PERFORMANCE OPTIMIZATION AND BENCHMARKING OF MODULAR SEARCH ENGINES: A COMPARATIVE STUDY OF INVERTED INDEX DATA STRUCTURES

Raquel Almeida Quesada, Gabriel Felipe Bernal, Guillermo Cubas Granado, Denys Kavkalo Gumeniuk, Jorge Morales Llerandi and Adonai Ojeda Martín

Universidad de Las Palmas de Gran Canaria

Grado en Ciencia e Ingeniería de Datos, Big Data

November, 2024

ABSTRACT

The development of efficient and scalable search engines remains a critical challenge in Big Data applications. This paper presents the design and implementation of a modular search engine using Java, employing Docker for containerization and Maven for dependency management. Benchmarking with JMH was conducted to compare the performance of different data structures for the inverted index. The results demonstrate the scalability of chosen structures and highlight the tradeoffs in query execution time and memory usage. The findings contribute to optimizing search engines for large-scale applications, showcasing the importance of benchmarking in system design.

1. INTRODUCTION

Search engines are pivotal in the era of Big Data, enabling fast and accurate retrieval of information from massive datasets. Optimizing their efficiency involves addressing computational challenges like indexing and query processing. Benchmarking plays a key role in evaluating and improving these systems by providing measurable insights into performance.

While several studies have explored data structures for indexing, the focus of this work is on practical implementation and evaluation in a real-world modular search engine. The use of Java ensures cross-platform compatibility, and Docker enhances deployment flexibility.

This paper details the implementation of a search engine, emphasizing benchmarking results obtained through Java Microbenchmarking Harness (JMH). The final section discusses the benefits of the selected approach and its relevance for real-world applications.

2. PROBLEM STATEMENT

Efficient indexing and query processing are essential for modern search engines. However, finding the right balance between speed, memory usage, and scalability is challenging. Existing approaches often overlook modular design and real-world benchmarking, focusing instead on theoretical improvements. This work aims to bridge that gap by comparing the performance of two data structures for inverted indexing in a modular search engine framework.

Search engines must efficiently index and retrieve data from large-scale datasets while balancing speed, memory usage, and scalability. Achieving this balance requires selecting and implementing suitable data structures for inverted indexing. While HashMaps and B-Trees are widely used, their performance characteristics can vary significantly depending on the use case.

This project seeks to address the following research questions:

- How do HashMap and B-Tree data structures compare in terms of memory usage, indexing speed, and query latency in a search engine context?
- How can a modular approach enhance the maintainability and scalability of search engines?

The collaborative nature of this project allowed the team to explore these questions by leveraging diverse expertise in programming, benchmarking, and system design.

3. METHODOLOGY

The project was structured into three modular components:

- **Crawler:** Collects web data for indexing. Designed to handle scalability challenges by processing data batches in parallel.
- **Indexer:** Implements two data structures for the inverted index—HashMaps and B-Trees. Supports flexible indexing strategies to accommodate future extensions.
- **Query Engine:** Processes queries to test index performance.

Each module was implemented in Java using Maven for dependency management. The project also utilized Docker to ensure consistency across development and deployment environments.

Tools and Technologies

- **Programming Language:** Java
- **Build and Dependency Management:** Maven was used to manage project dependencies and streamline development.
- **Containerization:** Docker ensured consistent environments across development and deployment.
- **Benchmarking:** JMH was employed to evaluate execution time, memory usage, and throughput.

The team worked collaboratively, with roles distributed to ensure efficient module development, benchmarking, and analysis.

4. CRAWLER MODULE

Purpose and Design Goals

The main objectives of the Crawler module are:

- Download eBooks in various formats (txt, html, epub, and mobi).
- Store the downloads in a folder structure based on the current date.
- Resume downloads seamlessly by identifying the last downloaded book.
- Provide utility functions to manage and analyze downloaded files.

Class Structure

Controller Class

This serves as the entry point for the application. It creates an instance of GutenbergCrawler and initiates the crawling process.

- **Method:** `main(String[] args)`: Calls the `crawlBooks` method in GutenbergCrawler to download 100 books.

FileManager Class

Responsible for managing the filesystem-related operations.

