TASK 1. Inverted Index Datamart Structure Experiments in Python

Terabyte Titans:
Diego Jesús Torrejón Cabrera
Daniel Talavera Hernández
Samuel Déniz Santana
David García Vera

October 2024

Source code in this GitHub repository.

1 Abstract

This project investigates the development of an efficient search engine leveraging an inverted index structure to optimize document retrieval from a curated dataset of books. The primary challenge is to enhance query performance while managing large volumes of textual data collected by a crawler that periodically downloads books. To address this, selecting suitable data structures for indexing is crucial in order to maintain efficient search speeds and ensure the system can scale as the dataset expands. The experimentation involves implementing two different data structures within an indexer to create inverted indexes for the downloaded documents. One approach uses a tree-based data structure, which organizes the index hierarchically, while the other employs a single JSON file that consolidates all terms. These indexes, along with metadata about the documents, are exported to a data mart, providing a structured format for efficient access. A query engine is then used to process user searches by referencing both the metadata and the two data structures, serving results based on the indexed data. The results reveal trade-offs between the two data structures. The tree-based structure enables faster query execution by allowing quick access to relevant documents but is slower during the export phase due to the complexity of organizing the data. Conversely, the single JSON approach speeds up data export since all terms are consolidated in one file, but it slows down query handling because the engine must read through the entire file to find relevant terms. These findings highlight the significant impact of data structure selection on both indexing and querying tasks. Overall, the insights gained from this project emphasize the importance of choosing the appropriate data structure based on the specific requirements for indexing speed and query efficiency. This

work lays a foundation for future enhancements, such as incorporating parallel processing techniques and experimenting with additional data structures, to further improve search efficiency and user experience.

2 Introduction

2.1 Context

The development of search engines involves various components and techniques, with indexing structures playing a central role. In this context, inverted indexes are widely used, which allow efficient text document retrieval by associating words with the documents that contain them. This structure is crucial for search engines and databases like Google, Elasticsearch, MongoDB, and Apache Solr. Evaluating the efficiency of data structures used in inverted indexes is essential, particularly in terms of their scalability and search performance. This project aims to implement and benchmark different data structures for inverted indexes to explore their efficiency in search tasks.

2.2 Review of Related Work

In our previous work on matrix multiplication, we benchmarked different programming languages by comparing execution times as the matrix size scaled. This benchmarking approach is relevant to the current project on inverted indexes, as both involve evaluating performance across different data structures based on execution time and scalability. In the matrix multiplication study, performance metrics revealed how execution times increased with larger matrices, which informed the choice of algorithms and optimizations.

Similarly, for the inverted index project, we aim to compare the performance of different data structures by evaluating how they scale with the dataset size. Just as matrix size influenced execution time in the previous work, here, the number of documents and terms in the search engine will affect indexing and query speeds. This parallel allows us to apply insights from our matrix multiplication benchmarks to optimize data structures for the inverted index, focusing on scalability and execution efficiency. Thus, the techniques for evaluating performance in matrix multiplication serve as a foundation for benchmarking the scalability and efficiency of inverted indexes.

2.3 Objectives of this project

This project focuses on developing a search engine using an inverted index structure, with a specific goal of benchmarking at least two different data structures for the inverted index to evaluate their efficiency and scalability. The project includes a crawler to periodically download documents from sources like Project Gutenberg, an indexer to update the inverted index and metadata store, and a minimal query engine to search through the indexed documents. The added value lies in the benchmarking process, which aims to identify scalable and efficient data structures suitable for search engines dealing with large datasets.

3 Problem statement

The task of creating an efficient and scalable search engine poses significant challenges due to the vast amount of text data involved and the need for rapid query responses. Traditional search mechanisms are often inadequate when handling massive collections of documents, leading to inefficiencies in data retrieval and processing. To address this, inverted index structures, widely adopted by systems such as Google and Apache Solr, are leveraged. However, building and optimizing such an index, especially when supporting both record-level and word-level indexing, introduces complexities in data structure management, query optimization, and scalability.

