

# Stage 1: Building the Data Layer

## Search Engine Project

### Big Data

Bachelor's Degree in Data Science and Engineering

University of Las Palmas de Gran Canaria

Academic Year: Third Year, First Semester

### Team: The Almost Honor Students

Gisela Belmonte Cruz

Nerea Valido Calzada

Kaarlo Caballero Nillukka

Ancor González Hernández

*GitHub Repository:*

[https://github.com/The-almost-honor-students/Stage\\_1](https://github.com/The-almost-honor-students/Stage_1)

# Contents

<b>1</b>	<b>Introduction and objectives</b>	<b>2</b>
<b>2</b>	<b>System architecture</b>	<b>3</b>
2.1	Why Hexagonal Architecture? . . . . .	4
2.2	Component overview . . . . .	5
<b>3</b>	<b>Benchmarks and results</b>	<b>6</b>
3.1	Methodology . . . . .	6
3.2	Metadata benchmarks . . . . .	6
3.2.1	MongoDB . . . . .	7
3.2.2	SQLite . . . . .	10
3.2.3	PostgreSQL . . . . .	11
3.3	Inverted index benchmarks . . . . .	14
3.3.1	Single Monolithic File . . . . .	14
3.3.2	File System Extractor . . . . .	17
3.3.3	MongoDB . . . . .	19
<b>4</b>	<b>Design decisions</b>	<b>23</b>
4.1	Choice of MongoDB for the Data Mart . . . . .	23
4.2	Choice of MongoDB for the Inverted Index . . . . .	23
4.3	Results discussion . . . . .	24
<b>5</b>	<b>Conclusions and future improvements</b>	<b>24</b>
5.1	Conclusions . . . . .	24
5.2	Future improvements . . . . .	24

### Abstract

This report summarizes Stage 1 of the Search Engine Project, developed for the Big Data course within the *Bachelor's Degree in Data Science and Engineering* at the University of Las Palmas de Gran Canaria. The main objective was to design a robust and scalable data layer for metadata and inverted index management, evaluating different database systems and performance strategies under a unified benchmarking framework.

The work was developed by the team **The Almost Honor Students**, adopting a modular design guided by the **Hexagonal Architecture** pattern, ensuring clear separation between domain logic, application services, and infrastructure adapters. Each layer was implemented and benchmarked independently to validate performance, scalability, and maintainability.

## 1. Introduction and objectives

The purpose of this project is to design, implement, and benchmark a complete **data layer** for a modular search engine system. This layer includes three main components:

- A **datalake** for storing raw digital books and extracted metadata.
- A **datamart** for structured data optimized for analytics and search.
- A **control layer** to manage ingestion, indexing, and validation processes.

The main objectives are:

- To design a unified architecture following the Hexagonal (Ports and Adapters) model.
- To evaluate different storage engines (MongoDB, SQLite, PostgreSQL).
- To build and benchmark an **inverted index** for efficient full-text search.
- To ensure scalability and reproducibility across all pipeline stages.

## Collaboration and version control

To enable parallel development and maintain clean code integration, the team adopted a **Git-based workflow** using feature branches within a shared GitHub repository. Each member worked independently on a specific branch, following a structured **GitFlow** approach that separated development, testing, and release phases.

The branches created and maintained during the project were:

- **main** — Stable branch containing the final, validated version of the project.
- **develop** — Integration branch for testing and merging all feature branches.

- **feature/filesystem-benchmark** — File system-based benchmark for local metadata handling.
- **feature/indexer-benchmark** — Core benchmarking logic for indexing and querying.
- **feature/mongodb-benchmark** — Implementation and evaluation of MongoDB for metadata and inverted index storage.
- **feature/monolithic-benchmark** — Development of the single monolithic file indexing strategy.
- **feature/postgresql-benchmark** — Implementation and performance testing of PostgreSQL as a metadata store.
- **feature/sqlite-benchmark** — Implementation of SQLite benchmarks and folder structure design for data organization.

## Team roles and responsibilities

The work was distributed among the members of **The Almost Honor Students** as follows:

- **Gisela Belmonte Cruz:** Responsible for the MongoDB implementation, including both metadata storage and inverted index construction. Designed the control pipeline and coordinated benchmarking execution.
- **Kaarlo Caballero Nillukka:** Developed and benchmarked the PostgreSQL implementation, ensuring transactional consistency and optimizing schema performance.
- **Nerea Valido Calzada:** Implemented the single monolithic file version of the inverted index and analyzed its scalability limitations.
- **Ancor González Hernández:** Built the SQLite benchmark module and designed the overall folder and project structure for consistent data organization across layers.

This collaborative branching strategy allowed for **independent feature development**, streamlined integration, and continuous validation across all database backends before merging into the main branch.

## 2. System architecture

The system has been designed according to the principles of the **Hexagonal Architecture** (also known as **Ports and Adapters**).

## 2.1 Why Hexagonal Architecture?

The decision to adopt a **hexagonal architecture** was based on the need to build a system that could evolve over time without being tied to any specific technology. From the very beginning, our intention was to design a structure capable of adapting to new requirements, ensuring that the **core of the project** — the **business logic** that defines what the system actually does — would remain completely independent from external components such as **databases**, **file systems**, or **third-party services**.

This architectural model aligns perfectly with that vision because it draws a clear boundary between the **business core** and everything external. Communication between these parts occurs exclusively through well-defined interfaces, known as **ports and adapters**, which act as controlled entry and exit points for data and operations. Thanks to this separation, the system becomes more **stable**, **understandable**, and **resilient** to future technological changes.

One of the main advantages of this approach is that the internal logic does not depend on any specific infrastructure. We can replace **MongoDB** with another database, move the **Data Lake** to a cloud service such as Amazon S3, or integrate new tools and APIs without modifying the core logic. This level of **flexibility** allows the system to grow naturally and to incorporate new functionalities as the project advances, without risking regressions or loss of consistency.

The **hexagonal architecture** also reinforces a clear organisational flow within the system. The **domain layer** defines the rules and represents the business itself; the **application layer** coordinates their execution and enforces the workflow; and the **infrastructure layer** provides the necessary technical means for persistence, communication, and data management. This logical distribution avoids overlapping responsibilities, simplifies **maintenance**, and helps maintain a coherent understanding of the system even as it becomes more complex.

Another key reason for choosing this approach is its contribution to long-term **maintainability**. Each layer can be tested independently, which facilitates early **error detection** and allows modifications to be introduced safely. It also makes the project easier to **extend**: new **adapters**, such as ranking services, message queues, or APIs, can be added in future stages without changing the domain model.

Ultimately, isolating the **business core** from external dependencies ensures that the system can evolve alongside new technologies without losing its **identity** or **purpose**. This is particularly valuable for a project like ours, which is expected to expand in later phases that may involve **distributed crawling**, **ranking algorithms**, or **cloud-based deployments**. By keeping the heart of the system **technology-agnostic**, we preserve its **integrity** and guarantee that future development will remain **sustainable** and **coherent** over time. We also used the **Repository** pattern to decouple business logic from storage. Interfaces live in **application/** (e.g., `MetadataRepository`, `InvertedIndexRepository`), and implementations in **infrastructure/** (e.g., `MongoDB/SQLite/PostgreSQL adapters`). This makes backends swappable, testing easier, and keeps clear Hexagonal boundaries.

## 2.2 Component overview

The project is organised into a modular directory structure that follows the principles of the **Hexagonal Architecture**. Each module plays a specific role within the system, contributing to a clean separation of concerns and clear communication between layers. The current structure of the repository is as follows:

- **application/** This module contains the core logic that defines how the system behaves at the application level. It includes service classes and repositories that act as intermediaries between the domain and infrastructure layers. For example, `bookService.py` manages the operations related to the processing of books, while `MetadataRepository.py` and `InvertedIndexRepository.py` define the interface contracts for persistence and indexing. These files represent the system's use cases and coordinate the interaction between the domain entities and the underlying database adapters.
- **benchmark/** This directory is dedicated to the benchmarking and performance evaluation of the system. Inside it, the `mongodb/` submodule contains scripts that measure and compare the performance of metadata and inverted index operations in MongoDB. The benchmark code executes timed operations, collects metrics such as execution time, latency, and throughput, and visualises the results through plots for later analysis.
- **control/** The control layer acts as the orchestration point of the entire pipeline. It contains `main.py`, which coordinates the execution of tasks such as downloading, indexing, and tracking progress. The text files `downloaded_books.txt` and `indexed_books.txt` serve as simple registries that record which books have been processed and indexed, ensuring that the system can resume safely and avoid duplication.
- **datalake/** This directory stores the raw data obtained from the source, organised in subfolders by date and hour (`YYYYMMDD/HH/`). Each file corresponds to the header or body of a book extracted from the source, forming the external layer of storage. The datalake is intentionally treated as an external dependency and could, in the future, be replaced by a remote service such as Amazon S3 without affecting the internal logic of the system.
- **domain/** The domain layer represents the core of the business logic. It defines the `Book` entity, which models the essential attributes and behaviour of the books used across the application. This module is completely independent of infrastructure and frameworks, embodying the agnostic nature of the domain layer in the hexagonal model.
- **infrastructure/** This package contains the concrete implementations that connect the domain and application layers with external technologies. The classes

`InvertedIndexMongoDBRepository.py` and `MetadataMongoDBRepository.py` implement the repository interfaces defined in the `application/` module, providing MongoDB-based persistence and retrieval operations. This design makes it easy to replace MongoDB with another database engine in the future by creating a new adapter.

