Prototype Search Engine for Project Gutenberg

Authors: Daniel Nosek, Lennart Schega, Domen Kac, Nico Brockmeyer, Anna Sowińska

Date: 07.10.2025

University: ULPGC

Course Name: 40386 Big Data

Group Name: Lennart

Repository: https://github.com/nosekdan/lennart-stage-1

Prototype Search Engine for Project Gutenberg Texts

Abstract

In the first stage of the Big Data course project we implemented a simple search engine prototype. The system consists of three main components, a crawler that downloads book texts from Project Gutenberg, an indexer that builds an inverted index, and a query engine that enables keyword search. We developed and compared two different data structures for the inverted index, the filesystem database and MongoDB. This paper presents the system architecture, implementation details, and experimental results.

1. Introduction

Search engines are an essential tool for managing and retrieving information from large collections of documents. A core concept enabling efficient full-text search is the inverted index, which maps terms to the documents in which they occur. Our project was to design and implement a prototype search engine capable of collecting textual data, building an inverted index, and answering basic user queries. Furthermore, we were required to implement two alternative data structures for the index in order to benchmark their performance and scalability.

The main objectives of Stage 1 are:

- To design a data ingestion pipeline (crawler + datalake).
- To implement an indexer that builds inverted indexes in the filesystem database and MongoDB.
- To provide a minimal query engine for metadata and Boolean full-text search.
- To benchmark the two indexing strategies with respect to indexing time, query latency, memory usage, and scalability.
- To analyze the results and discuss trade-offs between the two approaches.

2. Background

A key concept in information retrieval systems is the inverted index, a data structure that maps terms to the documents in which they occur. Unlike a traditional forward index, which stores documents and their corresponding terms, the inverted index enables fast lookup of all documents that contain a given word. This structure forms the backbone of most modern search engines, from small-scale prototypes to large-scale web search systems.

For the implementation of an inverted index, different storage solutions can be used. The filesystem database represents a lightweight relational database stored as a single file. It is widely used for small- to medium-scale applications due to its simplicity, portability, and ease of setup. However, as the dataset grows, the filesystem database may suffer from performance bottlenecks since it is not designed for distributed or large-scale workloads.

In contrast, MongoDB is a NoSQL document store designed to scale horizontally across multiple machines. It stores data in flexible JSON-like documents, making it well-suited for handling unstructured or semi-structured text. MongoDB supports distributed indexing and querying, which provides better performance and scalability in Big Data contexts. Previous studies in database performance comparisons have highlighted that while relational systems are efficient for structured queries, document stores often outperform them in full-text search scenarios and large-scale indexing tasks.

Our project builds on these established foundations by implementing and benchmarking two versions of the inverted index, in the filesystem database and in MongoDB.

3. System Architecture

The architecture of the prototype search engine follows a simplified big data pipeline consisting of three layers:

- Datalake, serves as the raw storage, containing the downloaded plain-text files from Project Gutenberg. Each book is stored together with title, author, language. This layer ensures that the original data is preserved and can be reprocessed if needed.

- Datamart, contains structured and queryable data derived from the datalake. It includes both the metadata stored in MongoDB and the filesystem database and the inverted indexes used for full-text search.

- Control layer, a minimal orchestration mechanism that coordinates the pipeline. It keeps track of downloaded and indexed books, ensuring that data is processed once and avoiding duplicates. In this stage, it mainly validates that the crawler, indexer, and query engine can operate together in a consistent workflow.

Design of our prototype search engine. Figure 1 shows the main components and their interactions.

