Search Engine

Álvaro Rodríguez González, Lucía Afonso Medina
Alejandro Alemán Alemán, Farid Sánchez Belmadi
Néstor Ortega Pérez, Dalia Valeria Barone

October 2024

Contents

Abstract

In the era of Big Data, efficient information retrieval from large text datasets is paramount. This project focuses on the development of a search engine, with particular emphasis on building a scalable crawler, indexer, and query engine. The crawler is designed to download and preprocess books from Project Gutenberg, while the indexer creates an inverted index to enable fast keyword searches. Two different data structures are tested for the inverted index, with performance benchmarks conducted to evaluate their scalability and efficiency. A minimal query engine is implemented to facilitate searches, and the results are analysed to determine the optimal approach. This paper details the development process, data structures tested, and the overall system performance, providing a foundation for future improvements in search engine scalability.

Introduction

With the exponential growth of data available in digital form, the need for efficient search engines has become more critical than ever. The ability to search vast amounts of text data quickly and accurately is essential for both academic research and practical applications. Search engines are a core component in numerous fields, from web searching to large-scale data analysis. The first stage of this project aims to design and implement the fundamental components of a search engine: a crawler, indexer, and query engine.

In this stage, we developed a Python-based system that downloads books from Project Gutenberg, preprocesses them, and builds an inverted index to allow fast keyword searches. The challenge of handling large datasets led to the exploration of different data structures for the inverted index, with a focus on their performance and scalability. In particular, we tested and benchmarked two different structures to identify the most efficient solution for large-scale data retrieval.

This paper outlines the design, implementation, and evaluation of the search engine components, providing insights into the scalability and performance of different data structures in the context of information retrieval. The results from this stage lay the groundwork for future improvements and expansion of the system.

1 Modules and data structures

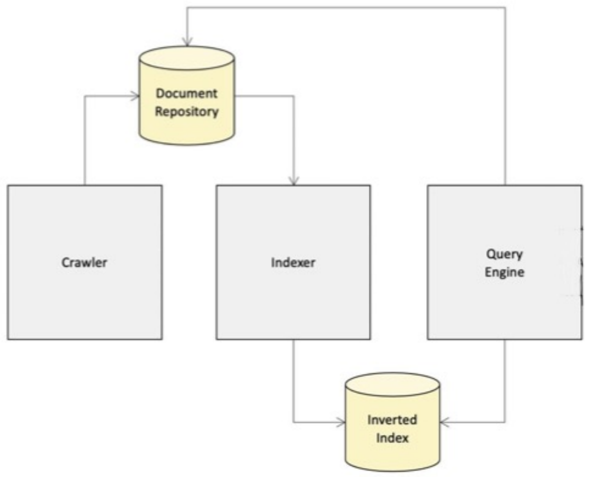

Figure 1: Modules

1.1 Crawler Module

The purpose of this module is to download a specified number of books from Project Gutenberg, filter them by language, and store the valid books locally. The module selects books randomly by generating unique book IDs and ensures that only books in English, Spanish, or French are retained. If a book fails to meet the language criteria or cannot be downloaded, the module skips it and continues until the required number of books is retrieved.

1.1.1 The `crawler()` Function

The core functionality of the module is managed by the `crawler()` function. This function automates the process of retrieving, filtering, and saving books, using the following steps:

- **Input:** The function takes two parameters, `num_of_books`, which defines how many books should be downloaded and `datalakepath` which saves the books in the datalake directory.
- **Random Book Selection:** The function generates random numbers between 1 and 99,999 to select books from Project Gutenberg. These numbers correspond to the unique IDs of books in Project Gutenberg's catalog.

- **Book Download:** For each randomly generated book ID, the function constructs a download URL and attempts to retrieve the book's plain text version using the `requests` library.
- **Language Filtering:** Once a book is successfully downloaded, the function verifies its language using a custom `language_filter()` function. This filter checks whether the book is written in English, Spanish, or French, based on the content of the first 50 lines.
- **Saving Valid Books:** If the book passes the language filter, it is saved in the `datalake` directory with a filename corresponding to its book ID (e.g., `12345.txt`). The file is saved in UTF-8 encoding to handle special characters.
- **Skipping Invalid Books:** If the book is not in the desired languages or cannot be downloaded due to errors (e.g., an invalid book ID), the function logs an appropriate error message and skips to the next book.

1.2 Indexer Module

In a search engine, an *indexer* processes the raw data (e.g., books or text) and creates an *inverted index*, which allows for fast information retrieval based on keywords. Instead of searching through all the text for each query, the search engine refers to a pre-built index that lists words and their locations within the text.

1.2.1 Inverted Index

An inverted index is essentially a mapping where each word is associated with the locations (e.g., document IDs, line numbers) where it appears. This enables fast keyword searching, as the search engine can refer to the word's locations directly without needing to scan the entire dataset.

