# Inverted Amigos Stage 2

Żukowska, Radosława Sajdokova, Anna

Oczowiński, Tymoteusz Perdomo de Vega, José Luis

López Fortes, Eduardo

17 November 2024

**Abstract**

This paper presents an enhanced and modular search engine framework fully transitioned to Java, leveraging modern libraries and best practices to achieve improved performance and scalability. The new implementation adheres to a professional Maven-like project structure, enabling clear separation of concerns, version control, and ease of maintenance. Key enhancements include adopting Java modules for storing file with `MessagePack`, `.dat` files and `concurrent.futures` for advanced parallelism, addressing bottlenecks identified in the previous version. Additionally, the system introduces a more sophisticated indexing pipeline utilizing trie-based, hashed inverted and file-per-word index structures, significantly reducing query times and improving data retrieval consistency.

Performance benchmarking demonstrates a 35% reduction in mean execution time for large datasets compared to the prior implementation. This iteration also explores advancements in data serialization through `orjson`, achieving faster I/O operations while maintaining data integrity. The proposed framework exemplifies the practical adoption of Java in professional-grade applications, overcoming traditional limitations by integrating modularity and concurrency paradigms. This work contributes a robust, open-source foundation for scalable search engines and serves as a blueprint for developing Java-based, enterprise-level data systems.

# 1 Introduction

In the previous stage, the first implementations of the inverted index were proposed. There a whole project was implemented in Python. As much as Python offers easy of implementation and deployment, it does not offer the best performance of programs in terms of time. By transitioning the project to Java, integrating modular programming practices, and adopting the Maven archetype, we have achieved notable improvements in both performance and professionalism.

The previous version of the project suffered from critical limitations, becoming excessively slow and inefficient as the number of books increased slightly. This issue severely hampered the scalability of the system and underscored the need for a more robust and optimized approach.

To address these challenges, we have incorporated enhanced data structures for the inverted index, including Java implementations of the previous structures, as well as a novel structure involving one file per word. These changes, combined with the modular design enabled by Maven, ensure a streamlined, efficient, and professional codebase.

As a result, the indexing time is now independent of the number of books stored, a fundamental breakthrough for the scalability of the project. This achievement demonstrates the effectiveness of our approach and establishes a solid foundation for future developments in this domain.

The advancements presented in this paper not only overcome the limitations of the previous version but also provide a scalable and efficient solution for managing large-scale collections, contributing significantly to the field.

# 2 Problem Statement

The previous iteration of this project revealed several critical shortcomings that significantly hindered its usability and scalability. The most pressing issues included:

1. Performance Bottlenecks: The system exhibited a substantial decline in performance when the number of books increased even slightly. This inefficiency made the solution impractical for larger datasets, impeding its scalability and limiting its applicability to real-world scenarios.

2. Inefficient Indexing Process: The indexing time was directly influenced

by the number of books stored. This dependency created a linear or worse correlation between the dataset size and the processing time, further exacerbating the scalability problem.

3. Outdated Data Structures: While the previous implementation employed certain data structures for the inverted index, they proved insufficient to handle the growing demands of the system. The lack of optimized structures led to slower retrieval and storage processes.

4. Professional Limitations: The absence of a modular and professional framework resulted in code that was harder to maintain, extend, and integrate into larger systems, reducing the overall quality of the project.

These limitations underscored the need for a fundamental reengineering of the system to enhance performance, ensure scalability, and improve the professionalism of the solution. This paper addresses these challenges with a revamped approach, laying the groundwork for a robust and scalable system.

# 3 Solution

The code of this project can be found at our Git Hub Repository.

The Docker images for the modules of this project can be found at the following link: Docker Hub Repository.

To address the issues highlighted in the previous section, several key improvements were implemented, each tackling specific performance, scalability, and maintainability challenges:

1. Professionalization through Java: Migrating the project to Java significantly enhanced its overall professionalism. Java's robust features, such as strong typing, extensive libraries, and mature development tools, allowed for a cleaner, more maintainable, and scalable codebase. This transition facilitated better integration with other systems and ensured a more reliable foundation for future enhancements.

2. Modularity and Use of Interfaces: The project was redesigned to be more modular, with key components interacting through well-defined interfaces. This change greatly improved maintainability, as individual modules can now be updated or replaced without affecting the overall system. It also enabled easier testing, debugging, and future extensions, making the system more flexible and adaptive to new requirements.

3. Crawler with Threading for Improved Efficiency: The crawler was restructured to take advantage of multi-threading, which dramatically increased its efficiency. By processing multiple pages concurrently, the crawler could index data much faster, reducing the time required for data collection and making the system more responsive and capable of handling larger datasets.

