Search Engine

Stage 2

María Alonso León, Víctor Gil Bernal, Jacob Jażdżyk, Kimberly Casimiro Torres

Big Data
School of Computer Engineering (EII)
Las Palmas de Gran Canaria University, Spain

November 2024

Contents

ABSTRACT

This project presents the development of a high-performance search engine implemented in Java, designed to efficiently index and query digital books from the Gutenberg Project. Leveraging a modular architecture and multiple storage technologies, including MongoDB, Neo4j, text files, and binary files, the system demonstrates significant advancements in performance, scalability, and maintainability compared to its predecessor developed in Python. Key improvements include the application of SOLID principles and design patterns, enhanced metadata and inverted index management, and the integration of an intuitive user interface alongside a RESTful API. Experimental results reveal that MongoDB excels in inverted index storage and scalability, while text file systems are particularly efficient for frequent and infrequent word searches. Neo4j, was less effective in text-heavy searches. Binary file storage, despite its compactness, exhibited limitations in processing speed, highlighting areas for future optimization. The transition to Java enabled notable enhancements in runtime performance, memory management, and the ability to scale with larger datasets. This project underscores the advantages of Java for large-scale search engine development, providing a robust and extensible framework for Big Data applications. The results and methodologies offer a valuable reference for researchers and developers aiming to optimize search engines for specialized datasets.

KEYWORDS: Search Engine, Java, MongoDB, Neo4j, File System, Inverted Index, Metadata Storage, Docker, Parallelization, Scalability, Performance Optimization, Benchmarking, Project Gutenberg, Big Data

1. Introduction

In recent years, the efficiency of search engines has become a major focus in computing due to the exponential increase in the amount of digital information. The design and optimization of search engines involve a multitude of considerations, including programming language choice, system architecture, and data storage approaches. Java has become a preferred language for high-performance applications, as it offers strong type-checking, efficient memory management, and adherence to principles such as SOLID, which facilitate maintainable and scalable code. These characteristics make Java particularly suitable for the development of robust search engines capable of handling large-scale datasets efficiently.

The current project focuses on building a search engine specifically designed to index and search digital books extracted from the Gutenberg Project, a vast public domain digital library. Books from this source are stored in a data lake and indexed with a combination of MongoDB, filesystem (both text and binary files), and Neo4j databases to allow for flexible metadata handling and high-performance inverted indexing. By leveraging this multi-database approach, the search engine aims to optimize query performance and efficiently store metadata and text indexes. Additionally, implementing Java in this project, as opposed to the initial Python prototype, enables better alignment with SOLID principles, enhanced performance, and improved resource management.

A key element in search engine optimization involves removing common words, or stopwords, from indexed texts to increase the relevance of search results. This project incorporates stopword removal as part of its preprocessing pipeline to improve indexing efficiency. Furthermore, by implementing binary encoding, this search engine offers a notable innovation: data stored with a custom format is compressed into binary to optimize storage space, reducing the overall data footprint.

Related work in the field of search engines typically includes large-scale systems like Apache Lucene and Elasticsearch, or even specialized website search engines [1], which also use Java for performance-critical components. These projects provide high-speed, full-text search capabilities, but they primarily cater to generalized search applications. In contrast, this work focuses on a specialized corpus—digital books—and introduces a unique multi-database architecture combined with the benefits of Java for performance benchmarking and code scalability.

In this paper, we present a powerful search engine architecture with a focus on efficient data management and optimized performance for book indexing. By incorporating RESTful APIs and an intuitive UI, this search engine demonstrates enhanced usability and query versatility for end-users. The proposed system offers a robust and flexible approach to digital book indexing, advancing both performance and storage efficiency in specialized search applications.

2. Problem Statement

As the volume of digital content continues to grow, the design and optimization of search engines are critical to ensuring efficient access to large datasets. In particular, selecting the appropriate data structures and technologies for indexing and querying vast collections of data, such as books, is essential for balancing performance, scalability, and resource efficiency. One of the key challenges faced by developers is choosing the right technology stack to manage the data in a way that ensures both fast query responses and low resource consumption, while also maintaining a high degree of usability for end users.

The primary problem addressed by this project is evaluating which data storage solution—MongoDB, Neo4j, or the filesystem—provides the best performance for a search engine designed specifically for digital books. Each technology has its strengths and weaknesses in terms of scalability, query speed, and storage efficiency. MongoDB is a document-based database, well-suited for storing large amounts of semi-structured data, while Neo4j offers a graph-based approach, which is ideal for handling relationships between entities. The filesystem is a simpler solution, but its efficiency depends on how data is stored and indexed. For that reason, text and binary files have been also considered. Evaluating these four storage systems and understanding their trade-offs is critical for improving the overall performance of search engines in specialized contexts, such as the book domain.

In addition to the evaluation of storage systems, this project aims to demonstrate the advantages of using Java for developing scalable and maintainable search engine systems. Java, with its strong object-oriented principles and adherence to SOLID principles, provides a more structured and extensible approach compared to the Python prototype previously developed. The shift to Java allows for better code organization, easier integration of design patterns, and a more professional development environment that is suitable for building production-level applications. Without a well-chosen technology stack, the system could suffer from slower search times, inefficient memory usage, and an overall poor user experience, thus undermining the purpose of providing users with a seamless and efficient search platform.

Furthermore, the Python prototype lacked a proper user interface (UI), which is crucial for a more interactive user experience. The Java implementation, on the other hand, incorporates an effective REST API and an intuitive UI, enhancing the overall user experience and enabling the system to support advanced search queries.

Ultimately, the main goal of this project is to provide a clear comparison of these data storage solutions in the context of a book search engine, while also showcasing the benefits of Java for developing scalable, maintainable, and user-friendly applications. The outcomes of this evaluation will be valuable to researchers and developers seeking to optimize search engine performance for large-scale text-based datasets.

3. Methodology

This section details the methodology used in the development of an optimized search engine in Java, specifically designed to manage large volumes of textual data robustly and efficiently. This project represents a significant extension and enhancement of the initial work done in Python, where key areas for improvement were identified in terms of performance, modularity, and scalability. To address the limitations observed in the previous implementation, the methodology in this phase focuses on employing advanced structural design principles, such as the SOLID principles and widely recognized design patterns, as well as implementing a modular architecture that facilitates the development, maintenance, and expansion of the system.

The transition from Python to Java is justified not only by Java's capabilities in terms of performance and memory management efficiency but also by the robustness and flexibility it offers for developing a system that must scale and handle concurrent requests. Java, being a strongly typed object-oriented language, enables a more structured and controlled framework, which is crucial to meet the scalability and maintainability standards required in Big Data applications. Additionally, this Java implementation incorporates significant improvements in data processing and storage, such as the use of high-performance libraries and the adoption of parallelization techniques, enabling more efficient management of queries and storage of indexed data.

To achieve a modular and highly efficient architecture, the system has been structured into independent components, each with a specific function aligned with the principles of single responsibility and decoupling. This structure not only facilitates system scalability but also allows each component to be tested and optimized, which is essential in a real-time search system that must respond quickly and accurately to user queries.

Before detailing the specific system components, this section will first explore the SOLID design principles that guided the development process. These principles, together with carefully chosen design patterns, provided a structured foundation that enhances the flexibility, robustness, and extensibility of the system. Each principle will be thoroughly examined to demonstrate its impact on the design and functionality, establishing a solid framework for the advanced search engine required in Big Data applications.

3.1. SOLID Design Principles and Design Patterns

3.1.1 Single Responsibility Principle (SRP)

This principle states that each module or class should have a single responsibility, meaning it should be responsible for a specific task within the system. In the search engine, this principle is respected by dividing each task into independent modules. For example, the Crawler is solely responsible for data extraction from Project Gutenberg, without interfering with indexing or query execution. The Indexer, on the other hand, is solely responsible for processing and storing information in the inverted index and metadata structures. This ensures that any change or improvement in one of these modules will not affect the others, facilitating system maintenance and debugging.

