Datamart-Builder Task2

Laura Lasso, Miriam Méndez, José Gabriel Reyes, Alejandro González y Andrea Mayor

November 2023

Contents

1 Abstract **4**

2 Introduction **4**

3 Module 1: Clean-Books **5**
- 3.1 Class: Controller . . . 5
  - 3.1.1 Method: execute() . . . 5
  - 3.1.2 Method: checkTodayFolderAndExecute() . . . 5
  - 3.1.3 Method: scheduleMissingFilesCheck() . . . 5
- 3.2 Class: GutenbergCleaner (implements BookCleaner) . . . 5
  - 3.2.1 Method: cleanText(content: String) . . . 6
  - 3.2.2 Method: readWords(path: String): List¡String¿ . . . 6
- 3.3 Class: GutenbergSplitter (implements DocumentSplitter) . . . 6
  - 3.3.1 Method: splitDocument(path: String) . . . 6
  - 3.3.2 Method: prepareMetadata(text: String) . . . 6
  - 3.3.3 Method: storeFile(name: String, content: String, type: String) . . . 7
- 3.4 Class: MissingFilesManagement . . . 7
  - 3.4.1 Method: getMissingFiles() . . . 7
  - 3.4.2 Method: removeDirectory() . . . 7
  - 3.4.3 Method: splitMissingFiles() . . . 7
- 3.5 Class: UTF8FileChecker . . . 7

4 Module 2: Crawler **7**
- 4.1 Class: BatchDownloader . . . 8
  - 4.1.1 Constructor: BatchDownloader(batchSize: int, gutenbergFileReader: GutenbergFileReader, bookIds: String[]) . . . 8
  - 4.1.2 Method: download() . . . 8
  - 4.1.3 Method: setBooksBatch(mod: int) . . . 8
  - 4.1.4 Method: getLastBookId(baseDirectory: String): int . . . 8
  - 4.1.5 Method: extractIdFromDirectoryName(directoryName: String): int . . . 8
  - 4.1.6 Method: URLSetter(id: int): String . . . 9
- 4.2 Class: Controller . . . 9
  - 4.2.1 Constructor: Controller(batchsize: int) . . . 9
  - 4.2.2 Method: execute() . . . 9
  - 4.2.3 Method: run() . . . 9
- 4.3 Class: GutenbergFileReader (implements BookReader) . . . 9
  - 4.3.1 Method: downloadFile(fileURL: String) . . . 9
  - 4.3.2 Method: read(bookID: String, type: String): String . . . 9
  - 4.3.3 Method: getFilePath(bookId: String, type: String): String 10
- 4.4 Class: GuttenbergDatalakeCreator (implements DatalakeCreator) 10
  - 4.4.1 Method: createFolder(folderPath: String) . . . 10

4.4.2 Method: setFilePath(filename: String, currentDate: Date, type: String): String . . . 10
4.5 Class: Main . . . 10
4.5.1 Method: main(String[] args) . . . 11
4.6 Interface: BookReader . . . 11
4.6.1 Method: read(bookID: String, type: String): String . . . 11
4.6.2 Method: downloadFile(fileURL: String) . . . 11
4.7 Interface: DatalakeCreator . . . 11
4.7.1 Method: createFolder(folderPath: String) . . . 11
4.7.2 Method: setFilePath(filename: String, currentDate: Date, type: String): String . . . 11

5 Module 3: Datamart-Builder 11
5.1 Book Class . . . 11
5.1.1 Important Methods . . . 12
5.2 Associate Class . . . 12
5.2.1 Important Methods . . . 12
5.3 Word Class . . . 12
5.3.1 Important Methods . . . 12
5.4 DatabaseWriter Class . . . 13
5.4.1 Important Methods . . . 13
5.5 FileDatalake Class . . . 13
5.5.1 Important Methods . . . 13
5.6 Datalake Interface . . . 13
5.7 Controller Class . . . 14
5.7.1 Important Methods . . . 14
5.8 Datamart Interface . . . 14
5.9 Main Class . . . 14
5.10 SqliteWriter Class . . . 14

