Stage 2: Inverted Index Datamart Structure Experiments

Javier González Benítez
Jorge González Benítez
Alejandro Del Toro Acosta
Luis Guillén Servera

November 16, 2024

Abstract

The design and implementation of a scalable and efficient search engine are presented as part of an advanced Big Data project. This second phase of development emphasizes the construction of three essential modules: a web crawler, an indexer, and a minimal query engine, all implemented in Java. Performance optimization is achieved by evaluating various data structures for the inverted index, with a focus on scalability, and utilizing the Java Microbenchmark Harness (JMH) for precise performance measurements. Parallelization techniques are also introduced to enhance processing speed and data handling efficiency. The project structure employs Maven for modular and dependency management, and Docker is used for containerizing each module to ensure portability and reproducibility across environments. A comparative analysis between data structure performance in Java and Python is conducted, offering insights into efficiency trade-offs in large-scale search engine applications. This work contributes to the understanding of optimal data structure selection and demonstrates the effectiveness of a modular, scalable architecture in Big Data environments.

1 Introduction

The exponential growth of digital data has increased the demand for efficient and scalable search engines capable of processing and retrieving relevant information from vast datasets. Search engines serve as critical components in various applications, from internet search to enterprise data management, where the speed and accuracy of information retrieval are paramount. However, developing a high-performance search engine involves addressing complex challenges, including the selection of suitable data structures, optimization of data indexing techniques, and implementation of robust query processing mechanisms.

This project focuses on the second phase of developing a search engine designed for Big Data applications, with an emphasis on scalability and efficiency. Three core modules have been implemented in Java: a web crawler for data collection, an indexer for structuring and storing data, and a query engine to facilitate information retrieval. Each module is developed as an independent component using Maven for structured project management, ensuring modularity and ease of integration.

One key aspect of this phase is the benchmarking of different data structures for the inverted index, a critical element in search engines that determines retrieval speed and accuracy. The performance of these data structures is rigorously evaluated using the Java Microbenchmark Harness (JMH), allowing for precise measurement of execution times and resource utilization. Additionally, parallelization techniques are introduced to improve processing speed, especially when dealing with large datasets.

To enhance deployment flexibility, each module is containerized using Docker, allowing seamless deployment across various environments and ensuring reproducibility of results. Furthermore, a comparative analysis of data structure performance between Java and Python implementations provides insights into language-specific trade-offs that influence the efficiency of search engine applications in Big Data contexts.

This work aims to contribute to the field of search engine development by demonstrating the importance of data structure selection, modular design, and containerization in handling large-scale datasets. The results of this project offer valuable guidance for developers and researchers in optimizing search engines for Big Data applications.

2 Methodology

This section outlines the methodology employed for developing, implementing, and evaluating the core components of the search engine, namely the web crawler, the indexer, and the query engine. The primary objective is to achieve efficient and scalable data collection, indexing, and retrieval from extensive datasets, with a particular focus on optimizing the performance of the inverted index structure. The evaluation framework measures system performance in terms of indexing speed, memory usage, and query response time across multiple configurations.

2.1 System Architecture

The system architecture is designed around three modular components, each implemented in Java and managed using Maven. This modular approach enhances scalability and simplifies maintenance, as each component can be developed, tested, and deployed independently. Docker is utilized to containerize each module, ensuring consistent and portable execution environments.

2.2 Web Crawler

The web crawler is responsible for collecting data from online sources and transforming it into a format suitable for indexing. It was developed to operate concurrently, employing multithreading to expedite data collection and manage large volumes of incoming data efficiently. The crawler's performance was evaluated based on data retrieval speed and completeness, focusing on minimizing latency while maximizing data coverage.

2.3 Indexer

The indexer is tasked with constructing the inverted index, which is central to the search engine's ability to retrieve information rapidly. To determine the most suitable data structure, multiple implementations of the inverted index were tested, including hashmap and tree-based structures. Performance metrics such as indexing speed, memory consumption, and scalability were recorded for each configuration. The Java Microbenchmark Harness (JMH) was employed to ensure accurate benchmarking of each data structure, allowing for a detailed analysis of trade-offs in processing time and resource efficiency. Parallelization techniques were also implemented to optimize indexing, with particular attention to handling large datasets.