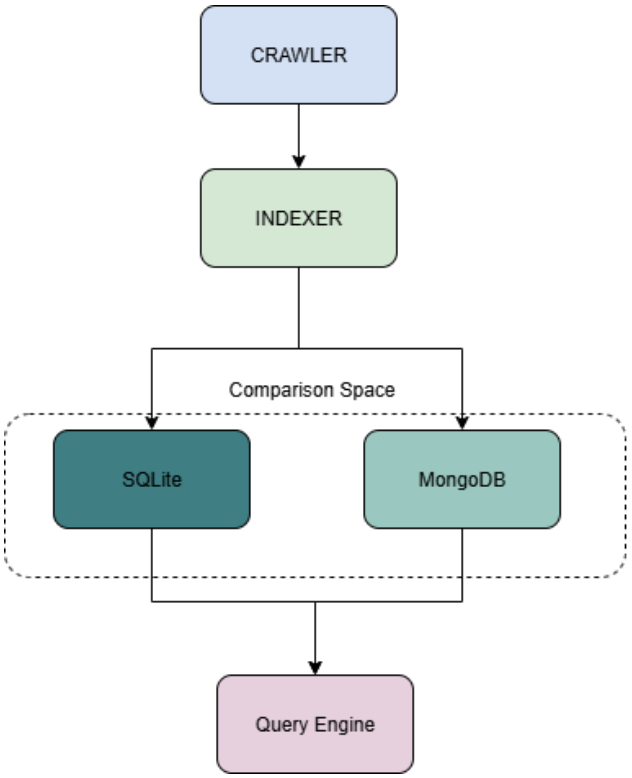

Figure 1. System architecture of the prototype search engine.

3.1 Crawler

The crawler module is responsible for downloading books from the Project Gutenberg digital library. The implementation uses the gutenbergpy.textget library to retrieve texts based on their unique Gutenberg ID. Each retrieved book is cleaned with the built-in strip_headers function to remove Project Gutenberg metadata and keep only the plain text content.

To identify the latest available book, the crawler uses requests and BeautifulSoup to parse the Gutenberg search page sorted by release date and extract the ID of the most recently published book. This functionality allows the system to fetch either a predefined range of books (get_books), only newly released items (get_new_books), or a single book (get_book).

All downloaded books are stored in the local repository through the repository_connection module. For each book, both the Gutenberg ID and the cleaned text are inserted into the database. During initial testing, we downloaded and stored two sample books to verify the crawler functionality.

3.2 Indexer (the filesystem database)

The indexer component is responsible for creating an inverted index from the book texts provided by the crawler. In the filesystem database implementation, the system uses a relational database stored locally as a single file. A simple tokenizer converts the raw text into lowercase terms, filtering out non-alphabetical characters and discarding words shorter than two characters.

For each unique term, the indexer maintains a posting list containing the IDs of books in which the term appears. In the filesystem database, this information is stored in a table with two columns: term and postings. The postings are represented as a JSON array of book IDs. During indexing, the program checks whether the term already exists, if so, the posting list is updated, otherwise a new entry is created. Transactions are used to ensure consistency while processing each book.

This approach provides a lightweight and portable solution where the entire index is contained within a single file, making it straightforward to query and analyze. However, its scalability is limited when handling large collections of documents, as the single-file structure can become a bottleneck.

3.3 Indexer (MongoDB)

In addition to the relational model, we implemented a NoSQL version of the inverted index using MongoDB. Each term is stored as a separate document in the invertedIndex collection. The document contains the field term and a posting list with the IDs of books that include the term. To optimize search performance, an index is created on the term field.

The MongoDB implementation benefits from its ability to scale horizontally and to handle large datasets more efficiently than the filesystem database. The $addToSet operation is used when updating posting lists, which ensures that duplicate book IDs are not inserted. This design makes the index suitable for distributed environments, where multiple workers could contribute to indexing simultaneously.

While MongoDB introduces more complexity compared to the filesystem database, it enables better scalability and is more appropriate for larger Big Data applications.

3.4 Query Engine

The query engine provides the interface for retrieving documents from the search system. It supports two types of queries, metadata-based and full-text. Metadata search allows users to look up documents by title, author, or language using the information stored in the repository.

In addition to metadata search, the query engine enables full-text retrieval using the inverted index. Terms are stored in a file system–based structure, where each term has a

corresponding JSON file containing its posting list (the IDs of the books in which the term occurs).

This minimal query engine serves as the foundation for testing the functionality of the crawler and indexer.

4. Experiments and Results

4.1 Dataset and Environment

For the experiments we used a subset of books from the *Project Gutenberg* digital library. The crawler downloaded a small collection of texts and stored them in our MongoDB database. Each book was represented by its Gutenberg ID and the plain text content.

4.2 Benchmark Setup

Search Benchmark

In addition to indexing, we benchmarked the query engine to evaluate retrieval performance. The file test_search_bench.py defines two categories of experiments:

1. Single-term queries – The test_single_term function measures the performance of looking up individual terms.. This benchmark reflects the common case of searching for a single keyword in the collection.

2. Multi-term queries – The test_multi_term function evaluates queries consisting of two or more keywords combined with Boolean AND. For example, searches such as love AND adventure, ship AND island, or sea AND journey AND storm require computing the intersection of multiple posting lists. This test highlights the efficiency of the inverted index in handling compound queries.

