Implementation of a Search Engine in Java for Gutenberg Project Books

Yang, Jia Hao | León Quintana, Gerardo | Nagy, Jakub
Guerra Déniz, Irene | García Nuez, Raúl | Rivero Sánchez, Raúl

University of Las Palmas de Gran Canaria, Las Palmas de Gran Canaria, Spain.

Abstract

The digital age has witnessed an unprecedented proliferation of information, and open-access repositories like the Gutenberg project have made an extensive collection of books available to the public. To facilitate the retrieval of relevant content from this vast literary treasure trove, this paper presents the design and implementation of a search engine in Java specifically tailored for indexing and searching books obtained from the open-source Gutenberg project.

Our Java-based search engine is designed to efficiently index and retrieve information from a large corpus of books, including classical literature, historical texts, and more. In this work, we provide a comprehensive overview of the architecture and components of our system. The system encompasses web crawling, text analysis, and indexing techniques optimized for books, such as metadata extraction and book-specific content analysis.

This paper also includes an evaluation of the search engine's performance, including indexing speed, retrieval times, and the quality of search results when applied to the Gutenberg project's books. Comparative analyses with other search engines further illustrate the effectiveness and efficiency of our system in the context of literary content retrieval.

In conclusion, our Java-based search engine, tailored for open-source Gutenberg project books, provides a practical and efficient means of searching through this vast collection of literary works. This contribution not only aids researchers and readers in accessing classical and historical texts but also offers insights into the development of specialized search engines for literary and academic purposes.

Introduction

In the current digital age, the volume of information available online has reached exponential proportions. This wealth of data has necessitated sophisticated processing and analysis techniques to extract meaningful insights. In this context, Project Gutenberg, an initiative aiming to digitize and archive an extensive collection of public domain books, stands as an invaluable source of literary information. However, to fully utilize this treasure trove of knowledge, efficient and precise search tools within this vast digital library are essential.

This paper presents an innovative approach to address this challenge using Big Data technologies and the Java programming language. We propose an inverted indexing system that leverages crawling capabilities to gather content from Project Gutenberg and data cleaning techniques to prepare texts for analysis. The crux of our approach lies in constructing an inverted index, a fundamental component of modern search engines, enabling users to find specific books based on keywords.

This work details the design and implementation of this system, highlighting challenges encountered during development and proposed solutions. Additionally, results obtained through rigorous testing are presented, demonstrating the efficacy and accuracy of our approach in searching for information within the extensive repository of Project Gutenberg. This study not only represents a significant advancement in the field of indexing large volumes of literary data but also showcases the feasibility and utility of applying Big Data technologies in the preservation and accessibility of historical and cultural knowledge.

Methodology

The project has been implemented using three modules, to divide the responsibilities of manufacturing the search engine. In this section we will explain the structure of each module, how they work and their main characteristics. This search engine is composed of three modules: Crawler, Indexer and Query.

Crawler

First, let's delve into the functionality of the Crawler module. This essential component of our system is responsible for procuring books from the Gutenberg Project and populating our datalake with the downloaded books. To efficiently achieve this, we've meticulously designed two core classes: the **BookDownloader** and the **FileEncodingHandler**.

- **BookDownloader**: This class is responsible for downloading books from the Gutenberg Project, using a randomly generated number within the range of 1 and 70,000 as a parameter for book selection.
- **FileEncodingHandler**: The Gutenberg Project houses an extensive collection of books, spanning a wide range from classic fairy tales to non-eligible content. To distinguish between eligible and non-eligible books, we've developed a specialized class designed to identify books that are written in plain text. If a downloaded book does not meet this criterion, it is automatically flagged for removal from the collection.

The datalake implemented for this project adheres to an expansion tree-directory structure (3), organized by the last digit of each downloaded book's ID. To prevent over-expansion of the directory tree, we've further subdivided the downloaded books based on their download date.To establish this data lake, we've developed two key classes: **DateTreeDirectoryBuilder** and **TreeDirectoryBuilder**.

- **TreeDirectoryBuilder**: The class is responsible for creating the initial tree-directory structure and expanding it as needed when the capacity limit is reached, ensuring efficient management of the data storage.
- **DateTreeDirectoryBuilder**: This class is tasked with segregating the tree-directory based on the download date of its contained books.