- **Attributes**
  - `DOWNLOAD FOLDER`: The root directory for storing downloaded files.
- **Methods**
  - `getDate()`: Returns the current date in `ddMMyyyy` format.
  - `createFolder(String folderPath)`: Creates a folder if it does not exist.
  - `getFoldersInPath()`: Lists all subdirectories in the `DOWNLOAD FOLDER`.
  - `getLatestNonEmptyFolder()`: Finds the latest folder that contains files.
  - `getFileWithLargestBookID(String folderName)`: Identifies the file with the highest numeric book ID in a folder.

GutenbergCrawler Class

Encapsulates the logic for orchestrating the crawling process.

- **Attributes**
  - `DOWNLOAD FOLDER`: Directory for storing downloaded files.
  - `webID`: Prefix for filenames to ensure uniqueness.
- **Methods**
  - `crawlBooks(int numBooks)`: Determines the starting book ID based on the latest downloads and invokes the `downloadBook` method in GutenbergDownloader for the specified number of books.

GutenbergDownloader Class

Handles the HTTP requests to download books from Project Gutenberg.

- **Attributes**

    - `BASE URL`: Base URL for accessing Project Gutenberg's books.

- **Methods**
    - `downloadBook(String bookId, String targetFolder, String webID)`: Attempts to download a book in multiple formats. Returns true if successful, false otherwise.

Innovations and Challenges

Innovations

- Dynamic File Naming: Ensures file uniqueness by combining a session-specific date and a prefix (webID).
- Continuity in Operations: Analyzes the metadata of previously downloaded files to avoid redundancy and resume from the correct point.

Challenges

- Handling non-sequential or non-standard book IDs.
- Managing variability in available formats for different books.
- Handling filesystem and network-related exceptions gracefully.

Future Enhancements

Potential Improvements

- Extend format support to include other media types or metadata extraction.
- Introduce parallelism for faster downloads.
- Integrate a database to store metadata and improve query efficiency.
- Refactor the `crawlBooks` method to include error recovery and adaptive retries.

Conclusion

The Crawler module demonstrates an efficient and extensible approach to web crawling. By breaking the functionality into modular components, the design ensures maintainability and scalability.

5. CLEANER MODULE

Purpose and Design Goals

The primary objectives of the Cleaner module are:

1 Process and clean eBook files, filtering out stopwords and irrelevant data.
2 Extract and standardize key metadata such as title, author, date, and language.
3 Keep track of the last processed file for seamless resumability.
4 Create an organized representation of eBooks, including their metadata and cleaned content.

Class Structure

Controller Class

Serves as the entry point for the module, orchestrating the cleaning process.

- **Method:** `main(String[] args)`: Initializes the Cleaner class, specifies the root directory of eBook files, and invokes the `processAllBooks` method to clean and process all files.

Cleaner Class

- **Attributes:**
    - `textCleaner`: An instance of the `TextCleaner` class initialized with stopwords.
- **Methods:**
    - `processBook(File file)`: Processes an individual eBook file by:
        * Extracting metadata using `MetadataExtractor`.
        * Cleaning text content with `TextCleaner`.
        * Returning a structured `Book` object with metadata and processed content.
    - `processAllBooks(String rootPath)`: Iterates through all eBooks in a directory hierarchy, processes each file, and keeps track of progress using `LastProcessedTracker`.

TextCleaner Class

- **Attributes:**
    - `stopwords`: A set of common words to exclude, loaded via `StopwordsLoader`.
- **Methods:**
    - `cleanText(String text)`: Processes the raw content to produce a list of meaningful words.

MetadataExtractor Class

- **Methods:**

  - `extractMetadata(String content)`: Returns a map of metadata fields such as title, author, date, and language.
  - `extractField(String regex, String content, String defaultValue)`: Helper method to retrieve specific metadata fields with fallback values.

LastProcessedTracker Class

- **Attributes:**
  - `lastProcessedFilePath`: Path to the tracker file recording the last processed file.
- **Methods:**
  - `getLastProcessed()`: Reads the last processed file from the tracker.
  - `updateLastProcessed(String fileName)`: Updates the tracker file with the latest processed file name.

StopwordsLoader Class

- **Methods:**
  - `loadStopwords()`: Returns a set of stopwords for text cleaning.