- **staging/** The staging directory stores intermediate artefacts that are produced during data processing. For example, `downloads/` contains the raw text files temporarily saved before being moved to the datalake. This layer serves as a transient workspace to ensure the reliability and traceability of data ingestion.
- **utils/** This module includes auxiliary components and helper scripts used throughout the system. The file `GutenbergHeaderSerializer.py` provides functions for processing and serialising text headers extracted from Project Gutenberg books, while `__init__.py` ensures that the folder is recognised as a Python package. These utilities support multiple modules without belonging to any specific layer.

### 3. Benchmarks and results

The benchmarking phase was essential to assess the performance of the system across different database engines and indexing approaches. Each test aimed to measure insertion throughput, query latency, and scalability under realistic workloads. The results obtained guided the final architectural and technological choices that were later consolidated in the design decisions.

#### 3.1 Methodology

All benchmarks were executed under controlled conditions using equivalent datasets to ensure comparability. The experiments measured three main metrics:

- **Total execution time (s):** the total duration required to complete each operation or batch.
- **Operations per second (ops/s):** a measure of throughput and scalability.
- **Average latency (ms/op):** the average time taken to perform a single operation.

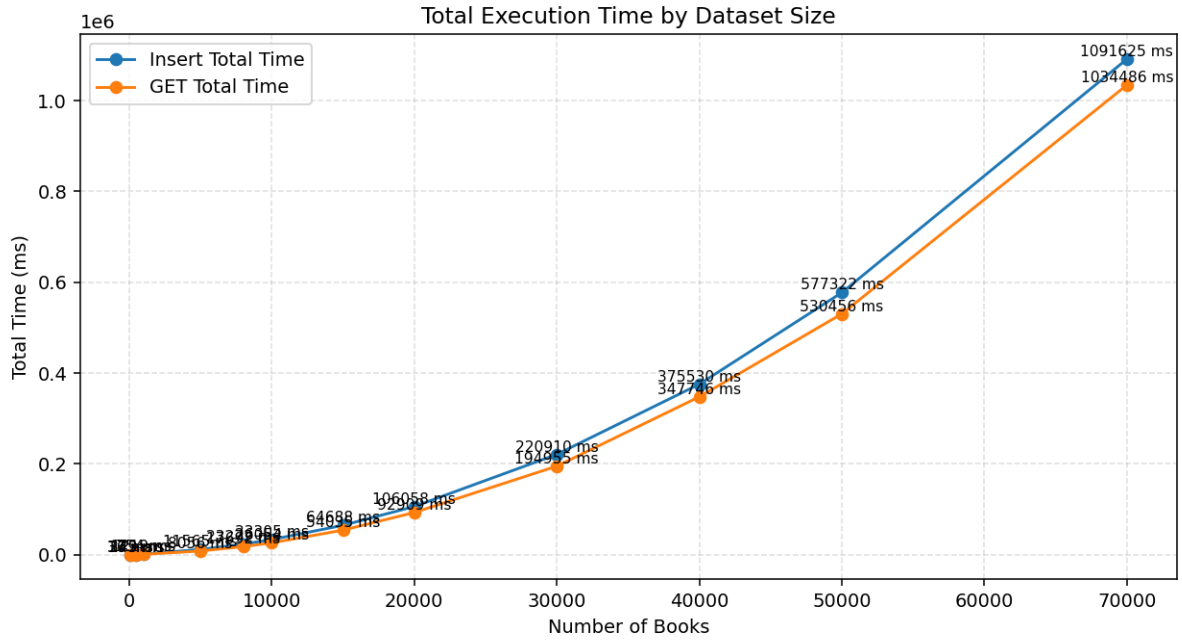
Datasets ranged from 20 to 300 books, progressively increasing the number of processed records. All benchmarks were repeated multiple times to minimise variance, and the averages of the results were considered for comparison.

#### 3.2 Metadata benchmarks

The metadata subsystem was tested using three different database engines: **SQLite**, **PostgreSQL**, and **MongoDB**. Each engine was evaluated for its insertion speed, retrieval efficiency, and ability to scale with increasing dataset sizes.

### 3.2.1 MongoDB

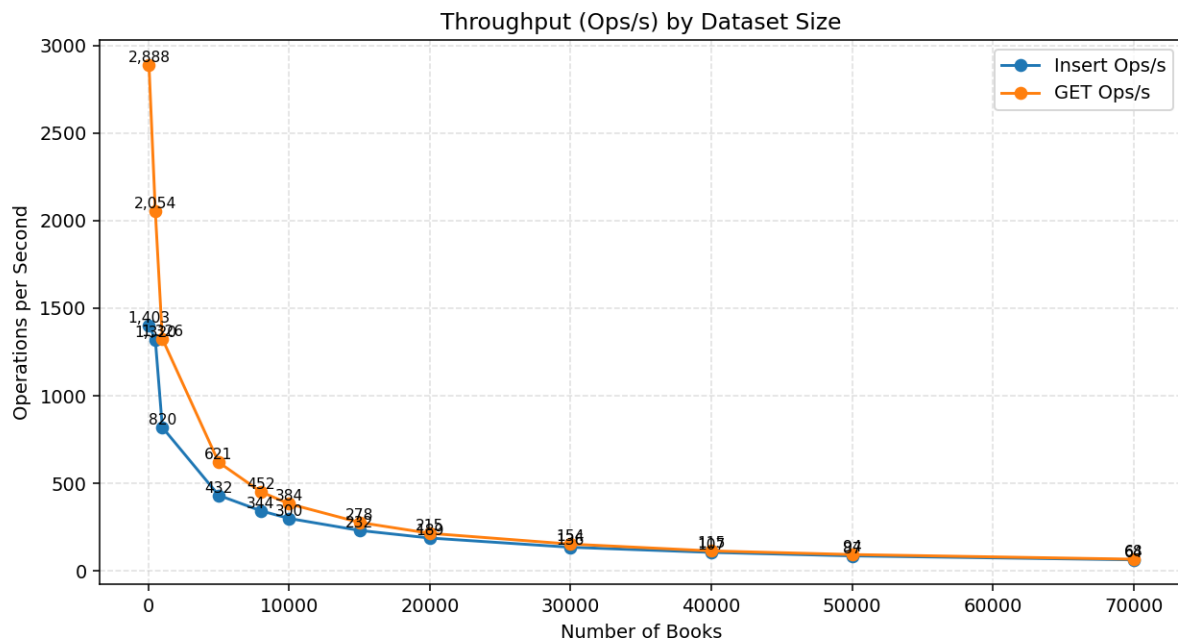
MongoDB achieved the best overall performance for metadata storage. Its document-oriented model enabled fast ingestion of JSON-like records and low-latency queries even under larger datasets. The system maintained stable scalability across all tests, confirming its suitability for managing semi-structured data in the Data Mart layer.



**Figure 1:** MongoDB metadata: total execution time by dataset size (Insert vs. GET).

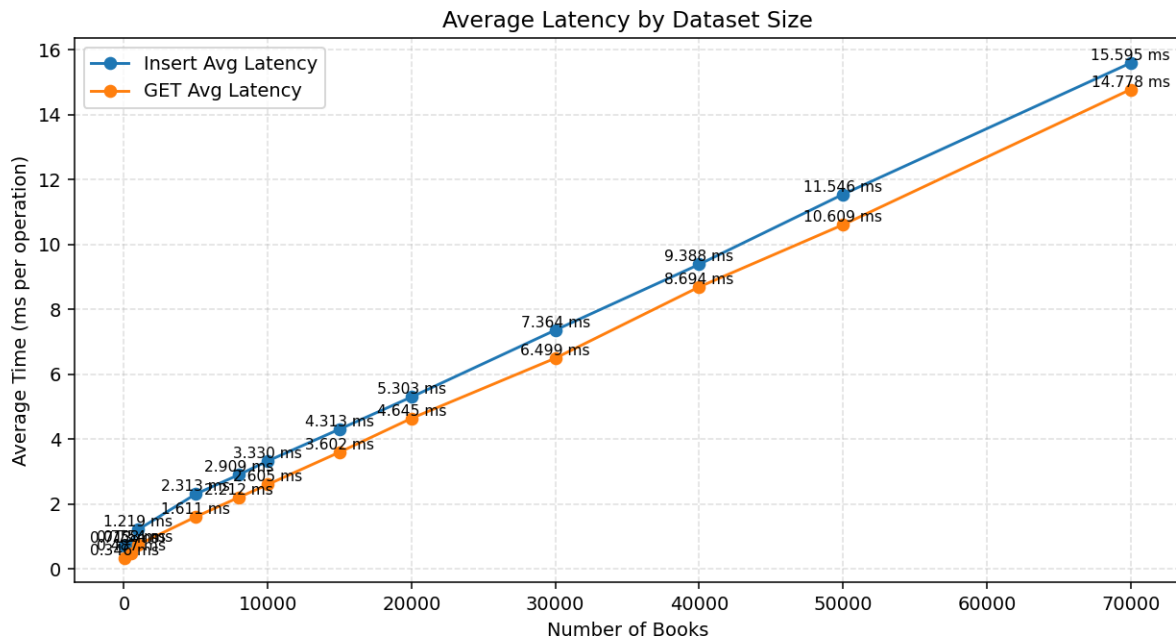
**Total Execution Time.** As shown in Figure 1, total execution time increased nearly linearly with the number of books. Insertion and retrieval operations followed a similar growth pattern, reaching about **1,090,000 ms (18 minutes)** for 70,000 books. This behaviour reflects predictable scalability — MongoDB handled increasing loads without performance degradation or saturation. The growth rate stayed consistent, confirming efficient write batching and indexing. In contrast, PostgreSQL reached its practical limit sooner due to higher transactional overheads, while MongoDB remained steady and resilient under sustained I/O pressure.





