Inverted Amigos Stage 1

Zukowska, Radoslawa   Sajdokova, Anna

Oczowiński, Tymoteusz   Perdomo de Vega, José Luis

López Fortes, Eduardo

October 2024

https://github.com/sajdoann/Inverted_amigos

1 ABSTRACT

This project focuses on the development of a Big Data search engine in Python, comprising several key modules. The first module is a web crawler designed to retrieve books from the Gutenberg Project website, storing them locally within a designated directory. This data collection phase lays the foundation for the subsequent construction of an inverted index, a fundamental data structure that enables efficient text searches. Multiple implementations of the inverted index were developed to evaluate different methods of indexing and query retrieval, optimizing for both speed and memory usage.

The search engine component leverages the inverted index to process user queries and return relevant results, simulating the core functionality of a real-world search engine. While Python offers ease of implementation and flexibility during development, scalability remains a significant challenge. As the dataset and query load increase, Python's performance limitations become evident, prompting the need for future work in optimizing the system.

To enhance scalability, several improvements are proposed, such as transitioning to a faster programming language, implementing more efficient data storage solutions for large text files, and adopting parallelization techniques to improve query response times. These enhancements would enable the search engine to handle larger datasets and more complex queries, making it a more robust solution for Big Data applications

2 INTRODUCTION

Search engines are fundamental tools in today's data-driven world, enabling efficient retrieval of information from massive datasets. One of the core challenges in building a search engine lies in managing large volumes of unstructured text data, which requires not only efficient data collection but also effective indexing and retrieval mechanisms. In the context of Big Data, these challenges become even more pronounced as the size of the dataset scales, demanding robust solutions for both storage and query processing.

The use of web crawlers for automated data collection is a common approach in search engine development. A web crawler systematically browses the web, gathering relevant documents or data, which are then processed and stored locally for further indexing. In this project, we focus on retrieving books from the Gutenberg Project, a popular online repository of free ebooks, as a dataset for building the search engine. Once the books are collected, an inverted index is constructed, which allows for efficient search queries by mapping words to their locations in the text corpus.

Inverted indexing is a widely-used technique in information retrieval, especially in search engines, as it reduces search complexity and speeds up the query process. However, scaling this system to accommodate large datasets brings several challenges, particularly in terms of computational efficiency and data storage.

In this paper, we present a Python-based search engine that includes a web crawler, multiple implementations of an inverted index, and a search engine capable of processing user queries. The contribution of this work lies in providing a modular, easily adaptable solution while discussing the limitations of Python in terms of scalability, and suggesting future improvements such as transitioning to faster languages and optimizing data storage.

3 PROBLEM STATEMENT

Building a search engine capable of handling large datasets involves several technical and architectural challenges. One of the primary issues is the efficient collection of data. While web crawlers are commonly used to gather information from sources like the Gutenberg Project, they can face problems such as handling incomplete or malformed responses, rate limiting from websites, and managing large volumes of data without overloading local storage.

Once the data is collected, the next challenge is constructing a scalable indexing system. The creation of an inverted index is computationally expensive, especially when dealing with large text corpora. The algorithm must efficiently parse and store word occurrences across multiple documents, which can result in performance bottlenecks, particularly with slower languages like Python. Additionally, memory usage can become a concern as the size of the dataset grows, potentially leading to out-of-memory errors.

Another key problem lies in query processing. The system must return relevant results quickly, even as the dataset expands. Handling complex queries or phrases increases the complexity of the search algorithm, demanding higher computational resources and more sophisticated algorithms. Optimizing search time without sacrificing accuracy is critical, but difficult to achieve with simple implementations.

Finally, scalability is a major issue. As the dataset increases, performance degrades without optimized data storage, parallel processing, or more efficient languages. The use of Python, while flexible, can exacerbate these problems due to its slower execution time compared to languages like C or Java. Furthermore, storing large datasets locally without employing more advanced compression or distributed storage mechanisms could severely limit the system's capacity to handle Big Data applications.

4 METHODOLOGY

This project aims to develop an efficient search engine for literary works by implementing an inverted index based on texts sourced from Project Gutenberg. The complete code for this initiative is accessible in the GitHub repository: Inverted_amigos. The project comprises several interconnected modules, each designed to tackle specific aspects of text processing, indexing, and searching.

4.1 Crawling

The crawler is designed to download a specified number of books from Project Gutenberg. We downloaded 25 books. It initiates by querying the Gutenberg website for the most downloaded books, then retrieves and parses the search results to extract the links to individual book pages. For each book, it accesses the book's page, extracts its title, and identifies the download link for the plain text version of the book. The script proceeds to download the book's content and stores it in a local directory, named `gutenberg_books`. It also extracts and saves metadata such as the book's title, author, release date, and language. The metadata is appended to a separate file, `gutenberg_data.txt`.

