Search Engine

Big Data

María Alonso León[1], Víctor Gil Bernal[2], Jacob Jażdżyk[3], Kimberly Casimiro Torres[4]
[1,2,3,4]Faculty of Computer Science
Las Palmas de Gran Canaria University, Spain

October 2024

Contents

ABSTRACT

This paper presents the design, development, and evaluation of a search engine tailored for large-scale book retrieval, specifically targeting the Project Gutenberg repository. The project focuses on comparing the performance of three distinct datamart technologies: MongoDB, Neo4j, and file system storage. Each technology is assessed based on its query response time, CPU usage, and scalability. The search engine is built using Python, with a FastAPI backend to provide RESTful API services. Keyword proximity queries and metadata retrieval are supported. The experiments reveal the strengths and weaknesses of each datamart technology in handling large datasets, offering insights into their suitability for large-scale text retrieval systems. Our findings suggest that file sytem provides the best balance between performance and scalability, making it ideal for managing extensive book collections.

KEYWORDS: Search Engine, MongoDB, Neo4j, File System, Project Gutenberg, Performance Comparison, Datamart, Scalability

1. Introduction

The exponential growth of digital information has amplified the need for efficient and scalable search engines that can manage and retrieve vast amounts of data. In this context, the search for relevant textual content, such as books, poses unique challenges due to the variability of the data structure, the size of the collections, and the user requirements. One of the most notable repositories of digital texts is Project Gutenberg, a free online library with over 60,000 books available for download. While the project provides basic search functionality, there is significant room for improvement in terms of the speed, relevance, and flexibility of search queries, especially when dealing with complex or large-scale queries.

To address this, the focus of this paper is on the development of a customized search engine designed specifically for the retrieval of books from Project Gutenberg. The main goal of the project is to experiment with different datamart technologies—namely MongoDB, Neo4j, and file system storage—and analyze their performance under a variety of conditions. The system architecture involves several key components, including a web crawler for data collection, an indexer, a query module, and a testing framework.

Several studies have focused on the development of search engines tailored to specific datasets, but few have directly addressed the challenges posed by large book repositories. Previous works have explored various aspects of information retrieval, including the use of different data storage methods such as [1], the implementation of search algorithms, and optimization techniques. However, to the best of our knowledge, there is limited research specifically comparing datamart technologies in the context of book search engines.

In this study, we compare three datamart technologies—MongoDB, Neo4j, and file system storage—to assess their performance in terms of query response time and scalability. By providing a diverse set of API endpoints, our search engine offers significant flexibility to users, enabling both simple and complex queries. The findings of this research will be valuable for understanding which data storage solutions are most suitable for handling large-scale text retrieval tasks in environments similar to Project Gutenberg.

The remainder of this paper is organized as follows: Section 2, Problem Statement, describes the key challenges and objectives addressed by this work, particularly in the context of book retrieval from Project Gutenberg. Section 3, Methodology, details the architecture and implementation of the search engine, including the design of the crawler, indexer, and query processor, as well as the technologies used (MongoDB, Neo4j, and file system). Section 4 provides an in-depth description of the experimental setup and the metrics used to evaluate system performance. Section 5 presents the results and discussion in the Conclusion, summarizing the performance comparison between the datamart technologies. Finally, Section 6 outlines potential Future Work, exploring directions for extending and optimizing the system. This paper contributes by offering a comparative analysis of different datamart technologies in the context of large-scale book retrieval and providing insights into their performance, scalability, and suitability for similar architectures

2. Problem Statement

The central issue addressed in this paper is the development of a Python-based search engine designed to retrieve books from Project Gutenberg, with a focus on comparing the performance of different datamart technologies. The primary objective is to determine which datamart solution—MongoDB, Neo4j, or file system storage—is best suited for managing and searching through large collections of books. Python was selected for this implementation due to its simplicity and the speed at which development can proceed. However, the trade-off is that Python presents certain limitations in comparison to languages like Java, especially regarding the application of design principles such as SOLID, which rely on stronger interface support. Despite these constraints, Python's rich ecosystem and rapid prototyping capabilities make it an ideal choice for this experiment.

The three datamart technologies explored in this study—MongoDB, Neo4j, and a file system—were chosen for their distinct approaches to handling data and their potential to enhance search performance. MongoDB, a NoSQL database, is optimized for handling large amounts of unstructured data, making it suitable for managing metadata and handling searches with a flexible schema. Neo4j, a graph database, excels at managing relationships between entities, such as authors and genres, which can be advantageous for certain types of queries. The file system, while simpler, can provide fast direct access to text data, particularly when searching through raw text without complex query logic. This study aims to provide a thorough evaluation of these technologies in terms of query performance, CPU consumption, and scalability, offering insights into their strengths and weaknesses in this context.

A particular complexity of this project lies in the distinction between book metadata and the inverted index used for search queries. Different datamart technologies may be employed for these two components; for instance, MongoDB may be used for metadata management, while the file system might be more appropriate for indexing and searching text content. This differentiation adds an additional layer of architectural complexity, but it also opens the door to optimizing the system for both read and write operations, which could lead to significant performance improvements.