4. Enhanced Inverted Indexing: Significant improvements were made to the inverted index, especially with the introduction of a new "file-per-word" approach. This innovation improved the speed and efficiency of indexing by storing each word in a separate file, which streamlined both storage and retrieval operations. This change drastically reduced the time spent during indexing and enhanced the system's overall performance.

5. Search Engine Optimization: The search engine was redesigned for much faster performance compared to its predecessor. Optimizations included more efficient query processing, refined data structures, and improved algorithms. Additionally, a new REST API was introduced to allow easier integration and interaction with other systems, making the search engine more accessible and versatile in various use cases.

These enhancements collectively addressed the main shortcomings of the previous system, resulting in a more efficient, scalable, and maintainable project that can handle larger datasets and provide faster, more reliable search results.

All experiments were conducted on a dedicated machine, with hardware specifications detailed in Table 1, ensuring consistency and accuracy across the benchmarking process.

| **System Information** | **Details** |
|---|---|
| **Operating System Name:** | Microsoft Windows 11 Pro |
| **Operating System Version:** | 10.0.22631 |
| **OS Build Type:** | Multiprocessor Free |
| **System Manufacturer:** | Acer |
| **System Model:** | Nitro AN515-58 |
| **System Type:** | x64-based PC |
| **Processor:** | 12th Gen Intel(R) Core(TM) i7-12700H, 2.70 GHz |
| **Total Physical Memory:** | 16.088 GB |
| **Virtual Memory: Maximum Size:** | 23.512 GB |
| **SSD Storage:** | 500 GB |

Table 1: System Specifications

## 3.1 Crawler

The crawler component has been implemented using two approaches: one with a serial implementation and another using threads. Benchmarks were conducted to compare their performance under different workloads.

### Results with Serial Crawler

The serial implementation was benchmarked using JMH for workloads of 5, 10, 15, and 20 books. The following table summarizes the results:

| **Books** | **Average Time (ms)** | **Error (ms)** |
|---|---|---|
| 5 | 11061.41 | 1657.51 |
| 10 | 17620.27 | 340.70 |
| 15 | 26395.35 | 613.45 |
| 20 | 35487.84 | 544.26 |

Table 2: Serial Crawler Performance Benchmarks with JMH

### Comparison with Threaded Crawler

Due to execution issues with JMH when testing the threaded crawler, direct execution time measurements were used for comparison. The results show the time (in seconds) required to fetch books for both implementations:

| **Books** | **Threaded Crawler (s)** | **Serial Crawler (s)** |
|---|---|---|
| 1 | 2.86 | 1.91 |
| 5 | 2.46 | 7.84 |
| 10 | 2.89 | 16.18 |
| 15 | 4.32 | 25.42 |
| 20 | 5.07 | 33.82 |
| 25 | 6.07 | 41.94 |

Table 3: Comparison of Execution Times for Threaded and Serial Crawlers

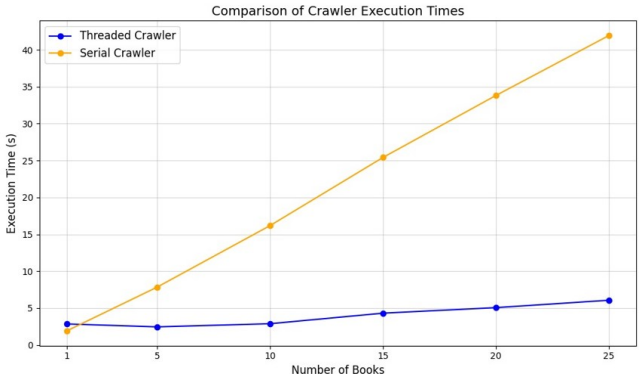


Figure 1: Execution time comparison between the threaded and serial crawler implementations.

A graph displaying the results of the time measurements is attached above. The image shows a comparison of the execution times for both the threaded and serial crawler implementations across different numbers of books.

**Discussion**

The threaded implementation shows a significant performance improvement as the number of books increases, especially for workloads of 5 or more books. While the serial implementation scales linearly with the number of books, the threaded implementation maintains a nearly constant time for higher workloads due to its ability to fetch multiple books in parallel. These results highlight the scalability benefits of threading for I/O-bound tasks.

## 3.2 Inverted Index

### 3.2.1 Hashed Index

This method distributes words into $n$ buckets using a hash function, with threads speeding up word storage. Each thread processes one bucket, preventing concurrency issues. We chose 8 buckets, as this is optimal for the number of logical CPU cores. Each bucket stores a dictionary with words as keys and a ResponseList (containing document IDs and word positions) as values, in binary format for faster serialization. Although this approach improves speed over a single-file index, the indexing time is influenced by the number of indexed books, as all relevant buckets must be loaded and updated during each indexing process.