This project explores the design and implementation of a search engine that uses different data structures for inverted indexing, evaluates their performance, and ensures scalability for large datasets. The objective is to benchmark these structures in terms of execution speed, while providing a query engine capable of rapid searches across large-scale document repositories. The search engine must efficiently handle continuous updates via a crawler, process documents for indexing, and support user queries through an API.

4 Solution

To tackle the problem of building an efficient and scalable search engine, we propose a solution based on two distinct data structures for inverted indexing: a unique JSON index and a tree-based structure. These data structures will be implemented and evaluated for their performance in handling large-scale document collections. The workflow for the solution involves three main components: Crawler, Indexer, and Query Engine.

4.1 Crawler

The provided crawler is designed to automate the downloading of books from Project Gutenberg, using web scraping techniques to systematically collect book links and download their text files. It navigates through different categories of books, retrieves all book page links, and then checks if a downloadable text version is available for each book. To enhance efficiency, the crawler uses a 'ThreadPoolExecutor' to download multiple books simultaneously, while also managing server load by implementing a delay between downloading blocks of books. The downloaded files are stored in a structured local directory, with formatted file names for easy organization.

The approach involves dynamically navigating through the website's paginated listings, concurrent downloading to improve speed, and error handling to ensure robustness in case of network issues. Additionally, the crawler introduces rate limiting to avoid overwhelming the server, making it suitable for collecting large datasets of books. This methodology can be reproduced to create a dataset for text analysis or other research applications, demonstrating an effective strategy for large-scale web scraping.

4.1.1 Configuration and Execution

To configure the crawler, set the BASEURL to https://www.gutenberg.org/ and specify the repository documents path where the downloaded books will be saved. Ensure that this directory exists or let the program create it automatically. The bookshelf range is defined by setting bookshelfnum (initial category) and maxbookshelfnum (final category). The program iterates through these bookshelf numbers, constructing URLs for different categories of books.

To execute the crawler, run the script from the command line or an integrated development environment (IDE). The script will start downloading books in blocks of 10, pausing for 30 minutes between blocks to manage server load. The crawler will log its progress and any errors encountered in a log file named crawler.log, providing information on the number of books downloaded and any issues faced.

4.2 Indexer

The Indexer module takes care of building an efficient inverted index for the stored documents, as well as extracting and storing the relevant metadata of the books in a CSV file.

4.3 Inverted Index Construction

We use two main approaches for the creation of inverted indexes:

4.3.1 Tree Indexer

The Tree Indexer aims to build an inverted index for the text files by breaking down the documents into individual words and tracking their positions and frequencies within each document. It utilizes multiple language models from SpaCy to filter out stop words in different languages. The inverted index is organized into a tree structure, categorizing words alphabetically by their first letter. The script exports this tree structure into a set of JSON files, with each file corresponding to a specific letter.

4.3.2 Unique JSON Indexer

The JSON Indexer is similar to the Tree Indexer, but it exports the entire inverted index as a single JSON file. It processes the text files to build an inverted index that maps words to their positions and frequencies within documents. The inverted index is cleaned by removing stop words and formatting each word.

4.3.3 Metadata Extraction

This script is designed to extract metadata from text files in a specified directory and export the information into a CSV file. It first scans through the directory to identify text files, then uses regular expressions to extract metadata fields such as title, author, release date, and language from the content of each file. The script also extracts the actual content of the book, which starts after a delimiter (usually "*** START OF... ***"). The extracted metadata and content are stored in a structured format and saved as a CSV file.

In this way, the Indexer not only facilitates efficient access to words within documents, but also enables more complex queries through detailed metadata, improving the scalability and performance of the search system as a whole.

4.4 Query Engine

The Query Engine is the module in charge of performing queries on the indexed data. Two different approaches were implemented to handle queries efficiently, depending on the data structure used for the inverted index. These approaches are: Tree Data Structure Query Engine and Unique JSON Query Engine.