To address potential concurrency issues, we've implemented a progressive download mechanism for the books. This is achieved through the use of a TimerTask encapsulated within a class called **DownloaderTimerTask**. The primary responsibility of this class is to oversee the complete process of creating the datalake and downloading books, and it operates on a 24-hour schedule.

On a daily basis, the **DownloaderTimerTask** class ensures the retrieval of a random selection of 250 books. This approach not only mitigates concurrency challenges but also allows for a controlled and manageable download process, ensuring the availability of fresh data in the datalake without overwhelming the system.

Indexer

Let's delve into the details of the Indexer module. This vital component plays a crucial role in our system by meticulously dissecting the books stored in the datalake into their respective content and metadata. Furthermore, it performs the intricate task of constructing an inverted index based on the words found within all the content files. This inverted index will subsequently function as a valuable datamart for our

final module.

The Indexer module is thoughtfully divided into two distinct parts: the Library package and the Indexer class itself. This organizational structure enhances the efficiency and manageability of our indexing process, ensuring a seamless transition of data from the datalake to the datamart.

Let's begin by discussing the significance of the Library package. This package serves a critical purpose in our system, facilitating the separation of metadata and content within the downloaded books. To achieve this, we have introduced an interface named **FileManager** . This interface is implemented by two distinct classes: **MetadataFileManager** and **ContentFileManager**. Both of these classes inherit the 'separate' method, which plays a pivotal role in dividing the content and metadata.

In addition to **ContentFileManager** and **MetadataFileManager**, there is another essential class responsible for creating the **Library** directory. This directory is designed to house both the Content and Metadata folders, where the results of the separation process conducted by the **FileManagers** are stored. This organized structure ensures the efficient management and storage of the split data, enhancing the overall functionality of the Library package.

Finally, let's explore the role of the Indexer class. This class is responsible for the generation of an inverted index from the content stored within the **Content** folder, which is located within the **Library** directory. Each word in the content is indexed and given a corresponding .txt file with its name. Within these files, each line represents an entry related to a book that contains the indexed word. These entries are structured with semicolons to separate the following information, **Book's ID**, **Book's Title**, **Word appearance**.

The inverted index, created by the **Indexer** class, plays a pivotal role as the datamart for the primary objective of our project. Similar to the datalake, the datamart is organized using an expansion tree-directory structure (4). However, in this instance, the classification is based on the first two letters of the indexed words.

This hierarchical organization ensures efficient and systematic data retrieval, allowing us to access information by navigating through the appropriate directory based on the initial letters of the indexed words. This approach enhances the accessibility and usability of the datamart, making it a valuable resource for our project's core functionality.

Query

Finally, we have the Query module. This module is the connection between the user and the inverted index, through this module we have implemented a Web-Service in charge of establishing an API connection in order to execute the search engine.

The functionality of the API is enabled through the integration of a third-party library known as Spark. This library allows us to establish a local server on our computer, serving as the intermediary link between users and the datamart, as previously discussed. The key class responsible for orchestrating this connection is the "WebService" class. This class comprises three essential methods to generate responses:

- **Browse Method:** This method facilitates the creation of the API's GET request, with the URL path "/search/<word>". Here, "<word>" is a variable representing the user's search query within

the datamart.

- **Response Method:** The "response" method takes on the responsibility of crafting responses for the API. It returns a list of book titles sorted in descending order, with the titles containing the most instances of the search term appearing first.
- **Sorting Method:** As previously mentioned, an additional method is required to sort the book titles based on the number of relevant words they contain.

Within the "Query" module, you will find additional classes with specific, lower-level responsibilities that are integral to the proper functioning of the module. Although these classes are essential for the module's operation, they are not detailed in this description for brevity.

Experiments

In this section, we will delve into the experiments conducted during the project's development, which revolved around enhancing time optimization, addressing concurrency issues, and improving memory efficiency.

Experiments in the Crawler

During the development of this module, various experiments were conducted to identify the most effective implementation strategies. As mentioned in the methodology, the organization of information was structured in a tree-directory format, categorized by day. This approach proved to be significantly more efficient than storing all data in a single directory. One key advantage is that it simplifies the process of searching through the datalake, particularly as the volume of data grows. Furthermore, this tree-directory structure can be readily expanded if necessary to accommodate an even larger data volume

Another experiment involved addressing the issue of potential book repetition during the downloading process. As random books from The Gutenberg Project were being downloaded, it became apparent that as the number of downloaded books increased, the likelihood of downloading non-repeated books in each iteration decreased. To mitigate this, a system was required to verify the uniqueness of each book identifier.