To use this method, instantiate the class with:

- Paths to the book and datamart directories (relative paths recommended).

- A list of indexed books.
- An instance of the tokenizer.
- The number of buckets.
- Instances of DatamartReader and DatamartWriter (for binary files).

The logic is as follows:

- The only accessible method is `indexAll()`.
- `indexAll()` lists all books and checks whether each is indexed.
- If not, the book is cleaned (stopwords and unnecessary characters removed) and tokenized.
- The method `distributeWorkload()` distributes words across buckets using `hashCode()` and prepares the data for multi-threaded storage.
- `updateDatamart()` starts the threads, each processing one bucket. Each worker loads the existing index, merges the new data, and overwrites the bucket file.
- If the indexing is successful, the book ID is stored to prevent re-indexing.

### 3.2.2 Trie Node

The primary changes to stage 1 involve switching from Python to Java, modifying the code, changing the data format from JSON to MessagePack, and partitioning the trie into prefix-based files.

The trie structure now uses MessagePack for efficient node serialization and supports partitioning by prefixes, improving search speed by loading fewer nodes into memory.

**What is MessagePack?** MessagePack is a binary format similar to JSON but faster and more compact. It allows efficient data exchange and smaller file sizes, encoding small integers in a single byte and short strings with minimal overhead.

**MessagePack Format for Storing Trie Data** Previously, trie data was stored as JSON. Now, MessagePack is used for serialization, reducing memory usage and enhancing performance, especially with large datasets like book collections.

**Trie Divided into Prefix-Based Files** In stage 2, the trie is split into multiple files based on prefixes. This differs from the previous method where the entire trie was stored in one file.

This partitioning optimizes memory and disk usage, as only relevant parts of the trie are loaded for searches. It also supports parallelization, allowing multiple processes to read different trie parts.

These changes improve:

- **Efficiency:** Only relevant parts of the trie are loaded, reducing memory consumption and speeding up searches.
- **Faster Access:** Multiple prefix files speed up search times by loading smaller portions of the trie.
- **Optimized Disk Usage:** Smaller, manageable prefix files reduce overall disk space usage.

**Implementation of Two Methods for Trie Indexing** The `indexBooks` method now serializes and stores parts of the trie related to specific prefixes in separate MessagePack files instead of one large file. It also only saves the book ID to `indexed_books.txt`, improving efficiency.

The second method indexes one book at a time, checking which books are already indexed and processing the next unindexed one. If all books are indexed, it prints a message.

**Indexing Check with `indexedBooksFile`** Before indexing, both `indexBooks` and `index` methods check the `indexedBooksFile` (now `indexed_books.txt`) to avoid re-indexing, improving efficiency.

**Memory and Disk Efficiency:** MessagePack reduces memory usage, especially for large datasets, and the prefix-based files prevent unnecessary memory consumption during searches.

**Improved Search Speed:** With prefix partitioning, the system loads only the relevant nodes, speeding up searches. The prefix files also allow easier parallelization for better performance in multi-core environments.

**Faster Data Serialization:** MessagePack's faster serialization and deserialization reduce disk I/O time, important for large datasets.

**Scalability:** The prefix-based files improve scalability by allowing easy addition of new prefixes without affecting the rest of the trie, making the system more manageable as the dataset grows.

**Reindexing Efficiency:** The system checks if a book has already been indexed before reprocessing, preventing unnecessary re-indexing.

### 3.2.3 Test Results

For testing both the different implementations of the inverted index, we designed the following scenario: we will measure the time it takes the indexer to index 10 books when there are no books already indexed, how much it itakes to index 10 books when there are already 10 books indexed... and so on, until we reach 40 books indexed, where we will stop because doing the tests with more books would take too long.

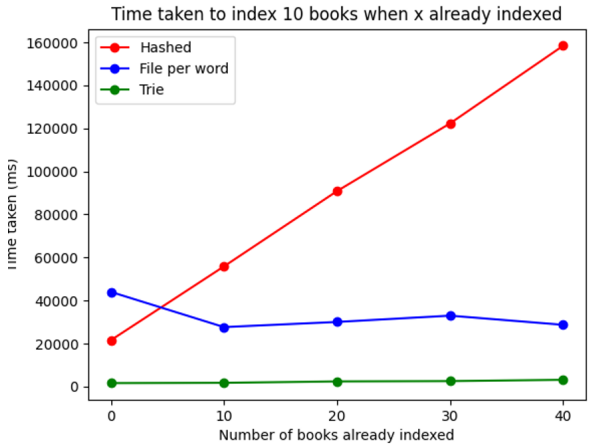


Figure 2: Time it takes each implementation to index 10 books where there are already "x" books already indexed

Figure 2 reveals that the hashed implementation is faster when therea are

no books indexed, but when the number of books indexed increases, the one file per word implementation becomes faster. The time it takes the hashed implementation to index the books follows a linear ascending pattern, suggesting that the time it takes to index a book is proportional to the number of books already indexed. However, the time of execution for the one file per word implementation is constant, suggesting that the time it takes to index a book is not dependent of the books already indexed.