The study is designed with the expectation that some datamart solutions may perform well with smaller datasets but struggle to maintain efficiency as the size of the dataset increases. By conducting a comprehensive set of tests, we aim to determine how each technology scales and which solution offers the most robust performance under various conditions. This will provide valuable information about the scalability and efficiency of different datamart technologies in the context of large book collections like Project Gutenberg.

Scalability is a crucial aspect of this project, as modern search engines must handle ever-increasing datasets. Ensuring that the chosen technology can accommodate growth without sacrificing performance is essential for the long-term success of this search engine. Moreover, the flexibility of the system is designed to allow for easy adaptation to other text-based datasets beyond Project Gutenberg, extending its potential applications. Despite Python's lack of some advanced language features, its simplicity and versatility make it a practical choice for this type of project. The lessons learned from this implementation could also inform future developments using more specialized languages.

Ultimately, this project not only addresses the technical challenges of building a scalable and efficient search engine, but also contributes to the broader understanding of how different datamart technologies perform in real-world scenarios. By identifying the strengths of each solution, this research will provide actionable insights for developers and researchers working with large-scale text datasets, offering a foundation for future work in optimizing data storage and retrieval systems.

3. Methodology

In this section, we will structure the methodology starting with an explanation of the SOLID principles, which are essential for ensuring a scalable, maintainable, and robust design. Each principle will be described and applied to specific components of the search engine project. After discussing these principles, we will continue with the detailed breakdown of the different components of the system, such as the crawler, indexer, query engine, and tests.

Single Responsibility Principle (SRP) Each component has a well-defined responsibility. For example, the `Crawler` module is solely responsible for extracting and downloading books from Project Gutenberg. The `Indexer` focuses only on creating and maintaining the inverted index and metadata structure.

Open/Closed Principle (OCP) The system is open to extension but closed to modification. For instance, new search functionalities, such as proximity or fuzzy search, can be added to the `Query Engine` without altering the existing code, by implementing new classes or modules.

However, fully adhering to certain SOLID principles—specifically the **Liskov Substitution Principle (LSP)**, **Interface Segregation Principle (ISP)**, and **Dependency Inversion Principle (DIP)**, has been challenging due to Python's limitations.

The LSP ensures that subclasses can replace their parent classes without altering program behavior, but Python's dynamic typing makes strict enforcement difficult. The ISP advocates for smaller, more specific interfaces, which is harder to achieve in Python due to the lack of an efficient built-in interface system. Similarly, the DIP recommends that high-level modules should depend on abstractions, not on concrete implementations, but Python's lack of static typing complicates this principle's implementation.

3.1. Structure of Architecture

After the SOLID principles, the architecture is divided into the following modules:

3.1.1 Crawler

The Crawler is responsible for periodically downloading books from Project Gutenberg, serving as a critical component to feed the search engine with updated content. The crawler was implemented in Python, utilizing the ‘requests‘ and ‘beautifulsoup4‘ libraries for web navigation and content parsing. The primary objective of the crawler is to extract both the text of the books and their associated metadata (author, publication date, language, etc.) for further processing and indexing. The crawler was designed to be scalable and efficient, allowing for continuous downloading of new books. Its operation follows these steps:

- **URL Discovery**: The crawler periodically visits the Project Gutenberg website, identifying book download links in full-text format.

- **Content Extraction**: Once a book's download link is identified, the crawler proceeds to download the text content with the associated metadata. The metadata includes relevant information such as the author, publication year, and book language. The book contents are stored in a local file system repository, the Datalake.

- **Storage**: Once the books are downloaded, they are stored in the datalake, which is a structure composed of text files and directories. For each day, a folder is created, named after the date, and the books are stored there with the identifier "book_n", where "n" corresponds to the book stored on that day.

3.1.2 Indexer

The Indexer module is designed to efficiently process data across three distinct phases, ultimately transforming raw input into structured, accessible information. It operates as an independent task, continuously monitoring the datalake for updates, which include newly downloaded books from the crawler module. Upon the addition of a new book to the datalake, the module initiates the first phase, where the book is read and converted into a dictionary. In this dictionary, the key corresponds to the book's title, and the value represents the textual content within the file.

In the second phase, the module is divided into two distinct sections. One section focuses on constructing an inverted index, a data structure that stores individual words from the text and associates them with the files in which they appear along with the total number of appearances. The second section handles the processing of metadata, extracting key details such as the author, language, and release date of the book.

- **Inverted Index**: Its purpose is to facilitate fast searches by linking each word to its respective document.
  - *How it works*: A set of stopwords is initialized to avoid indexing common words (e.g., determiners, pronouns). The text is then tokenized and converted to lowercase to ensure uniformity. As the module iterates through the text, it checks if each word is a stopword. If it is not, the word is added to the inverted index along with its position in the text and its total occurrences.

- **Metadata**:
  - *Implementation of the metadata processing*: Using regular expressions (regex), we initialize four regex patterns to search for metadata values in every line of the text. If some of these values are not found, the corresponding position in the dictionary is filled with the phrase "Not found".