The benchmark uses the pytest-benchmark framework to measure execution times in microseconds, providing detailed statistics such as minimum, maximum, mean, and standard deviation. These results allow us to compare the retrieval latency of the filesystem database and MongoDB indexes under realistic workloads.

Indexer Benchmark

To evaluate the performance of the indexer, we used the pytest-benchmark framework, which allows for systematic measurement of execution times.

Two main operations were benchmarked:

1. Tokenization – the conversion of raw text into a list of lowercase terms, filtering out non-alphabetic characters and very short words. The test_tokenize_benchmark function generates a synthetic text of 1,000 random words and measures the speed

of the tokenize function.

2. Book Processing – the complete indexing of a document using the process_book function, which updates both the filesystem database and MongoDB inverted indexes. The test_process_book_benchmark function measures the average execution time for inserting a randomly generated dummy book into the index.

The dummy text generator ensures reproducible experiments by creating random words of variable length. These synthetic datasets allow us to stress-test the indexer without depending exclusively on external sources like Project Gutenberg.

The results of these benchmarks provide insights into the relative performance of the filesystem database and MongoDB backends in terms of indexing time and scalability.

We benchmarked the two different inverted index implementations, the filesystem database and MongoDB which including:

- Tokenization time
- Search time
- Indexing time

4.3 Results

The experimental results are summarized in Table 1 and Table 2

Table 1. Benchmark results for tokenization, search, and indexing (18 books)

Metric/Test	Min (μs)	Max (μs)	Mean (μs)	Median (μs)	OPS (ops/sec)
Tokenization time	78.15	98.31	86.05	82.62	11,621
Search time	1,761.47	6,863.35	2,933.37	1,964.96	341
Indexing time	743,675.33	932,633.50	832,916.95	803,015.60	1.20

Table 2. Benchmark results comparing (18 books)

Metric/Test	Min (μs)	Max (μs)	Mean (μs)	Median (μs)	OPS (ops/sec)
Tokenization time	63.77	90.21	68.89	64.79	14,514
Search time	3,270.59	10,281.16	5,914.96	4,457.39	169,1
Indexing time	1,727,530.97	2,602,842.50	2,384,751.80	2,558,704.73	0,419

Table 3. Comparison of benchmark results for inverted indexes (18 books)

Metric/Test	filesystem database mean (µs)	MongoDB Mean (µs)	filesystem database OPS	MongoDB OPS
Tokenization time	86.05	68.89	11,621	14,515
Search time	2,933.37	5,914.96	341	169
Indexing time	832,916.95	2,384,751.80	1.20	0.42