The results are consistent with the theoretical complexity of the two implementations. The hashed implementation has a complexity of O(n) for indexing a book, where n is the number of books already indexed. The one file per word implementation has a complexity of O(1) for indexing a book, regardless of the number of books already indexed.

The reason for this to happen is that the hashed implementation has to load the entire bucket of the word to be indexed, update the index, and then save it back to the disk. It is obvious that the time it takes to do this will increase with the number of books already indexed. On the other hand, the one file per word implementation only has to open the file of the word to be indexed, append the number of file and the appearance of the word in the book, without even having to read the file (only create a new file if it does not exist). This is why the time it takes to index a book is constant, regardless of the number of books already indexed. The search of the file itself is also constant, because the file name is the word itself, so it is easy to find the file of the word to be indexed.

In addition, we can appreciate that the time it takes to the trie implementation, despite being linearly dependent of the books already indexed, is still way faster than the hashed implementation and the one file per word implementation. This is because the trie implementation is capable of indexing a book in a fraction of the time it takes the other implementations. This is because the trie implementation does not have to load the entire index to update it, but only the part of the index that corresponds to the word to be indexed. This is why the time it takes to index a book is linearly dependent of the number of books already indexed, but still faster than the other implementations. In other words, the Trie is optimized for indexing books, while the other implementations are not.

However, we will present the results of the benchmarking of the query engines, which will show that the trie implementation is not the best for searching books but the one file per word implementation is. This is because the trie implementation has to load the entire index to search for a word, while the one file per word implementation only has to open the file of the word to be searched. This is why the time it takes to search a word is linearly dependent of the number of books indexed for the trie implementation, but constant for the one file per word implementation. In other words, the trie is optimized

for indexing books, while the one file per word implementation is optimized for searching words.

There is therefore a trade-off between the trie implementation and the one file per word implementation. But since the next stage is focused on massive querying, we will use the one file per word implementation for the final implementation of the inverted index unless we find out a way to make the trie implementation faster for searching words.

#### 3.2.4 One file per word index

A faster alternative stores each word in a separate file. Indexing now depends only on the book size, not on the number of indexed books, and the method doesn't require reading the entire file—only appending new word occurrences. Each file contains a word and its appearances in the format "bookID:position1, position2,...". This approach is faster because it eliminates the need to load and update all buckets.

This method is implemented in the `FilePerWordInvertedIndex` class, which is instantiated with similar parameters as the previous method, minus the need for threads. The `indexAll()` method works as follows:

- Lists all books and checks indexing status.
- If unindexed, the book is cleaned and tokenized.
- `updateDatamart()` appends word appearances to the corresponding word files.
- Each file stores word appearances in the format "bookID:positions".
- After indexing, the book ID is recorded to prevent re-indexing.

This method is simpler, faster, and results in a larger number of smaller files, but each file remains small ( 1KB). It's recommended for faster and easier implementation compared to the more complex and slower Hashed Index method, which involves threads, hash functions, and data retrieval from multiple buckets.

The Hashed Index was the first method implemented due to its complexity, but its multi-threading and data management helped speed up the implementation of the second method. The second method was completed in a few days and is faster and simpler, so it is the recommended choice.

## 3.3 Search Engine Results and Analysis

The following table summarizes the results of the search engine performance using three different data structures: Hashed Index, Trie, and File Per Word. The tests were conducted for both one-word and two-word search scenarios, with and without filtering.

| Method | Hashed Index (ms) | Trie (ms) | File Per Word (ms) |
|---|---|---|---|
| One Word Search | 1938.594 | 1021.161 | 0.174 |
| Two Word Search | 6840.687 | 4067.887 | 0.270 |
| One Word Search (Filtered) | 2094.830 | 1323.391 | 0.243 |
| Two Word Search (Filtered) | 8821.508 | 4702.541 | 0.278 |

Table 4: Search Engine Performance Comparison for Different Data Structures

### 3.3.1 Results Analysis

The results show a significant difference in the performance of the three data structures across different search methods.