The third phase involves the storage of the processed information within the datamart technologies utilized in this project. Each module handles the data storage according to its specific method, ensuring that the information is organized and readily accessible for extraction whenever needed.

3.1.3 Datamart Technologies

This section outlines the different datamart technologies employed in the system to store and manage the data. Each technology serves a distinct purpose, optimized for specific aspects of the data structure and retrieval process. Below, we describe the key technologies utilized:

- **MongoDB**: Stores metadata and inverted index for each book. It also handles more flexible, schema-less data formats.

- **Neo4j**: Captures relationships between books, authors, and genres, enabling complex queries about related works or similar authors.

- **File System**: The raw book content is stored in the file system, with the inverted index linking search terms directly to these files for fast retrieval.

3.2. Query Processor

The Query Module is responsible for handling the search functionalities across the system, divided into two main components:

- **Word-based queries**: These handle searches for words or phrases using an inverted index mechanism.

- **Metadata queries**: These manage searches based on metadata attributes like author, date, and language.

Each section interacts with distinct datamarts (such as Neo4j, MongoDB, or simple text files) and is subsequently integrated through a REST API.

3.2.1 Word-based Queries (Inverted Index)

The word-based queries utilize an inverted index to efficiently search for occurrences of words across documents. These queries allow for the retrieval of word positions, context, and document frequency, which are vital for building a robust search engine. The functions below showcase how the word-based search is structured.

Key Functions:

- `search_word_in_all_documents(word, file_path, document_folder)`: This function searches for a specific word across all documents in a given directory. It returns a list of documents where the word appears, its frequency, along with the positions within those documents. It handles case-insensitivity by converting everything to lowercase.

- `search_word_with_context(word, file_path, document_folder)`: This function searches for the given word across all text documents in a specified directory. It reports the positions where the word appears and the frequency of occurrences.

3.2.2 Metadata Queries

The metadata queries provide a different type of search functionality by allowing the user to filter documents based on attributes like author, date, and language. These queries extract structured metadata associated with documents, stored in the datamarts.

Key Functions:

- `search_by_author(file_path, author)`: This function searches for documents authored by specific individuals.
- `search_by_date(file_path, dates)`: This function searches for documents by their publication date.
- `search_by_language(file_path, language)`: This function retrieves documents based on the language in which they are written.

3.3. API

The API module has been meticulously designed to work seamlessly with all three methods of index storage: Neo4j, MongoDB, and file-based storage. It is built using FastAPI, which provides a robust framework for creating various endpoints that interact with different data sources. The API runs on port 8000 and connects to a local database. Its primary function is to ensure the correct retrieval of data from each of the databases—Neo4j, MongoDB, and file-based—while also offering advanced querying capabilities.

It is important to highlight that the API's endpoints correspond directly to the functions implemented in the query module, maintaining consistency across the system.

The design of this module emphasizes flexibility, enabling the same queries to be executed across all three storage systems. This adaptability is achieved through the use of helper functions, which abstract the specifics of each storage mechanism, ensuring smooth integration regardless of the underlying technology.

3.4. Tests

In software development, testing is a critical part of ensuring the reliability, performance, and correctness of a system. Well-designed tests help identify bugs, prevent regressions, and verify that the software behaves as expected under various conditions. They are especially important in complex systems like search engines, where performance, scalability, and accuracy are key.

For this project, where we are comparing different datamarts (File System, MongoDB, and Neo4j), testing becomes even more crucial. We aim to determine which data storage system performs best in terms of search speed, scalability, and efficiency. By using automated tests, we can benchmark different aspects of the system and objectively evaluate their performance across varying data sizes.

In this context, we use the Pytest library with the pytest-benchmark plugin to measure and compare the performance of the different datamart implementations. The benchmark results give us concrete data on how the search functions behave under different conditions, providing valuable insights into which system is optimal for handling large-scale search queries.

3.4.1 Multiple Books

This test evaluates the search performance for different datamarts, specifically File System, MongoDB, and Neo4j. The test is designed to check how long it takes for each system to search for a randomly chosen word across a set of documents. The word choices are from a predefined list of common terms, such as "freedom", "rights", "constitution", "democracy", and "law".

Setup: The test uses different setup functions for each datamart (`setup_books_file`, `setup_books_mongo`, and `setup_books_neo4j`). These functions initialize the system with a number of books, ranging from 10 to 1,000. Each book contains a predefined string about topics such as freedom, rights, and democracy.

For File System, this setup involves creating text files with predefined content. Each file represents a book, and the number of books (count) is determined by the parameter passed to the test.

Benchmarking:

Random Search Word: A random word from the `search_words` list is chosen for each test. This ensures variability in the queries, mimicking real-world search scenarios where users might search for different terms.

Benchmark Fixture: The benchmark fixture provided by Pytest helps measure the time taken to execute the search function (`search_fs`, `search_mongo`, or `search_neo4j`) for each query. The benchmark results are recorded and can later be analyzed to assess performance across different datasets and datamarts.