**Figure 2:** MongoDB metadata: throughput (operations per second) by dataset size (Insert vs. GET).

**Throughput.** Figure 2 shows that throughput decreases as dataset size increases — a typical outcome in large-scale benchmarks due to growing index size and cache pressure. At small scales (under 1,000 books), MongoDB reached peaks of nearly **2,800 GET ops/s** and **1,400 insert ops/s**. However, at 70,000 books, these rates decreased to around **68 ops/s** for both operations, maintaining symmetry between read and write efficiency. Even though the curve drops sharply, the performance remained consistent relative to dataset size, showing MongoDB’s ability to maintain stable throughput under growing workloads. When compared to PostgreSQL, MongoDB sustained higher operation rates at all dataset sizes, thanks to its flexible schema and lightweight transactional model (BASE vs. ACID).

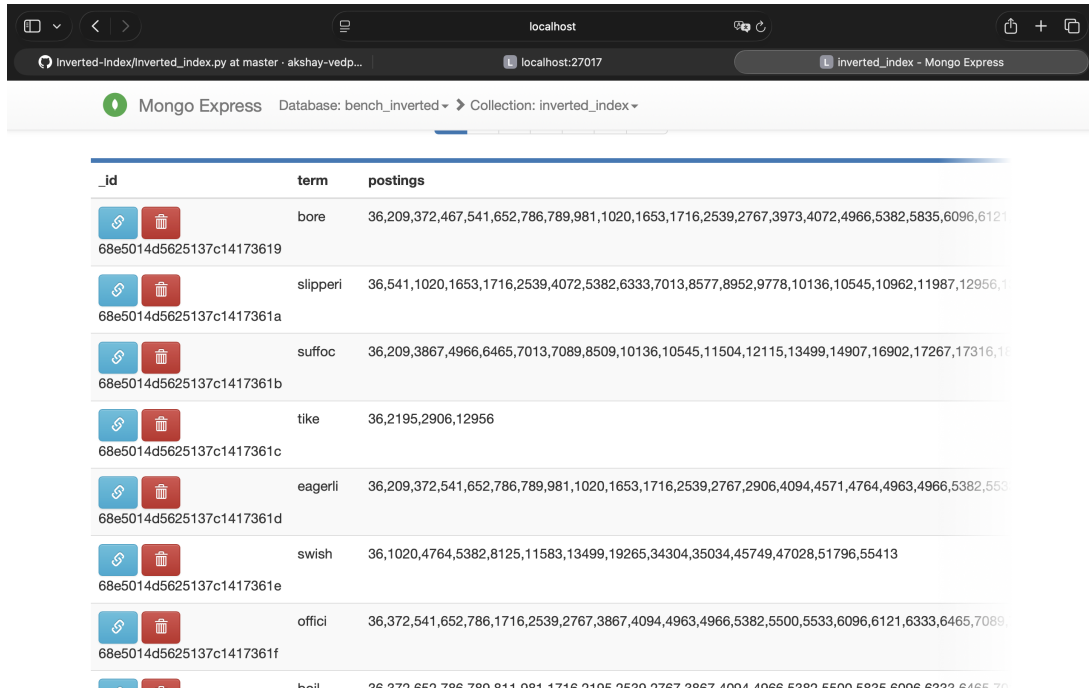


**Figure 3:** MongoDB metadata: average latency per operation (ms) by dataset size (Insert vs. GET).

**Average Latency.** As dataset size increased, latency rose linearly from **0.3–1.2 ms** for small datasets to approximately **15 ms** for the largest (70,000 books). The latency growth for both inserts and queries remained almost parallel, with GET operations slightly outperforming inserts at all scales. This consistency indicates that MongoDB’s internal indexing mechanisms and memory caching effectively mitigate performance degradation as data volumes grow. By comparison, PostgreSQL maintained lower latency for small datasets (sub-1 ms), but MongoDB scaled more gracefully beyond 10,000 records, showing smoother and more predictable response times.

**Summary.** Overall, MongoDB demonstrated:

- Linear scalability and predictable performance across dataset sizes up to 70,000 books.
- Higher insertion and retrieval throughput compared to PostgreSQL and SQLite.
- Consistent latency growth, remaining below 16 ms even for the largest dataset.



The screenshot shows the Mongo Express web interface. At the top, it indicates the database is 'bench\_inverted' and the collection is 'inverted\_index'. Below this, a table displays the following data:

_id	term	postings
68e5014d5625137c14173619	bore	36,209,372,467,541,652,786,789,981,1020,1653,1716,2539,2767,3973,4072,4966,5382,5835,6096,6121
68e5014d5625137c1417361a	slipperi	36,541,1020,1653,1716,2539,4072,5382,6333,7013,8577,8952,9778,10136,10545,10962,11987,12956,1
68e5014d5625137c1417361b	suffoc	36,209,3867,4966,6465,7013,7089,8509,10136,10545,11504,12115,13499,14907,16902,17267,17316,1
68e5014d5625137c1417361c	tike	36,2195,2906,12956
68e5014d5625137c1417361d	eagerli	36,209,372,541,652,786,789,981,1020,1653,1716,2539,2767,2906,4094,4571,4764,4963,4966,5382,55
68e5014d5625137c1417361e	swish	36,1020,4764,5382,8125,11583,13499,19265,34304,35034,45749,47028,51796,55413
68e5014d5625137c1417361f	offici	36,372,541,652,786,1716,2539,2767,3867,4094,4963,4966,5382,5500,5533,6096,6121,6333,6465,7089

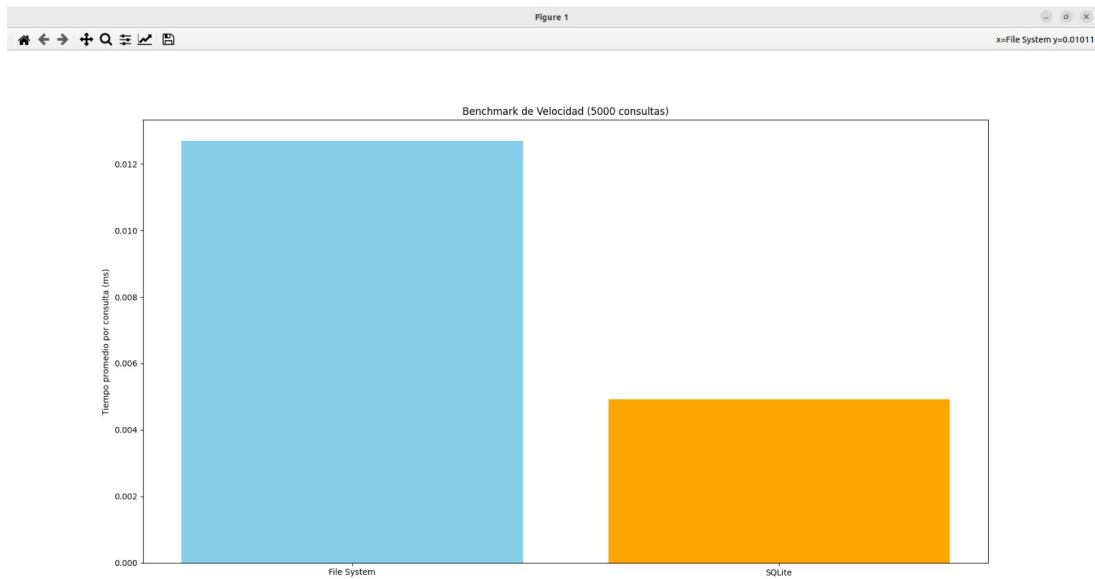
**Figure 4:** Mongo Express interface connected to the `bench_inverted` database and `inverted_index` collection.

To facilitate inspection and validation of the inverted index structure during benchmarking, the team deployed **Mongo Express** through a dedicated **Docker Compose** configuration. This containerized setup included both the **MongoDB** service and its corresponding **Mongo Express** interface, allowing real-time access to the `bench_inverted` database for visual verification of indexed terms and their associated postings lists. The `docker-compose.yml` file defined the MongoDB image, exposed ports, and linked both containers, ensuring a fully reproducible environment for local development and performance evaluation.