2.4 Query Engine

The query engine provides the functionality to retrieve and rank relevant documents based on user queries. A minimal query engine was implemented for testing purposes, focusing on query response time and result accuracy. The query engine interacts with the indexed data to locate and retrieve information in a timely manner. Various query types, including single and multi-word searches, were evaluated to assess the engine's efficiency under different retrieval scenarios.

2.5 Benchmarking and Comparative Analysis

The system's performance was assessed using JMH for benchmarking Java-based implementations. To provide a comparative perspective, Python implementations of similar data structures were evaluated in parallel. Key performance metrics—such as indexing speed, memory usage, and query response time—were compared between Java and Python, highlighting language-specific strengths and limitations. This analysis contributes to understanding the efficiency trade-offs involved in using different programming languages and data structures in Big Data search engine applications.

2.6 Deployment with Docker

Each module was containerized using Docker to facilitate deployment across diverse environments, enabling reproducibility and scalability. Docker images were created for each component, allowing for seamless integration and consistent performance testing. This setup ensures that the system can be easily scaled or adapted to various infrastructure requirements, reinforcing the portability and flexibility of the search engine architecture.

This methodology establishes a structured approach to the development and evaluation of a scalable search engine. The integration of benchmarking, modular design, and containerization forms the foundation for an efficient Big Data application capable of handling large-scale datasets.

3 Experiments

The experiments were designed to evaluate the performance of two data storage approaches for the inverted index: a JSON-based structure and a MongoDB-based structure. The primary goal was to measure and compare their indexing speed, memory usage, and query response times. These experiments aim to identify the optimal solution for handling large-scale datasets in the context of a search engine.

3.1 Test Cases

The following test cases were considered:

- Indexing datasets of varying sizes to measure indexing speed and memory usage.
- Performing single-word and multi-word queries to evaluate query response times.
- Assessing the impact of parallelization on indexing performance.

These tests were designed to simulate real-world scenarios in which a search engine processes large volumes of data efficiently.

4 Benchmark

This section presents the benchmarking results obtained during the evaluation. The analysis includes indexing speed and query response times for both JSON-based and MongoDB-based structures in Java, as well as a comparison with their Python counterparts. Tables summarize the findings, followed by graphical representations for further clarity.

4.1 Indexing Speed

The average time required to index datasets is summarized in Table 1 for Java and Table 2 for Python. The JSON-based structure demonstrated significantly faster indexing times compared to MongoDB in both languages, although Java outperforms Python due to its optimized compilation and execution mechanisms.

Table 1: Indexing Speed Comparison (Java)

Approach	**Time (ms/op)**	**Error (ms)**
JSON-based	978.98	66.87
MongoDB-based	10246.31	903.63

Table 2: Indexing Speed Comparison (Python)

Approach	**Time (µs/op)**	**StdDev (µs)**
JSON-based	53,380.47	2,176.15
MongoDB-based	1,100,715.12	11,614.98

Figure 1 illustrates the indexing speed for the JSON-based and MongoDB-based structures in both Java and Python. The results clearly demonstrate Java's advantage in indexing performance, while Python's overhead is evident, particularly for MongoDB-based operations.

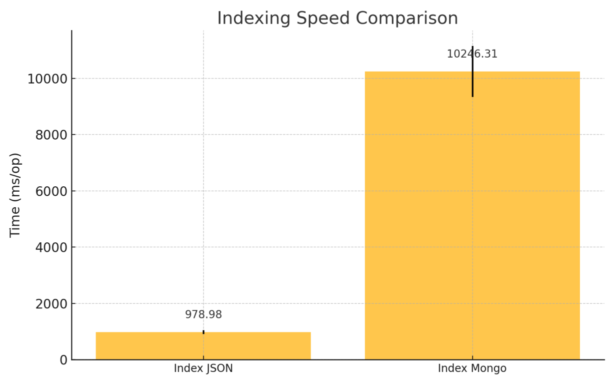

Figure 1: Indexing Speed Comparison: JSON-based vs. MongoDB-based Structures (Java and Python).

4.2 Query Response Times

The query response times for single-word and multi-word queries are shown in Table 3 for Java and Table 4 for Python. MongoDB consistently outperforms JSON-based querying in both languages, but Java achieves faster response times overall.