6 Module 4: book-api 14
6.1 Architecture and Design . . . 14
6.1.1 Database Connection Management . . . 15
6.1.2 Search Endpoint . . . 15
6.2 API Operations . . . 15
6.3 Indexing and Searching . . . 15
6.4 Development Considerations . . . 16

7 Conclusion 16

8 Future Work 16

1 Abstract

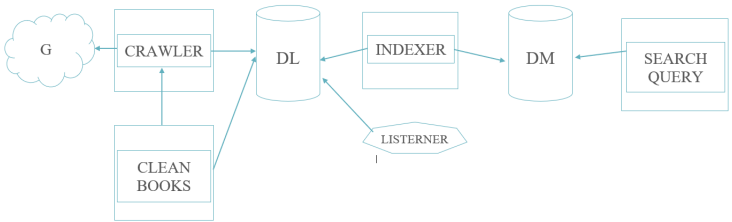
This project focuses on the development of a comprehensive system for managing digital books. It encompasses a series of critical phases ranging from the initial download and cleaning of books to their storage in a datalake. During this process, important metadata is extracted and content is normalized to ensure the consistency and quality of the data stored in a structured database. The culmination of this process involves the creation of a REST API, which provides an accessible and effective interface for accessing and manipulating book information.

2 Introduction

The initial phase of acquiring and cleaning books from external sources is crucial to ensure the quality and consistency of the stored data. Integrating a datalake as a central repository facilitates efficient management and organization of books and their associated metadata. This centralization streamlines the subsequent processes of data normalization and structuring, culminating in storage within a relational database.

To maintain an accurate and updated representation of book information, real-time monitoring and change detection are implemented through listeners in the datalake. This continuous vigilance ensures that the database remains up to date, optimizing indexing and search processes. The result is a system that excels in effectiveness and agility, offering a seamless and real-time user experience.

Additionally, the database is structured into three distinct tables, each with a specific purpose: storing words, books, and their associations. This organization allows for an efficient approach to managing and extracting relevant information, with a clear representation of word frequency in each book, enabling a deep understanding of patterns and themes present in the content. The structure described above is reflected in the following schema:



3 Module 1: Clean-Books

This module is responsible for cleaning and processing books obtained from external sources.

3.1 Class: Controller

The `Controller` class manages the execution flow of the cleaning process. It will contain as attributes a GutenbergSplitter object, as well as a MissingFiles-Management and a SimpleDateFormat("yyyyMMdd"). Its main goal is to clean all the books and separate them in metadata and content using a listener. It has the following methods:

3.1.1 Method: execute()

The `execute` method will be observing the folder where the books of tge day are located waiting for changes using a WatchService. Once it notices a change, we will split the document implied in the event in metadata and content, calling the function splitDocument of our GutenbergSplitter object. This process will be repeated for every event, an sleep is required to guarantee that the raw file is full written by the crawler. The The `startListening` method is pretty similar, but it looks the upper level to check if the folder of the day has been created.

3.1.2 Method: checkTodayFolderAndExecute()

The `checkTodayFolderAndExecute` will check if the folder for the current date already exists in our datalake. If exists we will call the function `activateExecution` (which will create a new Controller and call the execute method). If the folder does not exist we will use the method `startListening` to wait for the folder to be created.

3.1.3 Method: scheduleMissingFilesCheck()

First, we create a executor service which will allow us to schedule any task. In this case, we have notice that some files from the previous day could not be processed, so we need to check for them later. We will call `scheduleAtFixedRate` which will run the method `splitMissingFiles` from a missingFiles management object once a day.

In this module we have two other methods that are quite simple. The `datalakeChecker` function just confirms that the datalake folder exists in our project, creating it if it does not exist. The `isyyyyMMddFormat` method checks if a folder follows the format yyyyMMdd by checking its length.

3.2 Class: GutenbergCleaner (implements BookCleaner)

The `GutenbergCleaner` class is responsible for cleaning book content and metadata.