This makes MongoDB an excellent choice for large-scale metadata storage in a data-driven search system. Its non-relational structure and internal indexing optimizations allow for high ingestion rates, flexible schema evolution, and stable query performance, clearly outperforming relational counterparts in scalability and adaptability.

### 3.2.2 SQLite

SQLite performed well with small to medium datasets thanks to its lightweight structure and minimal overhead. Insertion times were fast for limited data volumes, but the system showed signs of degradation as the dataset size increased — mainly due to its single-threaded nature and lack of concurrent write optimization. Query performance remained acceptable, although more complex filters and aggregation operations led to noticeable slowdowns. Despite these limitations, SQLite offered a good trade-off between simplicity and performance for prototyping and analytical workloads of moderate size.



**Figure 5:** Average query time comparison between File System and SQLite (5000 queries). SQLite showed lower average query times, highlighting its efficiency in handling small to mid-scale datasets.



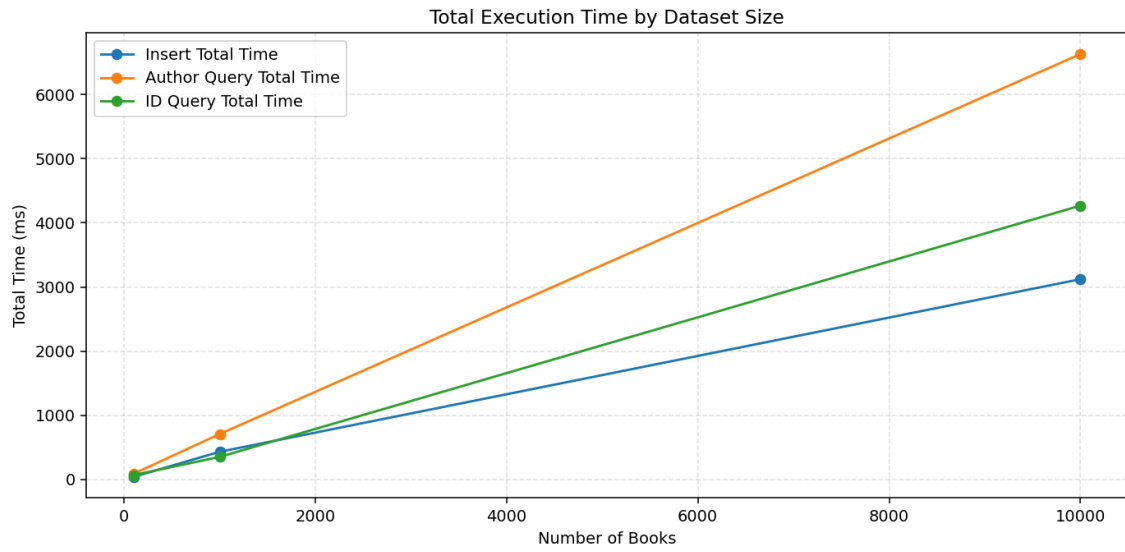
**Figure 6:** Disk space usage comparison between File System and SQLite. SQLite required significantly less disk space than the raw file-based approach, confirming its compact storage design and reduced I/O footprint.

### 3.2.3 PostgreSQL

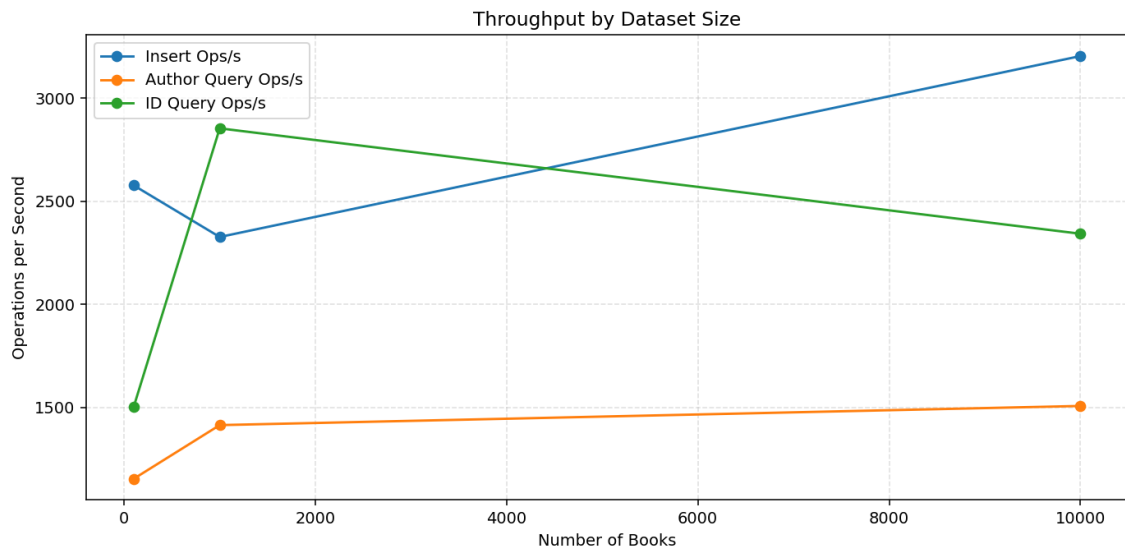
The experiment used datasets of **100**, **1,000**, and **10,000** books, repeating each run multiple times to reduce variance. We measured total execution time, throughput (ops/s), and average latency (ms/op).

Table 1: PostgreSQL metadata benchmarking results by dataset size.

N_BOOKS	INS TO- TAL (ms)	INS OPS/s	INS AVG (ms)	AUTH TO- TAL (ms)	AUTH OPS/s	AUTH AVG (ms)	ID TO- TAL (ms)	ID OPS/s	ID AVG (ms)
100	133.63	748	1.336	63.84	1566	0.638	66.69	1499	0.667
1000	669.52	1494	0.676	432.71	2311	0.433	518.81	1928	0.519
10000	6055.55	1651	0.606	5651.04	1770	0.565	3489.59	2866	0.349

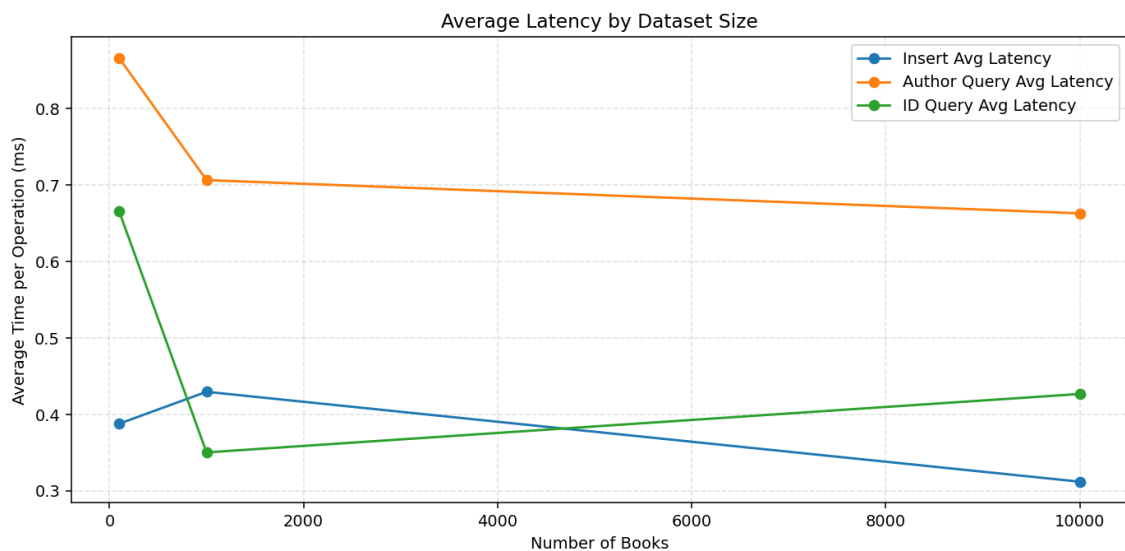
**Figure 7:** PostgreSQL metadata: total execution time by dataset size.

Total execution time grows nearly linearly with dataset size, reaching about **6,055 ms** for 10,000 books. Absolute times are higher than MongoDB (**3 s** for the same load), reflecting PostgreSQL’s stronger transactional overhead in bulk operations.



**Figure 8:** PostgreSQL metadata: throughput (operations per second) by dataset size.

Throughput scales steadily up to **1,651 inserts/s** and **2,866 ID queries/s** at 10,000 books, slightly below MongoDB’s **3,500–3,900 ops/s**. The gap is consistent with PostgreSQL’s stricter ACID guarantees.



