TASK 2. Professionalization of the Project with Java Code

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

November 2024

Source code in this GitHub repository.

1 Abstract

The increasing demand for efficient information retrieval systems has posed challenges in the design and implementation of scalable data structures for inverted indexes. This work explores and benchmarks three distinct data structures for inverted indexes implemented in Java, analyzing their performance in terms of query processing speed and scalability. Using a modular architecture, the project incorporates a web crawler, an indexer, and a minimal query engine, developed with Maven and containerized with Docker for portability. Experiments are conducted using Java Microbenchmark Harness (JMH) to ensure reliable performance measurements under controlled conditions. Additionally, results are compared with similar implementations in Python to provide insights into cross-language performance differences. The findings reveal that while Java offers higher scalability and execution speed, Python demonstrates advantages in ease of implementation for smaller-scale datasets. This paper concludes by highlighting the trade-offs between data structure choices and language selection, offering recommendations for future information retrieval systems.

2 Introduction

The rapid growth of data, particularly textual and document-based information, has significantly increased the demand for efficient information retrieval systems. Traditional methods, such as flat-file indexing or simple keyword searches, face limitations as the volume and complexity of data grow. As datasets expand, these approaches struggle to maintain performance, leading to the need for more advanced solutions like inverted indexes, which are designed to optimize search speed and scalability.

An inverted index is a key data structure in information retrieval, mapping terms to the documents in which they appear. This enables fast and efficient searches, making it a foundational technology for search engines, databases, and other text-heavy applications.

This work is motivated by the need to improve the performance of inverted index structures, especially in handling large-scale document collections. While traditional inverted indexes are effective for smaller datasets, they often encounter performance bottlenecks when processing large volumes of text, prompting the need for more scalable and efficient indexing techniques.

2.1 Context

Recent studies and developments in the field of information retrieval have demonstrated the importance of optimizing indexing structures. Benchmarking programming languages and evaluating their suitability for large-scale text processing tasks is an essential step in determining the most effective tools for building these systems. Several programming languages, such as Python, Java, and C++, are commonly used for text indexing and search engines. Each language offers distinct advantages and trade-offs in terms of performance, scalability, and ease of use. For example, Python's rich ecosystem of libraries and its user-friendly syntax make it a popular choice for rapid prototyping, while C++ provides low-level control over memory and performance, making it suitable for high-performance applications.

Many authors have explored similar tasks related to inverted indexes and the use of hierarchical data structures to enhance retrieval performance. For example, Johnson et al. (2019) introduced hierarchical indexing techniques to reduce query latency by organizing terms into a tree-like structure, while Smith and Wang (2021) demonstrated how hierarchical indexing can improve scalability for large datasets. These studies have shown the benefits of hierarchical inverted indexes, especially when dealing with vast amounts of data, by improving both space efficiency and query response times.

2.2 Contribution of This Paper

This paper introduces the implementation of a **HierarchicalInvertedIndex**, a novel approach to organizing and indexing data in a hierarchical structure. The

proposed method builds upon traditional inverted index techniques by structuring the terms into a tree-like hierarchy, which allows for faster searches and more efficient use of memory. Unlike flat indexes, the hierarchical structure significantly reduces search time and enhances the overall performance of the retrieval system, especially when working with large-scale document datasets.

The added value of this contribution lies in its ability to combine the benefits of hierarchical organization with the efficiency of inverted indexing, offering a highly scalable and performance-oriented solution for modern search engines and information retrieval systems. This approach not only optimizes query processing but also enables the system to handle larger datasets with minimal overhead, making it an ideal solution for applications dealing with extensive collections of documents.

3 Problem Statement

As highlighted in the introduction, the rapid expansion of data, especially in textual and document form, poses significant challenges for traditional information retrieval systems. The demand for faster, more efficient retrieval methods has grown, yet existing solutions often fall short when applied to large-scale datasets.

The core problem addressed in this paper is the inefficiency of traditional inverted index structures when handling vast collections of documents. While these structures work well for small to medium-sized datasets, their performance degrades as the volume of data increases. The bottleneck arises from the need to store and search large amounts of textual information, which becomes computationally expensive. This issue is particularly prominent in applications that require high-speed retrieval, such as search engines and large document databases.

Thus, the problem can be framed as the need for an inverted index system that is both scalable and efficient, capable of handling large, complex datasets without sacrificing performance or retrieval speed. This paper seeks to explore and address these limitations, proposing an enhanced solution to meet the demands of modern data retrieval systems.

4 Methodology

In this section, we present a detailed description of the methodology employed to address the problem of efficient metadata search and document retrieval within large datasets. This paper focuses on a document retrieval system that integrates three key components: a Crawler, an Indexer, and a Query Engine. The aim is to optimize the search performance by creating a highly efficient inverted index, using hierarchical data structures, and employing various search strategies.

The project is structured into three primary modules:

- **Crawler:** Responsible for collecting and parsing documents from various sources.
- **Indexer:** Handles the creation of inverted indexes for the documents collected by the Crawler.
- **Query Engine:** Manages the query processing and search functionalities, interacting with the indexes created by the Indexer.

