Big Data – Stage 1: Search Engine

Academic Year 2025/2026

Course: Big Data – Universidad de Las Palmas de Gran Canaria

Group Name: UD Playmaker Apps (PMA)

GitHub Repository: https://github.com/DieGodMF4/Playmaker-Apps-BD

Authors:

Víctor Blanco
Rafael Suárez
Diego Marrero
Sergio Hernández
Eduardo Sánchez

Universidad de Las Palmas de Gran Canaria

Grado en Ciencia e Ingeniería de Datos

October 7, 2025

Abstract

This report presents the implementation and evaluation of the data layer for a simple search engine developed as part of the Big Data course at the Universidad de Las Palmas de Gran Canaria. The system was designed to ingest, clean, and structure textual data from Project Gutenberg, building a scalable foundation for later stages of the project. The work includes the development of a crawler, a metadata storage component, an inverted index, and a control layer that orchestrates the data pipeline. Several data structures and storage strategies were benchmarked to evaluate performance and scalability. The results demonstrate that the architecture effectively supports large-scale data ingestion and retrieval while maintaining modularity and clarity of design.

1. Introduction and Objectives

The primary goal of this project is to design and implement the **data layer** of a search engine. This stage establishes the foundation for the subsequent phases of the system, focusing on efficient data ingestion, cleaning, storage, and indexing. The following objectives guide the implementation:

- Develop a crawler capable of downloading and processing books from Project Gutenberg.
- Build a datalake that stores raw and cleaned data in a traceable and organized hierarchy.
- Design datamarts that separate structured metadata from unstructured text content.
- Implement and benchmark different data structures for inverted indexing.
- Create a lightweight control layer that coordinates the data flow.

2. System Architecture

The system is organized into modular layers representing distinct responsibilities in the pipeline.

- **Crawler:** Downloads books from Project Gutenberg and separates headers (metadata) from bodies (text).
- **Datalake:** Stores raw and cleaned data, organized by ingestion date and hour.
- **Datamart:** Contains structured metadata and the inverted index.

- **Control Layer:** Coordinates downloads, cleaning, and indexing, ensuring fault tolerance and progress tracking.

The entire architecture has been designed to simulate a realistic big data ingestion pipeline, capable of scaling horizontally and integrating future stages (microservices in Java).

3. Design Decisions

3.1. Datalake Organization

The datalake serves as the central storage layer. Its directory hierarchy follows a date/hour convention:

```
datalake/
  YYYYMMDD/
    HH/
      <book_id>_header.txt
      <book_id>_body.txt
```

This structure provides traceability and incremental ingestion. It supports distributed processing, avoiding bottlenecks from large directories. Naming conventions ensure that metadata and text content remain synchronized.

3.2. Metadata Storage

The metadata component extracts structured information (title, author, language) from book headers using regular expressions. This data is stored in relational and NoSQL systems for comparison.

Three engines were implemented:

- **SQLite:** Fast and lightweight; ideal for local and small-scale use.
- **MongoDB:** Flexible schema; excellent for large-scale and distributed systems.

```python
def control_pipeline(  2 usages  Diego Marrero Ferrera +1
    target_new_downloads: int = 10,
    datalake_root: Path = Path("data/datalake"),
    raw_root: Path = Path("data/raw"),
    total_tries: int = DEFAULT_TOTAL_TRIES,
) -> bool:
    """
    Runs one CONTROL 'tick':
      1) Index any downloaded-but-not-indexed books (index ALL of them).
      2) If nothing to index, try to download up to 'target_new_downloads' NEW books.
         As soon as we download at least one, we return True (so the caller can loop and index them).
    Returns:
      True  -> some progress was made (indexed or downloaded)
      False -> nothing happened (no candidates; downloads all failed/duplicates)
    """
    CONTROL_PATH.mkdir(parents=True, exist_ok=True)

    downloaded = _read_ids(DOWNLOADS)
    indexed = _read_ids(INDEXINGS)
```

Figure 1: Main orchestration function `control_pipeline()` coordinating the data workflow.

SQLite was chosen as the baseline implementation due to its simplicity and integration ease, while MongoDB was used for performance testing. The schema used was:

`book_id | title | author | language`

3.3. Inverted Index

The inverted index maps each term to the documents where it appears. The implementation includes three alternative structures:

1. **Monolithic JSON:** All term-document mappings stored in a single JSON file.
2. **MongoDB:** Terms as documents with arrays of book IDs.
3. **SQLite:** Terms and their postings are stored in a relational table where each term is a primary key and its associated list of book IDs is serialized as text.

Each method has trade-offs in complexity, performance, and scalability.

3.4. Control Layer

The control layer ensures synchronization between modules. It tracks which books have been downloaded and indexed using two files: `downloaded_books.txt` and `indexed_books.txt`. This provides resilience and allows the pipeline to resume after interruptions. Its logic dynamically decides whether to trigger new downloads or index pending books.

3.5. Pipeline Entry Point

The `run_pipeline.py` script manages full automation. It defines parameters such as the number of downloads per run and the maximum number of iterations. It integrates logging for auditability and reproducibility, ensuring that every pipeline execution leaves a detailed log trace.

4. Benchmarks and Results

The system was benchmarked to evaluate both the metadata layer and the inverted index layer using three storage engines: SQLite, MongoDB, and JSON. These experiments aimed to measure insertion time, query latency, and scalability. Each test was performed using a consistent dataset of 20 books downloaded from Project Gutenberg to ensure comparability across all engines.