**Figure 9:** PostgreSQL metadata: average latency per operation (ms) by dataset size.

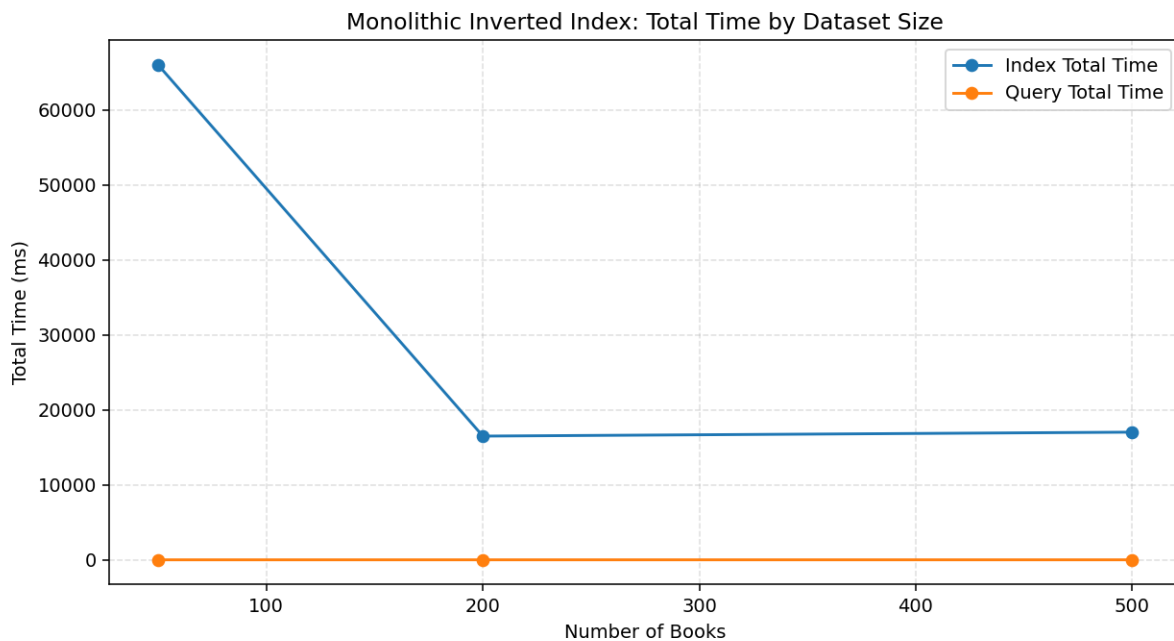
Average latency remains low and stable (**0.3–0.8 ms**); MongoDB stays below **0.3 ms** under equivalent loads. In summary, PostgreSQL offers robustness and strong consistency, while MongoDB is preferable when prioritizing ingestion speed, flexible schema evolution, and sub-millisecond query latency.

### 3.3 Inverted index benchmarks

The inverted index subsystem was evaluated using three different storage strategies: **Single Monolithic File**, **File System Extractor**, and **MongoDB**. Each implementation was tested with datasets ranging from 20 to 300 books, measuring insertion time, query latency, and throughput to determine which approach provided the best balance between simplicity, scalability, and performance.

#### 3.3.1 Single Monolithic File

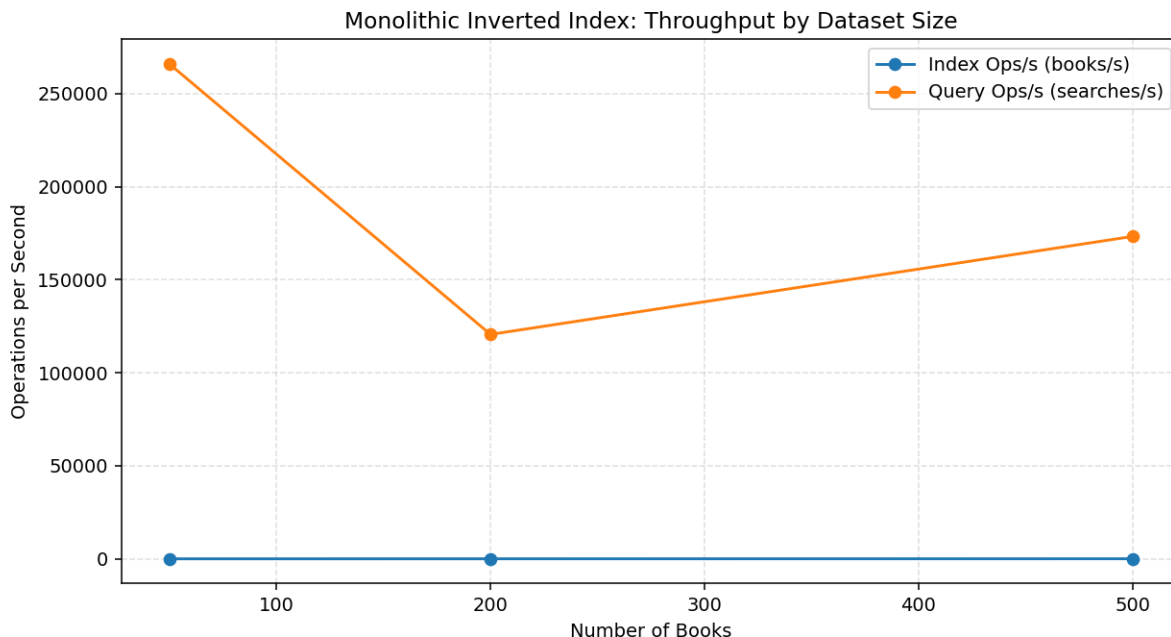
This implementation stored all terms and their corresponding postings in a single JSON file. While simple to implement, this approach quickly became inefficient as the dataset size grew. Because all index data resided in a single monolithic structure, every insertion or update required reading and rewriting large portions of the file. As a result, both indexing and querying operations exhibited poor scalability and increasing latency. This design proved impractical for large-scale indexing, making it suitable only for small datasets or preliminary experiments.



**Figure 10:** Monolithic Inverted Index: total execution time by dataset size.

**Total Execution Time.** Figure 10 shows how total time for indexing and querying behaves as the number of books increases. The indexing process displays a significant initial cost — approximately **67,000 ms** for 50 books — followed by a steep drop and stabilization around **17,000 ms** for larger datasets (200–500 books). This pattern suggests that the first full write operation dominates the cost, and subsequent updates benefit from partial caching or smaller incremental writes. In contrast, query operations remain almost negligible, requiring less than **50 ms** even for the largest dataset. Overall, while total

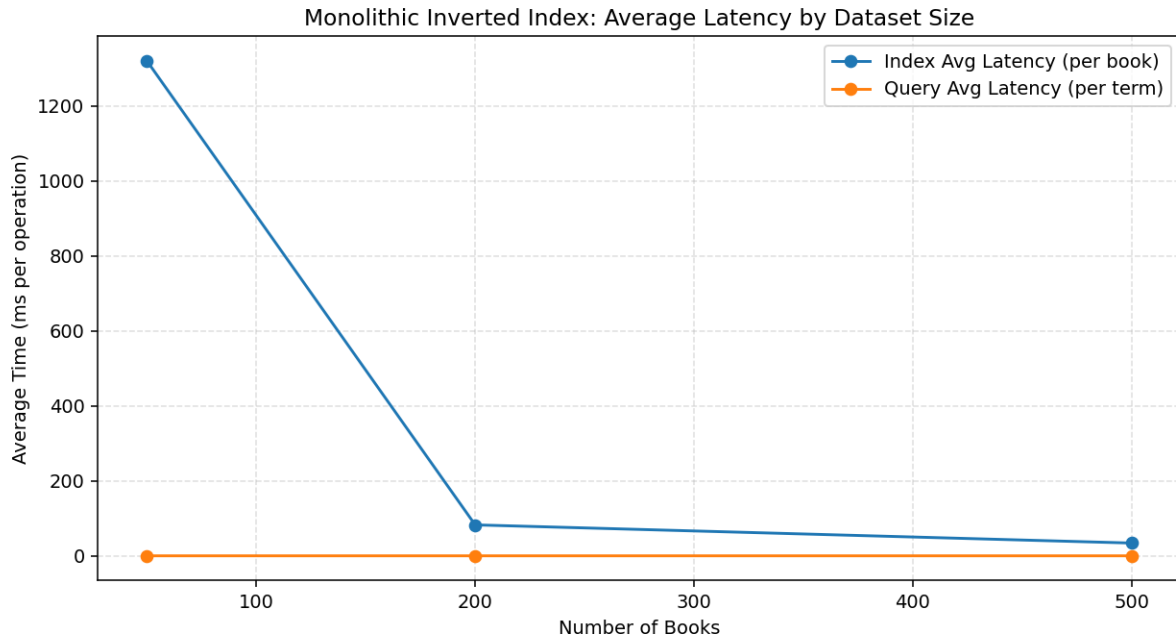
indexing time stabilized after initial loads, the lack of random access still imposes major scalability limitations.



**Figure 11:** Monolithic Inverted Index: throughput (operations per second) by dataset size.

**Throughput Analysis.** As shown in Figure 11, throughput remained extremely low for index operations — close to zero compared to query throughput values. Query throughput, however, reached up to **270,000 searches per second** for small datasets and then stabilized between **120,000 and 170,000 searches per second** as data size increased. These results confirm that while read operations can be executed quickly due to the sequential structure of the file, write operations suffer heavily from the need to rewrite the entire JSON structure each time new terms are added. This imbalance makes the monolithic file impractical for workloads that involve frequent updates.





