Search Engine Project

Inverted Index

Adam Breznen[1], Ricardo Juan Cárdenes[1], Joaquín Ibáñez[1], Mara Pareja[1], and Susana Suárez[1]

[1]Faculty of Computer Science
[1]Las Palmas de Gran Canaria University

November 5, 2023

Abstract

In this article, we present an approach to efficiently create an inverted index from text documents using the Java programming language. We aim to address the challenge of building an inverted index from large textual datasets.

Initially, we considered two possible approaches with the goal of achieving maximum speed and efficiency in execution.Consequently, we scrapped our initial idea, which was to use a NoSQL database fed from a full-book datalake, after a brief investigation. Instead, we chose to separate the content and metadata of such free books to implement a file structure which led to a noticeable increase in efficiency and speed of execution, The results derived from our experiments support the effectiveness of our solution.

Additionally, we analyze future improvements involving the development of user interface or the use of metadata to enrich search capabilities.

Keywords inverted index, google cloud platform, search engine, project guthenberg

1 Introduction

Generating an inverted index from text documents is an essential component for efficiently structuring and searching information in today's vast digital landscape. In this article, we meticulously investigate an approach to address this undertaking, making use of the Java programming language and data storage technology, in particular, One Drive and Google Cloud.

The need for an efficient inverted index in information processing and crawling is not new. It has been a mainstay in information retrieval, search engines and word processing applications. However, in an environment characterised by constant evolution and the proliferation of digital data, efficiency and speed in the design and use of these indexes are of paramount importance.

The problem we address in this research focuses on the urgency of creating an efficient system capable of processing large volumes of text and allowing agile and accurate searches in these documents. To this end, we ask ourselves whether a data container organised in files whose name is the word and the text it contains are the ids of documents in which it appears, such as a Google Cloud bucket, could represent a more efficient alternative in terms of performance compared to traditional relational and non-relational databases.

The main contribution of this study lies in the successful implementation of an approach that amalgamates metadata and content segregation, text normalisation and the adoption of such a data container, highlighting its efficiency and speed in the design of an inverted index. Furthermore, this research lays the groundwork for future developments, such as the refinement of metadata search and the development of a user interface. These collective contributions have the potential to have a substantial impact on the efficiency of information search and organisation in contexts characterised by the abundance and diversity of textual data.

2 Problem Statement

The challenge lies in effectively organizing and searching the information contained in a large number of text documents. As the volume of digital data continues to grow, there is a pressing need for a method to quickly and accurately index and retrieve information from these documents. The creation of an efficient inverted index presents itself as a fundamental solution to address this problem. The problem to be solved is broken down into several key aspects:

- **Document segmentation and construction of a Datalake.** It is essential to develop an approach that enables the division of documents into metadata (descriptive information) and textual content of the book. This separation is indispensable to achieve an organized and efficient structure in the Datalake, enabling effective search and retrieval of information in an ever-expanding digital environment.

- **Text normalization.** It must be ensured that the text is normalized so that searches are insensitive to differences in capitalization and irrelevant words are excluded.

- **Efficient storage and access.** The organisation and data container used must be efficient to allow fast searches and accurate retrieval of information.

- **Scalability** The solution must be scalable to handle large text datasets.

3 Solution/Methodology

The solution proposed in this study addresses the task of generating an inverted index from text documents in an efficient and scalable way. Our approach combines several essential steps that enable the creation of this index effectively and facilitate its replication by other researchers.

3.1 Crawler Application

Crawler service is designed to facilitate the automated retrieval and archival of books from Project Gutenberg. It accomplishes this by downloading books from the Project Gutenberg website and subsequently storing them in a local repository. Furthermore, it leverages a message broker to notify other components of the system whenever a book has been successfully downloaded and added to the repository. This workflow is designed for efficiency and scalability, making it suitable for various applications in the context of digital libraries, content curation, or text analysis. Additionally, the local repository directory is synchronized with OneDrive, enhancing flexibility and ensuring that the archived books are accessible across different platforms and devices.