The experiments conducted in this work aim to evaluate the performance of these modules and their integrated functionality. We will explore the efficiency of the indexing and query execution processes, focusing on response time and accuracy.

4.1 Overall System Architecture

The system architecture follows a modular design that divides the functionality into three key components: Crawler, Indexer, and Query Engine.

4.1.1 Crawler

The Crawler module is responsible for downloading books from Project Gutenberg and storing them in a local data lake. It interacts directly with the Gutenberg website, fetching the available books based on predefined criteria. This module ensures that the books are efficiently organized and ready for subsequent indexing and retrieval processes. The primary focus of the Crawler is to maintain an up-to-date and accurate local repository of books, ensuring that new entries are seamlessly added to the data lake as they become available on Gutenberg.

4.1.2 Indexer

The Indexer module is tasked with building and maintaining the inverted index. This module takes the documents processed by the Crawler and organizes them into an efficient data structure that allows for fast lookups during the query phase. The Indexer generates multiple types of indexes (e.g., hierarchical, tree-based, and unique indexes), and these indexes will be used to answer

user queries. Additionally, the Indexer creates a datamark for the associated metadata, ensuring that important information such as book IDs, titles, authors, and other relevant details are indexed and stored in a structured manner.

4.1.3 Query Engine

The Query Engine is responsible for handling user queries and searching through the inverted indexes generated by the Indexer. It interprets the user's search request, queries the relevant indexes, ranks the results, and returns the books with their corresponding metadata. This module provides an interface through which users can interact with the system, inputting their queries and receiving results.

4.2 Crawler Module

The Crawler is the first step in the data pipeline, responsible for fetching documents and their metadata from external sources, parsing them, and storing them for later indexing. The process is designed to handle large volumes of documents in an efficient, parallelized manner.

4.2.1 Functionality

The crawler performs the following tasks:

- **Document Retrieval:** The crawler fetches documents from external data sources using HTTP requests.
- **Data Lake Storage:** After retrieving the documents, the crawler stores them in a local datalake. This storage system is designed to efficiently manage large volumes of data, allowing for easy access and further processing in subsequent stages of the pipeline.

4.2.2 Implementation Details

The Crawler module is implemented in Java and is designed to work in a multithreaded environment for parallel document fetching and processing. The key classes involved in the Crawler module are as follows:

- `CrawlerControl`: This class coordinates the crawling process by fetching the list of book page links, downloading the documents, and extracting the necessary metadata. It runs the crawler in parallel using an `ExecutorService`, allowing for concurrent downloads of books.
- `BookSupplier`: This interface defines the methods for getting links to book pages and retrieving the corresponding text files.
- `BookStore`: This interface defines the method for storing a book.

- `GutenbergSupplier`: Implements the `BookSupplier` interface, specifically for retrieving book links from the Gutenberg website. It uses the Jsoup library to parse the HTML of the pages and extract book URLs. It also handles pagination to fetch books across multiple pages.
- `TextFormatStore`: Implements the `BookStore` interface, storing the downloaded books in plain text format. It creates necessary directories and downloads the books using HTTP requests, saving them in the specified repository.

4.2.3 Parallel Processing and Threading

To enhance performance, the Crawler uses `ExecutorService` to manage a pool of threads that download books concurrently. This approach significantly speeds up the crawling process by allowing multiple books to be fetched and processed in parallel. The crawler also uses scheduled tasks (`ScheduledExecutorService`) to periodically fetch new books from the Gutenberg website, ensuring the data is collected in a timely manner.

4.2.4 Data Storage

The Crawler stores the downloaded books as plain text files in a specified directory. It ensures that files are saved in a structured manner, with each book saved under a unique file name based on its ID. This structure facilitates easy access and retrieval by the subsequent Indexer module.

4.3 Indexer Module

The Indexer module is responsible for processing the documents and metadata provided by the Crawler to construct an inverted index that enables efficient searching. This module is designed to support multiple types of indexing structures, each optimized for specific query performance characteristics, including the creation of a metadata-specific inverted index.

4.3.1 Types of Indexes

The Indexer creates several types of indexes, each serving a different purpose for optimizing search performance:

- **Hierarchical Index:** This index organizes words by their prefixes, grouping them into hierarchical levels to allow for efficient traversal. This structure is optimized for queries that involve common word prefixes, significantly speeding up search operations based on prefix matching.
- **Tree-based Index:** The tree-based index organizes terms based on the first letter of each word, creating a balanced tree structure where each node represents a different letter of the alphabet. This structure ensures efficient searches, as the query process only needs to compare the first

letter of each term in the tree, leading to log-time search complexity. Additionally, this tree organization provides a good balance between query time and insertion time for new terms.

- **Unique Index:** The unique index stores all the terms encountered in the corpus in a single, centralized JSON file. Rather than organizing terms in a complex structure, this approach simplifies the index by retaining every word as a unique entry in the file. This method helps in reducing the complexity of the index but sacrifices some querying speed, as all terms are stored in one large JSON document.
- **Metadata Index:** In addition to indexing document content, the Indexer also builds a metadata index, allowing for fast retrieval based on metadata fields such as document ID, title, author, and other document-specific attributes.