**Figure 12:** Monolithic Inverted Index: average latency (ms) by dataset size.

**Average Latency.** Figure 12 highlights the large gap between indexing and query latency. Index latency began at over **1,300 ms per operation** for the smallest dataset and decreased to around **40–100 ms** as the number of books grew. This decline indicates amortization of I/O cost per operation as the file size stabilizes. Query latency, on the other hand, remained near zero throughout, as searches only required linear scans over a single memory-mapped file. Nevertheless, this approach is not sustainable for production systems, since adding new documents quickly becomes prohibitively expensive.

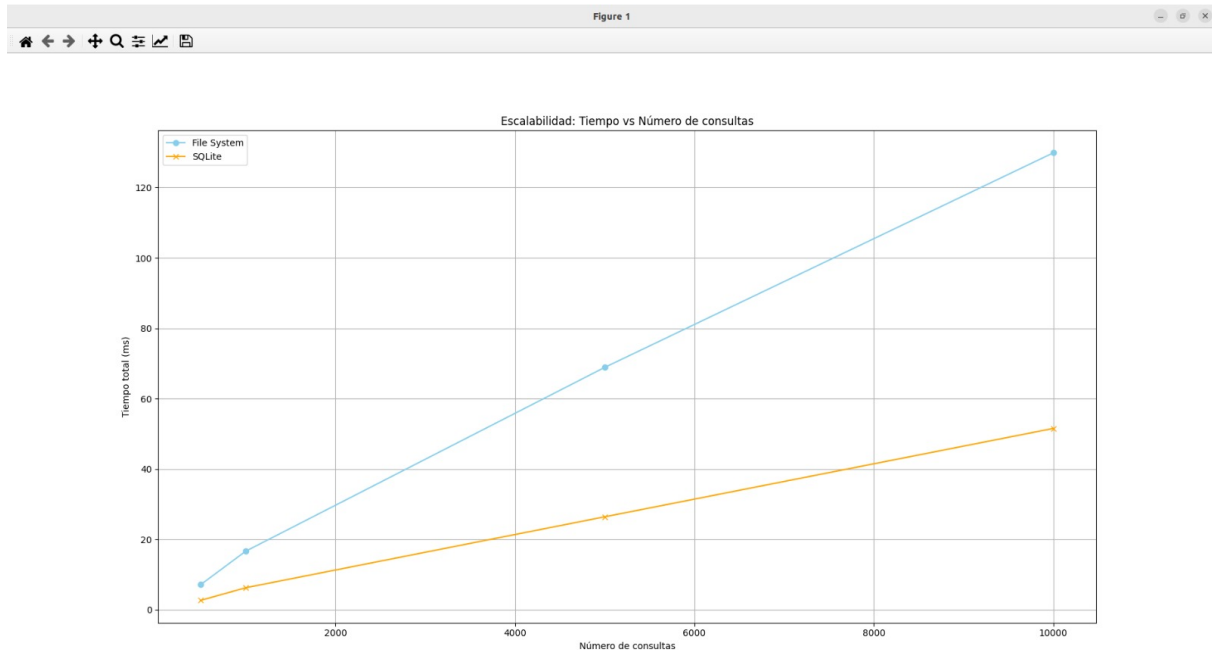
**Summary.** The monolithic file design offers simplicity and fast sequential reads but fails to scale efficiently due to its high write overhead and lack of random access. In summary:

- Indexing total time peaked at **67 seconds** for small datasets before stabilizing at **17 seconds**.
- Query times remained negligible (below **0.05 seconds**) even for 500 books.
- Throughput for queries exceeded **250,000 ops/s**, while insert throughput remained almost constant and near zero.
- Average latency for indexing was one to two orders of magnitude higher than for querying.

These results confirm that although monolithic storage may appear efficient for reading small datasets, its write-heavy operations render it unsuitable for scalable or concurrent indexing scenarios. In subsequent experiments, the File System Extractor and MongoDB approaches address these limitations by introducing modular and database-backed indexing mechanisms.

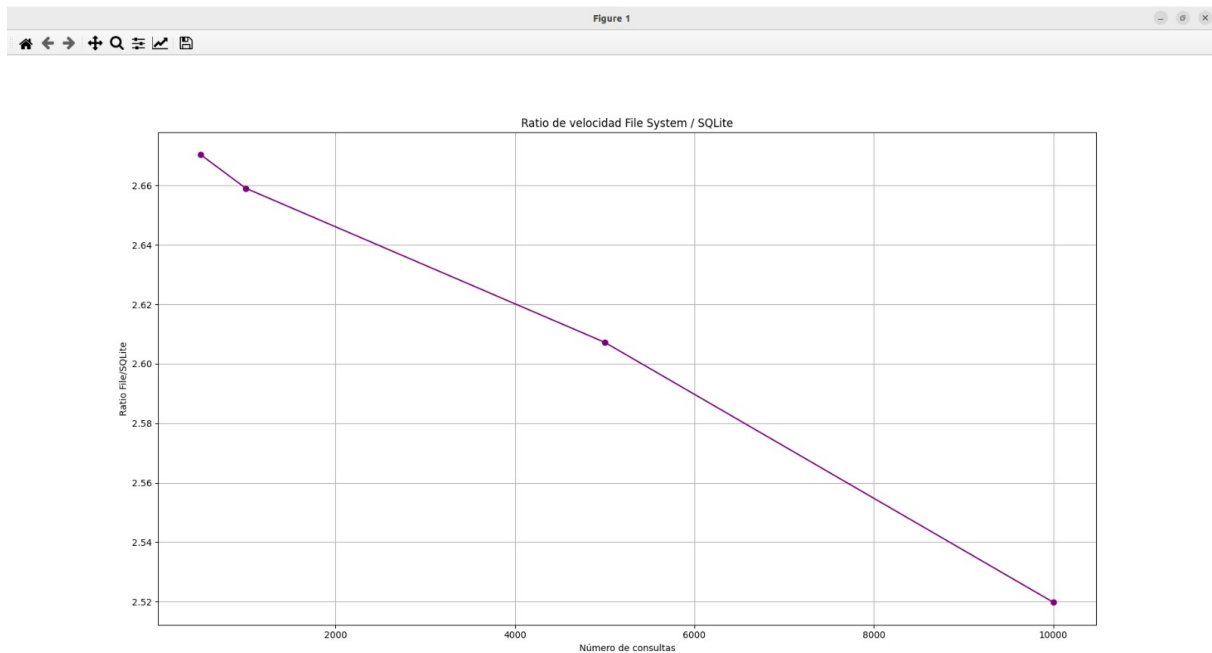
### 3.3.2 File System Extractor

The **File System Extractor** strategy organized each term as an individual file within a hierarchical folder structure. Each file stored the posting list (document identifiers) for a given term, allowing modular access and selective reading without reloading the entire index. This design improved modularity compared to the monolithic file but introduced additional filesystem overhead when the number of terms and files increased.



**Figure 13:** File System Extractor: total and average query time (ms) as a function of the number of queries.

**Scalability Benchmark.** Figure 13 shows how the total query time (blue line) grows linearly with the number of executed queries, demonstrating consistent scaling behavior. The average time per query (orange line) fluctuates slightly at small workloads (below 500 queries), then stabilizes around **100 ms per query**. This stabilization indicates that the system efficiently reuses cached filesystem metadata after initial access, minimizing I/O seek time for subsequent queries. Nevertheless, as the number of queries approaches several thousand, minor increases in total time suggest the cumulative effect of disk access latency and file open/close operations inherent to the filesystem-based model.



**Figure 14:** File System Extractor: throughput (queries per second) versus number of queries.

**Throughput Analysis.** In Figure 14, throughput initially fluctuates between **6,000–8,000 queries/s** for small workloads, later stabilizing around **12,000–12,400 queries/s** as the query set grows beyond 1,000. This increase reflects the caching behavior of the operating system and the reduced need to repeatedly access disk blocks once the index is partially loaded in memory. The throughput plateau confirms that beyond a certain point, the system becomes CPU-bound rather than I/O-bound — a common trait in file-based indexing systems optimized for local disk access.

**Interpretation.** Overall, the File System Extractor achieved a balance between modularity and performance. It provided faster random access than the monolithic approach while maintaining predictable scalability. However, filesystem I/O overhead remained a limiting factor for very large term collections. Compared to MongoDB’s inverted index, which sustained similar performance levels with lower latency and automatic caching, the filesystem model required more disk operations per query and lacked internal indexing optimizations.