Count Parameter: The `count` parameter is critical as it specifies the number of books (documents) in the dataset. This allows the tests to assess performance at varying scales, from small datasets (10 books) to large ones (1,000 books). By varying the number of books, we can evaluate how the datamarts handle different dataset sizes and how they scale as the dataset grows.

Assertions: The test asserts that the result returned by the search function is not `None`. A `None` result would indicate that the search did not return any matches, which is considered a failure for the test. This ensures that the search engine is functioning as expected and that valid results are returned for each query.

Additional Considerations *Real-world Simulation:*
By varying the dataset sizes (`book_counts = [10, 50, 100, 500, 1000]`), this test simulates real-world scenarios where the volume of data can range from small to large. This enables a comprehensive performance evaluation of the search engine across different data loads.

Test Extensibility:
The test can be easily extended to include other search parameters, such as different types of queries (e.g., phrase search), or to compare additional datamarts or data structures. This flexibility is important as it allows for more nuanced testing as the system evolves.

3.4.2 Frequency

This test reads the content of books stored in text format, constructs an inverted index from the text files, and processes it to identify the most frequent and least frequent words. The inverted index is key to efficient searching, as it maps each word to the documents in which it appears, allowing for fast lookups. Once the words are identified, as it was mentioned before, the test uses the pytest benchmarking tool to measure the time it takes for each datamart (File System, MongoDB, and Neo4j) to search for both frequent and infrequent words.

The rationale behind choosing both the most and least frequent words is to observe how the search engines perform under different conditions. For frequent words, we expect the search operations to require more processing since these words appear in multiple documents. Conversely, for infrequent words, the search may involve fewer documents, which could potentially make the search faster. The goal is to compare how well each datamart can handle these variations in search complexity.

The structure of the test involves two main phases:

- **Word Frequency Benchmarking**: This involves querying words that appear most frequently across all documents.
- **Word Infrequency Benchmarking**: This phase tests words that appear the least, helping to understand how each system handles sparse data.

Code Structure and Error Handling
Each test is divided into three main sections:

- **File System Query**: This uses a local file-based inverted index to retrieve documents containing the search word.
- **MongoDB Query**: This connects to a MongoDB instance to execute a similar search.
- **Neo4j Query**: Though newly added, Neo4j uses graph-based queries to search for words and their occurrences.

Errors are handled in a straightforward manner. A failed search is considered a failure if the result returns None, meaning no relevant documents were found, or if there is a significant performance degradation, such as an excessively high query time.

By implementing this testing structure, we ensure that the performance and scalability of each datamart are evaluated under realistic conditions, providing insights into their effectiveness in handling both high-frequency and low-frequency queries.

3.4.3 Process Inverted Index and Metadata

This test evaluates the performance of creating and processing both inverted indexes and metadata for a collection of documents. Unlike other tests, this script does not implement a specific setup process; instead, it processes documents directly from a folder and utilizes fixtures to manage the data flow.

Document Processing: The test begins by reading all `.txt` documents from a specified folder. The function `process_documents_from_folder` processes each document by reading its content and storing it in a dictionary, where the key is the filename and the value is the document's content. This dictionary is essential for simulating a document collection used in later indexing processes.

Fixtures: Several fixtures are defined to manage the document dataset and the associated indexes:

- `books:` This fixture processes the documents from the folder by calling `process_documents_from_folder` and returns a dictionary representing the document collection.
- `invertedIndex (ii):` This fixture creates an inverted index for the document collection using the `invertedIndex` class.
- `metadataIndex (md):` This fixture processes the metadata for the document collection using the `metadataIndex` class.

Benchmarking: The main focus of the tests is benchmarking the performance of two distinct operations:

- `Inverted Index Creation:` The test measures the time required to create an inverted index for the document collection. This is a key performance metric for evaluating how efficiently the system can process and organize document terms.

- `Metadata Index Creation:` Another critical operation is the processing of document metadata. The benchmark measures how long it takes to extract and organize metadata fields, which are essential for improving the precision of search queries and results.

Storage Tests: These tests measure the performance of storing both the inverted index and metadata into different storage systems:

- `Text File Storage:` Stores the inverted index or metadata in a local file.

- `MongoDB Storage:` Stores the data in a MongoDB database.

- `Neo4J Storage:` Stores the data in a Neo4J graph database, which is particularly useful for graph-based queries.

Key Differences from Other Tests: Unlike other more comprehensive tests, this script does not implement a specific setup phase to prepare the environment or data. Instead, the focus is on directly processing the documents and benchmarking the speed of index creation. Additionally, error handling is minimal, assuming that all documents can be read and processed without issue.

By focusing on core indexing and metadata processing, this test provides an efficient way to evaluate how well the system handles document ingestion and organization, without the overhead of storage or extensive configuration setups.