3.1.2 Open/Closed Principle (OCP)

The system is designed to be open for extension but closed for modification, meaning that new functionalities can be added without modifying existing code. This is achieved through the use of interfaces and abstract classes that allow additional search functionalities to be implemented without altering the core classes of the Query Engine. For instance, if a semantic or error-tolerant search feature is needed, new modules or classes can be created to implement these features and connect them to the engine without changing the existing structure. This helps maintain system stability and minimizes the risk of introducing errors when adding new features.

3.1.3 Liskov Substitution Principle (LSP)

This principle ensures that derived classes can replace their base classes without affecting system functionality. In the search engine, this principle is particularly useful in implementing different types of queries. For example, the Query Engine is designed to use specific classes that implement various query types, whether through the inverted index or by querying databases like MongoDB or Neo4j. Each query type can replace others in the search engine without requiring additional modifications, allowing the system to function consistently regardless of the query's internal implementation.

3.1.4 Interface Segregation Principle (ISP)

This principle suggests that interfaces should be specific and contain only the necessary methods for each implementing class. In this project, specific interfaces are applied to different modules, allowing each system component to use only the interfaces it requires. For example, the interfaces used for query management in the Query Engine differ from those in the Indexer. This reduces dependencies between modules and ensures that each component is isolated and fully independent, facilitating maintenance and expansion.

3.1.5 Dependency Inversion Principle (DIP)

This principle states that high-level classes should not depend on low-level classes, but rather on abstractions. In the search engine, this is applied through the use of interfaces that allow high-level modules, such as the Query Engine, to interact with low-level modules, like the adapters for MongoDB, Neo4j, and the file system, without relying on their specific implementations. This approach makes it possible to modify or replace these low-level modules without affecting the core logic of the system, enabling greater flexibility in technology choice and ensuring that the system is prepared for future updates or changes in data infrastructure.

Together, the application of SOLID principles in this project contributes to a robust and well-organized architecture that facilitates maintenance, improves scalability, and allows new features to be added without impacting the system structure. These principles also ensure that each component is easily testable, which is essential for a system that must manage large volumes of textual data efficiently and at scale.

3.2. Architecture Structure

The architecture of this project is designed to optimize the performance and scalability of a search engine capable of handling large volumes of textual data. The system structure follows principles of modularity and is divided into several main components, each with a specific function within the search engine. These modules include the Crawler, the Indexer, the Query Engine, the User Interface (UI). Each module facilitates the system's development, maintenance, and scalability by following design principles like SOLID and architectural patterns that ensure cohesion and reduce coupling.

3.2.1 Crawler

The Crawler module serves as the initial data entry point within the system. Its purpose is to periodically download documents from Project Gutenberg website, and store them for further processing. Implemented in Java, the Crawler uses specific libraries to efficiently handle internet connections and file downloads. A key improvement in this phase, compared to the initial Python project, is the introduction of concurrency techniques in Java, allowing multiple documents to be downloaded simultaneously and optimizing data acquisition time. The Crawler configuration allows specifying storage paths for the downloaded files and employs a logging mechanism to track sources and download dates, facilitating data monitoring and updates.

- **Datalake**
  Once downloaded, documents are stored in the Datalake, which serves as a central repository for raw files. This datalake is organized in a hierarchical file system, structured into folders by download date. Within the datalake, each document is assigned a unique identifier, preventing conflicts and ensuring that the system can handle large data volumes in an organized manner. The organization of the datalake allows other modules to access the information without altering its content or storage structure.

3.2.2 Indexer

The Indexer module is responsible for transforming downloaded documents into a structured format and creating the necessary indexes for search, which are the datamarts. This process occurs in multiple phases. In the first phase, each document is tokenized and converted into a map structure, where each keyword is associated with its respective document. To optimize the search process, an inverted index is implemented, recording each word along with its locations and frequencies within the texts. This allows for quick and precise keyword searches within the document corpus.

Additionally, the Indexer manages the metadata structure, which includes details such as author, publication year, and document language. Metadata processing is performed using regular expressions to extract relevant information and store it in MongoDB. The modularity of the Indexer allows the system to be extensible, enabling the addition of new metadata fields or processing functions in the future without requiring significant changes to the base code.

- **Datamart**
  The Datamart represents the next level in the data hierarchy. Unlike the Datalake, which stores raw documents, the Datamart contains structured, processed data that is ready to be indexed and queried.

3.2.3 Query Engine

The Query Engine is the core of the search engine and is responsible for processing user queries. This module is designed to support keyword-based queries and metadata-based queries, using the inverted index and metadata stored in the Datamart to efficiently return results.

The Query Engine is designed to be extensible; it is possible to add new search functionalities, such as semantic, by implementing new classes that integrate with the system without modifying existing modules. This ensures system stability and minimizes the risk of errors when introducing new features.

- **API**: The API acts as the access point for external applications to perform queries and operations within the system. Developed with Java and Spark, it provides a series of endpoints that allow interaction with the Query Engine and Datamart modules. The API allows users to perform queries and retrieve results without needing direct access to the underlying database. Each endpoint is designed to handle different query types, whether by keyword or metadata, and returns results in a structured, easily interpretable format.

  One of the main objectives of the API is to maintain security and efficiency in communications. To achieve this, authentication and request limiting mechanisms are implemented, which protect the system from misuse and ensure optimal performance at all times. Additionally, the API is configured to run in a Dockerized environment, enabling deployment across various environments and facilitating system scalability.

3.2.4 UI (User Interface)

The User Interface (UI) provides direct and easy access to the system for end users. Developed in React, it allows users to perform queries, view results, and obtain data statistics in an intuitive and visually appealing manner. The UI is fully interactive and communicates with the API to retrieve query results in real-time, allowing users to navigate data smoothly and efficiently.

3.3. Benchmarking and Testing

Performance evaluation is a critical part of the system, as it allows measuring and optimizing the efficiency of the different modules and components within the search engine. For this purpose, unit and performance tests have been implemented using JUnit and JMH (Java Microbenchmarking Harness). These tests evaluate different aspects of the system, such as query response time, inverted index efficiency, and system scalability when handling large data volumes.

Benchmark results are documented and compared with the original Python implementation, enabling the identification of significant improvements in terms of speed and memory usage. These tests also ensure that the system functions correctly under high-load conditions and remains stable even when processing large amounts of data or handling multiple simultaneous queries.

3.4. Docker and Deployment

The system leverages Docker to achieve scalability and flexibility in deployment. Each service is encapsulated within its own container, defined by specific `Dockerfile` configurations, which ensure that all necessary dependencies and runtime environments are included. This approach facilitates modularity, as each container operates independently and communicates with other components through an internal network.

To further enhance the system's capabilities, volumes and ports are used to manage data sharing and accessibility:

- **Volumes:** Shared volumes, such as `datalake` and `datamart`, are mounted across multiple containers. These volumes allow persistent data storage and seamless exchange of information between services, ensuring consistency and efficient data flow. For example, both the *crawler* and *indexer* services share the `datalake` volume, enabling the indexed data to be accessible for querying or further processing.
- **Ports:** The system exposes specific ports to enable communication between containers and external clients. For instance, the *query_engine* service is accessible through port 8080, while the *ui* service is exposed via port 3000. These mappings facilitate user interaction with the system and ensure that external requests are routed to the appropriate service.

The system's architecture is further enhanced through the use of Docker networks, such as the `app-network`, which provides isolated communication channels between containers. This design ensures secure and efficient interaction among services, while maintaining a clean separation from external traffic.

Overall, this architecture, based on Docker, enables the system to handle large-scale, complex workflows. By combining technologies like MongoDB, Neo4j, and Docker, the system achieves robustness, scalability, and ease of deployment, making it well-suited for Big Data applications requiring high performance and adaptability.

