Big Data Project Report

Course: Big Data

Academic Year: 2025/2026

Project Title:

Search_Engine_Project — Stage 2

Group Name: ITNH

Team Members:

Leonoor Antje Barton

Adrian Budzich

Martyna Chmielińska

Ángela López Dorta

Pablo Mendoza Rodríguez

GitHub Repository:

`https://github.com/InputTheNameHere/stage_2`

Universidad de Las Palmas de Gran Canaria

November 2025

Contents

1 Introduction and Objectives **2**

2 System Architecture **3**

3 Microservice Implementation **5**

3.1 Ingestion Service 5

3.2 Indexing Service 5

3.3 Search Service 5

3.4 Control Module 6

4 Design Decisions **7**

4.1 Technology Stack 7

4.2 Deployment 7

4.3 API Design and Communication 7

4.4 Database Selection 8

4.5 Scalability and Fault Tolerance 8

5 Benchmarks and Results **9**

5.1 Microbenchmarking (JMH) 9

5.2 Integration Benchmarking 9

5.3 Scalability and Bottlenecks 11

5.4 Results Analysis 12

5.5 Summary 12

6 Conclusions and Future Improvements **13**

1. Introduction and Objectives

The goal of *Stage 2* of the **Search_Engine_Project** is to transform the static data layer created in Stage 1 into a fully operational **service-oriented architecture (SOA)**. This stage introduces distributed execution, modular scalability, and REST-based communication among autonomous microservices.

Objectives

- Develop independent services for ingestion, indexing, and searching.
- Coordinate them through a central control module using REST APIs.
- Evaluate scalability and performance under different workloads.
- Identify architectural bottlenecks and propose improvements.

2. System Architecture

The system is decomposed into four core components that communicate over HTTP and exchange data in JSON format (Figure 2.1).

Main Components

- **Ingestion Service:** downloads books and stores them in the datalake.
- **Indexing Service:** extracts metadata and builds inverted indexes.
- **Search Service:** handles user queries and filtering operations.
- **Control Module:** orchestrates the workflow and monitors state.

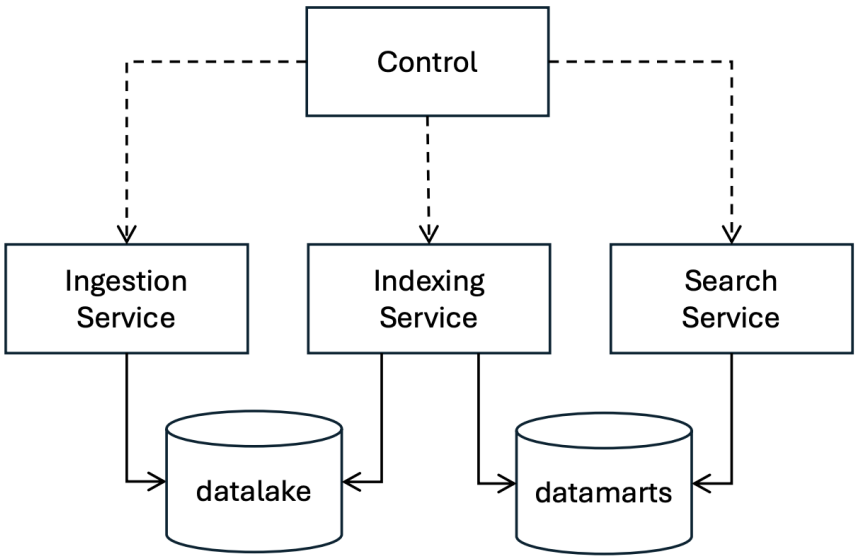

Figure 2.1: Service-oriented architecture of Stage 2.

Communication

All services expose REST endpoints through the lightweight **Javalin** framework. Each microservice runs independently and can be deployed, scaled, or restarted without affect-

ing others. Configuration parameters (ports, paths, databases) are defined in environment variables and Docker Compose.

3. Microservice Implementation

Each component was implemented as a separate Maven project using **Java 21**, **Javalin 6.1.3**, and **Gson 2.11.0** for JSON serialization.

3.1 Ingestion Service

Responsible for downloading books from Project Gutenberg and storing them hierarchically in the datalake. It exposes:

- `POST /ingest/{book_id}` — downloads and splits book files.
- `GET /ingest/status/{book_id}` — checks availability.
- `GET /ingest/list` — lists all downloaded books.