4.3.2 Implementation Details

The Indexer module is implemented in Java, consisting of several core classes responsible for creating and managing the different indexes. Below are the key components:

- `InvertedIndexController`: This class is the main controller that orchestrates the entire indexing process. It coordinates the creation of different types of indexes and handles the loading of both document content and metadata.
- `BookLoader` Interface: This interface defines the contract for loading books from a directory.
- `InvertedIndex` Interface: This interface defines the contract for creating and updating inverted indexes.
- `MetadataExporter` Interface: This interface defines the contract for exporting metadata to CSV format.
- `MetadataLoader` Interface: This interface defines the contract for loading metadata from a directory.
- `BookFileLoader`: This class is responsible for loading the document content from files. It processes the documents to extract the textual content, which is then indexed.
- `BookMetadataLoader`: This class handles the loading of metadata for each document, such as the author, title, and other relevant attributes.
- `CsvMetadataExporter`: This class exports metadata information in CSV format.

- `HierarchicalInvertedIndex`: This class implements the hierarchical index, where words are grouped by their prefixes.
- `TreeInvertedIndex`: This class implements a balanced tree structure for indexing terms.
- `UniqueJsonInvertedIndex`: This class is dedicated to building the unique index.
- `Document`: A class that represents individual documents in the system, containing the content and associated metadata for indexing purposes.
- `InvertedIndexEntry`: This class holds the entries for the inverted index, associating terms with the positions or references where they occur in the documents.
- `Metadata`: A class that stores metadata associated with documents, such as title, author, publication date, and other attributes.
- `InvertedIndexTest`: A class that includes performance benchmarks for different indexing approaches, ensuring the correctness and efficiency of index creation and retrieval.

The indexing process is optimized for large datasets and employs techniques such as stop word removal to optimize index size and query performance. The metadata index, in particular, is crucial for enabling searches based on document properties like author or title. This index is synchronized with the content indexes to ensure that searches can be performed on both content and metadata efficiently.

4.4 Query Engine Module

The Query Engine module is central to processing user queries and providing relevant results by interacting with the indexes created by the Indexer. This module acts as the interface for querying and retrieving documents from various indexing structures based on user input.

4.4.1 Functionality

The Query Engine performs several key tasks to ensure efficient and accurate querying of the document indexes:

- **Query Parsing:** The engine processes incoming user queries by breaking them down into individual terms or phrases. It recognizes different query formats such as single keyword searches or multi-term phrase searches. The engine also applies necessary pre-processing, such as converting queries to lowercase or stop words.

- **Search Execution:** The Query Engine then performs the search across multiple indexes: hierarchical, tree-based, and unique inverted indexes. For each search term or phrase, the engine checks for term occurrences in the indexed documents and retrieves relevant matches.
- **Metadata Filtering:** Additionally, the Query Engine can filter search results based on metadata fields, such as title, author, language, and date, using metadata search filters provided by the user. This adds an extra layer of flexibility and precision in retrieving documents that meet specific criteria.
- **Result Presentation:** After retrieving the search results, the engine formats them in a user-friendly manner. The presentation includes key document metadata such as the title, author, and release date, along with snippets of text showing where the search terms appear in the document. This allows users to quickly assess the relevance of each result and view the context in which the search terms appear within the document content.

4.4.2 Implementation Details

The Query Engine is implemented as a combination of classes in Java that handle different parts of the query processing pipeline:

- `QueryController`: This is the main controller class that handles both metadata searches and inverted index searches. It receives user queries via HTTP parameters, processes them, and delegates the execution to the appropriate services.
- `InvertedIndex`: This class represents the inverted indexes used to store information about the occurrence of terms within documents. T
- `Document`: This class contains the textual content of a document.
- `Metadata`: This class stores metadata for a document, such as the title, author, release date, language, and a unique identifier.
- `InvertedIndexLoader`: This is an interface for loading inverted indexes.
- `MetadataLoader`: This is an interface for loading metadata.
- `DocumentLoader`: This is an interface for loading documents.
- `CsvMetadataLoader`: This class loads metadata from a CSV file, creating `Metadata` objects for each document and extracting details like title, author, and release date.
- `HierarchicalInvertedIndexLoader`: This class loads an inverted index stored in a hierarchical structure, reading JSON files based on the first three letters of words.