3.2 Cleaner Application

In this module of the project, the incoming data, which consists of books, is segmented into two distinct components: metadata and content. Subsequently, they undergo a cleaning process in order to retain only the words that are relevant for the indexer.

The process starts when a new event is detected in a message queue. The new events are divided into metadata and content. Subsequently, a text cleaning procedure is employed through the Cleaner component. The purpose of this process is to remove stopwords and unwanted characters from the content. Finally, we store the content in a datalake together with the metadata. In addition, an event log is generated to document this operation.

The removal of stopwords is executed in a customised way, taking into account the language of the book in question. Initially, five languages have been considered: English, Spanish, French, Italian and Portuguese. The language-specific stopword lists are obtained from corresponding files, which guarantees an accurate cleaning of the content.

In addition, the project offers access to book information via a web-based application programming interface (API). This API provides complete data about the book, as well as individual access to its content and metadata.

3.3 Indexer Application

Indexer application, as it can be deduced by its name, is the one responsible for indexing datalake books to the inverted index that will be used for queries later on. It has been built following MVC software architecture. In model, we only have the class FileEvent, that is useful to keep track of each document that has been whether read or indexed.

In controller, we have the code that gives real functionality to the application. There, you can find a Controller class, which is in charge of running both indexer and reader threads, and make use of the mentioned broker architecture. That way, readerThread is always consuming messages form ‘datalakeEvents’ and publishing into ‘readEvents’ once it has load a processed file to the module resources. Then, indexerThread receives the notification in ‘readEvents’ and starts the indexing process for that file.

Finally, we have the View package, where the API is constructed. This API will give several information about the module, such as the indexed documents, the time it took, or the inverted index itself. All of this is bring to life in the Main class.

3.4 Message Broker Application

For this project to work, the task of listening to file additions in the datalake should be implemented in both Cleaner and Indexer modules. To do so, one could use some listeners already implemented in Java, such as WatchService, which allows you to listen to some kind of events related to a local repository in your computer. However, when the datalake organization gets more complex, as happens when structuring information by date, you need to use more scalable techniques.

Probably, using a broker is the best solution in terms of software engineering due to its modularity and ease of use. The trick here is to let Crawler publish each file addition to the datalake in the 'datalakeEvents' queue at the broker, which will be heard by the Cleaner module. That way, once the Cleaner receives the notification with the URL of the added book in 'datalakeEvents', it can get its content and process it. When finished, the URL of the resulting processed content file that Cleaner stores in the datalake will be posted to another queue, where the Indexer will be waiting for that notification to index the file.

The main purpose of using a broker, apart from project readability, is to ensure mutual exclusion between parallel modules. This way, due to the order that broker queues ensure, several cleaners and indexers could be running at the same time, without processing the same datalake file twice.

3.5 Query Engine Application

The Query Engine application is in charge of providing those queries necessary to access the datamart and the datalake in order to provide information about the contained data. Its construction follows the 'Model View Controller' software architecture.

- **Model package.** In the model package there are the different classes that are used to build objects that contain one or another information depending on the query made.
- **Controller package.** The controller has been divided into two main parts: 'readDatalake' and 'readDatamart'. Each package contains the code that processes the API queries. The first one is in charge of reading the datalake located in One Drive and the second one reads from the datamart located in a Google Cloud bucket. Both packages read from their respective data repositories, process and return an object depending on the query made.
- **View package.** Finally, the view is where the real functionality of the application is built, which consists of carrying out the queries. Here we find the construction of an Application Porgramming Interface with the following queries:

1. /datamart/:word - Receives a word and returns a JSON with the word and ids of the documents in which it appears.
2. /datamart-search/:phrase - Receives a phrase and returns a JSON which is a list of objects where each word appears and the IDs of the documents in which each word appears.
3. /datamart-recommend/:phrase - Receives a phrase just like the previous query but returns only a JSON with the book or books that contain the most words in order to recommend to the user the book that best fits their query.
4. /datamart-frequency/:word - Receives a word and returns a JSON with the word and its frequency of occurrence.
5. /datalake/metadata/:idbook - Receives the book's ID and returns all of its metadata.
6. /datalake/:idbook - Receives the book ID and returns the book as it was originally.
7. /datalake/content/:idbook - Receives the ID of the book and returns its content processed and without the metadata.

