Crawler and Inverted Index in Java

Aurora Zuoris
aurora.zuoris101@alu.ulgpc.es

Alejandra Ruiz de Adana Fleitas
alejandra.ruiz104@alu.ulpgc.es

Lam Truong Nguyen
lam.nguyen101@alu.ulpgc.es

Aris Vazdekis Soria
aris.vazdekis101@alu.ulpgc.es

Jaime Ballesteros Domínguez
jaime.ballesteros101@alu.ulpgc.es

Anna Barbara Król
anna.krol101@alu.ulpgc.es

November 5, 2023

Abstract

In this project, we undertook the task of transforming code in Java to enhance search efficiency among a wide variety of text documents. The primary challenge we faced was the need to quickly access documents containing specific terms within an extensive library. Our code encompasses three concurrent modules: Crawler, Indexer and QueryEngine, all working in unison to ensure seamless operation. To accomplish this task, we followed these steps: we retrieved and collected digital documents, then processed and indexed these documents for efficient search, and provided a user-friendly interface for searching and retrieving specific documents within the extensive digital library.

1 Introduction

In today's world there is an ever increasing number of documents that are being created and stored digitally. Thus the need to easily find and access these documents becomes ever more important. This is where the inverted index comes in. An inverted index is a method of creating indices for a given set of documents that allows for fast searching of the documents based on the words they contain. The 'crawler' module is responsible for retrieving, processing, and storing textual content from books available on the Gutenberg.org website. It ensures that these texts are integrated seamlessly into the inverted index. Furthermore, the 'Query Engine' module enables searching for and retrieving specific books within this extensive digital library. The harmonious synergy of three modules addresses the challenge of efficiently accessing and exploring an ever-expanding collection of digital documents.

2 Methodology

Our project follows a systematic approach, divided into three major modules: the Crawler, Inverted Index and Query Engine. First, we employ a data collection program within the Crawler module to download documents from the Gutenberg Project. Then, in the Inverted Index module, we process these documents by tokenization, creating a reverse keyword index. Subsequently, the Query Engine module provides a user-friendly interface for searching and retrieving specific documents. Finally, we store the indexed data in an SQLite database for future querying.

2.1 Crawler

The "Crawler" module consists of a set of interconnected classes, each contributing to the execution of this complex task. This detailed description sheds light on how these classes work together in the process of downloading and handling book content. The classes mentioned work as follows:

- The `ContentExtractor` class plays a fundamental role in the process by fetching text from specific books hosted on Gutenberg.org. It is capable of extracting textual data from the site using HTTP requests. To initiate the process, it requires a unique book identifier, and upon successful retrieval, it returns the book's text. This class serves as the initial point of contact with the online library.
- The `FileSystemDatalake` class is responsible for the storage and management of the downloaded textual content. It converts the raw book text into JSON format and stores it in files, with each file named after the respective book's identifier. This class plays a crucial role in creating and maintaining the Datalake directory structure, ensuring organized and efficient data storage.
- The `DocumentParser` class is pivotal in the process of extracting meaningful information from the downloaded book text. It utilizes regular expressions to search for and identify various attributes, such as the book's title, author, publication date, language, and, most importantly, the book's actual textual content. This class ensures the structured extraction of metadata and book text.
- The `TimerAPI` class provides an interface for configuring essential parameters related to the book retrieval process. It allows for the adjustment of the execution period, which determines how frequently the system fetches new books. Additionally, it enables users to specify the number of books to download per minute and set a maximum limit for the number of books to retrieve. This class is accessible through an HTTP interface, making it a valuable tool for process customization.
- The `Controller` class serves as the orchestrator of the entire process. It initiates a timer that cyclically triggers the execution of operations for downloading and saving book texts. During this continuous process, it leverages the functionality of the `ContentExtractor`,

`FileSystemDatalake`, and `LibroGutenberg` classes. By doing so, it ensures that books are fetched, processed, and stored systematically.

Finally The main `Main` class initializes the entire system. It sets the port for the system, initializes the Spark server, and passes the location of the Datalake directory to the `Controller` class.

2.2 Indexer

The Inverted Index module is essential for creating and managing inverted indices based on available textual documents. This module plays a crucial role in information retrieval, text analysis, and natural language processing tasks.