1.3 Query Engine Module

The purpose of this module is to process and analyze words entered by the user, searching for their occurrence in a set of books stored in JSON format. Once the words are found, the module extracts metadata from the books, such as the title, author, language, and download link. It also reads and presents specific lines from the books where the words appear. All of this is done efficiently through a modular structure and the use of classes, keeping the code clean, reusable, and easy to maintain.

This module is particularly useful for analyzing large volumes of text, searching for keywords, and retrieving detailed information about the books where these words are found. It could be employed in text analysis, research, or applications that need to process and retrieve specific information from large text datasets.

1.3.1 Two Module Versions

There are two versions of this module, each with a different approach to storing data:

- **First version (the main one)**: This version uses a *datamart* that organizes the information into a directory with two folders:
  - **Metadata folder**: This folder stores a file with relevant book information, such as title, author, and so on.
  - **Words folder**: Each processed word is stored in its own file, allowing individual access to each word and its respective occurrences.
- **Second version**: This version also uses the *datamart*, but instead of separate folders, it stores two files:
  - A file for **metadata**, which is identical to the one in the first version.
  - A single file called **words**, where all processed words are stored together. Unlike the first version, this one does not create a separate file for each word, grouping all the words into a single file, which can be simpler in some scenarios but requires more processing when searching for individual words.

The main difference between the two versions lies in how words are managed and stored, impacting the structure and access to the data depending on the application's needs.

1.4 Data structure: Single File for Words: `words`

In one approach to indexing, all the words and their occurrences are stored in a single file called `words`. This file acts as a comprehensive index, containing information about all the words found in the dataset, as well as their occurrences across different books.

1.4.1 metadatos (Metadata File)

As in both approaches, we use a `metadatos` file to store the metadata for each book. The metadata includes important details such as:

- **Book ID**: A unique identifier for each book.
- **Title**: The title of the book.
- **Author**: The author's name.
- **Language**: The language of the book.
- **Download Link**: A URL from where the book can be accessed.

An example of an entry in `metadatos`:

```
{
  "book_id_123": {
    "title": "Pride and Prejudice",
    "author": "Jane Austen",
    "language": "English",
    "download_link": "http://www.gutenberg.org/ebooks/1342"
  }
}
```

1.4.2 `words` (Words File)

The `words` file stores all words in a single dictionary structure. Each word is a key in the dictionary, and it has for value a dictionary, which it keys are book ids, and it values are dictionaries which have the frecuency of the word on this book, and every line where it appears, it follows a structure like that:

```
{
  "project": {
    "45091": {
      "frecuency": 88,
      "lines": [25, 78, 102,...]
    },
    "43456": {
      "frecuency": 20
      "lines": [15, 67, 220,...]
    }
  },
  "banana": {
    "45331": {
      "frecuency": 21,
      "lines": [55, 77, 192,...]
    },
    "47776": {
      "frecuency": 3
      "lines": [52, 47, 55]
    }
  }
}
```

1.4.3 How It Works

When a user searches for a word, the system reads from the `words` file to find the relevant entries. The file contains the occurrences of each word, the books in which they appear, and the specific lines for each book. This approach is straightforward but can lead to performance bottlenecks as the file grows with more words and books.

1.5 Alternative Approach: One File Per Word

In contrast to the single file approach, another method is to store each word's occurrences in a separate file. This structure involves having a dedicated file for each word in a directory, such as `words/`, while still maintaining the `metadatos.json` for book metadata.

1.5.1 metadatos/ (Metadata Directory)

As in the previous method, the `metadatos` file remains the same, storing important metadata for each book, such as title, author, language, and download link. This file structure is identical across both indexing methods.

1.5.2 One File Per Word: `words/` Directory

Instead of storing all words in a single file, this approach stores each word in its own file within the `words/` directory. Each word file contains the same information as the other version, having the same structure each file.

For example, the file `words/apple` would contain:

```
{
  "45331": {
      "frecuency": 21,
      "lines": [55, 77, 192,...]
    },
    "47776": {
      "frecuency": 3
      "lines": [52, 47, 55]
    }
}
```

This structure allows the system to retrieve word occurrences by reading only the relevant word file, reducing the amount of data that needs to be loaded for each query.

1.5.3 How It Works

When a user searches for a word, the system navigates to the `words/` directory and locates the file named after the search word (e.g., `apple`). The file contains all the occurrences of the word across different books, along with the specific line numbers. This approach is more scalable than the single file approach, as the file system can handle a large number of smaller files more efficiently than one large file.

1.6 Comparison of Both Approaches

- **Single File (`words`)**: This method is simpler in terms of file management but may face scalability issues as the number of words and books grows.

The file becomes larger, and every query requires reading the entire file, which can slow down the search process.