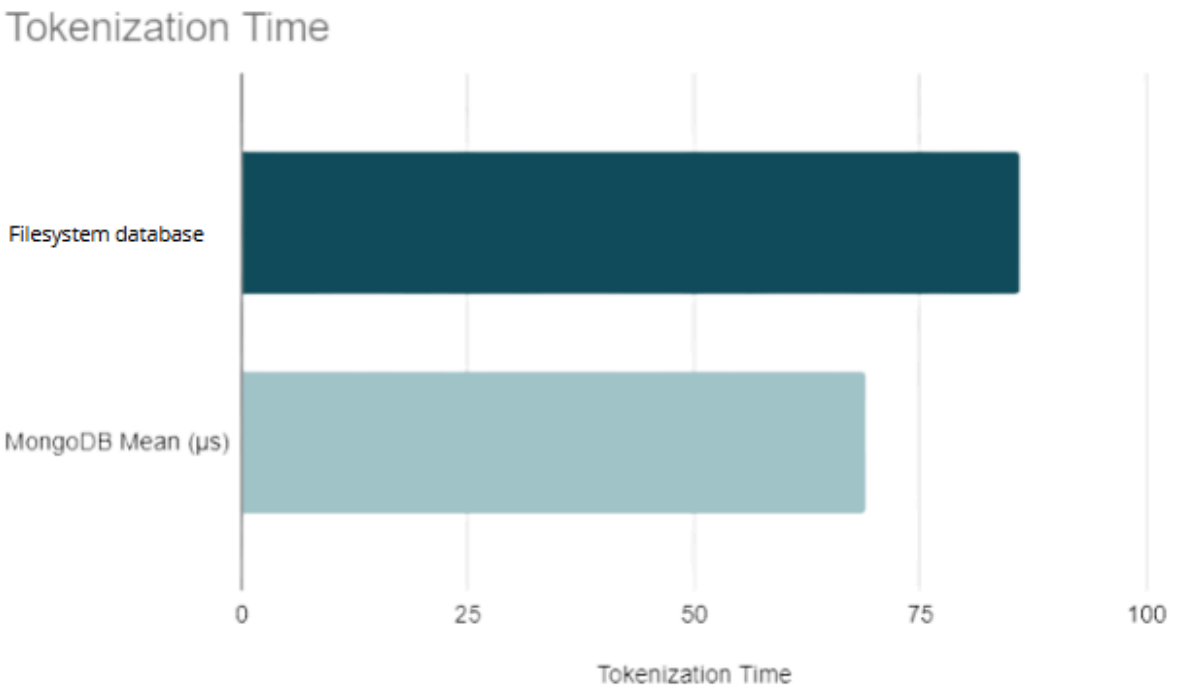

Figure 2. Tokenization Time

Comparison of mean tokenization times (µs) for the filesystem database and MongoDB.

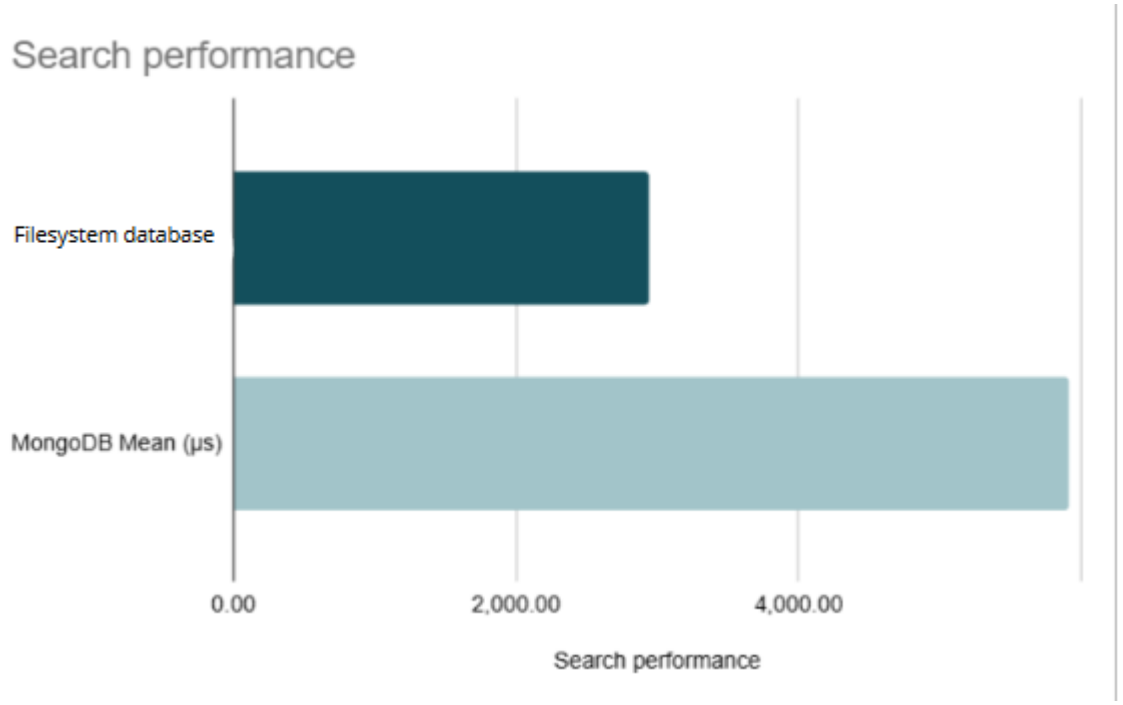

Figure 3. Search Performance

Comparison of mean search query times (µs) for the filesystem database and MongoDB.