3.2.1 Method: cleanText(content: String)

The `cleanText` method takes a string of book content and cleans it by removing stop words, special characters, and other unnecessary elements. The stop words are contained in a .txt file which contains the stop words for languages such as English, Spanish or French gathered from the nltk python library. The function will clear everything which is not a unicode character, as well as remove single letters, words composed only by consonants, duplicated spaces and the before mentioned stopwords.

3.2.2 Method: readWords(path: String): List¡String¿

The `readWords` method obtains a list of stop words, once we read the file indicated through the path. We will just create a List of strings splitting each line of the document by whitespaces. This method will be called by `cleanText`.

3.3 Class: GutenbergSplitter (implements DocumentSplitter)

The `GutenbergSplitter` class is responsible for splitting books into content and metadata. It has as attributes a GutenbergFileReader(to read the raw books), a GutenbergDatalakeCreator(to create the structure of the datalake), a GutenbergCleaner(to normalize the text from the book), a UTF8FileChecker(to check that the file is encoded in UTF-8), and a dictionary of extensions (used to store the files in the correct format).

3.3.1 Method: splitDocument(path: String)

The `splitDocument` method takes the path to a book, identifies its content and metadata, and stores them separately. First we need to read the raw content of the book, as well as checking if the file has the UTF-8 encoding. We have observed all the e-books have similar pattern of where are located the metadata and content. Using this patterns as delimiters we can split the text. Finally, we will call the function storeFile for each text. For the content we will be using as argument the method `cleanText` from GutenbergCleaner and for the metadata we will use the the method `prepareMetadata`.

3.3.2 Method: prepareMetadata(text: String)

The `prepareMetadata` method processes metadata text, converting it into a structured format. We will apply a pattern to indicate how to separate the text by "key: value". We have to take into account the value field could have line breaks. For every match to our pattern we will add a key-value pair to a dictionary, which will be returned in json format using gson.

3.3.3 Method: storeFile(name: String, content: String, type: String)

The `storeFile` method saves content and metadata to specified files. We will use the method `setFilePath` from GutenbergDatalakeCreator, to obtain a part of the path where we should store our file. To complete it, we will need to add the id of the book(parameter name) with the corresponding extension from extensionDictionary based on the type. We will write the content passed as parameter to the file specified in the path.

3.4 Class: MissingFilesManagement

The `MissingFilesManagement` has as its main goal to check and split books from the previous day. We needed to add this class due to problems managing the first event in a date folder(which is modify instead of create) and for covering possible fails. It will use a GutenbergSplitterObject.

3.4.1 Method: getMissingFiles()

The `getMissingFiles` function will return the files that had not been splitted. We will check the elements in each book folder to determine it.

3.4.2 Method: removeDirectory()

The `removeDirectory` will be used to remove the directories of books which has been splitted later in a new directory.

3.4.3 Method: splitMissingFiles()

The splitMissingFiles will call splitDocument for each missing file returned by getMissingFiles. We will also use storeFiles to change the location of the raw file cleaned to the current day folder(where the content and metadata of the book are stored), because of that, we will need to remove the book directory from the previous day.

3.5 Class: UTF8FileChecker

The `isUTF8` is the only method in this class which receives a File as parameter. Will we benefit from the UTF-8 Byte Order Mark, using the bytes 0xEF, 0xBB, and 0xBF to detect if a file is encoded in UTF-8.

4 Module 2: Crawler

This module is responsible for crawling and downloading books from external sources. For this proyect, we implemented several classes to download files from gutenberg.org in UTF-8 format. For the download process, we decided to download books in sequential order starting from the bookID number 1 and

so on. We decided to do it in this way so, in case we download all books in **Gutenberg.org**, we can get the new books easily.

4.1 Class: BatchDownloader

This class is incharged of setting the batch of books to download from gutenberg.org for the controller to execute it, where with the following methods, we will set up an array with the bookd's ids to download.

4.1.1 Constructor: BatchDownloader(batchSize: int, gutenbergFileReader: GutenbergFileReader, bookIds: String[])

The constructor initializes the `BatchDownloader` with the specified batch size, a `GutenbergFileReader` instance, and an empty array of book IDs that we will fill with the mentioned .