Several approaches were considered, each with its own set of advantages and drawbacks. These approaches included using a database for verification, but its efficiency depended on indexing and structure. Another idea was to employ plain text or JSON files for verification, but this method became progressively slower as the file size grew. The final approach considered was utilizing a Bloom filter.

A Bloom filter is a probabilistic data structure designed to test whether an element belongs to a set. It operates by hashing elements and storing the hashed values in a bit array. When querying the filter for an element, it hashes the element and checks if all corresponding bits are set. If any bit is not set, the element is confirmed as not in the set. However, if all bits are set, the element is considered probably in the set. It offers fast set-membership testing but may produce false positives.

In the context of a Bloom filter, it is used for probabilistic book identifier verification, meaning it can sometimes generate false positives (indicating repetition when there is none). This might not be the most memory-efficient approach and can potentially impact program performance. Ultimately, the chosen implementation relied on a Java Set to verify uniqueness. While this method may consume more memory

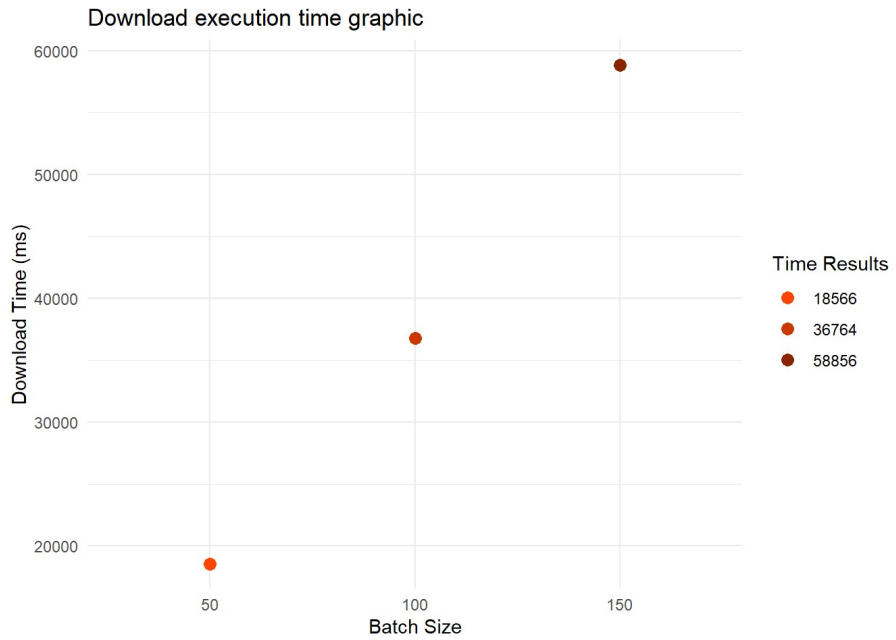
and potentially slow down the program, it was deemed the most practical solution for addressing the issue of random download uniqueness.

Ultimately, we will present the outcomes of simultaneously downloading multiple books in an effort to determine the most optimal batch size for downloads.

Table 1: Download execution time.

Download Batch Size	Time (ms)
50	18566
100	36764
150	58856

Figure 1: This is a graphic for the download execution time



Experiments in the Indexer

During the development phase of the Indexer module, we established two critical foundational principles to guide our path forward. First, we determined that the datamart should closely mirror the structure of the datalake. Second, we recognized the importance of segregating metadata from the content of downloaded books. These three key points formed the basis for our experimentation with this module.

In our quest to separate content and metadata, we embarked on a journey that involved altering our approach to classifying these components. Instead of using a single class to handle both content and metadata, we opted to create distinct classes for each. This decision was driven by the challenges we encountered, particularly with large content-rich books like Don Quixote. We faced concurrent processing issues, and it became evident that a more specialized approach was needed.