4. Experiments and Tests

This section presents the experiments designed to evaluate the performance and efficiency of the search engine's storage system when handling different data volumes in various storage configurations. Each experiment has specific objectives that allow for the analysis of different aspects of performance and responsiveness in technologies such as MongoDB, Neo4j, text files, and binary files, within Big Data contexts. These experiments range from measuring search times for frequent and infrequent words to comparing processing times as the number of stored documents increases.

The purpose of these experiments is to observe how each system responds to varying workloads, measuring the average time and the average time pe per operation and analyzing how each technology behaves as the number of documents in the system grows. By evaluating each system's ability to perform searches and process large volumes of data, patterns of performance, efficiency, and scalability limitations can be identified. These analyses are crucial for determining which of these technologies is more suitable for large-scale, intensive search environments.

The following sections present and analyze the obtained graphs, exploring in detail each system's behavior based on the type of search and the volume of data. This analysis enables direct comparisons between the technologies, identifying strengths and weaknesses in different scenarios and configurations.

4.1. Processing Time

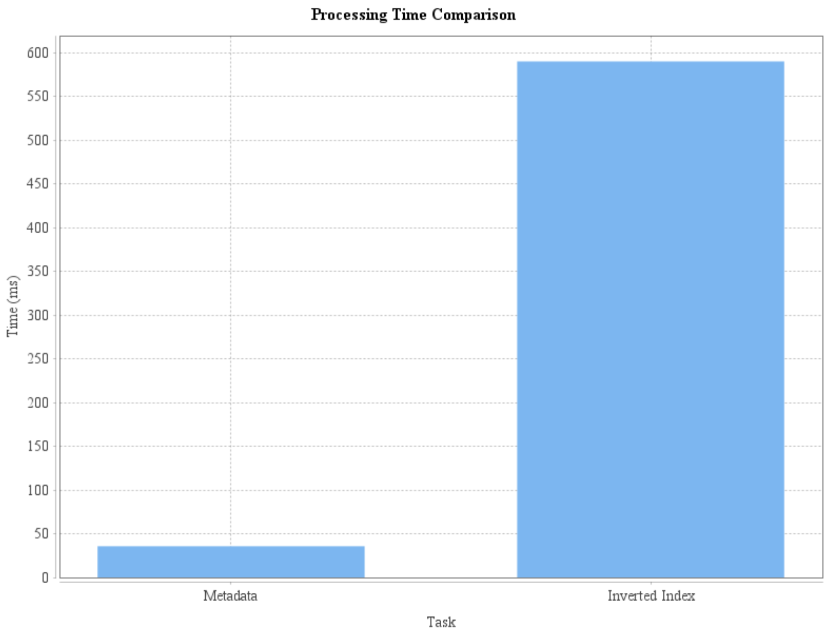

Figure 1: Processing Time Comparison

The figure 1 compares the processing time between two main tasks of the system: metadata handling (Metadata) and inverted index management (Inverted Index). The X-axis presents the two categories analyzed, while the Y-axis shows the processing time in milliseconds (ms). The purpose of this visualization is to highlight the performance difference between these tasks, providing a clear perspective on which is more efficient in terms of the time required to complete operations.

The metadata handling task is clearly the most efficient, showing a significantly lower average processing time compared to the inverted index. This reflects that operations related to metadata—such as extracting, storing, and querying descriptive information (author, publication date, language, etc.)—are well-optimized within the system. Metadata operations typically involve smaller and more specific structures, which means that they require fewer computational resources and are processed faster. This high level of efficiency is crucial since metadata is often used for quick and direct searches that do not depend on the full text content but instead rely on key descriptive information.

On the other hand, inverted index management is considerably slower, showing the highest processing time among the two tasks evaluated. This is due to the inherent complexity of the task, which involves processing large volumes of text, breaking it into tokens (keywords), and building a structure that associates each word with its location in the corresponding documents. This process is computationally intensive, especially when the data volume is significantly large, as is the case with the digital library from the Gutenberg Project. Although the inverted index is essential for efficient keyword-based searches, its initial construction and updates require considerable time, explaining its poorer performance in this comparison.

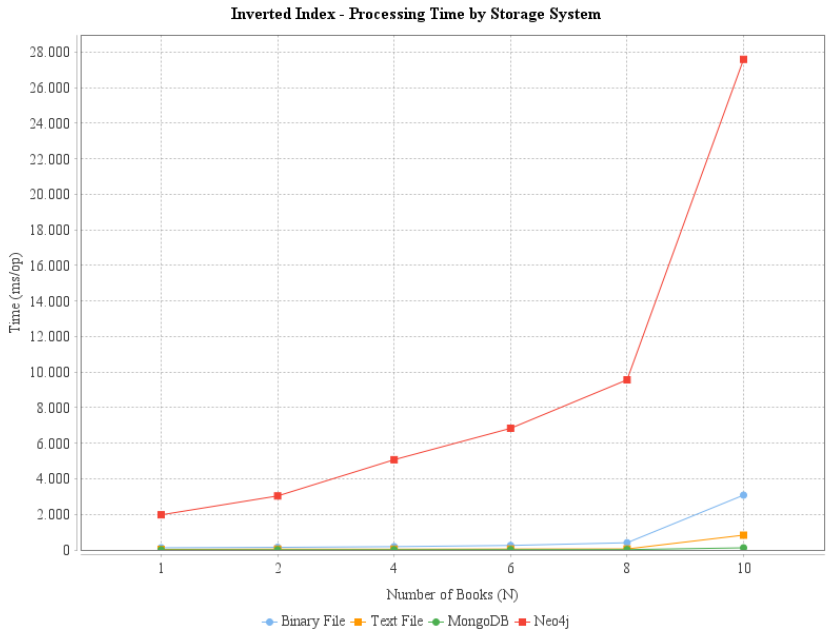

Figure 2: Inverted Index - Processing Time by Storage System

The figure 2 illustrates the storage operation processing time for different inverted index storage systems as the number of documents or books increases. On the X-axis, the number of books (representing data volume) is shown, and on the Y-axis, the processing time in milliseconds per operation (ms/op) is displayed. This chart aims to compare how each system (MongoDB, Text File, Binary File, and Neo4j) responds in terms of processing time as the data volume grows, helping to assess the scalability and efficiency of each technology within a Big Data environment.

The specific behavior of each system as observed in the chart is detailed below:

MongoDB stands out as the most efficient storage system in terms of processing time. As the number of documents increases, MongoDB consistently shows a low and stable processing time. This behavior reflects an optimized structure for handling large volumes of data without significant performance penalties, making it the most scalable and reliable option for large-scale search applications in Big Data.

The text file system ranks second in processing efficiency. Although its performance is lower than MongoDB's, the text file maintains a relatively stable processing time as the number of documents increases. However, there is a slight increase in operation time as the data grows, indicating that while the text file is a reasonable option for moderate data volumes, it may not be as scalable as MongoDB for high-volume data applications.

The binary file shows moderate performance, ranking as the third option in processing efficiency. Although it handles small and medium data volumes well, its processing time begins to increase gradually as the document volume grows. This suggests that the binary file is not fully optimized for large data volumes and may experience performance limitations in Big Data environments.

Neo4j performs the worst among all evaluated systems, showing a significant increase in processing time as the number of books grows. This exponential increase in processing time suggests that Neo4j is not suitable for handling large volumes of data in inverted indexes. Its graph-oriented structure appears to introduce considerable overhead when managing large data volumes, limiting its effectiveness in large-scale search applications. This behavior implies that, while Neo4j may be useful in applications requiring complex data relationships, it is not the most suitable option for intensive data search operations with high document volumes.

In conclusion, the analysis suggests that MongoDB is the most recommended technology for storing inverted indexes in large-scale search applications due to its ability to maintain low and stable performance even with large data volumes. Text File may be an acceptable choice for moderate volumes, while Binary File and Neo4j present significant scalability limitations, with Neo4j being the least suitable for this type of application due to its high processing time in high-volume scenarios.