4.4.1 Tree Data Structure Query Engine

This approach organises the inverted index data in JSON files, where words are stored according to the initial letter. This method takes advantage of a hierarchical structure where each letter of the alphabet has its own file. The query process is performed in the following steps:

Inverted Index Load: Given a search term, the search engine loads the JSON files corresponding to the initial letters of the words in the query. This allows for optimised data loading and reduced memory usage.

Inverted Index Query: Once the index is loaded, the query words are searched against the document set. If all query words are present in a document, a result is returned with the frequencies, positions and context of each word in that document.

Metadata Search: The engine also supports queries based on metadata such as title, author, language or release date. The metadata is stored in a CSV file that is loaded at the start of the search engine execution. This process allows queries to be performed efficiently for both content and additional information relevant to each book.

4.4.2 Unique JSON Query Engine

In this approach, the entire reverse index is stored in a single JSON file. Although this method is less scalable than the previous one for large volumes of data, its simplicity makes it more efficient in terms of queries for smaller datasets. The process is similar to the Tree Data Structure:

Inverted Index Load: The entire JSON file containing the index is loaded into memory at the start of the process. This allows queries to be performed directly on the loaded data structure, avoiding additional accesses to the file system.

Inverted Index Query: For each word in the query, the inverted index is searched and it is determined which documents contain all the words in the query. The results include the positions, frequencies and context of each word in the document.

Metadata Search: Similar to the Tree Data Structure approach, the engine supports metadata-based queries, using the same CSV file.

Both search engines offer a REST query interface, where HTTP GET requests allow users to perform both content and metadata queries. Responses are returned in JSON format, allowing easy integration with other applications or user interfaces.

The implementation of both approaches allows for performance and scalability testing, providing key metrics on the efficiency of each data structure in terms of query time and resource usage.

4.5 Benchmarking and Performance Measurement

The benchmark is designed to perform automated tests on two search engines implemented in Flask applications: one that uses a single JSON inverted index and one that uses a tree structure. Using ‘pytest‘ and the ‘pytest-benchmark‘ plugin, the code sets up test clients for each engine, allowing HTTP requests to be sent and their performance evaluated. Tests are created to measure the time in miliseconds it takes to export inverted indexes to JSON files, as well as search tests to ensure that queries (such as ‘African’ and ‘History of Africa’) are processed correctly and return successful responses. In essence, the code allows the efficiency of both search engines in index construction and query handling to be analysed and compared.

The experiments will be run on a MacBook Pro with the following specifications: Apple Silicon M3 Pro chip, which includes 11 CPU cores and 18 GB of unified memory.

4.6 Evaluation Criteria

4.6.1 Speed

The indexing and querying speeds will be the primary evaluation criteria. The speed-up factor will be calculated by comparing the time taken by each structure.

4.6.2 Scalability

The ability of each data structure to handle increasingly large datasets will be measured.

4.6.3 Efficiency

The efficiency of each data structure in terms of time complexity will be analyzed.

The experiments will provide insights into the trade-offs between using a single JSON file versus a tree-structured approach for the inverted index in terms of both performance and scalability. This will allow us to recommend the optimal structure for large-scale search engines.

5 Experiments

To evaluate the performance of the indexing and querying systems, we conducted two main experiments, focusing on the time taken by the Tree Data Structure Inverted Index and the Unique JSON Inverted Index to both index documents and respond to search queries. The purpose of these experiments was to compare the efficiency and scalability of the two different data structures in the indexing process and their performance during query execution.

5.1 Experiment 1: Indexing Time Comparison

5.1.1 Objective

The first experiment aimed to measure the time it takes for each indexing structure to write the inverted index to the Datamarts. We wanted to evaluate how well each structure handles the document indexing process, specifically focusing on the time it takes to export the generated inverted index to the Datamarts.

5.1.2 Setup

We used the Gutenberg Project dataset with 10,000 documents and we store 10 books each 30 minutes in the datalake. Two indexing methods were tested: the Tree Data Structure Inverted Index and the Unique JSON Inverted Index. Each method was executed five times, and the average time taken to export the index to the Datamarts was recorded. All tests were conducted on the same machine.