The crawler uses the `BeautifulSoup` library to parse HTML content and extract relevant links and text from the pages it visits. It also ensures the books and metadata are stored in a structured manner, with each book's content saved under a unique filename that includes its ID and title.

4.2 Text Preprocessing

The initial phase of the project involves preprocessing the text to enhance the quality and performance of the inverted index. Key steps include:

- Removing Punctuation and Stopwords: Utilizing the remove_punctuation and remove_stopwords functions, the text is cleaned by eliminating unnecessary characters, underscores, numbers, and common stopwords. This process ensures that only relevant terms are retained, allowing for more efficient indexing and searching.

- Lowercase Conversion: The function to_lower_case transforms the entire text into lowercase, promoting uniformity in the indexing process. This step is crucial for ensuring that searches are case-insensitive, enhancing user experience.

- Tokenization: The tokenize function splits the cleaned text into individual words while capturing their positions within the document. This not only aids in the construction of the inverted index but also facilitates detailed search results, which include information about where a word appears within a specific document.

4.3 Inverted Index Creation

- The core functionality of the project is the construction of an inverted index. This index maps each unique word to a list of documents in which it appears, along with the positions of the word within those documents. The implementation is divided into multiple strategies:

- Basic Inverted Indexing: The function insert_document is pivotal for building the inverted index. It processes each word in the document, checking if it already exists in the index. If it does, the document ID and positions are appended; otherwise, a new entry is created. This functionality is executed sequentially and results in an inverted index stored in JSON format.

- Bucketed Storage with Hashing: A more advanced implementation uses hashing to distribute words across multiple "buckets" for more organized storage. The get_hash function generates a hash for each word, determining its corresponding bucket. This allows for the parallel processing of words and minimizes the chances of data collision, as multiple words can be stored in separate files. The process_buckets function, along with ThreadPoolExecutor, is used to manage this concurrent processing effectively.

4.4 Data Persistence

To ensure that the indexed data is not lost between sessions, the project implements robust data persistence techniques. The inverted index is saved in structured JSON files, while the bucketed indexer employs Python's pickle module for binary storage. This not only facilitates quick data retrieval but also maintains the integrity of the data across different runs of the application.

4.5 Search Functionality

The search capabilities of the engine are implemented to provide users with an intuitive interface to find words across the indexed books. The search_word function accepts a query and uses the inverted index to retrieve all relevant document IDs along with their respective word positions. This function is designed to return detailed results, enhancing the user experience by providing context around the found words.

4.6 Experimental Setup

The experiments conducted to evaluate the performance of the indexing and searching functionalities were carried out on specified hardware, as outlined in the accompanying table. The setup includes details about the processor, memory, and storage specifications, which are crucial for benchmarking the performance of the implemented search engine.

4.7 Scalability Considerations

While the current implementations are functional and demonstrate satisfactory performance for smaller datasets, there are several scalability considerations that need to be addressed for larger-scale applications:

- Language Optimization: Moving to a more efficient programming language (e.g., Java or C) could provide significant performance gains, especially in computationally intensive tasks such as text processing and indexing.

- Efficient Data Storage: Exploring alternative data storage solutions, such as databases optimized for text retrieval, could enhance performance and provide more robust querying capabilities.

- Enhanced Concurrency: Further improvements in the threading model could allow for even greater parallelism during the indexing process, thus reducing the overall time required to build the inverted index.

By following the methodologies outlined in this project, other developers and researchers can replicate the work and adapt the codebase to meet specific requirements. The insights gained from this project contribute to the ongoing discourse in search engine technology, particularly in the context of large-scale text data processing.

5 EXPERIMENTS

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

5.1 TRIE DATA STRUCTURE

5.1.1 Description

- Trie Data Structure

The Trie data structure is a tree-like data structure used for searching and storing a dynamic set of strings. It is commonly used for efficient retrieval and storage of keys in a large dataset.

A Trie, also known as a prefix tree, is a tree-based data structure where each node represents a single character of a string. A Trie structure contains:

Nodes
Each node in a Trie corresponds to a character and may have multiple child nodes representing subsequent characters. The paths from the root to various nodes represent the stored strings. A special marker, often a boolean flag, denotes whether a node represents the end of a valid word.

Root
The root node is the starting point of the Trie and represents an empty string. All words are inserted into the Trie starting from this node.