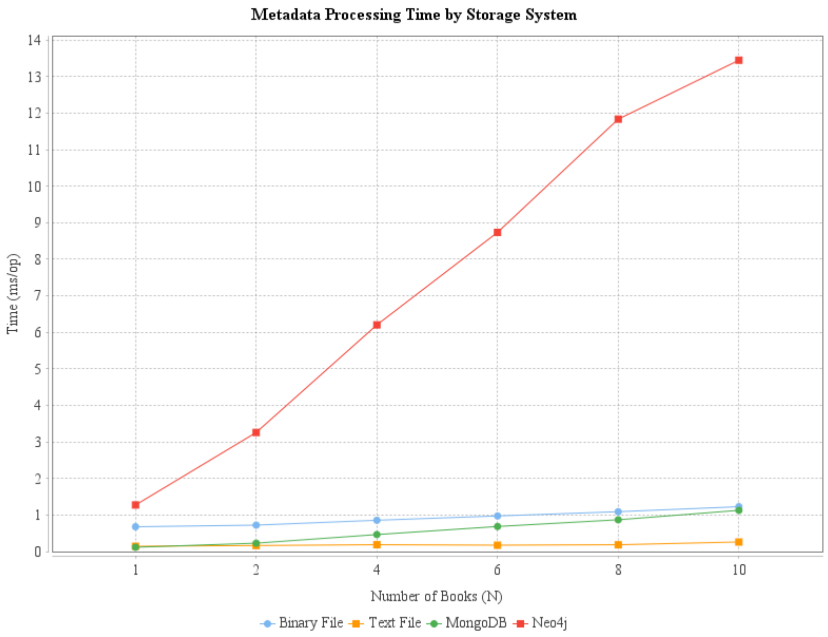

Figure 3: Metadata - Processing Time by Storage System

The figure 3 demonstrates the processing time per operation for different storage systems managing metadata as the number of documents or books increases. The X-axis represents the number of books (indicating data volume), while the Y-axis shows the processing time in milliseconds per operation (ms/op). This graph aims to compare the performance of each system (Text File, MongoDB, Binary File, and Neo4j) in terms of metadata processing time as the data volume scales up, helping assess the scalability and efficiency of each technology in a Big Data context.

Here is a detailed breakdown of the performance of each system as observed in the chart:

Text File emerges as the most efficient storage system for handling metadata, demonstrating the lowest and most stable processing time across all data volumes. Even as the number of books increases, the Text File system maintains a minimal increase in processing time, indicating strong scalability and suitability for large-scale applications. This stability suggests that the structure of the Text File is particularly well-suited for managing metadata, allowing it to handle data growth without significant performance degradation.

MongoDB ranks as the second most efficient system for metadata processing. Although it does not outperform the Text File in terms of processing speed, MongoDB shows relatively stable and low processing times as the data volume grows. This suggests that MongoDB's internal architecture is optimized to a certain extent for managing metadata, making it a viable alternative for applications requiring reliable and scalable metadata storage, albeit with slightly higher processing times than Text File.

The Binary File system ranks third in terms of processing efficiency for metadata. While it performs reasonably well with smaller data volumes, its processing time increases at a slightly higher rate compared to Text File and MongoDB as the number of documents grows. This indicates that Binary File, while adequate for small to medium-sized datasets, may face scalability challenges in handling larger volumes of metadata, potentially leading to bottlenecks in performance in Big Data environments.

Neo4j is the least efficient among the evaluated systems for metadata processing, showing the highest increase in processing time as the number of books grows. The chart shows a steep and consistent rise in processing time with each increase in data volume, reflecting Neo4j's lack of optimization for managing metadata in large datasets. This significant performance penalty suggests that Neo4j's graph-oriented structure may not be ideal for metadata storage in scenarios requiring low latency and scalability. While Neo4j is powerful for applications involving complex relationships, it is less suitable for metadata storage where rapid access and minimal processing time are crucial.

When comparing the results of the metadata processing times with those from the inverted index processing times, some notable differences and consistencies emerge.

Firstly, Text File consistently performs well across both metadata and inverted index contexts, maintaining a stable and low processing time, but it shines more noticeably in metadata management, where it achieves the best results. This indicates that Text File's straightforward structure is particularly effective for metadata storage, possibly due to simpler data access patterns when managing metadata compared to handling a full-text index.

MongoDB also shows strong performance in both metadata and inverted index scenarios, but it performs slightly better with inverted index storage. This suggests that MongoDB's document-oriented design is versatile, handling complex data structures effectively in both contexts, though with a slight edge in full-text indexing, where it secures the top position.

The Binary File system shows moderate efficiency in both cases, but it performs relatively better in the inverted index storage compared to metadata. Its gradual increase in processing time with metadata suggests that it faces more limitations in this context, possibly due to less flexibility in managing metadata compared to full-text indexing.

Finally, Neo4j demonstrates consistent performance issues in both scenarios, showing the steepest increase in processing time as data volumes grow. This consistency in high processing times reaffirms that Neo4j's graph structure is not optimized for rapid data retrieval and storage operations required for metadata or inverted index management in Big Data environments. Neo4j's strengths in relationship handling do not translate effectively to these storage needs, making it the least suitable choice across both contexts.

In summary, while Text File and MongoDB show flexibility and strong performance in both metadata and inverted index management, each system's relative efficiency differs slightly based on the type of data being stored. Text File is particularly strong with metadata, while MongoDB leads slightly with inverted index storage. Binary File holds a moderate position in both scenarios, and Neo4j consistently underperforms, highlighting its unsuitability for high-speed storage operations in Big Data applications.

4.2. Search Times with Varying Number of Books

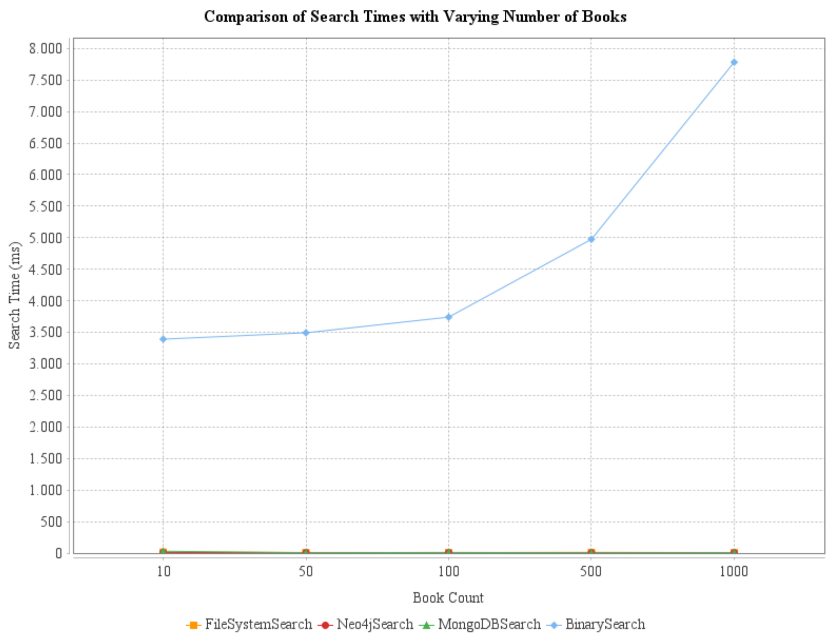

Figure 4: Search Times with Varying Number of Books

The figure 4 shows the search time in milliseconds (ms) for different storage systems as the number of books increases. The X-axis represents the number of books, reflecting the data volume, while the Y-axis shows the search time measured in milliseconds. The purpose of this chart is to evaluate the performance of each system (MongoDB, Neo4j, text file, and binary file) in terms of search time as the data volume grows, providing a clear perspective on the scalability and efficiency of each technology in a Big Data context.