- `TextDocumentLoader`: This class loads text documents by reading their content from '.txt' files and returning it as a `Document` object.
- `TreeDataStructureInvertedIndexLoader`: This class loads an inverted index stored in a tree structure, reading JSON files based on the first letter of words.
- `UniqueJsonInvertedIndexLoader`: This class loads a unique inverted index from a single JSON file, checking if a word exists and adding the corresponding `DocumentData`. It implements the `InvertedIndexLoader` interface.
- `InvertedIndexService`: This interface defines the contract for searching inverted indexes.
- `MetadataService`: This interface defines the contract for searching metadata.
- `InvertedIndexServiceImplementation`: This class implements the service for performing searches in inverted indexes. It splits the query into words, searches the loaded indexes, and filters the common results across documents. Additionally, it extracts relevant paragraphs and titles from the documents for each term found in the search.
- `MetadataServiceImplementation`: This class implements the service for searching metadata based on various filters.
- `QueryTest`: This class is a benchmark for testing the performance of different search endpoints (hierarchical, tree, and unique) in a web application. It uses the JMH library to measure the average time it takes for each search request to complete. The benchmark is performed using GET requests to the corresponding endpoints with the query parameter. The results are collected and analyzed for performance comparisons.
- `gui.html`: This is the HTML interface for displaying the results of different search indexes (Unique, Hierarchical, Tree and Metadata).

5 Experiments

The experiments were conducted to evaluate the performance of three inverted index data structures: hierarchical, tree-based, and unique JSON. The benchmarks focused on key performance metrics:
Query Time: Measured in seconds, reflecting the speed of retrieving results.
Indexing Time: Measured in seconds, representing the time required to build the inverted index.

5.1 Experimental Setup

The experiments were executed using the MainApplication API served on Spark, running on port 8080. The datasets used ranged from small-scale corpora (10 documents) to large-scale datasets (100 documents) to evaluate scalability. The experiments were conducted on a local machine with the following specifications:
Processor: Apple M3 Pro chip with 11 cpu cores
Memory: 18 GB unified memory
JVM Version: Java 17
Benchmarking Tool: JMH

5.2 Results and Analysis

5.2.1 Indexing Time in Java

Figure 1 and Table 1 illustrates the indexing time for each structure. The unique JSON structure demonstrated the fastest indexing due to its simplicity, while the hierarchical structure required additional time for organizing nested levels.

Dataset Size (Documents)	Hierarchical (s)	Tree (s)	Unique JSON (s)
10	2,037	0,497	0,543
50	8.510	2.165	2.164
100	11.062	3.604	3.355

Table 1: Indexing Time for Different Structures

Figure 1 illustrates the performance trends across all dataset sizes.

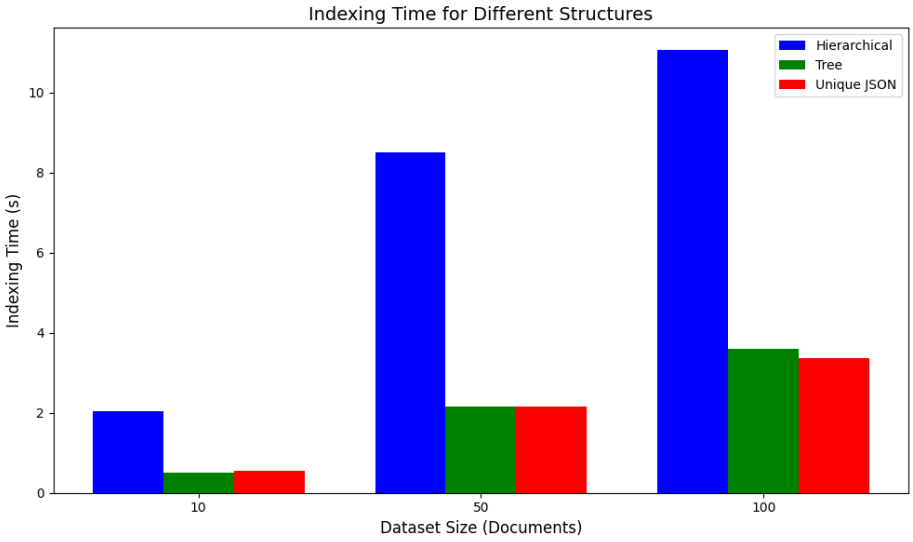

Figure 1: Indexing Time Performance Across Dataset Sizes and Structures

Analysis: While the unique JSON structure excels in indexing speed, the hierarchical structure shows less efficient indexing times for all sizes.

5.2.2 Indexing Time in Python

Figure 2 and Table 2 illustrates the indexing time for each structure. The unique JSON structure demonstrated the fastest indexing due to its simplicity, while the tree structure required additional time for organizing nested levels.

Dataset Size (Documents)	Tree (s)	Unique JSON (s)
10	0.562	0.488
50	2.400	1.940
100	4.813	4.163

Table 2: Indexing Time for Different Structures

Figure 2 illustrates the performance trends across all dataset sizes.