4. Experiments

In this section, we present the experimental results obtained from benchmarking various aspects of the system. The performance of different data storage technologies—MongoDB, Neo4j, and the file system—is analyzed in terms of execution times and scalability across a series of operations. The results provide valuable insights into how each technology performs under different workloads, such as storing and processing metadata and inverted indexes, querying for both frequent and infrequent words, and scaling with increasing dataset sizes. These experiments serve as a foundation for understanding the strengths and limitations of each storage system, ultimately guiding future improvements and optimizations.

4.1. Time Results

The following figures show the benchmark results for the three datamart technologies—MongoDB, Neo4j, and file system (txt)—when processing the inverted index (II) and metadata (MD). These results help us analyze the execution time required for storing and processing the information in the system.

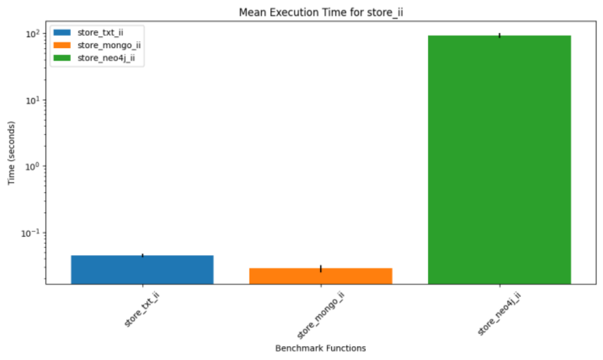

Figure 1: Mean Execution Time for Storing Inverted Index (store_ii)

Explanation of Figure 1: This figure shows the mean execution time for storing the inverted index across the three storage systems:

- **store_txt_ii**: The file system performed well, maintaining a competitive execution time.

- **store_mongo_ii**: MongoDB performed the fastest, with an average time significantly lower than file system and Neo4j

- **store_neo4j_ii**: Neo4j had the highest execution time, taking considerably longer than both MongoDB and the file system to store the inverted index.

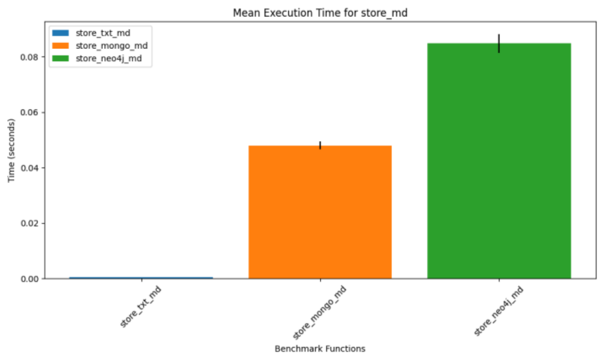

Figure 2: Mean Execution Time for Storing Metadata (store_md)

Explanation of Figure 2: This figure illustrates the mean execution time for storing metadata (author, language, etc.):

- **store_txt_md**: In this case, the file system displayed the lowest execution time, indicating that it is the most efficient for storing metadata.

- **store_mongo_md**: MongoDB took longer than the file system but still outperformed Neo4j.

- **store_neo4j_md**: Neo4j required the most time, particularly for storing metadata, which highlights a potential bottleneck for metadata-intensive tasks.

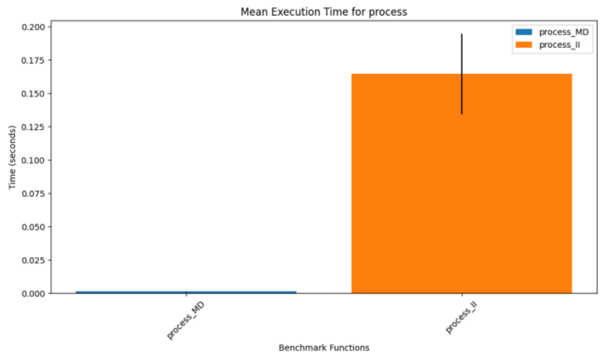

Figure 3: Mean Execution Time for Processing Metadata and Inverted Index (process)

Explanation of Figure 3: This figure compares the processing time for metadata (MD) and inverted index (II) tasks:

- **process_MD**: The time required to process metadata was minimal compared to the inverted index processing, which shows that metadata handling is more efficient.

- **process_II**: Processing the inverted index took considerably longer, which is expected due to the complexity of indexing large text files and linking each word to the corresponding documents.

Overall Analysis:

These results indicate that the file system outperforms MongoDB and Neo4j for metadata, but the fastest in the inverted index is MongoDB. While MongoDB provides a good trade-off between flexibility and performance. Neo4j's execution times were the highest across all tests, suggesting that it might not be the optimal choice for tasks involving frequent data storage or retrieval operations.

