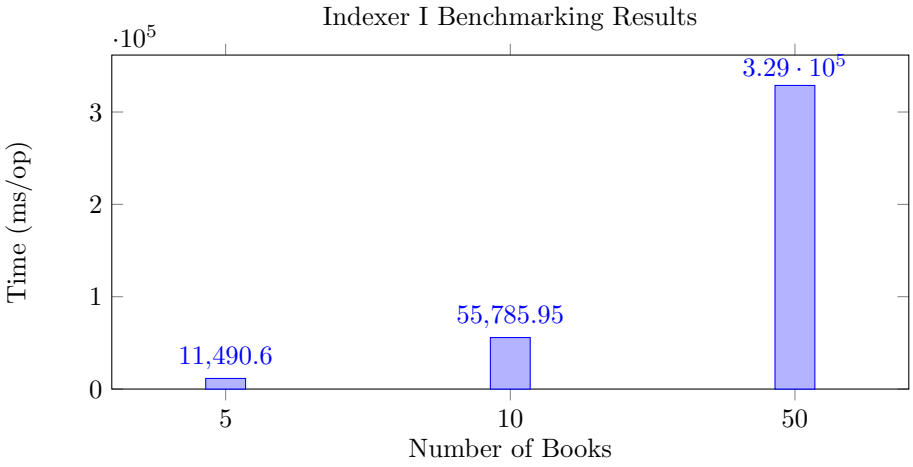

Figure 2: Time taken by Indexer I to index books.

4.3 TSV Indexer

For `Indexer II`, we followed a similar approach, measuring the time required to index books at workloads of 5, 10, and 50 books. The results provide insights into the system's behavior under varying workloads.

Results:

- **5 books:** 6378.885 ms/op
- **10 books:** 173536.166 ms/op
- **50 books:** 436389.777 ms/op

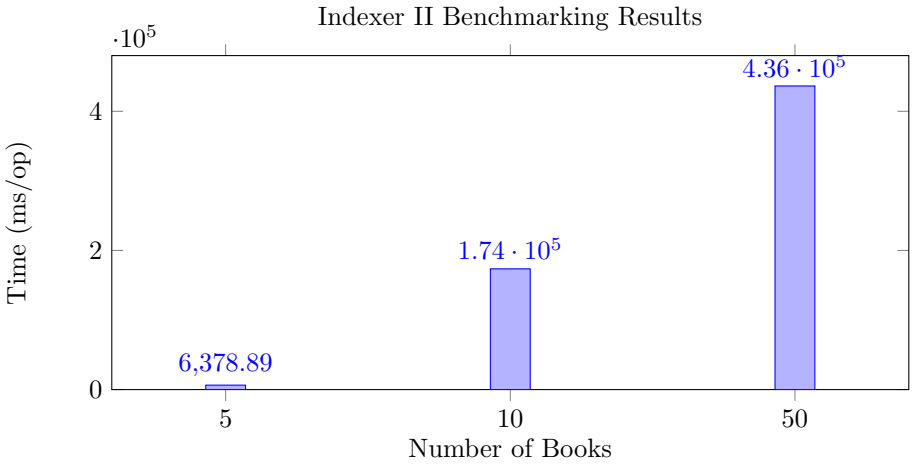

Figure 3: Time taken by Indexer II to index books.

Conclusion:

The experiments reveal notable differences between `Json Indexer` and `TSV Indexer`:

- **Efficiency:** `Json Indexer` performs better for small workloads, with faster indexing times for 5 and 10 books. However, `TSV Indexer` scales more efficiently for larger workloads.
- **Use Case:** For smaller datasets, `Json Indexer` is more practical due to its simplicity and speed. For larger datasets, `TSV Indexer` is preferable as it handles the workload more effectively.

Overall, the choice of indexer depends on the scale of the dataset and the specific requirements of the application.

4.4 Query Engine

To evaluate the performance of the `Query Engine`, benchmarking tests were conducted to measure the time it takes to search for words in indices stored in both JSON and TSV formats. The tests involved searching for 5, 10, 25, and 50 words, with results recorded in milliseconds per operation (ms/op). The aim was to assess how the number of words and the storage format affect query processing times.