Below is a description of the behavior of each system as observed in the chart:

MongoDB, Neo4j, and the text file system demonstrate outstanding efficiency in terms of search times. All three systems maintain low and stable search times as the number of books increases, indicating that they are well optimized to handle search workloads in a scalable data environment. The stability of these three systems suggests they are highly suitable for data-intensive search applications, as their search time does not experience a significant increase even as the book volume in the system reaches higher levels. This consistency in response times is crucial for maintaining a fast and efficient user experience in large-scale applications.

The binary file, shows very different behavior compared to the other systems. From the beginning, the binary file presents noticeably higher search times than the others, and this time continues to increase significantly as the number of books in the system grows. The steep, upward curve indicates that the binary file is not optimized to handle searches in large data volumes, showing very limited scalability for this type of operation. As the data volume increases, the binary file becomes a less viable option due to its high search time, which could lead to an inefficient user experience and negatively impact the overall performance of the system in large-scale applications.

In summary, the chart reveals a clear difference in the efficiency of the evaluated storage systems for search operations in large data volumes. MongoDB, Neo4j, and the text file are the most recommended options, showing low and consistent search times even with a high number of books in the system. On the other hand, the binary file stands out negatively, as its search time increases significantly with the growth of the data volume, limiting its suitability for applications that require fast and efficient searches in Big Data contexts. This comparison provides a better understanding of each system's strengths and limitations regarding search times, offering crucial information for selecting the most suitable technology based on specific performance and scalability requirements.

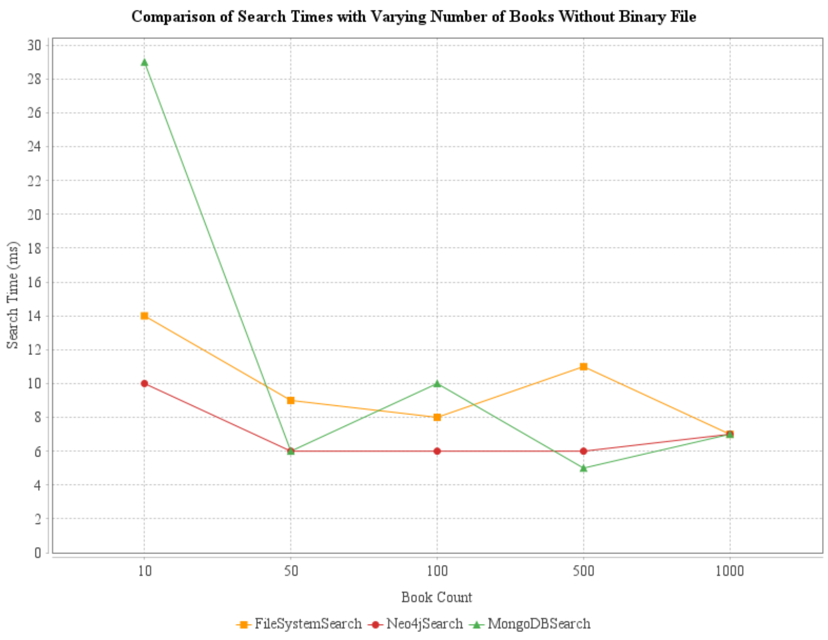

Figure 5: Search Times with Varying Number of Books Without Binary File

The figure 5 shows the search time in milliseconds (ms) for different storage systems, excluding the binary file, as the number of books in the system increases. The X-axis displays the number of books, representing the data volume, while the Y-axis indicates the search time in milliseconds. This chart aims to illustrate the behavior of Neo4j, the file system, and MongoDB in terms of search time as data volume grows, allowing for an analysis of their efficiency and consistency in a Big Data context.

Neo4j, stands out as the most efficient storage system in this chart. From the start, it maintains a low and stable search time, and throughout the chart, it shows remarkable consistency. Although there are some minor fluctuations, they are minimal compared to the other systems. This suggests that Neo4j, when handling an increasing number of documents, achieves good optimization in its searches, maintaining efficiency in a growing data environment. At the end of the chart, Neo4j converges with the other systems at the same point, reflecting stable search times for large data volumes.

The file system, ranks second in terms of search efficiency. Its behavior throughout the chart shows some variations, with certain rises and dips in search time as the volume of books grows. Despite these fluctuations, the file system maintains a relatively low and stable search time compared to MongoDB, indicating that it can handle moderate data volumes with adequate performance. Like Neo4j, the file system finishes at the same point in time as the other systems, consolidating its stability when managing large volumes of data.

MongoDB, exhibits the most unstable performance in this comparison, with more pronounced fluctuations in search time throughout the chart. Initially, MongoDB shows a high peak in search time, which then decreases, but as the number of books increases, it displays significant variations, indicating less consistent performance compared to Neo4j and the file system. These oscillations suggest that MongoDB may face challenges in efficiently handling searches as data volume grows, though it ultimately reaches the same final time point as the other systems, indicating that it achieves a certain level of stability under high-scale conditions, albeit with less consistency.

When comparing this chart to the previous one, where the binary file was included, clear differences in the overall behavior of the systems can be observed. In the chart without the binary file, the three systems (Neo4j, file system, and MongoDB) converge at the same endpoint regarding search time, indicating a common stability at high data volumes. This contrasts with the previous chart, where the binary file showed much higher search times, making it the least efficient option and standing out negatively compared to the other systems. The absence of the binary file in this chart allows for a clearer comparison between Neo4j, the file system, and MongoDB, highlighting that Neo4j and the file system are preferable options in terms of stability and consistency with large data volumes, while MongoDB, although still viable, shows variability that could impact its performance in intensive data search applications.

4.3. Average Time - Least/Most Frequent Words

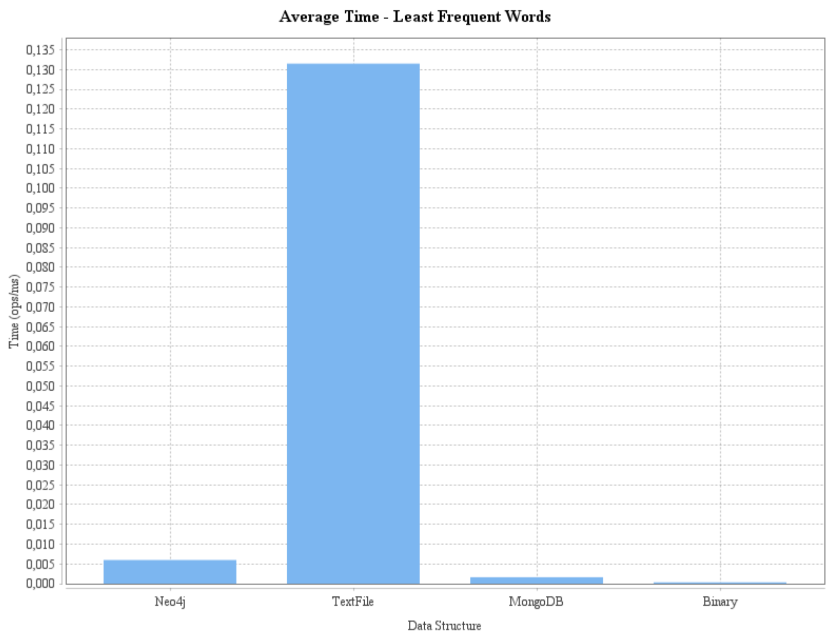

Figure 6: Average Time - Least Frequent Words

The figure 6 shows the average time per operation (op/ms), expressed in operations per millisecond, taken by each storage system to perform searches for words that appear infrequently in the dataset. On the X-axis, the different storage systems evaluated (Neo4j, Text File, MongoDB, and Binary File) are represented, while the Y-axis shows the average number of operations performed per millisecond.

The Text File system stands out as the most efficient for searching infrequent words, showing the highest number of operations performed per millisecond compared to the other evaluated systems. This indicates that the Text File is optimized for handling quick queries of low-frequency terms, making it the most recommended option for this type of operation. Its high performance reflects an efficient and straightforward storage structure for performing searches with minimal latency.