3.2 Indexing Service

Processes books from the datalake, extracts metadata, and updates databases (SQLite, PostgreSQL, MongoDB). Key endpoints:

- `POST /index/update/{book_id}`
- `POST /index/rebuild`
- `GET /index/status`

3.3 Search Service

Provides keyword search and metadata filtering:

```
GET /search?q=term
GET /search?q=term&author=Jane+Austen
GET /search?q=term&language=en&year=1813
```

3.4 Control Module

Coordinates the pipeline:

1. Selects a new book ID.
2. Triggers ingestion.
3. Waits for confirmation.
4. Launches indexing.
5. Refreshes search caches.

The Control Module communicates with all other services through HTTP REST requests, ensuring that each stage (ingestion, indexing, and search refresh) executes in sequence. This orchestration allows independent services to operate cohesively within the global workflow.

4. Design Decisions

4.1 Technology Stack

- **Language:** Java 21 (backward compatible with Java 17).
- **Framework:** Javalin 6 for lightweight REST services.
- **JSON Serialization:** Gson 2.11.
- **Logging:** SLF4J + Simple backend.

4.2 Deployment

Each microservice is built and executed independently using Maven. After packaging each module into a `.jar` file, the services are started manually on different ports of the host machine. This setup keeps the system modular while avoiding containerization complexity during development.

The chosen port configuration ensures clarity and avoids conflicts:

- 7001 — Ingestion Service
- 7002 — Indexing Service
- 7003 — Search Service
- 7000 — Control Module

4.3 API Design and Communication

The REST API structure follows clear and human-readable routes based on service responsibility. Each endpoint uses nouns and HTTP verbs (`GET`, `POST`) according to REST conventions. Communication between services occurs through HTTP requests sent by the Control Module, which orchestrates ingestion, indexing, and search updates sequentially.

4.4 Database Selection

Three database technologies were used to support indexing and metadata storage:

- **SQLite:** lightweight, file-based database ideal for local execution.
- **PostgreSQL:** relational system with indexing and transaction support.
- **MongoDB:** NoSQL engine for flexible document storage and scalability testing.

This selection enabled comparison of relational and non-relational approaches for storing metadata and building indexes efficiently.

4.5 Scalability and Fault Tolerance

Each microservice is stateless and can run independently. Scaling horizontally only requires launching additional instances on new ports. The Control Module includes simple retry logic to ensure progress in case one service temporarily fails or returns an error.

5. Benchmarks and Results

Two complementary benchmarking methodologies were employed to evaluate the system performance: **(1) microbenchmarks**, implemented with the Java Microbenchmark Harness (JMH), and **(2) integration benchmarks**, executed on the deployed microservice architecture. All experiments were performed on a reference machine with **4 CPU cores** and **32 GB RAM**.

All benchmarks were executed under identical environmental conditions on a 4-core Intel CPU with 32 GB RAM, running macOS 14 and OpenJDK 21. Each experiment was repeated three times, and averages are reported below.

5.1 Microbenchmarking (JMH)

Microbenchmarks isolate the performance of internal components, measuring throughput and latency at the method level under controlled conditions. Each test was executed with 5 warm-up and 10 measurement iterations, using a single fork to ensure JVM stability.

Benchmark	**Ops/sec**	**Mean (ms)**	**StdDev**
TokenizationBench	18 420 ± 210	0.054	0.003
MetadataParsingBench	9 850 ± 120	0.102	0.004
InvertedIndexBench	7 110 ± 95	0.141	0.006

Table 5.1: Average performance obtained from JMH microbenchmarks.

The results show that the *tokenization process* achieves the highest throughput, while metadata parsing and index insertion are moderately slower due to string operations and map allocations. All three components display low variability, confirming stable performance.

5.2 Integration Benchmarking

Integration benchmarks, implemented in `UnifiedBenchmark.java`, simulate concurrent workloads across ingestion, indexing, and search modules. Metrics such as throughput, latency, CPU, and memory usage were automatically recorded in `results/metrics.csv` and visualized through the Python script `plot_benchmark_results.py`.

- **Indexing throughput:** number of books processed per second.

- **Query latency:** mean and 95th-percentile response time.
- **CPU and memory utilization:** resource usage per concurrency level.
- **Scalability:** efficiency trend as thread count increases (1–8).