Results:

- **JSON Format:**
  - **5 words:** 0.534 ± 0.225 ms/op
  - **10 words:** 1.830 ± 0.204 ms/op

- **25 words:** 2.442 ± 0.786 ms/op
- **50 words:** 5.245 ± 2.021 ms/op

- **TSV Format:**
  - **5 words:** 0.543 ± 0.127 ms/op
  - **10 words:** 1.890 ± 0.663 ms/op
  - **25 words:** 2.440 ± 0.695 ms/op
  - **50 words:** 4.242 ± 0.425 ms/op

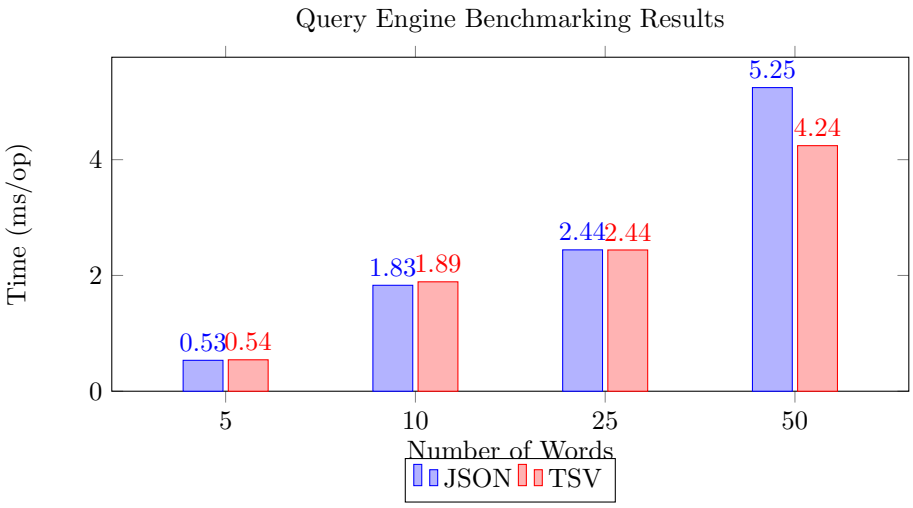

Figure 4: Comparison of query processing times between JSON and TSV indices.

Conclusion:

The results reveal that the `Query Engine` performs efficiently for small-scale queries (5 and 10 words) in both JSON and TSV formats, with processing times under 2 ms/op. As the number of words increases, query times grow linearly, with TSV generally outperforming JSON for larger queries (50 words). This can be attributed to the simpler structure and smaller size of TSV files, which reduce parsing overhead.

5 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.

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.

One notable inefficiency in the current design of the query system lies in its reliance on command-line input for processing search queries. Specifically, the system interprets the word `exit` as a command to terminate the program, thus disregarding any legitimate searches for the word `exit`. This limitation reduces the system's usability and can lead to confusion for end-users.

To address this issue, a more robust approach should be implemented. The ideal solution would involve transitioning from free-form command-line input to a structured menu or switch-based system. In this design, users would be presented with a clear set of options, such as:

- Search for a word.
- Search for multiple consecutive words or those sharing a paragraph.
- Exclude specific words from search results.
- Perform advanced searches that incorporate metadata, such as filtering by author, title, or publication year.
- Exit the program.

These enhancements would eliminate ambiguities in input interpretation, allowing `exit` to be treated either as a valid search query or as a distinct command, depending on the context. Additionally, incorporating features like consecutive word searches and negative word queries would significantly improve the flexibility and power of the system. For instance, users could search for phrases, locate words appearing in the same paragraph, or exclude certain words to refine their results.

Moreover, enabling metadata-based searches alongside basic word queries would provide a richer user experience. This could include features such as searching for all occurrences of a word within books authored by a specific writer or filtering results by publication date.

6 GitHub Repository

The codes this paper refers to can be found in the following GitHub repository:

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