4.1.2 Method: download()

The `download` method fetches a batch of books according to the specified batch size and book IDs. To achieve this, we utilize the `setBooksBatch` method to configure the list of URLs for download. Subsequently, we invoke the `downloadFile` function from the `GutenbergFileReader` to download each book. sleep is incorporated within this method. This deliberate pause is crucial to prevent chaos in data access within the datamart, where simultaneous access by multiple books could otherwise violate numerous unique constraints.

4.1.3 Method: setBooksBatch(mod: int)

The `setBooksBatch` method sets a batch of book IDs to be downloaded. For this, we search in the datalake for the latest book downloaded and get the id so we can fill the list with the url of the following books with the function URLSetter.

4.1.4 Method: getLastBookId(baseDirectory: String): int

The `getLastBookId` method retrieves the last book ID from the specified directory. For this implemetation, provide the whole datalake directory so it can search among all the date directories, all the different book directories and compare their ids to get the highest one.

4.1.5 Method: extractIdFromDirectoryName(directoryName: String): int

The `extractIdFromDirectoryName` is the function that allows us to extract the id of the book from the different routes that we obtain in the function getLastBookId.

4.1.6 Method: URLSetter(id: int): String

The `URLSetter` method generates the url of the book that we want to download using it's ID from gutenberg.org.

4.2 Class: Controller

The `Controller` class manages the execution flow of the crawling and downloading process.

4.2.1 Constructor: Controller(batchsize: int)

The constructor initializes the `Controller` with the specified batch size required. and creates the different instances needed for the controller to work such as a GutenbergFileReader instance.

4.2.2 Method: execute()

The `execute` method initiates the book downloading process by creating the day folder if is not created and then the process is stopped 1 second so the different modules have enough time to access to the directory, and then we start the download process.

4.2.3 Method: run()

The `run` method schedules the execution of the `execute` method at regular intervals so the module keeps downloading files continuously.

4.3 Class: GutenbergFileReader (implements BookReader)

The `GutenbergFileReader` class is responsible for reading and downloading book files.

4.3.1 Method: downloadFile(fileURL: String)

The `downloadFile` method downloads a book file from a given URL, where we get the book ID from the URL given and store it in the raw subdirectory of the specific book directory.

4.3.2 Method: read(bookID: String, type: String): String

The `read` method reads the content of a book file for a specified book ID and file type, being the file type options: raw, content and metadata, which are the subdirectories of each book directory,

4.3.3 Method: getFilePath(bookId: String, type: String): String

The `getFilePath` method retrieves the file path for a specified book ID and file type, being the type the same as mentioned in the method before. This method is used by the read() function and has the logic to retrieve the path specified.

4.4 Class: GuttenbergDatalakeCreator (implements DatalakeCreator)

The `GuttenbergDatalakeCreator` class is responsible for creating the data lake structure for storing downloaded books. The structure of the datalake is:

- The root datalake directory called **datalake**.
- Subdirectories of **datalake** with a **yyyyMMdd** format for each day downloading.
- Subdirectories of the **date directories** with the **booksId**, with each book having it's own directory as a result of it.
- Subdirectories of the **booksId** directories with the type of document stored with each book having a:
  - **raw**, with a .txt file inside with the whole book.
  - **content** directory with a .txt file with the content of the book without the metadata and the license explanation at the bottom of the book.
  - **metadata** directory with a .json file with the metadata of the book such as the title or the author.

4.4.1 Method: createFolder(folderPath: String)

The `createFolder` method creates the subdirectories of **raw**, **content** and **metadata** for each bookId directory.

4.4.2 Method: setFilePath(filename: String, currentDate: Date, type: String): String

The `setFilePath` method generates and returns the file path for storing downloaded books and makes the setups of the type subdirectories calling the **createFolder()** method.

4.5 Class: Main

The `Main` class contains the application's entry point to the module and is incharged of rule the execution of the module.

4.5.1 Method: main(String[] args)

The `main` method starts the execution of the book downloading process by calling to the controller's **run()** method.

4.6 Interface: BookReader

The `BookReader` interface defines methods for reading and downloading book files.

4.6.1 Method: read(bookID: String, type: String): String

The `read` method reads the content of a book file for a specified book ID and file type and sets the way it should be implemented.