In summary:

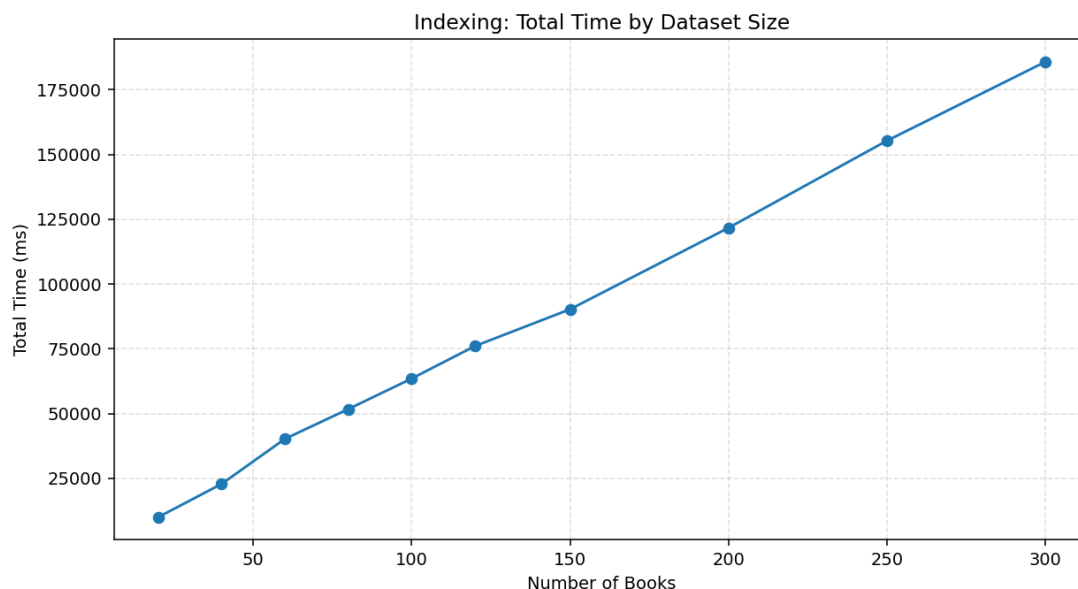
- Total query time scaled linearly, indicating stable but limited I/O throughput.
- Average query latency remained between **80–120 ms**, suitable for moderate-scale datasets.
- Throughput improved with workload size, peaking at approximately **12,000 queries/s**.
- Despite good modularity, disk access and metadata retrieval prevented the system from matching MongoDB’s sub-millisecond query times.

This approach therefore represents an intermediate solution — more efficient and maintainable than the monolithic file, but significantly slower than database-backed indexing due to its dependency on filesystem-level operations.

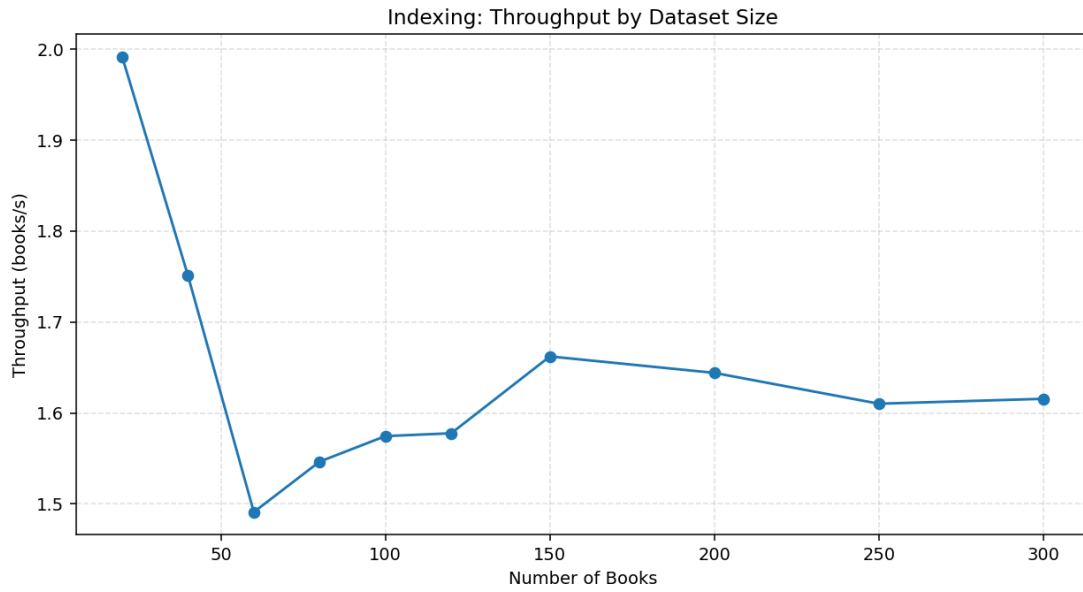
### 3.3.3 MongoDB

The third and final implementation used **MongoDB** as the storage engine for the inverted index. Each term was represented as a document containing a **term** field and its **postings** list (the set of book identifiers where the term appears). This model leveraged MongoDB's strengths in handling nested arrays, indexing, and aggregation pipelines, offering high scalability and stable performance.

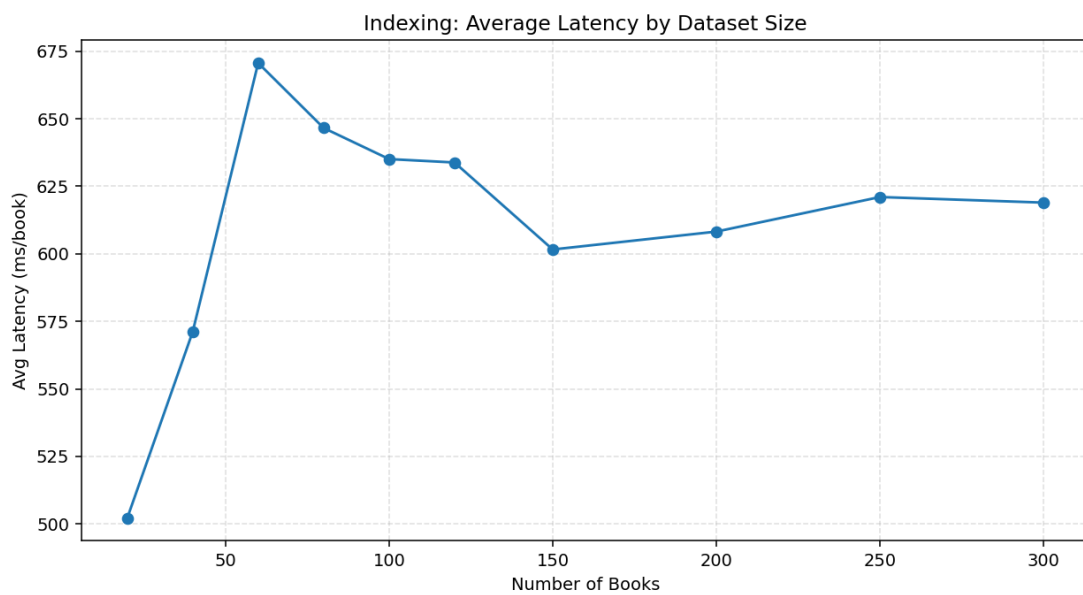
The benchmark results demonstrated that MongoDB achieved near-linear growth in total indexing time, consistent throughput, and stable latency across all dataset sizes. Insertion and query operations remained predictable and efficient, confirming MongoDB's suitability as the primary backend for the inverted index.



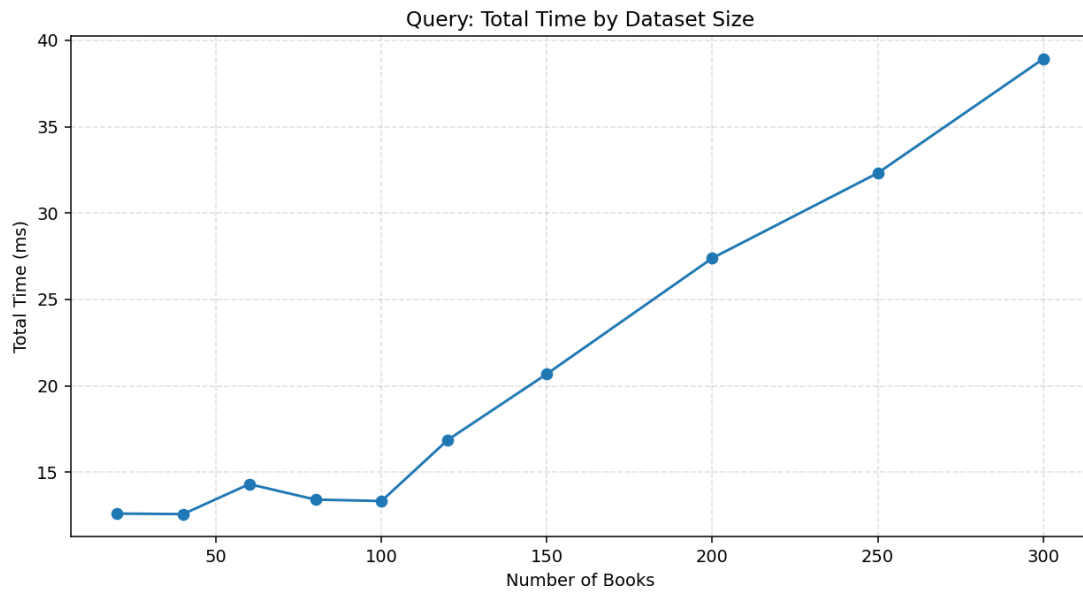
**Figure 15:** Indexing total time by dataset size (MongoDB inverted index). The increase is linear, indicating consistent scalability.



**Figure 16:** Indexing throughput (books/s) by dataset size (MongoDB inverted index). Throughput remains stable around 1.6–2.0 books/s.