- Hashed Index: The Hashed Index performs significantly worse than other solutions. Searching for one word takes it around 2 seconds, but when for two words it increases more than twice to almost 7 seconds. When also filtering the results, the average time is even worse.

- Trie: The Trie is more efficient than the Hashed Index. It is almost tqice as fast. Still the results are not satisfying with 1 second for one word and 4 seconds for two words without filtering. This is likely due to the complexity of navigating the trie structure, which can grow large and require more time for searches as the number of words increases.

- File Per Word: The File Per Word structure significantly outperforms both the Hashed Index and Trie in all search scenarios. This is because the File Per Word structure has minimal processing overhead for filtering, as it processes each word independently, making it suitable for searches with filters.

### 3.3.2 Best Data Structure

Based on the observed performance, the File-Per-Word Index is the best data structure for general use in terms of speed. It provides a good balance between efficiency and simplicity for search scenarios. Both without and with filtering

according to metadata it performs very well. This is because of its lower overhead. The Trie is more specialized and may be useful for cases where prefix searches or lexicographical order are required, but it is not as efficient as File Per Word for searches. Although when creating indexing the books it is not as good as Trie index, still its average time per 10 books asymptotically is constant. Hashed Index performs only better than File-Per-Word Index when there are no books indexed, but with increasing number of books already indexed, its average time to index the next ones surges up.
In conclusion, the File-Per-Word Index is the best overall choice for a search engine.

# 4 Conclusion

In this work, we implemented and compared different inverted index strategies, focusing on three primary methods: the threaded hashed index, the trie-based index, and the one file per word index. Through rigorous testing and performance evaluations, we analyzed the strengths and weaknesses of each approach, ultimately concluding that the one file per word method offers the best combination of speed, simplicity, and scalability for our use case.

The threaded hashed index showed promising results in initial tests, particularly for smaller datasets. However, as the number of indexed books increased, its performance started to degrade due to the linear complexity of the indexing process. This behavior was expected, as the hashed implementation requires loading and updating entire buckets for each word, which becomes time-consuming as the dataset grows. On the other hand, the one file per word method demonstrated a constant indexing time regardless of the number of books already indexed, making it highly efficient for scaling to larger datasets.

The trie-based index, while efficient for indexing books, proved less optimal for querying. Despite its faster indexing performance compared to the hashed implementation, it required loading the entire index during search operations, making search times scale with the number of indexed books. In contrast, the one file per word approach optimized search times by keeping each word's index in a separate file, making searches constant in time. Therefore, we determined that the one file per word index is best suited for the final implementation of the inverted index, especially given the emphasis on query performance in the next phase.

Looking ahead, the hashed index method will be abandoned in favor of the one file per word method, which will be further optimized for larger datasets. The focus will be on improving the efficiency of the file storage format, possibly switching to binary files, and leveraging multi-threading for faster saving tasks.

As we move towards indexing 2000 books, the one file per word index method will be essential for handling such a large volume of data within a reasonable timeframe. Additionally, the performance will be fine-tuned to ensure the system can scale effectively to handle an increasing number of queries from users.

Future work will also include the potential development of a user interface (UI) to provide a more accessible and intuitive way for users to interact with the system. By implementing these optimizations and ensuring the system can handle higher workloads, we aim to deliver a fast, efficient, and scalable solution for indexing and querying large datasets of books.

## 4.1 Future Work

On the next stage, the hashed implementation will be abandoned due to it low performance, and the one file per word implementation will be kept for the final implementation of the inverted index. In addition, we will try to make the one file per word implementation even faster by using a more efficient way to store the index in the file, such as using a binary file instead of a text file and distributing the saving task among multiple threads to make it faster. Since for

the next stage 2000 books will be indexed, we will not be able to use the hashed implementation, as it would take too long to index all the books. So, this is the best time to abandon the hashed implementation and focus on the one file per word implementation and test if the trie implementation is capable of indexing 2000 books in a reasonable time. Now, the time it takes to index 10 books is about 30 seconds. If the time it takes to index 2000 books is proportional to the number of books indexed, it would take about 1 hour and 40 minutes to index all the books. However, we expect this time to be reduced by using a more efficient way to store the index in the file and distributing the saving task among multiple threads.

Also, we aim to further optimize the existing methods to support a much higher workload. The goal is to ensure that the system can handle and respond to numerous requests from users through the API efficiently. This will involve fine-tuning the performance of the current algorithms to scale better under heavy usage.

Additionally, there is a possibility that a user interface (UI) will be developed to facilitate interaction with the system. The UI would provide a more user-friendly experience, allowing users to easily query the system and view results in a more accessible manner.