Neo4j ranks second in efficiency for searching infrequent words. Although its performance is slightly lower than that of the Text File, Neo4j maintains a fast and stable performance. Its graph-oriented structure, generally better suited for managing complex relationships, demonstrates acceptable efficiency for handling infrequent terms. This positions Neo4j as a solid option, although less efficient than the Text File system.

MongoDB shows moderate performance in this chart, placing third. While its average number of operations per millisecond is lower than both Text File and Neo4j, MongoDB still offers acceptable performance for low-frequency queries. Its document-oriented design is versatile but appears less optimized for handling this specific type of operation compared to the Text File and Neo4j systems.

The Binary File system stands out negatively in this chart, showing the lowest average number of operations performed per millisecond among all the evaluated systems. This result suggests that the Binary File is not well-suited for searching infrequent words, as its storage structure requires significantly more time to process these queries. This inefficiency makes the Binary File the least suitable option for handling large volumes of data in low-frequency queries, as its performance is markedly inferior compared to the other technologies.

In summary, the Text File emerges as the most efficient option for searching infrequent words, closely followed by Neo4j, which also demonstrates solid and reliable performance. MongoDB, while functional, shows lower efficiency compared to the top two systems, and the Binary File lags significantly behind, with considerably lower performance in terms of operations per millisecond. This chart confirms that in contexts where high efficiency is required for low-frequency term searches, the Text File and Neo4j are the most recommended options, showing a clear advantage in terms of speed and efficiency, while the

Binary File exhibits significant limitations.

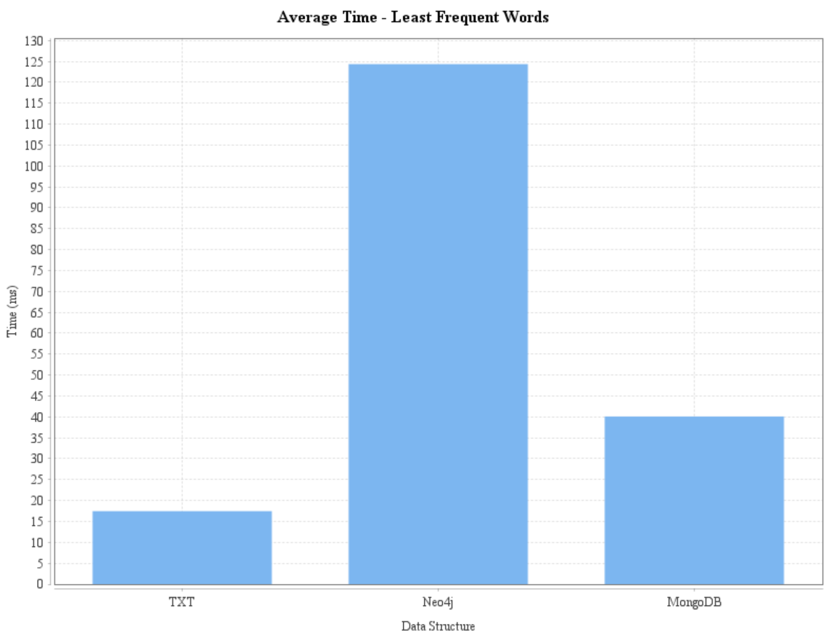

Figure 7: Average Time - Least Frequent Words Without Binary File

The figure 7 shows the average processing time for less frequent words for TXT, MongoDB, and Neo4j. In this chart, the X-axis represents the different storage systems, while the Y-axis indicates the processing time in milliseconds (ms) per operation. This analysis focuses on evaluating each technology's efficiency in handling searches for infrequent words in an environment without the binary storage system, allowing for a more direct comparison between TXT, MongoDB, and Neo4j.

Regarding the results, the TXT storage system demonstrates the best performance, with the lowest average processing time among the three systems. This suggests that, in the absence of the binary system, TXT file storage is an optimal choice for infrequent word searches, maintaining operations at a minimal time cost. TXT proves to be especially efficient and stable, reaffirming its viability in search scenarios where word frequency is low.

MongoDB, on the other hand, ranks second in efficiency. Although its processing time is higher than TXT, MongoDB remains at acceptable and relatively low levels, indicating it is also capable of effectively handling searches for less frequent words. This performance by MongoDB suggests that, while not as fast as TXT in this specific context, it remains a solid and reliable option, especially in environments that require a more structured and accessible database.

Neo4j, once again, shows the worst performance among the three evaluated systems. Its processing time is significantly higher, indicating inefficiency in handling searches for infrequent words. This inefficiency could be due to Neo4j's graph-oriented structure, which seems to introduce significant overhead in such operations. As a result, Neo4j proves to be less suitable for this type of search compared to TXT and MongoDB.

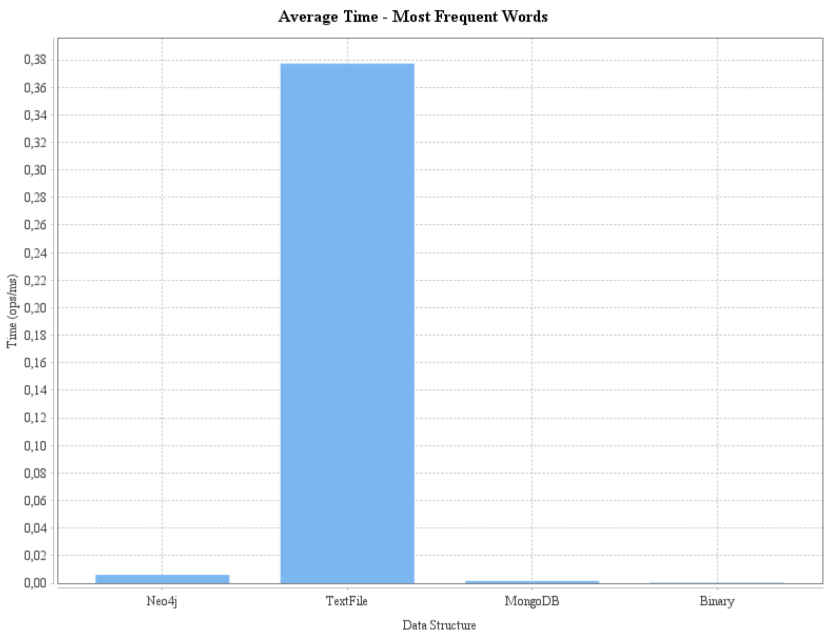

Figure 8: Average Time - Most Frequent Words

The figure 8 shows the average number of operations performed per millisecond (ops/ms) when searching for the most frequent words across four storage systems: Neo4j, Text File, MongoDB, and Binary File. The X-axis lists these evaluated storage systems, while the Y-axis represents the average number of operations per millisecond. The objective of this graph is to analyze the efficiency of each system in handling high-frequency terms, a crucial aspect in search applications where certain terms are used repeatedly.

In terms of performance, the Text File stands out as the most efficient system, achieving the highest number of operations per millisecond among the evaluated systems. This indicates that the Text File is highly optimized for handling frequent terms, maintaining exceptional performance even in repetitive operations. This result reinforces the suitability of the Text File for scenarios requiring fast and efficient access to high-frequency data.

Neo4j ranks as the second most efficient system in this comparison. Although its performance is lower than that of the Text File, Neo4j maintains fast and stable performance. Its graph-oriented structure, typically designed for handling complex relationships, demonstrates good capability for managing frequent terms, positioning it as a solid option, albeit less efficient than the Text File.

MongoDB comes in third place in this comparison. Although its average number of operations per millisecond is lower than that of Neo4j and the Text File, MongoDB still offers acceptable performance for high-frequency searches. Its document-oriented design is versatile but appears to be less optimized for this specific type of operation compared to the top two systems.

The Binary File, on the other hand, exhibits the worst performance in this graph, with the lowest number of operations per millisecond among all the evaluated systems. This result suggests that the Binary File is not well-suited for handling frequent terms, as its storage structure introduces higher processing times for this type of query. This inefficiency makes it the least suitable option for intensive search operations involving common terms.