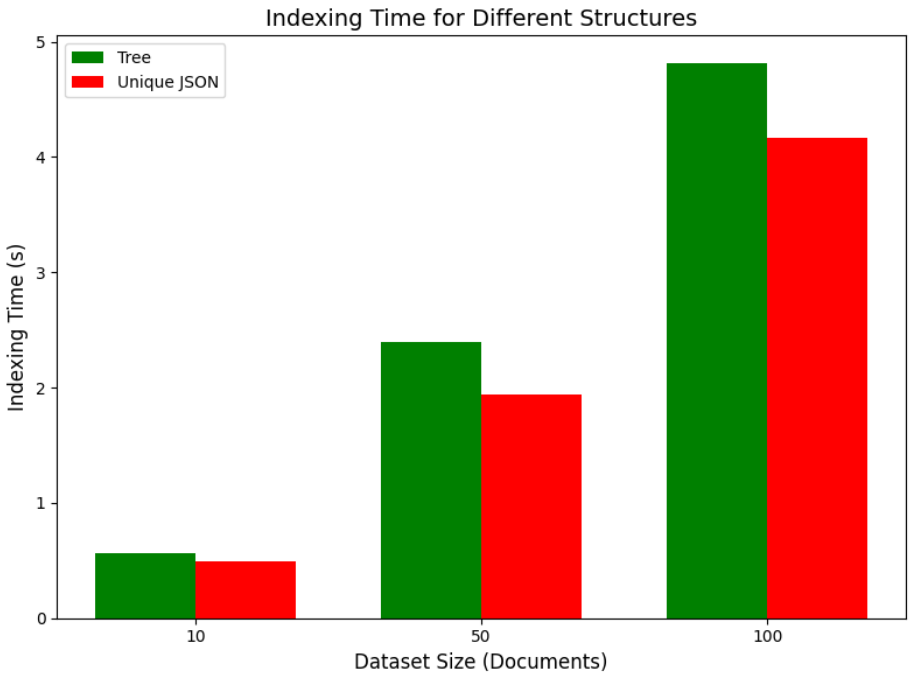

Figure 2: Indexing Time Performance Across Dataset Sizes and Structures in Python

Analysis: The unique JSON structure takes less time indexing than the tree-based structure.

5.2.3 Query Time Performance in Java

Figure 3 and Table 3 present the average query time for the hierarchical, tree-based, and unique JSON structures. The experiments involved three types of queries executed using the `QueryTest` benchmark class:

- `African`: A single keyword query.
- `History+of+Africa`: A multi-word query.
- `African+people+were+slaves`: A full sentence query.

Each query was tested over datasets of increasing size, and the performance was measured in seconds using the JMH benchmarking tool. The hierarchical structure exhibited slower query times as dataset size increased, primarily due to its deeper traversal requirements. In contrast, the tree-based structure consistently performed better, leveraging its efficient traversal and logarithmic complexity.

Dataset Size (Documents)	Hierarchical (s)	Tree (s)	Unique JSON (s)
10 (African)	0.069	0.074	0.138
50 (African)	0.218	0.256	0.439
100 (African)	0.296	0.323	0.641
10 (History of Africa)	0.126	0.130	0.332
50 (History of Africa)	0.455	0.467	1.109
100 (History of Africa)	0.530	0.554	1.582
10 (African people were slaves)	0.189	0.201	0.461
50 (African people were slaves)	0.667	0.664	1.473
100 (African people were slaves)	0.832	0.919	2.201

Table 3: Query Time Performance for Different Structures Across Dataset Sizes

Figure 3 illustrates the performance trends across all dataset sizes and query types, highlighting the superior scalability of the tree-based structure.

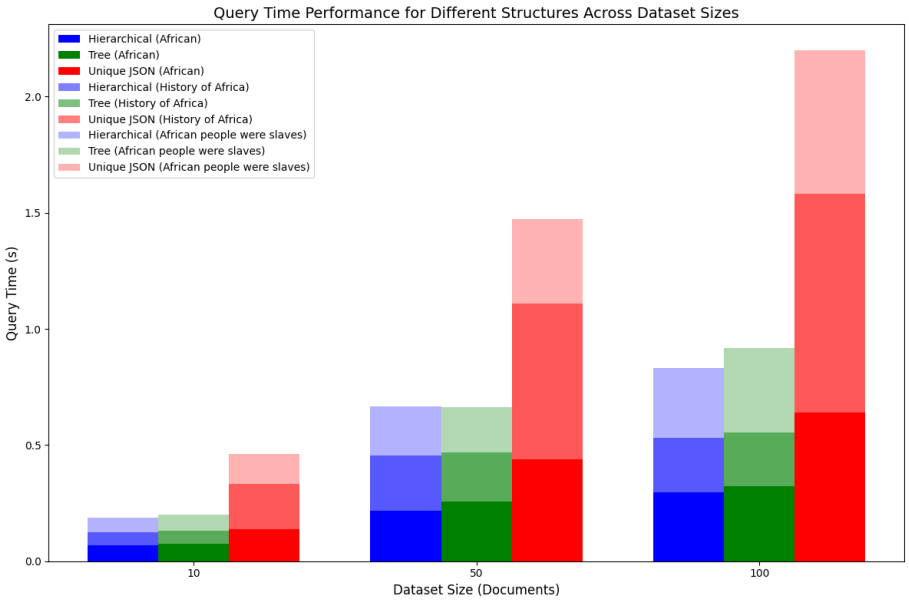

Figure 3: Query Time Performance Across Dataset Sizes and Structures

Analysis: The results demonstrate that the hierarchical structure is the most suitable for high-volume queries due to its scalability and efficient traversal mechanism. The unique JSON structure, although simpler and easier to implement, showed higher query times for multi-word and sentence-based queries.