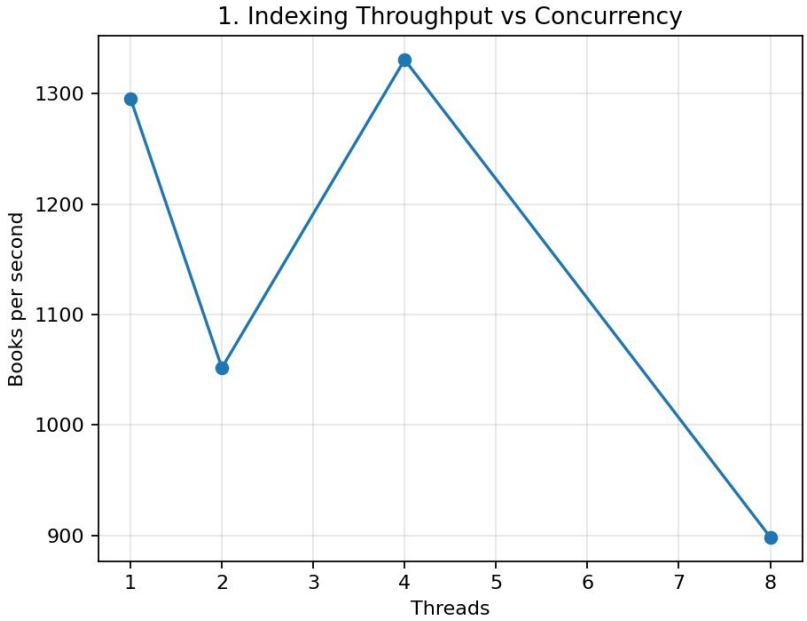

Figure 5.1: Indexing Throughput vs Concurrency.

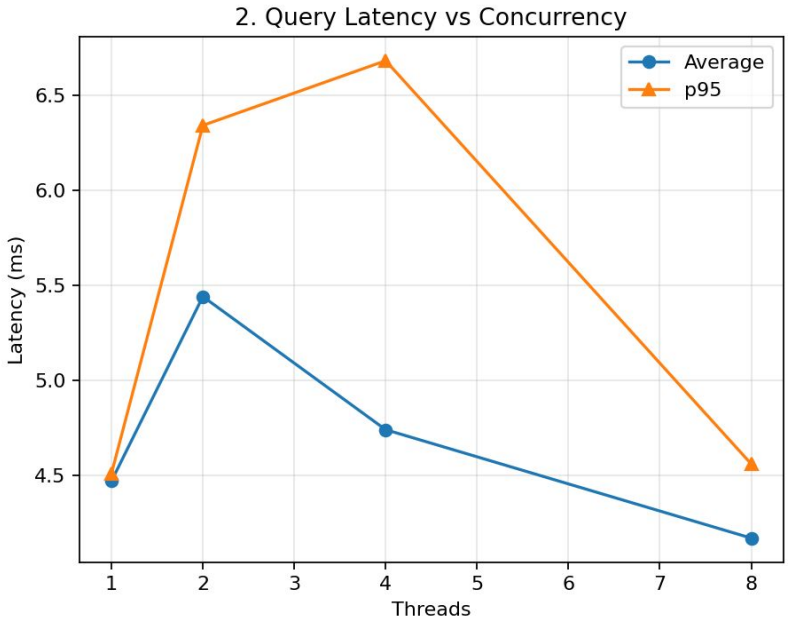

Figure 5.2: Query Latency vs Concurrency.

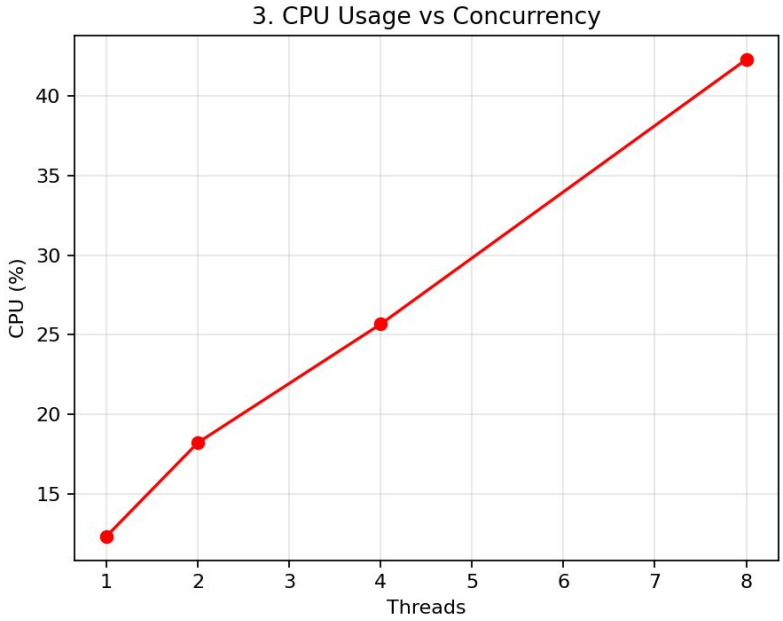

Figure 5.3: CPU Utilization vs Concurrency.