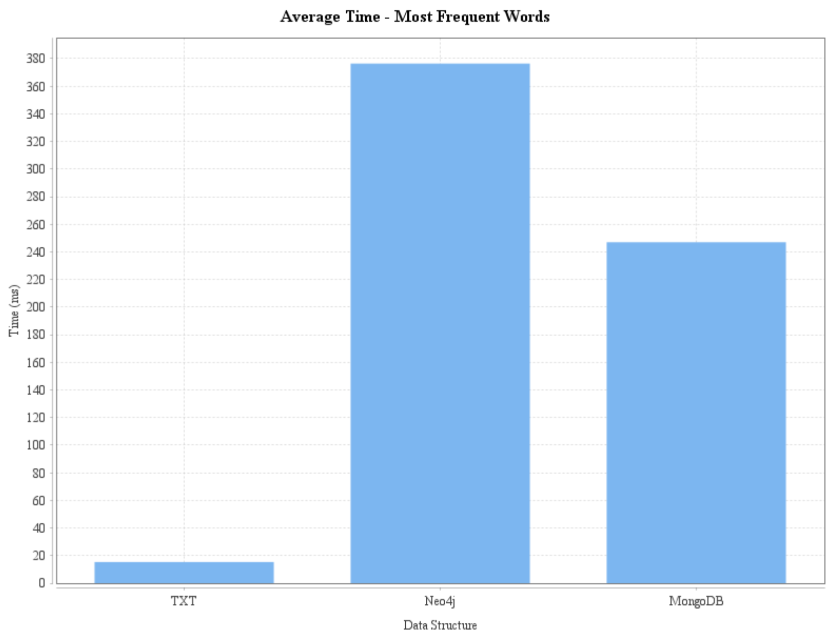

Figure 9: Average Time - Most Frequent Words Without Binary File

The figure 9 shows the average processing time in milliseconds (ms) required to search for the most frequent words for these storage systems: TXT, MongoDB, and Neo4j. The X-axis represents the evaluated storage systems, while the Y-axis indicates the average processing time in milliseconds. The purpose of this graph is to assess each system's efficiency in handling high-frequency terms, a critical aspect in search applications where certain terms are repeatedly used.

In terms of performance, the TXT system stands out as the most efficient, with the lowest average processing time among the evaluated systems. This result indicates that the TXT system is optimal for handling frequent words, maintaining a considerably low response time even during repetitive operations. This suggests that the TXT system is suitable for applications that require quick and efficient access to common terms.

MongoDB ranks second in terms of efficiency. Although its processing time is higher than that of the TXT system, it remains relatively stable and effective in handling high-frequency terms. MongoDB continues to demonstrate solid performance, though not as fast as the TXT system, making it a reliable option for search applications with a high term frequency.

On the other hand, Neo4j exhibits the worst performance in this graph. Its average processing time is significantly higher than the other systems, indicating less optimization for managing frequent words. This behavior suggests that Neo4j is not the most suitable system for search-intensive operations involving common terms due to its longer response time compared to TXT and MongoDB.

When comparing these results to the previous graph, which evaluated less frequent words and included the Binary File system, consistent patterns emerge regarding the relative capabilities of each system. In that graph, the TXT system had already proven to be the most efficient, followed by Neo4j, MongoDB, and the Binary File as the least efficient. The absence of the Binary File in this graph further reinforces TXT's position as the top-performing system for searches involving both frequent and infrequent words, standing out as a versatile and highly efficient solution. MongoDB maintains solid performance in both cases, while Neo4j continues to show significant limitations, making it the least suitable option for intensive search scenarios.

4.4. Crawler Performance

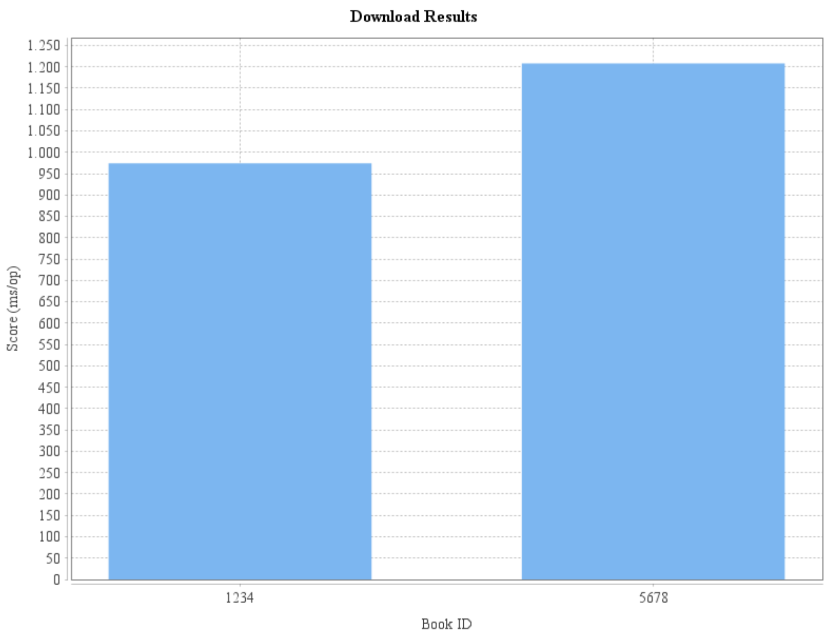

Figure 10: Download Results

The figure 10 presents the performance results of the Crawler module in terms of download time for specific books identified by their IDs, in this case, books with IDs 1734 and 5678. The X-axis shows the identifiers of the downloaded books, while the Y-axis indicates the download time measured in milliseconds per operation (ms/op). This graph aims to evaluate the efficiency of the download process based on the size of the books and the associated data structure.

Analyzing the results, it is observed that the book with ID 5678 requires significantly more download time compared to the book with ID 1734. This may be due to several factors: the file size, as the book with ID 5678 likely has a larger size compared to the book with ID 1734, which increases the time required to complete the download; the file structure, as books may vary in terms of format and structural complexity, which can influence the download speed; and network conditions, as download speed can be affected by the network conditions at the time of the operation, although this factor is generally less significant in controlled tests.

In general terms, the Crawler's performance shows a proportional response to file size, with download times increasing as the data volume of the book grows. This highlights the importance of implementing optimization techniques in the download module to reduce times in scenarios with large data volumes.

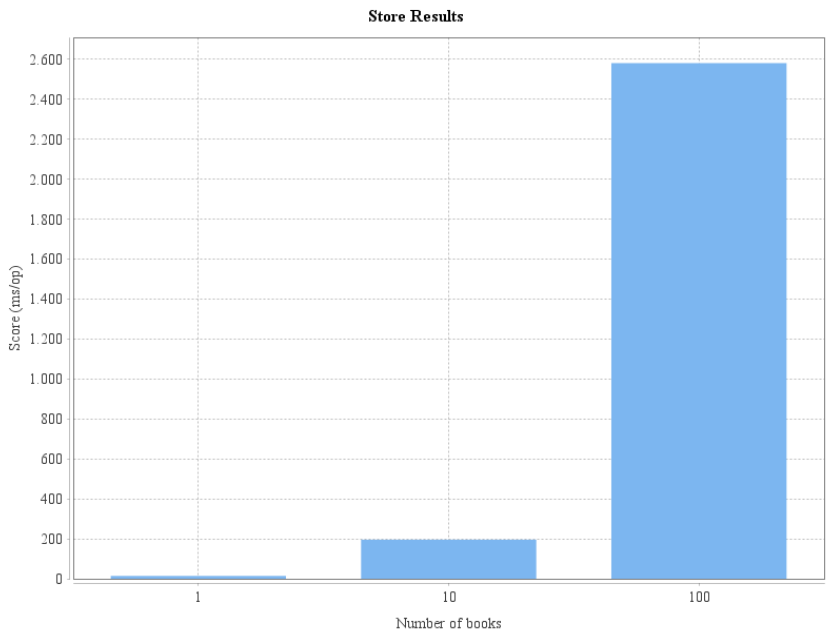

Figure 11: Store Results

The figure 11 analyzes the performance of the Crawler module in storing downloaded books, evaluating the average storage time per operation measured in milliseconds (ms/op). In this case, the X-axis represents the number of books stored (1, 10, and 100 books), while the Y-axis indicates the average storage time in milliseconds per operation.