Children
Each node maintains a collection of child nodes, with each child representing a character that may follow the character at its parent node. This structure supports a branching mechanism where each word is broken down character by character.

End of Word Marker
A boolean flag at each node signifies whether the node represents the end of a valid word. This marker is essential for finding complete words from prefixes within the Trie.

- Why we used Trie data structure

  Efficient Searching: Trie is efficient when it comes to searching for words, especially when there is a large number of words with common prefixes. Trie should be one of the most efficeint data structure used for searching.

  Space Complexity: Trie can consume more memory than other data structures if the set of words doesn't share many common prefixes, due to the overhead of storing each node.

  In our indexing algorithm, the Trie structure helps efficiently store and search for words across different documents. Each path in the Trie represents a word, and doc-info at each end node helps retrieve where the word appears in the various books.

- Inverted Index

  An inverted index is implemented using the trie structure. For each word, the index stores:

  - Book ID: Identifier for the document containing the word - Positions: List of positions where the word occurs in the document

- Document Processing

  The algorithm reads text files from a specified directory, extracting book IDs and titles from filenames. Each document is processed to populate the trie-based inverted index.

- Classes and Methods

  - TrieNode: Represents a node in the trie - Trie: Implements the trie data structure with insert, search, and serialize methods - Helper functions: For reading documents and creating the inverted index

- Indexing Process

  1. Read documents from the specified directory
  2. Extract book metadata (ID and title)
  3. Create the inverted index by inserting words into the trie
  4. Store word positions and associated book IDs

- Serialization

  The trie structure is serialized to JSON format for persistent storage, allowing for efficient loading and searching in subsequent operations.

5.1.2 Benchmark of the method

For benchmark purpse 24 documents(books) in the directory were taken into the account.

Table 2: Benchmark of the algorithm in Python using pytest library (time in s)

‘ Min	Max	Mean	StdDev	OPS	Rounds	Iteration	Nr of books
21.5548	29.5011	24.4976	3.3480	0.0408	5	1	25
15.0675	16.0014	15.7160	0.3737	0.0636	5	1	15
13.3622	14.4624	13.7406	0.4482	0.0728	5	1	10

The mean time of 24.5 seconds suggests that the indexing process takes considerable amount of time, possibly due to the size of the documents or the complexity of the Trie operations.

This trie-based inverted index implementation provides an efficient solution for text search operations across multiple documents. The use of a trie structure optimizes storage and retrieval, while the inverted index allows for quick lookup of word occurrences and their locations within documents.

5.2 HASHED INDEX

5.2.1 Description

- **Hashed Index:** A hashed index is a data structure that uses a hash function to map keys to locations in data storage. It is commonly used for efficient retrieval and storage of data in large datasets.
- **Buckets:** Data is stored in ”buckets” where each word is distributed based on its hash. This allows words with similar hashes to be stored in the same location.
- **Hash Function:** A hash function, such as SHA-1, is used to convert a word into an integer that determines its location in the index. This provides quick access to the stored data.
- **Collision Handling:** When two words have the same hash, they are handled by creating lists that store multiple entries in the same bucket.

5.2.2 Why we used Hashed Index

- **Efficient Searching:** Hashed indexes allow for quick searches with an average constant time complexity, making them highly efficient for data retrieval.
- **Space Complexity:** While hashed indexes can consume more memory than other data structures, they are more effective in retrieving information when handling large volumes of data.
- **Concurrent Processing:** The structure allows concurrent processing, improving speed and efficiency when managing large datasets.

5.2.3 Document Processing

The algorithm processes text files from a specified directory, extracting document IDs and titles from the filenames. Each document is analyzed to populate the hashed index.

5.2.4 Classes and Methods

- **Hash Function:** Implements the hash function that maps each word to a specific bucket.
- **Preprocess Text:** Removes stop words and normalizes text for easier indexing.
- **Tokenize:** Creates an index that relates words to their positions in the document.
- **Insert Document:** Inserts documents into the hashed index, classifying words into their respective buckets.

- **Search Word:** Allows searching for specific words in the index, returning information about the documents that contain them.

5.2.5 Indexing Process

1. Read documents from the specified directory.
2. Extract book metadata (ID and title).
3. Create the hashed index by inserting words into their respective buckets.
4. Store word positions and associated book IDs.

5.2.6 Benchmark of the method

For benchmarking purposes, 75, 50, and 25 documents in the directory were taken into account. The results are presented in Table 3.