4.2. Least Frequent Words

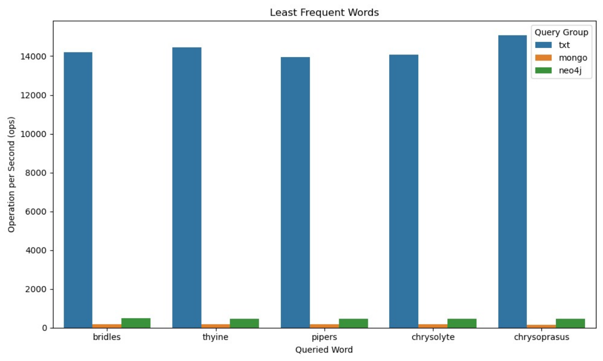

Figure 4: Mean Time Comparison by Group and Word

Explanation of Figure 4: This bar chart compares the operations per second (OPS) for the least frequent words in the dataset across three different query systems: txt (blue), MongoDB (orange), and Neo4j (green). The horizontal axis lists the queried words, while the vertical axis represents the OPS.

- **txt Group:** The blue bars indicate the performance of the file system, which demonstrates a high and consistent OPS of around 14,000 for all least frequent words. This performance is significantly better than MongoDB and Neo4j, highlighting the efficiency of txt in processing rare terms. The uniformity of the results suggests that txt handles both frequent and rare terms with the same level of optimization.

- **MongoDB Group:** The orange bars show that MongoDB has a much lower OPS, averaging around 1,000 to 2,000 operations per second. This indicates that MongoDB struggles with rare words, likely due to its indexing structure, which may not be as optimized for handling sparse data compared to frequent terms.

- **Neo4j Group:** The green bars, representing Neo4j, also display lower OPS, hovering between 500 and 1,000 operations per second. Neo4j performs similarly to MongoDB for rare words, suggesting that its graph-based architecture, while powerful for relational queries, may not be as efficient for handling infrequent, isolated terms in text searches.

Overall Analysis: The file system clearly outperforms both MongoDB and Neo4j in handling rare words, maintaining a consistently high OPS. The performance of MongoDB and Neo4j, while lower, indicates that these systems may be less suitable for handling sparse or infrequent terms in large datasets. Txt's superior performance for rare words suggests that it may be the optimal choice for systems that need to handle a wide range of word frequencies efficiently.

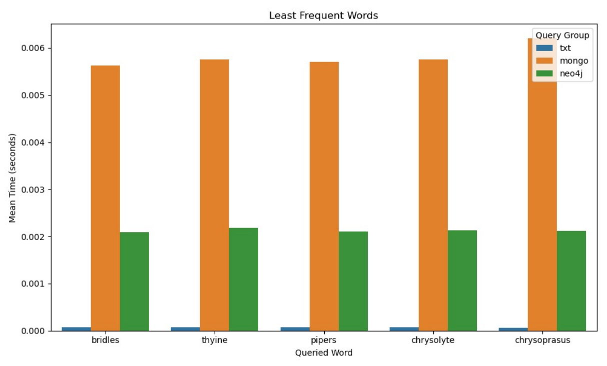

Figure 5: Mean Query Time for Least Frequent Words

Explanation of Figure 5: This bar chart compares the mean query times (in seconds) for the least frequent words across three query systems: txt (blue), MongoDB (orange), and Neo4j (green). The horizontal axis lists the least frequent queried words, while the vertical axis shows the mean query time.

- **txt Group:** The blue bars show that the file system consistently has the lowest query times for all the least frequent words, with times close to zero. This performance suggests that txt is highly efficient for both frequent and infrequent words, maintaining its ability to retrieve data quickly even when processing rare terms.

- **MongoDB Group:** The orange bars represent MongoDB, which consistently shows the highest query times, around 0.005 to 0.006 seconds for all least frequent words. This indicates that MongoDB struggles with rare terms, likely due to inefficiencies in indexing and retrieval when handling less common entries in the dataset.

- **Neo4j Group:** The green bars, representing Neo4j, show better performance than MongoDB but still exhibit higher query times compared to txt. Neo4j's performance, averaging between 0.002 and 0.003 seconds, suggests that while it is faster than MongoDB, its graph-based architecture does not offer significant advantages for isolated rare terms.

Overall Analysis: This graph highlights the substantial differences in how each system handles rare terms. The file system is the most efficient, maintaining minimal query times across all words, while MongoDB shows the most significant delays, making it less ideal for processing rare terms. Neo4j performs moderately well but still cannot match the performance of txt for these infrequent queries.

4.3. Most Frequent Words

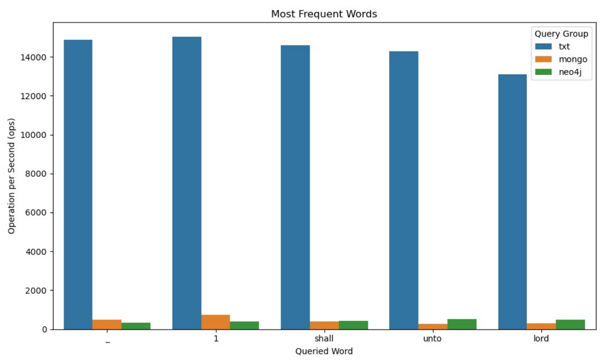

Figure 6: Operations per Second (OPS) for Most Frequent Words

Explanation of Figure 6: This bar chart presents the operations per second (OPS) for the most frequent words in the dataset, comparing the performance of three query systems: txt (blue), MongoDB (orange), and Neo4j (green). The horizontal axis lists the queried words, and the vertical axis represents the OPS for each system.