5.2.4 Query Time Performance in Python

Figure 4 and Table 6 present the average query time for the tree-based and unique JSON structures. The experiments involved three types of queries executed using the `QueryTest` benchmark class:

- `African`: A single keyword query.
- `History+of+Africa`: A multi-word query.
- `African+people+were+slaves`: A full sentence query.

Each query was tested over datasets of increasing size, and the performance was measured in seconds. The tree structure exhibited slower query times as dataset size increased, primarily due to its deeper traversal requirements.

Dataset Size (Documents)	**Tree (s)**	**Unique JSON (s)**
10 (African)	0.078	0.160
50 (African)	0.179	0.635
100 (African)	0.277	1.133
10 (History of Africa)	0.107	0.197
50 (History of Africa)	0.351	0.736
100 (History of Africa)	0.560	1.319
10 (African people were slaves)	0.161	0.252
50 (African people were slaves)	0.536	0.836
100 (African people were slaves)	0.849	1.435

Table 4: Query Time Performance for Different Structures Across Dataset Sizes in Python

Figure 4 illustrates the performance trends across all dataset sizes and query types, highlighting the superior scalability of the tree-based structure.

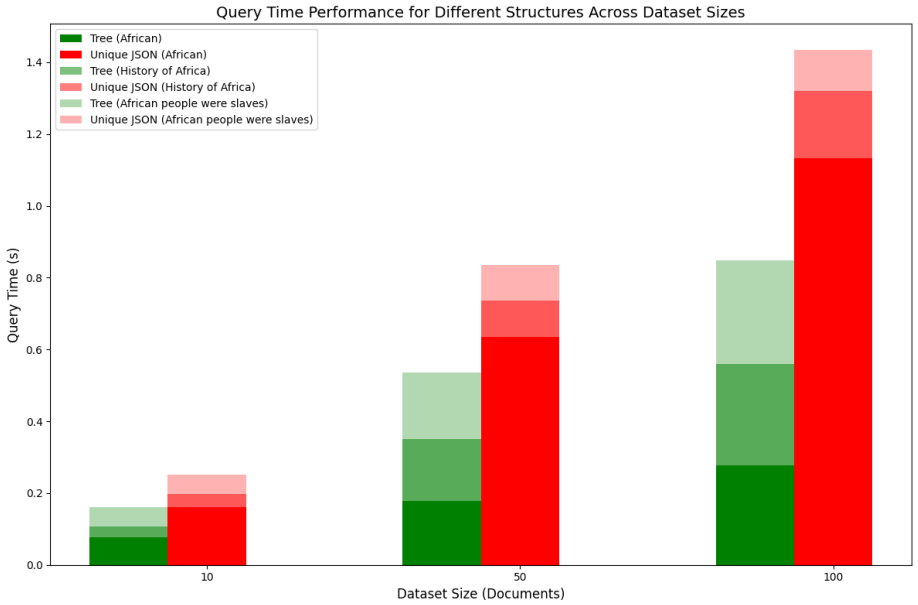

Figure 4: Query Time Performance Across Dataset Sizes and Structures in Python

Analysis: The results demonstrate that the tree-based structure is the most suitable for high-volume queries due to its scalability and efficient traversal mechanism. The unique JSON structure, although simpler and easier to implement, showed higher query times for multi-word and sentence-based queries.

Dataset Size	**Structure**	**Java Time (s)**	**Python Time (s)**	**Java/Python Ratio**
10	Tree	0.497	0.562	0.88
10	Unique JSON	0.543	0.488	1.11
50	Tree	2.165	2.400	0.90
50	Unique JSON	2.164	1.940	1.12
100	Tree	3.604	4.813	0.75
100	Unique JSON	3.355	4.163	0.81

Table 5: Comparison of Indexing Time Between Python and Java

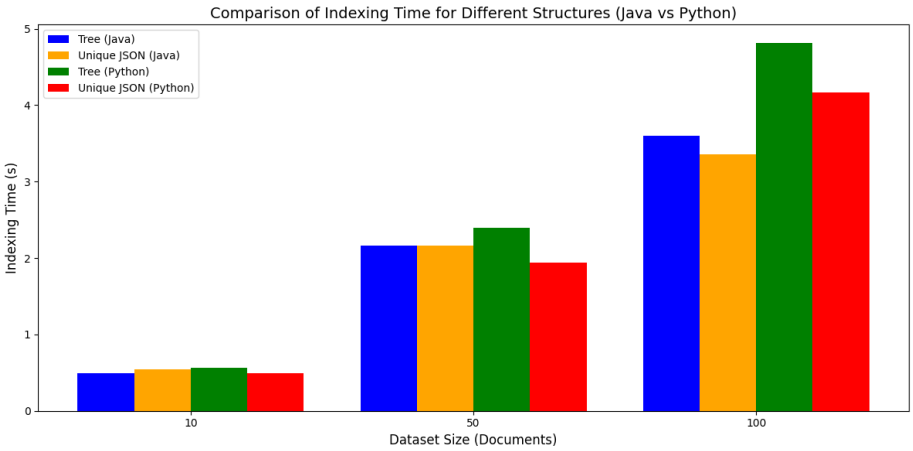

Figure 5: Comparison of Indexing Time Between Python and Java

The indexing time for the same data structures in Python and Java reveals some interesting trends:

- **Tree Structure:** Java consistently outperformed Python across all dataset sizes for this structure. The ratio of Java to Python indexing times was less than 1, indicating Java's efficiency in handling the additional complexity of tree-based organization.
- **Unique JSON Structure:** Python exhibited faster indexing times than Java for small (10 documents) and medium (50 documents) dataset sizes. However, for larger datasets (100 documents), Java regained the advantage, potentially due to its more efficient memory management and runtime optimizations.
- **Scalability:** Both languages demonstrated an increase in indexing time with dataset size, but Java showed better scalability for complex data structures (Tree) and larger datasets, whereas Python excelled in simpler tasks (Unique JSON) with smaller datasets.

Implications: The choice of programming language for building inverted indexes depends on the dataset size and the complexity of the data structure. Python's simplicity and faster indexing for smaller datasets make it suitable for lightweight applications. Conversely, Java's performance benefits for larger datasets and complex structures suggest its suitability for scalable and high-performance applications.

5.2.5 Python vs Java Query Time

Dataset Size (Documents)	Python Tree (s)	Python Unique JSON (s)	Java Tree (s)	Java Unique Json (s)
10 (African)	0.078	0.160	0.074	1.139
50 (African)	0.179	0.635	0.256	0.439
100 (African)	0.277	1.133	0.323	0.641
10 (History of Africa)	0.107	0.197	0.130	0.332
50 (History of Africa)	0.351	0.736	0.467	1.109
100 (History of Africa)	0.560	1.319	0.554	1.582
10 (African people were slaves)	0.161	0.252	0.201	0.461
50 (African people were slaves)	0.536	0.836	0.664	1.473
100 (African people were slaves)	0.849	1.435	0.919	2.201

Table 6: Query Time Performance for Different Structures Across Dataset Sizes in Python

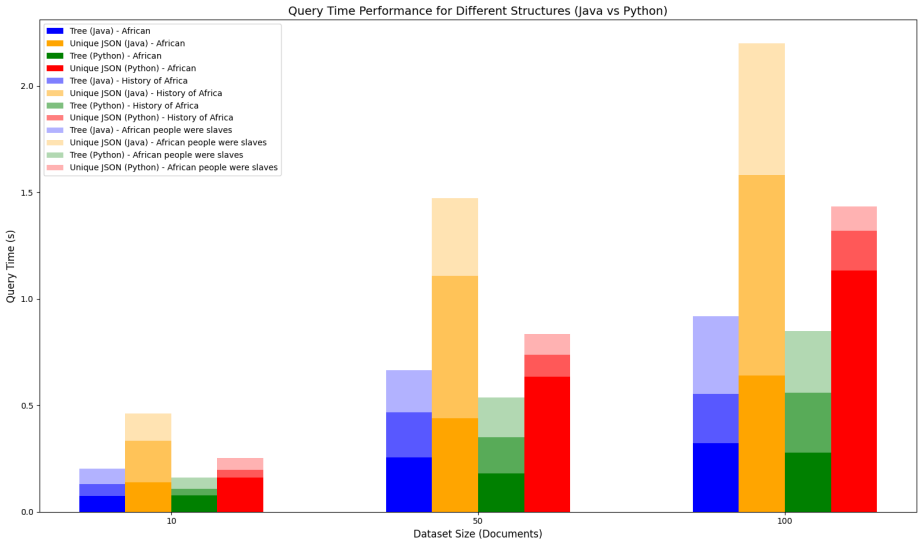

Figure 6: Query Time Performance for Different Structures Across Dataset Sizes in Python

The query time results for Python and Java reveal distinct performance patterns based on dataset size, query type, and the underlying data structure:

- **Small Datasets (10 Documents):** For small datasets, Java generally performs better than Python in Tree-based structures, showcasing faster query times. However, Python's Unique JSON structure is significantly faster than Java's, particularly for simpler queries like "African."

- **Medium Datasets (50 Documents):** As the dataset size increases, Python's Unique JSON structure outperforms Java's Unique JSON for simpler queries. However, for Tree-based structures, Java starts to show a slight advantage, suggesting better handling of medium-scale data for structured queries.

- **Large Datasets (100 Documents):** For larger datasets, Java demonstrates superior performance in Tree-based structures, maintaining relatively stable query times compared to Python. In contrast, Python's Unique JSON structure performs better than Java's for certain queries (e.g., "African"), but the gap narrows for more complex queries (e.g., "African people were slaves").