The results show a clear upward trend in storage time as the number of books increases. When moving from storing 1 book to 10 books, the storage time grows but at a manageable rate. This suggests that the system can efficiently handle moderate amounts of data. However, when increasing to 100 books, the storage time rises significantly. This behavior can be attributed to the overhead associated with managing large volumes of data, such as updating storage structures and concurrent resource access.

This chart highlights the importance of optimizing storage operations, especially in scenarios with large data volumes. The exponential increase in times suggests that system performance could benefit from techniques such as parallelization, asynchronous resource access, or the use of more efficient storage structures to mitigate the impact of large-scale data. In summary, both charts highlight the Crawler module's ability to manage book download and storage processes but also identify key areas for future optimizations, particularly in system scalability when handling large volumes of data.

5. Comparison Between Python and Java in Search Engine Development

Aspect	**Python**	**Java**
Language	Easy to learn and use, ideal for rapid prototyping.	More robust, designed for large-scale applications.
Design Principles	Difficulties in fully adhering to SOLID due to Python's dynamic typing.	Solid implementation of SOLID principles and design patterns.
API	Backend based on FastAPI, flexible and fast for REST.	API based on Spark, tailored for scalability and Docker deployments.
Storage	Use of MongoDB, Neo4j, and text files.	Binary files were added and the previous ones were improved to inverted indexes and metadata more efficiently.
User Interface (UI)	It was not done for the previous stage.	Interactive interface developed in React for a better experience.
Performance	Adequate for prototypes but limited in scalability and performance.	Significant improvements in runtime.
Scalability	Faces challenges in large-scale applications.	More prepared for large data volumes.
Flexibility	Versatile for rapid changes.	More rigid design, but easily extensible without affecting the core.
Testing and Benchmarking	Done with Pytest and plugins like pytest-benchmark.	Use and JMH for detailed testing and benchmarking.
Deployment	Relies on the local environment.	Docker containers for portability and flexibility.

Table 1: Comparison Between Python and Java in Search Engine Development.

Python is ideal for initial phases and prototypes due to its simplicity and rapid development. However, it has clear limitations in terms of performance, scalability, and structure for large-scale projects.

On the other hand, Java excels in environments that demand high performance and scalability, being capable of handling more complex and optimized applications, as evident in the improvements achieved in the search engine version implemented with this language.

In relation to the comparison between different programming languages, the graphs turned out exactly the same. However, by making changes to the way data is stored in MongoDB, there was a slight improvement, and with that small improvement, MongoDB now generally outperforms the filesystem. In the first submission, their performance was very similar, but MongoDB was better. However, when it comes to operations measured in milliseconds, text files are still better.

6. Conclusion

The development of this specialized search engine for digital books from the Gutenberg Project, implemented in Java, has resulted in significant advancements in terms of performance, scalability, and efficient resource management. This project presents considerable improvements compared to the initial version developed in Python, demonstrating Java's superiority in handling large-scale applications with more robust modular structures. One of the most relevant conclusions is the comparison between the storage technologies used: MongoDB, Neo4j, and the file system (in text and binary formats). Throughout the experiments, it was demonstrated that each system has specific strengths and weaknesses depending on the context of use.

MongoDB stood out as the most balanced and scalable solution, especially for operations related to the inverted index. Its search times were consistently low, both in low and high data load scenarios, thanks to its query optimization capabilities. Additionally, its schema-less architecture allowed for efficient metadata handling, making it highly flexible for applications requiring complex data structures. Compared to the Python version, MongoDB maintained significantly more stable performance under high loads, highlighting the advantages of optimization in Java. However, when it comes to operations measured in milliseconds, text files are still better.

Text file storage proved to be the most efficient solution for searching both frequent and infrequent words, thanks to its simplicity and direct data access. However, its scalability was limited compared to MongoDB, particularly when dealing with large document volumes. The binary format, while innovative in its approach to saving space, exhibited deficiencies in performance, especially in search operations. These observations underline the need for further optimization in the handling of binary data if its large-scale implementation is to be considered.

Although its graph-based structure makes it ideal for complex queries involving relationships between entities, Neo4j performed worse than the other technologies for specific term searches. This reflects that Neo4j is not the most suitable option for intensive text search operations, though it could be valuable in scenarios where relationships between data are critical.

The transition from Python to Java not only improved the system's performance but also allowed for the more rigorous implementation of SOLID design principles. This resulted in a modular and extensible architecture, facilitating maintenance and the incorporation of new features. Additionally, the incorporation of a RESTful API and an intuitive user interface significantly enhanced the end-user experience, addressing a clear limitation in the initial Python version.

From a methodological perspective, stopword removal and binary encoding were key strategies for optimizing storage and queries. While these techniques introduced additional complexities in implementation, the results confirm their effectiveness in reducing data size and improving search performance.

Java's performance far surpassed that of the Python version in several key aspects. Regarding runtime, search and indexing processing times in Java were considerably lower, particularly in concurrent operations. In memory management, Java exhibited more efficient handling, enabling the processing of larger data volumes without bottlenecks. In terms of scalability, while the Python version experienced performance issues as the dataset size increased, the Java implementation maintained stable response times even with significant data volumes.

The results of this project demonstrate that the proper selection of storage technologies and solid architectural design are fundamental to developing scalable and efficient search engines. While MongoDB stands out as the most robust solution for inverted indexes and metadata management, the text file system remains a competitive option in terms of simplicity and performance in specific contexts. In the future, the implementation of semantic and contextual queries could add significant value, especially for complex searches. Additionally, the integration of tools like Docker will allow for greater portability and flexibility in deployments. These potential extensions will not only enhance the system's performance but also make it more adaptable to other domains beyond the Gutenberg Project.

This project not only validates the superiority of Java for high-performance applications but also establishes a solid foundation for future developments in the field of specialized search engines, providing valuable tools and results for researchers and developers.

7. Future Work

The future stages of this project will focus on addressing key scalability and deployment challenges to ensure the system can handle high query volumes effectively.

7.1. Enhancements for Scalability

To improve the scalability of the system, work will be directed towards optimizing the inverted index implementations to enhance query performance and resource efficiency. Additionally, modular design principles will be applied to enable deployment across multiple computers. This will facilitate the simultaneous operation of multiple crawlers, indexers, and query engines, ensuring robust support for distributed and parallel processing in large-scale environments.

7.2. Testing and Performance Evaluation

A significant effort will be devoted to establishing a comprehensive testing environment. This includes deploying and evaluating multiple instances of the system to assess its scalability under real-world conditions. By leveraging `nginx` as a load balancer, the system will distribute query loads effectively among instances. A dataset of 2,000 documents will be indexed to simulate operational conditions, followed by load testing to identify performance bottlenecks and evaluate system throughput. A key metric will be the determination of the maximum number of concurrent clients that the system can support.

7.3. Deployment

Deployment strategies will involve containerization using Docker, ensuring the system is modular and portable. A cluster will be composed of multiple search engine containers and a load balancer container to facilitate distributed operation. This architecture will enable dynamic scaling to accommodate varying workload demands, making the system resilient and adaptable to high-performance requirements.

This future work will establish the groundwork for a scalable, efficient, and deployable system capable of managing substantial query loads and large datasets in distributed environments.

References

[1] B. Barla Cambazoglu and R. Baeza-Yates, *Scalability Challenges in Web Search Engines*, Available: `https://www.researchgate.net/publication/226351869_Scalability_Challenges_in_Web_Search_Engines`.