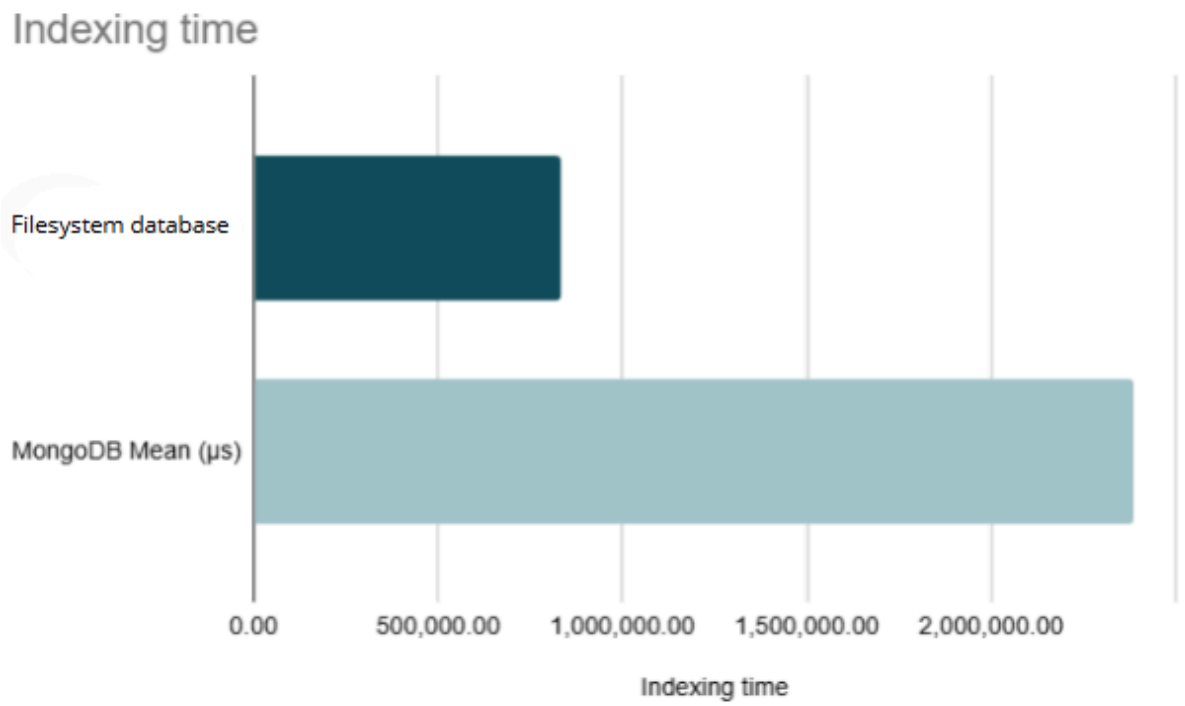

Figure 4. Indexing Time

Comparison of mean indexing times (µs) for the filesystem database and MongoDB.

5. Discussion

The benchmark highlights distinct trade-offs between the two inverted index implementations.

Tokenization was fast in both systems, with MongoDB slightly outperforming the filesystem database.

Search performance showed clear differences, the filesystem database returned results almost twice as fast as MongoDB.. For single-node, small-scale experiments, the filesystem database lightweight file-based architecture provides more efficient query execution.

Indexing speed proved to be the most resource-intensive operation. This confirms the higher overhead of document-oriented updates in MongoDB. However, while slower in this small-scale setup, MongoDB provides features like horizontal scaling, distributed indexing that could compensate for these disadvantages in larger deployments.

In summary, the results suggest that the filesystem database is more efficient and lightweight for small collections and prototyping, while MongoDB is more suitable for scenarios that require scalability and distributed architectures, at the cost of higher indexing and query latency in a local environment.

6. Conclusion

In this paper we presented the design and implementation of a prototype search engine for Project Gutenberg texts. The system integrates three core components: a crawler for data ingestion, an indexer for building inverted indexes, and a query engine for retrieving documents.

The experimental benchmarks demonstrated clear trade-offs between the two approaches. The filesystem database achieved faster query times and lower indexing overhead, making it a suitable choice for small to medium-scale collections and for rapid prototyping. MongoDB, on the other hand, exhibited higher indexing and query latency in our local setup, but offers scalability features and distributed capabilities that may become advantageous in larger Big Data contexts. These results confirm that there is no single “best” solution, but rather that the choice of backend depends on the scale and requirements of the application.

The project provided valuable insights into the core building blocks of search engines and the importance of data structures in determining performance. It also highlighted practical challenges such as preprocessing, ensuring consistency of indexes, and benchmarking under reproducible conditions.

Future work will include completing a more extensive performance evaluation on larger datasets, extending the query functionality, and potentially developing a web-based interface to make the system more user-friendly.

7. References

Witten, I. H., Moffat, A., & Bell, T. C. (1999). *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann.

Project Gutenberg. Available at: https://www.gutenberg.org/

MongoDB Documentation. Available at: https://www.mongodb.com/docs/

pytest-benchmark Documentation. Available at: https://pytest-benchmark.readthedocs.io/

Note about using AI: During the development of this project, we made use of AI-based tools to support the coding and documentation process.