Book Class

- **Attributes:**
  - `title`, `author`, `date`, `language`, `credits`, `ebookNumber`: Metadata fields.
  - `words`: List of sanitized, meaningful words.
  - `fullContent`: Raw content of the eBook post-cleaning.
- **Constructor:** `Book(String title, String author, String date, String language, String credits, String ebookNumber, List<String> words, String fullContent)`: Initializes the `Book` object with extracted data.

Innovations and Challenges

Innovations:

- Dynamic Metadata Extraction: Extracts key fields using customizable patterns.
- Resumability: Tracks and resumes processing from the last successfully cleaned file.
- Stopword Filtering: Provides meaningful word lists by removing irrelevant terms.

Challenges:

- Handling variability in metadata formats across different eBooks.
- Ensuring efficient processing of large directories with nested structures.
- Managing edge cases in content formatting and incomplete metadata.

Future Enhancements

Potential Improvements:

- Extend metadata extraction to include genre, summary, or additional descriptive fields.
- Implement parallel processing for increased throughput.
- Integrate machine learning models for advanced text analysis.
- Refactor text cleaning to support multilingual eBooks with dynamic stopword lists.

Conclusion

The Cleaner module exemplifies a comprehensive approach to eBook data processing. Its modular structure, robust text processing, and metadata extraction capabilities ensure scalability and reliability, making it an essential component of the Indexer system.

6. INDEXER MODULE

Purpose

The **Indexer** module is designed to create a structured, searchable, and efficient system for organizing and retrieving book data. By indexing both the metadata and content of books, the module enables fast searches for keywords and provides a foundation for advanced data analysis and querying. The primary purpose of the module is to transform unstructured book data into organized indexes that can be persisted in various formats (CSV or DataMart) for easy access and further processing.

Class Details

Indexer Class

- **Description**: Coordinates the indexing process, using `BookIndexer`, `CSVWriter`, and `DataMartWriter` to build indexes and store them in various formats.
- **Attributes**:
  - `bookIndexer`: Instance of `BookIndexer` to handle the indexing process.
  - `csvWriter`: Instance of `CSVWriter` to store indexes in CSV format.
  - `dataMartWriter`: Instance of `DataMartWriter` to store indexes in a DataMart.
- **Methods**:
  - `buildIndexes(books: List<Book>)`: Creates indexes using the `BookIndexer` class.
  - `indexBooks(books: List<Book>, outputType: String)`: Controls the indexing flow and persists data in the specified format (`csv` or `datamart`).

BookIndexer Class

- **Description**: Indexes books using two structures: a `HashMap` for quick associations and a `Trie` for efficient searches.
- **Attributes**:
  - `hashMapIndexer`: Instance of `HashMapIndexer` to manage a `HashMap`-based index.
  - `trie`: Instance of `Trie` to manage a prefix-based index.
- **Methods**:
  - `indexBook(book: Book)`: Adds all words from a book to the `HashMap` and the `Trie`.
  - `getHashMapIndexer()`: Returns the `HashMapIndexer` instance.
  - `getTrie()`: Returns the `Trie` instance.

HashMapIndexer Class

- **Description**: Implements a `HashMap`-based index where words are associated with a set of book identifiers.
- **Attributes**:
  - `hashMapIndex`: Map associating words (`String`) with sets of book identifiers (`Set<String>`).
- **Methods**:
  - `addWord(word: String, ebookNumber: String)`: Adds a word to the index along with its book identifier.
  - `search(word: String)`: Returns the set of book identifiers associated with a word.
  - `getIndex()`: Returns the complete index map.

Trie Class

- **Description**: Represents a prefix-based index, enabling fast word searches.
- **Attributes**:
  - `root`: Root node of the Trie.
- **Methods**:
  - `insert(word: String, ebookNumber: String)`: Adds a word to the Trie along with its book identifier.
  - `search(word: String)`: Returns the set of book identifiers associated with a word.

CSVWriter Class

- **Description**: Manages the persistence of indexes in CSV format.
- **Methods**:
  - `saveMetadataToCSV(books: Iterable<Book>)`: Saves book metadata to a CSV file.
  - `saveContentToCSV(wordToEbookNums: Map<String, Set<String»)`: Saves the word index to a CSV file.