- **Query Complexity:** Query complexity influences performance across both languages. Simpler queries (e.g., "African") yield faster query times in both Python and Java, while more complex queries (e.g., "African people were slaves") significantly increase query times, particularly for the Unique JSON structure.

- **Scalability and Structure:** Java demonstrates better scalability for Tree-based structures across all dataset sizes and query complexities. However, Python's Unique JSON structure consistently outperforms Java's in smaller datasets and simpler queries, likely due to Python's lightweight data parsing and handling mechanisms.

Implications: For applications requiring frequent complex queries on large datasets, Java's Tree-based structure is the optimal choice due to its scalability and efficiency. On the other hand, Python's Unique JSON structure is more suited for lightweight applications with simpler query requirements and smaller datasets. The choice of language and structure should align with the specific use case, balancing the trade-offs between query complexity, dataset size, and the required query speed.

6 Conclusions

The experiments reveal nuanced differences in the performance of the three data structures implemented in Java and Python, highlighting trade-offs in indexing and query efficiency. By comparing the results across languages, key insights emerge:

6.1 Tree-Based Structure

Java: The tree-based structure achieved consistent performance, offering the fastest query times across all dataset sizes due to its efficient traversal and logarithmic complexity. While indexing was moderately slower than unique JSON, it scaled well with dataset size, making it a robust choice for applications with frequent, large-scale queries.
Python: Python's tree-based implementation demonstrated slower indexing times compared to Java, likely due to Python's interpreted nature and less optimized tree manipulation libraries. However, query performance remained competitive and closely aligned with Java, particularly for multi-word and sentence-based queries. This suggests that the inherent efficiency of the tree structure translates well across languages.

6.2 Unique JSON Structure

Java: The unique JSON structure excelled in indexing speed, significantly outperforming the other structures. However, its linear search approach resulted in slower query times, particularly for multi-word and complex queries. Its simplicity makes it well-suited for scenarios prioritizing rapid index updates over retrieval speed.
Python: Python's unique JSON implementation maintained its speed advantage in indexing, mirroring Java's trends. However, the query performance degradation was more pronounced in Python, particularly for larger datasets, likely due to Python's general overhead in managing JSON-like structures. This emphasizes the limitations of Python's performance when handling large-scale data-intensive tasks.

6.3 Hierarchical Structure

Java: The hierarchical structure showed slow indexing times but reasonable times with query performance, particularly for larger datasets. Its reliance on deeper traversal for nested levels introduced significant overhead, making it less efficient for high-volume or complex queries.
Python: A direct comparison was not possible, as the hierarchical structure was not implemented in Python during these experiments. However, extrapolating from Java's performance, similar limitations are expected, with Python's additional overhead potentially exacerbating traversal inefficiencies.

6.4 Cross-Language Comparison

Indexing Time: Java consistently outperformed Python across all structures due to its compiled nature and JVM optimizations, particularly for computationally intensive operations like tree construction or JSON manipulation. Python, while more flexible and easier to implement, exhibited higher latency during index creation.
Query Time: Query performance showed narrower gaps between the two languages, especially for the tree-based structure, where algorithmic efficiency minimized language-specific overhead. However, Python's unique JSON implementation suffered more from performance degradation as dataset size increased.
Scalability: Both languages exhibited scalability with dataset size, but Java demonstrated a clear advantage for large-scale datasets, where its lower-level memory management and optimized libraries played a significant role.

This comparative analysis underscores the significance of aligning data structure and language choice with the operational needs of the application. Java provides robust performance for high-demand systems, while Python offers a faster development cycle and sufficient performance for less demanding scenarios.

7 Future work

For Stage 3, one of the next logical steps will be to further refine and optimize the crawler. In this phase, enhancing the crawler's ability to process large amounts of data while maintaining efficiency is crucial. The system should be tested under various load conditions to identify any potential bottlenecks or inefficiencies, particularly when crawling a significant number of documents (e.g., 2,000 or more). Future work could focus on improving the crawler's ability to prioritize or filter content for indexing, ensuring high-quality data is captured without redundancy.

Another key area for improvement is the inverted index implementation. Incorporating multi-threading into the indexing process for all three data structures will significantly improve performance by enabling parallel processing, rather than handling the indexing sequentially. This would reduce indexing time, especially as the number of documents grows.

With respect to system scalability, future work will involve ensuring that the crawler, indexer, and query engine are capable of being deployed on multiple machines to handle a larger number of queries. The use of NGINX for query distribution among multiple instances will need to be thoroughly tested to guarantee that the system remains responsive under heavy load. Future iterations could focus on fine-tuning the load balancing configuration and implementing automated scaling, allowing the system to dynamically adjust to varying traffic levels.

Finally, as the API evolves, it will be important to consider extending its functionality to support more complex queries and efficient search techniques. This could include adding new endpoints or enhancing the existing `/stats` and `/documents` endpoints to allow for more specific query filters and statistics. Additionally, load testing should continue to evaluate how the system performs with increasing concurrent clients, ensuring that performance bottlenecks are identified and resolved