5.1.3 Results (Table 1)

Indexing Method	**Average Export Time (miliseconds)**
Tree Data Structure Inverted Index	559.9923
Unique JSON Inverted Index	497.8255

Table 1: Comparison of export times between Tree Data Structure and Unique JSON Inverted Index

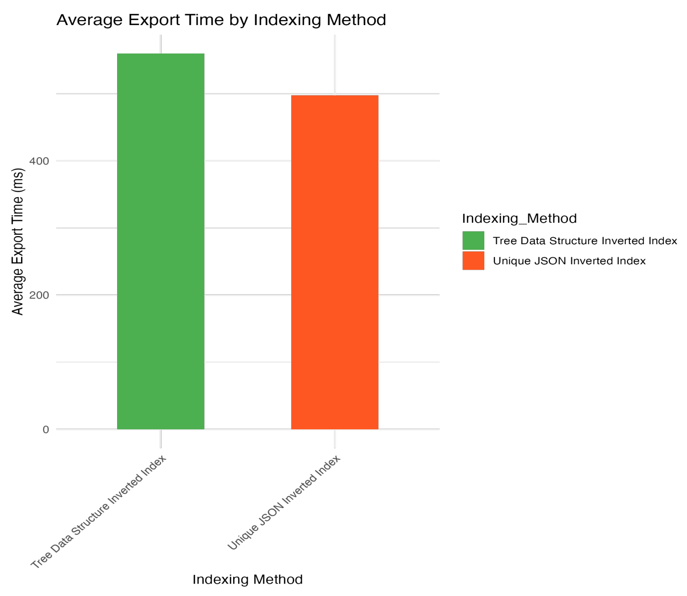

Figure 1: Graph that shows how the execution time evolves depending on the size of the matrices

As shown in Table 1, the Unique JSON Inverted Index outperforms the Tree Data Structure Inverted Index in terms of export time. The Unique JSON Inverted Index takes approximately 497.83 miliseconds on average to export the inverted index to the Datamarts, whereas the Tree Data Structure takes longer, with an average export time of 559.99 miliseconds. This indicates that the JSON-based structure is more efficient in writing the index to disk, possibly due to its simpler structure.

5.2 Experiment 2: Query Execution Time Comparison

5.2.1 Objective

The second experiment aimed to measure the time taken for both the Tree Data Structure Inverted Index and the Unique JSON Inverted Index to execute search queries. The objective was to evaluate the efficiency of each structure in retrieving relevant documents using the same set of search terms.

5.2.2 Setup

Three different search queries were used for both indexing methods: "African," "History of Africa," and "African people were slaves." Both the Tree Data Structure and Unique JSON indexes were queried, and the response times were recorded. Each query was run five times for both methods, and the average query response time was calculated.

5.2.3 Results (Table 2)

Query	Tree Data Structure (ms)	Unique JSON (ms)
African	46.5	169.0
History of Africa	93.2	187.5
African people were slaves	143.5	222.5

Table 2: Average query execution times for Tree Data Structure and Unique JSON Inverted Index

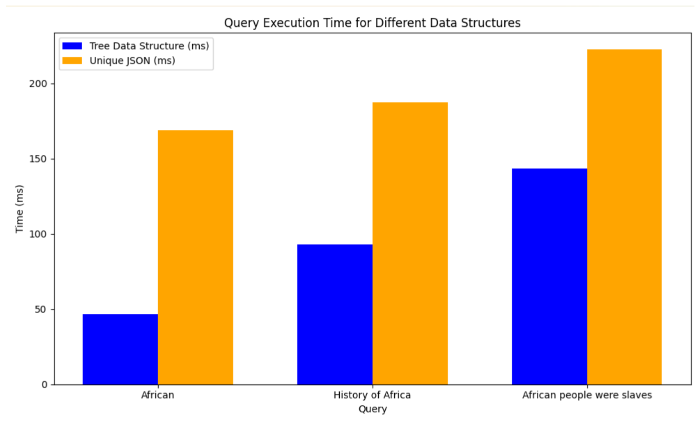

Figure 2: Enter Caption

Table 2 shows that the Tree Data Structure Inverted Index consistently outperformed the Unique JSON Inverted Index in all tested queries. For example, the ”African” query took an average of 46.5 milliseconds using the tree structure, whereas the JSON-based index took significantly longer at 169.0 milliseconds. For more complex queries like ”African people were slaves,” the tree structure maintained a faster response time at 143.5 milliseconds, compared to 222.5 milliseconds for the JSON-based structure.