- **txt Group:** The file system, represented by the blue bars, continues to show a consistent OPS of around 14,000 across all the frequent words, demonstrating that it can efficiently process high-frequency terms. This result is in line with its performance on rare words, indicating that the file system is equally optimized for both frequent and infrequent queries.

- **MongoDB Group:** MongoDB, represented by the orange bars, shows a slight improvement compared to its performance with rare words. The OPS values range from 1,000 to 2,500, indicating that MongoDB handles frequent words more efficiently than rare ones. However, it still lags significantly behind the file system, suggesting room for optimization in terms of processing speed for frequent terms.

- **Neo4j Group:** The green bars representing Neo4j show an OPS between 500 and 1,000, which is consistent with its performance for rare words. This further indicates that Neo4j's architecture, though well-suited for graph-based relationships, does not offer significant performance gains when handling frequent terms in text-based queries.

Overall Analysis: For frequent words, the file system continues to demonstrate superior performance, maintaining a high OPS compared to MongoDB and Neo4j. While MongoDB performs slightly better with frequent words than with rare ones, it still falls short in comparison to txt. Neo4j's OPS remains consistent but relatively low, further emphasizing that it may not be the best choice for high-speed text retrieval tasks, especially when handling frequent terms.

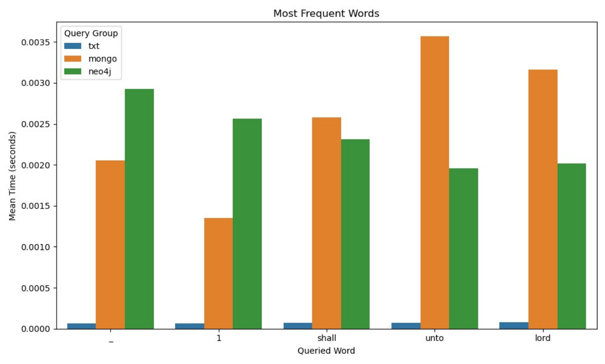

Figure 7: Mean Query Time for Most Frequent Words

Explanation of Figure 7: This bar chart shows the mean query times for the most frequent words across the txt (blue), MongoDB (orange), and Neo4j (green) systems. The horizontal axis lists the most frequent words, while the vertical axis represents the mean query time in seconds.

- **txt Group:** As in the previous graph, the blue bars for txt show query times near zero across all frequent words, reaffirming the system's efficiency in handling frequent terms. The minimal query time suggests that txt's architecture is well-optimized for both common and rare terms.

- **MongoDB Group:** The orange bars show that MongoDB's query times are relatively high, especially for words like *"unto"* and *"lord"* where query times approach 0.003 seconds. MongoDB's performance shows that while it may handle frequent words better than rare ones, its performance is still notably slower than txt.

- **Neo4j Group:** The green bars represent Neo4j, showing query times between 0.002 and 0.003 seconds, similar to its performance for rare words. Neo4j's results here indicate that it does not significantly improve its performance for frequent terms, and while it outperforms MongoDB in some cases, it remains slower than txt.

Overall Analysis: This graph reinforces the conclusion that the file system is the most efficient, providing near-instantaneous query results for frequent terms. While MongoDB and Neo4j show improved performance for frequent words compared to rare ones, they still lag behind txt in terms of speed. Neo4j offers a slight performance advantage over MongoDB, but both systems would benefit from further optimization when handling frequent queries.

4.4. Search Scalability

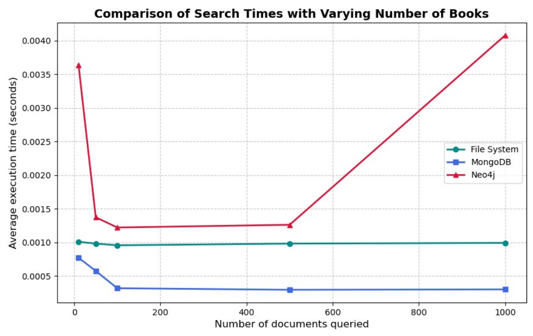

Figure 8: Comparison of Search Times with Varying Number of Books

Explanation of Figure 8: This figure presents a comparison of the average search times for three different data storage systems—File System, MongoDB, and Neo4j—across varying numbers of documents (books) queried. The horizontal axis represents the number of documents queried, while the vertical axis shows the average execution time in seconds.

- **File System:** Represented by the green line, the File System demonstrates relatively consistent query times, remaining nearly flat as the number of documents increases. This indicates that the File System's performance is fairly stable and predictable. However, its search time is higher than MongoDB, indicating it may not be the most efficient for large-scale datasets. Despite its linear trend, it shows no significant improvement or degradation in performance as more books are queried.

- **MongoDB:** The blue line, representing MongoDB, shows the lowest search times across all dataset sizes. It also reveals a slight decrease in search time as the number of books increases, indicating that MongoDB benefits from its indexing and query optimization mechanisms. This behavior suggests that MongoDB is highly efficient at managing large datasets, likely due to its ability to quickly locate relevant documents through its well-optimized indexing structures.