- The `FileDatalake` class represents access to the source of textual data stored as files.
  - Function `documents()` returns a set of document identifiers available in the data source.
  - Function `readDocument(int document)` reads the text content of a specific document based on its identifier.
- The `Indexer` class is responsible for creating an inverted index based on documents stored in the `FileDatalake` and storing it in the `Datamart`, with the actual implementation being injected as a dependency on creation. The class provides the following functionalities:
  - `tokenize(String document)`: Tokenizes the text content of a document, this is done using a state machine pattern that processes the text character by character, trying to be much more efficient than a regular expression. It returns a set of tokens found in the document.
  - `index()`: Initiates the indexing process. It creates an inverted index from the available documents in the `FileDatalake` and stores it in the `Datamart`. After it finishes, it blocks in a loop, listening in on new documents being added to the datalake, and updating the index accordingly as they are added.
- The `IndexerState` enumeration defines the states for the tokenization state machine used by the `Indexer` class. The states are as follows:
  - `OutOfToken` The state machine is not currently processing a token.
  - `InToken` The state machine is currently processing a token.
  - `InValidTokenUnknownEnd` The state machine is in an ambiguous position, where whenever the current token ends early or continues depends on future characters.
  - `InInvalidToken` The state machine is currently processing an invalid token, and thus should ignore future valid characters until the token ends. This can for example occur when the token is too long, as no one is going to search for tokens that are many times longer than an average word.

Finally the `Main` class serves as the entry point for the program. It accepts two command-line arguments: the path to the `datalake` (data source) and `datamart` (destination for the inverted index). The program creates instances of `FileDatalake` and an instance the `Datamart` and then initiates the indexing process using the `Indexer` class. Aditionally, this module has various unit tests for ensuring the correctness and robustness of the tokenization implementation, as this part is the most complex and error-prone in the project, especially considering that it deals with any arbitrary unvetted text coming in from the internet.

2.3 Query Engine

The Query Engine module is responsible for providing an API interface and search functionality for indexed documents. This section provides an overview of the key classes and their interactions within the Query Engine module.

- The `SparkWebService` class utilizes the Spark library to launch an HTTP server. This server listens on a specified port and handles HTTP requests. It provides two endpoints,
  - `/v1/search`, allowing users to search for documents based on keywords, with a pagination feature for retrieving results in batches.
  - `/v1/document/:id`, allowing users to get a particular document.

- The `Document` class represents an individual textual document. It includes information such as document ID, date, author, content, title, and language.

The `Main` class contains the `main` function, which serves as the entry point for the Query Engine module. It accepts two command-line arguments: the path to the `datalake`, which is where the documents themselves are stored, and to the datamart `datamart`, which is the index of all the documents. It then initializes the `SparkWebService` class and makes it listen for HTTP requests.

2.4 Datamart

The Datamart module handles the storage and management of the inverted index. It defines methods for indexing keywords, retrieving documents associated with a keyword, and obtaining a list of all available documents in the index. Two implementations are provided: a SQL-based Datamart using SQLite and a File-based Datamart using file storage.

- The SQL-based Datamart in `SqlDatamart` module stores the inverted index using a SQL database, associating keywords with document lists.
- The File-based Datamart in `FileDatamart` module employs file storage to maintain the inverted index, where each keyword corresponds to a file containing document IDs.

The Datamart module provides a reliable solution for the storage and retrieval of the inverted index. The choice between provided implementations depends on project requirements and scale. File-Based approach is suitable for smaller-scale projects where a full database system might be unnecessary, offering flexibility and efficiency in data storage.

3 Results and conclusions

The biggest improvement is the new tokenization algorithm, which is orders of magnitude faster than the old methods used in the project. Also given that it is implemented by hand, it is much more flexible and can be easily modified to suit the needs of the project.

The code of this project can be found in the following repository: `https://github.com/Aurora2500/inverse-index-big-data`

4 Future work

In the future, the new file datamart could be improved further to be faster using all sorts of different caching techniques to reduce the number of IO operations. Aditionally, various parts of the indexer could be improved further by having its own cache and flush features added, so that it doesn't flush all the indexed documents to the datamart one by one.