- **One File Per Word (`words/` directory)**: This method reduces the load for each query by limiting the search to a single word file. It is more scalable as the system grows since the search engine only needs to load the file corresponding to the searched word. However, managing a large number of small files can become complex as the dataset expands.

2 Benchmark Comparison Between the Two Query Engine Versions

In this section, we will present a comparison between the two versions of the query engine based on their performance. By executing a series of benchmarks, we aim to measure the efficiency of each version.

The benchmarks will help us understand the trade-offs between the modular approach, where each word is stored in an individual file, and the version that consolidates all words into a single file. The we will take into count is the searching time, it will provide valuable insights into which version is more suitable for different use cases, particularly when dealing with large datasets.

2.1 Version 1: Individual Files for Each Word

In this version, each processed word is stored in its own file within a dedicated folder. This structure allows for easier access to individual words but may have an impact on storage and search performance.

Min	**Max**	**Mean**	**StdDev**	**Median**	**IQR**	**Outliers**	**OPS**
4.9209	14.9472	5.1407	0.8697	5.0446	0.0783	2;11	194.5244

Table 1: Benchmark results for Version 1 (time in milliseconds).

2.2 Version 2: Single File for All Words

This version consolidates all processed words into a single file. While this reduces the number of files and may improve storage efficiency, it might introduce overhead in searching for specific words.

Min	**Max**	**Mean**	**StdDev**	**Median**	**IQR**	**Outliers**	**OPS**
24.5000	98.9580	26.0669	2.7333	25.6250	0.7910	112;214	38.3628

Table 2: Benchmark results for Version 2 (time in microseconds).

2.3 Analysis

Based on the benchmark results, we can conclude that the first version of the module, which uses one file per word, is significantly more efficient and scalable compared to the second version, which stores all words in a single file. Although the second version may show a slight improvement in terms of storage space by consolidating the words, this structure introduces several issues. First, the search time in the second version increases considerably because all words are contained in a single file, forcing the system to process and search through a larger amount of data for each query. This is clearly reflected in the higher search times and the larger standard deviation in the results.

Moreover, the first version allows for greater modularity, as each word is stored independently, making it easier to manage parallel or distributed queries. The second version does not adapt well to scenarios where individual words need to be modified, deleted, or added, as any change requires rewriting the entire file. Finally, scalability becomes a challenge in the second version, as the number of words grows, handling a single file becomes slower and less efficient, impacting both performance and resource usage. Therefore, the first version is preferable in terms of efficiency, scalability, and long-term maintenance.

3 Program Execution

1. **Indexer**: The first step is to execute the corresponding version of the **indexer** module, depending on the chosen data structure. The processed books are read from the datalake and stored in the datamart for future queries. In this step, it is important to specify the paths for both the datalake and the datamart.

2. **Crawler**: Next, the **crawler** is executed to download the specified number of books from external sources and store them as text files in the datalake. As with the indexer, it is crucial to define the datalake path.

3. **Query Engine**: Finally, the **query engine** allows searches to be performed once all books have been processed by the indexer. The query engine accesses the indexed data in the datamart and provides the relevant results. In this step, both the datalake and datamart paths must be specified.

4 Conclusions

This project has successfully developed and benchmarked a scalable search engine system, consisting of a crawler, indexer, and query engine. Through the evaluation of two different data structures for the inverted index, it was observed that using a separate file for each word resulted in significantly better performance and scalability compared to storing all words in a single file. The modular approach, with individual word files, allows for faster searches and easier system management, particularly as the dataset grows. This data structure also enables more efficient updates and modifications. Overall, the system demonstrates high efficiency and provides a solid foundation for handling large-scale text data retrieval.

5 Future Works

This project lays the foundation for a scalable search engine, but several improvements and extensions can be explored in future works to enhance both performance and functionality.

5.1 Optimizing the Inverted Index

Future work could focus on making the inverted index faster by using advanced data structures like *tries*, *suffix trees*, or *B-trees*. These can help speed up searches and use less memory when there are many repeated words in multiple books.

5.2 Adding Support for Phrase Searching

Right now, the system can only search for single words. A future improvement could allow searching for phrases (e.g., "artificial intelligence" instead of just "artificial" or "intelligence"). This would require adjusting the index to track word sequences and their positions.

5.3 Implementing Ranked Retrieval

Currently, the system matches keywords exactly. Adding a ranking system for search results based on word frequency, how close the words are to each other, or using techniques like TF-IDF, could show more relevant results at the top.

5.4 Parallelizing the Indexing Process

To handle larger datasets, future work could include making the indexing process run in parallel. This means processing multiple books at the same time, which would make the system faster when building or updating the index.

5.5 Expanding the Dataset and Language Support

The current system works with books in English, Spanish, and French. Future work could expand it to support more languages.

A Link to GitHub repository

https://github.com/alejandroalemanaleman/SearchEngine.git