DataMartWriter Class

- **Description**: Stores indexes and metadata in a hierarchical directory structure (DataMart).
- **Methods**:
  - `saveContentToDataMart(wordToEbookNumbers Map<String, Set<String»)`: Stores the word index in a Trie-like structure within directories.
  - `saveMetadataToDataMart(books: Iterable<Book>)`: Stores book metadata as JSON files within separate directories.

TrieNode Class

- **Description**: Internal node of the `Trie` structure.
- **Attributes**:
  - `children`: Map associating characters (`Character`) with child nodes.
  - `ebookNumbers`: Set of book identifiers associated with the node.

Main Class

- **Description**: Entry point of the program. Allows the user to choose the output type (`CSV` or `DataMart`) and processes the books from the `datalake` directory.

Future Enhancements

- **Advanced Query Capabilities**: Integrate support for more complex queries, such as phrase searches, wildcard searches, and fuzzy matching.
- **Parallel Processing**: Enable multithreading or distributed processing for indexing and storage.
- **Real-Time Updates**: Develop mechanisms for real-time updates to indexes as new data arrives.
- **Improved Compression Techniques**: Implement compression algorithms to reduce disk usage.
- **Support for Synonyms and Semantics**: Include synonyms, stemming, and lemmatization in the indexing process.
- **Visualization Tools**: Add graphical tools for visualizing index data.
- **Mobile and API Accessibility**: Develop a REST API or mobile interface for remote access.
- **Versioning and History**: Introduce versioning for indexes to enable rollbacks and comparisons.

Conclusion

The **Indexer** module is a robust system designed for efficient storage and retrieval of book-related metadata and content. Its modular approach, combining `HashMaps` and `Tries`, supports a range of search functionalities. The dual output formats, `CSV` and a trie-like `DataMart`, enhance its usability for different user needs. Challenges such as optimization and data consistency are addressed through its extensible design.

7. QUERY ENGINE MODULE

Purpose and Design Goals

The primary objectives of the Query Engine are:

1 Support multiple data sources, including CSV files and Datamart structures.
2 Process user queries and tokenize them for search.
3 Integrate a GUI for user interaction, enabling query submission and displaying results.
4 Ensure modularity and scalability for future data sources and query types.

Class Structure

CSVDataSource Class

Implements the `DataSource` interface, loading an inverted index from a CSV file.

Attributes:

- `contentFilePath`: Path to the CSV file containing the index data.

Methods:

- `loadIndex()`: Reads the CSV file and loads an inverted index mapping words to eBook identifiers.

DatamartDataSource Class

Implements the `DataSource` interface, enabling searches directly within a structured folder hierarchy.

Methods:

- `loadIndex()`: Placeholder for loading a datamart-based index.
- `searchWord(String word)`: Searches for a specific word in the datamart folder hierarchy.

InvertedIndex Class

Provides the core functionality for querying the search engine. It can search either in memory (e.g., CSV-based indices) or directly from the datamart.

Attributes:

- `index`: A map representing the inverted index when loaded from memory.
- `datamartDataSource`: Reference to the `DatamartDataSource` for direct datamart queries.

Methods:

- `search(String term)`: Performs a search for a term in the appropriate data source.

QueryTokenizer Class

Processes user queries, breaking them into individual tokens for search.

Methods:

- `tokenize(Query query)`: Tokenizes the input query into a list of lowercased words.

SearchEngineGUI Class

Provides a graphical user interface for interacting with the search engine.

Attributes:

- `dataSourceSelector`: Dropdown menu to select the data source.
- `queryField`: Text field for user queries.
- `resultsArea`: Text area to display search results.
- `invertedIndex`: The search engine instance used for executing queries.

Methods:

- `configureEvents()`: Sets up event handlers for loading data sources and executing queries.

Innovations and Challenges

Innovations:

- Multi-Source Support: Integrates CSV and Datamart-based data sources seamlessly.
- GUI Integration: Provides an intuitive interface for user interaction.
- Tokenization: Ensures robust query handling through efficient token processing.

Challenges:

- Managing large datasets in memory while maintaining responsiveness.
- Ensuring compatibility between different data source formats (CSV vs. Datamart).
- Handling edge cases in user input, such as empty queries or invalid characters.