4.1. Experimental Setup

All experiments were conducted in a controlled environment on the same hardware configuration. Each benchmark script was executed multiple times, and the mean values were reported. The tests were divided into two categories:

1. **Metadata Storage Benchmark:** measuring insertion and query times for book records.
2. **Inverted Index Benchmark:** measuring index build time and average query latency for retrieving matching documents.

Both benchmarks were implemented as standalone Python scripts located in the `benchmarks/` directory, ensuring that the pipeline's internal logic remained unaffected by the testing procedures.

4.2. Metadata Benchmark

This benchmark evaluated how efficiently each database stored and retrieved structured metadata (title, author, language). The results are summarized in Table 1.

Table 1: Performance of metadata insertion and query operations.

Engine	Records	Insert (s)	Query Author (s)	Query ID (s)
SQLite	20	0.000149	0.000008	0.000005
MongoDB	20	0.002520	0.001522	0.001594

SQLite clearly outperformed MongoDB for this workload, achieving insertion times nearly **17 times faster** and query latencies more than **200 times faster**. This performance advantage stems from SQLite's in-memory operations and its extremely lightweight architecture. Since SQLite operates locally and does not require network communication or serialization overhead, it excels at low-volume, single-node operations.

By contrast, MongoDB introduces a client-server communication layer, as well as additional overhead for document management and BSON serialization. These factors make it less efficient for small datasets or local testing environments. However, it is important to note that MongoDB's performance penalty in this stage is expected; it is designed to handle **horizontal scalability**, high concurrency, and distributed writes—features that are not yet fully utilized in Stage 1.

Therefore, while SQLite demonstrates superior raw speed for the current setup, MongoDB's advantages will become evident in later stages, when data volume and concurrency increase. In addition, MongoDB's flexible schema and support for horizontal sharding make it ideal for dynamic datasets that may evolve over time.

4.3. Inverted Index Benchmark

The inverted index benchmark assessed how efficiently each engine constructed and queried the inverted index, which maps terms to document identifiers. Table 2 summarizes the performance results for JSON, SQLite, and MongoDB implementations.

Table 2: Inverted index construction and query times.

Engine	Documents	Build (s)	Query 50 (s)
JSON	20	0.186164	0.000057
SQLite	20	0.915841	0.000614
MongoDB	20	17.909873	0.028310

The JSON-based index showed the best overall performance for this dataset. Its build time was more than **4.9 times faster than SQLite** and approximately **96 times faster than MongoDB**. This is primarily because JSON writes occur sequentially in a single file without transactional locks or database-level consistency checks. As such, it provides minimal overhead and maximum throughput—ideal for rapid prototyping or small-scale systems.

SQLite, while slower to build, offers transactional integrity and a structured schema, ensuring data persistence and consistency even in case of unexpected interruptions. Its query performance was reasonable, completing in 0.000614 seconds, which remains sufficiently fast for moderate datasets.

MongoDB, on the other hand, exhibited significantly higher build and query times, largely due to its network abstraction layer and indexing mechanisms. Each term insertion in MongoDB involves additional write acknowledgment and index management operations, which adds latency when processing small batches. However, these mechanisms ensure data durability and optimized performance for large-scale distributed environments. MongoDB's indexing becomes more advantageous as datasets scale into the millions of documents, where it can parallelize queries across shards and replicas.

4.4. Comparative Discussion

To better understand the trade-offs, it is useful to categorize each approach based on three dimensions: **speed**, **scalability**, and **reliability**.

- **Speed:** SQLite and JSON consistently outperform MongoDB for small-scale operations. JSON excels in build time due to its simplicity, while SQLite achieves the lowest query latency thanks to its efficient indexing mechanisms.
- **Scalability:** MongoDB clearly dominates in potential scalability. Although its overhead is noticeable with 20 documents, it supports replication, sharding, and distributed queries—capabilities essential for large-scale search engines.

- **Reliability:** SQLite provides a strong middle ground with ACID-compliant transactions and minimal setup. JSON lacks concurrency control and can become corrupted if multiple processes access it simultaneously, while MongoDB ensures robust fault tolerance through journaling and replication.

Overall, the results reveal that:

1. SQLite is the **best performer** for local and small datasets.
2. JSON is the **simplest and fastest** for prototyping and sequential writes.
3. MongoDB is the **most scalable and future-ready**, but requires larger datasets to fully exploit its architecture.

These findings align with the project's progression plan: use SQLite for Stage 1 (Python prototype) and migrate to MongoDB during Stage 2 (Java microservices). The benchmarks confirm that the design is consistent with real-world data engineering practices, balancing simplicity with scalability.

5. Performance Discussion

The benchmark data highlights clear trade-offs:

- **SQLite** excels at rapid local processing and minimal configuration.
- **MongoDB** offers robustness for future distributed systems but incurs higher overhead on small sets.
- **JSON** is ideal for prototyping and single-user environments but lacks concurrency control.

From a scalability standpoint, MongoDB's architecture is more suited for a microservice ecosystem, aligning with the project's future Stage 2 objectives. The hierarchical datalake and modular Python architecture further contribute to system extensibility.

6. Conclusions and Future Work

The Stage 1 data layer achieves all goals set for this milestone:

- A functioning pipeline from crawling to indexing.
- Modular structure separating datalake, datamart, and control logic.
- Extensive benchmarking demonstrating strengths and trade-offs between engines.

Future improvements include:

- Expanding to distributed microservices in Java.
- Implementing parallel downloads and asynchronous indexing.
- Integrating real-time query APIs for search functionality.

References

- Project Gutenberg: `https://www.gutenberg.org`
- Big Data Course Materials, Universidad de Las Palmas de Gran Canaria, 2025/2026.