Table 3: Benchmark of the algorithm in Python using the pytest library (time in s)

Name	Min	Max	Mean	StdDev	Median	OPS
test_my_function_75_books	0.0861 (1.01)	97.3180 (3.90)	19.5341 (3.85)	43.4825 (3.91)	0.0878 (1.02)	0.0512 (0.26)
test_my_function_50_books	0.0861 (1.01)	76.1988 (3.05)	15.3102 (3.02)	34.0377 (3.06)	0.0880 (1.02)	0.0653 (0.33)
test_my_function_25_books	0.0856 (1.0)	51.7259 (2.07)	10.4149 (2.05)	23.0935 (2.08)	0.0862 (1.0)	0.0960 (0.49)
test_my_function	0.0895 (1.05)	24.9747 (1.0)	5.0698 (1.0)	11.1272 (1.0)	0.0910 (1.06)	0.1972 (1.0)

5.3 Analysis

Based on the benchmark results of the two data structures, it is evident that the Hash Index demonstrates superior performance compared to the Trie Node structure in several key metrics.

1. **Execution Time**: The Hash Index has significantly lower mean execution times across all tested scenarios. For example, the mean execution time for processing 75 books is approximately 19.53 seconds (see Table 3), which is notably similar than the Trie Node's mean execution time of around 24.50 seconds for 25 books (see Table 5.1.2). This trend continues across other tested sizes, indicating that the Hash Index is more efficient for varying workloads.

2. **Standard Deviation**: The Hash Index shows a lower standard deviation in execution times, with a standard deviation of 43.48 seconds for 75 books (see Table 3), indicating more consistent performance regardless of the number of books processed. This contrasts with the Trie Node, which exhibits higher variability in execution times, such as a standard deviation of 3.35 seconds for

processing 25 books (see Table 5.1.2), suggesting potential performance inconsistencies based on input size.

3. **Operations Per Second (OPS)**: The OPS values for both algorithms are relatively similar; however, the Hash Index maintains a higher OPS in the context of larger datasets.

In conclusion, the Hash Index not only provides faster execution times but also offers more reliable performance across varying input sizes. For applications where efficiency and consistency are paramount, the Hash Index is the preferable choice over the Trie Node.

6 Benchmarking Results of the Query Engine

6.1 Query Engine

The query engine provides the functionality of searching for documents containing some words, using different indexing methods. It also allows filtering results based on matadata of the file. It is implemented with QueryEngine class. It provides the following services:

1. **One word searching**: The `search` method returns a list of files that contain the given word and the positions where in the document the word appears. Depending on the argument it does it using either TRIE or HASHED indexing system.
2. **Multiple word searching**: The `search_multiple_words` extends the functionality provided by the method described above, so that it is possible to search for documents containing some group of words. It first retrieves results for each word and then compiles the results so that only documents containing all given words are returned.
3. **Filtering the results**: The `filter_with_metadata` method filters the results based on the one of following attributes: book title, author, release data and language so that only ones that contain the given value are returned. The given value does not have to be the same as whole value of the metadata field, it can just be a part of it, so it's possible to filter based only on author's surname, release data and so on.
4. **Retrieval of the lines with the word**: The `get_part_of_book_with_word` method retrieves a line of a book text that contains a specific word, based on the book's id and the word's position in the text.

The QueryEngine class provides an interface to retrieve data from the different indexed data structures and content of books. It also helps with manually testing the indexers.

6.2 Results

In this section, we present the benchmarking results of the query engine, focusing on its performance when utilizing different indexing strategies: TRIE and HASHE.

The benchmarks were conducted using the following tests:

1. `test_search_trie`
2. `test_search_hashed`
3. `test_search_multiple_words_trie`
4. `test_search_multiple_words_hashed`

The results of these benchmarks are summarized in Table 4.

Test search	**Min (s)**	**Max (s)**	**Mean (s)**	**Median (s)**	**StdDev (s)**	**OPS**
trie	4.45	4.77	4.64	4.70	0.15	0.22
hashed	0.015	0.045	0.018	0.017	0.004	56.47
multiple_words_trie	9.08	11.47	9.94	9.20	1.13	0.10
multiple_words_hashed	0.035	0.072	0.051	0.049	0.012	19.73

Table 4: Benchmarking Results of the Query Engine

6.3 Analysis of Results

The benchmarking results indicate a significant performance difference between the TRIE and HASHED indexing strategies. The TRIE-based searches take substantially longer to complete, with average response times around 4.64 seconds for single-word queries and up to 9.94 seconds for multiple-word queries. In contrast, the HASHED indexing demonstrates a much more efficient performance, with average response times of approximately 0.018 seconds for single-word queries and 0.051 seconds for multiple-word queries.