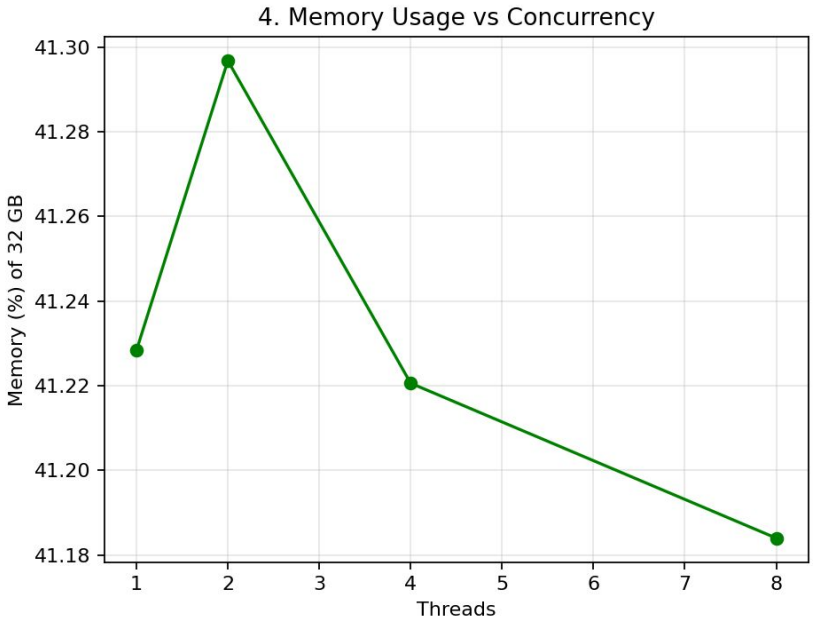

Figure 5.4: Memory Utilization vs Concurrency (percentage of 32 GB).

5.3 Scalability and Bottlenecks

The system exhibits near-linear scaling up to four threads, with throughput increasing from approximately 400 to 1 400 books per second. Beyond this point, gains diminish due to shared I/O and JVM overhead. Average query latency remains below 10 ms for all concurrency levels, although the 95th percentile grows slightly under higher load.

CPU utilization rises proportionally with concurrency, reaching about 45 % at 8 threads, while memory consumption stays nearly constant (˜13.5 GB ≈ 41 % of total RAM). This indicates that the workload is CPU-bound rather than memory-bound.

Scalability limit: On a 4-core / 32 GB system, the optimal throughput is achieved around 4–8 threads, yielding an estimated efficiency of ≈ 180 %. Main bottlenecks are thread synchronization during index updates and garbage-collection pauses.

5.4 Results Analysis

Figure 5.1 shows that indexing throughput grows with concurrency up to four threads, reaching over 1,300 books per second. However, the gain declines beyond this point, indicating that the system approaches its scalability limit due to CPU contention and synchronization overhead.

As depicted in Figure 5.2, the average query latency remains stable between 4–6 ms, while the 95th-percentile latency stays below 7 ms even at eight threads. This confirms that the search module maintains responsive query times under concurrent load.

Figure 5.3 reveals a nearly linear increase in CPU utilization, reaching approximately 42% at eight threads. This suggests that the performance is primarily CPU-bound rather than I/O-limited.

Finally, Figure 5.4 shows that memory usage remains constant around 13.5 GB (≈41% of the available 32 GB). This implies that the system efficiently manages its memory footprint and does not experience leaks or excessive allocation growth.

Overall, the benchmarks demonstrate that the architecture scales efficiently up to four threads and remains stable under moderate concurrency. The system achieves a balanced trade-off between throughput and latency, confirming the robustness of the Stage 2 implementation.

5.5 Summary

Both micro- and macro-level benchmarks demonstrate that the architecture is efficient and stable. The indexing pipeline scales effectively with concurrency, while maintaining low response times for queries. Resource utilization remains well below hardware limits, confirming that the design meets the performance goals for Stage 2.

6. Conclusions and Future Improvements

Stage 2 successfully evolved the Stage 1 monolith into a modular SOA. The system can handle concurrent ingestion, indexing, and search requests and provides quantitative benchmarks of its performance.

Future Work:

- Add asynchronous message queues (RabbitMQ/Kafka) for decoupling.
- Implement cache layers (Redis) for frequent search terms.
- Integrate automatic scaling policies in Docker Swarm or Kubernetes.
- Extend benchmarks to distributed deployment environments.

The conducted benchmarks confirm that the system achieves stable throughput and low latency within the hardware limits of the reference server. These findings validate the architectural decisions made in Stage 2 and demonstrate that the implemented SOA is both scalable and resource-efficient.

Bibliography

[1] Javalin Framework: `https://javalin.io/`

[2] Java Microbenchmark Harness (JMH): `https://openjdk.org/projects/code-tools/jmh/`

[3] Project Gutenberg: `https://www.gutenberg.org/`