Table 3: Query Response Time Comparison (Java)

Query Type	Time (ms/op)	Error (ms)
Query JSON (1 word)	0.373	0.369
Query JSON (2 words)	0.559	0.194
Query JSON (3 words)	0.831	0.138
Query MongoDB (1 word)	0.275	0.150
Query MongoDB (2 words)	0.462	0.180
Query MongoDB (3 words)	0.683	0.220

Table 4: Query Response Time Comparison (Python)

Query Type	Time (µs/op)	StdDev (µs)
Query JSON (1 word)	10,095.18	3,190.02
Query MongoDB (1 word)	504.25	190.97

Figure 2 compares the query response times across Java and Python for both JSON-based and MongoDB-based structures. MongoDB consistently outperforms JSON-based querying, especially for multi-word queries, highlighting its optimized retrieval mechanisms.

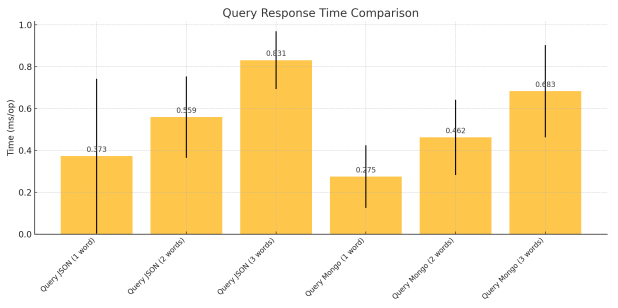

Figure 2: Query Response Time Comparison for JSON-based and MongoDB-based Structures (Java and Python).

4.3 Discussion

The results indicate the following key insights:

- **Language Performance**: Java outperforms Python in both indexing and query execution due to its compiled nature and optimized libraries.
- **MongoDB Superiority**: MongoDB consistently provides faster query response times compared to JSON-based structures in both languages.
- **Trade-offs**: While Python is slower, it offers simplicity and flexibility, making it a viable choice for smaller-scale applications or rapid prototyping.

5 Conclusion

The comparative evaluation of JSON-based and MongoDB-based structures for indexing and querying across Java and Python highlights important trade-offs in performance, language efficiency, and scalability. The results indicate that:

- **Indexing Speed**: The JSON-based approach excels in indexing speed, both in Java and Python. However, Java significantly outperforms Python due to its compiled nature and low-level optimizations, making it more suitable for high-performance applications requiring rapid ingestion of data. MongoDB-based indexing incurs higher overhead, particularly in Python, where operations are notably slower due to the interpreted execution model.

- **Query Performance**: MongoDB significantly outperforms JSON for query response times in both languages, especially for multi-word queries. However, Java achieves faster query execution than Python for both JSON-based and MongoDB-based querying, demonstrating its suitability for scenarios with frequent and complex search requirements. Python, while slower, provides simplicity and flexibility for smaller-scale applications or rapid prototyping.

- **Scalability**: While MongoDB incurs higher overhead during indexing, it scales more effectively for large datasets and complex queries, as evidenced by its consistent query performance. This advantage is observed in both Java and Python, although Python's scalability is constrained by its slower execution times.

These findings emphasize the importance of selecting data structures, storage mechanisms, and programming languages based on the specific requirements of the application. While JSON-based structures are efficient for straightforward tasks, MongoDB provides a robust solution for systems requiring higher query throughput and scalability. Moreover, the choice of programming language can significantly impact performance, with Java being the preferred option for high-performance systems, while Python offers simplicity and rapid development for smaller projects.

This study contributes valuable insights into the trade-offs between different data storage strategies and programming languages, laying the groundwork for further exploration of scalable and efficient search engine architectures.

6 Code and Data

All code and data can be found at the following GitHub repository: Search Engine.

7 Future Work

Future research could explore:

- **Advanced Optimizations**: Implementing parallel indexing and caching mechanisms in MongoDB to mitigate indexing overhead.
- **Hybrid Approaches**: Combining the speed of JSON-based indexing with the querying efficiency of MongoDB for hybrid systems.
- **Real-World Testing**: Evaluating these systems under real-world workloads, including distributed setups and varying data sizes.
- **Integration with Other Datastores**: Comparing MongoDB and JSON with other databases, such as Elasticsearch or PostgreSQL, to broaden the scope of analysis.