4 Experiments

This section will explain why certain decisions have been made regarding the architecture and design of the different applications.

4.1 Construction of Datalake

On the one hand, we would like to highlight the decision to split documents into "Content" and "Metadata" directories, rather than continuously analyzing the document structure to identify metadata boundaries. This approach, involving upfront separation of metadata and data a scalable option going forward rather than the second option, processing the documents "raw" each time to locate the metadata sections.

On the other hand, the choice to use the OneDrive platform for the creation of a Data Lake in a project is based on several key considerations. First, OneDrive offers a secure and highly scalable cloud storage solution that allows organisations to efficiently store, organise and access large volumes of data. In addition, OneDrive is integrated with other Microsoft tools, making it easy to collaborate and share data between different applications.

4.2 Crawler and Cleaner in the same server

The decision to host the Crawler and Cleaner applications on the same server is based on the fact that the physical proximity of both applications on the same server can reduce latency and network overhead, resulting in more efficient communication between the two. This is especially important in situations where the Crawler must deliver collected data to the Cleaner for immediate processing.

In addition, coexistence on the same server facilitates the administration and monitoring of both applications, as they share hardware and operating system resources, which simplifies the management of updates, patches and maintenance tasks. It can also help optimise resource usage, as server resources can be allocated and shared more efficiently between the two applications.

4.3 Datamart composed of files

Our DataMart model is based on the organisation of data in files divided by keywords. Each file is named after a specific keyword, and within the file are listed the identifiers (IDs) of the documents in which that keyword appears. This approach facilitates the search and retrieval of information related to specific keywords and allows quick access to relevant documents based on their content.

Originally, the datamart consisted of a non-relational database. This database was much faster than the relational database. However, a faster alternative has been the model explained above. For this, two different tests were carried out.

4.3.1 Database creation time

In the first set of experiments, we evaluate the time needed to create an empty database for both NoSQL and file database management systems. Specifically, we aim to measure the time it takes to initialise the datamart without documents. This benchmarking process is crucial to understand the comparative performance of each system when there is no existing data to process. We repeated this experiment for both the NoSQL database and the file-organised structure, allowing a direct comparison of their creation times.

In this scenario, the NoSQL database actually underperformed, systematically outperforming the file repository in terms of database creation time. This observation underlines the advantage of such a model for quickly building data warehousing systems when starting from scratch.

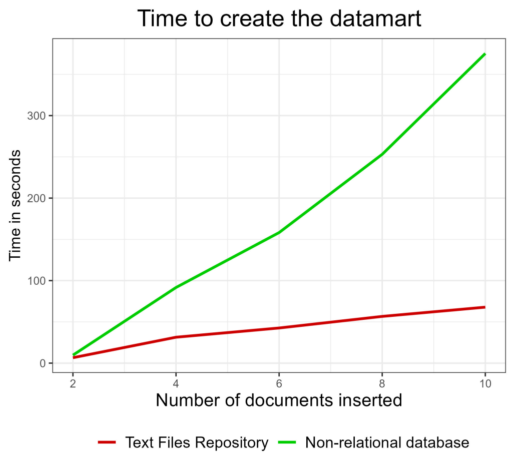

Figure 1: Datamart Creation Time Comparison

We conducted these experiments with different document counts, from two to ten text files, which allowed us to observe how the creation time of each database system increases with document count.