4.6.2 Method: downloadFile(fileURL: String)

The `downloadFile` method downloads a book file from a given URL and sets the way it should be implemented.

4.7 Interface: DatalakeCreator

The `DatalakeCreator` interface defines methods for creating the data lake structure to store downloaded books into the datalake.

4.7.1 Method: createFolder(folderPath: String)

The `createFolder` method creates the necessary folder structure for storing downloaded books and sets the structure that the method should have in the implementations

4.7.2 Method: setFilePath(filename: String, currentDate: Date, type: String): String

The `setFilePath` method generates and returns the file path for storing downloaded books, in the interace, sets the structure that the implementation must follow.

5 Module 3: Datamart-Builder

5.1 Book Class

The `Book` class represents a book and contains the following attributes:

- `id`: An integer representing the book's unique identifier.
- `author`: A string representing the author's name.
- `title`: A string representing the title of the book.

5.1.1 Important Methods

- `Book(int id, String author, String title)`: Constructor to create a new `Book` object with the provided parameters.
- `getId()`: Returns the book's ID.
- `getAuthor()`: Returns the author's name.
- `getTitle()`: Returns the book's title.

5.2 Associate Class

The `Associate` class represents an association between a word and a book and contains the following attributes:

- `wordId`: An integer representing the ID of the associated word.
- `bookId`: An integer representing the ID of the associated book.
- `count`: An integer representing the count of occurrences of the word in the book.

5.2.1 Important Methods

- `Associate(int wordId, int bookId, int count)`: Constructor to create a new `Associate` object with the provided parameters.
- `getWordId()`: Returns the word's ID.
- `getBookId()`: Returns the book's ID.
- `getCount()`: Returns the count of occurrences.

5.3 Word Class

The `Word` class represents a word and contains the following attributes:

- `id`: An integer representing the unique identifier for the word.
- `label`: A string representing the word's label or text.

5.3.1 Important Methods

- `Word(int id, String label)`: Constructor to create a new `Word` object with the provided parameters.
- `getId()`: Returns the word's ID.
- `getLabel()`: Returns the word's label.

5.4 DatabaseWriter Class

The `DatabaseWriter` class is responsible for interacting with a database. It implements the `Datamart` interface, which defines methods for managing data in the database.

5.4.1 Important Methods

- `DatabaseWriter()`: Constructor that establishes a connection to a SQLite database and initializes it with necessary tables.
- `initDatabase()`: Initializes the database tables.
- `addBook(Book book)`: Adds a `Book` object to the database.
- `addWord(Word word)`: Adds a `Word` object to the database.
- `addAssociation(Associate associate)`: Adds an `Associate` object to the database.
- `deleteTable(String sql)`: Deletes a table in the database.
- `findWordByLabel(String label)`: Finds a `Word` object by its label in the database.
- `getMaxId()`: Retrieves the maximum ID of words in the database.

5.5 FileDatalake Class

The `FileDatalake` class implements the `Datalake` interface, which defines a method for reading data from files.

5.5.1 Important Methods

- `countWordOccurrences(List<String> words, String word)`: Counts the occurrences of a word in a list of words.
- `invertedIndexFromFilesWithCount(String booksDirectory, Book book, int wordMaxId)`: Generates an inverted index mapping words to their occurrences in a book's content.
- `read(File directory, int wordMaxId)`: Reads data from files in a specified directory and constructs a mapping of books, associates, and words.

5.6 Datalake Interface

The `Datalake` interface defines a method `read(File directory, int wordMaxId)` for reading data from a directory.

5.7 Controller Class

The `Controller` class coordinates the processing of data. It watches for changes in a directory, triggers data processing tasks, and manages the interaction between the `Datalake` and `Datamart` components.

5.7.1 Important Methods

- `start()`: Monitors a directory for changes and initiates data processing tasks.
- `task(Path filename)`: Processes data from a file, including adding books, words, and associations to the database.
- `taskDelete()`: Initializes the database by deleting existing tables.

5.8 Datamart Interface

The `Datamart` interface defines methods for initializing the database, adding books, words, and associations to the database, and retrieving word information.