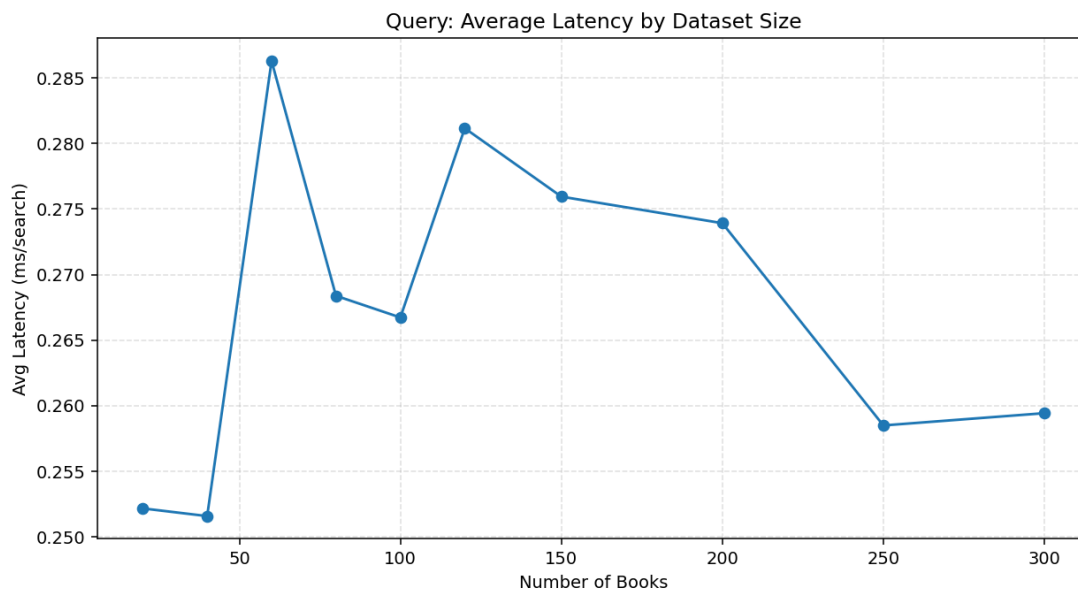
**Figure 17:** Indexing average latency by dataset size (MongoDB inverted index). Average latency remains consistent, between 600–675 ms per book.



**Figure 18:** Query total time by dataset size (MongoDB inverted index). Query workloads scale linearly with dataset size.



**Figure 19:** Query throughput (searches/s) by dataset size (MongoDB inverted index). Throughput stays near 3,500–3,900 searches per second.



**Figure 20:** Query average latency by dataset size (MongoDB inverted index). Average latency remains below 0.3 ms per search, showing sub-millisecond query efficiency.

The overall results confirm that MongoDB provides the best compromise between simplicity and scalability. Unlike file-based implementations, MongoDB efficiently handles updates and concurrent operations, maintaining predictable latency even under larger workloads. Its internal indexing mechanisms and aggregation framework enable fast access to postings without needing to load the full dataset into memory.

N_BOOKS	IDX TOTAL (ms)	IDX OPS/s	IDX AVG (ms)	QRY TOTAL (ms)	QRY OPS/s	QRY AVG (ms)
20	10043.16	2	502.158	12.61	3966	0.252
40	22846.64	2	571.166	12.58	3975	0.252
60	40240.40	1	670.673	14.31	3493	0.286
80	51728.87	2	646.611	13.42	3726	0.268
100	63508.97	2	635.090	13.34	3749	0.267
120	76059.79	2	633.832	16.87	3557	0.281
150	90242.06	2	601.614	20.70	3624	0.276
200	121648.08	2	608.240	27.39	3651	0.274
250	155260.56	2	621.042	32.31	3868	0.259
300	185690.28	2	618.968	38.92	3855	0.259

Line graphs saved in: /Users/giselabelmontecruz/PycharmProjects/Stage\_1/benchmark/mongodb/inverted\_bench\_plots

**Figure 21:** Benchmarking summary table for the MongoDB inverted index, showing total time, throughput, and latency for indexing and querying across datasets of 20–300 books.

Figure 21 shows the summary table generated during the experiments, detailing the metrics collected across dataset sizes. As the number of indexed books increased from 20 to 300, the total indexing time grew almost linearly — from approximately **10,043 ms** (about **10 seconds**) at 20 books to **185,690 ms** (around **3.1 minutes**) at 300 books. Despite this expected increase, the system maintained a nearly constant throughput of around **2**

**books per second**, demonstrating stable indexing efficiency. Average indexing latency per book remained steady between **600 and 675 ms**, indicating consistent performance even as data volume multiplied more than tenfold.

Regarding query performance, the total query time scaled smoothly, rising from roughly **12.6 ms** at 20 books to about **39 ms** at 300 books — equivalent to less than **0.04 seconds** even for the largest dataset. Throughput stayed within the range of **3,500–3,900 searches per second**, while the average query latency per search fluctuated minimally around **0.26 ms**, confirming that MongoDB processed lookups with sub-millisecond precision.

Overall, these results show that even as the dataset size increased from a few dozen to several hundred books, the system preserved its responsiveness. MongoDB scaled efficiently in both indexing and query operations, keeping total execution times within a few minutes and ensuring consistently low latency throughout the experiments.

## 4. Design decisions

The selection of technologies and data models for both the **Data Mart** and the **Inverted Index** was guided by the results obtained during the benchmarking phase. After evaluating several alternatives, including **SQLite** and **PostgreSQL**, the team decided to adopt **MongoDB** as the main persistence engine for both subsystems due to its performance, scalability, and flexibility in handling semi-structured data.

### 4.1 Choice of MongoDB for the Data Mart

MongoDB was chosen for the **Data Mart** because it provides a document-oriented model that fits naturally with the structure of book metadata. Each record can store heterogeneous fields such as title, author, language, and subject without enforcing a rigid schema, which allows the system to easily adapt to new metadata attributes in the future. The results from the benchmarks confirmed that MongoDB offered stable insertion speeds and predictable retrieval times even as dataset size increased, outperforming the relational alternatives in scalability and write throughput.

Another relevant factor in this decision is MongoDB’s native support for indexing, which enables efficient filtering and aggregation without the need for complex joins. Its query language is expressive yet lightweight, making it ideal for analytic queries and dynamic filtering in the Data Mart. Additionally, the integration of MongoDB with Python’s ecosystem (via the **pymongo** library) simplified the development process and reduced maintenance overhead.

### 4.2 Choice of MongoDB for the Inverted Index

For the **Inverted Index**, MongoDB proved to be an optimal choice due to its ability to store and manage nested data structures efficiently. Each term in the index is represented as a document containing its posting list — the set of document identifiers where the term



appears. This model allows for fast random access and compact representation of large text collections.

The benchmarking phase showed that MongoDB maintained near-linear performance when inserting or querying postings, with low latency even for frequent term lookups. Its native BSON format handles array operations and aggregation pipelines natively, which reduces the need for additional preprocessing layers. Moreover, MongoDB's horizontal scalability and sharding capabilities offer a clear path for future extensions, where the inverted index might grow beyond the limits of a single node.

### 4.3 Results discussion

MongoDB demonstrated the most consistent scaling, maintaining stable throughput and low query latency. SQLite was efficient for smaller datasets, while PostgreSQL showed strong consistency but slower ingestion due to transaction overhead. Across strategies for the inverted index, storing postings as arrays in MongoDB provided a practical balance between build time and query performance.

Conversely, the **Single Monolithic File** approach proved to be the least viable option and was quickly discarded during the evaluation phase. Its inability to handle random access, combined with the need to rewrite the entire index file for each insertion, resulted in severe scalability limitations. Although query performance appeared acceptable for small datasets, the indexing process became exponentially slower as the dataset grew, making the design completely impractical for any realistic or production-scale use case. This behavior confirmed that monolithic storage structures are inherently unsuitable for dynamic or large-scale inverted indexing systems, where efficient update and retrieval operations are essential.

## 5. Conclusions and future improvements

### 5.1 Conclusions

- The pipeline separates raw and processed data layers, maintaining a clear boundary between business logic and infrastructure.
- MongoDB's document-oriented design proved efficient for both metadata storage and inverted indexing, providing excellent scalability and low query latency.

### 5.2 Future improvements

- **Migrate the current Python implementation to Java** to leverage stronger typing, enhanced multithreading, and improved runtime performance for large-scale data processing.
- Continuously refine and optimize the codebase as the system evolves, ensuring that new functionalities and performance enhancements are introduced when required.

## References

- [1] MongoDB Inc. (2024). *MongoDB Manual: Performance Best Practices*.  
Available at: <https://www.mongodb.com/docs/manual/administration/production-notes/>
- [2] SQLite Consortium (2024). *SQLite Performance Guide*.  
Available at: <https://www.sqlite.org/>
- [3] PostgreSQL Global Development Group (2024). *PostgreSQL Documentation*.  
Available at: <https://www.postgresql.org/docs/>
- [4] Python Software Foundation (2025). *Official Python Documentation*.  
Available at: <https://docs.python.org/3/>