4.3.2 Insertion performance

Now, we focus on the performance of inserting documents into an already initialized database for both technologies. To do so, we start with an existing document and measure the time it takes to insert it into the 10-book database created for the previous section. We continue this process incrementally, inserting two, three, four, and five more documents, in pursuit of measuring how much time it takes to update our datamart depending on the number of files that are going to be indexed.

Consistent with our results from the datamart creation time experiments, the file repository demonstrated higher insertion performance. This trend was evident in all phases of document insertion, highlighting the efficiency and scalability of such a system in accommodating data growth and updates in a dynamic environment. These results reaffirm the advantages of a file-structured repository in scenarios that require fast and efficient data manipulation and updates.

The graphs presented in this section, while showing how this model can outperform docu-

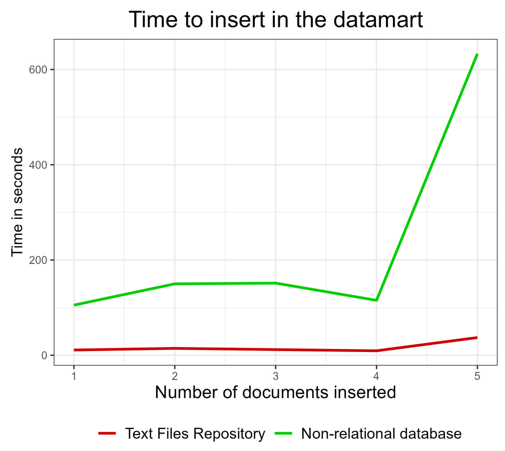

Figure 2: Datamart Insertion Time Comparison

ment indexing, do not show the actual behaviour of this technology. If we take a look at Figure 2, it may seem that inserting five documents is much worse than inserting only four. Furthermore, it seems that inserting four files is less complex than inserting three. The explanation behind this phenomenon is that the files involved do not have the same size in terms of memory for each step. Thus, inserting a single document could be even slower than inserting five, if the first one is larger than all the others put together.

4.4 Datamart in Google Cloud Platform

The decision to host the above DataMart in a Google Cloud bucket is based on several strategic factors. First, Google Cloud provides a highly scalable and secure cloud storage platform that facilitates large-scale data management. Google Cloud Storage buckets enable efficient storage and data organisation, which is critical to the file-based DataMart model.

In addition, Google Cloud provides a range of integrated tools and services that simplify data management, access and analysis, streamlining the process of extracting useful information from the DataMart. Security and access control through accounts and passwords with different permissions ensures that sensitive information is protected.

5 Conclusion

This project presents a comprehensive approach to efficiently create an inverted index for textual data. It highlights the importance of metadata extraction, text normalization, and selecting a good architecture for the datamart in order to improve retrieval performance.

Separating metadata from data and normalizing text contribute to accurate and efficient query processing. In addition, the results of the benchmarks favor the file repository architecture as the preferred system, due to its superior runtime performance, which ensures fast retrieval of information.

This project not only addresses the specific challenges of creating inverted indexes but also provides valuable insights into best practices for effectively managing and querying textual data.

6 Future Work

Future work on this project will explore the use of metadata to improve search capabilities. By leveraging stored metadata, the system can provide more context-sensitive search results and improve the user experience.

In addition, as the project grows, consider optimizing the performance and scalability of each of the applications. This could include using parallelism techniques in cleaner modules or querys. It is also considered for the future, moving the datalake to Google Cloud with different permissions or even making different applications that work in parallel.

On the other hand, it is considered to develop a user interface to facilitate the user a personalized search rather than through an API.

Finally, another idea we had is that incorporating machine learning algorithms for text classification and sentiment analysis could enable the system to provide deeper insight into document content. This feature could be especially useful for applications such as content recommendation and trend analysis. On the other hand, it is considered to develop a user interface to facilitate the user a personalized search rather than through an API.