Future Enhancements

- Extend support for additional data sources such as XML or JSON files.
- Enhance the GUI with real-time suggestions and query previews.
- Implement parallel processing for faster query execution.
- Add advanced query handling, including Boolean queries and wildcard searches.

Conclusion

The Query Engine module is a pivotal component of the search engine, enabling efficient query processing and user interaction. Its modular design, multi-source support, and robust tokenization capabilities ensure scalability and reliability, making it a key contributor to the overall system's functionality.

8. BENCHMARKING

8.1. Python - Performance Analysis of Each Step

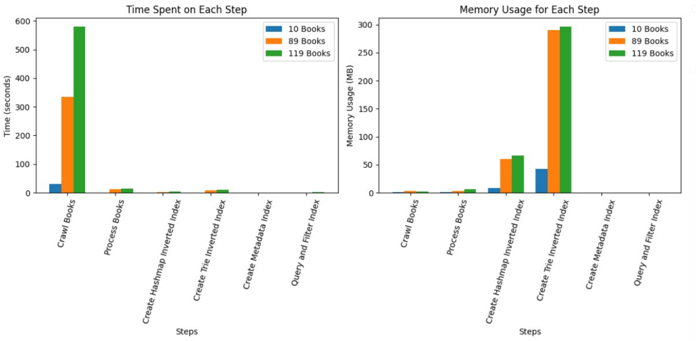


Figure 1. Time and Memory Usage for Each Step

The performance analysis illustrated in Figure 1 consists of two graphs. The first graph shows the time spent on each step of the process for datasets of different sizes (10, 89, and 119 books). The second graph displays the memory usage for each of these steps under the same conditions.

Interpretation of Time Spent on Each Step The left graph reveals that the step of crawling books takes significantly more time as the dataset size increases. For 10 books, the time is minimal, while for 119 books, the crawling process takes around 600 seconds. This indicates that the crawling process scales linearly or worse with the number of books due to network latency and data processing.

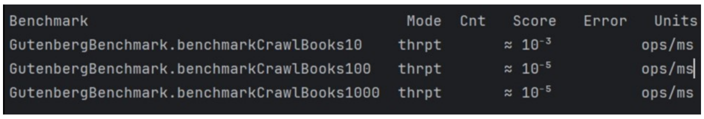
The subsequent steps (processing books, creating HashMap inverted index, Trie inverted index, and creating metadata index) show minimal time increases regardless of the dataset size. This implies that the indexing and metadata creation processes are highly optimized and scale well with the number of books.

Interpretation of Memory Usage The right graph demonstrates that memory usage varies greatly when creating the Trie inverted index, which requires significantly more memory compared to other steps. For all dataset sizes, the memory usage for this step approaches or exceeds 250 MB. This suggests that the Trie structure, while efficient for search, has a higher memory footprint.

In contrast, creating a HashMap inverted index and processing books show moderate memory consumption, which increases slightly with the number of books. The steps involving crawling and creating metadata have minimal memory usage.

Overall Analysis The analysis shows that while the crawling step is time-intensive, the subsequent steps are more memory-bound, especially when creating the Trie inverted index. This balance between time and memory usage provides insights into potential areas for optimization. For instance, parallelizing the crawling step or optimizing the Trie structure could reduce time and memory usage, respectively.

8.2. Java - Benchmarks for Crawler Module



```
Benchmark                                   Mode   Cnt    Score    Error    Units
GutenbergBenchmark.benchmarkCrawlBooks10    thrpt        ≈ 10^-3            ops/ms
GutenbergBenchmark.benchmarkCrawlBooks100   thrpt        ≈ 10^-5            ops/ms
GutenbergBenchmark.benchmarkCrawlBooks1000  thrpt        ≈ 10^-5            ops/ms
```

Figure 2. Benchmark results for the crawler module.

As shown in the **Figure 2**, the benchmark results show a significant performance degradation when downloading more books with the `crawlBooks` function. For smaller workloads (e.g., downloading 10 books), the throughput was around 10^{-3} ops/ms, indicating efficient processing. However, as the number of books increased to 100 and 1000, the throughput dropped to 10^{-5} ops/ms, suggesting a performance bottleneck. This indicates limitations related to network bandwidth, disk I/O, and inefficiencies in the download process.