5.9 Main Class

The `Main` class is the entry point of the application, which creates an instance of the `Controller` class and starts the data processing.

5.10 SqliteWriter Class

The `SqliteWriter` class provides SQL statements for inserting data into the database tables. It is used by the `DatabaseWriter` class to construct and execute SQL statements for inserting data.

6 Module 4: book-api

The *book-api* module serves as the application's interface between the end-user and the book indexing system's database. This module functions as a web server, providing an access point to the system's search functionality and enabling users to query and retrieve information via a RESTful API.

6.1 Architecture and Design

The module is constructed using the Spark micro-framework in Java, which facilitates the rapid deployment of HTTP endpoints and the straightforward configuration of business logic and request handling. The API is structured around domain-representative resources, specifically books (*books*), words (*words*), and associations (*associations*).

6.1.1 Database Connection Management

The management of the SQLite database connection is carried out through the *SqliteReader* class, which encapsulates the operations of opening and closing connections, executing SQL queries, and transforming results into domain objects.

6.1.2 Search Endpoint

The centerpiece of *book-api* is the search endpoint, which allows users to conduct queries within the indexed database. This endpoint accepts search terms through a query parameter and returns a list of relevant books. The search processing is supported by the *DataManager* class, which contains the logic to interact with the database and return search results.

6.2 API Operations

The RESTful API operations are designed to allow the manipulation and retrieval of database resources. The main endpoints and their functions are detailed below:

- **GET /api/books**: Retrieves a list of all the books available in the database.
- **GET /api/words**: Obtains a list of all indexed words.
- **GET /api/associations**: Provides the associations between books and words, including the frequency of the words.
- **GET /api/search**: Accepts a search term and returns the books containing that term.
- **POST /api/books**: Allows the insertion of a new book into the database.
- **PUT /api/books/{id}**: Updates the information of an existing book.
- **DELETE /api/books/{id}**: Removes a book from the database.

6.3 Indexing and Searching

Book indexing is a critical part of the search system, enabling efficient retrieval of information. The *Datamart-Builder* module, responsible for indexing, creates a structured representation of the data that is then consumed by *book-api*. The search implements a straightforward algorithm utilizing SQL pattern matching to find relevant books based on provided search terms.

6.4 Development Considerations

Throughout the implementation of *book-api*, particular emphasis was placed on code clarity, ease of maintenance, and search efficiency. Development practices such as separation of concerns and design patterns usage were implemented. Additionally, comprehensive test coverage was performed to ensure the module's reliability and robustness.

7 Conclusion

For the conclusions of the project, we can conclude that this implementation is a good starting point for a solid implementation of an Inverted Index. The noteworthy challenges encountered in the implementation process involved difficulties in processing data from external sources where documents were in various formats. Another significant issue was the module CleanBooks' access to the datalake raw files before the document completed downloading, requiring a resolution. To address this, we attempted to execute the **splitDocument()** process multiple times. If unsuccessful, we proceeded to the next book, and the **MissingFiles()** routine attempted indexing the following day, assuming the book was fully downloaded. This security measure aimed to ensure comprehensive book indexing. As a result of our efforts, the project operates at an acceptable speed, with potential future implementations to further enhance its performance. Another thing to take into consideration is the time in the sleep needed in the **batchDownloader** because it could cause errors in the datamart with unique constriaint violations depending on the computer is executed on.

8 Future Work

For future work we think that the things that may be good to implement and upgrade are:

1. **Event Management Upgrade With a Broker:** working with listeners h
2. **Search Efficiency:** Optimizing the search functionality for faster response times, particularly for complex queries and larger datasets. Implementing caching strategies and more efficient database indexing are potential approaches.
3. **Advanced Ranking Algorithms:** Developing more sophisticated algorithms for ranking search results, which could include machine learning techniques to personalize and improve relevance based on user behavior.
4. **Enhanced Concurrency and Database Access:** Implementing an improved strategy for introducing data to the database would allow us to eliminate the need of the sleep function within the **BatchDownloader()**

or a new way to see if a book isn't indexed from the day before, to index it now.