- **Neo4j:** Neo4j, represented by the red line, shows significantly different behavior compared to the other two systems. Initially, Neo4j starts with higher search times, but these times decrease quickly as the number of books increases, stabilizing for mid-sized datasets. However, as the dataset grows to around 1000 documents, the query time for Neo4j increases dramatically, suggesting that its performance scales poorly with larger datasets. This spike in query time may be attributed to the graph structure of Neo4j, which might require traversing more relationships as the dataset grows, making it less efficient for querying large collections of documents.

Overall Analysis: The results show that while MongoDB outperforms both File System and Neo4j in terms of speed, it also demonstrates a slight improvement as more documents are queried, making it an ideal solution for scaling with large datasets. The File System, though stable, lacks the query speed advantages of MongoDB, and Neo4j, while useful for smaller datasets, shows significant performance issues as the number of documents reaches higher values. These findings highlight the importance of selecting the right database technology depending on the scale of the dataset and the expected query performance.

5. Conclusion

This project presented the design, implementation, and evaluation of a custom search engine tailored for large-scale text retrieval, specifically targeting the Project Gutenberg repository. By comparing the performance of three different datamart technologies—MongoDB, Neo4j, and a file system—we were able to analyze their respective strengths and weaknesses in handling search queries across datasets of varying sizes.

MongoDB demonstrated clear advantages in scalability and consistency across all search-related performance metrics. Its indexing and query optimization mechanisms provided low search times, both in average and worst-case scenarios, regardless of the dataset size. MongoDB's schema-less architecture also proved advantageous for managing metadata, allowing flexible and efficient handling of book attributes such as author and publication date.

Neo4j, while showing strength in managing complex relationships, such as author collaborations, struggled with higher execution times in search operations. This graph-based database is best suited for scenarios requiring complex relationship queries rather than high-speed text retrieval.

Although the file system exhibited longer execution times when handling large datasets, its simplicity and fast performance make it a strong candidate for the project. Furthermore, it has demonstrated a high degree of stability.

In summary, while MongoDB proved to be a balanced and robust solution for large-scale book retrieval systems, offering excellent performance and scalability across varying data loads, the simplicity and operational efficiency of the file system make it the preferred choice for future implementations of similar search engines. The file system's superior performance in certain operational contexts further underscores its suitability for such use cases.

6. Future Work

The current implementation lays a solid foundation for efficient search and indexing across different data storage systems. However, there are several avenues for improvement and extension that we aim to explore in future iterations of the system, each of which is focused on optimizing performance, usability, and flexibility.

Java Implementation One of the primary areas for future work is the re-implementation of the system in Java. While Python has allowed for rapid development and ease of use, it presents limitations, particularly in terms of performance and adherence to software design principles. Java offers significant advantages for large-scale systems, including faster execution times, better memory management, and stronger support for object-oriented design. More specifically, Java's support for SOLID principles—such as clear interface definitions, strong typing, and better modularity—will enable us to develop more scalable and maintainable code. Although we have adhered to SOLID principles in the Python version to the best extent possible, Python's lack of an efficient built-in interface system limits its flexibility. Transitioning to Java will allow us to overcome these limitations, leading to a more robust and extensible architecture.

Docker Integration To enhance the system's deployment and testing flexibility, we plan to integrate Docker. By containerizing the system, we can ensure consistent behavior across various environments, which is essential for large-scale distributed systems. Docker will allow us to encapsulate the entire application, including its dependencies, in a standardized unit that can be easily deployed across different platforms. This will streamline not only the deployment process but also facilitate easier scaling, as Docker supports rapid provisioning of containers across multiple servers. Moreover, Docker will allow for simplified testing, as isolated containers enable us to evaluate system components in different environments with minimal configuration effort.

UI Improving the user experience remains a high priority. We are planning to build it an UI using the React framework. Future developments will focus on expanding the metadata search options and designing an intuitive, feature-rich interface. By introducing additional search filters for metadata, such as author, publication date, and language, we will provide users with greater control over the search process. In parallel, we will optimize the search functionalities to ensure faster, more precise results. The planned UI design will aim to create a smoother navigation experience and a more interactive interface, significantly increasing the overall usability of the system.

Context Query Improvement Inspired by the contextual search capabilities of major search engines like Google, we aim to enhance the context query functionality of the system. Currently, the system retrieves search results based on exact matches within the indexed data. In future iterations, we intend to implement a more sophisticated context-aware search, which will allow users to search for terms in a manner that is more natural and user-friendly. This feature will improve search accuracy by considering the semantic context in which words appear, providing users with more relevant results. Additionally, by refining the presentation of search results to display relevant context around query terms, the system will offer a more intelligent and useful search experience.

7. References

References

[1] Mehul A. ShahAuthors Dimitris Tsirogiannis, Stavros Harizopoulos. Analyzing the energy efficiency of a database server. *Publication history*, 2010.