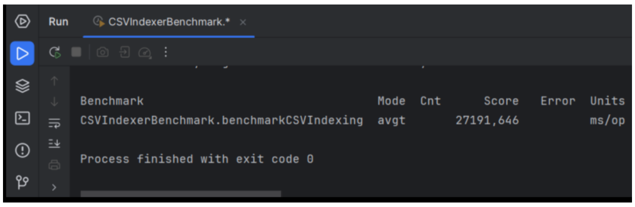
Opportunities for optimization include enhancing concurrency and parallelism, optimizing I/O operations, and addressing network constraints to improve scalability and throughput.

8.3. Java - Benchmarks for Indexer Module

As shown in the **Figure 3** and **Figure 4**, the benchmark results for the indexer module show a noticeable disparity in performance between the two data structures used for indexing. For the CSV-based indexing, the benchmark measured an average time of 27191.646 ms per operation. In contrast, the DataMart indexing performed significantly slower, with an average time of 653799.752 ms per operation.

This substantial performance difference suggests that the DataMart structure may introduce higher overheads or inefficiencies compared to the CSV structure. It is likely that the DataMart structure requires more complex processing, resulting in increased computational costs and slower indexing speeds.

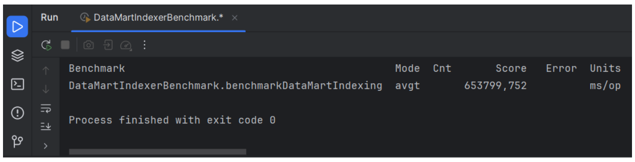
Opportunities for optimization include investigating the underlying data structures to identify areas for performance improvement, optimizing the indexing algorithm, and evaluating the possibility of parallelizing the indexing process to reduce the overall time required for larger datasets. Additionally, exploring more efficient I/O operations or reducing memory usage could contribute to improving the performance of the indexing module.



```
Benchmark                                  Mode  Cnt      Score   Error  Units
CSVIndexerBenchmark.benchmarkCSVIndexing   avgt        27191,646         ms/op

Process finished with exit code 0
```

Figure 3. Benchmark results for CSV-based indexing



```
Benchmark                                              Mode  Cnt       Score   Error  Units
DataMartIndexerBenchmark.benchmarkDataMartIndexing     avgt        653799,752         ms/op

Process finished with exit code 0
```

Figure 4. Benchmark results for DataMart-based indexing

Conclusion

This project presents an efficient indexing system that leverages HashMap and Trie data structures to handle large datasets, demonstrating good scalability for smaller volumes. However, performance degradation in the crawling process becomes apparent as the dataset size grows, primarily due to network and processing constraints. Indexing operations, especially with the HashMap and Trie structures, show minimal increases in processing time, though the Trie structure's higher memory consumption suggests potential areas for optimization.

Benchmark results reveal a notable performance gap between CSV-based indexing and DataMart indexing, with the CSV approach outperforming in speed. This difference underscores the need for further optimization within the DataMart structure. Additionally, parallel processing and more efficient I/O operations could help alleviate time and memory bottlenecks, as evidenced in both the Java and Python benchmarks.

In the Python implementation, memory usage analysis indicates that Trie-based indexing consumes substantial memory, impacting overall system performance. Optimization techniques, such as employing more memory-efficient data structures or compression methods, could help mitigate these issues. Implementing parallelism in the crawling step also presents a promising improvement pathway.

Overall, this system provides a flexible solution with room for future enhancements in performance optimization, query handling, and scalability. Future work should focus on refining the DataMart structure, improving memory efficiency, and enabling more complex query capabilities.

References

- GitHub Repository: Stage 1 <https://github.com/raquelaq/Benchmark>
- GitHub Repository: Stage 2 <https://github.com/Gabriel-BP/SearchEngine_Stage2>
- Project Gutenberg: <https://www.gutenberg.org/>
- Python Documentation: <https://docs.python.org/3/>
- JSON Documentation: <https://www.json.org/json-en.html>
- Stack Overflow: <https://stackoverflow.com/>
- ChatGPT: <https://chatgpt.com/>
- Docker Documentation: <https://docs.docker.com/>
- JMH (Java Microbenchmarking Harness) Guide: <https://openjdk.org/projects/code-tools/jmh/>
- Baeldung: <https://www.baeldung.com/>