5.2.4 Conclusion

The results of this experiment indicate that the Tree Data Structure Inverted Index performs better in terms of query execution time, particularly for more complex queries. This improved performance can be attributed to the hierarchical organization of the tree structure, which allows for faster retrieval by narrowing down relevant documents more efficiently. In contrast, the Unique JSON Inverted Index, although simpler to export and manage, demonstrated slower query response times.

Thus, for use cases where search speed is a priority, especially with complex or large-scale queries, the Tree Data Structure Inverted Index is the preferred choice. The trade-off, however, lies in the longer export time during the indexing phase, as observed in the first experiment. Therefore, systems that prioritize query performance over indexing speed will benefit more from the tree-based structure.

6 Conclusion

This study has addressed the creation of a search engine based on an inverted index structure, comparing the efficiency of two distinct approaches: an index based on a tree structure and a single index in JSON format. Through the implementation and evaluation of these structures, relevant conclusions have been drawn regarding their performance and scalability.

Firstly, the results obtained during the export time comparison tests have demonstrated that the single JSON index exhibits superior performance in the export phase, with an average time of 497.83 milliseconds compared to 559.99 milliseconds for the tree-based index. This finding suggests that for tasks requiring rapid consolidation and storage of data, the option of a single JSON file is preferable, especially in scenarios where the data volume is high and simplicity in management is sought.

However, despite its advantage in export, the tree-based index offers more agile access to documents during query execution. The hierarchical structure allows for more efficient retrieval of relevant documents, resulting in faster response times when searching for specific information. This balance between query speed and export time is crucial for designing search engines that handle large volumes of data.

Furthermore, the implementation of a metadata system alongside inverted indexes has expanded the capability to perform complex searches, enhancing the user experience by enabling result filtering based on attributes such as author or publication date. This underscores the importance of integrating contextual information with the index structure to improve the relevance and accuracy of query responses.

In conclusion, the choice of data structure for an inverted index should be guided by the specific requirements of the system, carefully weighing the trade-off between speed in data export and efficiency in queries. The findings of this project provide valuable insights that contribute to the development of more robust and scalable search engines.

7 Future work

As this project has focused exclusively on developing a search engine using Python, several directions for future work can be identified, particularly as we transition to implementing a Java-based version in the next phase. Here are some potential avenues for exploration:

- **Algorithm Optimization:** Future iterations could focus on optimizing algorithms used for both indexing and querying. For example, experimenting with different methods for bulk loading the inverted index or exploring optimized search algorithms could significantly enhance performance.
- **Parallel Processing:** Given the computationally intensive nature of indexing and searching, the implementation of parallel processing techniques in Java should be considered. By utilizing Java's concurrency features, we could potentially reduce the time required for these operations, making the system more scalable.
- **Cross-Language Comparisons:** As we transition from Python to Java, it would be beneficial to conduct a comparative analysis of the performance metrics obtained in both languages. This would provide insights into how the choice of programming language affects the implementation and efficiency of the inverted index.
- **Extending Dataset Sources:** Future experiments should also involve expanding the range of documents used for indexing. By incorporating more diverse and larger datasets, we can better evaluate the system's performance and robustness in real-world scenarios.
- **User Experience Enhancements:** While the current project includes a minimal query engine, future work could focus on developing a more sophisticated user interface for the Java implementation. This could involve creating a web application or enhancing the command-line interface to improve user interaction with the search engine.

By focusing on these areas in future work, we can build upon the groundwork laid in this Python project and create a more efficient, scalable, and user-friendly search engine in Java. These improvements will not only enhance the system's performance but also broaden its applicability in various information retrieval contexts.