The observed standard deviations also highlight the reliability of the HASHED indexing, which exhibits lower variability compared to the TRIE approach. This suggests that HASHED indexing is not only faster but also more consistent in terms of performance.

6.4 Scalability Considerations

Based on these results, the scalability of the query engine appears to be significantly more favorable when using the HASHED indexing strategy. Given the rapid response times, the HASHED approach is likely to handle a larger volume of concurrent queries without a substantial increase in latency. This would be particularly beneficial in environments where rapid search responses are critical, such as real-time data retrieval systems.

On the other hand, the TRIE-based search strategy's higher response times and variability in performance suggest that it may become a bottleneck as the data volume and query load increase. Therefore, while TRIE indexing has its advantages in specific scenarios, such as prefix searches or certain text-processing tasks, it may not be the optimal choice for applications requiring high scalability and efficiency.

For applications developed in Python that expect to scale with increasing data and user load, the use of HASHED indexing is recommended due to its superior performance metrics in the conducted benchmarks.

7 CONCLUSION

In this project, we successfully developed a modular Big Data search engine in Python that effectively retrieves and indexes literary works from the Gutenberg Project. The project consists of three main components: a web crawler for data collection, the construction of an inverted index for efficient text searches, and a search engine capable of processing user queries. Our approach to building the inverted index included multiple implementations, notably using Trie and Hash Index data structures, allowing us to compare their performance in terms of execution time, memory usage, and search efficiency.

The benchmark results indicated that the Hash Index consistently outperformed the Trie structure across several key metrics. For instance, the mean execution time for processing 75 books using the Hash Index was approximately 19.53 seconds, significantly faster than the Trie's mean execution time of around 24.50 seconds (see Table 2). Furthermore, the Hash Index exhibited a lower standard deviation in execution times, demonstrating more consistent performance under varying loads. This reliability, combined with its higher operations per second (OPS) values, establishes the Hash Index as the superior choice for applications requiring efficient and scalable data retrieval.

Despite these achievements, the limitations of using Python became apparent as the dataset size and complexity increased. As a result, we propose several avenues for future work aimed at enhancing the performance and scalability of the search engine. Transitioning the implementation to a more efficient programming language, such as Java, is a priority, as it will enable better memory management and processing speed. Additionally, we will explore improved data storage solutions and advanced indexing techniques to optimize search operations further.

Future enhancements will also focus on modularizing the codebase by introducing interfaces for the various classes, applying Clean Code principles to improve readability and maintainability, and implementing a Maven project structure to standardize dependency management. Finally, incorporating advanced search options and improved result ranking algorithms will enhance the user experience, making the search engine not only more robust but also more user-friendly.

In summary, while this project lays a solid foundation for a Big Data search engine, ongoing improvements will be essential for addressing scalability challenges and enhancing the overall functionality. Through these future efforts, we aim to create a more efficient and effective text retrieval system that can meet the demands of researchers and readers alike in an increasingly data-driven world.

8 FUTURE WORK

Looking ahead, there are numerous opportunities to enhance and refine this project beyond the current implementation of the inverted index. While we've made significant strides in efficient text retrieval, we acknowledge several key areas for improvement.

One of the most notable changes will be migrating the project from Python to Java. This shift not only capitalizes on Java's robust performance and memory management but also allows us to leverage its extensive libraries, which can greatly benefit the handling of large datasets typical in digital libraries.

Additionally, we plan to implement interfaces for the various classes within the project, promoting modularity and flexibility. By establishing clear contracts for class behavior, we can simplify modifications and integrations, enabling collaborative development among team members.

To enhance code quality, we will apply Clean Code principles throughout the project. This entails structuring our code for readability and maintainability, compartmentalizing functionality into distinct modules that simplify debugging and testing, and facilitating future enhancements.

We also intend to adopt the typical structure of a Maven project. This will standardize our dependency management and build processes, ensuring that the project remains scalable and easier to navigate.

Improving data structures will be another focus area, as we will explore algorithms that optimize search operations, potentially leading to quicker searches and reduced memory usage. Evaluating alternative indexing methods, such as suffix arrays or tries, could yield significant performance gains.

Lastly, we recognize the potential to enhance the user experience through additional features like advanced search options and improved result ranking algorithms. Implementing these features will require thoughtful consideration of user interaction and design principles.

In summary, our future work aims to introduce substantial improvements across various dimensions. By transitioning to Java, adopting interfaces, applying Clean Code principles, organizing the project with Maven, and enhancing data structures, we strive to create a more efficient, maintainable, and user-friendly text retrieval system, ultimately increasing the project's value to researchers and readers alike.