In the context of building the datamart, our initial attempt involved assigning a unique ID to each word and then classifying these IDs, as we had done in the datalake. However, this method proved to be inefficient and posed a risk to the overall project's performance. As a solution, we opted to emulate

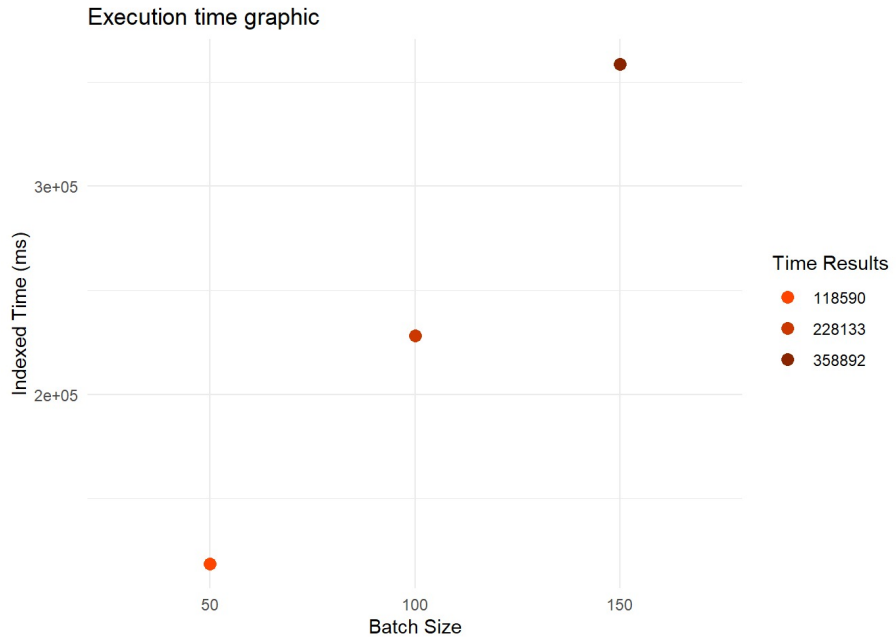
certain aspects of the datalake's structure. In this new approach, each file was named after the word itself instead of an ID, and words were classified based on the first two letters, creating a system reminiscent of a traditional dictionary. This experiment not only resolved efficiency problems but also significantly improved the response time of our search engine.

In conclusion, we will unveil a selection of findings stemming from the indexing of a substantial collection of books, all in pursuit of pinpointing the most efficient batch size for book indexing.

Table 2: Indexing execution time.

Indexing Batch Size	**Time (ms)**
50	118590
100	228133
150	358892

Figure 2: This is a graphic for the indexer execution time



Conclusions

In summary, the incorporation of a tree-directory structure within the data lake emerges as a pivotal factor for enhancing efficiency and performance, particularly when migrating from SQL or dictionary inverted indexing to an alternative tree-directory system. The integration of the Library folder has not only streamlined the identification of recently downloaded books but has also significantly expedited the indexing process.

The project's primary drawback lies in the execution time of the indexer, which can be addressed by adopting a strategy of indexing smaller batches of downloaded books rather than the entire batch.

Future work

In this project, we have introduced a variety of ideas with potential applications in the field of Big Data. These include the utilization of tree directories for data organization instead of relying on a single database or a monolithic folder structure, as well as the practice of storing words within individual files while embedding the necessary information. However, it's important to acknowledge that these implementations, while effective, are not without room for improvement.

Crawler improvements

As previously mentioned, our current implementation involves the downloading of random books from The Gutenberg Project and relies on a Set stored in memory to control these downloads. While this approach has been effective, it has its limitations, especially when dealing with larger volumes of data. As we look to further enhance the efficiency and scalability of our application, we can focus on improving something like optimizing the download control and the parallelization of downloads.

Indexer improvements

In the context of indexer improvements, similar to the enhancements mentioned in the crawler section, the current implementation relies on a Set to keep a record of indexed books. For future iterations, optimizing this feature to conserve memory for other processes is advisable. Additionally, exploring parallelization of the indexing process is worthwhile, recognizing its inherent slowness due to the need to read the entire book and perform various intermediary tasks. Looking ahead, a potential enhancement could involve selectively downloading books that are not in English. Currently, our implementation verifies the language of each book, ensuring only English books are downloaded.

Query improvements

As of the current implementation, the system operates smoothly, largely owing to the effective development of the indexer. The structured tree directory within the indexer significantly streamlines the search process. To illustrate, consider that with just 40 downloaded books, we have accumulated more than 48,000 distinct words. Indexing a specific word from this extensive set could be time-consuming. However, the design of the tree directory greatly expedites the search. For instance, when searching for a word like "glass," it is located within the sub-directory "./g/l/," which contains only 160 different words. This hierarchical organization substantially enhances the efficiency of the search process.

Appendices

Figure 3: Example of the expansion tree-directory

Figure 4: Example of the datamart structure